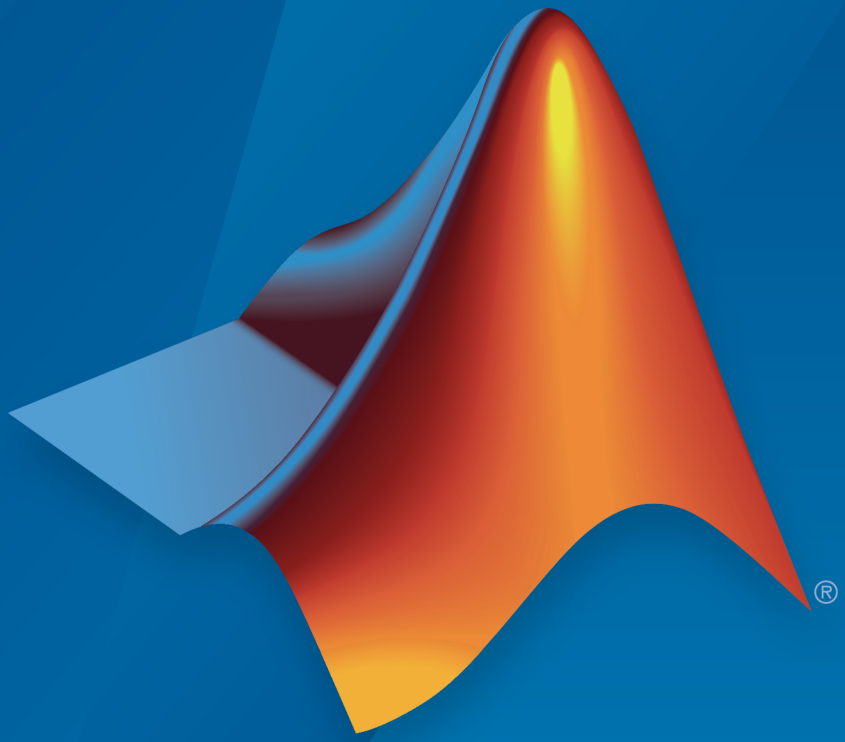
Econometrics Toolbox™

User's Guide



MATLAB®

R2015b

MathWorks®

How to Contact MathWorks

Latest news: `www.mathworks.com`

Sales and services: `www.mathworks.com/sales_and_services`

User community: `www.mathworks.com/matlabcentral`

Technical support: `www.mathworks.com/support/contact_us`

Phone: 508-647-7000

The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760-2098

Econometrics Toolbox™ User's Guide

© COPYRIGHT 1999–2015 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See `www.mathworks.com/trademarks` for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see `www.mathworks.com/patents` for more information.

Revision History

October 2008	Online only	Version 1.0 (Release 2008b)
March 2009	Online only	Revised for Version 1.1 (Release 2009a)
September 2009	Online only	Revised for Version 1.2 (Release 2009b)
March 2010	Online only	Revised for Version 1.3 (Release 2010a)
September 2010	Online only	Revised for Version 1.4 (Release 2010b)
April 2011	Online only	Revised for Version 2.0 (Release 2011a)
September 2011	Online only	Revised for Version 2.0.1 (Release 2011b)
March 2012	Online only	Revised for Version 2.1 (Release 2012a)
September 2012	Online only	Revised for Version 2.2 (Release 2012b)
March 2013	Online only	Revised for Version 2.3 (Release 2013a)
September 2013	Online only	Revised for Version 2.4 (Release 2013b)
March 2014	Online Only	Revised for Version 3.0 (Release 2014a)
October 2014	Online Only	Revised for Version 3.1 (Release 2014b)
March 2015	Online Only	Revised for Version 3.2 (Release 2015a)
September 2015	Online Only	Revised for Version 3.3 (Release 2015b)

Contents

1 Getting Started

2 Data Preprocessing

3 Model Selection

4 Time Series Regression Models

5 Conditional Mean Models

6 Conditional Variance Models

7 Multivariate Time Series Models

8 State-Space Models

1

Getting Started

Econometrics Toolbox Product Description

Model and analyze financial and economic systems using statistical methods

Econometrics Toolbox™ provides functions for modeling economic data. You can select and calibrate economic models for simulation and forecasting. For time series modeling and analysis, the toolbox includes univariate ARMAX/GARCH composite models with several GARCH variants, multivariate VARMAX models, and cointegration analysis. It also provides methods for modeling economic systems using state-space models and for estimating using the Kalman filter. You can use a variety of diagnostic functions for model selection, including hypothesis, unit root, and stationarity tests.

Key Features

- Univariate ARMAX/GARCH composite models, including EGARCH, GJR, and other variants
- Multivariate simulation and forecasting of VAR, VEC, and cointegrated models
- State-space models and the Kalman filter for estimation
- Tests for unit root (Dickey-Fuller, Phillips-Perron) and stationarity (Leybourne-McCabe, KPSS)
- Statistical tests, including likelihood ratio, LM, Wald, Engle's ARCH, and Ljung-Box Q
- Cointegration tests, including Engle-Granger and Johansen
- Diagnostics and utilities, including AIC/BIC model selection and partial-, auto-, and cross-correlations
- Hodrick-Prescott filter for business-cycle analysis

Econometric Modeling

In this section...

Model Selection

A probabilistic time series model is necessary for a wide variety of analysis goals, including regression inference, forecasting, and Monte Carlo simulation. When selecting a model, aim to find the most parsimonious model that adequately describes your data. A simple model is easier to estimate, forecast, and interpret.

- *Specification tests* help you identify one or more model families that could plausibly describe the data generating process.
- *Model comparisons* help you compare the fit of competing models, with penalties for complexity.
- *Goodness-of-fit* checks help you assess the in-sample adequacy of your model, verify that all model assumptions hold, and evaluate out-of-sample forecast performance.

Model selection is an iterative process. When goodness-of-fit checks suggest model assumptions are not satisfied—or the predictive performance of the model is not satisfactory—consider making model adjustments. Additional specification tests, model comparisons, and goodness-of-fit checks help guide this process.

Econometrics Toolbox Features

Modeling Questions	Features	Related Functions
What is the dimension of my response variable?	• The conditional mean and variance models in this toolbox are for modeling univariate, discrete data. • Separate models are available for multivariate, discrete data, such as VAR and VEC models. • State-space models support univariate or multivariate response variables.	• arima • `egarch` • `egcitest` • `garch` • `gjr` • `jcontest` • ssm

Modeling Questions	Features	Related Functions
		• `vgxpred` • `vgxsim` • `vgxvarx`
Is my time series stationary?	• Stationarity tests are available. If your data is not stationary, consider transforming your data. Stationarity is the foundation of many time series models. • Or, consider using a nonstationary ARIMA model if there is evidence of a unit root in your data.	• arima • `i1Otest` • `kpsstest` • `lmctest`
Does my time series have a unit root?	• Unit root tests are available. Evidence in favor of a unit root suggests your data is difference stationary. • You can difference a series with a unit root until it is stationary, or model it using a nonstationary ARIMA model.	• `adftest` • arima • `i1Otest` • `pptest` • `vratiotest`
How can I handle seasonal effects?	• You can deseasonalize (seasonally adjust) your data. Use seasonal filters or regression models to estimate the seasonal component. • Seasonal ARIMA models use seasonal differencing to remove seasonal effects. You can also include seasonal lags to model seasonal autocorrelation (both additively and multiplicatively).	• arima • regARIMA
Is my data autocorrelated?	• Sample autocorrelation and partial autocorrelation functions help identify autocorrelation. • Conduct a Ljung-Box Q-test to test autocorrelations at several lags jointly. • If autocorrelation is present, consider using a conditional mean model. • For regression models with autocorrelated errors, consider using FGLS or HAC estimators. If the error model structure is an ARIMA model, consider using a regression model with ARIMA errors.	• arima • `autocorr` • `fgls` • `hac` • `lbqtest` • `parcorr` • regARIMA

Modeling Questions	Features	Related Functions
What if my data is heteroscedastic (exhibits volatility clustering)?	• Looking for autocorrelation in the squared residual series is one way to detect conditional heteroscedasticity. • Engle's ARCH test evaluates evidence against the null of independent innovations in favor of an ARCH model alternative. • To model conditional heteroscedasticity, consider using a conditional variance model. • For regression models that exhibit heteroscedastic errors, consider using FGLS or HAC estimators.	• `archtest` • `egarch` • `fgls` • `garch` • `gjr` • `hac`
Is there an alternative to a Gaussian innovation distribution for leptokurtic data?	• You can use a Student's *t* distribution to model fatter tails than a Gaussian distribution (excess kurtosis). • You can specify a *t* innovation distribution for all conditional mean and variance models, and ARIMA error models in Econometrics Toolbox. • You can estimate the degrees of freedom of the *t* distribution along with other model parameters.	• arima • `egarch` • `garch` • `gjr` • regARIMA
How do I decide between these models?	• You can compare nested models using misspecification tests, such as the likelihood ratio test, Wald's test, or Lagrange multiplier test. • Information criteria, such as AIC or BIC, compare model fit with a penalty for complexity.	• `aicbic` • `lmtest` • `lratiotest` • `waldtest`
Do I have two or more time series that are cointegrated?	• The Johansen and Engle-Granger cointegration tests assess evidence of cointegration. • Consider using the VEC model for modeling multivariate, cointegrated series. • Also consider cointegration when regressing time series. If present, it can introduce spurious regression effects.	• `egcitest` • `jcitest` • `jcontest`

Modeling Questions	Features	Related Functions
What if I want to include predictor variables?	• ARIMAX and VARX models are available in this toolbox. • State-space models support predictor data.	• arima • ssm • `vgxvarx`
What if I want to implement regression, but the classical linear model assumptions do not apply?	• Regression models with ARIMA errors are available in this toolbox. • Regress robustly using FGLS or HAC estimators. • For a series of examples on time series regression techniques that illustrate common principles and tasks in time series regression modeling, see Econometrics Toolbox Examples. • For more regression options, see Statistics and Machine Learning Toolbox™ documentation.	• `fgls` • `hac` • `mvregress` • regARIMA • `regress` (Statistics and Machine Learning Toolbox)
How do use the Kalman filter to analyze several unobservable, linear, stochastic time series and several, observable, linear, stochastic functions of them?	Standard, linear state-space modeling is available in this toolbox.	ssm

Related Examples

- "Box-Jenkins Model Selection" on page 3-4
- "Detect Autocorrelation" on page 3-18
- "Detect ARCH Effects" on page 3-28
- "Unit Root Tests" on page 3-44
- "Time Series Regression I: Linear Models"
- "Time Series Regression II: Collinearity and Estimator Variance"
- "Time Series Regression III: Influential Observations"

- "Time Series Regression IV: Spurious Regression"
- "Time Series Regression V: Predictor Selection"
- "Time Series Regression VI: Residual Diagnostics"
- "Time Series Regression VII: Forecasting"
- "Time Series Regression VIII: Lagged Variables and Estimator Bias"
- "Time Series Regression IX: Lag Order Selection"
- "Time Series Regression X: Generalized Least Squares and HAC Estimators"

More About

- "Trend-Stationary vs. Difference-Stationary Processes" on page 2-7
- "Box-Jenkins Methodology" on page 3-2
- "Goodness of Fit" on page 3-88
- "Regression Models with Time Series Errors" on page 4-6
- "Nonspherical Models" on page 3-94
- "Conditional Mean Models" on page 5-3
- "Conditional Variance Models" on page 6-2
- "Vector Autoregressive (VAR) Models" on page 7-3
- "Cointegration and Error Correction Analysis" on page 7-108

Model Objects, Properties, and Methods

In this section...

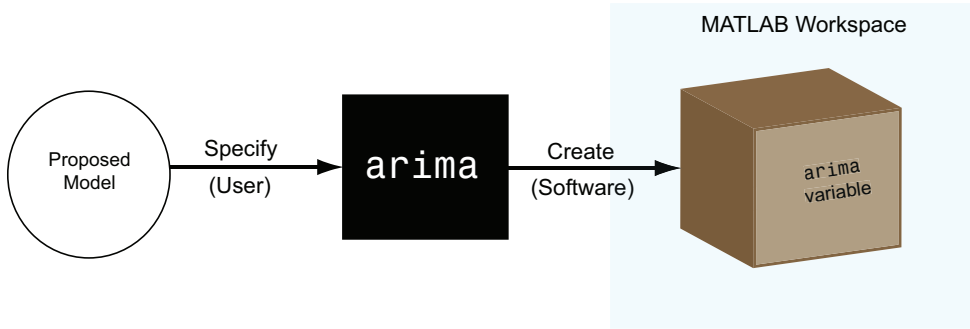
Model Objects

After you have a potential model for your data, you must specify the model to MATLAB® to proceed with your analysis. Econometrics Toolbox has model objects for storing specified econometric models. For univariate, discrete time series analysis, there are five available model objects:

- `arima` — for integrated, autoregressive, moving average (ARIMA) models optionally containing exogenous predictor variables.
- `garch` — for generalized autoregressive conditional heteroscedaticity models (GARCH)
- `egarch` — for exponential GARCH models
- `gjr` — for Glosten-Jagannathan-Runkle models
- `regARIMA` — for regression models with ARIMA errors

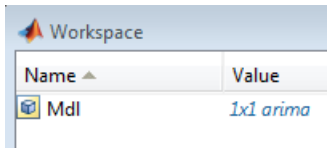
For multivariate, discrete time series analysis you can create a state-space model using an ssm model obvect.

To create a model object, specify the form of your model to one of the model functions (e.g., `arima` or `garch`). The function creates the model object of the corresponding type in the MATLAB workspace, as shown in the figure.



You can work with model objects as you would with any other variable in MATLAB. For example, you can assign the object variable a name, view it in the MATLAB Workspace, and display its value in the Command Window by typing its name.

This image shows a workspace containing an `arima` model named `Mdl`.



Model Properties

A model object holds all the information necessary to estimate, simulate, and forecast econometric models. This information includes the:

- Parametric form of the model
- Number of model parameters (e.g., the degree of the model)
- Innovation distribution (Gaussian or Student's *t*)
- Amount of presample data needed to initialize the model

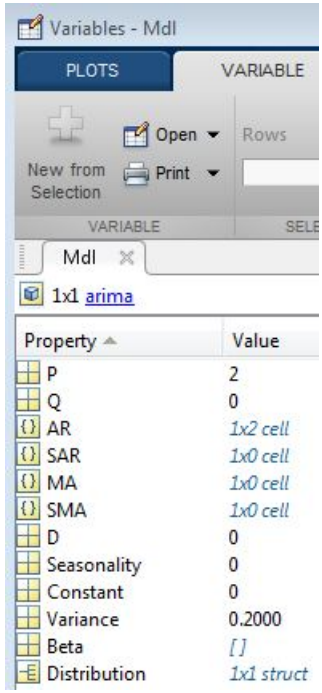
Such pieces of information are *properties* of the model, which are stored as fields *within* the model object. In this way, a model object resembles a MATLAB data structure (`struct` array).

The five model types—`arima`, `garch`, `egarch`, `gjr`, and `regARIMA`—have properties according to the econometric models they support. Each property has a predefined name, which you cannot change.

For example, `arima` supports conditional mean models (multiplicative and additive AR, MA, ARMA, ARIMA, and ARIMAX processes). Every `arima` model object has these properties, shown with their corresponding names.

Property Name	Property Description
`Constant`	Model constant
`AR`	Nonseasonal AR coefficients
`MA`	Nonseasonal MA coefficients
`SAR`	Seasonal AR coefficients (in a multiplicative model)
`SMA`	Seasonal MA coefficients (in a multiplicative model)
`D`	Degree of nonseasonal differencing
`Seasonality`	Degree of seasonal differencing
`Variance`	Variance of the innovation distribution
`Distribution`	Parametric family of the innovation distribution
`P`	Amount of presample data needed to initialize the AR component of the model
`Q`	Amount of presample data needed to initialize the MA component of the model

When a model object exists in the workspace, double-click its name in the Workspace window to open the Variable Editor. The Variable Editor shows all model properties and their names.



Notice that in addition to a name, each property has a value.

Specify Models

Specify a model by assigning values to model properties. You do not need, nor are you able, to specify a value for every property. The constructor function assigns default values to any properties you do not, or cannot, specify.

Tip It is good practice to be aware of the default property values for any model you create.

In addition to having a predefined name, each model property has a predefined data type. When assigning or modifying a property's value, the assignment must be consistent with the property data type.

For example, the `arima` properties have these data types.

Property Name	Property Data Type
`Constant`	Scalar
`AR`	Cell array
`MA`	Cell array
`SAR`	Cell array
`SMA`	Cell array
`D`	Nonnegative integer
`Seasonality`	Nonnegative integer
`Variance`	Positive scalar
`Distribution`	`struct` array
`P`	Nonnegative integer (you cannot specify)
`Q`	Nonnegative integer (you cannot specify)

Specify an AR(2) Model

To illustrate assigning property values, consider specifying the AR(2) model

$$y_t = 0.8y_{t-1} - 0.2y_{t-2} + \varepsilon_t,$$

where the innovations are independent and identically distributed normal random variables with mean 0 and variance 0.2. This is a conditional mean model, so use `arima`. Assign values to model properties using name-value pair arguments.

This model has two AR coefficients, 0.8 and -0.2. Assign these values to the property `AR` as a cell array, `{0.8,-0.2}`. Assign the value `0.2` to `Variance`, and `0` to `Constant`. You do not need to assign a value to `Distribution` because the default innovation distribution is `'Gaussian'`. There are no MA terms, seasonal terms, or degrees of integration, so do not assign values to these properties. You cannot specify values for the properties `P` and `Q`.

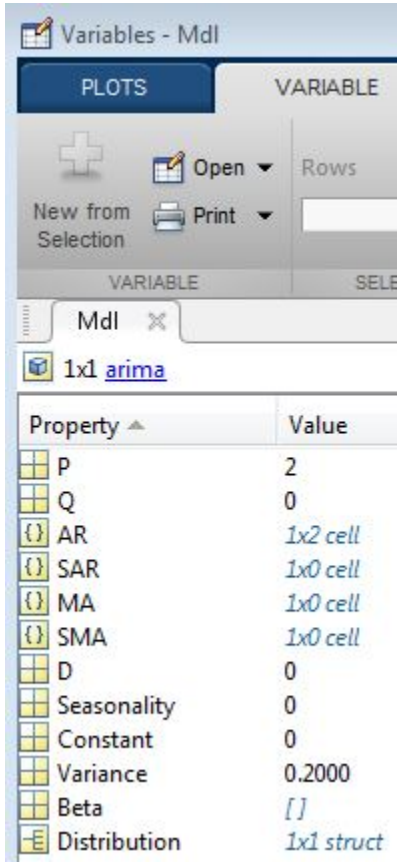
In summary, specify the model as follows:

```
Mdl = arima('AR',{0.8,-0.2},'Variance',0.2,'Constant',0)
```

```
Mdl =
```

```
ARIMA(2,0,0) Model:
--------------------
Distribution: Name = 'Gaussian'
           P: 2
           D: 0
           Q: 0
    Constant: 0
          AR: {0.8 -0.2} at Lags [1 2]
         SAR: {}
          MA: {}
         SMA: {}
    Variance: 0.2
```

The output displays the value of the created model, `Mdl`. Notice that the property `Seasonality` is not in the output. `Seasonality` only displays for models with seasonal integration. The property is still present, however, as seen in the Variable Editor.



`Mdl` has values for every `arima` property, even though the specification included only three. `arima` assigns default values for the unspecified properties. The values of `SAR`, `MA`, and `SMA` are empty cell arrays because the model has no seasonal or MA terms. The values of `D` and `Seasonality` are `0` because there is no nonseasonal or seasonal differencing. `arima` sets:

- `P` equal to `2`, the number of presample observations needed to initialize an AR(2) model.

- `Q` equal to `0` because there is no MA component to the model (i.e., no presample innovations are needed).

Specify a GARCH(1,1) Model

As another illustration, consider specifying the GARCH(1,1) model

$$y_t = \varepsilon_t,$$

where

$$\begin{aligned} \varepsilon_t &= \sigma_t z_t \\ \sigma_t^2 &= \kappa + \gamma_1 \sigma_{t-1}^2 + \alpha_1 \varepsilon_{t-1}^2 \end{aligned}$$

Assume z_t follows a standard normal distribution.

This model has one GARCH coefficient (corresponding to the lagged variance term) and one ARCH coefficient (corresponding to the lagged squared innovation term), both with unknown values. To specify this model, enter:

```
Mdl = garch('GARCH',NaN,'ARCH',NaN)
```

```
Mdl =

    GARCH(1,1) Conditional Variance Model:
    --------------------------------------
    Distribution: Name = 'Gaussian'
               P: 1
               Q: 1
        Constant: NaN
           GARCH: {NaN} at Lags [1]
            ARCH: {NaN} at Lags [1]
```

The default value for the constant term is also `NaN`. Parameters with `NaN` values need to be estimated or otherwise specified before you can forecast or simulate the model. There is also a shorthand syntax to create a default GARCH(1,1) model:

```
Mdl = garch(1,1)
```

```
Mdl =

    GARCH(1,1) Conditional Variance Model:
```

```
    ------------------------------------
    Distribution: Name = 'Gaussian'
               P: 1
               Q: 1
        Constant: NaN
           GARCH: {NaN} at Lags [1]
            ARCH: {NaN} at Lags [1]
```

The shorthand syntax returns a GARCH model with one GARCH coefficient and one ARCH coefficient, with default `NaN` values.

Retrieve Model Properties

The property values in an existing model are retrievable. Working with models resembles working with `struct` arrays because you can access model properties using dot notation. That is, type the model name, then the property name, separated by `'.'` (a period).

For example, consider the `arima` model with this AR(2) specification:

```
Mdl = arima('AR',{0.8,-0.2},'Variance',0.2,'Constant',0);
```

To display the value of the property `AR` for the created model, enter:

```
arCoefficients = Mdl.AR
```

```
arCoefficients =

    [0.8000]    [-0.2000]
```

`AR` is a cell array, so you must use cell-array syntax. The coefficient cell arrays are lag-indexed, so entering

```
secondARCoefficient = Mdl.AR{2}
```

```
secondARCoefficient =

   -0.2000
```

returns the coefficient at lag 2. You can also assign any property value to a new variable:

```
ar = Mdl.AR
```

```
ar =

    [0.8000]    [-0.2000]
```

Modify Model Properties

You can also modify model properties using dot notation. For example, consider this AR(2) specification:

```
Mdl = arima('AR',{0.8,-0.2},'Variance',0.2,'Constant',0)
```

```
Mdl =

    ARIMA(2,0,0) Model:
    --------------------
    Distribution: Name = 'Gaussian'
               P: 2
               D: 0
               Q: 0
        Constant: 0
              AR: {0.8 -0.2} at Lags [1 2]
             SAR: {}
              MA: {}
             SMA: {}
        Variance: 0.2
```

The created model has the default Gaussian innovation distribution. Change the innovation distribution to a Student's t distribution with eight degrees of freedom. The data type for `Distribution` is a `struct` array.

```
Mdl.Distribution = struct('Name','t','DoF',8)
```

```
Mdl =

    ARIMA(2,0,0) Model:
    --------------------
    Distribution: Name = 't', DoF = 8
               P: 2
               D: 0
               Q: 0
```

```
Constant: 0
      AR: {0.8 -0.2} at Lags [1 2]
     SAR: {}
      MA: {}
     SMA: {}
Variance: 0.2
```

The variable `Mdl` is updated accordingly.

Methods

Methods are functions that accept models as inputs. In Econometrics Toolbox, these functions accept `arima`, `garch`, `egarch`, `gjr`, and `regARIMA` models:

- `estimate`
- `infer`
- `forecast`
- `simulate`

Methods can distinguish between model objects (e.g., an `arima` model vs. a `garch` model). That is, some methods accept different optional inputs and return different outputs depending on the type of model that is input.

Find method reference pages for a specific model by entering, for example, `doc arima.estimate`.

See Also

regARIMA | ssm | arima | `egarch` | `garch` | `gjr` | `struct`

Related Examples

- "Specify Conditional Mean Models Using arima" on page 5-6
- "Specify GARCH Models Using garch" on page 6-8
- "Specify EGARCH Models Using egarch" on page 6-19
- "Specify GJR Models Using gjr" on page 6-31

More About

- Using garch Objects

- Using egarch Objects
- Using gjr Objects
- “Econometric Modeling” on page 1-3
- “Conditional Mean Models” on page 5-3
- “Conditional Variance Models” on page 6-2

Stochastic Process Characteristics

In this section...

What Is a Stochastic Process?

A time series y_t is a collection of observations on a variable indexed sequentially over several time points $t = 1, 2,...,T$. Time series observations $y_1, y_2,...,y_T$ are inherently dependent. From a statistical modeling perspective, this means it is inappropriate to treat a time series as a random sample of independent observations.

The goal of statistical modeling is finding a compact representation of the data-generating process for your data. The statistical building block of econometric time series modeling is the stochastic process. Heuristically, a stochastic process is a joint probability distribution for a collection of random variables. By modeling the observed time series y_t as a realization from a stochastic process $y = \{y_t, t = 1,...T\}$, it is possible to accommodate the high-dimensional and dependent nature of the data. The set of observation times T can be discrete or continuous. Figure 1-1 displays the monthly average CO_2 concentration (ppm) recorded by the Mauna Loa Observatory in Hawaii from 1980 to 2012 [2].

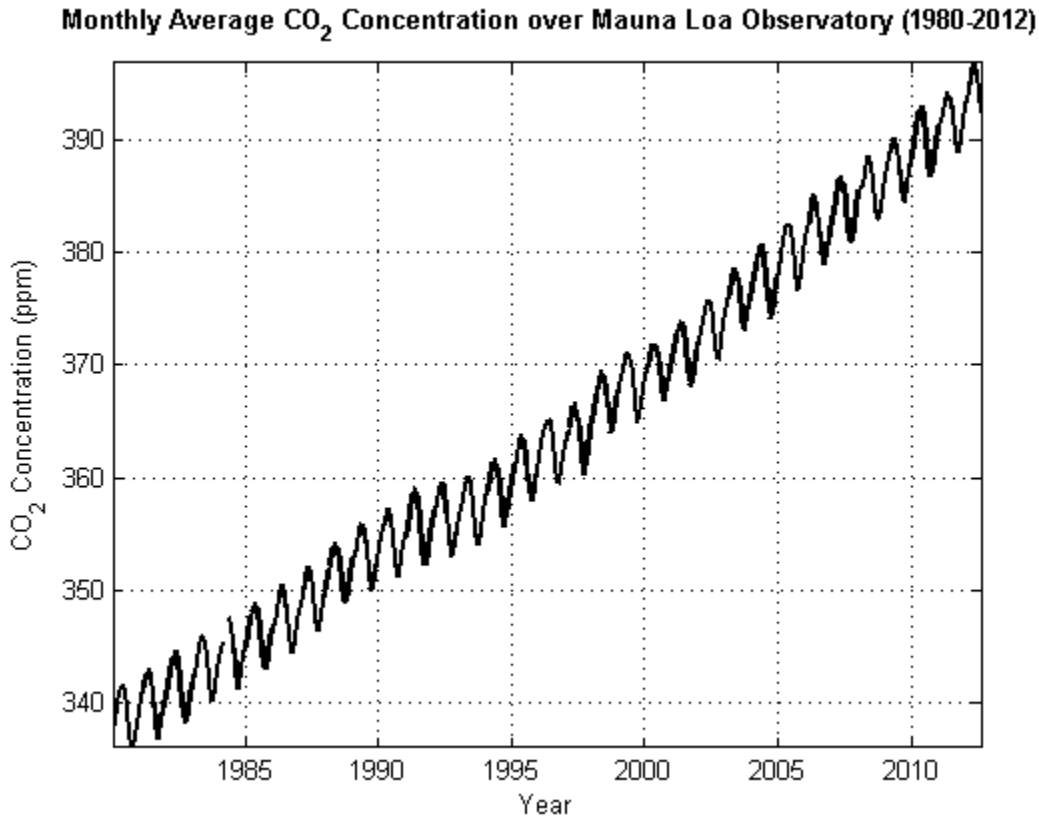


Figure 1-1. Monthly Average CO2

Stationary Processes

Stochastic processes are *weakly stationary* or *covariance stationary* (or simply, *stationary*) if their first two moments are finite and constant over time. Specifically, if y_t is a stationary stochastic process, then for all t:

- $E(y_t) = \mu < \infty$.
- $V(y_t) = \sigma^2 < \infty$.
- $Cov(y_t, y_{t-h}) = \gamma_h$ for all lags $h \neq 0$.

Does a plot of your stochastic process seem to increase or decrease without bound? The answer to this question indicates whether the stochastic process is stationary. “Yes” indicates that the stochastic process might be nonstationary. In Monthly Average CO2, the concentration of CO_2 is increasing without bound which indicates a nonstationary stochastic process.

Linear Time Series Model

Wold’s theorem [1] states that you can write all weakly stationary stochastic processes in the general linear form

$$y_t = \mu + \sum_{i=1}^{\infty} \psi_i \varepsilon_{t-i} + \varepsilon_t.$$

Here, ε_t denotes a sequence of uncorrelated (but not necessarily independent) random variables from a well-defined probability distribution with mean zero. It is often called the *innovation process* because it captures all new information in the system at time *t*.

Lag Operator Notation

The *lag operator L* operates on a time series y_t such that $L^i y_t = y_{t-i}$.

An *m*th-degree lag polynomial of coefficients b_1, b_2,...,b_m is defined as

$$B(L) = (1 + b_1 L + b_2 L^2 + \ldots + b_m L^m).$$

In lag operator notation, you can write the general linear model using an infinite-degree polynomial $\psi(L) = (1 + \psi_1 L + \psi_2 L^2 + \ldots)$,

$$y_t = \mu + \psi(L)\varepsilon_t.$$

You cannot estimate a model that has an infinite-degree polynomial of coefficients with a finite amount of data. However, if $\psi(L)$ is a rational polynomial (or approximately

rational), you can write it (at least approximately) as the quotient of two finite-degree polynomials.

Define the q-degree polynomial $\theta(L) = (1 + \theta_1 L + \theta_2 L^2 + \ldots + \theta_q L^q)$ and the p-degree polynomial $\phi(L) = (1 + \phi_1 L + \phi_2 L^2 + \ldots + \phi_p L^p)$. If $\psi(L)$ is rational, then

$$\psi(L) = \frac{\theta(L)}{\phi(L)}.$$

Thus, by Wold's theorem, you can model (or closely approximate) every stationary stochastic process as

$$y_t = \mu + \frac{\theta(L)}{\phi(L)}\varepsilon_t,$$

which has $p + q$ coefficients (a finite number).

Characteristic Equation

A degree p *characteristic polynomial* of the linear times series model $y_t = \phi_1 y_{t-1} + \phi_2 y_{t-2} + \ldots + \phi_p y_{t-p} + \varepsilon_t$ is

$$\phi(a) = a^p - \phi_1 a^{p-1} - \phi_2 a^{p-2} - \ldots - \phi_p.$$

It is another way to assess that a series is a stationary process. For example, the characteristic equation of $y_t = 0.5 y_{t-1} - 0.02 y_{t-2} + \varepsilon_t$ is $\phi(a) = a^2 - 0.5a + 0.02.$

The roots of the *homogeneous characteristic equation* $\phi(a) = 0$ (called the *characteristic roots*) determine whether the linear time series is stationary. If every root in $\phi(a)$ lies inside the unit circle, then the process is stationary. Roots lie within the unit circle if

they have an absolute value less than one. This is a unit root process if one or more roots lie inside the unit circle (i.e., have absolute value of one). Continuing the example, the characteristic roots of $\phi(a) = 0$ are $a = \{0.4562, 0.0438\}$. Since the absolute values of these roots are less than one, the linear time series model is stationary.

Unit Root Process

A linear time series model is a *unit root process* if the solution set to its characteristic equation contains a root that is on the unit circle (i.e., has an absolute value of one). Subsequently, the expected value, variance, or covariance of the elements of the stochastic process grows with time, and therefore is nonstationary. If your series has a unit root, then differencing it might make it stationary.

For example, consider the linear time series model $y_t = y_{t-1} + \varepsilon_t$, where ε_t is a white noise sequence of innovations with variance σ^2 (this is called the random walk). The characteristic equation of this model is $z - 1 = 0$, which has a root of one. If the initial observation y_0 is fixed, then you can write the model as $y_t = y_0 + \sum_{i=1}^{t} \varepsilon_i$. Its expected value is y_0, which is independent of time. However, the variance of the series is to^2, which grows with time making the series unstable. Take the first difference to transform the series and the model becomes $d_t = y_t - y_{t-1} = \varepsilon_t$. The characteristic equation for this series is $z = 0$, so it does not have a unit root. Note that

- $E(d_t) = 0$, which is independent of time,
- $V(d_t) = \sigma^2$, which is independent of time, and
- $Cov(d_t, d_{t-s}) = 0$, which is independent of time for all integers $0 < s < t$.

Monthly Average CO2 appears nonstationary. What happens if you plot the first difference $d_t = y_t - y_{t-1}$ of this series? Figure 1-2 displays the d_t. Ignoring the fluctuations, the stochastic process does not seem to increase or decrease in general. You can conclude that d_t is stationary, and that y_t is unit root nonstationary. For details, see "Differencing" on page 2-3.

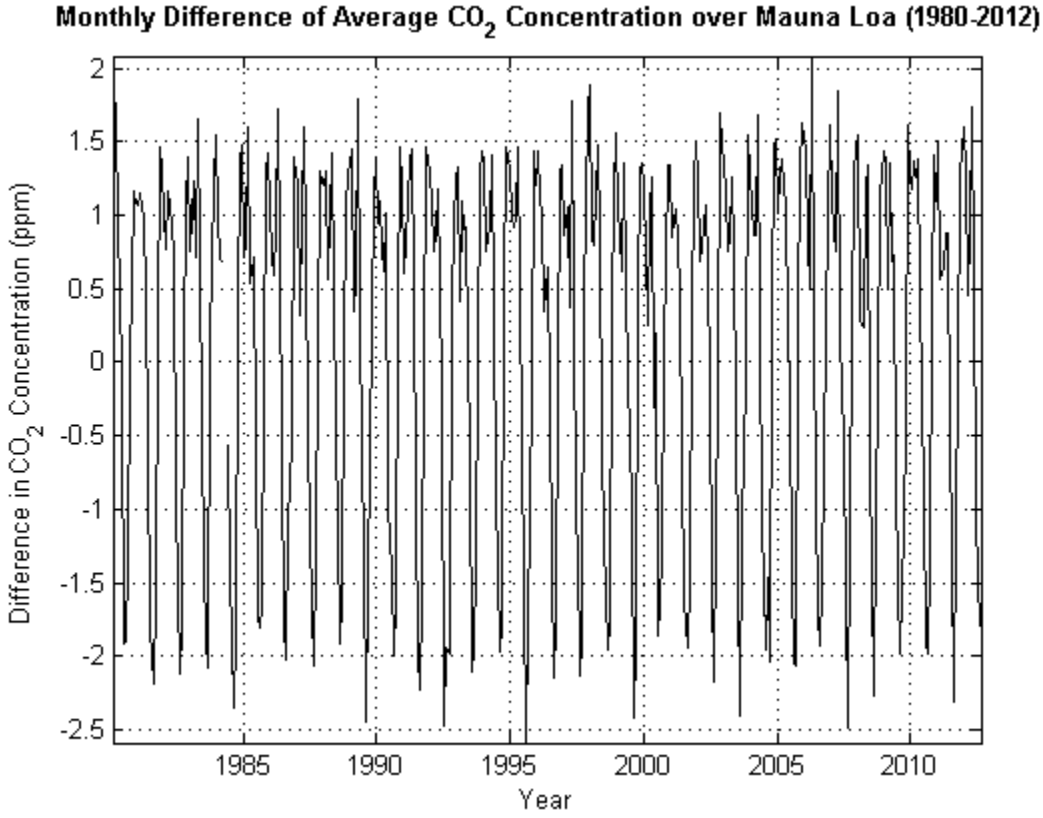


Figure 1-2. Monthly Difference in CO2

References

[1] Wold, H. *A Study in the Analysis of Stationary Time Series*. Uppsala, Sweden: Almqvist & Wiksell, 1938.

[2] Tans, P., and R. Keeling. (2012, August). "Trends in Atmospheric Carbon Dioxide." *NOAA Research*. Retrieved October 5, 2012 from http://www.esrl.noaa.gov/gmd/ccgg/trends/mlo.html.

Related Examples

- “Specify Conditional Mean Models Using arima” on page 5-6
- “Specify GARCH Models Using garch” on page 6-8
- “Specify EGARCH Models Using egarch” on page 6-19
- “Specify GJR Models Using gjr” on page 6-31
- “Simulate Stationary Processes” on page 5-151
- “Assess Stationarity of a Time Series” on page 3-58

More About

- “Econometric Modeling” on page 1-3
- “Conditional Mean Models” on page 5-3
- “Conditional Variance Models” on page 6-2

Bibliography

[1] Ait-Sahalia, Y. "Testing Continuous-Time Models of the Spot Interest Rate." *The Review of Financial Studies*. Spring 1996, Vol. 9, No. 2, pp. 385–426.

[2] Ait-Sahalia, Y. "Transition Densities for Interest Rate and Other Nonlinear Diffusions." *The Journal of Finance*. Vol. 54, No. 4, August 1999.

[3] Amano, R. A., and S. van Norden. "Unit Root Tests and the Burden of Proof." Bank of Canada. Working paper 92–7, 1992.

[4] Andrews, D. W. K. "Heteroskedasticity and Autocorrelation Consistent Covariance Matrix Estimation." *Econometrica*. v. 59, 1991, pp. 817-858.

[5] Andrews, D. W. K., and J. C. Monohan. "An Improved Heteroskedasticity and Autocorrelation Consistent Covariance Matrix Estimator." *Econometrica*. v. 60, 1992, pp. 953-966.

[6] Baillie, R. T., and T. Bollerslev. "Prediction in Dynamic Models with Time-Dependent Conditional Variances." *Journal of Econometrics*. Vol. 52, 1992, pp. 91–113.

[7] Belsley, D. A., E. Kuh, and R. E. Welsh. *Regression Diagnostics*. New York, NY: John Wiley & Sons, Inc., 1980.

[8] Bera, A. K., and H. L. Higgins. "A Survey of ARCH Models: Properties, Estimation and Testing." *Journal of Economic Surveys*. Vol. 7, No. 4, 1993.

[9] Bollerslev, T. "A Conditionally Heteroskedastic Time Series Model for Speculative Prices and Rates of Return." *Review of Economics and Statistics*. Vol. 69, 1987, pp. 542–547.

[10] Bollerslev, T. "Generalized Autoregressive Conditional Heteroskedasticity." *Journal of Econometrics*. Vol. 31, 1986, pp. 307–327.

[11] Bollerslev, T., R. Y. Chou, and K. F. Kroner. "ARCH Modeling in Finance: A Review of the Theory and Empirical Evidence." *Journal of Econometrics*. Vol. 52, 1992, pp. 5–59.

[12] Bollerslev, T., R. F. Engle, and D. B. Nelson. "ARCH Models." *Handbook of Econometrics*. Vol. 4, Chapter 49, Amsterdam: Elsevier Science B.V., 1994, pp. 2959–3038.

[13] Bollerslev, T., and E. Ghysels. “Periodic Autoregressive Conditional Heteroscedasticity.” *Journal of Business and Economic Statistics*. Vol. 14, 1996, pp. 139–151.

[14] Box, G. E. P. and D. Pierce. "Distribution of Residual Autocorrelations in Autoregressive-Integrated Moving Average Time Series Models." *Journal of the American Statistical Association*. Vol. 65, 1970, pp. 1509–1526.

[15] Box, G. E. P., G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis: Forecasting and Control*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

[16] Breusch, T.S., and Pagan, A.R. "Simple test for heteroscedasticity and random coefficient variation". *Econometrica*. v. 47, 1979, pp. 1287–1294.

[17] Brockwell, P. J. and R. A. Davis. *Introduction to Time Series and Forecasting*. 2nd ed. New York, NY: Springer, 2002.

[18] Brooks, C., S. P. Burke, and G. Persand. “Benchmarks and the Accuracy of GARCH Model Estimation.” *International Journal of Forecasting*. Vol. 17, 2001, pp. 45–56.

[19] Brown, M. B. and Forsythe, A. B. "Robust Tests for Equality of Variances." *Journal of the American Statistical Association*. 69, 1974, pp. 364–367.

[20] Burke, S. P. "Confirmatory Data Analysis: The Joint Application of Stationarity and Unit Root Tests." University of Reading, UK. Discussion paper 20, 1994.

[21] Campbell, J. Y., A. W. Lo, and A. C. MacKinlay. Chapter 12. “The Econometrics of Financial Markets.” *Nonlinearities in Financial Data*. Princeton, NJ: Princeton University Press, 1997.

[22] Caner, M., and L. Kilian. “Size Distortions of Tests of the Null Hypothesis of Stationarity: Evidence and Implications for the PPP Debate.” *Journal of International Money and Finance*. Vol. 20, 2001, pp. 639–657.

[23] Cecchetti, S. G., and P. S. Lam. “Variance-Ratio Tests: Small-Sample Properties with an Application to International Output Data.” *Journal of Business and Economic Statistics*. Vol. 12, 1994, pp. 177–186.

[24] Chow, G. C. “Tests of Equality Between Sets of Coefficients in Two Linear Regressions.” *Econometrica*. Vol. 28, 1960, pp. 591–605.

[25] Cochrane, J. “How Big is the Random Walk in GNP?” *Journal of Political Economy*. Vol. 96, 1988, pp. 893–920.

[26] Cribari-Neto, F. "Asymptotic Inference Under Heteroskedasticity of Unknown Form." *Computational Statistics & Data Analysis*. v. 45, 2004, pp. 215-233.

[27] Dagum, E. B. *The X-11-ARIMA Seasonal Adjustment Method*. Number 12–564E. Statistics Canada, Ottawa, 1980.

[28] Davidson, R., and J. G. MacKinnon. *Econometric Theory and Methods*. Oxford, UK: Oxford University Press, 2004.

[29] den Haan, W. J., and A. Levin. "A Practitioner's Guide to Robust Covariance Matrix Estimation." In *Handbook of Statistics*. Edited by G. S. Maddala and C. R. Rao. Amsterdam: Elsevier, 1997.

[30] Dickey, D. A., and W. A. Fuller. “Distribution of the Estimators for Autoregressive Time Series with a Unit Root.” *Journal of the American Statistical Association*. Vol. 74, 1979, pp. 427–431.

[31] Dickey, D. A., and W. A. Fuller. “Likelihood Ratio Statistics for Autoregressive Time Series with a Unit Root.” *Econometrica*. Vol. 49, 1981, pp. 1057–1072.

[32] Durbin J., and S. J. Koopman. “A Simple and Efficient Simulation Smoother for State Space Time Series Analysis.” *Biometrika*. Vol 89., No. 3, 2002, pp. 603–615.

[33] Durbin J., and S. J. Koopman. *Time Series Analysis by State Space Methods*. 2nd ed. Oxford: Oxford University Press, 2012.

[34] Elder, J., and P. E. Kennedy. “Testing for Unit Roots: What Should Students Be Taught?” *Journal of Economic Education*. Vol. 32, 2001, pp. 137–146.

[35] Enders, W. *Applied Econometric Time Series*. Hoboken, NJ: John Wiley & Sons, Inc., 1995.

[36] Engle, Robert F. “Autoregressive Conditional Heteroskedasticity with Estimates of the Variance of United Kingdom Inflation.” *Econometrica*. Vol. 50, 1982, pp. 987–1007.

[37] Engle, R. F. and C. W. J. Granger. “Co-Integration and Error-Correction: Representation, Estimation, and Testing.” Econometrica. v. 55, 1987, pp. 251–276.

[38] Engle, Robert F., D. M. Lilien, and R. P. Robins. “Estimating Time Varying Risk Premia in the Term Structure: The ARCH-M Model.” *Econometrica*. Vol. 59, 1987, pp. 391–407.

[39] Faust, J. “When Are Variance Ratio Tests for Serial Dependence Optimal?” *Econometrica*. Vol. 60, 1992, pp. 1215–1226.

[40] Findley, D. F., B. C. Monsell, W. R. Bell, M. C. Otto, and B.-C. Chen. "New Capabilities and Methods of the X-12-ARIMA Seasonal-Adjustment Program." *Journal of Business & Economic Statistics*. Vol. 16, Number 2, 1998, pp. 127–152 .

[41] Fisher, F. M. “Tests of Equality Between Sets of Coefficients in Two Linear Regressions: An Expository Note.” *Econometrica*. Vol. 38, 1970, pp. 361–66.

[42] Gallant, A. R. *Nonlinear Statistical Models*. Hoboken, NJ: John Wiley & Sons, Inc., 1987.

[43] Glasserman, P. *Monte Carlo Methods in Financial Engineering*. New York: Springer-Verlag, 2004.

[44] Glosten, L. R., R. Jagannathan, and D. E. Runkle. “On the Relation between the Expected Value and the Volatility of the Nominal Excess Return on Stocks.” *The Journal of Finance*. Vol. 48, No. 5, 1993, pp. 1779–1801.

[45] Godfrey, L. G. *Misspecification Tests in Econometrics*. Cambridge, UK: Cambridge University Press, 1997.

[46] Gourieroux, C. *ARCH Models and Financial Applications*. New York: Springer-Verlag, 1997.

[47] Granger, C. W. J., and P. Newbold. “Spurious Regressions in Econometrics.” *Journal of Econometrics*. Vol2, 1974, pp. 111–120.

[48] Greene, W. H. *Econometric Analysis*. 6th ed. Upper Saddle River, NJ: Prentice Hall, 2008.

[49] Goldfeld, S. M., and Quandt, R. E. "Some Tests for Homoscedasticity". *Journal of the American Statistical Association*. v. 60, 1965, pp. 539–547.

[50] Hagerud, G. E. “Modeling Nordic Stock Returns with Asymmetric GARCH.” *Working Paper Series in Economics and Finance*. No.164, Stockholm: Department of Finance, Stockholm School of Economics, 1997.

[51] Hagerud, G. E. "Specification Tests for Asymmetric GARCH." *Working Paper Series in Economics and Finance*. No. 163, Stockholm: Department of Finance, Stockholm School of Economics, 1997.

[52] Hamilton, J. D. *Time Series Analysis*. Princeton, NJ: Princeton University Press, 1994.

[53] Haug, A. "Testing Linear Restrictions on Cointegrating Vectors: Sizes and Powers of Wald Tests in Finite Samples." *Econometric Theory*. v. 18, 2002, pp. 505–524.

[54] Helwege, J., and P. Kleiman. "Understanding Aggregate Default Rates of High Yield Bonds." Federal Reserve Bank of New York *Current Issues in Economics and Finance*. Vol.2, No. 6, 1996, pp. 1-6.

[55] Hentschel, L. "All in the Family: Nesting Symmetric and Asymmetric GARCH Models." *Journal of Financial Economics*. Vol. 39, 1995, pp. 71–104.

[56] Hull, J. C. *Options, Futures, and Other Derivatives*. 5th ed. Englewood Cliffs, NJ: Prentice Hall, 2002.

[57] Hodrick, Robert J, and Edward C. Prescott. "Postwar U.S. Business Cycles: An Empirical Investigation." *Journal of Money, Credit, and Banking*. Vol. 29, No. 1, February 1997, pp. 1–16.

[58] Kutner, M. H., C. J. Nachtsheim, J. Neter, and W. Li. *Applied Linear Statistical Models*. 5th Ed. New York: McGraw-Hill/Irwin, 2005.

[59] Kwiatkowski, D., P. C. B. Phillips, P. Schmidt and Y. Shin. "Testing the Null Hypothesis of Stationarity against the Alternative of a Unit Root." *Journal of Econometrics*. Vol. 54, 1992, pp. 159–178.

[60] Jarrow, A. *Finance Theory*. Englewood Cliffs, NJ: Prentice-Hall, 1988.

[61] Johansen, S. *Likelihood-Based Inference in Cointegrated Vector Autoregressive Models*. Oxford: Oxford University Press, 1995.

[62] Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*. Vol. 2, 2nd ed. New York: John Wiley & Sons, 1995.

[63] Judge, G. G., W. E. Griffiths, R. C. Hill, H. Lütkepohl, and T. C. Lee. *The Theory and Practice of Econometrics*. New York, NY: John Wiley & Sons, Inc., 1985.

[64] Juselius, K. *The Cointegrated VAR Model*. Oxford: Oxford University Press, 2006.

[65] Leybourne, S. J. and B. P. M. McCabe. “A Consistent Test for a Unit Root.” *Journal of Business and Economic Statistics*. Vol. 12, 1994, pp. 157–166.

[66] Leybourne, S. J. and B. P. M. McCabe. “Modified Stationarity Tests with Data-Dependent Model-Selection Rules.” *Journal of Business and Economic Statistics*. Vol. 17, 1999, pp. 264–270.

[67] Ljung, G. and G. E. P. Box. "On a Measure of Lack of Fit in Time Series Models." *Biometrika*. Vol. 66, 1978, pp. 67–72.

[68] Lo, A. W., and A. C. MacKinlay. “Stock Market Prices Do Not Follow Random Walks: Evidence from a Simple Specification Test.” *Review of Financial Studies*. Vol. 1, 1988, pp. 41–66.

[69] Lo, A. W., and A. C. MacKinlay. “The Size and Power of the Variance Ratio Test.” *Journal of Econometrics*. Vol. 40, 1989, pp. 203–238.

[70] Lo, A. W., and A. C. MacKinlay. *A Non-Random Walk Down Wall St*. Princeton, NJ: Princeton University Press, 2001.

[71] Loeffler, G., and P. N. Posch. *Credit Risk Modeling Using Excel and VBA*. West Sussex, England: Wiley Finance, 2007.

[72] Long, J. S., and L. H. Ervin. "Using Heteroscedasticity-Consistent Standard Errors in the Linear Regression Model." *The American Statistician*. v. 54, 2000, pp. 217-224.

[73] Longstaff, F. A., and E. S. Schwartz. “Valuing American Options by Simulation: A Simple Least-Squares Approach.” *The Review of Financial Studies*. Spring 2001, Vol. 14, No. 1, pp. 113–147.

[74] Lütkepohl, H. *New Introduction to Multiple Time Series Analysis*. Berlin: Springer, 2005.

[75] MacKinnon, J. G. “Numerical Distribution Functions for Unit Root and Cointegration Tests.” Journal of Applied Econometrics. v. 11, 1996, pp. 601–618.

[76] MacKinnon, J. G., and H. White. "Some Heteroskedasticity-Consistent Covariance Matrix Estimators with Improved Finite Sample Properties." *Journal of Econometrics*. v. 29, 1985, pp. 305-325.

[77] McCullough, B. D., and C. G. Renfro. “Benchmarks and Software Standards: A Case Study of GARCH Procedures.” *Journal of Economic and Social Measurement*. Vol. 25, 1998, pp. 59–71.

[78] McLeod, A.I. and W.K. Li. “Diagnostic Checking ARMA Time Series Models Using Squared-Residual Autocorrelations.”*Journal of Time Series Analysis*. Vol. 4, 1983, pp. 269–273.

[79] Morin, N. "Likelihood Ratio Tests on Cointegrating Vectors, Disequilibrium Adjustment Vectors, and their Orthogonal Complements." *European Journal of Pure and Applied Mathematics*. v. 3, 2010, pp. 541–571.

[80] Nelson, D. B. “Conditional Heteroskedasticity in Asset Returns: A New Approach.” *Econometrica*. Vol. 59, 1991, pp. 347–370.

[81] Nelson, C., and C. Plosser. “Trends and Random Walks in Macroeconomic Time Series: Some Evidence and Implications.” *Journal of Monetary Economics*. Vol. 10, 1982, pp. 130–162.

[82] Newey, W. K., and K. D. West. “A Simple Positive Semidefinite, Heteroskedasticity and Autocorrelation Consistent Covariance Matrix.” *Econometrica*. Vol. 55, 1987, pp. 703–708.

[83] Newey, W. K, and K. D. West. “Automatic Lag Selection in Covariance Matrix Estimation.” *The Review of Economic Studies*. v. 61 No. 4, 1994, pp. 631–653.

[84] Pankratz, A. *Forecasting with Dynamic Regression Models*. John Wiley & Sons, 1991˙.

[85] Ng, S., and P. Perron. “Unit Root Tests in ARMA Models with Data-Dependent Methods for the Selection of the Truncation Lag.” *Journal of the American Statistical Association*. Vol. 90, 1995, pp. 268–281.

[86] Park, R. E. "Estimation with Heteroscedastic Error Terms". *Econometrica*. 34, 1966 p. 888.

[87] Perron, P. “Trends and Random Walks in Macroeconomic Time Series: Further Evidence from a New Approach.” *Journal of Economic Dynamics and Control*. Vol. 12, 1988, pp. 297–332.

[88] Pesaran, H. H. and Y. Shin. “Generalized Impulse Response Analysis in Linear Multivariate Models.” *Economic Letters*. Vol. 58, 1998, 17–29.

[89] Peters, J. P. “Estimating and Forecasting Volatility of Stock Indices Using Asymmetric GARCH Models and Skewed Student-t Densities.” Working Paper. Belgium: École d'Administration des Affaires, University of Liège, March 20, 2001.

[90] Phillips, P. “Time Series Regression with a Unit Root.” *Econometrica*. Vol. 55, 1987, pp. 277–301.

[91] Phillips, P., and P. Perron. “Testing for a Unit Root in Time Series Regression." *Biometrika*. Vol. 75, 1988, pp. 335–346.

[92] Rea, J. D. “Indeterminacy of the Chow Test When the Number of Observations is Insufficient.” *Econometrica*. Vol. 46, 1978, p. 229.

[93] Schwert, W. “Effects of Model Specification on Tests for Unit Roots in Macroeconomic Data.” *Journal of Monetary Economics*. Vol. 20, 1987, pp. 73–103.

[94] Schwert, W. “Tests for Unit Roots: A Monte Carlo Investigation.” *Journal of Business and Economic Statistics*. Vol. 7, 1989, pp. 147–159.

[95] Sharpe, W. F. “Capital Asset Prices: A Theory of Market Equilibrium under Conditions of Risk.” *Journal of Finance*. Vol. 19, 1964, pp. 425–442.

[96] Shreve, S. E. *Stochastic Calculus for Finance II: Continuous-Time Models*. New York: Springer-Verlag, 2004.

[97] Sims, C., Stock, J., and Watson, M. “Inference in Linear Time Series Models with Some Unit Roots.” *Econometrica*. Vol. 58, 1990, pp. 113–144.

[98] Tsay,R.S. *Analysis of Financial Time Series*. Hoboken, NJ: John Wiley & Sons, Inc., 2005.

[99] Turner, P. M. "Testing for Cointegration Using the Johansen Approach: Are We Using the Correct Critical Values?" *Journal of Applied Econometrics*. v. 24, 2009, pp. 825–831.

[100] U.S. Federal Reserve Economic Data (FRED), Federal Reserve Bank of St. Louis, http://research.stlouisfed.org/fred.

[101] White, H. "A Heteroskedasticity-Consistent Covariance Matrix and a Direct Test for Heteroskedasticity." *Econometrica*. v. 48, 1980, pp. 817-838.

[102] White, H. *Asymptotic Theory for Econometricians*. New York: Academic Press, 1984.

[103] White, H., and I. Domowitz. "Nonlinear Regression with Dependent Observations." *Econometrica*. Vol. 52, 1984, pp. 143–162.

[104] Wilson, A. L. "When is the Chow Test UMP?" *The American Statistician*. Vol. 32, 1978, pp. 66–68.

[105] Wold, H. *A Study in the Analysis of Stationary Time Series*. Uppsala, Sweden: Almqvist & Wiksell, 1938.

Data Preprocessing

Data Transformations

In this section...

Why Transform?

You can transform time series to:

- Isolate temporal components of interest.
- Remove the effect of nuisance components (like seasonality).
- Make a series stationary.
- Reduce spurious regression effects.
- Stabilize variability that grows with the level of the series.
- Make two or more time series more directly comparable.

You can choose among many data transformation to address these (and other) aims.

For example, you can use decomposition methods to describe and estimate time series components. Seasonal adjustment is a decomposition method you can use to remove a nuisance seasonal component.

Detrending and differencing are transformations you can use to address nonstationarity due to a trending mean. Differencing can also help remove spurious regression effects due to cointegration.

In general, if you apply a data transformation before modeling your data, you then need to back-transform model forecasts to return to the original scale. This is not necessary in Econometrics Toolbox if you are modeling difference-stationary data. Use `arima` to model integrated series that are not *a priori* differenced. A key advantage of this is that `arima` also returns forecasts on the original scale automatically.

Common Data Transformations

- "Detrending" on page 2-3
- "Differencing" on page 2-3

Detrending

Some nonstationary series can be modeled as the sum of a deterministic trend and a stationary stochastic process. That is, you can write the series y_t as

$$y_t = \mu_t + \varepsilon_t,$$

where ε_t is a stationary stochastic process with mean zero.

The deterministic trend, μ_t, can have multiple components, such as nonseasonal and seasonal components. You can detrend (or decompose) the data to identify and estimate its various components. The detrending process proceeds as follows:

1. Estimate the deterministic trend component.
2. Remove the trend from the original data.
3. (Optional) Model the remaining residual series with an appropriate stationary stochastic process.

Several techniques are available for estimating the trend component. You can estimate it parametrically using least squares, nonparametrically using filters (moving averages), or a combination of both.

Detrending yields estimates of all trend and stochastic components, which might be desirable. However, estimating trend components can require making additional assumptions, performing extra steps, and estimating additional parameters.

Differencing

Differencing is an alternative transformation for removing a mean trend from a nonstationary series. This approach is advocated in the Box-Jenkins approach to model specification [1]. According to this methodology, the first step to build models is differencing your data until it looks stationary. Differencing is appropriate for removing *stochastic* trends (e.g., random walks).

Define the first difference as

$$\Delta y_t = y_t - y_{t-1},$$

where Δ is called the *differencing operator*. In lag operator notation, where $L^i y_t = y_{t-i}$,

$$\Delta y_t = (1 - L) y_t.$$

You can create lag operator polynomial objects using `LagOp`.

Similarly, define the second difference as

$$\Delta^2 y_t = (1 - L)^2 y_t = (y_t - y_{t-1}) - (y_{t-1} - y_{t-2}) = y_t - 2y_{t-1} + y_{t-2}.$$

Like taking derivatives, taking a first difference makes a linear trend constant, taking a second difference makes a quadratic trend constant, and so on for higher-degree polynomials. Many complex stochastic trends can also be eliminated by taking relatively low-order differences. Taking D differences makes a process with D unit roots stationary.

For series with seasonal periodicity, *seasonal* differencing can address seasonal unit roots. For data with periodicity s (e.g., quarterly data have $s = 4$ and monthly data have $s = 12$), the seasonal differencing operator is defined as

$$\Delta_s y_t = (1 - L^s) y_t = y_t - y_{t-s}.$$

Using a differencing transformation eliminates the intermediate estimation steps required for detrending. However, this means you can't obtain separate estimates of the trend and stochastic components.

Log Transformations

For a series with exponential growth and variance that grows with the level of the series, a log transformation can help linearize and stabilize the series. If you have negative values in your time series, you should add a constant large enough to make all observations greater than zero before taking the log transformation.

In some application areas, working with differenced, logged series is the norm. For example, the first differences of a logged time series,

$$\Delta \log y_t = \log y_t - \log y_{t-1},$$

are approximately the *rates of change* of the series.

Prices, Returns, and Compounding

The rates of change of a price series are called *returns*. Whereas price series do not typically fluctuate around a constant level, the returns series often looks stationary. Thus, returns series are typically used instead of price series in many applications.

Denote successive price observations made at times t and $t + 1$ as y_t and y_{t+1}, respectively. The *continuously compounded returns series* is the transformed series

$$r_t = \log \frac{y_{t+1}}{y_t} = \log y_{t+1} - \log y_t.$$

This is the first difference of the log price series, and is sometimes called the *log return*.

An alternative transformation for price series is *simple returns*,

$$r_t = \frac{y_{t+1} - y_t}{y_t} = \frac{y_{t+1}}{y_t} - 1.$$

For series with relatively high frequency (e.g., daily or weekly observations), the difference between the two transformations is small. Econometrics Toolbox has `price2ret` for converting price series to returns series (with either continuous or simple compounding), and `ret2price` for the inverse operation.

References

[1] Box, G. E. P., G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis: Forecasting and Control*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

See Also

LagOp | `price2ret` | `ret2price`

Related Examples

- “Moving Average Trend Estimation” on page 2-33
- “Nonseasonal Differencing” on page 2-18
- “Nonseasonal and Seasonal Differencing” on page 2-23
- “Parametric Trend Estimation” on page 2-37

- "Specify Lag Operator Polynomials" on page 2-11

More About

- "Trend-Stationary vs. Difference-Stationary Processes" on page 2-7
- "Moving Average Filter" on page 2-31
- "Seasonal Adjustment" on page 2-54
- "Time Series Decomposition" on page 2-28

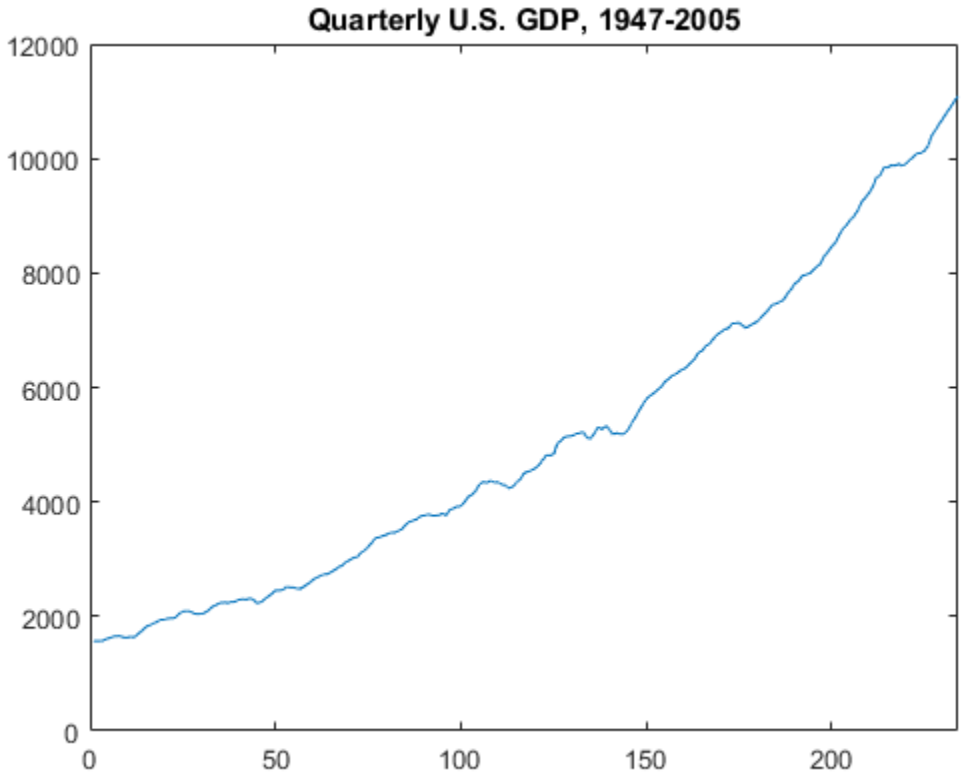
Trend-Stationary vs. Difference-Stationary Processes

In this section...

Nonstationary Processes

The stationary stochastic process is a building block of many econometric time series models. Many observed time series, however, have empirical features that are inconsistent with the assumptions of stationarity. For example, the following plot shows quarterly U.S. GDP measured from 1947 to 2005. There is a very obvious upward trend in this series that one should incorporate into any model for the process.

```
load Data_GDP
plot(Data)
xlim([0,234])
title('Quarterly U.S. GDP, 1947-2005')
```



A trending mean is a common violation of stationarity. There are two popular models for nonstationary series with a trending mean.

- *Trend stationary*: The mean trend is deterministic. Once the trend is estimated and removed from the data, the residual series is a stationary stochastic process.
- *Difference stationary*: The mean trend is stochastic. Differencing the series D times yields a stationary stochastic process.

The distinction between a deterministic and stochastic trend has important implications for the long-term behavior of a process:

- Time series with a deterministic trend always revert to the trend in the long run (the effects of shocks are eventually eliminated). Forecast intervals have constant width.

- Time series with a stochastic trend never recover from shocks to the system (the effects of shocks are permanent). Forecast intervals grow over time.

Unfortunately, for any finite amount of data there is a deterministic and stochastic trend that fits the data equally well (Hamilton, 1994). Unit root tests are a tool for assessing the presence of a stochastic trend in an observed series.

Trend Stationary

You can write a trend-stationary process, y_t, as

$$y_t = \mu_t + \varepsilon_t,$$

where:

- μ_t is a deterministic mean trend.
- ε_t is a stationary stochastic process with mean zero.

In some applications, the trend is of primary interest. Time series decomposition methods focus on decomposing μ_t into different trend sources (e.g., secular trend component and seasonal component). You can decompose series nonparametrically using filters (moving averages), or parametrically using regression methods.

Given an estimate $\hat{\mu}_t$, you can explore the residual series $y_t - \hat{\mu}_t$ for autocorrelation, and optionally model it using a stationary stochastic process model.

Difference Stationary

In the Box-Jenkins modeling approach [2], nonstationary time series are differenced until stationarity is achieved. You can write a difference-stationary process, y_t, as

$$\Delta^D y_t = \mu + \psi(L)\varepsilon_t,$$

where:

- $\Delta^D = (1-L)^D$ is a Dth-degree differencing operator.

- $\psi(L) = (1 + \psi_1 L + \psi_2 L^2 + \ldots)$ is an infinite-degree lag operator polynomial with absolutely summable coefficients and all roots lying outside the unit circle.
- ε_t is an uncorrelated innovation process with mean zero.

Time series that can be made stationary by differencing are called *integrated* processes. Specifically, when D differences are required to make a series stationary, that series is said to be *integrated of order D*, denoted $I(D)$. Processes with $D \geq 1$ are often said to have a *unit root*.

References

[1] Hamilton, J. D. *Time Series Analysis*. Princeton, NJ: Princeton University Press, 1994.

[2] Box, G. E. P., G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis: Forecasting and Control*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

Related Examples

- "Nonseasonal Differencing" on page 2-18
- "Moving Average Trend Estimation" on page 2-33
- "Specify Lag Operator Polynomials" on page 2-11

More About

- "Moving Average Filter" on page 2-31
- "Time Series Decomposition" on page 2-28
- "ARIMA Model" on page 5-41

Specify Lag Operator Polynomials

In this section...

Lag Operator Polynomial of Coefficients

Define the lag operator L such that $L^i y_t = y_{t-i}$. An m-degree polynomial of coefficients A in the lag operator L is given by

$$A(L) = (A_0 + A_1 L^1 + \ldots + A_m L^m).$$

Here, the coefficient A_0 corresponds to lag 0, A_1 corresponds to lag 1, and so on, to A_m, which corresponds to lag m.

To specify a coefficient lag operator polynomial in Econometrics Toolbox, use `LagOp`. Specify the (nonzero) coefficients $A_0,\ldots,A_m$ as a cell array, and the lags of the nonzero coefficients as a vector.

The coefficients of lag operator polynomial objects are designed to look and feel like traditional MATLAB cell arrays. There is, however, an important difference: elements of cell arrays are accessible by positive integer sequential indexing, i.e., 1, 2, 3,.... The coefficients of lag operator polynomial objects are accessible by lag-based indexing. That is, you can specify any nonnegative integer lags, including lag 0.

For example, consider specifying the polynomial $A(L) = (1 - 0.3L + 0.6L^4)$. This polynomial has coefficient 1 at lag 0, coefficient –0.3 at lag 1, and coefficient 0.6 at lag 4. Enter:

```
A = LagOp({1,-0.3,0.6},'Lags',[0,1,4])
```

```
A =

    1-D Lag Operator Polynomial:
    ----------------------------
```

```
    Coefficients: [1 -0.3 0.6]
            Lags: [0 1 4]
          Degree: 4
       Dimension: 1
```

The created lag operator object `A` corresponds to a lag operator polynomial of degree 4. A `LagOp` object has a number of properties describing it:

- `Coefficients`, a cell array of coefficients.
- `Lags`, a vector indicating the lags of nonzero coefficients.
- `Degree`, the degree of the polynomial.
- `Dimension`, the dimension of the polynomial (relevant for multivariate time series).

To access properties of the model, use dot notation. That is, enter the variable name and then the property name, separated by a period. To access specific coefficients, use dot notation along with cell array syntax (consistent with the `Coefficients` data type).

To illustrate, returns the coefficient at lag 4:

```
A.Coefficients{4}
```

```
ans =

    0.6000
```

Return the coefficient at lag 0:

```
A.Coefficients{0}
```

```
ans =

     1
```

This last command illustrates lag indexing. The index 0 is valid, and corresponds to the lag 0 coefficient.

Notice what happens if you index a lag larger than the degree of the polynomial:

```
A.Coefficients{6}
```

```
ans =

     0
```

This does not return an error. Rather, it returns `0`, the coefficient at lag 6 (and all other lags with coefficient zero).

Use similar syntax to add new nonzero coefficients. For example, to add the coefficient 0.4 at lag 6,

```
A.Coefficients{6} = 0.4
```

```
A =

    1-D Lag Operator Polynomial:
    ----------------------------
        Coefficients: [1 -0.3 0.6 0.4]
                Lags: [0 1 4 6]
              Degree: 6
           Dimension: 1
```

The lag operator polynomial object `A` now has nonzero coefficients at lags 0, 1, 4, and 6, and is degree 6.

When lag indices are placed inside of parentheses the result is another lag-based cell array that represent a subset of the original polynomial.

```
A0 = A.Coefficients(0)
```

```
A0 =

    1-D Lag-Indexed Cell Array Created at Lags [0] with
    Non-Zero Coefficients at Lags [0].
```

`A0` is a new object that preserves lag-based indexing and is suitable for assignment to and from lag operator polynomial.

```
class(A0)
```

```
ans =
```

```
internal.econ.LagIndexedArray
```

In contrast, when lag indices are placed inside curly braces, the result is the same data type as the indices themselves:

```
class(A.Coefficients{0})
```

```
ans =

double
```

Difference Lag Operator Polynomials

You can express the differencing operator, Δ, in lag operator polynomial notation as

$$\Delta = (1 - L).$$

More generally,

$$\Delta^D = (1 - L)^D.$$

To specify a first differencing operator polynomial using `LagOp`, specify coefficients 1 and –1 at lags 0 and 1:

```
D1 = LagOp({1,-1},'Lags',[0,1])
```

```
D1 =

    1-D Lag Operator Polynomial:
    ----------------------------
        Coefficients: [1 -1]
                Lags: [0 1]
              Degree: 1
           Dimension: 1
```

Similarly, the seasonal differencing operator in lag polynomial notation is

$\Delta_s = (1 - L^s)$.

This has coefficients 1 and –1 at lags 0 and s, where s is the periodicity of the seasonality. For example, for monthly data with periodicity $s = 12$,

```
D12 = LagOp({1,-1},'Lags',[0,12])
```

```
D12 =

    1-D Lag Operator Polynomial:
    ----------------------------
        Coefficients: [1 -1]
                Lags: [0 12]
              Degree: 12
           Dimension: 1
```

This results in a polynomial object with degree 12.

When a difference lag operator polynomial is applied to a time series y_t, $(1 - L)^D y_t$, this is equivalent to filtering the time series. Note that filtering a time series using a polynomial of degree D results in the loss of the first D observations.

Consider taking second differences of a time series y_t, $(1 - L)^2 y_t$. You can write this differencing polynomial as $(1 - L)^2 = (1 - L)(1 - L)$.

Create the second differencing polynomial by multiplying the polynomial `D1` to itself to get the second-degree differencing polynomial:

```
D2 = D1*D1
```

```
D2 =

    1-D Lag Operator Polynomial:
    ----------------------------
        Coefficients: [1 -2 1]
                Lags: [0 1 2]
              Degree: 2
           Dimension: 1
```

The coefficients in the second-degree differencing polynomial correspond to the coefficients in the difference equation

$$(1-L)^2 y_t = y_t - 2y_{t-1} + y_{t-2}.$$

To see the effect of filtering (differencing) on the length of a time series, simulate a data set with 10 observations to filter:

```
rng('default')
Y = randn(10,1);
```

Filter the time series `Y` using `D2`:

```
Yf = filter(D2,Y);
length(Yf)
```

```
ans =

     8
```

The filtered series has two observations less than the original series. The time indices for the new series can be optionally returned:

```
[Yf,Tidx] = filter(D2,Y);
Tidx
```

```
Tidx =

     2
     3
     4
     5
     6
     7
     8
     9
```

Note that the time indices are given relative to time 0. That is, the original series corresponds to times 0,...,9. The filtered series loses the observations at the first two times (times 0 and 1), resulting in a series corresponding to times 2,...,9.

You can also filter a time series, say `Y`, with a lag operator polynomial, say `D2`, using this shorthand syntax:

```
Yf = D2(Y);
```

See Also

filter | LagOp

Related Examples

- "Nonseasonal Differencing" on page 2-18
- "Nonseasonal and Seasonal Differencing" on page 2-23
- "Plot the Impulse Response Function" on page 5-88

More About

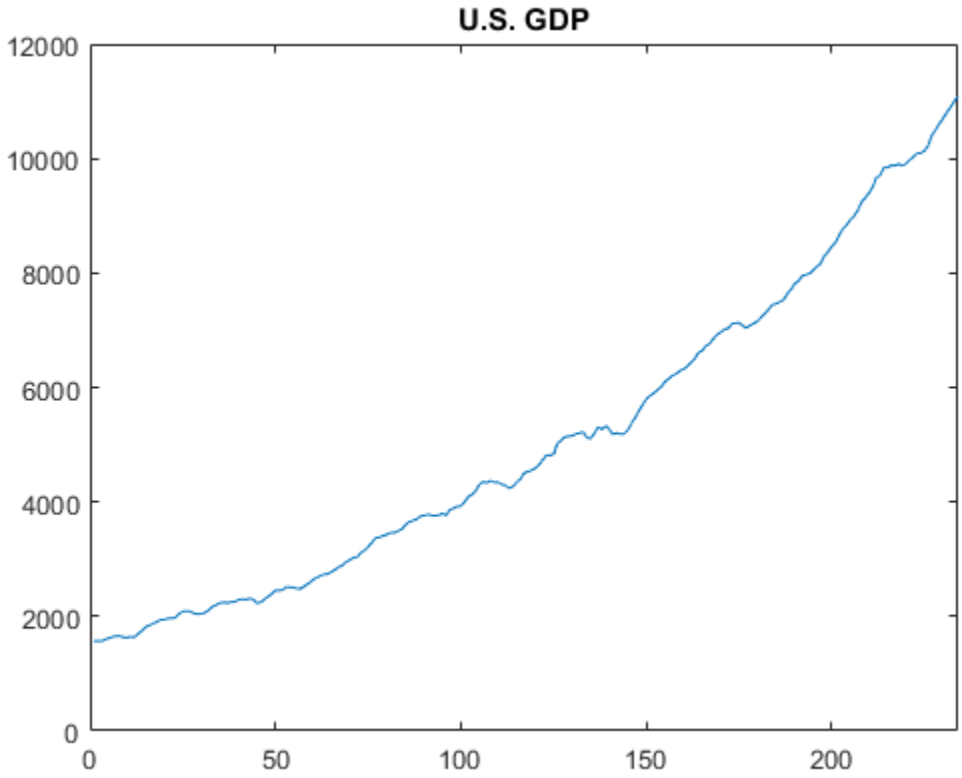
- "Moving Average Filter" on page 2-31

Nonseasonal Differencing

This example shows how to take a nonseasonal difference of a time series. The time series is quarterly U.S. GDP measured from 1947 to 2005.

Load the GDP data set included with the toolbox.

```
load Data_GDP
Y = Data;
N = length(Y);

figure
plot(Y)
xlim([0,N])
title('U.S. GDP')
```



The time series has a clear upward trend.

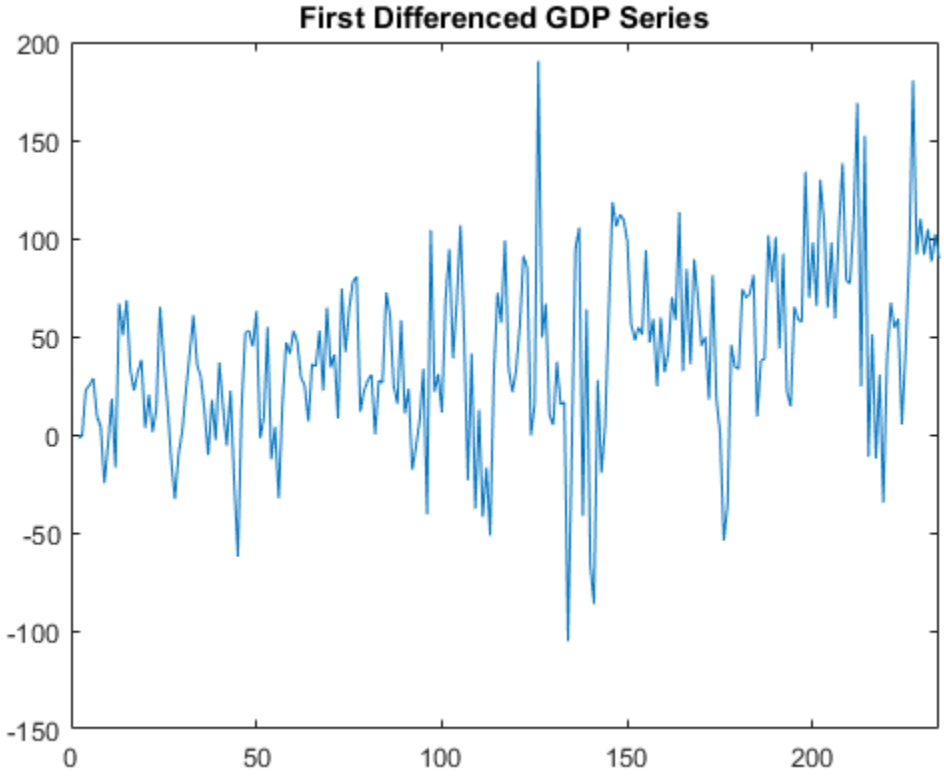
Take a first difference of the series to remove the trend,

$$\Delta y_t = (1 - L)y_t = y_t - y_{t-1}.$$

First create a differencing lag operator polynomial object, and then use it to filter the observed series.

```
D1 = LagOp({1,-1},'Lags',[0,1]);
dY = filter(D1,Y);

figure
plot(2:N,dY)
```

```
xlim([0,N])
title('First Differenced GDP Series')
```



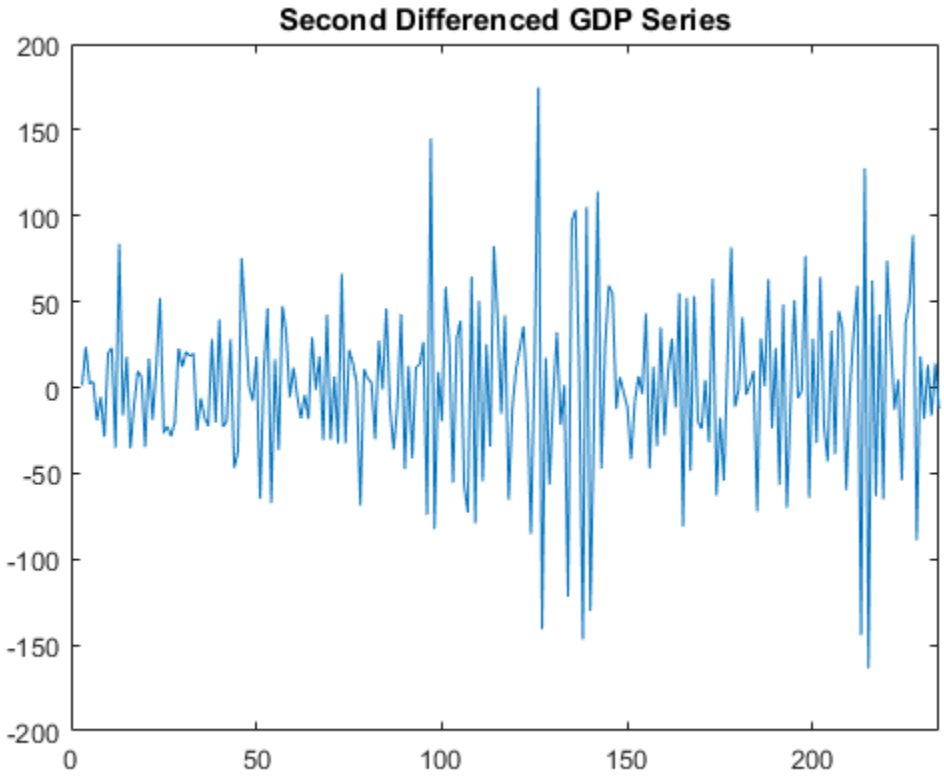
The series still has some remaining upward trend after taking first differences.

Take a second difference of the series,

$$\Delta^2 y_t = (1-L)^2 y_t = y_t - 2y_{t-1} + y_{t-2}.$$

```
D2 = D1*D1;
ddY = filter(D2,Y);

figure
plot(3:N,ddY)
```

```
xlim([0,N])
title('Second Differenced GDP Series')
```



The second-differenced series appears more stationary.

See Also

filter | LagOp

Related Examples

- “Nonseasonal and Seasonal Differencing” on page 2-23
- “Specify Lag Operator Polynomials” on page 2-11

More About

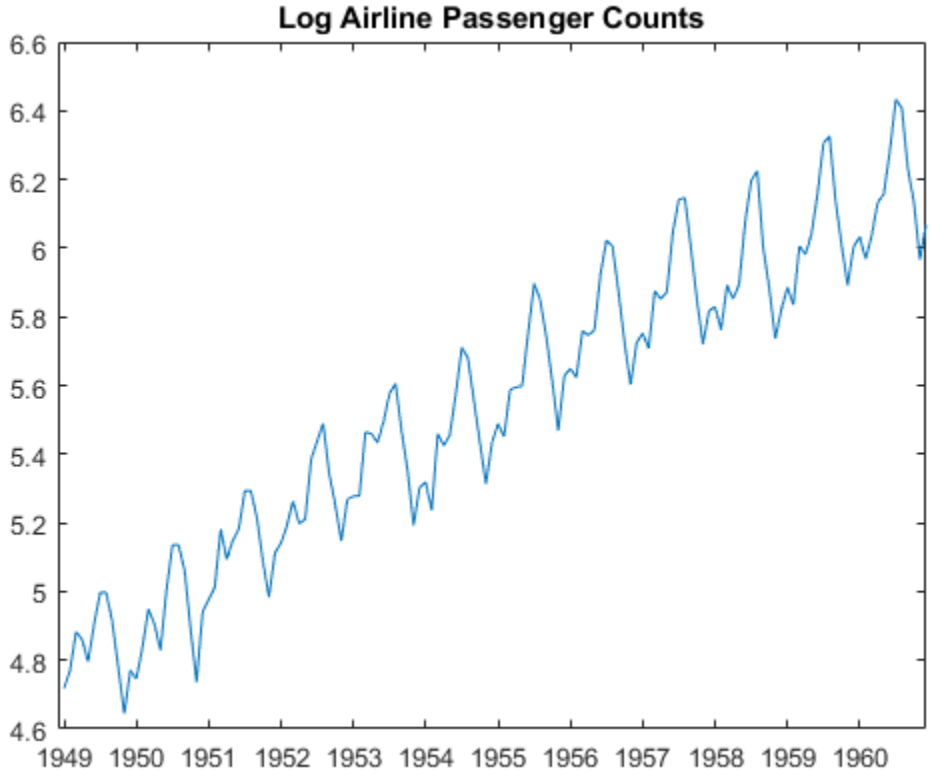
- "Data Transformations" on page 2-2
- "Trend-Stationary vs. Difference-Stationary Processes" on page 2-7

Nonseasonal and Seasonal Differencing

This example shows how to apply both nonseasonal and seasonal differencing using lag operator polynomial objects. The time series is monthly international airline passenger counts from 1949 to 1960.

Load the airline data set (`Data_Airline.mat`).

```
load(fullfile(matlabroot,'examples','econ','Data_Airline.mat'))
y = log(Data);
T = length(y);

figure
plot(y)
h1 = gca;
h1.XLim = [0,T];
h1.XTick = [1:12:T];
h1.XTickLabel = datestr(dates(1:12:T),10);
title 'Log Airline Passenger Counts';
```



The data shows a linear trend and a seasonal component with periodicity 12.

Take the first difference to address the linear trend, and the 12th difference to address the periodicity. If y_t is the series to be transformed, the transformation is

$$\Delta\Delta_{12}y_t = (1-L)(1-L^{12})y_t,$$

where Δ denotes the difference operator, and L denotes the lag operator.

Create the lag operator polynomials $1-L$ and $1-L^{12}$. Then, multiply them to get the desired lag operator polynomial.

```
D1 = LagOp({1 -1},'Lags',[0,1]);
```

```
D12 = LagOp({1 -1},'Lags',[0,12]);
D = D1*D12
```

```
D =

    1-D Lag Operator Polynomial:
    ----------------------------
        Coefficients: [1 -1 -1 1]
                Lags: [0 1 12 13]
              Degree: 13
           Dimension: 1
```

The first polynomial, $1 - L$, has coefficient 1 at lag 0 and coefficient -1 at lag 1. The seasonal differencing polynomial, $1 - L^{12}$, has coefficient 1 at lag 0, and -1 at lag 12. The product of these polynomials is

$$(1 - L)(1 - L^{12}) = 1 - L - L^{12} + L^{13},$$

which has coefficient 1 at lags 0 and 13, and coefficient -1 at lags 1 and 12.

Filter the data with differencing polynomial `D` to get the nonseasonally and seasonally differenced series.

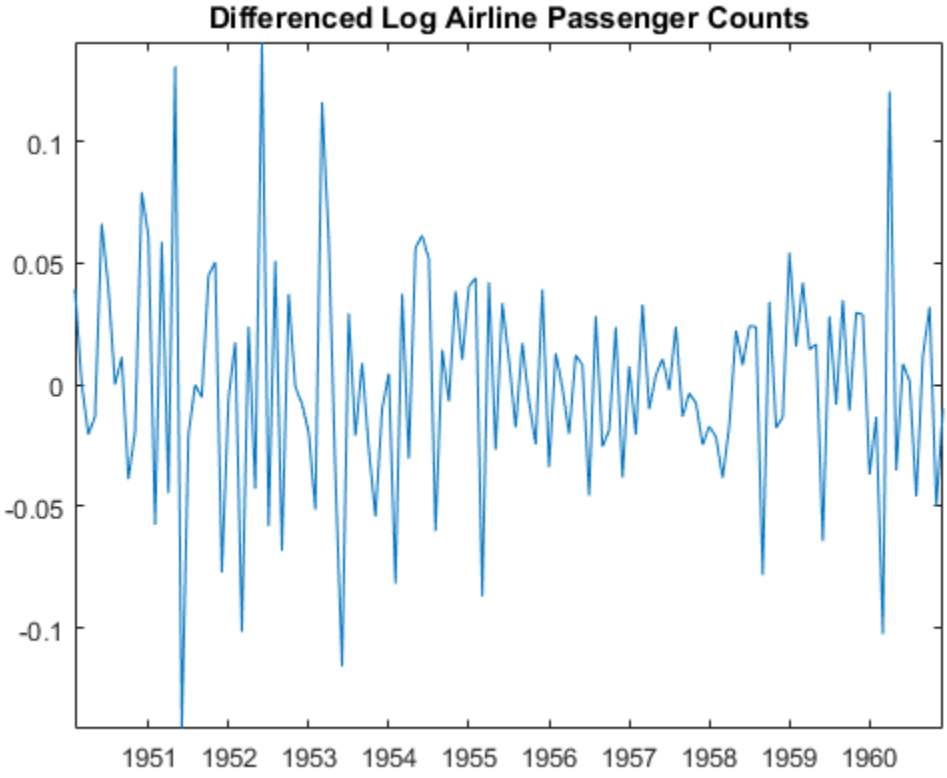
```
dY = filter(D,y);
length(y) - length(dY)
```

```
ans =

    13
```

The filtered series is 13 observations shorter than the original series. This is due to applying a degree 13 polynomial filter.

```
figure
plot(14:T,dY)
h2 = gca;
h2.XLim = [0,T];
h2.XTick = [1:12:T];
h2.XTickLabel = datestr(dates(1:12:T),10);
axis tight;
```

```
title 'Differenced Log Airline Passenger Counts';
```



The differenced series has neither the trend nor seasonal component exhibited by the original series.

See Also

filter | LagOp

Related Examples

- "Nonseasonal Differencing" on page 2-18
- "Specify Lag Operator Polynomials" on page 2-11

More About

- "Data Transformations" on page 2-2
- "Trend-Stationary vs. Difference-Stationary Processes" on page 2-7

Time Series Decomposition

Time series decomposition involves separating a time series into several distinct components. There are three components that are typically of interest:

- T_t, a deterministic, nonseasonal secular trend component. This component is sometimes restricted to being a linear trend, though higher-degree polynomials are also used.
- S_t, a deterministic seasonal component with known periodicity. This component captures level shifts that repeat systematically within the same period (e.g., month or quarter) between successive years. It is often considered to be a nuisance component, and seasonal adjustment is a process for eliminating it.
- I_t, a stochastic irregular component. This component is not necessarily a white noise process. It can exhibit autocorrelation and cycles of unpredictable duration. For this reason, it is often thought to contain information about the business cycle, and is usually the most interesting component.

There are three functional forms that are most often used for representing a time series y_t as a function of its trend, seasonal, and irregular components:

- *Additive decomposition*, where

  $$y_t = T_t + S_t + I_t.$$

  This is the classical decomposition. It is appropriate when there is no exponential growth in the series, and the amplitude of the seasonal component remains constant over time. For identifiability from the trend component, the seasonal and irregular components are assumed to fluctuate around zero.
- *Multiplicative decomposition*, where

  $$y_t = T_t S_t I_t.$$

  This decomposition is appropriate when there is exponential growth in the series, and the amplitude of the seasonal component grows with the level of the series. For identifiability from the trend component, the seasonal and irregular components are assumed to fluctuate around one.
- *Log-additive decomposition*, where

$$\log y_t = T_t + S_t + I_t.$$

This is an alternative to the multiplicative decomposition. If the original series has a multiplicative decomposition, then the logged series has an additive decomposition. Using the logs can be preferable when the time series contains many small observations. For identifiability from the trend component, the seasonal and irregular components are assumed to fluctuate around zero.

You can estimate the trend and seasonal components by using filters (moving averages) or parametric regression models. Given estimates $\hat{T}_t$ and $\hat{S}_t$, the irregular component is estimated as

$$\hat{I}_t = y_t - \hat{T}_t - \hat{S}_t$$

using the additive decomposition, and

$$\hat{I}_t = \frac{y_t}{\left(\hat{T}_t \hat{S}_t\right)}$$

using the multiplicative decomposition.

The series

$$y_t - \hat{T}_t$$

(or $y_t / \hat{T}_t$ using the multiplicative decomposition) is called a *detrended* series.

Similarly, the series $y_t - \hat{S}_t$ (or $y_t / \hat{S}_t$) is called a *deseasonalized* series.

Related Examples

- "Moving Average Trend Estimation" on page 2-33
- "Seasonal Adjustment Using a Stable Seasonal Filter" on page 2-57
- "Seasonal Adjustment Using S(n,m) Seasonal Filters" on page 2-64

- "Parametric Trend Estimation" on page 2-37

More About

- "Data Transformations" on page 2-2
- "Moving Average Filter" on page 2-31
- "Seasonal Adjustment" on page 2-54

Moving Average Filter

Some time series are decomposable into various trend components. To estimate a trend component without making parametric assumptions, you can consider using a *filter*.

Filters are functions that turn one time series into another. By appropriate filter selection, certain patterns in the original time series can be clarified or eliminated in the new series. For example, a low-pass filter removes high frequency components, yielding an estimate of the slow-moving trend.

A specific example of a linear filter is the *moving average*. Consider a time series y_t, $t = 1,...,N$. A symmetric (centered) moving average filter of window length $2q + 1$ is given by

$$\hat{m}_t = \sum_{j=-q}^{q} b_j y_{t+j}, \quad q < t < N - q.$$

You can choose any weights b_j that sum to one. To estimate a slow-moving trend, typically $q = 2$ is a good choice for quarterly data (a 5-term moving average), or $q = 6$ for monthly data (a 13-term moving average). Because symmetric moving averages have an odd number of terms, a reasonable choice for the weights is $b_j = 1/4q$ for $j = \pm q$, and $b_j = 1/2q$ otherwise. Implement a moving average by convolving a time series with a vector of weights using `conv`.

You cannot apply a symmetric moving average to the q observations at the beginning and end of the series. This results in observation loss. One option is to use an asymmetric moving average at the ends of the series to preserve all observations.

See Also

`conv`

Related Examples

- “Moving Average Trend Estimation” on page 2-33
- “Parametric Trend Estimation” on page 2-37

More About

- “Data Transformations” on page 2-2

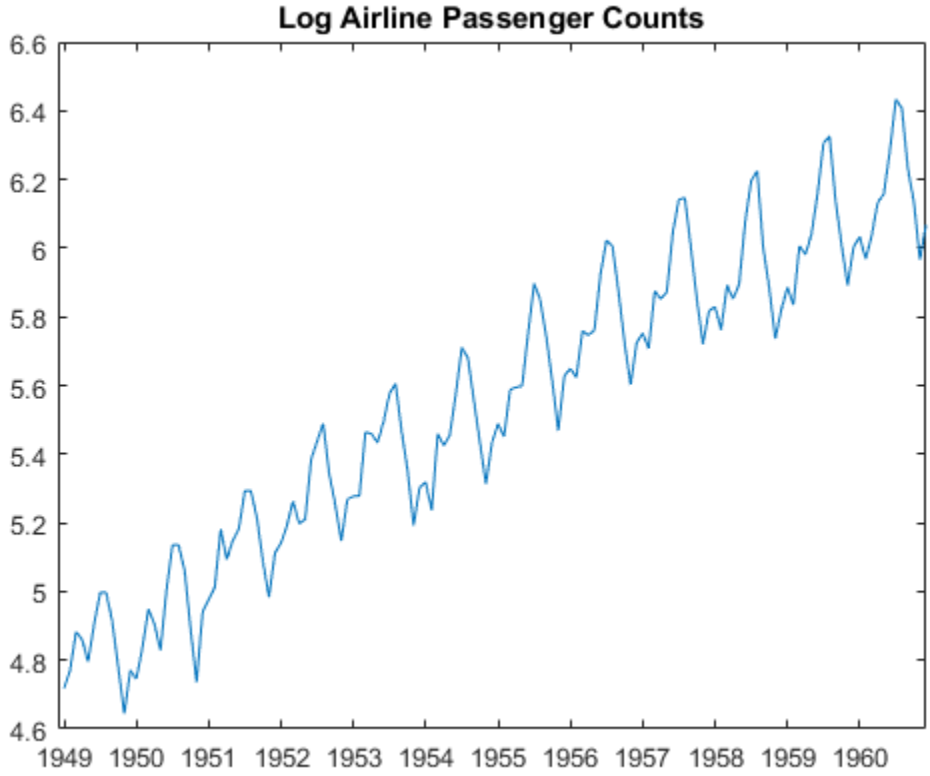
- “Time Series Decomposition” on page 2-28
- “Seasonal Filters” on page 2-51

Moving Average Trend Estimation

This example shows how to estimate long-term trend using a symmetric moving average function. This is a convolution that you can implement using `conv`. The time series is monthly international airline passenger counts from 1949 to 1960.

Load the airline data set (`Data_Airline`).

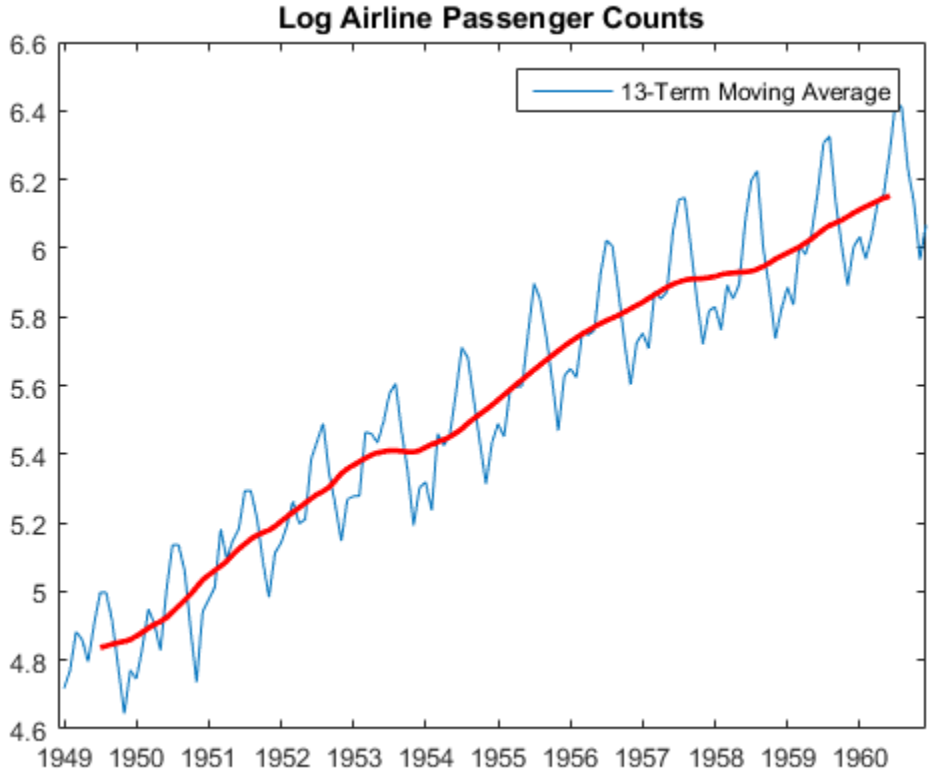
```
load(fullfile(matlabroot,'examples','econ','Data_Airline.mat'))
y = log(Data);
T = length(y);

figure
plot(y)
h = gca;
h.XLim = [0,T];
h.XTick = [1:12:T];
h.XTickLabel = datestr(dates(1:12:T),10);
title 'Log Airline Passenger Counts';
hold on
```



The data shows a linear trend and a seasonal component with periodicity 12.

The periodicity of the data is monthly, so a 13-term moving average is a reasonable choice for estimating the long-term trend. Use weight 1/24 for the first and last terms, and weight 1/12 for the interior terms. Add the moving average trend estimate to the observed time series plot.

```
wts = [1/24;repmat(1/12,11,1);1/24];
yS = conv(y,wts,'valid');

plot(7:T-6,yS,'r','LineWidth',2);
legend('13-Term Moving Average')
hold off
```



When you use the shape parameter `'valid'` in the call to `conv`, observations at the beginning and end of the series are lost. Here, the moving average has window length 13, so the first and last 6 observations do not have smoothed values.

See Also

`conv`

Related Examples

- "Seasonal Adjustment Using a Stable Seasonal Filter" on page 2-57
- "Seasonal Adjustment Using S(n,m) Seasonal Filters" on page 2-64
- "Parametric Trend Estimation" on page 2-37

More About

- "Time Series Decomposition" on page 2-28
- "Moving Average Filter" on page 2-31

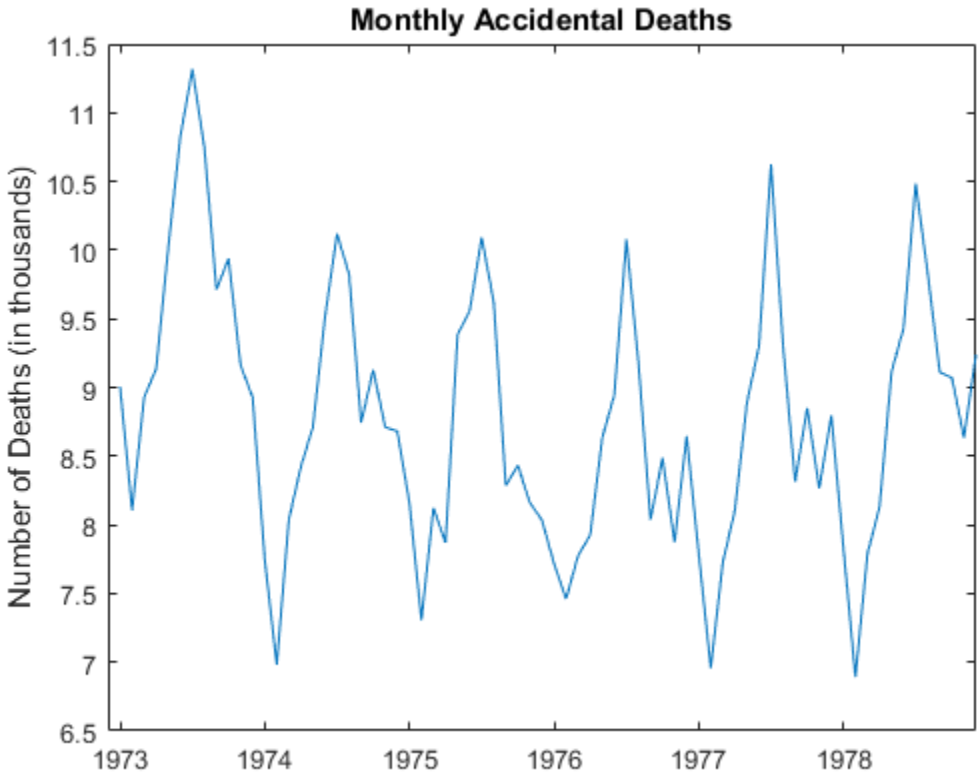
Parametric Trend Estimation

This example shows how to estimate nonseasonal and seasonal trend components using parametric models. The time series is monthly accidental deaths in the U.S. from 1973 to 1978 (Brockwell and Davis, 2002).

Step 1: Load the Data

Load the accidental deaths data set.

```
load(fullfile(matlabroot,'examples','econ','Data_Accidental.mat'))
y = Data;
T = length(y);

figure
plot(y/1000)
h1 = gca;
h1.XLim = [0,T];
h1.XTick = 1:12:T;
h1.XTickLabel = datestr(dates(1:12:T),10);
title 'Monthly Accidental Deaths';
ylabel 'Number of Deaths (in thousands)';
hold on
```



The data shows a potential quadratic trend and a strong seasonal component with periodicity 12.

Step 2: Fit Quadratic Trend

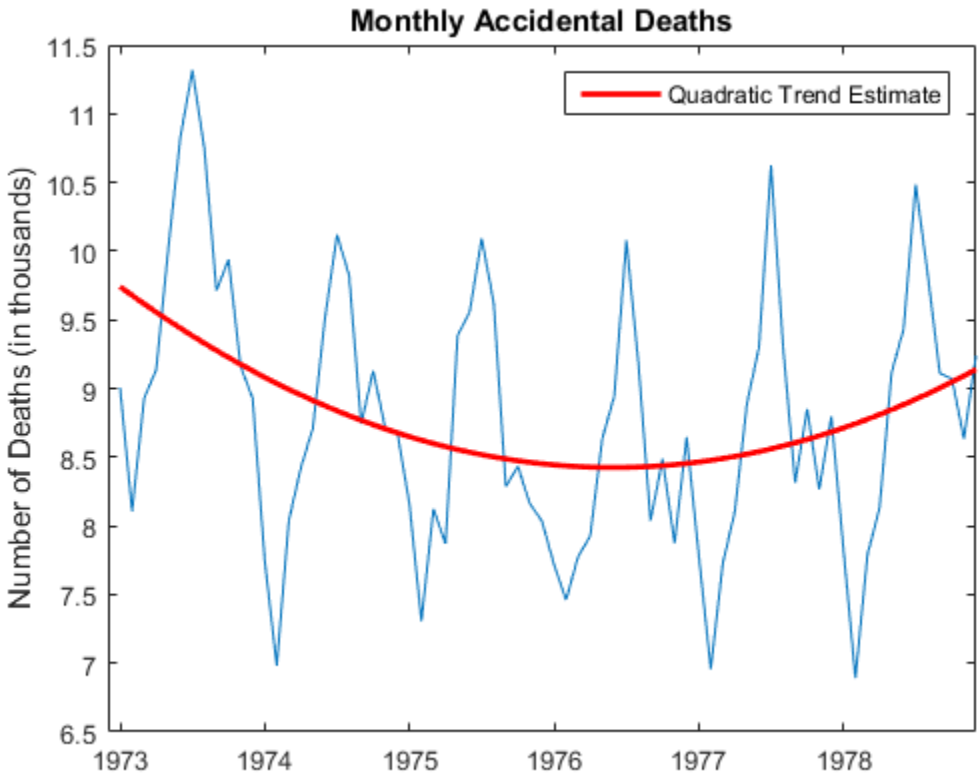
Fit the polynomial

$$T_t = \beta_0 + \beta_1 t + \beta_2 t^2$$

to the observed series.

```
t = (1:T)';
X = [ones(T,1) t t.^2];
```

```
b = X\y;
tH = X*b;

h2 = plot(tH/1000,'r','LineWidth',2);
legend(h2,'Quadratic Trend Estimate')
hold off
```



Step 3. Detrend Original Series.

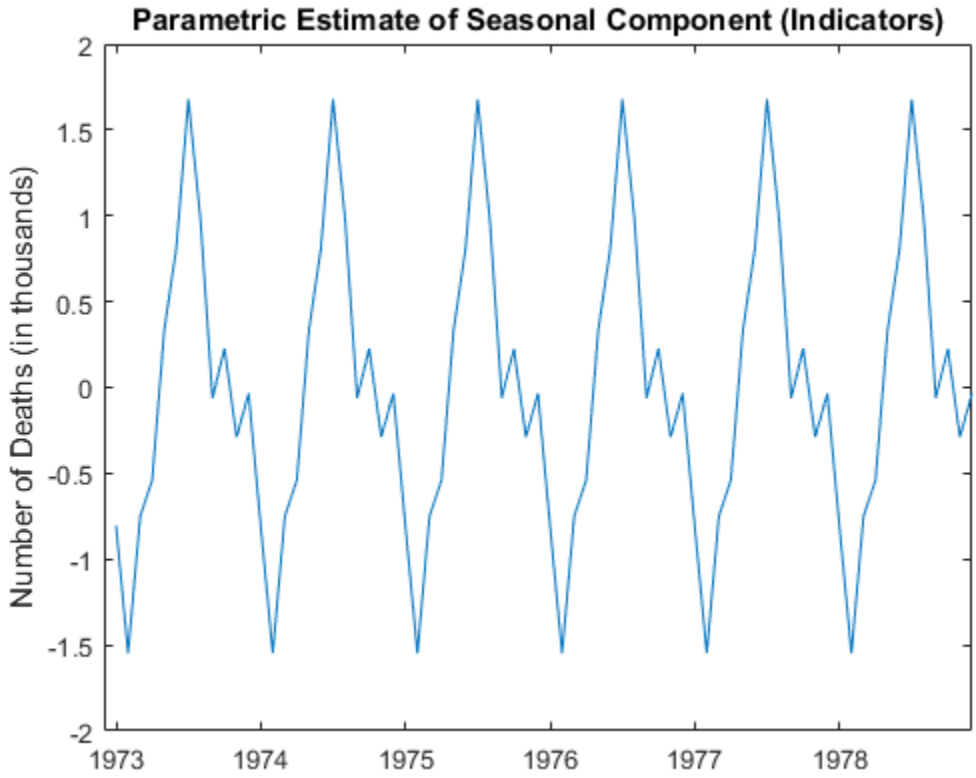
Subtract the fitted quadratic line from the original data.

```
xt = y - tH;
```

Step 4. Estimate Seasonal Indicator Variables

Create indicator (dummy) variables for each month. The first indicator is equal to one for January observations, and zero otherwise. The second indicator is equal to one for February observations, and zero otherwise. A total of 12 indicator variables are created for the 12 months. Regress the detrended series against the seasonal indicators.

```
mo = repmat((1:12)',6,1);
sX = dummyvar(mo);

bS = sX\xt;
st = sX*bS;

figure
plot(st/1000)
title 'Parametric Estimate of Seasonal Component (Indicators)';
h3 = gca;
h3.XLim = [0,T];
ylabel 'Number of Deaths (in thousands)';
h3.XTick = 1:12:T;
h3.XTickLabel = datestr(dates(1:12:T),10);
```



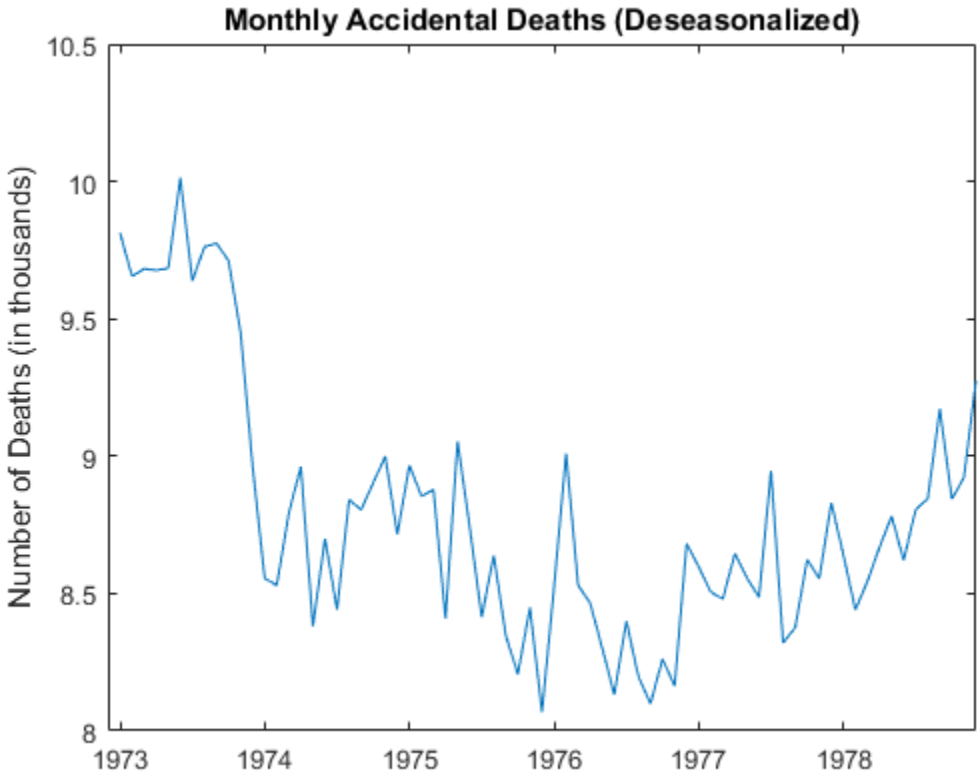
In this regression, all 12 seasonal indicators are included in the design matrix. To prevent collinearity, an intercept term is not included (alternatively, you can include 11 indicators and an intercept term).

Step 5. Deseasonalize Original Series

Subtract the estimated seasonal component from the original series.

```
dt = y - st;

figure
plot(dt/1000)
title 'Monthly Accidental Deaths (Deseasonalized)';
h4 = gca;
```

```
h4.XLim = [0,T];
ylabel 'Number of Deaths (in thousands)';
h4.XTick = 1:12:T;
h4.XTickLabel = datestr(dates(1:12:T),10);
```



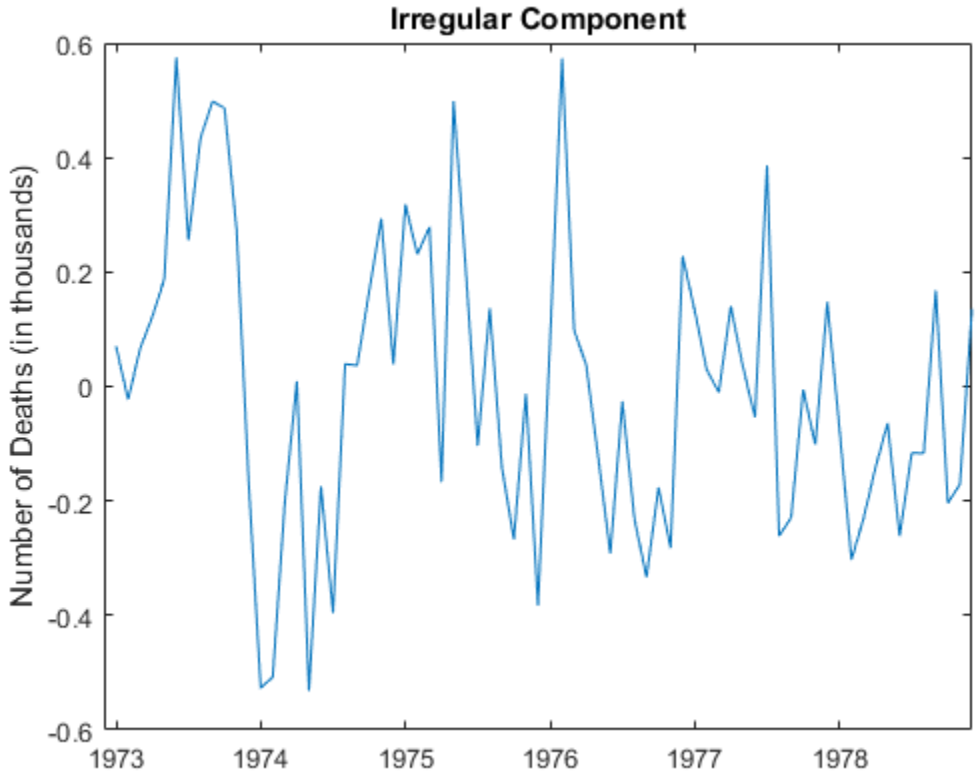
The quadratic trend is much clearer with the seasonal component removed.

Step 6. Estimate Irregular Component

Subtract the trend and seasonal estimates from the original series. The remainder is an estimate of the irregular component.

```
bt = y - tH - st;
```

```
figure
plot(bt/1000)
title('Irregular Component')
h5 = gca;
h5.XLim = [0,T];
ylabel 'Number of Deaths (in thousands)';
h5.XTick = 1:12:T;
h5.XTickLabel = datestr(dates(1:12:T),10);
```



You can optionally model the irregular component using a stochastic process model.

References:

Box, G. E. P., G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis: Forecasting and Control*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

See Also

`dummyvar`

Related Examples

- "Moving Average Trend Estimation" on page 2-33
- "Seasonal Adjustment Using a Stable Seasonal Filter" on page 2-57
- "Seasonal Adjustment Using S(n,m) Seasonal Filters" on page 2-64

More About

- "Time Series Decomposition" on page 2-28
- "Seasonal Adjustment" on page 2-54

Hodrick-Prescott Filter

The Hodrick-Prescott (HP) filter is a specialized filter for trend and business cycle estimation (no seasonal component). Suppose a time series y_t can be additively decomposed into a trend and business cycle component. Denote the trend component g_t and the cycle component c_t. Then $y_t = g_t + c_t$.

The HP filter finds a trend estimate, $\hat{g}_t$, by solving a penalized optimization problem. The smoothness of the trend estimate depends on the choice of penalty parameter. The cycle component, which is often of interest to business cycle analysts, is estimated as $\hat{c}_t = y_t - \hat{g}_t$.

`hpfilter` returns the estimated trend and cycle components of a time series.

See Also

`hpfilter`

Related Examples

- "Using the Hodrick-Prescott Filter to Reproduce Their Original Result" on page 2-46

More About

- "Moving Average Filter" on page 2-31
- "Seasonal Filters" on page 2-51
- "Time Series Decomposition" on page 2-28

Using the Hodrick-Prescott Filter to Reproduce Their Original Result

This example shows how to use the Hodrick-Prescott filter to decompose a time series.

The Hodrick-Prescott filter separates a time series into growth and cyclical components with

$$y_t = g_t + c_t$$

where yt is a time series, gt is the growth component of yt, and ct is the cyclical component of yt for t = 1, ... , T.

The objective function for the Hodrick-Prescott filter has the form

$$\sum_{t=1}^{T} c_t^2 + \lambda \sum_{t=2}^{T-1} ((g_{t+1} - g_t) - (g_t - g_{t-1}))^2$$

with a smoothing parameter lambda. The programming problem is to minimize the objective over all g1, ... , gT.

The conceptual basis for this programming problem is that the first sum minimizes the difference between the data and its growth component (which is the cyclical component) and the second sum minimizes the second-order difference of the growth component, which is analogous to minimization of the second derivative of the growth component.

Note that this filter is equivalent to a cubic spline smoother.

Use of the Hodrick-Prescott Filter to Analyze GNP Cyclicality

Using data similar to the data found in Hodrick and Prescott [1], plot the cyclical component of GNP. This result should coincide with the results in the paper. However, since the GNP data here and in the paper are both adjusted for seasonal variations with conversion from nominal to real values, differences can be expected due to differences in the sources for the pair of adjustments. Note that our data comes from the St. Louis Federal Reserve FRED database [2] which was downloaded with the Datafeed Toolbox™.

```
load Data_GNP

startdate = 1950; % date range is from 1947.0 to 2005.5 (quarterly)
```

```
enddate = 1979.25; % change these two lines to try alternative periods

startindex = find(dates == startdate);
endindex = find(dates == enddate);

gnpdates = dates(startindex:endindex);
gnpraw = log(DataTable.GNPR(startindex:endindex));
```

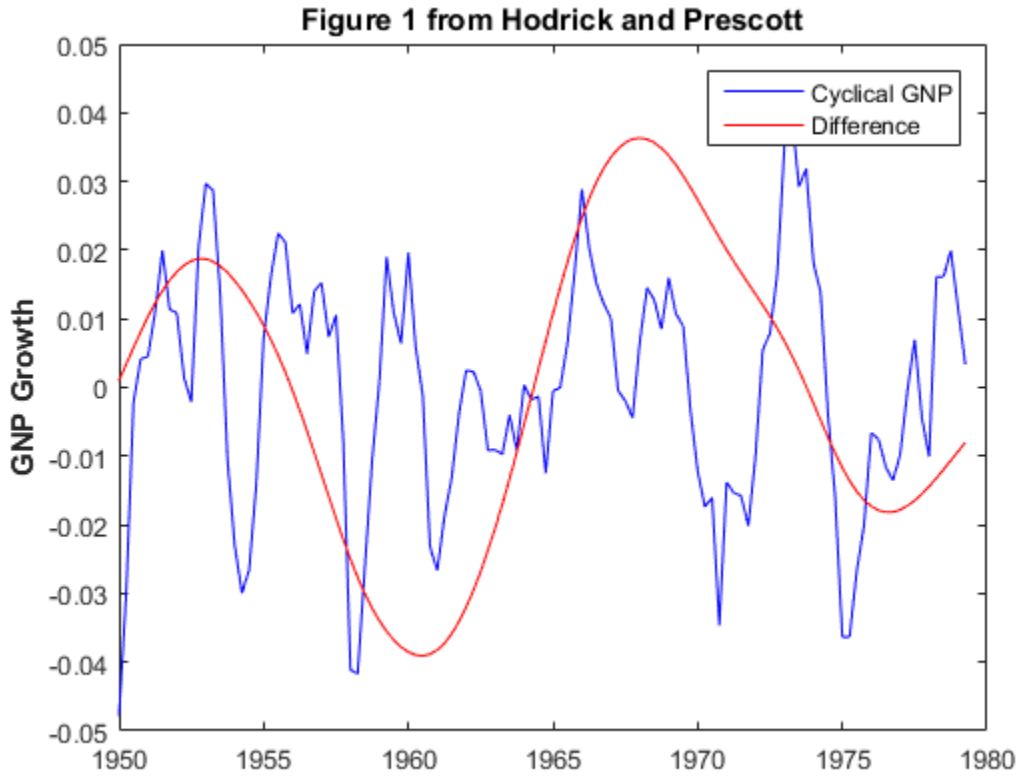
We run the filter for different smoothing parameters lambda = 400, 1600, 6400, and Inf. The infinite smoothing parameter just detrends the data.

```
[gnptrend4, gnpcycle4] = hpfilter(gnpraw,400);
[gnptrend16, gnpcycle16] = hpfilter(gnpraw,1600);
[gnptrend64, gnpcycle64] = hpfilter(gnpraw,6400);
[gnptrendinf, gnpcycleinf] = hpfilter(gnpraw,Inf);
```

Plot Cyclical GNP and Its Relationship with Long-Term Trend

The following code generates Figure 1 from Hodrick and Prescott [1].

```
plot(gnpdates,gnpcycle16,'b');
hold all
plot(gnpdates,gnpcycleinf - gnpcycle16,'r');
title('\bfFigure 1 from Hodrick and Prescott');
ylabel('\bfGNP Growth');
legend('Cyclical GNP','Difference');
hold off
```



The blue line is the cyclical component with smoothing parameter 1600 and the red line is the difference with respect to the detrended cyclical component. The difference is smooth enough to suggest that the choice of smoothing parameter is appropriate.

Statistical Tests on Cyclical GNP

We will now reconstruct Table 1 from Hodrick and Prescott [1]. With the cyclical components, we compute standard deviations, autocorrelations for lags 1 to 10, and perform a Dickey-Fuller unit root test to assess non-stationarity. As in the article, we see that as lambda increases, standard deviations increase, autocorrelations increase over longer lags, and the unit root hypothesis is rejected for all but the detrended case. Together, these results imply that any of the cyclical series with finite smoothing is effectively stationary.

```
gnpacf4 = autocorr(gnpcycle4,10);
gnpacf16 = autocorr(gnpcycle16,10);
gnpacf64 = autocorr(gnpcycle64,10);
gnpacfinf = autocorr(gnpcycleinf,10);

WarnState = warning('off','econ:adftest:LeftTailStatTooSmall');

[H4, ~, gnptest4] = adftest(gnpcycle4,'model','ARD');
[H16, ~, gnptest16] = adftest(gnpcycle16,'model','ARD');
[H64, ~, gnptest64] = adftest(gnpcycle64,'model','ARD');
[Hinf, ~, gnptestinf] = adftest(gnpcycleinf,'model','ARD');

warning(WarnState);

fprintf(1,'Table 1 from Hodrick and Prescott Reference\n');
fprintf(1,'  %10s  %s\n',' ','Smoothing Parameter');
fprintf(1,'  %10s  %10s  %10s  %10s  %10s\n',' ','400','1600','6400','Infinity');
fprintf(1,'  %-10s  %10.2f  %10.2f  %10.2f  %10.2f\n','Std. Dev.', ...
 100*std(gnpcycle4),100*std(gnpcycle16),100*std(gnpcycle64),100*std(gnpcycleinf));
fprintf(1,'  Autocorrelations\n');
for i=2:11
 fprintf(1,'  %10g  %10.2f  %10.2f  %10.2f  %10.2f\n',(i-1), ...
  gnpacf4(i),gnpacf16(i),gnpacf64(i),gnpacfinf(i))
end
fprintf(1,'  %-10s  %10.2f  %10.2f  %10.2f  %10.2f\n','Unit Root', ...
 gnptest4,gnptest16,gnptest64,gnptestinf);
fprintf(1,'  %-10s  %10d  %10d  %10d  %10d\n','Reject H0',H4,H16,H64,Hinf);
```

```
Table 1 from Hodrick and Prescott Reference
              Smoothing Parameter
                    400        1600        6400    Infinity
  Std. Dev.        1.52        1.75        2.06        3.11
  Autocorrelations
           1        0.74        0.78        0.82        0.92
           2        0.38        0.47        0.57        0.81
           3        0.05        0.17        0.33        0.70
           4       -0.21       -0.07        0.12        0.59
           5       -0.36       -0.24       -0.03        0.50
           6       -0.39       -0.30       -0.10        0.44
           7       -0.35       -0.31       -0.13        0.39
           8       -0.28       -0.29       -0.15        0.35
           9       -0.22       -0.26       -0.15        0.31
          10       -0.19       -0.25       -0.17        0.26
  Unit Root        -4.35       -4.13       -3.79       -2.28
  Reject H0            1           1           1           0
```

References

[1] Robert J. Hodrick and Edward C. Prescott, "Postwar U.S. Business Cycles: An Empirical Investigation," *Journal of Money, Credit, and Banking*, Vol. 29, No. 1, February 1997, pp. 1-16.

[2] U.S. Federal Reserve Economic Data (FRED), Federal Reserve Bank of St. Louis, http://research.stlouisfed.org/fred.

See Also

`hpfilter`

More About

- “Hodrick-Prescott Filter” on page 2-45
- “Time Series Decomposition” on page 2-28

Seasonal Filters

In this section...

What Is a Seasonal Filter?

You can use a seasonal filter (moving average) to estimate the seasonal component of a time series. For example, seasonal moving averages play a large role in the X-11-ARIMA seasonal adjustment program of Statistics Canada [1] and the X-12-ARIMA seasonal adjustment program of the U.S. Census Bureau [2].

For observations made during period k, $k = 1,...,s$ (where s is the known periodicity of the seasonality), a seasonal filter is a convolution of weights and observations made during past and future periods k. For example, given monthly data ($s = 12$), a smoothed January observation is a symmetric, weighted average of January data.

In general, for a time series x_t, $t = 1,...,N$, the seasonally smoothed observation at time $k + js$, $j = 1, ...,N/s - 1$, is

$$\tilde{s}_{k+js} = \sum_{l=-r}^{r} a_l x_{k+(j+l)s},$$

with weights a_l such that $\sum_{l=-r}^{r} a_l = 1$.

The two most commonly used seasonal filters are the stable seasonal filter and the $S_{n \times m}$ seasonal filter.

Stable Seasonal Filter

Use a stable seasonal filter if the seasonal level does not change over time, or if you have a short time series (under 5 years).

Let n_k be the total number of observations made in period k. A stable seasonal filter is given by

$$\tilde{s}_k = \frac{1}{n_k} \sum_{j=1}^{(N/s)-1} x_{k+js},$$

for $k = 1,...,s$, and $\tilde{s}_k = \tilde{s}_{k-s}$ for $k > s$.

Define $\bar{s} = (1/s)\sum_{k=1}^{s} \tilde{s}_k$. For identifiability from the trend component,

- Use $\hat{s}_k = \tilde{s}_k - \bar{s}$ to estimate the seasonal component for an additive decomposition model (that is, constrain the component to fluctuate around zero).
- Use $\hat{s}_k = \tilde{s}_k / \bar{s}$ to estimate the seasonal component for a multiplicative decomposition model (that is, constrain the component to fluctuate around one).

$S_{n \times m}$ seasonal filter

To apply an $S_{n \times m}$ seasonal filter, take a symmetric n-term moving average of m-term averages. This is equivalent to taking a symmetric, unequally weighted moving average with $n + m - 1$ terms (that is, use $r = (n + m - 1)/2$ in Equation 2-1).

An $S_{3\times3}$ filter has five terms with weights

$$(1/9, 2/9, 1/3, 2/9, 1/9).$$

To illustrate, suppose you have monthly data over 10 years. Let Jan_{yy} denote the value observed in January, 20yy. The $S_{3\times3}$-filtered value for January 2005 is

$$\hat{J}an_{05} = \frac{1}{3}\left[\frac{1}{3}(Jan_{03} + Jan_{04} + Jan_{05}) + \frac{1}{3}(Jan_{04} + Jan_{05} + Jan_{06}) + \frac{1}{3}(Jan_{05} + Jan_{06} + Jan_{07})\right].$$

Similarly, an $S_{3\times5}$ filter has seven terms with weights

$$(1/15, 2/15, 1/5, 1/5, 1/5, 2/15, 1/15).$$

When using a symmetric filter, observations are lost at the beginning and end of the series. You can apply asymmetric weights at the ends of the series to prevent observation loss.

To center the seasonal estimate, define a moving average of the seasonally filtered series, $\bar{s}_t = \sum_{j=-q}^{q} b_j \tilde{s}_{t+j}$. A reasonable choice for the weights are $b_j = 1/4q$ for $j = \pm q$ and $b_j = 1/2q$ otherwise. Here, $q = 2$ for quarterly data (a 5-term average), or $q = 6$ for monthly data (a 13-term average).

For identifiability from the trend component,

- Use $\hat{s}_t = \tilde{s}_t - \bar{s}_t$ to estimate the seasonal component of an additive model (that is, constrain the component to fluctuate approximately around zero).
- Use $\hat{s}_t = \tilde{s}_t / \bar{s}_t$ to estimate the seasonal component of a multiplicative model (that is, constrain the component to fluctuate approximately around one).

References

[1] Dagum, E. B. *The X-11-ARIMA Seasonal Adjustment Method*. Number 12–564E. Statistics Canada, Ottawa, 1980.

[2] Findley, D. F., B. C. Monsell, W. R. Bell, M. C. Otto, and B.-C. Chen. "New Capabilities and Methods of the X-12-ARIMA Seasonal-Adjustment Program." *Journal of Business & Economic Statistics*. Vol. 16, Number 2, 1998, pp. 127–152.

Related Examples

- "Seasonal Adjustment Using a Stable Seasonal Filter" on page 2-57
- "Seasonal Adjustment Using S(n,m) Seasonal Filters" on page 2-64

More About

- "Moving Average Filter" on page 2-31
- "Seasonal Adjustment" on page 2-54
- "Time Series Decomposition" on page 2-28

Seasonal Adjustment

In this section...

What Is Seasonal Adjustment?

Economists and other practitioners are sometimes interested in extracting the global trends and business cycles of a time series, free from the effect of known seasonality. Small movements in the trend can be masked by a *seasonal component*, a trend with fixed and known periodicity (e.g., monthly or quarterly). The presence of seasonality can make it difficult to compare relative changes in two or more series.

Seasonal adjustment is the process of removing a nuisance periodic component. The result of a seasonal adjustment is a *deseasonalized* time series. Deseasonalized data is useful for exploring the trend and any remaining irregular component. Because information is lost during the seasonal adjustment process, you should retain the original data for future modeling purposes.

Deseasonalized Series

Consider decomposing a time series, y_t, into three components:

- Trend component, T_t
- Seasonal component, S_t with known periodicity s
- Irregular (stationary) stochastic component, I_t

The most common decompositions are additive, multiplicative, and log-additive.

To seasonally adjust a time series, first obtain an estimate of the seasonal component, $\hat{S}_t$. The estimate $\hat{S}_t$ should be constrained to fluctuate around zero (at least approximately) for additive models, and around one, approximately, for multiplicative models. These constraints allow the seasonal component to be identifiable from the trend component.

Given $\hat{S}_t$, the deseasonalized series is calculated by subtracting (or dividing by) the estimated seasonal component, depending on the assumed decomposition.

- For an additive decomposition, the deseasonalized series is given by $d_t = y_t - \hat{S}_t$.
- For a multiplicative decomposition, the deseasonalized series is given by $d_t = y_t / \hat{S}_t$.

Seasonal Adjustment Process

To best estimate the seasonal component of a series, you should first estimate and remove the trend component. Conversely, to best estimate the trend component, you should first estimate and remove the seasonal component. Thus, seasonal adjustment is typically performed as an iterative process. The following steps for seasonal adjustment resemble those used within the X-12-ARIMA seasonal adjustment program of the U.S. Census Bureau [1].

1. Obtain a first estimate of the trend component, $\hat{T}_t$, using a moving average or parametric trend estimate.
2. Detrend the original series. For an additive decomposition, calculate $x_t = y_t - \hat{T}_t$. For a multiplicative decomposition, calculate $x_t = y_t / \hat{T}_t$.
3. Apply a seasonal filter to the detrended series, x_t, to obtain an estimate of the seasonal component, $\hat{S}_t$. Center the estimate to fluctuate around zero or one, depending on the chosen decomposition. Use an $S_{3\times3}$ seasonal filter if you have adequate data, or a stable seasonal filter otherwise.
4. Deseasonalize the original series. For an additive decomposition, calculate $d_t = y_t - \hat{S}_t$. For a multiplicative decomposition, calculate $d_t = y_t / \hat{S}_t$. .
5. Obtain a second estimate of the trend component, $\hat{T}_t$, , using the deseasonalized series d_t. Consider using a Henderson filter [1], with asymmetric weights at the ends of the series.
6. Detrend the original series again. For an additive decomposition, calculate $x_t = y_t - \hat{T}_t$. For a multiplicative decomposition, calculate $x_t = y_t / \hat{T}_t$.

7 Apply a seasonal filter to the detrended series, x_t, to obtain an estimate of the seasonal component, $\hat{S}_t$. Consider using an $S_{3\times5}$ seasonal filter if you have adequate data, or a stable seasonal filter otherwise.

8 Deseasonalize the original series. For an additive decomposition, calculate $d_t = y_t - \hat{S}_t$. For a multiplicative decomposition, calculate $d_t = y_t / \hat{S}_t$. This is the final deseasonalized series.

References

[1] Findley, D. F., B. C. Monsell, W. R. Bell, M. C. Otto, and B.-C. Chen. "New Capabilities and Methods of the X-12-ARIMA Seasonal-Adjustment Program." *Journal of Business & Economic Statistics*. Vol. 16, Number 2, 1998, pp. 127–152.

Related Examples

- "Moving Average Trend Estimation" on page 2-33
- "Seasonal Adjustment Using a Stable Seasonal Filter" on page 2-57
- "Seasonal Adjustment Using S(n,m) Seasonal Filters" on page 2-64
- "Parametric Trend Estimation" on page 2-37

More About

- "Time Series Decomposition" on page 2-28
- "Seasonal Filters" on page 2-51
- "Moving Average Filter" on page 2-31

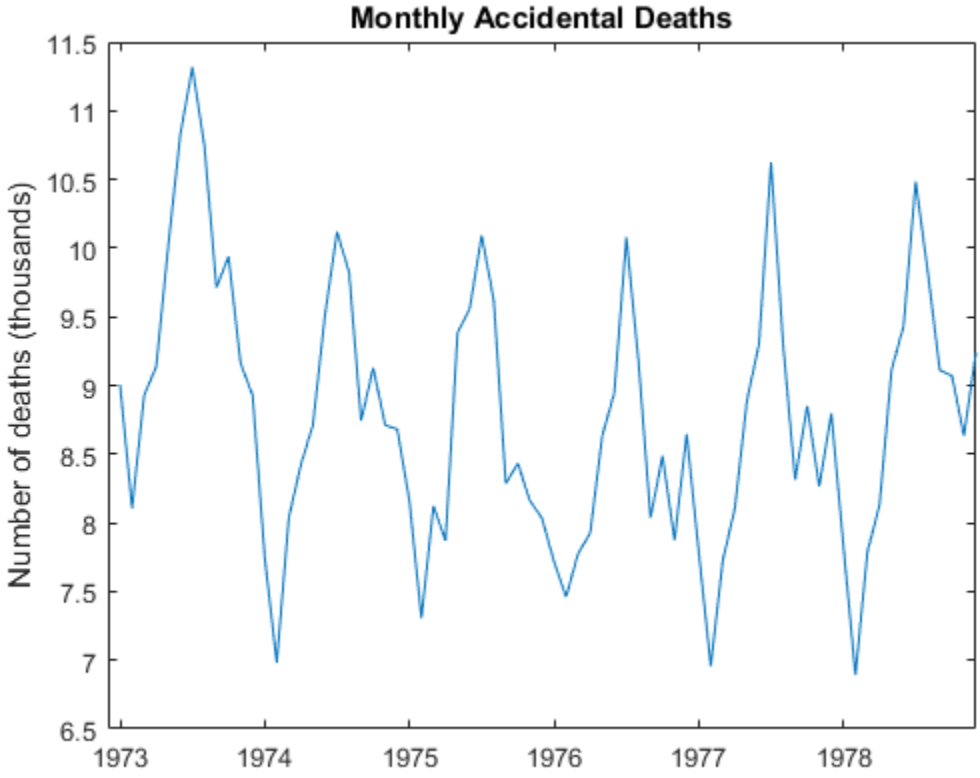
Seasonal Adjustment Using a Stable Seasonal Filter

This example shows how to use a stable seasonal filter to deseasonalize a time series (using an additive decomposition). The time series is monthly accidental deaths in the U.S. from 1973 to 1978 (Brockwell and Davis, 2002).

Load the data.

Load the accidental deaths data set.

```
load(fullfile(matlabroot,'examples','econ','Data_Accidental.mat'))
y = Data;
T = length(y);

figure
plot(y/1000)
h1 = gca;
h1.XLim = [0,T];
h1.XTick = 1:12:T;
h1.XTickLabel = datestr(dates(1:12:T),10);
title 'Monthly Accidental Deaths';
ylabel 'Number of deaths (thousands)';
hold on
```



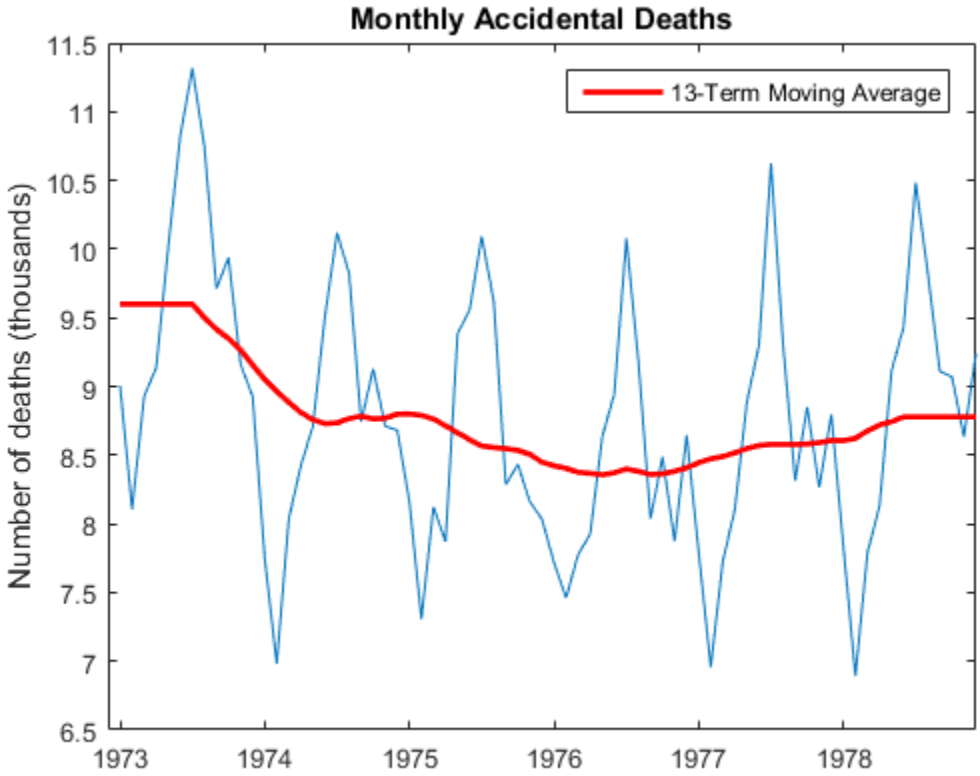
The data exhibits a strong seasonal component with periodicity 12.

Apply a 13-term moving average.

Smooth the data using a 13-term moving average. To prevent observation loss, repeat the first and last smoothed values six times. Subtract the smoothed series from the original series to detrend the data. Add the moving average trend estimate to the observed time series plot.

```
sW13 = [1/24;repmat(1/12,11,1);1/24];
yS = conv(y,sW13,'same');
yS(1:6) = yS(7); yS(T-5:T) = yS(T-6);

xt = y-yS;
```

```
h = plot(yS/1000,'r','LineWidth',2);
legend(h,'13-Term Moving Average')
hold off
```



The detrended time series is `xt`.

Using the shape parameter `'same'` when calling `conv` returns a smoothed series the same length as the original series.

Step 3. Create seasonal indices.

Create a cell array, `sidx`, to store the indices corresponding to each period. The data is monthly, with periodicity 12, so the first element of `sidx` is a vector with elements 1, 13,

25,...,61 (corresponding to January observations). The second element of `sidx` is a vector with elements 2, 14, 16,...,62 (corresponding to February observations). This is repeated for all 12 months.

```
s = 12;
sidx = cell(s,1);
for i = 1:s
 sidx{i,1} = i:s:T;
end

sidx{1:2}


ans =

     1    13    25    37    49    61


ans =

     2    14    26    38    50    62
```

Using a cell array to store the indices allows for the possibility that each period does not occur the same number of times within the span of the observed series.

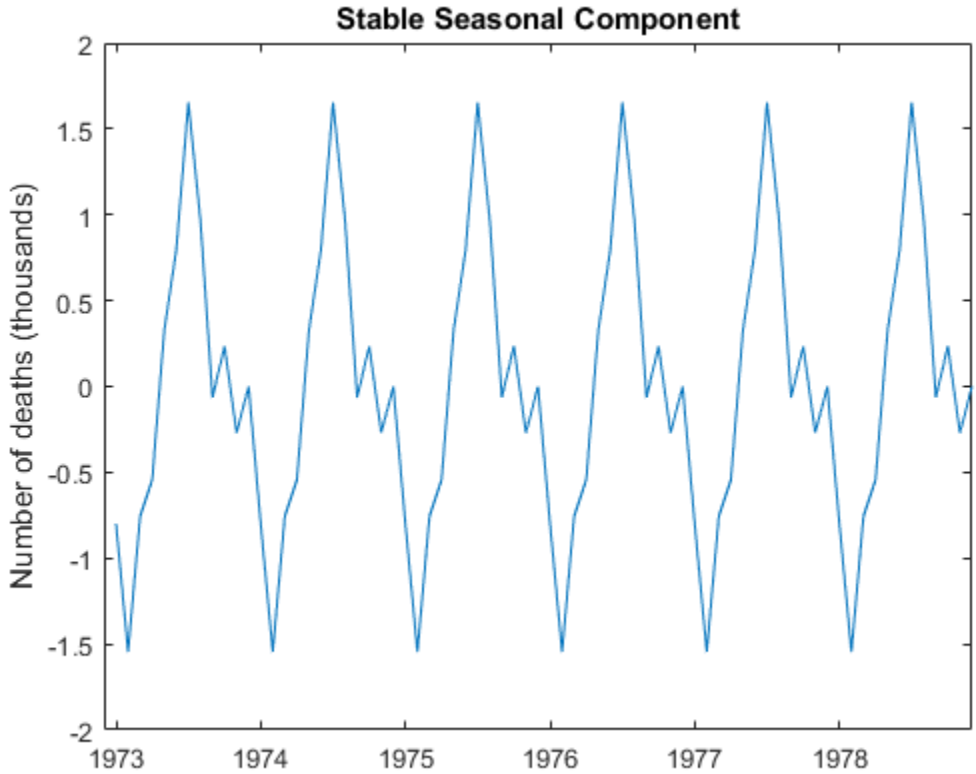
Step 4. Apply a stable seasonal filter.

Apply a stable seasonal filter to the detrended series, `xt`. Using the indices constructed in Step 3, average the detrended data corresponding to each period. That is, average all of the January values (at indices 1, 13, 25,...,61), and then average all of the February values (at indices 2, 14, 26,...,62), and so on for the remaining months. Put the smoothed values back into a single vector.

Center the seasonal estimate to fluctuate around zero.

```
sst = cellfun(@(x) mean(xt(x)),sidx);

% Put smoothed values back into a vector of length N
nc = floor(T/s); % no. complete years
rm = mod(T,s); % no. extra months
sst = [repmat(sst,nc,1);sst(1:rm)];

% Center the seasonal estimate (additive)
```

```
sBar = mean(sst); % for centering
sst = sst-sBar;

figure
plot(sst/1000)
title 'Stable Seasonal Component';
h2 = gca;
h2.XLim = [0 T];
ylabel 'Number of deaths (thousands)';
h2.XTick = 1:12:T;
h2.XTickLabel = datestr(dates(1:12:T),10);
```

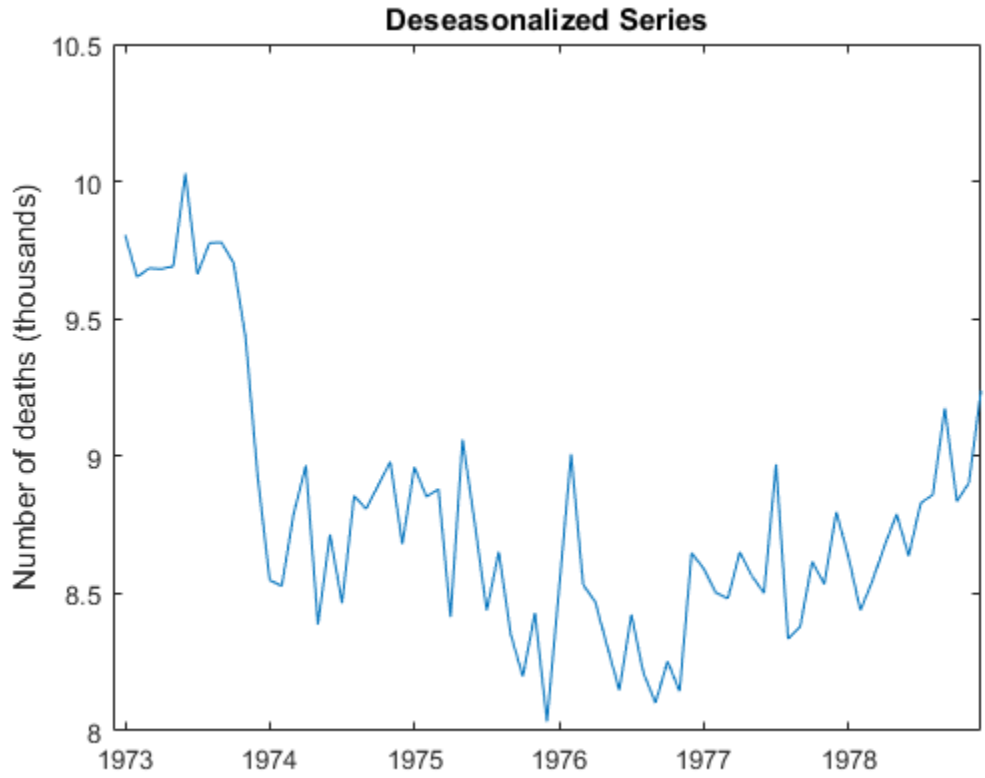


The stable seasonal component has constant amplitude across the series. The seasonal estimate is centered, and fluctuates around zero.

Step 5. Deseasonalize the series.

Subtract the estimated seasonal component from the original data.

```
dt = y - sst;

figure
plot(dt/1000)
title 'Deseasonalized Series';
ylabel 'Number of deaths (thousands)';
h3 = gca;
h3.XLim = [0 T];
h3.XTick = 1:12:T;
h3.XTickLabel = datestr(dates(1:12:T),10);
```



The deseasonalized series consists of the long-term trend and irregular components. A large-scale quadratic trend in the number of accidental deaths is clear with the seasonal component removed.

References:

Brockwell, P. J. and R. A. Davis. *Introduction to Time Series and Forecasting*. 2nd ed. New York, NY: Springer, 2002.

See Also

`cellfun` | `conv`

Related Examples

- "Moving Average Trend Estimation" on page 2-33
- "Seasonal Adjustment Using S(n,m) Seasonal Filters" on page 2-64
- "Parametric Trend Estimation" on page 2-37

More About

- "Time Series Decomposition" on page 2-28
- "Moving Average Filter" on page 2-31
- "Seasonal Filters" on page 2-51
- "Seasonal Adjustment" on page 2-54

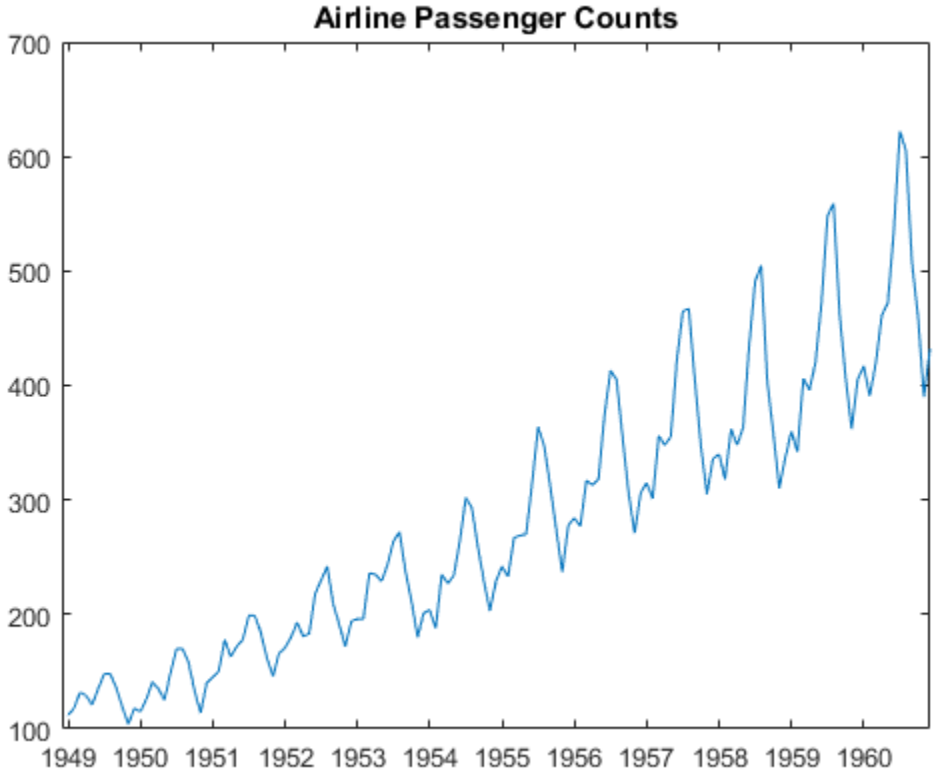
Seasonal Adjustment Using S(n,m) Seasonal Filters

This example shows how to apply $S_{n\times m}$ seasonal filters to deseasonalize a time series (using a multiplicative decomposition). The time series is monthly international airline passenger counts from 1949 to 1960.

Load the Data

Load the airline data set.

```
load(fullfile(matlabroot,'examples','econ','Data_Airline.mat'))
y = Data;
T = length(y);

figure
plot(y)
h1 = gca;
h1.XLim = [0,T];
h1.XTick = 1:12:T;
h1.XTickLabel = datestr(dates(1:12:T),10);
title 'Airline Passenger Counts';
hold on
```



The data shows an upward linear trend and a seasonal component with periodicity 12.

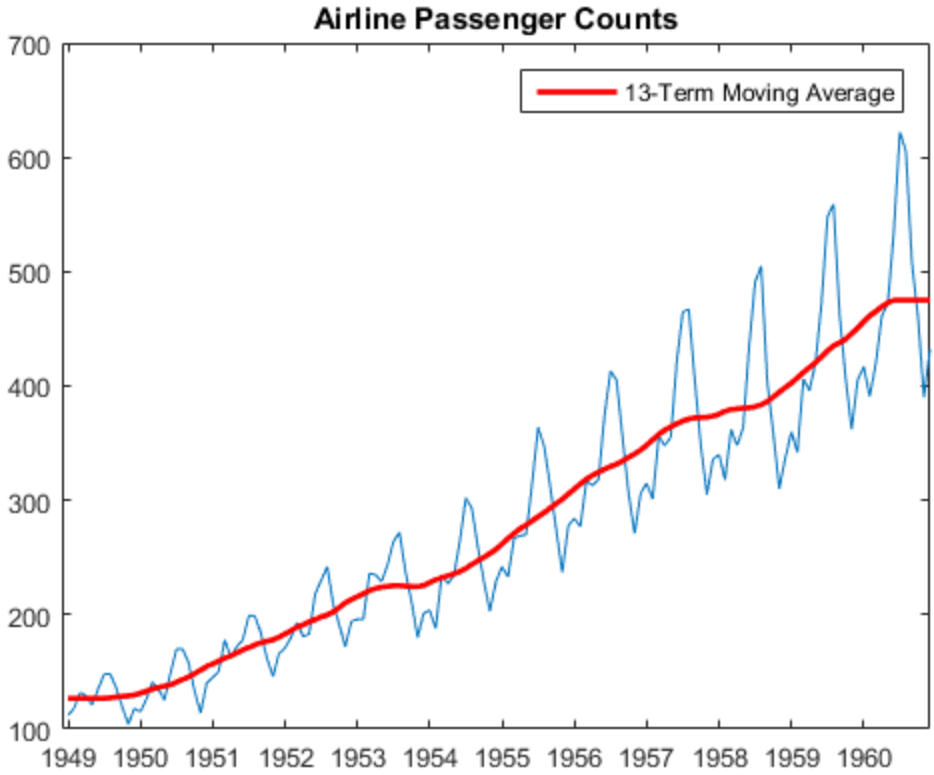
Detrend the data using a 13-term moving average.

Before estimating the seasonal component, estimate and remove the linear trend. Apply a 13-term symmetric moving average, repeating the first and last observations six times to prevent data loss. Use weight 1/24 for the first and last terms in the moving average, and weight 1/12 for all interior terms.

Divide the original series by the smoothed series to detrend the data. Add the moving average trend estimate to the observed time series plot.

```
sW13 = [1/24;repmat(1/12,11,1);1/24];
yS = conv(y,sW13,'same');
```

```
yS(1:6) = yS(7); yS(T-5:T) = yS(T-6);

xt = y./yS;

h = plot(yS,'r','LineWidth',2);
legend(h,'13-Term Moving Average')
hold off
```



Create seasonal indices.

Create a cell array, `sidx`, to store the indices corresponding to each period. The data is monthly, with periodicity 12, so the first element of `sidx` is a vector with elements 1, 13, 25,...,133 (corresponding to January observations). The second element of `sidx` is a

vector with elements 2, 14, 16,...,134 (corresponding to February observations). This is repeated for all 12 months.

```
s = 12;
sidx = cell(s,1); % Preallocation

for i = 1:s
 sidx{i,1} = i:s:T;
end

sidx{1:2}
```

```
ans =

     1    13    25    37    49    61    73    85    97   109   121   133


ans =

     2    14    26    38    50    62    74    86    98   110   122   134
```

Using a cell array to store the indices allows for the possibility that each period does not occur the same number of times within the span of the observed series.

Apply an S(3,3) filter.

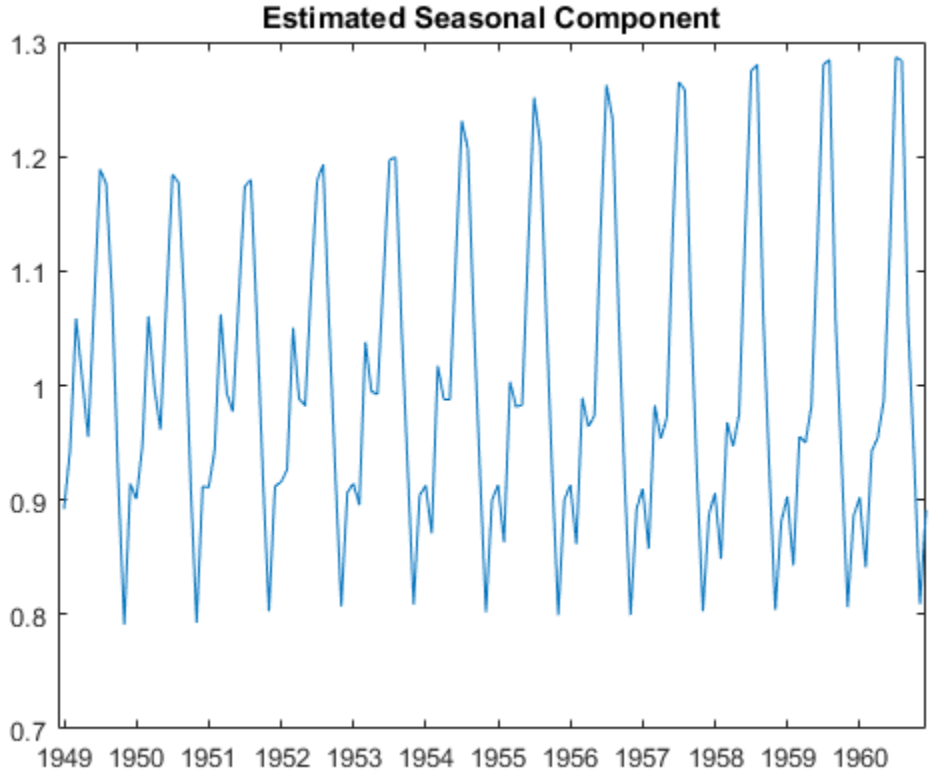
Apply a 5-term $S_{3\times3}$ seasonal moving average to the detrended series `xt`. That is, apply a moving average to the January values (at indices 1, 13, 25,...,133), and then apply a moving average to the February series (at indices 2, 14, 26,...,134), and so on for the remaining months.

Use asymmetric weights at the ends of the moving average (using `conv2`). Put the smoothed values back into a single vector.

To center the seasonal component around one, estimate, and then divide by, a 13-term moving average of the estimated seasonal component.

```
% S3x3 seasonal filter
% Symmetric weights
sW3 = [1/9;2/9;1/3;2/9;1/9];
% Asymmetric weights for end of series
aW3 = [.259 .407;.37 .407;.259 .185;.111 0];
```

```
% Apply filter to each month
shat = NaN*y;
for i = 1:s
    ns = length(sidx{i});
    first = 1:4;
    last = ns - 3:ns;
    dat = xt(sidx{i});

    sd = conv(dat,sW3,'same');
    sd(1:2) = conv2(dat(first),1,rot90(aW3,2),'valid');
    sd(ns  -1:ns) = conv2(dat(last),1,aW3,'valid');
    shat(sidx{i}) = sd;
end

% 13-term moving average of filtered series
sW13 = [1/24;repmat(1/12,11,1);1/24];
sb = conv(shat,sW13,'same');
sb(1:6) = sb(s+1:s+6);
sb(T-5:T) = sb(T-s-5:T-s);

% Center to get final estimate
s33 = shat./sb;

figure
plot(s33)
h2 = gca;
h2.XLim = [0,T];
h2.XTick = 1:12:T;
h2.XTickLabel = datestr(dates(1:12:T),10);
title 'Estimated Seasonal Component';
```



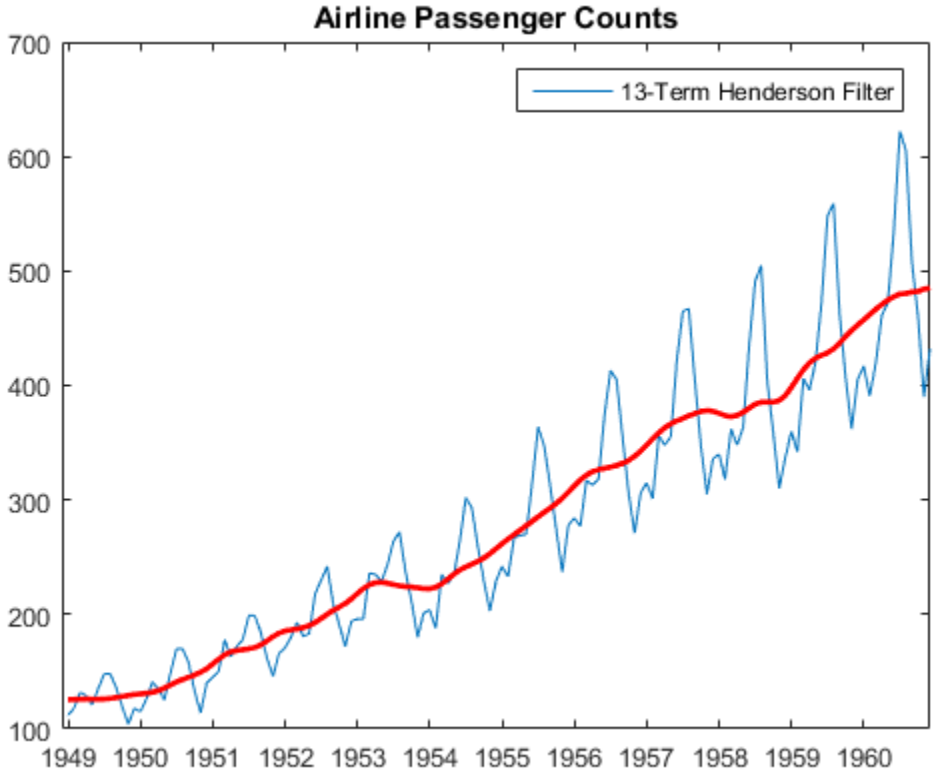
Notice that the seasonal level changes over the range of the data. This illustrates the difference between an $S_{n\times m}$ seasonal filter and a stable seasonal filter. A stable seasonal filter assumes that the seasonal level is constant over the range of the data.

Apply a 13-term Henderson filter.

To get an improved estimate of the trend component, apply a 13-term Henderson filter to the seasonally adjusted series. The necessary symmetric and asymmetric weights are provided in the following code.

```
% Deseasonalize series
dt = y./s33;

% Henderson filter weights
```

```
sWH = [-0.019;-0.028;0;.066;.147;.214;
      .24;.214;.147;.066;0;-0.028;-0.019];
% Asymmetric weights for end of series
aWH = [-.034  -.017   .045   .148   .279   .421;
       -.005   .051   .130   .215   .292   .353;
        .061   .135   .201   .241   .254   .244;
        .144   .205   .230   .216   .174   .120;
        .211   .233   .208   .149   .080   .012;
        .238   .210   .144   .068   .002  -.058;
        .213   .146   .066   .003  -.039  -.092;
        .147   .066   .004  -.025  -.042  0    ;
        .066   .003  -.020  -.016  0      0    ;
        .001  -.022  -.008  0      0      0    ;
       -.026  -.011   0     0      0      0    ;
       -.016   0      0     0      0      0    ];

% Apply 13-term Henderson filter
first = 1:12;
last = T-11:T;
h13 = conv(dt,sWH,'same');
h13(T-5:end) = conv2(dt(last),1,aWH,'valid');
h13(1:6) = conv2(dt(first),1,rot90(aWH,2),'valid');

% New detrended series
xt = y./h13;

figure
plot(y)
h3 = gca;
h3.XLim = [0,T];
h3.XTick = 1:12:T;
h3.XTickLabel = datestr(dates(1:12:T),10);
title 'Airline Passenger Counts';
hold on
plot(h13,'r','LineWidth',2);
legend('13-Term Henderson Filter')
hold off
```



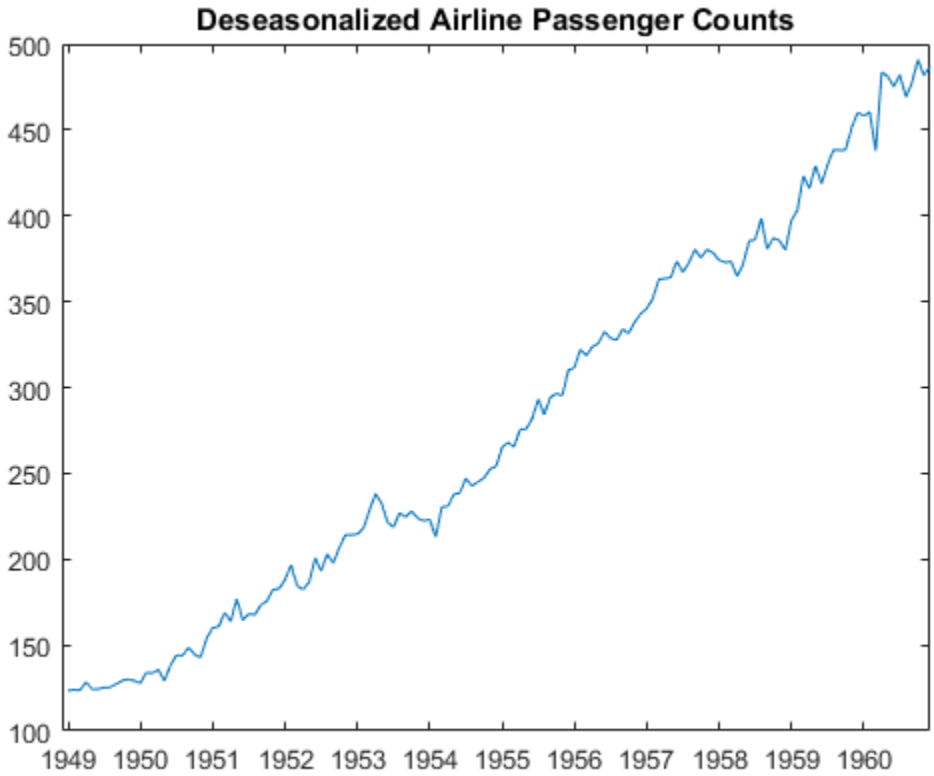
Apply an S(3,5) seasonal filter.

To get 6. an improved estimate of the seasonal component, apply a 7-term $S_{3\times5}$ seasonal moving average to the newly detrended series. The symmetric and asymmetric weights are provided in the following code. Center the seasonal estimate to fluctuate around 1.

Deseasonalize the original series by dividing it by the centered seasonal estimate.

```
% S3x5 seasonal filter
% Symmetric weights
sW5 = [1/15;2/15;repmat(1/5,3,1);2/15;1/15];
% Asymmetric weights for end of series
aW5 = [.150 .250 .293;
       .217 .250 .283;
```

```
        .217 .250 .283;
        .217 .183 .150;
        .133 .067    0;
        .067   0     0];

% Apply filter to each month
shat = NaN*y;
for i = 1:s
    ns = length(sidx{i});
    first = 1:6;
    last = ns-5:ns;
    dat = xt(sidx{i});

    sd = conv(dat,sW5,'same');
    sd(1:3) = conv2(dat(first),1,rot90(aW5,2),'valid');
    sd(ns-2:ns) = conv2(dat(last),1,aW5,'valid');
    shat(sidx{i}) = sd;
end

% 13-term moving average of filtered series
sW13 = [1/24;repmat(1/12,11,1);1/24];
sb = conv(shat,sW13,'same');
sb(1:6) = sb(s+1:s+6);
sb(T-5:T) = sb(T-s-5:T-s);


% Center to get final estimate
s35 = shat./sb;

% Deseasonalized series
dt = y./s35;

figure
plot(dt)
h4 = gca;
h4.XLim = [0,T];
h4.XTick = 1:12:T;
h4.XTickLabel = datestr(dates(1:12:T),10);
title 'Deseasonalized Airline Passenger Counts';
```



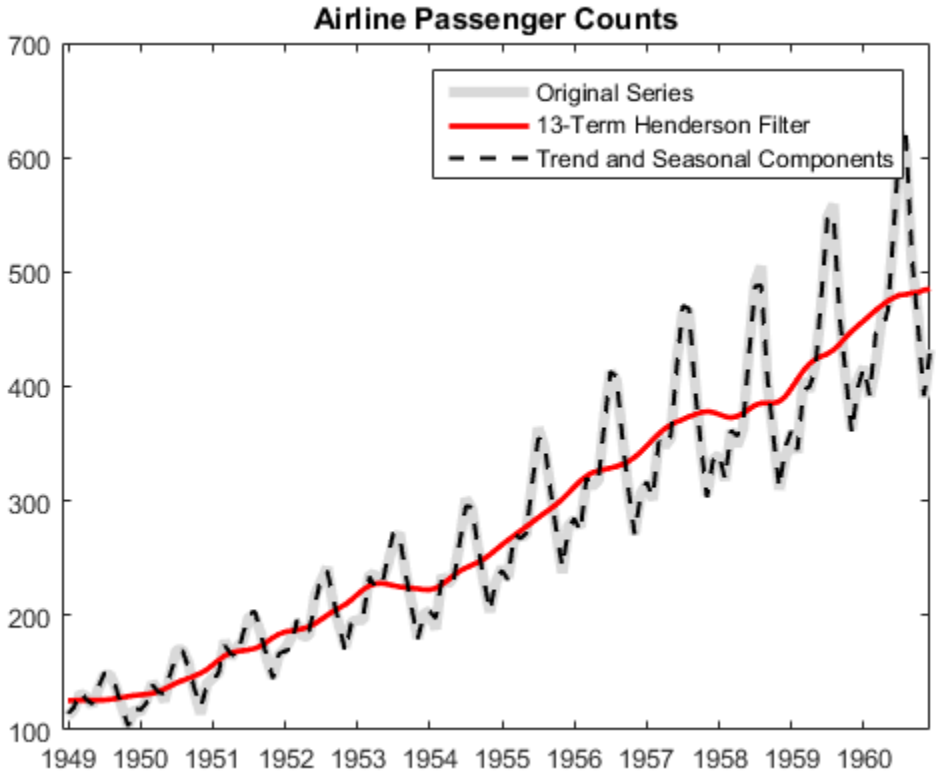
The deseasonalized series consists of the long-term trend and irregular components. With the seasonal component removed, it is easier to see turning points in the trend.

Plot the components and the original series.

Compare the original series to a series reconstructed using the component estimates.

```
figure
plot(y,'Color',[.85,.85,.85],'LineWidth',4)
h5 = gca;
h5.XLim = [0,T];
h5.XTick = 1:12:T;
h5.XTickLabel = datestr(dates(1:12:T),10);
title 'Airline Passenger Counts';
```

```
hold on
plot(h13,'r','LineWidth',2)
plot(h13.*s35,'k--','LineWidth',1.5)
legend('Original Series','13-Term Henderson Filter',...
       'Trend and Seasonal Components')
hold off
```

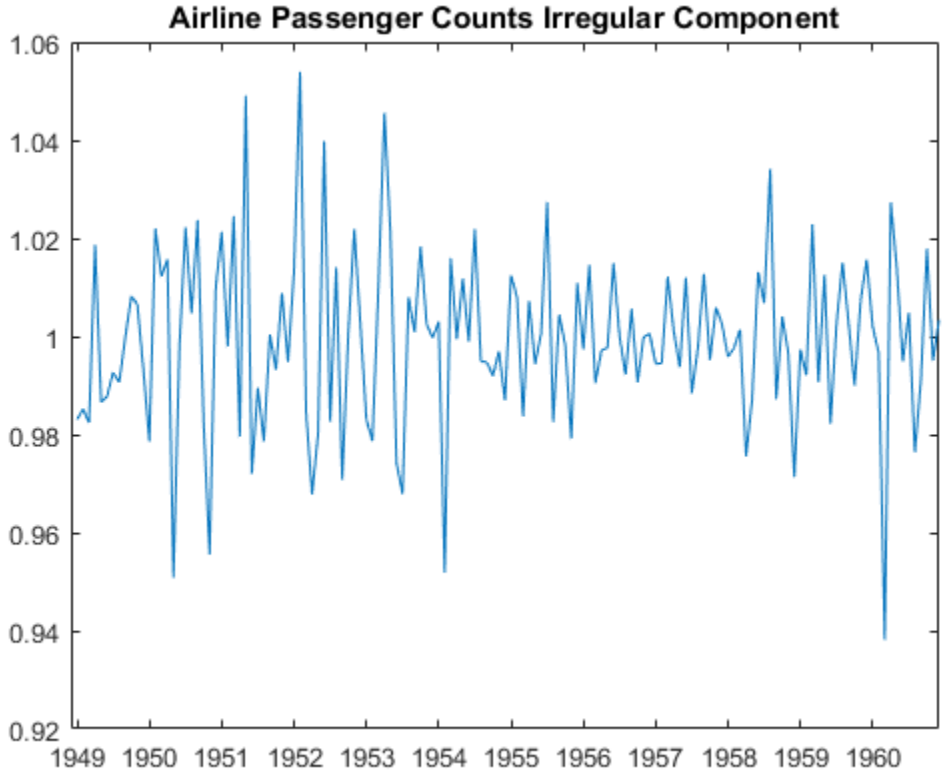


Estimate the irregular component.

Detrend and deseasonalize the original series. Plot the remaining estimate of the irregular component.

```
Irr = dt./h13;
```

```
figure
plot(Irr)
h6 = gca;
h6.XLim = [0,T];
h6.XTick = 1:12:T;
h6.XTickLabel = datestr(dates(1:12:T),10);
title 'Airline Passenger Counts Irregular Component';
```



You can optionally model the detrended and deseasonalized series using a stationary stochastic process model.

See Also

`cellfun` | `conv` | `conv2`

Related Examples

- “Moving Average Trend Estimation” on page 2-33
- “Seasonal Adjustment Using a Stable Seasonal Filter” on page 2-57
- “Parametric Trend Estimation” on page 2-37

More About

- “Time Series Decomposition” on page 2-28
- “Moving Average Filter” on page 2-31
- “Seasonal Filters” on page 2-51
- “Seasonal Adjustment” on page 2-54

Model Selection

Box-Jenkins Methodology

The Box-Jenkins methodology [1] is a five-step process for identifying, selecting, and assessing conditional mean models (for discrete, univariate time series data).

1. Establish the stationarity of your time series. If your series is not stationary, successively difference your series to attain stationarity. The sample autocorrelation function (ACF) and partial autocorrelation function (PACF) of a stationary series decay exponentially (or cut off completely after a few lags).
2. Identify a (stationary) conditional mean model for your data. The sample ACF and PACF functions can help with this selection. For an autoregressive (AR) process, the sample ACF decays gradually, but the sample PACF cuts off after a few lags. Conversely, for a moving average (MA) process, the sample ACF cuts off after a few lags, but the sample PACF decays gradually. If both the ACF and PACF decay gradually, consider an ARMA model.
3. Specify the model, and estimate the model parameters. When fitting nonstationary models in Econometrics Toolbox, it is not necessary to manually difference your data and fit a stationary model. Instead, use your data on the original scale, and create an arima model object with the desired degree of nonseasonal and seasonal differencing. Fitting an ARIMA model directly is advantageous for forecasting: forecasts are returned on the original scale (not differenced).
4. Conduct goodness-of-fit checks to ensure the model describes your data adequately. Residuals should be uncorrelated, homoscedastic, and normally distributed with constant mean and variance. If the residuals are not normally distributed, you can change your innovation distribution to a Student's t.
5. After choosing a model—and checking its fit and forecasting ability—you can use the model to forecast or generate Monte Carlo simulations over a future time horizon.

References

[1] Box, G. E. P., G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis: Forecasting and Control*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

See Also

arima | `autocorr` | `parcorr`

Related Examples

- "Box-Jenkins Model Selection" on page 3-4

- "Check Predictive Performance" on page 3-92
- "Check Fit of Multiplicative ARIMA Model" on page 3-81
- "Box-Jenkins Differencing vs. ARIMA Estimation" on page 5-94

More About

- "Trend-Stationary vs. Difference-Stationary Processes" on page 2-7
- "Autocorrelation and Partial Autocorrelation" on page 3-13
- "Goodness of Fit" on page 3-88
- "Conditional Mean Models" on page 5-3

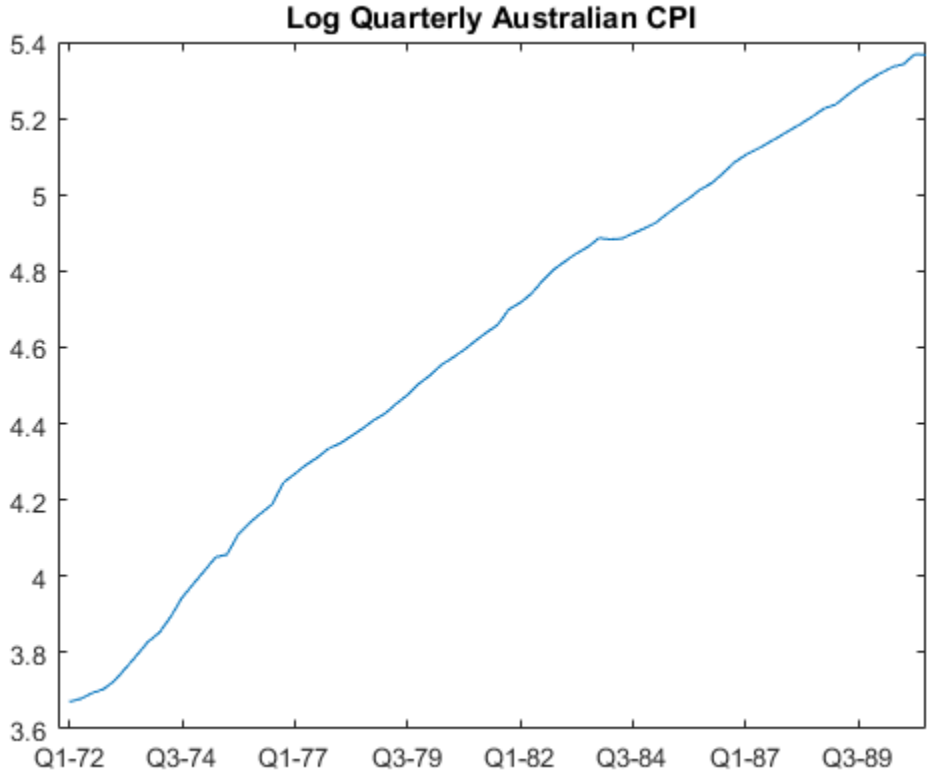
Box-Jenkins Model Selection

This example shows how to use the Box-Jenkins methodology to select an ARIMA model. The time series is the log quarterly Australian Consumer Price Index (CPI) measured from 1972 and 1991.

Load the Data

Load and plot the Australian CPI data.

```
load Data_JAustralian
y = DataTable.PAU;
T = length(y);

figure
plot(y)
h1 = gca;
h1.XLim = [0,T];
h1.XTick = 1:10:T;
h1.XTickLabel = datestr(dates(1:10:T),17);
title('Log Quarterly Australian CPI')
```

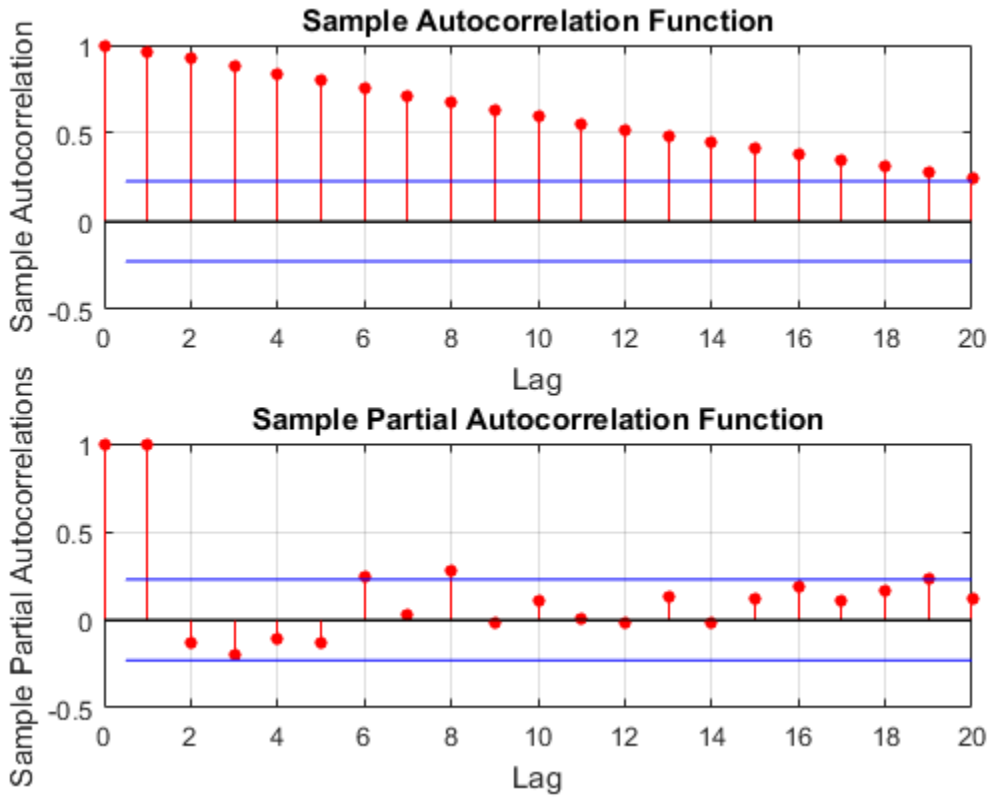


The series is nonstationary, with a clear upward trend.

Plot the Sample ACF and PACF

Plot the sample autocorrelation function (ACF) and partial autocorrelation function (PACF) for the CPI series.

```
figure
subplot(2,1,1)
autocorr(y)
subplot(2,1,2)
parcorr(y)
```



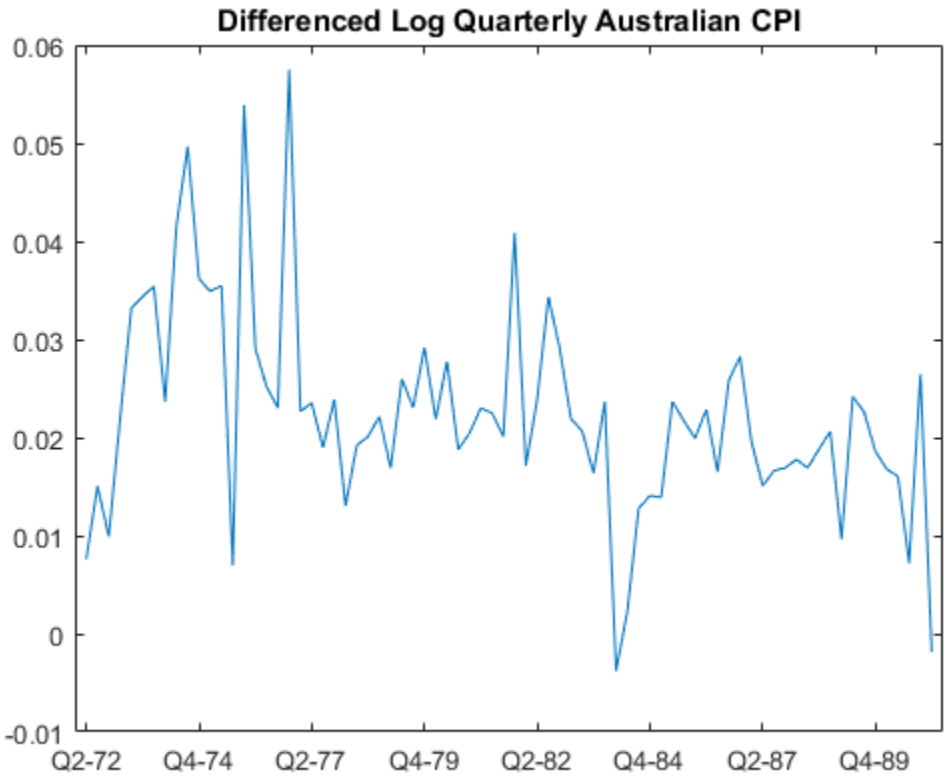
The significant, linearly decaying sample ACF indicates a nonstationary process.

Difference the Data

Take a first difference of the data, and plot the differenced series.

```
dY = diff(y);

figure
plot(dY)
h2 = gca;
h2.XLim = [0,T];
h2.XTick = 1:10:T;
h2.XTickLabel = datestr(dates(2:10:T),17);
```

```
title('Differenced Log Quarterly Australian CPI')
```



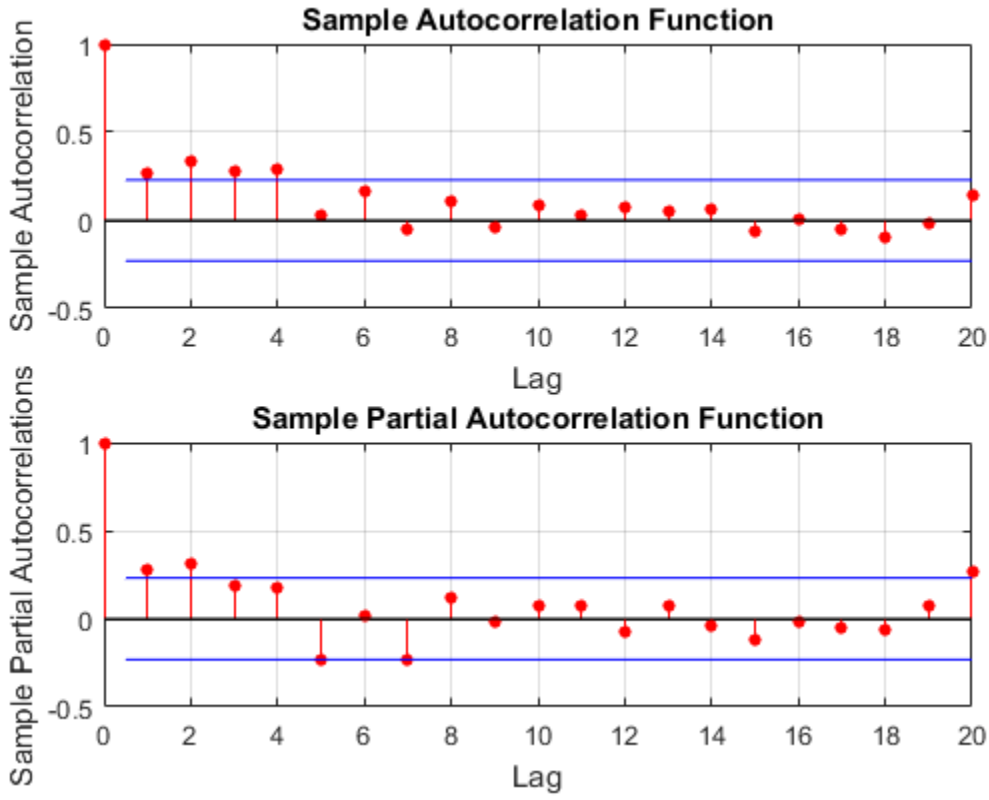
Differencing removes the linear trend. The differenced series appears more stationary.

Plot the Sample ACF and PACF of the Differenced Series

Plot the sample ACF and PACF of the differenced series to look for behavior more consistent with a stationary process.

```
figure
subplot(2,1,1)
autocorr(dY)
subplot(2,1,2)
```

```
parcorr(dY)
```



The sample ACF of the differenced series decays more quickly. The sample PACF cuts off after lag 2. This behavior is consistent with a second-degree autoregressive (AR(2)) model.

Specify and Estimate an ARIMA(2,1,0) Model

Specify, and then estimate, an ARIMA(2,1,0) model for the log quarterly Australian CPI. This model has one degree of nonseasonal differencing and two AR lags. By default, the innovation distribution is Gaussian with a constant variance.

```
Mdl = arima(2,1,0);
```

```
EstMdl = estimate(Mdl,y);
```

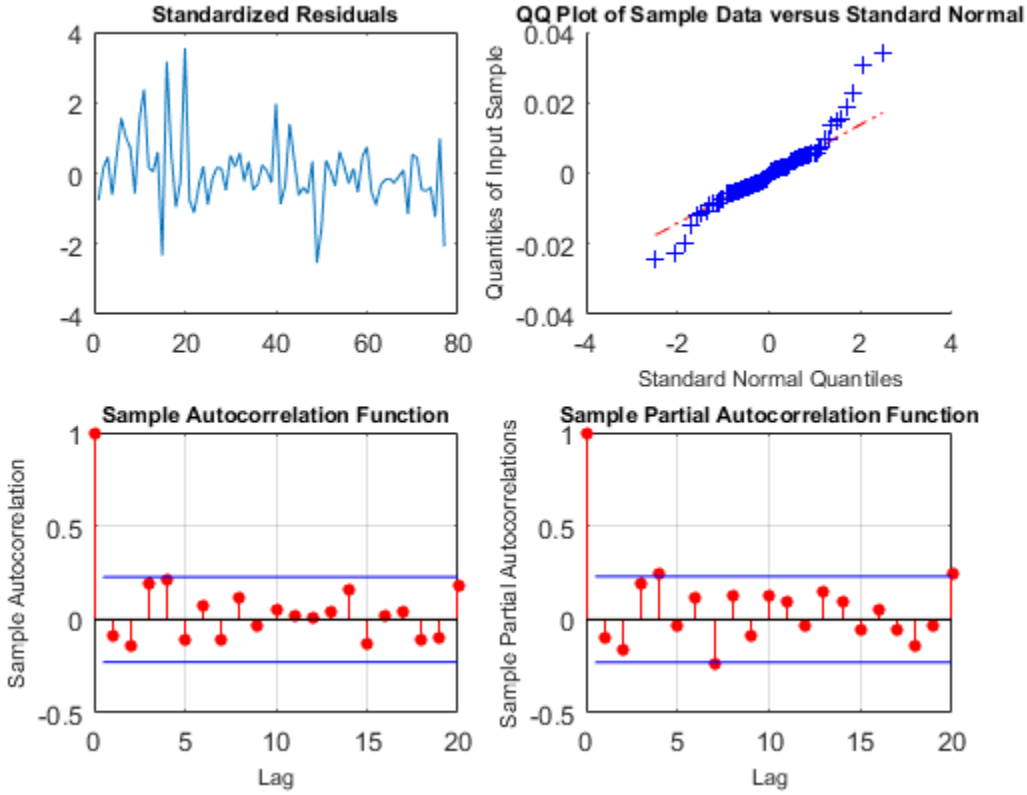
```
    ARIMA(2,1,0) Model:
    --------------------
    Conditional Probability Distribution: Gaussian

                                Standard          t
     Parameter       Value        Error       Statistic
    -----------   -----------   ------------   -----------
     Constant      0.0100723    0.00328015        3.07069
        AR{1}       0.212059     0.0954278        2.22219
        AR{2}       0.337282      0.103781        3.24994
     Variance    9.23017e-05   1.11119e-05        8.30659
```

Both AR coefficients are significant at the 0.05 significance level.

Check Goodness of Fit

Infer the residuals from the fitted model. Check that the residuals are normally distributed and uncorrelated.

```
res = infer(EstMdl,y);

figure
subplot(2,2,1)
plot(res./sqrt(EstMdl.Variance))
title('Standardized Residuals')
subplot(2,2,2)
qqplot(res)
subplot(2,2,3)
autocorr(res)
subplot(2,2,4)
parcorr(res)

hvec = findall(gcf,'Type','axes');
set(hvec,'TitleFontSizeMultiplier',0.8,...
    'LabelFontSizeMultiplier',0.8);
```



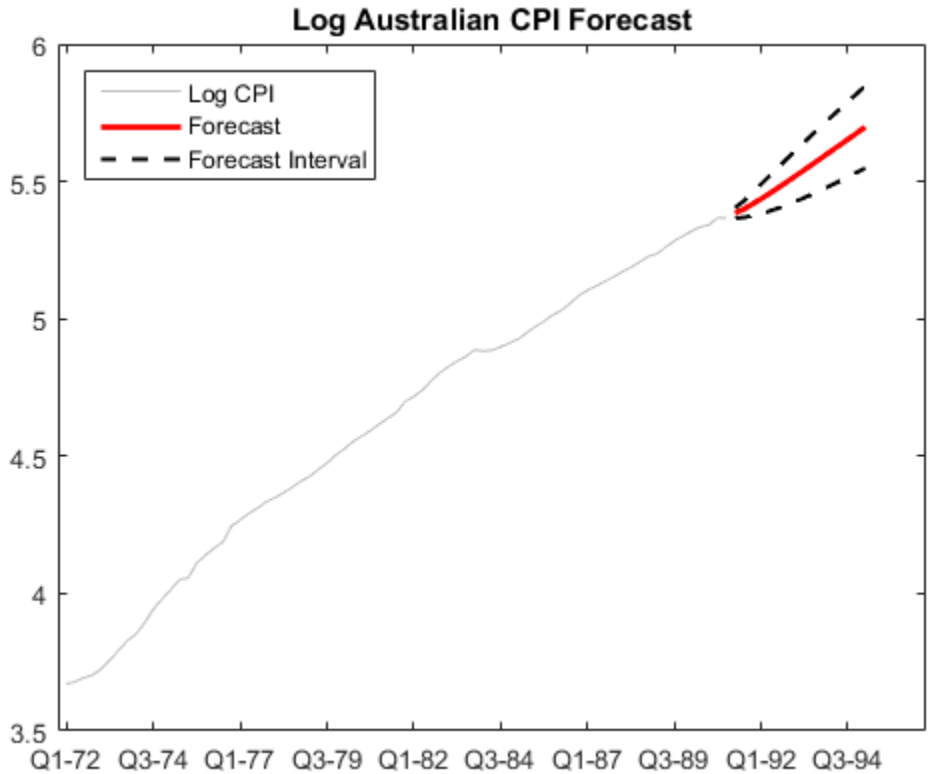
The residuals are reasonably normally distributed and uncorrelated.

Generate Forecasts

Generate forecasts and approximate 95% forecast intervals for the next 4 years (16 quarters).

```
[yF,yMSE] = forecast(EstMdl,16,'Y0',y);
UB = yF + 1.96*sqrt(yMSE);
LB = yF - 1.96*sqrt(yMSE);

figure
h4 = plot(y,'Color',[.75,.75,.75]);
hold on
```

```
h5 = plot(78:93,yF,'r','LineWidth',2);
h6 = plot(78:93,UB,'k--','LineWidth',1.5);
plot(78:93,LB,'k--','LineWidth',1.5);
fDates = [dates; dates(T) + cumsum(diff(dates(T-16:T)))];
h7 = gca;
h7.XTick = 1:10:(T+16);
h7.XTickLabel = datestr(fDates(1:10:end),17);
legend([h4,h5,h6],'Log CPI','Forecast',...
       'Forecast Interval','Location','Northwest')
title('Log Australian CPI Forecast')
hold off
```



References:

Box, G. E. P., G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis: Forecasting and Control*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

See Also

arima | `autocorr` | estimate | forecast | infer | `parcorr`

Related Examples

- "Box-Jenkins Differencing vs. ARIMA Estimation" on page 5-94
- "Nonseasonal Differencing" on page 2-18
- "Infer Residuals for Diagnostic Checking" on page 5-140
- "Specify Conditional Mean Models Using arima" on page 5-6

More About

- "Box-Jenkins Methodology" on page 3-2
- "Trend-Stationary vs. Difference-Stationary Processes" on page 2-7
- "Goodness of Fit" on page 3-88
- "MMSE Forecasting of Conditional Mean Models" on page 5-182

Autocorrelation and Partial Autocorrelation

In this section...

What Are Autocorrelation and Partial Autocorrelation?

Autocorrelation is the linear dependence of a variable with itself at two points in time. For stationary processes, autocorrelation between any two observations only depends on the time lag h between them. Define $Cov(y_t, y_{t-h}) = \gamma_h$. Lag-$h$ autocorrelation is given by

$$\rho_h = Corr(y_t, y_{t-h}) = \frac{\gamma_h}{\gamma_0}.$$

The denominator γ_0 is the lag 0 covariance, i.e., the unconditional variance of the process.

Correlation between two variables can result from a mutual linear dependence on other variables (confounding). *Partial autocorrelation* is the autocorrelation between y_t and y_{t-h} after removing any linear dependence on $y_1, y_2, \ldots, y_{t-h+1}$. The partial lag-$h$ autocorrelation is denoted $\phi_{h,h}$.

Theoretical ACF and PACF

The autocorrelation function (ACF) for a time series y_t, $t = 1,\ldots,N$, is the sequence ρ_h, $h = 1, 2,\ldots,N-1$. The partial autocorrelation function (PACF) is the sequence $\phi_{h,h}$, $h = 1, 2,\ldots,N-1$.

The theoretical ACF and PACF for the AR, MA, and ARMA conditional mean models are known, and quite different for each model. The differences in ACF and PACF among models are useful when selecting models. The following summarizes the ACF and PACF behavior for these models.

Conditional Mean Model	ACF	PACF
AR(p)	Tails off gradually	Cuts off after p lags
MA(q)	Cuts off after q lags	Tails off gradually
ARMA(p,q)	Tails off gradually	Tails off gradually

Sample ACF and PACF

Sample autocorrelation and sample partial autocorrelation are statistics that estimate the theoretical autocorrelation and partial autocorrelation. As a qualitative model selection tool, you can compare the sample ACF and PACF of your data against known theoretical autocorrelation functions [1].

For an observed series $y_1, y_2,...,y_T$, denote the sample mean $\bar{y}$. The sample lag-h autocorrelation is given by

$$\hat{\rho}_h = \frac{\sum_{t=h+1}^{T}(y_t - \bar{y})(y_{t-h} - \bar{y})}{\sum_{t=1}^{T}(y_t - \bar{y})^2}.$$

The standard error for testing the significance of a single lag-h autocorrelation, $\hat{\rho}_h$, is approximately

$$SE_\rho = \sqrt{(1 + 2\sum_{i=1}^{h-1}\rho_i^2)/N}.$$

When you use `autocorr` to plot the sample autocorrelation function (also known as the correlogram), approximate 95% confidence intervals are drawn at $\pm 2SE\rho$ by default. Optional input arguments let you modify the calculation of the confidence bounds.

The sample lag-h partial autocorrelation is the estimated lag-h coefficient in an AR model containing h lags, $\hat{\phi}_{h,h}$. The standard error for testing the significance of a single lag-h partial autocorrelation is approximately $1/\sqrt{N-1}$. When you use `parcorr` to plot the sample partial autocorrelation function, approximate 95% confidence intervals are

drawn at $\pm 2/\sqrt{N-1}$ by default. Optional input arguments let you modify the calculation of the confidence bounds.

References

[1] Box, G. E. P., G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis: Forecasting and Control*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

See Also

`autocorr` | `parcorr`

Related Examples

- "Detect Autocorrelation" on page 3-18
- "Detect ARCH Effects" on page 3-28
- "Box-Jenkins Model Selection" on page 3-4

More About

- "Ljung-Box Q-Test" on page 3-16
- "Autoregressive Model" on page 5-18
- "Moving Average Model" on page 5-27
- "Autoregressive Moving Average Model" on page 5-34

Ljung-Box Q-Test

The sample autocorrelation function (ACF) and partial autocorrelation function (PACF) are useful qualitative tools to assess the presence of autocorrelation at individual lags. The Ljung-Box Q-test is a more quantitative way to test for autocorrelation at multiple lags *jointly* [1]. The null hypothesis for this test is that the first m autocorrelations are jointly zero,

$$H_0 : \rho_1 = \rho_2 = \ldots = \rho_m = 0.$$

The choice of m affects test performance. If N is the length of your observed time series, choosing $m \approx \ln(N)$ is recommended for power [2]. You can test at multiple values of m. If seasonal autocorrelation is possible, you might consider testing at larger values of m, such as 10 or 15.

The Ljung-Box test statistic is given by

$$Q(m) = N(N+2)\sum_{h=1}^{m} \frac{\rho_h^2}{N-h}.$$

This is a modification of the Box-Pierce Portmanteau "Q" statistic [3]. Under the null hypothesis, Q(m) follows a χ^2_m distribution.

You can use the Ljung-Box Q-test to assess autocorrelation in any series with a constant mean. This includes residual series, which can be tested for autocorrelation during model diagnostic checks. If the residuals result from fitting a model with g parameters, you should compare the test statistic to a χ^2 distribution with $m - g$ degrees of freedom. Optional input arguments to `lbqtest` let you modify the degrees of freedom of the null distribution.

You can also test for conditional heteroscedasticity by conducting a Ljung-Box Q-test on a squared residual series. An alternative test for conditional heteroscedasticity is Engle's ARCH test (`archtest`).

References

[1] Ljung, G. and G. E. P. Box. "On a Measure of Lack of Fit in Time Series Models." *Biometrika*. Vol. 66, 1978, pp. 67–72.

[2] Tsay, R. S. *Analysis of Financial Time Series*. 3rd ed. Hoboken, NJ: John Wiley & Sons, Inc., 2010.

[3] Box, G. E. P. and D. Pierce. "Distribution of Residual Autocorrelations in Autoregressive-Integrated Moving Average Time Series Models." *Journal of the American Statistical Association*. Vol. 65, 1970, pp. 1509–1526.

See Also

`archtest` | `lbqtest`

Related Examples

- "Detect Autocorrelation" on page 3-18
- "Detect ARCH Effects" on page 3-28

More About

- "Autocorrelation and Partial Autocorrelation" on page 3-13
- "Engle's ARCH Test" on page 3-25
- "Residual Diagnostics" on page 3-90
- "Conditional Mean Models" on page 5-3

Detect Autocorrelation

In this section...

Compute Sample ACF and PACF

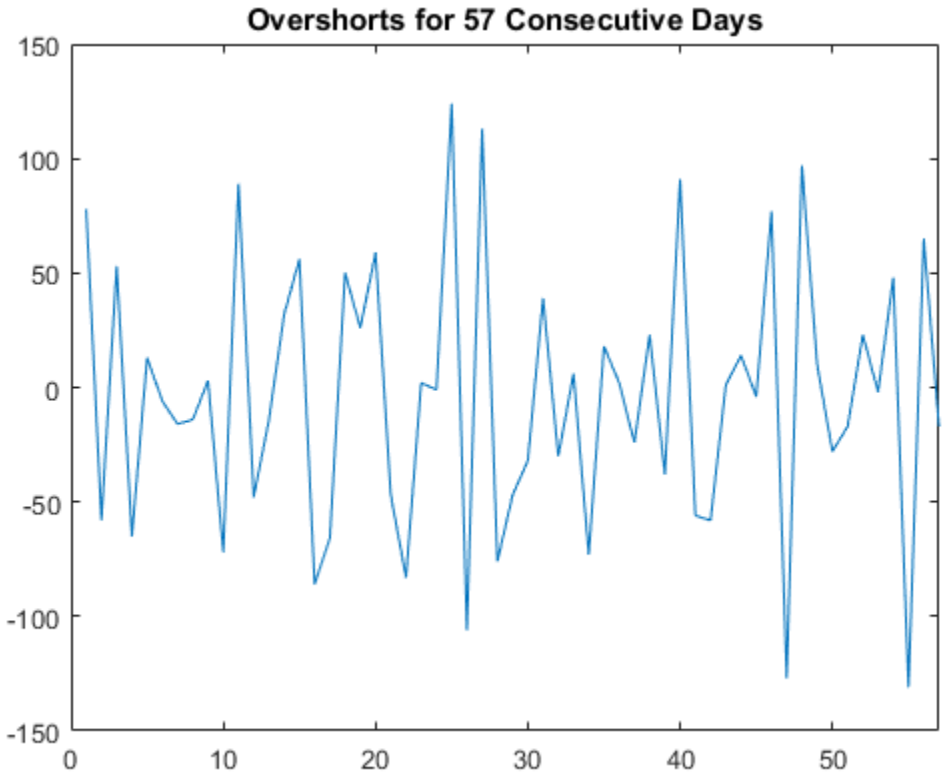
This example shows how to compute the sample autocorrelation function (ACF) and partial autocorrelation function (PACF) to qualitatively assess autocorrelation.

The time series is 57 consecutive days of overshorts from a gasoline tank in Colorado.

Step 1. Load the data.

Load the time series of overshorts.

```
load(fullfile(matlabroot,'examples','econ','Data_Overshort.mat'))
Y = Data;
N = length(Y);

figure
plot(Y)
xlim([0,N])
title('Overshorts for 57 Consecutive Days')
```

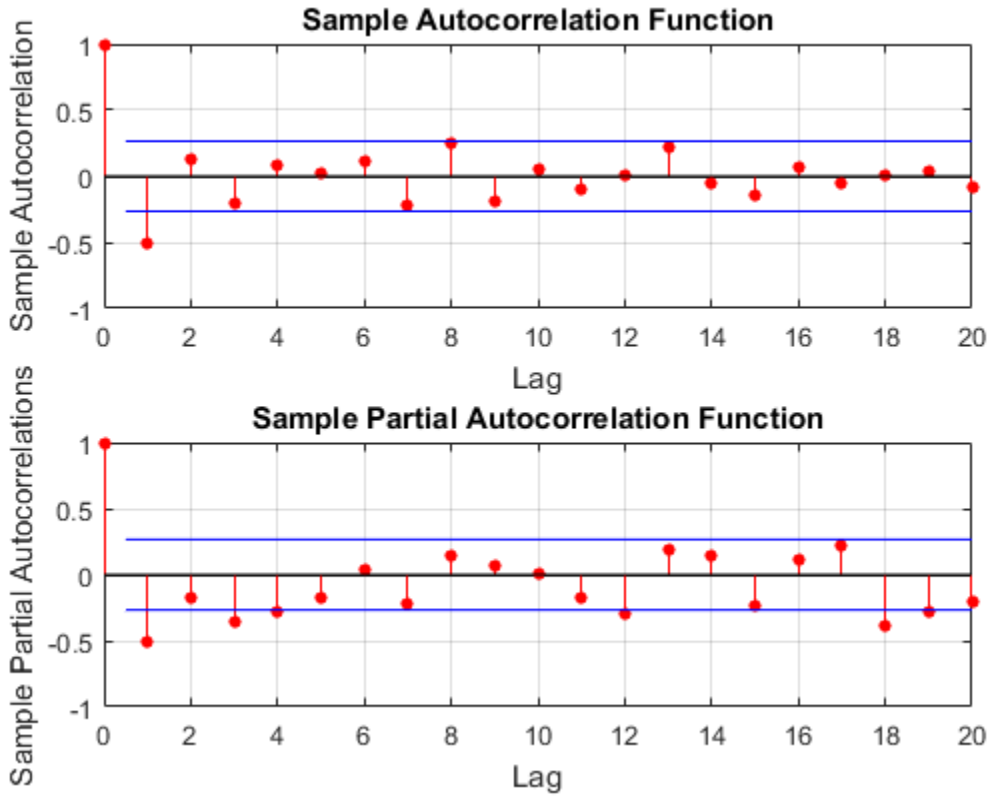


The series appears to be stationary.

Step 2. Plot the sample ACF and PACF.

Plot the sample autocorrelation function (ACF) and partial autocorrelation function (PACF).

```
figure
subplot(2,1,1)
autocorr(Y)
subplot(2,1,2)
parcorr(Y)
```



The sample ACF and PACF exhibit significant autocorrelation. The sample ACF has significant autocorrelation at lag 1. The sample PACF has significant autocorrelation at lags 1, 3, and 4.

The distinct cutoff of the ACF combined with the more gradual decay of the PACF suggests an MA(1) model might be appropriate for this data.

Step 3. Store the sample ACF and PACF values.

Store the sample ACF and PACF values up to lag 15.

```
acf = autocorr(Y,15);
pacf = parcorr(Y,15);
[length(acf) length(pacf)]
```

```
ans =

    16    16
```

The outputs `acf` and `pacf` are vectors storing the sample autocorrelation and partial autocorrelation at lags 0, 1,...,15 (a total of 16 lags).

Conduct the Ljung-Box Q-Test

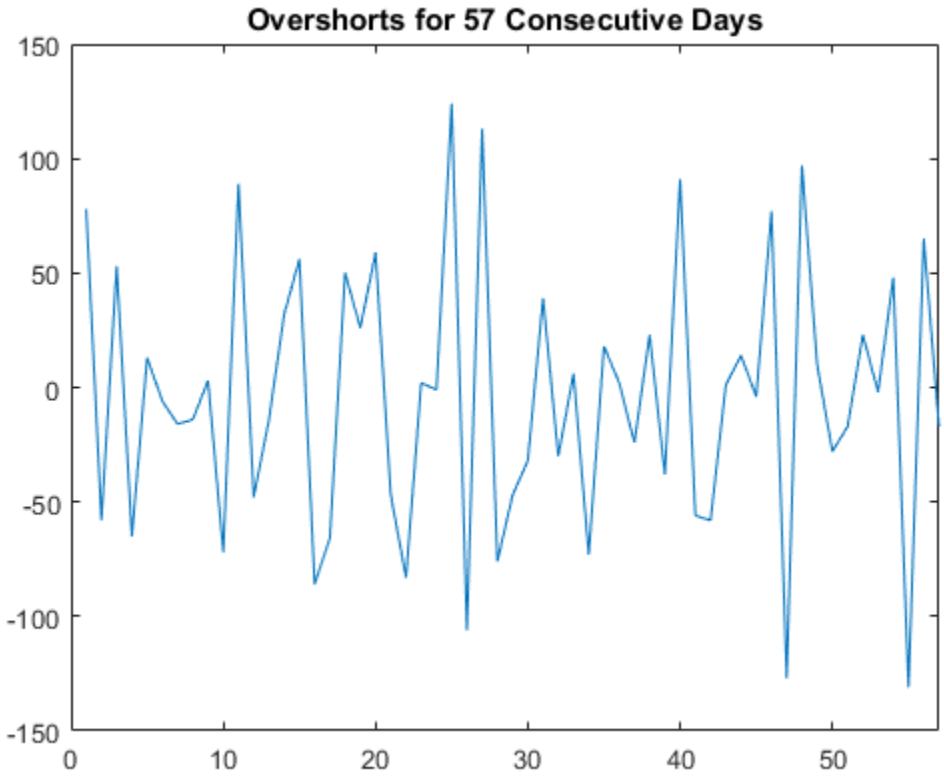
This example shows how to conduct the Ljung-Box Q-test for autocorrelation.

The time series is 57 consecutive days of overshorts from a gasoline tank in Colorado.

Step 1. Load the data.

Load the time series of overshorts.

```
load(fullfile(matlabroot,'examples','econ','Data_Overshort.mat'))
Y = Data;
N = length(Y);

figure
plot(Y)
xlim([0,N])
title('Overshorts for 57 Consecutive Days')
```



The data appears to fluctuate around a constant mean, so no data transformations are needed before conducting the Ljung-Box Q-test.

Step 2. Conduct the Ljung-Box Q-test.

Conduct the Ljung-Box Q-test for autocorrelation at lags 5, 10, and 15.

```
[h,p,Qstat,crit] = lbqtest(Y,'Lags',[5,10,15])
```

```
h =

     1     1     1
```

```
p =

    0.0016    0.0007    0.0013

Qstat =

   19.3604   30.5986   36.9639

crit =

   11.0705   18.3070   24.9958
```

All outputs are vectors with three elements, corresponding to tests at each of the three lags. The first element of each output corresponds to the test at lag 5, the second element corresponds to the test at lag 10, and the third element corresponds to the test at lag 15.

The test decisions are stored in the vector `h`. The value `h = 1` means reject the null hypothesis. Vector `p` contains the p-values for the three tests. At the $\alpha = 0.05$ significance level, the null hypothesis of no autocorrelation is rejected at all three lags. The conclusion is that there is significant autocorrelation in the series.

The test statistics and χ^2 critical values are given in outputs `Qstat` and `crit`, respectively.

References

[1] Brockwell, P. J. and R. A. Davis. *Introduction to Time Series and Forecasting*. 2nd ed. New York, NY: Springer, 2002.

See Also

`autocorr` | `lbqtest` | `parcorr`

Related Examples

- "Detect ARCH Effects" on page 3-28
- "Choose ARMA Lags Using BIC" on page 5-135

- "Specify Multiplicative ARIMA Model" on page 5-52
- "Specify Conditional Mean and Variance Models" on page 5-79

More About

- "Autocorrelation and Partial Autocorrelation" on page 3-13
- "Ljung-Box Q-Test" on page 3-16
- "Moving Average Model" on page 5-27
- "Goodness of Fit" on page 3-88

Engle's ARCH Test

An uncorrelated time series can still be serially dependent due to a dynamic conditional variance process. A time series exhibiting conditional heteroscedasticity —or autocorrelation in the squared series—is said to have *autoregressive conditional heteroscedastic* (ARCH) effects. Engle's ARCH test is a Lagrange multiplier test to assess the significance of ARCH effects [1].

Consider a time series

$$y_t = \mu_t + \varepsilon_t,$$

where μ_t is the conditional mean of the process, and ε_t is an innovation process with mean zero.

Suppose the innovations are generated as

$$\varepsilon_t = \sigma_t z_t,$$

where z_t is an independent and identically distributed process with mean 0 and variance 1. Thus,

$$E(\varepsilon_t \varepsilon_{t+h}) = 0$$

for all lags $h \neq 0$ and the innovations are uncorrelated.

Let H_t denote the history of the process available at time t. The conditional variance of y_t is

$$Var(y_t \mid H_{t-1}) = Var(\varepsilon_t \mid H_{t-1}) = E(\varepsilon_t^2 \mid H_{t-1}) = \sigma_t^2.$$

Thus, conditional heteroscedasticity in the variance process is equivalent to autocorrelation in the squared innovation process.

Define the residual series

$$e_t = y_t - \hat{\mu}_t.$$

If all autocorrelation in the original series, y_t, is accounted for in the conditional mean model, then the residuals are uncorrelated with mean zero. However, the residuals can still be serially dependent.

The alternative hypothesis for Engle's ARCH test is autocorrelation in the squared residuals, given by the regression

$$H_a : e_t^2 = \alpha_0 + \alpha_1 e_{t-1}^2 + \ldots + \alpha_m e_{t-m}^2 + u_t,$$

where u_t is a white noise error process. The null hypothesis is

$$H_0 : \alpha_0 = \alpha_1 = \ldots = \alpha_m = 0.$$

To conduct Engle's ARCH test using `archtest`, you need to specify the lag m in the alternative hypothesis. One way to choose m is to compare loglikelihood values for different choices of m. You can use the likelihood ratio test (`lratiotest`) or information criteria (`aicbic`) to compare loglikelihood values.

To generalize to a GARCH alternative, note that a GARCH(P,Q) model is locally equivalent to an ARCH($P + Q$) model. This suggests also considering values $m = P + Q$ for reasonable choices of P and Q.

The test statistic for Engle's ARCH test is the usual F statistic for the regression on the squared residuals. Under the null hypothesis, the F statistic follows a χ^2 distribution with m degrees of freedom. A large critical value indicates rejection of the null hypothesis in favor of the alternative.

As an alternative to Engle's ARCH test, you can check for serial dependence (ARCH effects) in a residual series by conducting a Ljung-Box Q-test on the first m lags of the squared residual series with `lbqtest`. Similarly, you can explore the sample autocorrelation and partial autocorrelation functions of the squared residual series for evidence of significant autocorrelation.

References

[1] Engle, Robert F. "Autoregressive Conditional Heteroskedasticity with Estimates of the Variance of United Kingdom Inflation." *Econometrica*. Vol. 50, 1982, pp. 987–1007.

See Also

`aicbic` | `archtest` | `lbqtest` | `lratiotest`

Related Examples

- "Detect ARCH Effects" on page 3-28
- "Specify Conditional Mean and Variance Models" on page 5-79

More About

- "Ljung-Box Q-Test" on page 3-16
- "Autocorrelation and Partial Autocorrelation" on page 3-13
- "Model Comparison Tests" on page 3-65
- "Information Criteria" on page 3-63
- "Conditional Variance Models" on page 6-2

Detect ARCH Effects

In this section...

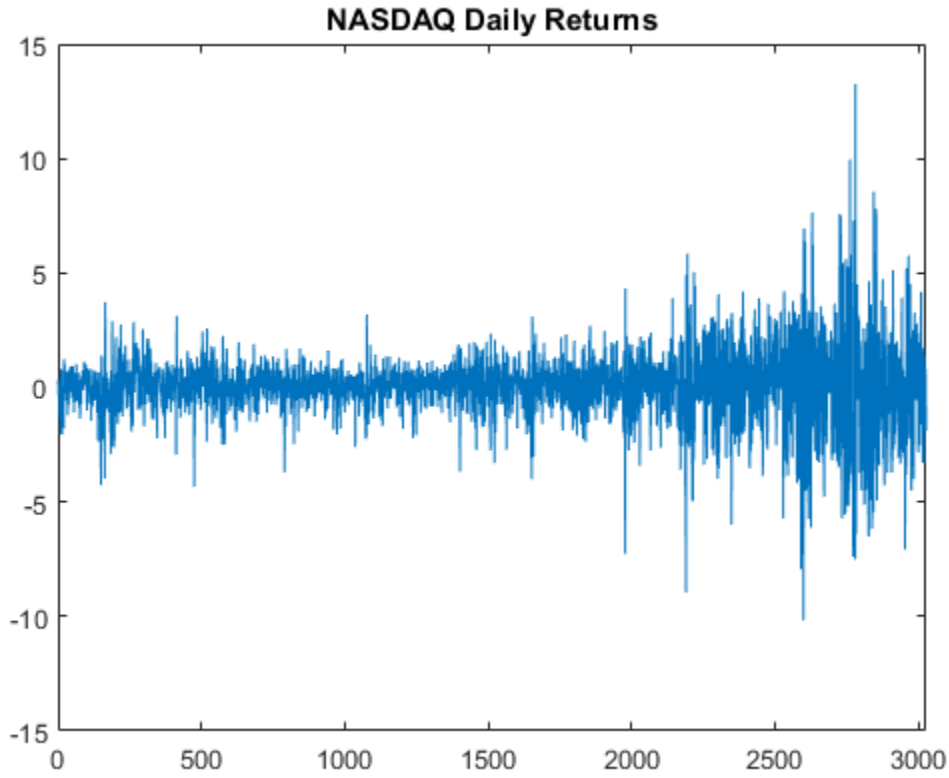
Test Autocorrelation of Squared Residuals

This example shows how to inspect a squared residual series for autocorrelation by plotting the sample autocorrelation function (ACF) and partial autocorrelation function (PACF). Then, conduct a Ljung-Box Q-test to more formally assess autocorrelation.

Load the Data.

Load the NASDAQ data included with the toolbox. Convert the daily close composite index series to a percentage return series.

```
load Data_EquityIdx;
y = DataTable.NASDAQ;
r = 100*price2ret(y);
T = length(r);

figure
plot(r)
xlim([0,T])
title('NASDAQ Daily Returns')
```



The returns appear to fluctuate around a constant level, but exhibit volatility clustering. Large changes in the returns tend to cluster together, and small changes tend to cluster together. That is, the series exhibits conditional heteroscedasticity.

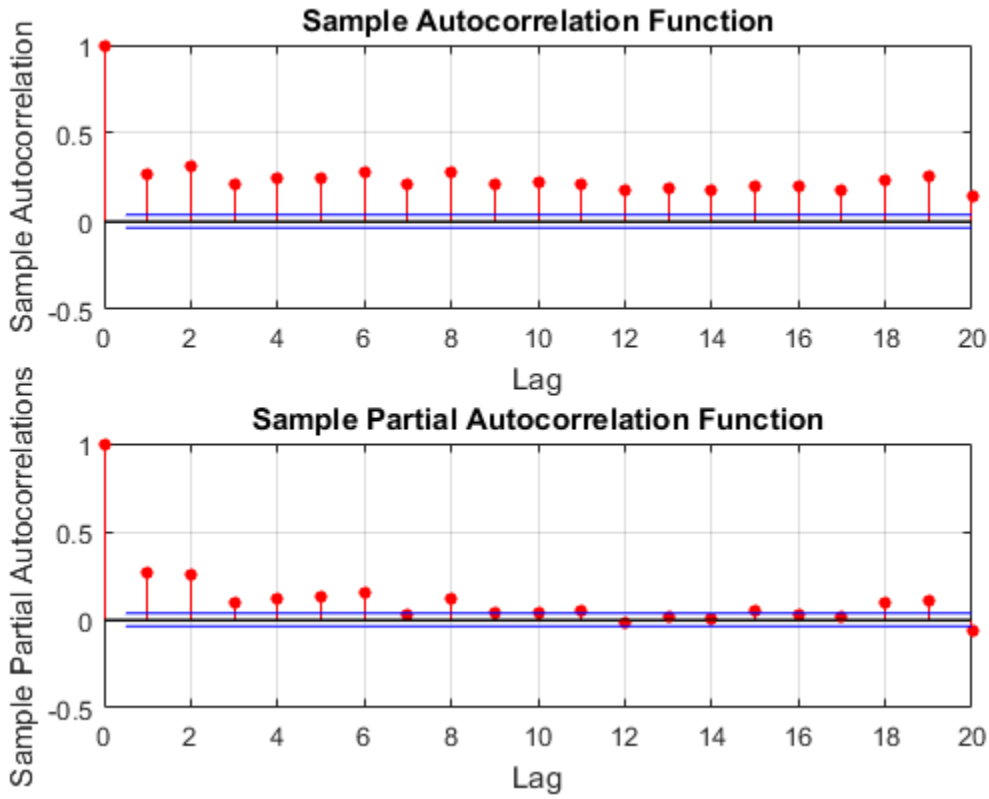
The returns are of relatively high frequency. Therefore, the daily changes can be small. For numerical stability, it is good practice to scale such data.

Plot the Sample ACF and PACF.

Plot the sample ACF and PACF for the squared residual series.

```
e = r - mean(r);

figure
```

```
subplot(2,1,1)
autocorr(e.^2)
subplot(2,1,2)
parcorr(e.^2)
```



The sample ACF and PACF show significant autocorrelation in the squared residual series. This indicates that volatility clustering is present in the residual series.

Conduct a Ljung-Box Q-test.

Conduct a Ljung-Box Q-test on the squared residual series at lags 5 and 10.

```
[h,p] = lbqtest(e.^2,'Lags',[5,10])
```

```
h =

     1     1


p =

     0     0
```

The null hypothesis is rejected for the two tests (`h = 1`). The p values for both tests is `0`. Thus, not all of the autocorrelations up to lag 5 (or 10) are zero, indicating volatility clustering in the residual series.

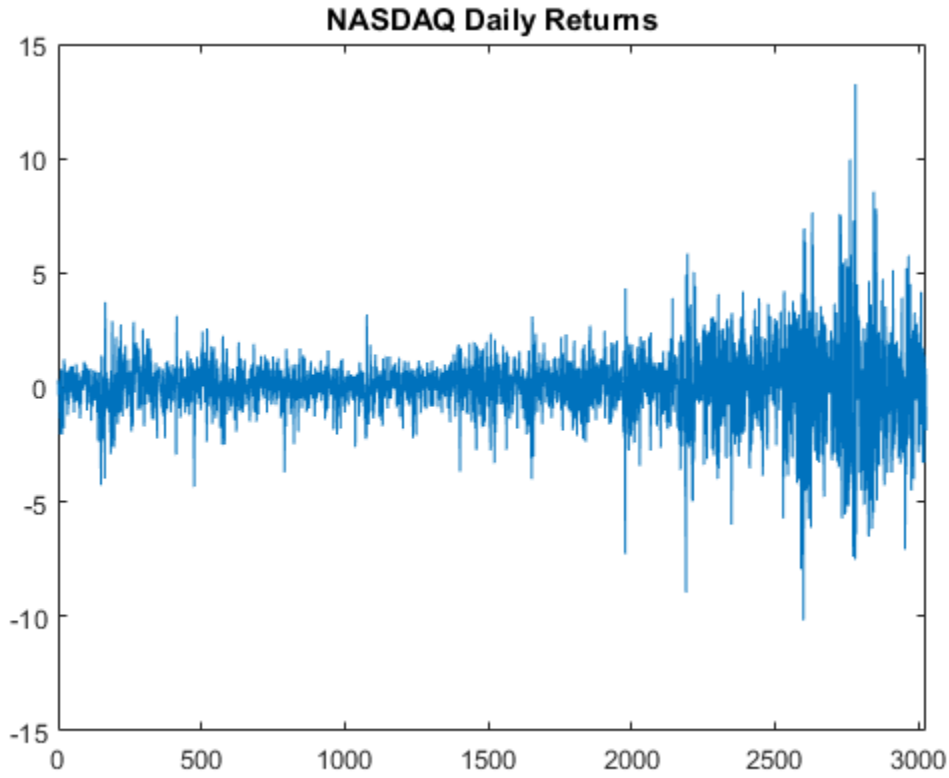
Conduct Engle's ARCH Test

This example shows how to conduct Engle's ARCH test for conditional heteroscedasticity.

Load the Data.

Load the NASDAQ data included with the toolbox. Convert the daily close composite index series to a percentage return series.

```
load Data_EquityIdx;
y = DataTable.NASDAQ;
r = 100*price2ret(y);
T = length(r);

figure
plot(r)
xlim([0,T])
title('NASDAQ Daily Returns')
```



The returns appear to fluctuate around a constant level, but exhibit volatility clustering. Large changes in the returns tend to cluster together, and small changes tend to cluster together. That is, the series exhibits conditional heteroscedasticity.

The returns are of relatively high frequency. Therefore, the daily changes can be small. For numerical stability, it is good practice to scale such data.

Conduct Engle's ARCH Test.

Conduct Engle's ARCH test for conditional heteroscedasticity on the residual series, using two lags in the alternative hypothesis.

```
e = r - mean(r);
[h,p,fStat,crit] = archtest(e,'Lags',2)
```

```
h =

     1


p =

     0


fStat =

  399.9693


crit =

    5.9915
```

The null hypothesis is soundly rejected (`h = 1`, `p = 0`) in favor of the ARCH(2) alternative. The F statistic for the test is `399.97`, much larger than the critical value from the χ^2 distribution with two degrees of freedom, `5.99`.

The test concludes there is significant volatility clustering in the residual series.

See Also

`archtest` | `autocorr` | `lbqtest` | `parcorr`

Related Examples

- “Detect Autocorrelation” on page 3-18
- “Specify Conditional Mean and Variance Models” on page 5-79

More About

- “Engle’s ARCH Test” on page 3-25
- “Autocorrelation and Partial Autocorrelation” on page 3-13
- “Conditional Variance Models” on page 6-2

Unit Root Nonstationarity

In this section...

What Is a Unit Root Test?

A *unit root* process is a data-generating process whose first difference is stationary. In other words, a unit root process y_t has the form
$y_t = y_{t-1}$ + stationary process.

A unit root test attempts to determine whether a given time series is consistent with a unit root process.

The next section gives more details of unit root processes, and suggests why it is important to detect them.

Modeling Unit Root Processes

There are two basic models for economic data with linear growth characteristics:

- Trend-stationary process (TSP): $y_t = c + \delta t$ + stationary process
- Unit root process, also called a difference-stationary process (DSP): $\Delta y_t = \delta$ + stationary process

Here Δ is the differencing operator, $\Delta y_t = y_t - y_{t-1} = (1 - L)y_t$, where L is the lag operator defined by $L^i y_t = y_{t-i}$.

The processes are indistinguishable for finite data. In other words, there are both a TSP and a DSP that fit a finite data set arbitrarily well. However, the processes are distinguishable when restricted to a particular subclass of data-generating processes, such as AR(p) processes. After fitting a model to data, a unit root test checks if the AR(1) coefficient is 1.

There are two main reasons to distinguish between these types of processes:

- "Forecasting" on page 3-35
- "Spurious Regression" on page 3-38

Forecasting

A TSP and a DSP produce different forecasts. Basically, shocks to a TSP return to the trend line $c + \delta t$ as time increases. In contrast, shocks to a DSP might be persistent over time.

For example, consider the simple trend-stationary model
$y_{1,t} = 0.9y_{1,t-1} + 0.02t + \varepsilon_{1,t}$

and the difference-stationary model
$y_{2,t} = 0.2 + y_{2,t-1} + \varepsilon_{2,t}$.

In these models, $\varepsilon_{1,t}$ and $\varepsilon_{2,t}$ are independent innovation processes. For this example, the innovations are independent and distributed $N(0,1)$.

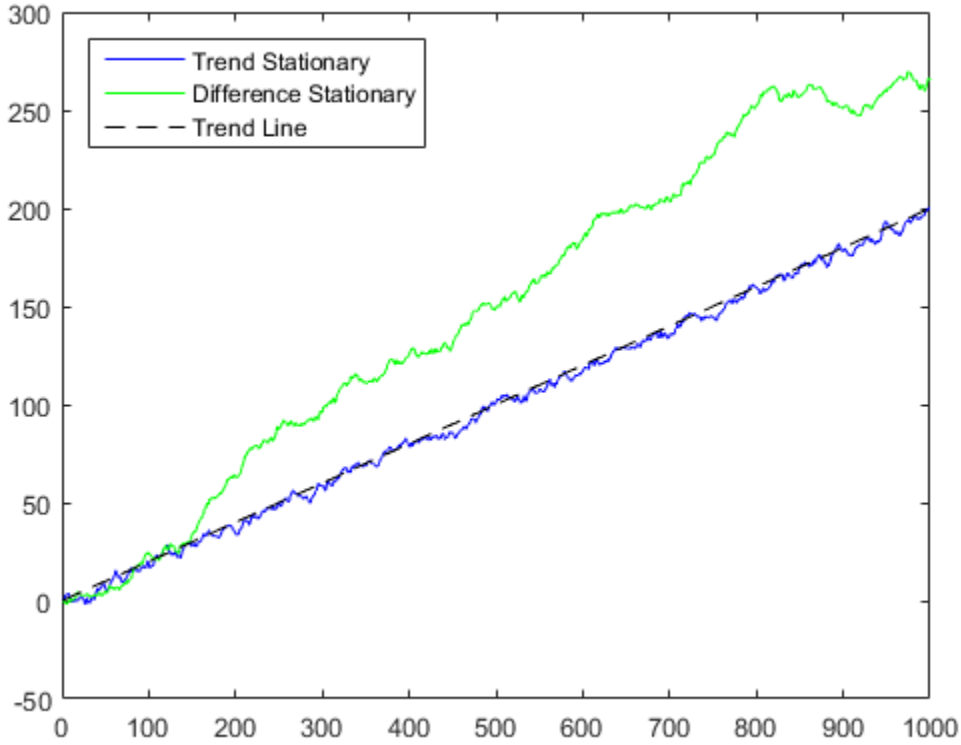
Both processes grow at rate 0.2. To calculate the growth rate for the TSP, which has a linear term $0.02t$, set $\varepsilon_1(t) = 0$. Then solve the model $y_1(t) = c + \delta t$ for c and δ:
$c + \delta t = 0.9(c + \delta(t-1)) + 0.02t$.

The solution is $c = -1.8$, $\delta = 0.2$.

A plot for $t = 1{:}1000$ shows the TSP stays very close to the trend line, while the DSP has persistent deviations away from the trend line.

```
T = 1000;   % Sample size
t = (1:T)'; % Period vector
rng(5);     % For reproducibility

randm = randn(T,2); % Innovations
y = zeros(T,2);     % Columns of y are data series

% Build trend stationary series
y(:,1) = .02*t + randm(:,1);
for ii = 2:T
    y(ii,1) = y(ii,1) + y(ii-1,1)*.9;
end

% Build difference stationary series
```

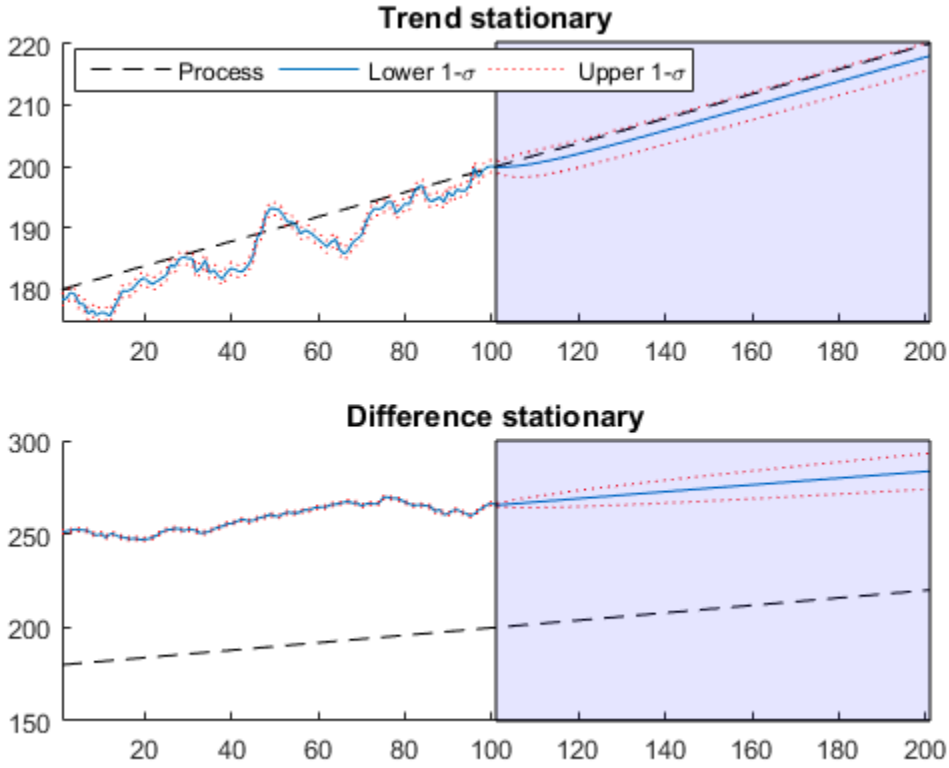
```
y(:,2) = .2 + randm(:,2);
y(:,2) = cumsum(y(:,2));

figure
plot(y(:,1),'b')
hold on
plot(y(:,2),'g')
plot((1:T)*0.2,'k--')
legend('Trend Stationary','Difference Stationary',...
    'Trend Line','Location','NorthWest')
hold off
```



Forecasts based on the two series are different. To see this difference, plot the predicted behavior of the two series using `vgxpred`. The following plot shows the last 100 data

points in the two series and predictions of the next 100 points, including confidence bounds.

```
Mdl = vgxset('AR',zeros(2),'ARSolve',...
     [true false;false true],'nx',1,'Constant',...
     true,'n',2);     % Model for independent processes
tcell = cell(1000,1); % Time as exogenous input
for i=1:1000
    tcell{i} = [i;0];
end

MdlFitted = vgxvarx(Mdl,y,tcell);
MdlFitted = vgxset(MdlFitted,'Series',...
    {'Trend stationary','Difference stationary'});
fx = cell(100,1);
for i = 1:100
    fx{i} = [i+1000;0]; % Future times for prediction
end
[ynew,ycov] = vgxpred(MdlFitted,100,fx,y);
% This generates predictions for 100 time steps

figure;
subplot(2,1,1);
hold on;
plot((T-100:T+100)*0.2,'k--');
axis tight;
subplot(2,1,2);
hold on;
plot((T-100:T+100)*0.2,'k--');
vgxplot(MdlFitted,y(end-100:end,:),ynew,ycov);
axis tight;
```



Examine the fitted parameters by executing `vgxdisp(specfitted)` and you find `vgxvarx` did an excellent job.

The TSP has confidence intervals that do not grow with time, whereas the DSP has confidence intervals that grow. Furthermore, the TSP goes to the trend line quickly, while the DSP does not tend towards the trend line $y = 0.2t$ asymptotically.

Spurious Regression

The presence of unit roots can lead to false inferences in regressions between time series.

Suppose x_t and y_t are unit root processes with independent increments, such as random walks with drift

$x_t = c_1 + x_{t-1} + \varepsilon_1(t)$
$y_t = c_2 + y_{t-1} + \varepsilon_2(t)$,

where $\varepsilon_i(t)$ are independent innovations processes. Regressing y on x results, in general, in a nonzero regression coefficient, and significant coefficient of determination R^2. This result holds despite x_t and y_t being independent random walks.

If both processes have trends ($c_i \neq 0$), there is a correlation between x and y because of their linear trends. However, even if the $c_i = 0$, the presence of unit roots in the x_t and y_t processes yields correlation. For more information on spurious regression, see Granger and Newbold [1].

Available Tests

There are four Econometrics Toolbox tests for unit roots. These functions test for the existence of a *single* unit root. When there are two or more unit roots, the results of these tests might not be valid.

- "Dickey-Fuller and Phillips-Perron Tests" on page 3-39
- "KPSS Test" on page 3-40
- "Variance Ratio Test" on page 3-40

Dickey-Fuller and Phillips-Perron Tests

`adftest` performs the augmented Dickey-Fuller test. `pptest` performs the Phillips-Perron test. These two classes of tests have a null hypothesis of a unit root process of the form
$y_t = y_{t-1} + c + \delta t + \varepsilon_t$,

which the functions test against an alternative model
$y_t = \gamma y_{t-1} + c + \delta t + \varepsilon_t$,

where $\gamma < 1$. The null and alternative models for a Dickey-Fuller test are like those for a Phillips-Perron test. The difference is `adftest` extends the model with extra parameters accounting for serial correlation among the innovations:
$y_t = c + \delta t + \gamma y_{t-1} + \phi_1 \Delta y_{t-1} + \phi_2 \Delta y_{t-2} + ... + \phi_p \Delta y_{t-p} + \varepsilon_t$,

where

- L is the lag operator: $Ly_t = y_{t-1}$.

- $\Delta = 1 - L$, so $\Delta y_t = y_t - y_{t-1}$.
- ε_t is the innovations process.

Phillips-Perron adjusts the test statistics to account for serial correlation.

There are three variants of both `adftest` and `pptest`, corresponding to the following values of the `'model'` parameter:

- `'AR'` assumes c and δ, which appear in the preceding equations, are both `0`; the `'AR'` alternative has mean 0.
- `'ARD'` assumes δ is `0`. The `'ARD'` alternative has mean $c/(1-\gamma)$.
- `'TS'` makes no assumption about c and δ.

For information on how to choose the appropriate value of `'model'`, see "Choose Models to Test" on page 3-41.

KPSS Test

The KPSS test, `kpsstest`, is an inverse of the Phillips-Perron test: it reverses the null and alternative hypotheses. The KPSS test uses the model:
$y_t = c_t + \delta t + u_t$, with
$c_t = c_{t-1} + v_t$.

Here u_t is a stationary process, and v_t is an i.i.d. process with mean 0 and variance o^2. The null hypothesis is that $o^2 = 0$, so that the random walk term c_t becomes a constant intercept. The alternative is $o^2 > 0$, which introduces the unit root in the random walk.

Variance Ratio Test

The variance ratio test, `vratiotest`, is based on the fact that the variance of a random walk increases linearly with time. `vratiotest` can also take into account heteroscedasticity, where the variance increases at a variable rate with time. The test has a null hypotheses of a random walk:
$\Delta y_t = \varepsilon_t$.

Testing for Unit Roots

- "Transform Data" on page 3-41
- "Choose Models to Test" on page 3-41
- "Determine Appropriate Lags" on page 3-41

- "Conduct Unit Root Tests at Multiple Lags" on page 3-42

Transform Data

Transform your time series to be approximately linear before testing for a unit root. If a series has exponential growth, take its logarithm. For example, GDP and consumer prices typically have exponential growth, so test their logarithms for unit roots.

If you want to transform your data to be stationary instead of approximately linear, unit root tests can help you determine whether to difference your data, or to subtract a linear trend. For a discussion of this topic, see "What Is a Unit Root Test?" on page 3-34

Choose Models to Test

- For `adftest` or `pptest`, choose `model` in as follows:
  - If your data shows a linear trend, set `model` to `'TS'`.
  - If your data shows no trend, but seem to have a nonzero mean, set `model` to `'ARD'`.
  - If your data shows no trend and seem to have a zero mean, set `model` to `'AR'` (the default).
- For `kpsstest`, set `trend` to `true` (default) if the data shows a linear trend. Otherwise, set `trend` to `false`.
- For `vratiotest`, set `IID` to `true` if you want to test for independent, identically distributed innovations (no heteroscedasticity). Otherwise, leave `IID` at the default value, `false`. Linear trends do not affect `vratiotest`.

Determine Appropriate Lags

Setting appropriate lags depends on the test you use:

- `adftest` — One method is to begin with a maximum lag, such as the one recommended by Schwert [2]. Then, test down by assessing the significance of the coefficient of the term at lag $p_{\max}$. Schwert recommends a maximum lag of

  $$p_{\max} = \text{maximum lag} = \left\lfloor 12(T/100)^{1/4} \right\rfloor,$$

  where $\lfloor x \rfloor$ is the integer part of x. The usual t statistic is appropriate for testing the significance of coefficients, as reported in the `reg` output structure.

Another method is to combine a measure of fit, such as SSR, with information criteria such as AIC, BIC, and HQC. These statistics also appear in the `reg` output structure. Ng and Perron [3] provide further guidelines.

- `kpsstest` — One method is to begin with few lags, and then evaluate the sensitivity of the results by adding more lags. For consistency of the Newey-West estimator, the number of lags must go to infinity as the sample size increases. Kwiatkowski et al. [4] suggest using a number of lags on the order of $T^{1/2}$, where T is the sample size.

  For an example of choosing lags for `kpsstest`, see "Test Time Series Data for a Unit Root" on page 3-50.

- `pptest` — One method is to begin with few lags, and then evaluate the sensitivity of the results by adding more lags. Another method is to look at sample autocorrelations of $y_t - y_{t-1}$; slow rates of decay require more lags. The Newey-West estimator is consistent if the number of lags is $O(T^{1/4})$, where T is the effective sample size, adjusted for lag and missing values. White and Domowitz [5] and Perron [6] provide further guidelines.

  For an example of choosing lags for `pptest`, see "Test Time Series Data for a Unit Root" on page 3-50.

- `vratiotest` does not use lags.

Conduct Unit Root Tests at Multiple Lags

Run multiple tests simultaneously by entering a vector of parameters for `lags`, `alpha`, `model`, or `test`. All vector parameters must have the same length. The test expands any scalar parameter to the length of a vector parameter. For an example using this technique, see "Test Time Series Data for a Unit Root" on page 3-50.

References

[1] Granger, C. W. J., and P. Newbold. "Spurious Regressions in Econometrics." *Journal of Econometrics*. Vol2, 1974, pp. 111–120.

[2] Schwert, W. "Tests for Unit Roots: A Monte Carlo Investigation." *Journal of Business and Economic Statistics*. Vol. 7, 1989, pp. 147–159.

[3] Ng, S., and P. Perron. "Unit Root Tests in ARMA Models with Data-Dependent Methods for the Selection of the Truncation Lag." *Journal of the American Statistical Association*. Vol. 90, 1995, pp. 268–281.

[4] Kwiatkowski, D., P. C. B. Phillips, P. Schmidt and Y. Shin. "Testing the Null Hypothesis of Stationarity against the Alternative of a Unit Root." *Journal of Econometrics*. Vol. 54, 1992, pp. 159–178.

[5] White, H., and I. Domowitz. "Nonlinear Regression with Dependent Observations." *Econometrica*. Vol. 52, 1984, pp. 143–162.

[6] Perron, P. "Trends and Random Walks in Macroeconomic Time Series: Further Evidence from a New Approach." *Journal of Economic Dynamics and Control*. Vol. 12, 1988, pp. 297–332.

See Also

`adftest` | `kpsstest` | `pptest` | `vgxpred` | `vratiotest`

Related Examples

- "Unit Root Tests" on page 3-44
- "Assess Stationarity of a Time Series" on page 3-58

Unit Root Tests

In this section...

Test Simulated Data for a Unit Root

This example shows how to test univariate time series models for stationarity. It shows how to simulate data from four types of models: trend stationary, difference stationary, stationary (AR(1)), and a heteroscedastic, random walk model. It also shows that the tests yield expected results.

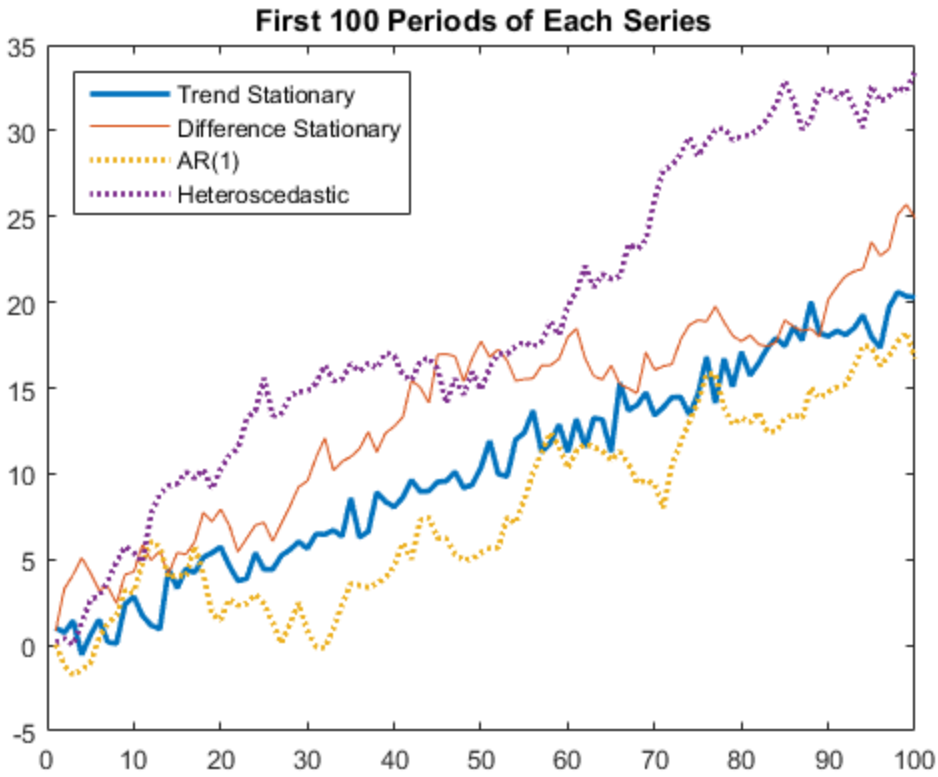
Simulate four time series.

```
T = 1e3;       % Sample size
t = (1:T)';    % Time multiple

rng(142857);   % For reproducibility

y1 = randn(T,1) + .2*t; % Trend stationary

Mdl2 = arima('D',1,'Constant',0.2,'Variance',1);
y2 = simulate(Mdl2,T,'Y0',0); % Difference stationary

Mdl3 = arima('AR',0.99,'Constant',0.2,'Variance',1);
y3 = simulate(Mdl3,T,'Y0',0); % AR(1)

Mdl4 = arima('D',1,'Constant',0.2,'Variance',1);
sigma = (sin(t/200) + 1.5)/2; % Std deviation
e = randn(T,1).*sigma;        % Innovations
y4 = filter(Mdl4,e,'Y0',0);   % Heteroscedastic
```

Plot the first 100 points in each series.

```
y = [y1 y2 y3 y4];
figure;
plot1 = plot(y(1:100,:));
plot1(1).LineWidth = 2;
```

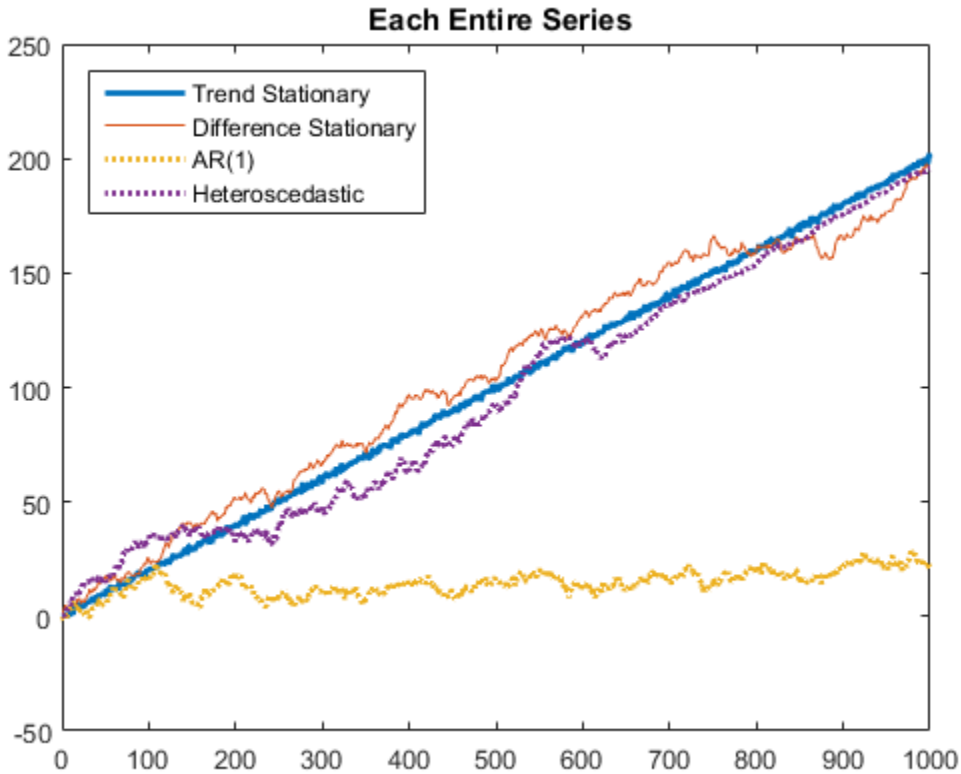
```
plot1(3).LineStyle = ':';
plot1(3).LineWidth = 2;
plot1(4).LineStyle = ':';
plot1(4).LineWidth = 2;
title '{\bf First 100 Periods of Each Series}';
legend('Trend Stationary','Difference Stationary','AR(1)',...
   'Heteroscedastic','location','northwest');
```



All of the models appear nonstationary and behave similarly. Therefore, you might find it difficult to distinguish which series comes from which model simply by looking at their initial segments.

Plot the entire data set.

```
plot2 = plot(y);
plot2(1).LineWidth = 2;
plot2(3).LineStyle = ':';
plot2(3).LineWidth = 2;
plot2(4).LineStyle = ':';
plot2(4).LineWidth = 2;
title '{\bf Each Entire Series}';
legend('Trend Stationary','Difference Stationary','AR(1)',...
   'Heteroscedastic','location','northwest');
```



The differences between the series are clearer here:

- The trend stationary series has little deviation from its mean trend.

- The difference stationary and heteroscedastic series have persistent deviations away from the trend line.
- The AR(1) series exhibits long-run stationary behavior; the others grow linearly.
- The difference stationary and heteroscedastic series appear similar. However, that the heteroscedastic series has much more local variability near period 300, and much less near period 900. The model variance is maximal when $\sin(t/200) = 1$, at time $100\pi \approx 314$. The model variance is minimal when $\sin(t/200) = -1$, at time $300\pi \approx 942$. Therefore, the visual variability matches the model.

Use the Augmented Dicky-Fuller test on the three growing series (`y1`, `y2`, and `y4`) to assess whether the series have a unit root. Since the series are growing, specify that there is a trend. In this case, the null hypothesis is $H_0 : y_t = y_{t-1} + c + b_1\Delta y_{t-1} + b_2\Delta y_{t-2} + \varepsilon_t$ and the alternative hypothesis is $H_1 : y_t = ay_{t-1} + c + \delta t + b_1\Delta y_{t-1} + b_2\Delta y_{t-2} + \varepsilon_t$. Set the number of lags to 2 for demonstration purposes.

```
hY1 = adftest(y1, 'model','ts', 'lags',2)
hY2 = adftest(y2, 'model','ts', 'lags',2)
hY4 = adftest(y4, 'model','ts', 'lags',2)
```

```
hY1 =

     1


hY2 =

     0


hY4 =

     0
```

- `hY1 = 1` indicates that there is sufficient evidence to auggest that `y1` is trend stationary. This is the correct decision because `y1` is trend stationary by construction.
- `hY2 = 0` indicates that there is not enough evidence to suggest that `y2` is trend stationary. This is the correct decision since `y2` is difference stationary by construction.

- `hY4 = 0` indicates that there is not enough evidence to suggest that `y4` is trend stationary. This is the correct decision, however, the Dickey-Fuller test is not appropriate for a heteroscedastic series.

Use the Augmented Dickey-Fuller test on the AR(1) series (`y3`) to assess whether the series has a unit root. Since the series is not growing, specify that the series is autoregressive with a drift term. In this case, the null hypothesis is $H_0 : y_t = y_{t-1} + b_1\Delta y_{t-1} + b_2\Delta y_{t-2} + \varepsilon_t$ and the alternative hypothesis is $H_1 : y_t = ay_{t-1} + b_1\Delta y_{t-1} + b_2\Delta y_{t-2} + \varepsilon_t$. Set the number of lags to 2 for demonstration purposes.

```
hY3 = adftest(y3, 'model','ard', 'lags',2)
```

```
hY3 =

     1
```

`hY3 = 1` indicates that there is enough evidence to suggest that `y3` is a stationary, autoregressive process with a drift term. This is the correct decision because `y3` is an autoregressive process with a drift term by construction.

Use the KPSS test to assess whether the series are unit root nonstationary. Specify that there is a trend in the growing series (`y1`, `y2`, and `y4`). The KPSS test assumes the following model:

$$y_y = c_t + \delta t + u_t$$

$$c_t = c_{t-1} + \varepsilon_t,$$

where u_t is a stationary process and ε_t is an independent and identically distributed process with mean 0 and variance σ^2. Whether there is a trend in the model, the null hypothesis is $H_0 : \sigma^2 = 0$ (the series is trend stationary) and the alternative hypothesis is $H_1 : \sigma^2 > 0$ (not trend stationary). Set the number of lags to 2 for demonstration purposes.

```
hY1 = kpsstest(y1, 'lags',2, 'trend',true)
hY2 = kpsstest(y2, 'lags',2, 'trend',true)
hY3 = kpsstest(y3, 'lags',2)
hY4 = kpsstest(y4, 'lags',2, 'trend',true)
```

```
hY1 =

     0


hY2 =

     1


hY3 =

     1


hY4 =

     1
```

All is tests result in the correct decision.

Use the variance ratio test on al four series to assess whether the series are random walks. The null hypothesis is H_0: $Var(\Delta y_t)$ is constant, and the alternative hypothesis is H_1: $Var(\Delta y_t)$ is not constant. Specify that the innovations are independent and identically distributed for all but `y1`. Test `y4` both ways.

```
hY1 = vratiotest(y1)
hY2 = vratiotest(y2,'IID',true)
hY3 = vratiotest(y3,'IID',true)
hY4NotIID = vratiotest(y4)
hY4IID = vratiotest(y4, 'IID',true)
```

```
hY1 =

     1


hY2 =

     0
```

```
hY3 =

     0


hY4NotIID =

     0


hY4IID =

     0
```

All tests result in the correct decisions, except for `hY4_2 = 0`. This test does not reject the hypothesis that the heteroscedastic process is an IID random walk. This inconsistency might be associated with the random seed.

Alternatively, you can assess stationarity using `pptest`

Test Time Series Data for a Unit Root

This example shows how to test a univariate time series for a unit root. It uses wages data (1900-1970) in the manufacturing sector. The series is in the Nelson-Plosser data set.

Load the Nelson-Plosser data. Extract the nominal wages data.

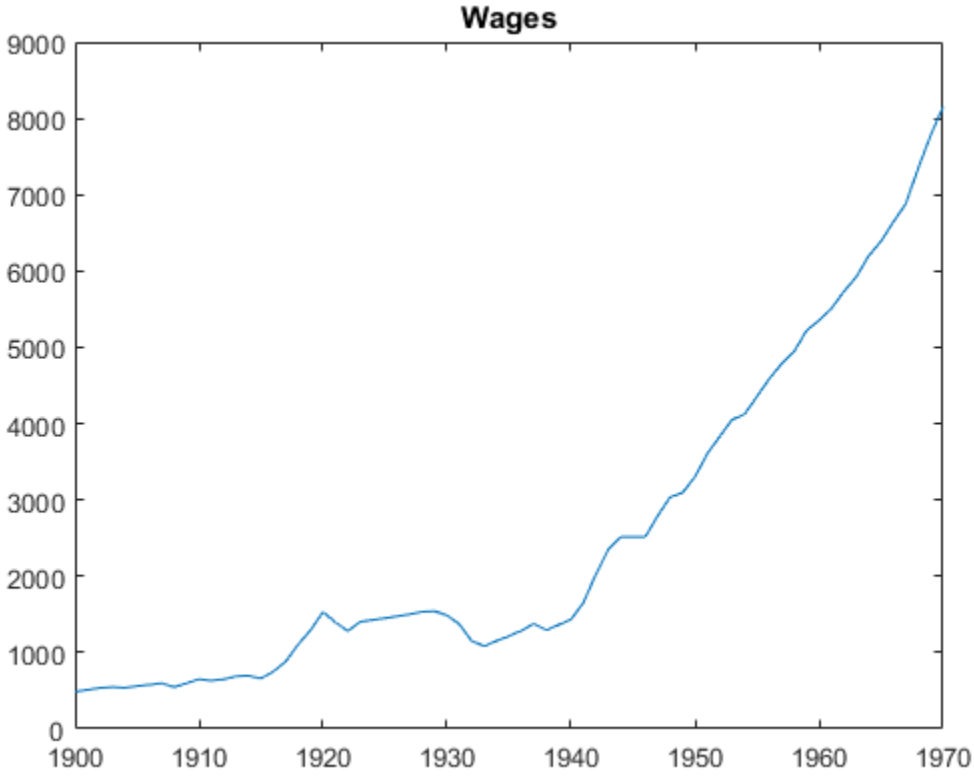
```
load Data_NelsonPlosser
wages = DataTable.WN;
```

Trim the `NaN` values from the series and the corresponding dates (this step is optional, since the test ignores `NaN` values).

```
wDates = dates(isfinite(wages));
wages = wages(isfinite(wages));
```

Plot the data to look for trends.

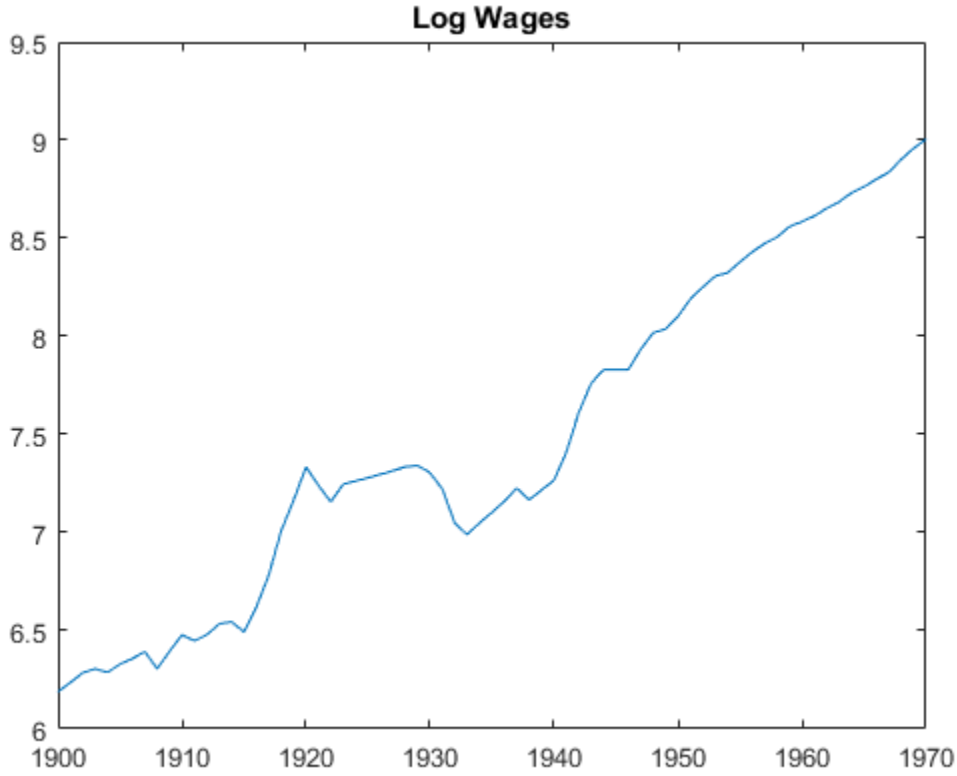
```
plot(wDates,wages)
title('Wages')
```



The plot suggests exponential growth.

Transform the data using the log function to linearize the series.

```
logWages = log(wages);
plot(wDates,logWages)
title('Log Wages')
```



The data appear to have a linear trend.

Test the hypothesis that the series is a unit root process with a trend (difference stationary), against the alternative that there is no unit root (trend stationary). Set `'lags',[7:2:11]`, as suggested in Kwiatkowski et al., 1992.

```
[h,pValue] = kpsstest(logWages,'lags',[7:2:11])
```

```
Warning: Test statistic #1 below tabulated critical values:
maximum p-value = 0.100 reported.
Warning: Test statistic #2 below tabulated critical values:
maximum p-value = 0.100 reported.
Warning: Test statistic #3 below tabulated critical values:
maximum p-value = 0.100 reported.
```

```
h =

     0     0     0


pValue =

    0.1000    0.1000    0.1000
```

`kpsstest` fails to reject the hypothesis that the wages series is trend stationary. If the result would have been [1 1 1], the two inferences would provide consistent evidence of a unit root. It remains unclear whether the data has a unit root. This is a typical result of tests on many macroeconomic series.

The warnings that the test statistic "...is below tabulated critical values" does not indicate a problem. `kpsstest` has a limited set of calculated critical values. When it calculates a test statistic that is outside this range, the test reports the p-value at the appropriate endpoint. So, in this case, `pValue` reflects the closest tabulated value. When a test statistic lies inside the span of tabulated values, `kpsstest` linearly interpolates the p-value.

Test Stock Data for a Random Walk

This example shows how to assess whether a time series is a random walk. It uses market data for daily returns of stocks and cash (money market) from the period January 1, 2000 to November 7, 2005.

Load the data.

```
load CAPMuniverse
```

Extract two series to test. The first column of data is the daily return of a technology stock. The last (14th) column is the daily return for cash (the daily money market rate).

```
tech1 = Data(:,1);
money = Data(:,14);
```

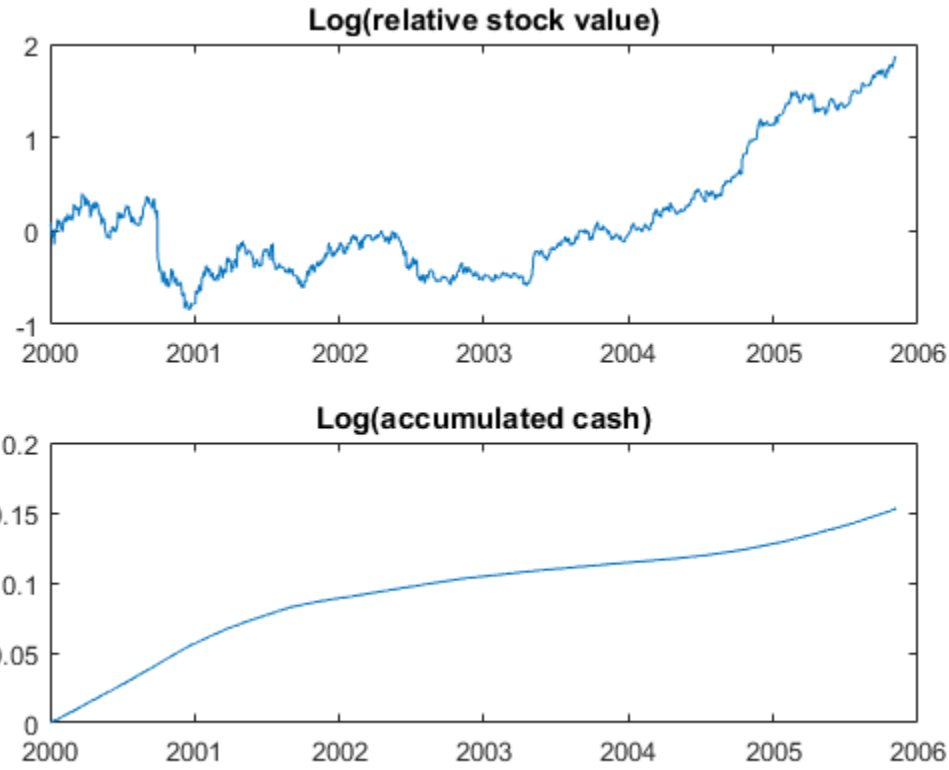
The returns are the logs of the ratios of values at the end of a day over the values at the beginning of the day.

Convert the data to prices (values) instead of returns. `vratiotest` takes prices as inputs, as opposed to returns.

```
tech1 = cumsum(tech1);
money = cumsum(money);
```

Plot the data to see whether they appear to be stationary.

```
subplot(2,1,1)
plot(Dates,tech1);
title('Log(relative stock value)')
datetick('x')
hold on
subplot(2,1,2);
plot(Dates,money)
title('Log(accumulated cash)')
datetick('x')
hold off
```



Cash has a small variability, and appears to have long-term trends. The stock series has a good deal of variability, and no definite trend, though it appears to increase towards the end.

Test whether the stock series matches a random walk.

```
[h,pValue,stat,cValue,ratio] = vratiotest(tech1)
```

```
h =

     0


pValue =

    0.1646


stat =

   -1.3899


cValue =

    1.9600


ratio =

    0.9436
```

`vratiotest` does not reject the hypothesis that a random walk is a reasonable model for the stock series.

Test whether an i.i.d. random walk is a reasonable model for the stock series.

```
[h,pValue,stat,cValue,ratio] = vratiotest(tech1,'IID',true)
```

```
h =

     1
```

```
pValue =

    0.0304


stat =

   -2.1642


cValue =

    1.9600


ratio =

    0.9436
```

`vratiotest` rejects the hypothesis that an i.i.d. random walk is a reasonable model for the `tech1` stock series at the 5% level. Thus, `vratiotest` indicates that the most appropriate model of the `tech1` series is a heteroscedastic random walk.

Test whether the cash series matches a random walk.

```
[h,pValue,stat,cValue,ratio] = vratiotest(money)
```

```
h =

     1


pValue =

  4.6093e-145


stat =

   25.6466
```

```
cValue =

    1.9600


ratio =

    2.0006
```

`vratiotest` emphatically rejects the hypothesis that a random walk is a reasonable model for the cash series (`pValue = 4.6093e-145`). The removal of a trend from the series does not affect the resulting statistics.

References

[1] Kwiatkowski, D., P. C. B. Phillips, P. Schmidt and Y. Shin. "Testing the Null Hypothesis of Stationarity against the Alternative of a Unit Root." *Journal of Econometrics*. Vol. 54, 1992, pp. 159–178.

See Also

`adftest` | `kpsstest` | `pptest` | `vratiotest`

More About

- "Unit Root Nonstationarity" on page 3-34

Assess Stationarity of a Time Series

This example shows how to check whether a linear time series is a unit root process in several ways. You can assess unit root nonstationarity statistically, visually, and algebraically.

Simulate Data

Suppose that the true model for a linear time series is

$$(1-0.2L)(1-L)y_t = (1-0.5L)\varepsilon_t$$

where the innovation series ε_t is iid with mean 0 and variance 1.5. Simulate data from this model. This model is a unit root process because the lag polynomial of the right side has characteristic root 1.

```
SimMod = arima('AR',0.2,'MA',-0.5,'D',1,'Constant',0,...
'Variance',1.5);
T = 30;
rng(5);
Y = simulate(SimMod,T);
```

Assess Stationarity Statistically

Econometrics Toolbox™ has four formal tests to choose from to check if a time series is nonstationary: `adftest`, `kpsstest`, `pptest`, and `vratiotest`. Use `adftest` to perform the Dickey-Fuller test on the data that you simulated in the previous steps.

```
adftest(Y)
```

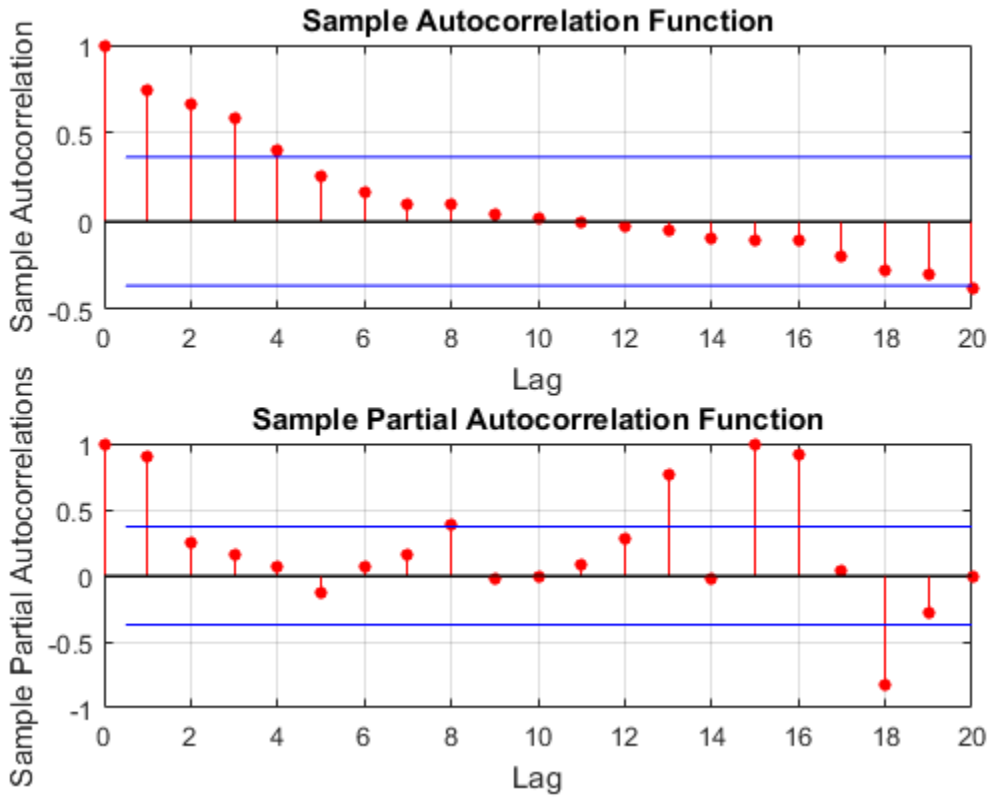
```
ans =

     0
```

The test result indicates that you should not reject the null hypothesis that the series is a unit root process.

Assess Stationarity Visually

Suppose you don't have the time series model, but you have the data. Inspect a plot of the data. Also, inspect the plots of the sample autocorrelation function (ACF) and sample partial autocorrelation function (PACF).

```
plot(Y);
title('Simulated Time Series')
xlabel('t')
ylabel('Y')

subplot(2,1,1)
autocorr(Y)
subplot(2,1,2)
parcorr(Y)
```



The downward sloping of the plot indicates a unit root process. The lengths of the line segments on the ACF plot gradually decay, and continue this pattern for increasing lags. This behavior indicates a nonstationary series.

Assess Stationarity Algebraically

Suppose you have the model in standard form:

$$y_t = 1.2y_{t-1} - 0.2y_{t-2} + \varepsilon_t - 0.5\varepsilon_{t-1}.$$

Write the equation in lag operator notation and solve for y_t to get

$$y_t = \frac{1 - 0.5L}{1 - 1.2L + 0.2L^2}\varepsilon_t.$$

Use `LagOp` to convert the rational polynomial to a polynomial. Also, use `isStable` to inspect the characteristic roots of the denominator.

```
num = LagOp([1 -0.5]);
denom = LagOp([1 -1.2 0.2]);
quot = mrdivide(num,denom);

[r1,r2] = isStable(denom)
```

```
Warning: Termination window not currently open and coefficients
         are not below tolerance.

r1 =

     0


r2 =

    1.0000
    0.2000
```

This warning indicates that the resulting quotient has a degree larger than 1001, e.g., there might not be a terminating degree. This indicates instability. `r1 = 0` indicates that the denominator is unstable. `r2` is a vector of characteristics roots, one of the roots is 1. Therefore, this is a unit root process.

`isStable` is a *numerical* routine that calculates the characteristic values of a polynomial. If you use `quot` as an argument to `isStable`, then the output might indicate that the polynomial is stable (i.e., all characteristic values are slightly less than

1). You might need to adjust the tolerance options of `isStable` to get more accurate results.

Test Multiple Time Series

"VAR Model Case Study" on page 7-89 contains an example that uses `vgxvarx` to estimate the loglikelihoods of several models, and uses `lratiotest` to reject some restricted models in favor of an unrestricted model. The calculation appears in the example "Classical Model Misspecification Tests".

Information Criteria

Model comparison tests—such as the likelihood ratio, Lagrange multiplier, or Wald test—are only appropriate for comparing nested models. In contrast, information criteria are model selection tools that you can use to compare any models fit to the same data. That is, the models being compared do not need to be nested.

Basically, information criteria are likelihood-based measures of model fit that include a penalty for complexity (specifically, the number of parameters). Different information criteria are distinguished by the form of the penalty, and can prefer different models.

Let $\log L(\theta)$ denote the value of the maximized loglikelihood objective function for a model with k parameters fit to N data points. Two commonly used information criteria are:

- **Akaike information criterion (AIC).** The AIC compares models from the perspective of information entropy, as measured by Kullback-Leibler divergence. The AIC for a given model is

  $$-2\log L(\theta) + 2k.$$

  When comparing AIC values for multiple models, smaller values of the criterion are better.
- **Bayesian information criterion (BIC)**. The BIC, also known as Schwarz information criterion, compares models from the perspective of decision theory, as measured by expected loss. The BIC for a given model is

  $$-2\log L(\theta) + k\log(N).$$

  When comparing BIC values for multiple models, smaller values of the criterion are better.

Note: Some references scale information criteria values by the number of observations (N). Econometrics Toolbox does not do this scaling. As a result, the absolute value of measures the toolbox returns might differ from other sources by a factor of N.

See Also

`aicbic`

Related Examples

- "Choose ARMA Lags Using BIC" on page 5-135
- "Compare Conditional Variance Models Using Information Criteria" on page 6-87

More About

- "Model Comparison Tests" on page 3-65
- "Goodness of Fit" on page 3-88

Model Comparison Tests

In this section...

Available Tests

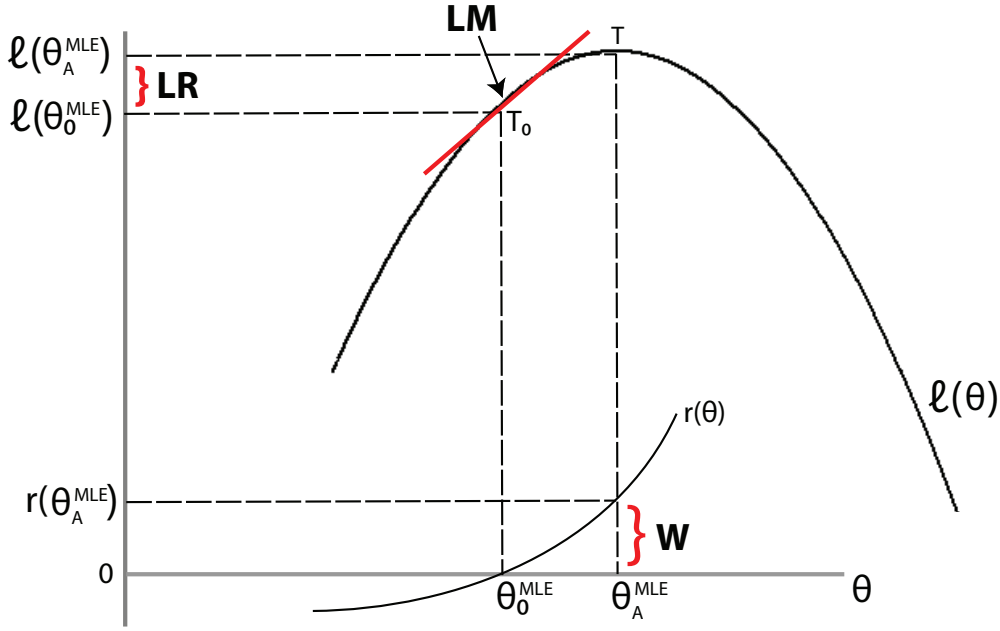
The primary goal of model selection is choosing the most parsimonious model that adequately fits your data. Three asymptotically equivalent tests compare a restricted model (the null model) against an unrestricted model (the alternative model), fit to the same data:

- Likelihood ratio (LR) test
- Lagrange multiplier (LM) test
- Wald (W) test

For a model with parameters θ, consider the restriction $r(\theta) = 0,$ which is satisfied by the null model. For example, consider testing the null hypothesis $\theta = \theta_0.$ The restriction function for this test is

$$r(\theta) = \theta - \theta_0.$$

The LR, LM, and Wald tests approach the problem of comparing the fit of a restricted model against an unrestricted model differently. For a given data set, let $l(\theta_0^{MLE})$ denote the loglikelihood function evaluated at the maximum likelihood estimate (MLE) of the restricted (null) model. Let $l(\theta_A^{MLE})$ denote the loglikelihood function evaluated at the MLE of the unrestricted (alternative) model. The following figure illustrates the rationale behind each test.



- **Likelihood ratio test.** If the restricted model is adequate, then the difference between the maximized objective functions, $l(\theta_A^{MLE}) - l(\theta_0^{MLE})$, should not significantly differ from zero.
- **Lagrange multiplier test.** If the restricted model is adequate, then the slope of the tangent of the loglikelihood function at the restricted MLE (indicated by T_0 in the figure) should not significantly differ from zero (which is the slope of the tangent of the loglikelihood function at the unrestricted MLE, indicated by T).
- **Wald test.** If the restricted model is adequate, then the restriction function evaluated at the unrestricted MLE should not significantly differ from zero (which is the value of the restriction function at the restricted MLE).

The three tests are asymptotically equivalent. Under the null, the LR, LM, and Wald test statistics are all distributed as χ^2 with degrees of freedom equal to the number of restrictions. If the test statistic exceeds the test critical value (equivalently, the p-value is less than or equal to the significance level), the null hypothesis is rejected. That is, the restricted model is rejected in favor of the unrestricted model.

Choosing among the LR, LM, and Wald test is largely determined by computational cost:

- To conduct a likelihood ratio test, you need to estimate both the restricted and unrestricted models.
- To conduct a Lagrange multiplier test, you only need to estimate the restricted model (but the test requires an estimate of the variance-covariance matrix).
- To conduct a Wald test, you only need to estimate the unrestricted model (but the test requires an estimate of the variance-covariance matrix).

All things being equal, the LR test is often the preferred choice for comparing nested models. Econometrics Toolbox has functionality for all three tests.

Likelihood Ratio Test

You can conduct a likelihood ratio test using `lratiotest`. The required inputs are:

- Value of the maximized unrestricted loglikelihood, $l(\theta_A^{MLE})$
- Value of the maximized restricted loglikelihood, $l(\theta_0^{MLE})$
- Number of restrictions (degrees of freedom)

Given these inputs, the likelihood ratio test statistic is

$$G^2 = 2\times\left[l(\theta_A^{MLE}) - l(\theta_0^{MLE}) \right].$$

When estimating conditional mean and variance models (using `arima`, `garch`, `egarch`, or `gjr`), you can return the value of the loglikelihood objective function as an optional output argument of `estimate` or `infer`. For multivariate time series models, you can get the value of the loglikelihood objective function using `vgxvarx`.

Lagrange Multiplier Test

The required inputs for conducting a Lagrange multiplier test are:

- Gradient of the unrestricted likelihood evaluated at the restricted MLEs (the score), S
- Variance-covariance matrix for the unrestricted parameters evaluated at the restricted MLEs, V

Given these inputs, the LM test statistic is

$$LM = S'VS.$$

You can conduct an LM test using `lmtest`. A specific example of an LM test is Engle's ARCH test, which you can conduct using `archtest`.

Wald Test

The required inputs for conducting a Wald test are:

- Restriction function evaluated at the unrestricted MLE, r
- Jacobian of the restriction function evaluated at the unrestricted MLEs, R
- Variance-covariance matrix for the unrestricted parameters evaluated at the unrestricted MLEs, V

Given these inputs, the test statistic for the Wald test is

$$W = r'(RVR')^{-1}r.$$

You can conduct a Wald test using `waldtest`.

Tip You can often compute the Jacobian of the restriction function analytically. Or, if you have Symbolic Math Toolbox™, you can use the function `jacobian`.

Covariance Matrix Estimation

For estimating a variance-covariance matrix, there are several common methods, including:

- Outer product of gradients (OPG). Let G be the matrix of gradients of the loglikelihood function. If your data set has N observations, and there are m parameters in the unrestricted likelihood, then G is an $N \times m$ matrix.

  The matrix $(G'G)^{-1}$ is the OPG estimate of the variance-covariance matrix.

  For `arima`, `garch`, `egarch`, and `gjr` models, the `estimate` method returns the OPG estimate of the variance-covariance matrix.

- Inverse negative Hessian (INH). Given the loglikelihood function $l(\theta)$, the INH covariance estimate has elements

$$\text{cov}(i,j) = \left(-\frac{\partial^2 l(\theta)}{\partial\theta_i \partial\theta_j} \right)^{-1}.$$

  The estimation function for multivariate models, `vgxvarx`, returns the expected Hessian variance-covariance matrix.

Tip If you have Symbolic Math Toolbox, you can use `jacobian` twice to calculate the Hessian matrix for your loglikelihood function.

See Also

arima | `egarch` | `garch` | `gjr` | `lmtest` | `lratiotest` | `waldtest`

Related Examples

- "Conduct a Lagrange Multiplier Test" on page 3-70
- "Conduct a Wald Test" on page 3-74
- "Compare GARCH Models Using Likelihood Ratio Test" on page 3-77

More About

- Using garch Objects
- Using egarch Objects
- Using gjr Objects
- "Goodness of Fit" on page 3-88
- "Information Criteria" on page 3-63
- "Maximum Likelihood Estimation for Conditional Mean Models" on page 5-98
- "Maximum Likelihood Estimation for Conditional Variance Models" on page 6-62

Conduct a Lagrange Multiplier Test

This example shows how to calculate the required inputs for conducting a Lagrange multiplier (LM) test with `lmtest`. The LM test compares the fit of a restricted model against an unrestricted model by testing whether the gradient of the loglikelihood function of the unrestricted model, evaluated at the restricted maximum likelihood estimates (MLEs), is significantly different from zero.

The required inputs for `lmtest` are the score function and an estimate of the unrestricted variance-covariance matrix evaluated at the restricted MLEs. This example compares the fit of an AR(1) model against an AR(2) model.

Step 1. Compute the restricted MLE.

Obtain the restricted MLE by fitting an AR(1) model (with a Gaussian innovation distribution) to the given data. Assume you have presample observations (y_{-1}, y_0) = (9.6249,9.6396).

```
Y = [10.1591; 10.1675; 10.1957; 10.6558; 10.2243; 10.4429;
     10.5965; 10.3848; 10.3972;  9.9478;  9.6402;  9.7761;
     10.0357; 10.8202; 10.3668; 10.3980; 10.2892;  9.6310;
      9.6318;  9.1378;  9.6318;  9.1378];
Y0 = [9.6249; 9.6396];

model = arima(1,0,0);
fit = estimate(model,Y,'Y0',Y0);
```

```
    ARIMA(1,0,0) Model:
    --------------------
    Conditional Probability Distribution: Gaussian

                               Standard          t
     Parameter       Value       Error       Statistic
    -----------   -----------   ------------   -----------
     Constant        3.29993       2.46057        1.34112
        AR{1}       0.670972       0.24635        2.72366
     Variance       0.125064     0.0430152        2.90743
```

When conducting an LM test, only the restricted model needs to be fit.

Step 2. Compute the gradient matrix.

Estimate the variance-covariance matrix for the unrestricted AR(2) model using the outer product of gradients (OPG) method.

For an AR(2) model with Gaussian innovations, the contribution to the loglikelihood function at time t is given by

$$\log L_t = -0.5\log(2\pi\sigma_\varepsilon^2) - \frac{(y_t - c - \phi_1 y_{t-1} - \phi_2 y_{t-2})^2}{2\sigma_\varepsilon^2}$$

where σ_ε^2 is the variance of the innovation distribution.

The contribution to the gradient at time t is

$$\left[\begin{array}{cccc} \frac{\partial \log L_t}{\partial c} & \frac{\partial \log L_t}{\partial \phi_1} & \frac{\partial \log L_t}{\partial \phi_2} & \frac{\partial \log L_t}{\partial \sigma_\varepsilon^2} \end{array}\right],$$

where

$$\begin{aligned} \frac{\partial \log L_t}{\partial c} &= \frac{y_t - c - \phi_1 y_{t-1} - \phi_2 y_{t-2}}{\sigma_\varepsilon^2} \\ \frac{\partial \log L_t}{\partial \phi_1} &= \frac{y_{t-1}(y_t - c - \phi_1 y_{t-1} - \phi_2 y_{t-2})}{\sigma_\varepsilon^2} \\ \frac{\partial \log L_t}{\partial \phi_2} &= \frac{y_{t-2}(y_t - c - \phi_1 y_{t-1} - \phi_2 y_{t-2})}{\sigma_\varepsilon^2} \\ \frac{\partial \log L_t}{\partial \sigma_\varepsilon^2} &= -\frac{1}{2\sigma_\varepsilon^2} + \frac{(y_t - c - \phi_1 y_{t-1} - \phi_2 y_{t-2})^2}{2\sigma_\varepsilon^4} \end{aligned}$$

Evaluate the gradient matrix, G, at the restricted MLEs (using $\hat{\phi}_2 = 0$).

```
c = fit.Constant;
phi1 = fit.AR{1};
phi2 = 0;
sig2 = fit.Variance;

Yt = Y;
Yt1 = [9.6396; Y(1:end-1)];
Yt2 = [9.6249; Yt1(1:end-1)];

N = length(Y);
G = zeros(N,4);
G(:,1) = (Yt-c-phi1*Yt1-phi2*Yt2)/sig2;
G(:,2) = Yt1.*(Yt-c-phi1*Yt1-phi2*Yt2)/sig2;
```

```
G(:,3) = Yt2.*(Yt-c-phi1*Yt1-phi2*Yt2)/sig2;
G(:,4) = -0.5/sig2 + 0.5*(Yt-c-phi1*Yt1-phi2*Yt2).^2/sig2^2;
```

Step 3. Estimate the variance-covariance matrix.

Compute the OPG variance-covariance matrix estimate.

```
V = inv(G'*G)
```

```
V =

    6.1431   -0.6966    0.0827    0.0367
   -0.6966    0.1535   -0.0846   -0.0061
    0.0827   -0.0846    0.0771    0.0024
    0.0367   -0.0061    0.0024    0.0019
```

Step 4. Calculate the score function.

Evaluate the score function (the sum of the individual contributions to the gradient).

```
score = sum(G);
```

Step 5. Conduct the Lagrange multiplier test.

Conduct the Lagrange multiplier test to compare the restricted AR(1) model against the unrestricted AR(2) model. The number of restrictions (the degree of freedom) is one.

```
[h,p,LMstat,crit] = lmtest(score,V,1)
```

```
h =

     0


p =

    0.5787


LMstat =

    0.3084
```

```
crit =

    3.8415
```

The restricted AR(1) model is not rejected in favor of the AR(2) model (`h = 0`).

See Also

arima | estimate | `lmtest`

Related Examples

- "Conduct a Wald Test" on page 3-74
- "Compare GARCH Models Using Likelihood Ratio Test" on page 3-77

More About

- "Model Comparison Tests" on page 3-65
- "Goodness of Fit" on page 3-88
- "Autoregressive Model" on page 5-18

Conduct a Wald Test

This example shows how to calculate the required inputs for conducting a Wald test with `waldtest`. The Wald test compares the fit of a restricted model against an unrestricted model by testing whether the restriction function, evaluated at the unrestricted maximum likelihood estimates (MLEs), is significantly different from zero.

The required inputs for `waldtest` are a restriction function, the Jacobian of the restriction function evaluated at the unrestricted MLEs, and an estimate of the variance-covariance matrix evaluated at the unrestricted MLEs. This example compares the fit of an AR(1) model against an AR(2) model.

Step 1. Compute the unrestricted MLE.

Obtain the unrestricted MLEs by fitting an AR(2) model (with a Gaussian innovation distribution) to the given data. Assume you have presample observations (y_{-1}, y_0) = (9.6249,9.6396)

```
Y = [10.1591; 10.1675; 10.1957; 10.6558; 10.2243; 10.4429;
     10.5965; 10.3848; 10.3972;  9.9478;  9.6402;  9.7761;
     10.0357; 10.8202; 10.3668; 10.3980; 10.2892;  9.6310;
      9.6318;  9.1378;  9.6318;  9.1378];
Y0 = [9.6249; 9.6396];

model = arima(2,0,0);
[fit,V] = estimate(model,Y,'Y0',Y0);
```

```
    ARIMA(2,0,0) Model:
    --------------------
    Conditional Probability Distribution: Gaussian

                               Standard          t
     Parameter       Value       Error       Statistic
    -----------   -----------   ------------   -----------
     Constant       2.88021       2.52387        1.14119
        AR{1}      0.606229       0.40372        1.50161
        AR{2}      0.106309      0.292833       0.363034
     Variance      0.123855     0.0425975        2.90756
```

When conducting a Wald test, only the unrestricted model needs to be fit. `estimate` returns the estimated variance-covariance matrix as an optional output.

Step 2. Compute the Jacobian matrix.

Define the restriction function, and calculate its Jacobian matrix.

For comparing an AR(1) model to an AR(2) model, the restriction function is

$$r(c,\phi_1,\phi_2,\sigma_\varepsilon^2) = \phi_2 - 0 = 0.$$

The Jacobian of the restriction function is

$$\left[\begin{array}{cccc} \frac{\partial r}{\partial c} & \frac{\partial r}{\partial \phi_1} & \frac{\partial r}{\partial \phi_2} & \frac{\partial r}{\partial \sigma_\varepsilon^2} \end{array}\right] = \left[\begin{array}{cccc} 0 & 0 & 1 & 0 \end{array}\right]$$

Evaluate the restriction function and Jacobian at the unrestricted MLEs.

```
r = fit.AR{2};
R = [0 0 1 0];
```

Step 3. Conduct a Wald test.

Conduct a Wald test to compare the restricted AR(1) model against the unrestricted AR(2) model.

```
[h,p,Wstat,crit] = waldtest(r,R,V)
```

```
h =

     0


p =

    0.7166


Wstat =

    0.1318


crit =

    3.8415
```

The restricted AR(1) model is not rejected in favor of the AR(2) model (`h = 0`).

See Also

arima | estimate | `waldtest`

Related Examples

- “Conduct a Lagrange Multiplier Test” on page 3-70
- “Compare GARCH Models Using Likelihood Ratio Test” on page 3-77

More About

- “Model Comparison Tests” on page 3-65
- “Goodness of Fit” on page 3-88
- “Autoregressive Model” on page 5-18

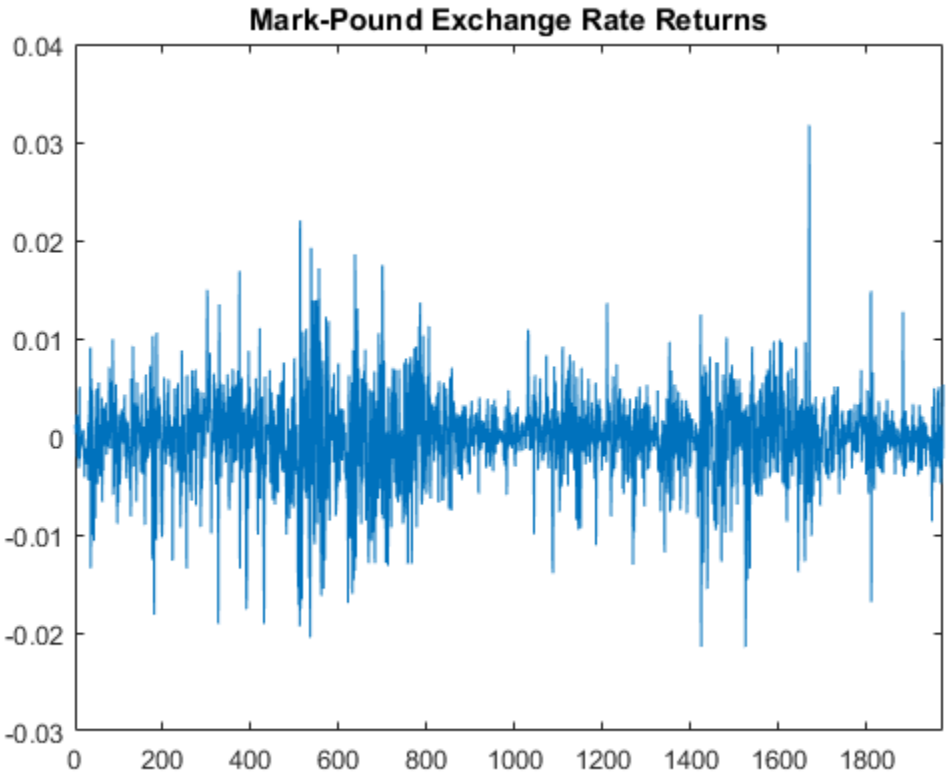
Compare GARCH Models Using Likelihood Ratio Test

This example shows how to conduct a likelihood ratio test to choose the number of lags in a GARCH model.

Load the Data.

Load the Deutschmark/British pound foreign-exchange rate data included with the toolbox. Convert the daily rates to returns.

```
load Data_MarkPound
Y = Data;
r = price2ret(Y);
N = length(r);

figure
plot(r)
xlim([0,N])
title('Mark-Pound Exchange Rate Returns')
```



The daily returns exhibit volatility clustering. Large changes in the returns tend to cluster together, and small changes tend to cluster together. That is, the series exhibits conditional heteroscedasticity.

The returns are of relatively high frequency. Therefore, the daily changes can be small. For numerical stability, it is good practice to scale such data. In this case, scale the returns to percentage returns.

```
r = 100*r;
```

Specify and Fit a GARCH(1,1) Model.

Specify and fit a GARCH(1,1) model (with a mean offset) to the returns series. Return the value of the loglikelihood objective function.

```
model1 = garch('Offset',NaN,'GARCHLags',1,'ARCHLags',1);
[fit1,~,LogL1] = estimate(model1,r);
```

```
    GARCH(1,1) Conditional Variance Model:
    ----------------------------------------
    Conditional Probability Distribution: Gaussian

                                Standard          t
     Parameter       Value        Error       Statistic
    -----------   -----------   ------------   -----------
     Constant      0.0107613    0.00132297        8.13424
     GARCH{1}       0.805974     0.0165603         48.669
      ARCH{1}       0.153134     0.0139737        10.9587
       Offset    -0.00619042    0.00843359       -0.73402
```

Specify and Fit a GARCH(2,1) Model.

Specify and fit a GARCH(2,1) model with a mean offset.

```
model2 = garch(2,1);
model2.Offset = NaN;
[fit2,~,LogL2] = estimate(model2,r);
```

```
    GARCH(2,1) Conditional Variance Model:
    ----------------------------------------
    Conditional Probability Distribution: Gaussian

                                Standard          t
     Parameter       Value        Error       Statistic
    -----------   -----------   ------------   -----------
     Constant      0.0112262      0.001538        7.29921
     GARCH{1}       0.489644      0.111593        4.38776
     GARCH{2}       0.297688      0.102181        2.91333
      ARCH{1}       0.168419     0.0165832         10.156
       Offset     -0.0049837    0.00847645      -0.587947
```

Conduct a Likelihood Ratio Test.

Conduct a likelihood ratio test to compare the restricted GARCH(1,1) model fit to the unrestricted GARCH(2,1) model fit. The degree of freedom for this test is one (the number of restrictions).

```
[h,p] = lratiotest(LogL2,LogL1,1)
```

```
h =

     1


p =

    0.0218
```

At the 0.05 significance level, the null GARCH(1,1) model is rejected (`h = 1`) in favor of the unrestricted GARCH(2,1) alternative.

See Also

`estimate` | `garch` | `lratiotest`

Related Examples

- "Conduct a Lagrange Multiplier Test" on page 3-70
- "Conduct a Wald Test" on page 3-74
- "Compare Conditional Variance Models Using Information Criteria" on page 6-87

More About

- Using garch Objects
- "Model Comparison Tests" on page 3-65
- "Goodness of Fit" on page 3-88
- "GARCH Model" on page 6-3

Check Fit of Multiplicative ARIMA Model

This example shows how to do goodness of fit checks. Residual diagnostic plots help verify model assumptions, and cross-validation prediction checks help assess predictive performance. The time series is monthly international airline passenger numbers from 1949 to 1960.

Load the data and estimate a model.

Load the airline data set.

```
load(fullfile(matlabroot,'examples','econ','Data_Airline.mat'))
y = log(Data);
T = length(y);

Mdl = arima('Constant',0,'D',1,'Seasonality',12,...
             'MALags',1,'SMALags',12);
EstMdl = estimate(Mdl,y);
```

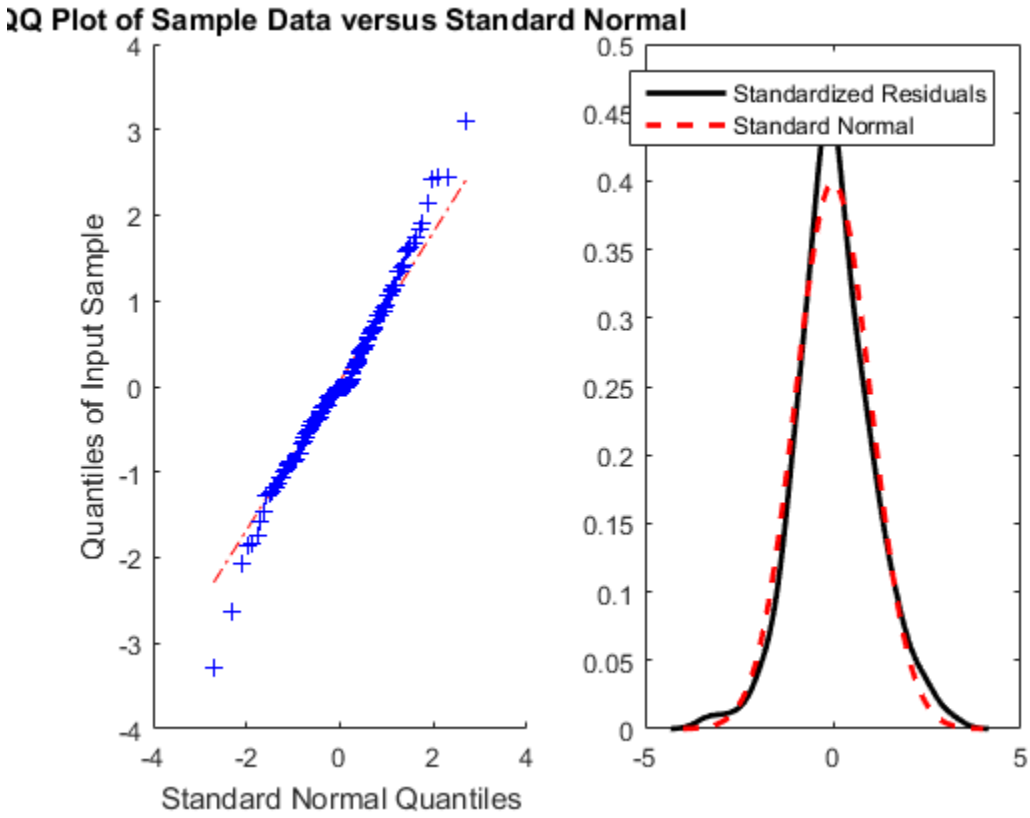
```
    ARIMA(0,1,1) Model Seasonally Integrated with Seasonal MA(12):
    --------------------------------------------------------------
    Conditional Probability Distribution: Gaussian

                               Standard          t
     Parameter       Value       Error       Statistic
    -----------   -----------   ------------   -----------
     Constant              0         Fixed          Fixed
        MA{1}      -0.377162     0.0667944       -5.64661
      SMA{12}      -0.572378     0.0854395       -6.69923
     Variance     0.00126337    0.00012395        10.1926
```

Check the residuals for normality.

One assumption of the fitted model is that the innovations follow a Gaussian distribution. Infer the residuals, and check them for normality.

```
res = infer(EstMdl,y);
stres = res/sqrt(EstMdl.Variance);

figure
subplot(1,2,1)
qqplot(stres)
```

```
x = -4:.05:4;
[f,xi] = ksdensity(stres);
subplot(1,2,2)
plot(xi,f,'k','LineWidth',2);
hold on
plot(x,normpdf(x),'r--','LineWidth',2)
legend('Standardized Residuals','Standard Normal')
hold off
```



The quantile-quantile plot (QQ-plot) and kernel density estimate show no obvious violations of the normality assumption.

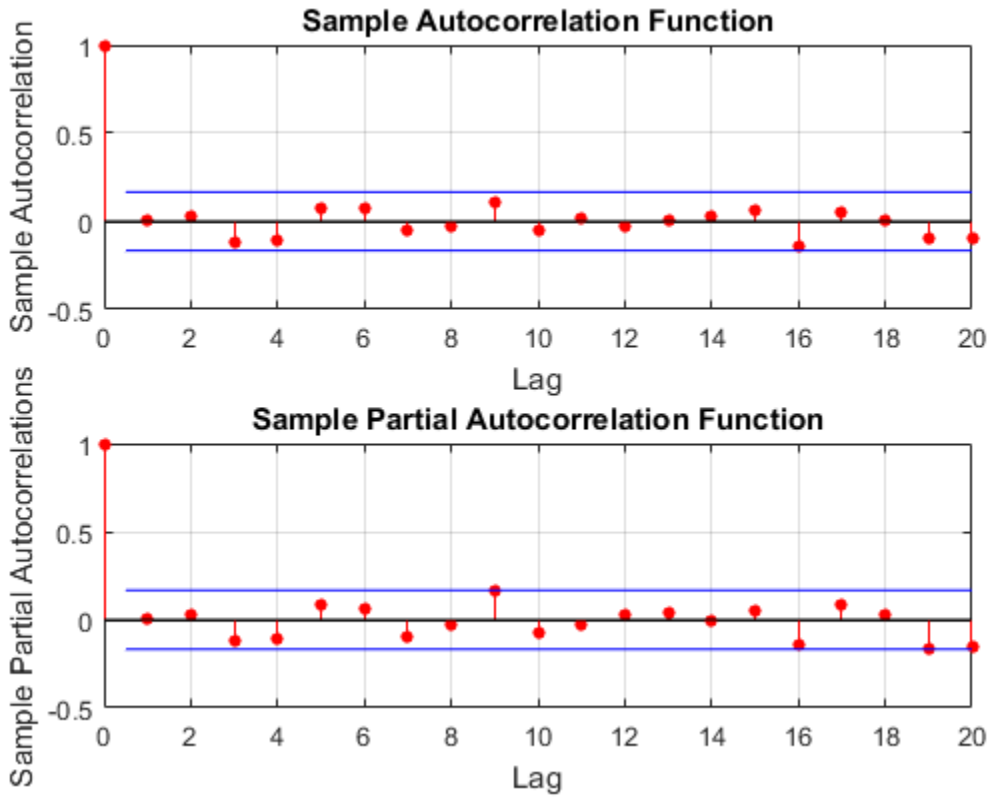
Check the residuals for autocorrelation.

Confirm that the residuals are uncorrelated. Look at the sample autocorrelation function (ACF) and partial autocorrelation function (PACF) plots for the standardized residuals.

```
figure
subplot(2,1,1)
autocorr(stres)
subplot(2,1,2)
parcorr(stres)
```

```
[h,p] = lbqtest(stres,'lags',[5,10,15],'dof',[3,8,13])
```

```
h =

     0     0     0


p =

    0.1842    0.3835    0.7321
```



The sample ACF and PACF plots show no significant autocorrelation. More formally, conduct a Ljung-Box Q-test at lags 5, 10, and 15, with degrees of freedom 3, 8, and 13, respectively. The degrees of freedom account for the two estimated moving average coefficients.

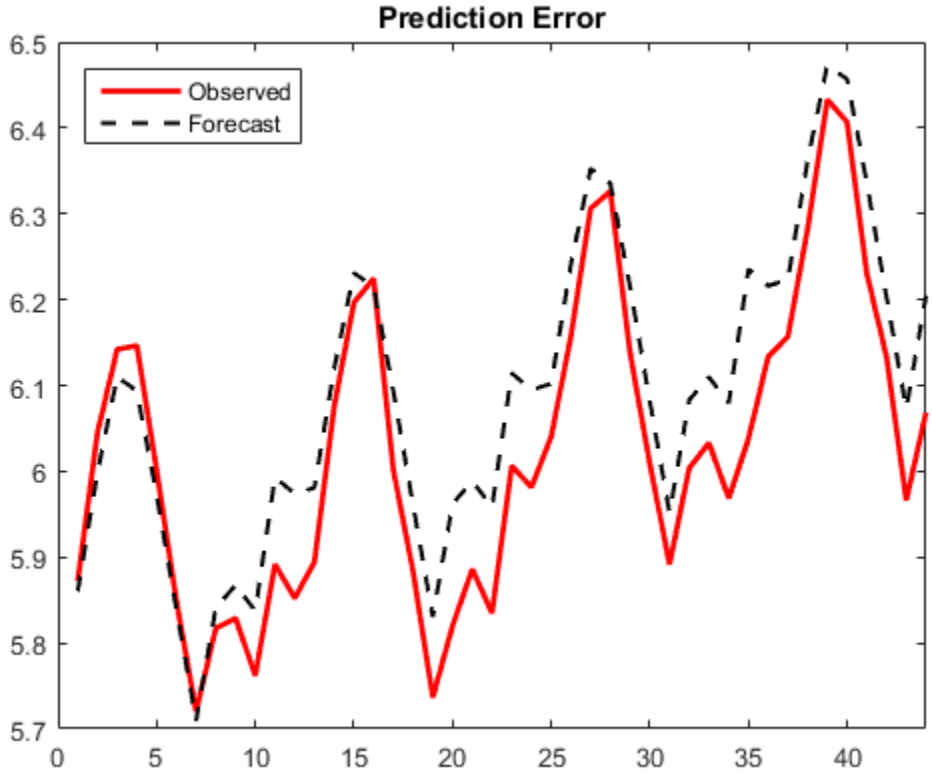
The Ljung-Box Q-test confirms the sample ACF and PACF results. The null hypothesis that all autocorrelations are jointly equal to zero up to the tested lag is not rejected (`h = 0`) for any of the three lags.

Check predictive performance.

Use a holdout sample to compute the predictive MSE of the model. Use the first 100 observations to estimate the model, and then forecast the next 44 periods.

```
y1 = y(1:100);
y2 = y(101:end);

Mdl1 = estimate(Mdl,y1);
yF1 = forecast(Mdl1,44,'Y0',y1);
pmse = mean((y2-yF1).^2)

figure
plot(y2,'r','LineWidth',2)
hold on
plot(yF1,'k--','LineWidth',1.5)
xlim([0,44])
title('Prediction Error')
legend('Observed','Forecast','Location','NorthWest')
hold off
```

```
    ARIMA(0,1,1) Model Seasonally Integrated with Seasonal MA(12):
    --------------------------------------------------------------
    Conditional Probability Distribution: Gaussian

                                Standard          t
     Parameter       Value          Error       Statistic
    -----------   -----------   ------------   -----------
     Constant             0         Fixed          Fixed
        MA{1}      -0.356736      0.089461       -3.98762
       SMA{12}     -0.633186     0.0987442       -6.41239
     Variance     0.00132855   0.000158823        8.36497

pmse =

    0.0069
```



The predictive ability of the model is quite good. You can optionally compare the PMSE for this model with the PMSE for a competing model to help with model selection.

See Also

arima | `autocorr` | estimate | forecast | infer | `lbqtest` | `parcorr`

Related Examples

- "Specify Multiplicative ARIMA Model" on page 5-52
- "Estimate Multiplicative ARIMA Model" on page 5-113
- "Simulate Multiplicative ARIMA Models" on page 5-169

- “Forecast Multiplicative ARIMA Model” on page 5-192
- “Detect Autocorrelation” on page 3-18

More About

- “Goodness of Fit” on page 3-88
- “Residual Diagnostics” on page 3-90
- “Check Predictive Performance” on page 3-92
- “MMSE Forecasting of Conditional Mean Models” on page 5-182
- “Autocorrelation and Partial Autocorrelation” on page 3-13
- “Ljung-Box Q-Test” on page 3-16

Goodness of Fit

After specifying a model and estimating its parameters, it is good practice to perform goodness-of-fit checks to diagnose the adequacy of your fitted model. When assessing model adequacy, areas of primary concern are:

- Violations of model assumptions, potentially resulting in bias and inaccurate standard errors
- Poor predictive performance
- Missing explanatory variables

Goodness-of-fit checks can help you identify areas of model inadequacy. They can also suggest ways to improve your model. For example, if you conduct a test for residual autocorrelation and get a significant result, you might be able to improve your model fit by adding additional autoregressive or moving average terms.

Some strategies for evaluating goodness of fit are:

- Compare your model against an augmented alternative. Make comparisons, for example, by conducting a likelihood ratio test. Testing your model against a more elaborate alternative model is a way to assess evidence of inadequacy. Give careful thought when choosing an alternative model.
- Making residual diagnostic plots is an informal—but useful—way to assess violation of model assumptions. You can plot residuals to check for normality, residual autocorrelation, residual heteroscedasticity, and missing predictors. Formal tests for autocorrelation and heteroscedasticity can also help quantify possible model violations.
- Predictive performance checks. Divide your data into two parts: a training set and a validation set. Fit your model using only the training data, and then forecast the fitted model over the validation period. By comparing model forecasts against the true, holdout observations, you can assess the predictive performance of your model. Prediction mean square error (PMSE) can be calculated as a numerical summary of the predictive performance. When choosing among competing models, you can look at their respective PMSE values to compare predictive performance.

Related Examples

- “Box-Jenkins Model Selection” on page 3-4
- “Check Fit of Multiplicative ARIMA Model” on page 3-81

- “Compare GARCH Models Using Likelihood Ratio Test” on page 3-77

More About

- “Residual Diagnostics” on page 3-90
- “Model Comparison Tests” on page 3-65
- “Check Predictive Performance” on page 3-92

Residual Diagnostics

In this section...

Check Residuals for Normality

A common assumption of time series models is a Gaussian innovation distribution. After fitting a model, you can infer residuals and check them for normality. If the Gaussian innovation assumption holds, the residuals should look approximately normally distributed.

Some plots for assessing normality are:

- Histogram
- Box plot
- Quantile-quantile plot
- Kernel density estimate

The last three plots are in Statistics and Machine Learning Toolbox.

If you see that your standardized residuals have excess kurtosis (fatter tails) compared to a standard normal distribution, you can consider using a Student's t innovation distribution.

Check Residuals for Autocorrelation

In time series models, the innovation process is assumed to be uncorrelated. After fitting a model, you can infer residuals and check them for any unmodeled autocorrelation.

As an informal check, you can plot the sample autocorrelation function (ACF) and partial autocorrelation function (PACF). If either plot shows significant autocorrelation in the residuals, you can consider modifying your model to include additional autoregression or moving average terms.

More formally, you can conduct a Ljung-Box Q-test on the residual series. This tests the null hypothesis of jointly zero autocorrelations up to lag m, against the alternative of at least one nonzero autocorrelation. You can conduct the test at several values of m. The degrees of freedom for the Q-test are usually m. However, for testing a residual series, you should use degrees of freedom $m - p - q$, where p and q are the number of AR and MA coefficients in the fitted model, respectively.

Check Residuals for Conditional Heteroscedasticity

A white noise innovation process has constant variance. After fitting a model, you can infer residuals and check them for heteroscedasticity (nonconstant variance).

As an informal check, you can plot the sample ACF and PACF of the squared residual series. If either plot shows significant autocorrelation, you can consider modifying your model to include a conditional variance process.

More formally, you can conduct an Engle's ARCH test on the residual series. This tests the null hypothesis of no ARCH effects against the alternative ARCH model with k lags.

See Also

`archtest` | `autocorr` | `boxplot` | `histogram` | `ksdensity` | `lbqtest` | `parcorr` | `qqplot`

Related Examples

- "Box-Jenkins Model Selection" on page 3-4
- "Detect Autocorrelation" on page 3-18
- "Detect ARCH Effects" on page 3-28
- "Check Fit of Multiplicative ARIMA Model" on page 3-81

More About

- "Goodness of Fit" on page 3-88
- "Check Predictive Performance" on page 3-92
- "Ljung-Box Q-Test" on page 3-16
- "Engle's ARCH Test" on page 3-25
- "Autocorrelation and Partial Autocorrelation" on page 3-13

Check Predictive Performance

If you plan to use a fitted model for forecasting, it is good practice to assess the predictive ability of the model. Models that fit well in-sample are not guaranteed to forecast well. For example, overfitting can lead to good in-sample fit, but poor predictive performance.

When checking predictive performance, it is important to not use your data twice. That is, the data you use to fit your model should be different than the data you use to assess forecasts. You can use cross validation to evaluate out-of-sample forecasting ability:

1. Divide your time series into two parts: a training set and a validation set.
2. Fit a model to your training data.
3. Forecast the fitted model over the validation period.
4. Compare the forecasts to the holdout validation observations using plots and numerical summaries (such as predictive mean square error).

Prediction mean square error (PMSE) measures the discrepancy between model forecasts and observed data. Suppose you have a time series of length N, and you set aside M validation points, denoted $y_1^v, y_2^v, \ldots, y_M^v$. After fitting your model to the first $N - M$ data points (the training set), generate forecasts $\hat{y}_1^v, \hat{y}_2^v, \ldots, \hat{y}_M^v$.

The model PMSE is calculated as

$$PMSE = \frac{1}{M}\sum_{i=1}^{M}\left(y_i^v - \hat{y}_i^v\right)^2.$$

You can calculate PMSE for various choices of M to verify the robustness of your results.

Related Examples

- “Check Fit of Multiplicative ARIMA Model” on page 3-81

More About

- “Goodness of Fit” on page 3-88
- “Residual Diagnostics” on page 3-90

- "MMSE Forecasting of Conditional Mean Models" on page 5-182
- "MMSE Forecasting of Conditional Variance Models" on page 6-117

Nonspherical Models

What Are Nonspherical Models?

Consider the linear time series model $y_t = X_t\beta + \varepsilon_t$, where y_t is the response, x_t is a vector of values for the r predictors, β is the vector of regression coefficients, and ε_t is the random innovation at time t.

Ordinary least squares (OLS) estimation and inference techniques for this framework depend on certain assumptions, e.g., homoscedastic and uncorrelated innovations. For more details on the classical linear model, see "Time Series Regression I: Linear Models". If your data exhibits signs of assumption violations, then OLS estimates or inferences based on them might not be valid.

In particular, if the data is generated with an innovations process that exhibits autocorrelation or heteroscedasticity, then the model (or the residuals) are *nonspherical*. These characteristics are often detected through testing of model residuals (for details, see "Time Series Regression VI: Residual Diagnostics").

Nonspherical residuals are often considered a sign of model misspecification, and models are revised to whiten the residuals and improve the reliability of standard estimation techniques. In some cases, however, nonspherical models must be accepted as they are, and estimated as accurately as possible using revised techniques. Cases include:

- Models presented by theory
- Models with predictors that are dictated by policy
- Models without available data sources, for which predictor proxies must be found

A variety of alternative estimation techniques have been developed to deal with these situations.

Related Examples

- "Classical Model Misspecification Tests"
- "Time Series Regression I: Linear Models"
- "Time Series Regression VI: Residual Diagnostics"
- "Plot a Confidence Band Using HAC Estimates" on page 3-95
- "Change the Bandwidth of a HAC Estimator" on page 3-105

Plot a Confidence Band Using HAC Estimates

This example shows how to plot heteroscedastic-and-autocorrelation consistent (HAC) corrected confidence bands using Newey-West robust standard errors.

One way to estimate the coefficients of a linear model is by OLS. However, time series models tend to have innovations that are autocorrelated and heteroscedastic (i.e., the errors are nonspherical). If a times series model has nonspherical errors, then usual formulae for standard errors of OLS coefficients are biased and inconsistent. Inference based on these inefficient standard errors tends to inflate the Type I error rate. One way to account for nonspherical errors is to use HAC standard errors. In particular, the Newey-West estimator of the OLS coefficient covariance is relatively robust against nonspherical errors.

Load the Data

Load the Canadian electric power consumption data set from the World Bank. The response is Canada's electrical energy consumption in kWh (`consump`), the predictor is Canada's GDP in year 2000 USD, and the data set also contains two other variables: year (`year`) and the GDP defaltor (`gdpDeflator`).

```
load(fullfile(matlabroot,'examples','econ','Data_PowerConsumption.mat'));
consump = Data(:,4);
gdp = Data(:,3);
gdpDeflator = Data(:,2);
year = Data(2:end,1);
```

Define the Model

Model the behavior of the annual difference in electrical energy consumption with respect to real GDP as a linear model:

$$\mathtt{comsumpDiff}_t = \beta_0 + \beta_1 \mathtt{rGDP}_t + \varepsilon_t.$$

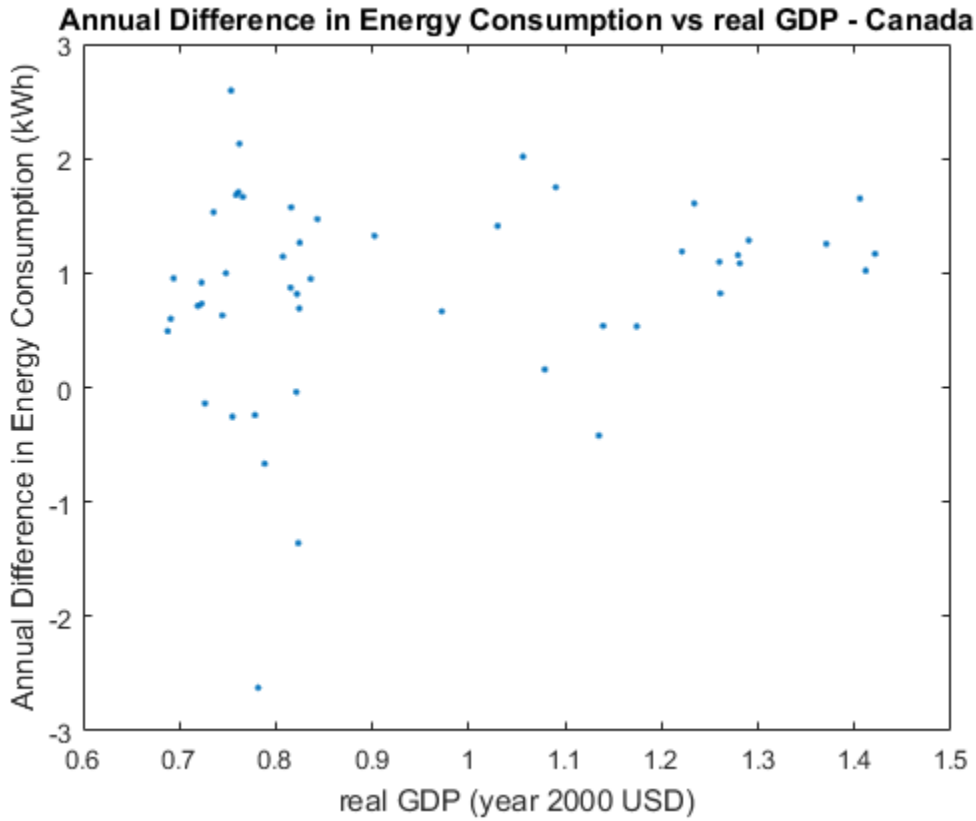
```
consumpDiff = consump - lagmatrix(consump,1); ...
    % Annual difference in consumption
T = size(consumpDiff,1);
consumpDiff = consumpDiff(2:end)/1.0e+10;
    % Scale for numerical stability
rGDP = gdp./(gdpDeflator);    % Deflate GDP
rGDP = rGDP(2:end)/1.0e+10;   % Scale for numerical stability
rGDPdes = [ones(T-1,1) rGDP]; % Design matrix

Mdl = fitlm(rGDP,consumpDiff);
```

```
coeff = Mdl.Coefficients(:,1);
EstParamCov = Mdl.CoefficientCovariance;
resid = Mdl.Residuals.Raw;
```

Plot the Data

Plot the difference in energy consumption, `consumpDiff` versus the real GDP, to check for possible heteroscedasticity.

```
figure
plot(rGDP,consumpDiff,'.')
title 'Annual Difference in Energy Consumption vs real GDP - Canada';
xlabel 'real GDP (year 2000 USD)';
ylabel 'Annual Difference in Energy Consumption (kWh)';
```

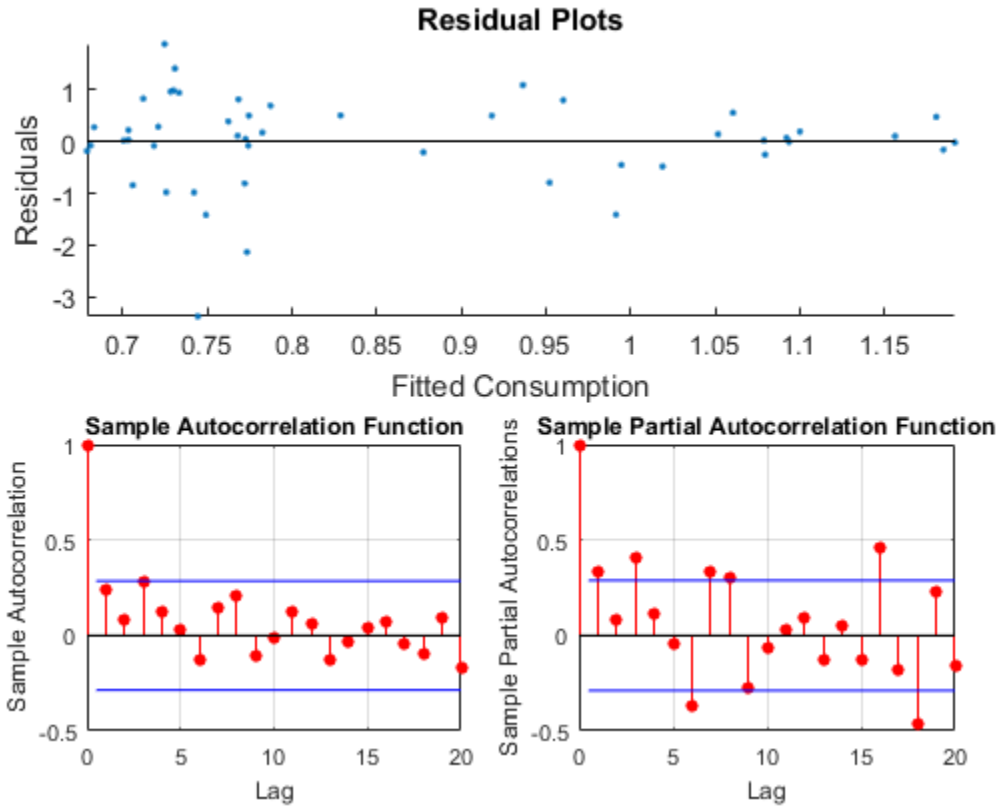


The figure indicates that heteroscedasticity might be present in the annual difference in energy consumption. As real GDP increases, the annual difference in energy consumption seems to be less variable.

Plot the residuals.

Plot the residuals from `Mdl` against the fitted values and year to assess heteroscedasticity and autocorrelation.

```
figure
subplot(2,1,1)
hold on
plot(Mdl.Fitted,resid,'.')
plot([min(Mdl.Fitted) max(Mdl.Fitted)],[0 0],'k-')
title 'Residual Plots';
xlabel 'Fitted Consumption';
ylabel 'Residuals';
axis tight
hold off
subplot(2,2,3)
autocorr(resid)
h1 = gca;
h1.FontSize = 8;
subplot(2,2,4)
parcorr(resid)
h2 = gca;
h2.FontSize = 8;
```



The residual plot reveals decreasing residual variance with increasing fitted consumption. The autocorrelation function shows that autocorrelation might be present in the first few lagged residuals.

Test for heteroscedasticity and autocorrelation.

Test for conditional heteroscedasticity using Engle's ARCH test. Test for autocorrelation using the Ljung-Box Q test. Test for overall correlation using the Durbin-Watson test.

```
[~,englePValue] = archtest(resid);
englePValue
[~,lbqPValue] = lbqtest(resid,'lags',1:3);...
    % Significance of first three lags
lbqPValue
```

```
[dwPValue] = dwtest(Mdl);
dwPValue
```

```
englePValue =

    0.1463


lbqPValue =

    0.0905    0.1966    0.0522


dwPValue =

    0.0013
```

The p value of Engle's ARCH test suggests significant conditional heteroscedasticity at 15% significance level. The p value for the Ljung-Box Q test suggests significant autocorrelation with the first and third lagged residuals at 10% significance level. The p value for the Durbin-Watson test suggests that there is strong evidence for overall residual autocorrelation. The results of the tests suggest that the standard linear model conditions of homoscedasticity and uncorrelated errors are violated, and inferences based on the OLS coefficient covariance matrix are suspect.

One way to proceed with inference (such as constructing a confidence band) is to correct the OLS coefficient covariance matrix by estimating the Newey-West coefficient covariance.

Estimate the Newey-West coefficient covariance.

Correct the OLS coefficient covariance matrix by estimating the Newey-West coefficient covariance using `hac`. Compute the maximum lag to be weighted for the standard Newey-West estimate, `maxLag` (Newey and West, 1994). Use `hac` to estimate the standard Newey-West coefficient covariance.

```
maxLag = floor(4*(T/100)^(2/9));
[NWEstParamCov,~,NWCoeff] = hac(Mdl,'type','hac',...
    'bandwidth',maxLag + 1);
```

```
Estimator type: HAC
Estimation method: BT
```

```
Bandwidth: 4.0000
Whitening order: 0
Effective sample size: 49
Small sample correction: on

Coefficient Covariances:

       |  Const      x1
-------------------------
 Const |  0.3720  -0.2990
 x1    | -0.2990   0.2454
```

The Newey-West standard error for the coefficient of `rGDP`, labeled x_1 in the table, is less than the usual OLS standard error. This suggests that, in this data set, correcting for residual heteroscedasticity and autocorrelation increases the precision in measuring the linear effect of real GDP on energy consumption.

Calculate the Working-Hotelling confidence bands.

Compute the 95% Working-Hotelling confidence band for each covariance estimate using `nlpredci` (Kutner et al., 2005).

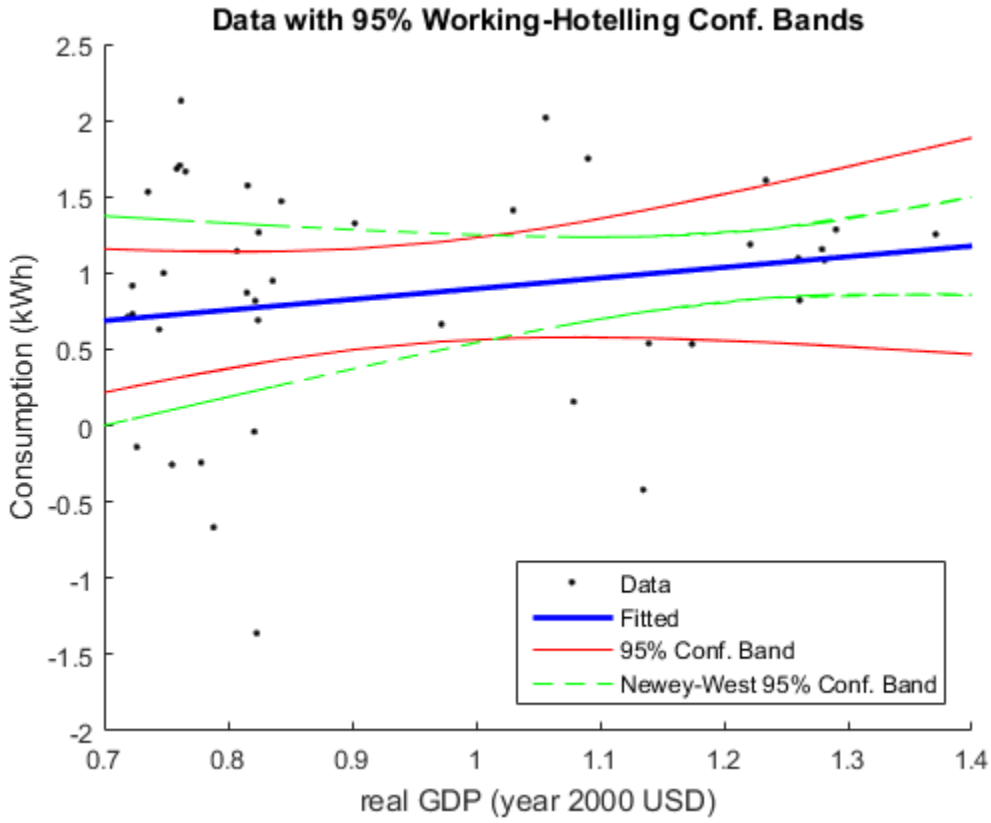
```
modelfun = @(b,x)(b(1)*x(:,1)+b(2)*x(:,2));
    % Define the linear model
[beta,nlresid,~,EstParamCov] = nlinfit(rGDPdes,...
    consumpDiff,modelfun,[1,1]); % estimate the model
[fity,fitcb] = nlpredci(modelfun,rGDPdes,beta,nlresid,...
    'Covar',EstParamCov,'SimOpt','on');
    % Margin of errors
conbandnl = [fity - fitcb fity + fitcb];
    % Confidence bands
[fity,NWfitcb] = nlpredci(modelfun,rGDPdes,...
    beta,nlresid,'Covar',NWEstParamCov,'SimOpt','on');
    % Corrected margin of error
NWconbandnl = [fity - NWfitcb fity + NWfitcb];
    % Corrected confidence bands
```

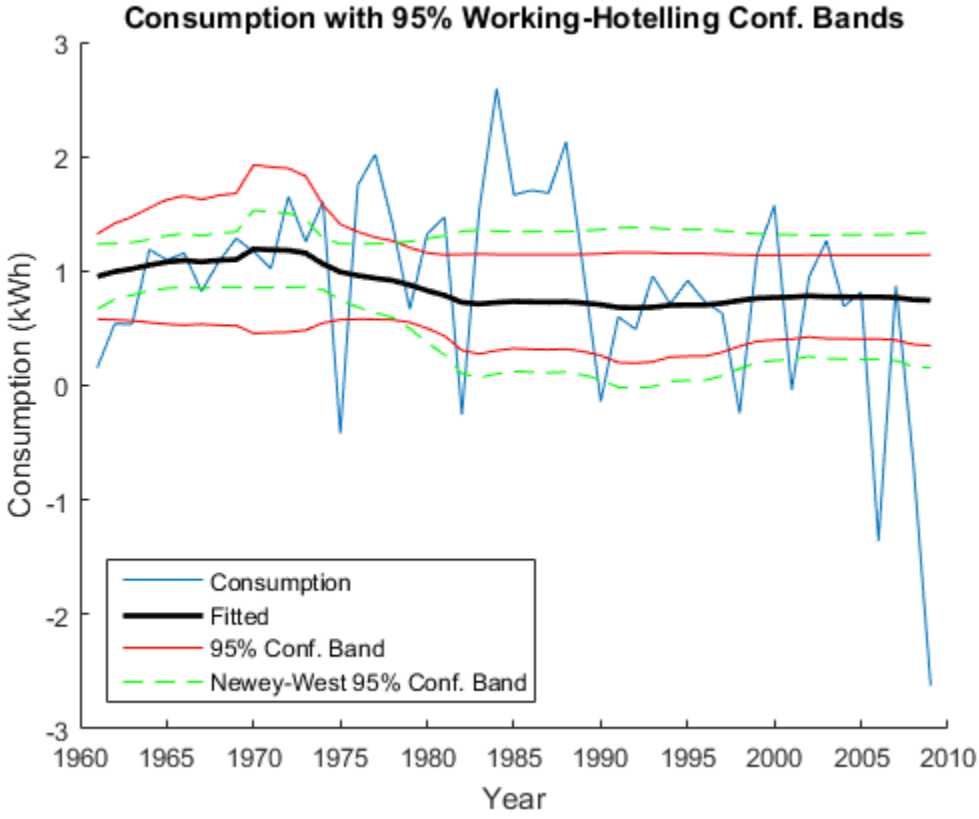
Plot the Working-Hotelling confidence bands.

Plot the Working-Hotelling confidence bands on the same axes twice: one plot displaying electrical energy consumption with respect to real GDP, and the other displaying the electrical energy consumption time series.

```
figure
hold on
```

```
l1 = plot(rGDP,consumpDiff,'k.');
l2 = plot(rGDP,fity,'b-','LineWidth',2);
l3 = plot(rGDP,conbandnl,'r-');
l4 = plot(rGDP,NWconbandnl,'g--');
title 'Data with 95% Working-Hotelling Conf. Bands';
xlabel 'real GDP (year 2000 USD)';
ylabel 'Consumption (kWh)';
axis([0.7 1.4 -2 2.5])
legend([l1 l2 l3(1) l4(1)],'Data','Fitted','95% Conf. Band',...
    'Newey-West 95% Conf. Band','Location','SouthEast')
hold off

figure
hold on
l1 = plot(year,consumpDiff);
l2 = plot(year,fity,'k-','LineWidth',2);
l3 = plot(year,conbandnl,'r-');
l4 = plot(year,NWconbandnl,'g--');
title 'Consumption with 95% Working-Hotelling Conf. Bands';
xlabel 'Year';
ylabel 'Consumption (kWh)';
legend([l1 l2 l3(1) l4(1)],'Consumption','Fitted',...
    '95% Conf. Band','Newey-West 95% Conf. Band',...
    'Location','SouthWest')
hold off
```


Data with 95% Working-Hotelling Conf. Bands
Consumption (kWh)
real GDP (year 2000 USD)
Data
Fitted
95% Conf. Band
Newey-West 95% Conf. Band
2.5
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
0.7
0.8
0.9
1
1.1
1.2
1.3
1.4



The plots show that the Newey-West estimator accounts for the heteroscedasticity in that the confidence band is wide in areas of high volatility, and thin in areas of low volatility. The OLS coefficient covariance estimator ignores this pattern of volatility.

References:

1 Kutner, M. H., C. J. Nachtsheim, J. Neter, and W. Li. Applied Linear Statistical Models. 5th Ed. New York: McGraw-Hill/Irwin, 2005.

2 Newey, W. K., and K. D. West. "A Simple Positive Semidefinite, Heteroskedasticity and Autocorrelation Consistent Covariance Matrix." *Econometrica*. Vol. 55, 1987, pp. 703-708.

3 Newey, W. K, and K. D. West. "Automatic Lag Selection in Covariance Matrix Estimation." *The Review of Economic Studies*. Vol. 61 No. 4, 1994, pp. 631-653.

Related Examples

- "Time Series Regression I: Linear Models"
- "Time Series Regression VI: Residual Diagnostics"
- "Change the Bandwidth of a HAC Estimator" on page 3-105

More About

- "Nonspherical Models" on page 3-94

Change the Bandwidth of a HAC Estimator

This example shows how to change the bandwidth when estimating a HAC coefficient covariance, and compare estimates over varying bandwidths and kernels.

How does the bandwidth affect HAC estimators? If you change it, are there large differences in the estimates, and, if so, are the differences practically significant? Explore bandwidth effects by estimating HAC coefficient covariances over a grid of bandwidths.

Load and Plot the Data.

Determine how the cost of living affects the behavior of nominal wages. Load the Nelson Plosser data set to explore their statistical relationship.

```
load Data_NelsonPlosser
isNaN = any(ismissing(DataTable),2);       % Flag periods containing NaNs
cpi = DataTable.CPI(~isNaN); % Cost of living
wm = DataTable.WN(~isNaN);             % Nominal wages

figure
plot(cpi,wm,'o')
hFit = lsline; % Regression line
xlabel('Consumer Price Index (1967 = 100)')
ylabel('Nominal Wages (current $)')
legend(hFit,'OLS Line','Location','SE')
title('{\bf Cost of Living}')
grid on
```

The plot suggests that a linear model might capture the relationship between the two variables.

Define the Model.

Model the behavior of nominal wages with respect to CPI as this linear model.

$$\mathtt{wm}_t = \beta_0 + \beta_1 \mathtt{cpi}_t + \varepsilon_t$$

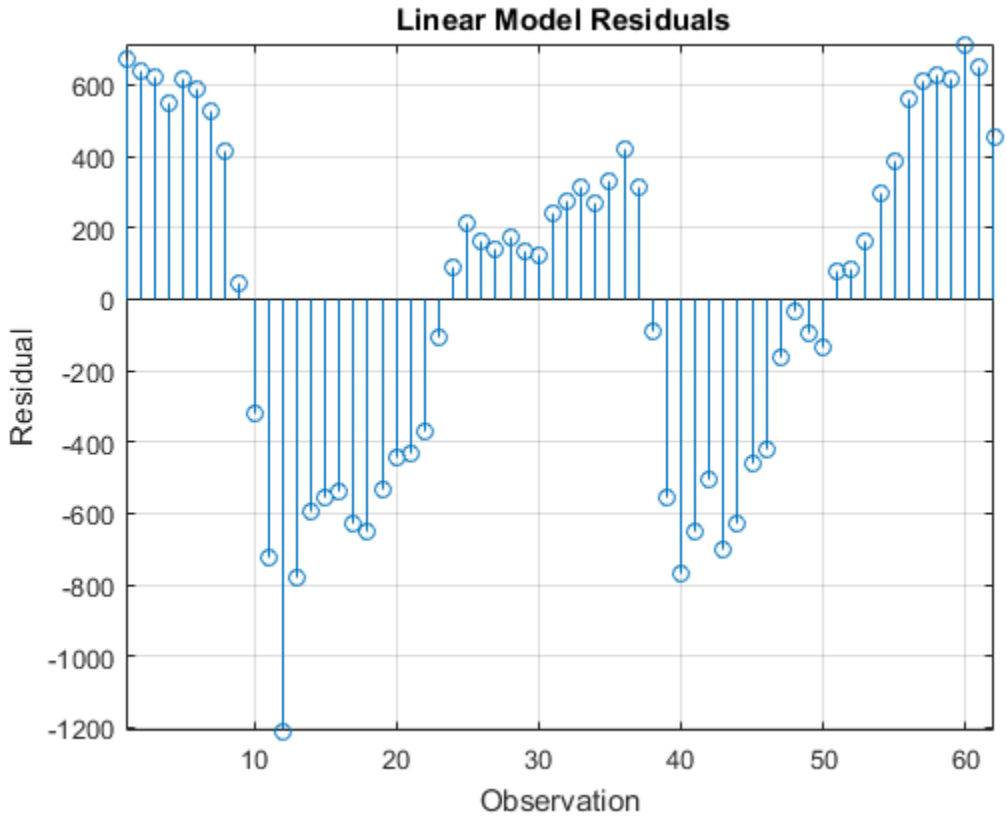
```
Mdl = fitlm(cpi,wm)
coeffCPI = Mdl.Coefficients.Estimate(2);
seCPI = Mdl.Coefficients.SE(2);
```

```
Mdl =


Linear regression model:
    y ~ 1 + x1

Estimated Coefficients:
                   Estimate      SE       tStat      pValue
                   ________    ______    _______    _________

    (Intercept)    -2541.5     174.64    -14.553    2.407e-21
    x1              88.041     2.6784     32.871    4.507e-40


Number of observations: 62, Error degrees of freedom: 60
Root Mean Squared Error: 494
R-squared: 0.947,  Adjusted R-Squared 0.947
F-statistic vs. constant model: 1.08e+03, p-value = 4.51e-40
```

Plot Residuals.

Plot the residuals from `Mdl` against the fitted values to assess heteroscedasticity and autocorrelation.

```
figure;
stem(Mdl.Residuals.Raw);
xlabel('Observation');
ylabel('Residual');
title('{\bf Linear Model Residuals}');
axis tight;
grid on;
```



The residual plot shows varying levels of dispersion, which indicates heteroscedasticity. Neighboring residuals (with respect to observation) tend to have the same sign and magnitude, which indicates the presence of autocorrelation.

Estimate HAC standard errors.

Obtain HAC standard errors over varying bandwidths using the Bartlett (for the Newey-West estimate) and quadratic spectral kernels.

```
numEstimates = 10;
stdErrBT = zeros(numEstimates,1);
stdErrQS = zeros(numEstimates,1);
for bw = 1:numEstimates
    [~,seBT] = hac(cpi,wm,'bandwidth',bw,'display','off'); ...
```

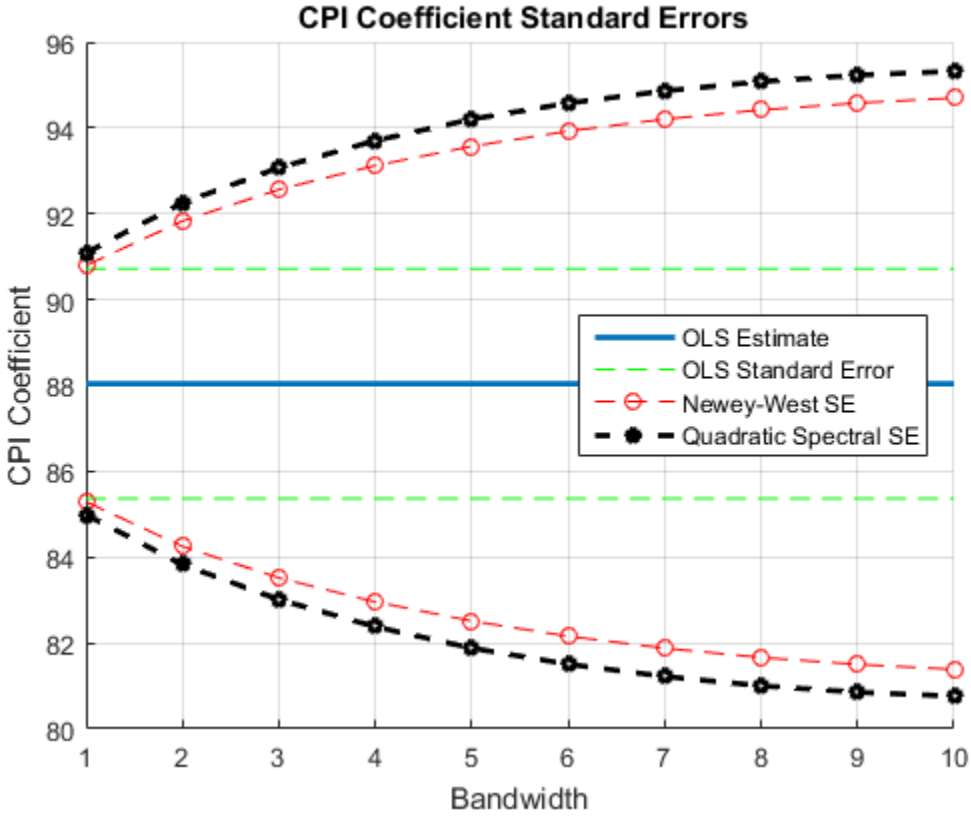
```
        % Newey-West
    [~,seQS] = hac(cpi,wm,'weights','QS','bandwidth',bw, ...
        'display','off'); % HAC using quadratic spectral kernel
    stdErrBT(bw) = seBT(2);
    stdErrQS(bw) = seQS(2);
end
```

You can increase `numEstimates` to discover how increasing bandwidths affect the HAC estimates.

Plot the standard errors.

Visually compare the Newey-West standard errors of $\hat{\beta}_1$ to those using the quadratic spectral kernel over the bandwidth grid.

```
figure
hold on
hCoeff = plot(1:numEstimates,repmat(coeffCPI,numEstimates, ...
    1),'LineWidth',2);
hOLS = plot(1:numEstimates,repmat(coeffCPI+seCPI, ...
    numEstimates,1),'g--');
plot(1:numEstimates,repmat(coeffCPI-seCPI,numEstimates,1),'g--')
hBT = plot(1:numEstimates,coeffCPI+stdErrBT,'ro--');
plot(1:numEstimates,coeffCPI-stdErrBT,'ro--')
hQS = plot(1:numEstimates,coeffCPI+stdErrQS,'kp--',...
    'LineWidth',2);
plot(1:numEstimates,coeffCPI-stdErrQS,'kp--','LineWidth',2)
hold off
xlabel('Bandwidth')
ylabel('CPI Coefficient')
legend([hCoeff,hOLS,hBT,hQS],{'OLS Estimate', ...
    'OLS Standard Error','Newey-West SE', ...
    'Quadratic Spectral SE'},'Location','E')
title('{\bf CPI Coefficient Standard Errors}')
grid on
```



The plot suggests that, for this data set, accounting for heteroscedasticity and autocorrelation using either HAC estimate results in more conservative intervals than the usual OLS standard error. The precision of the HAC estimates decreases as the bandwidth increases along the defined grid.

For this data set, the Newey-West estimates are slightly more precise than those using the quadratic spectral kernel. This might be because the latter captures heteroscedasticity and autocorrelation better than the former.

References:

1 Andrews, D. W. K. "Heteroskedasticity and Autocorrelation Consistent Covariance Matrix Estimation." *Econometrica*. Vol. 59, 1991, pp. 817-858.

2 Newey, W. K., and K. D. West. "A Simple, Positive Semi-definite, Heteroskedasticity and Autocorrelation Consistent Covariance Matrix." *Econometrica*. Vol. 55, No. 3, 1987, pp. 703-708.\

3 Newey, W. K., and K. D. West. "Automatic Lag Selection in Covariance Matrix Estimation." *The Review of Economic Studies*. Vol. 61, No. 4, 1994, pp. 631-653.

Related Examples

- "Classical Model Misspecification Tests"
- "Time Series Regression I: Linear Models"
- "Time Series Regression VI: Residual Diagnostics"
- "Plot a Confidence Band Using HAC Estimates" on page 3-95

More About

- "Nonspherical Models" on page 3-94

Check Model Assumptions for Chow Test

This example shows how to check the model assumptions for a Chow test. The model is of U.S. gross domestic product (GDP), with consumer price index (CPI) and paid compensation of employees (COE) as predictors. The forecast horizon is 2007 - 2009, just before and after the 2008 U.S. recession began.

Load and Inspect Data

Load the U.S. macroeconomic data set.

```
load Data_USEconModel
```

The time series in the data set contain quarterly, macroeconomic measurements from 1947 to 2009. For more details, a list of variables, and descriptions, enter `Description` at the command line.

Extract the predictors, and then the response (the response should be the last column). Focus the sample on observations taken from 1960 - 2009.

```
idx = year(dates) >= 1960;
y = DataTable.GDP(idx);
X = DataTable{idx,{'CPIAUCSL' 'COE'}};
varNames = {'CPIAUCSL' 'COE' 'GDP'};
dates = dates(idx);
```

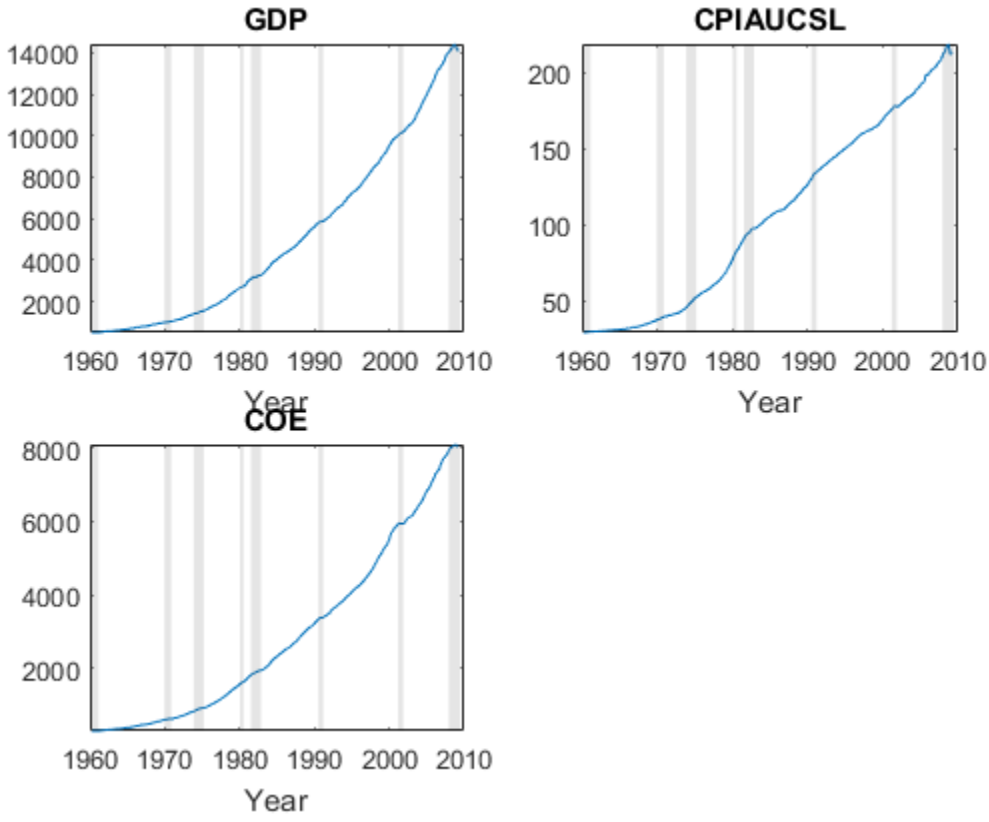
Identify forecast horizon indices.

```
fHIdx = year(dates) >= 2007;
```

Plot all series individually. Identify the periods of recession.

```
figure;
subplot(2,2,1);
plot(dates,y)
title(varNames{end});
xlabel('Year');
axis tight
datetick;
recessionplot;
for j = 1:size(X,2);
    subplot(2,2,j + 1);
    plot(dates,X(:,j))
    title(varNames{j});
```

```
    xlabel('Year');
    axis tight
    datetick;
    recessionplot;
end
```



All variables appear to grow exponentially. Also, around the last recession, a decline appears. Suppose that a linear regreession model of GDP onto CPI and COE is appropriate, and you want to test whether there is a structural change in the model in 2007.

Check Chow Test Assumptions

Chow tests rely on:

- Independent, Gaussian-distributed innovations
- Constancy of the innovations variance within subsamples
- Constancy of the innovations across any structural breaks

If a model violates these assumptions, then the Chow test result might not be correct, or the Chow test might lack power. Investigate whether the assumptions hold. If any do not, preprocess the data further.

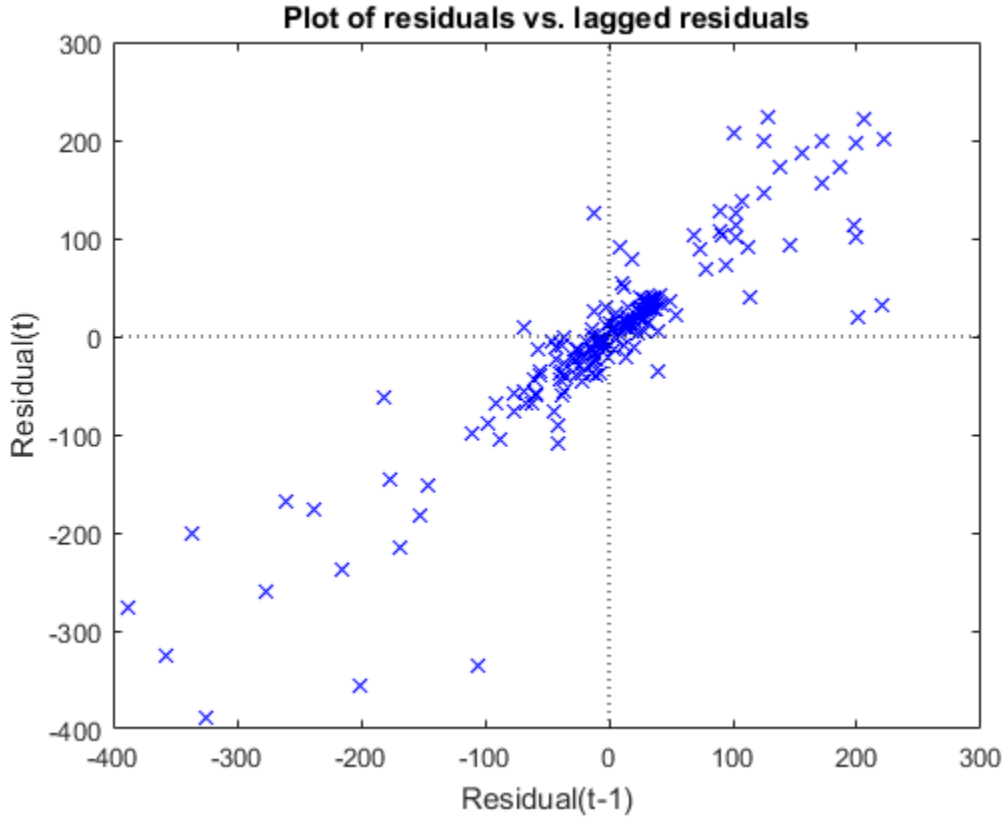
Fit the linear model to the entire series. Include an intercept.

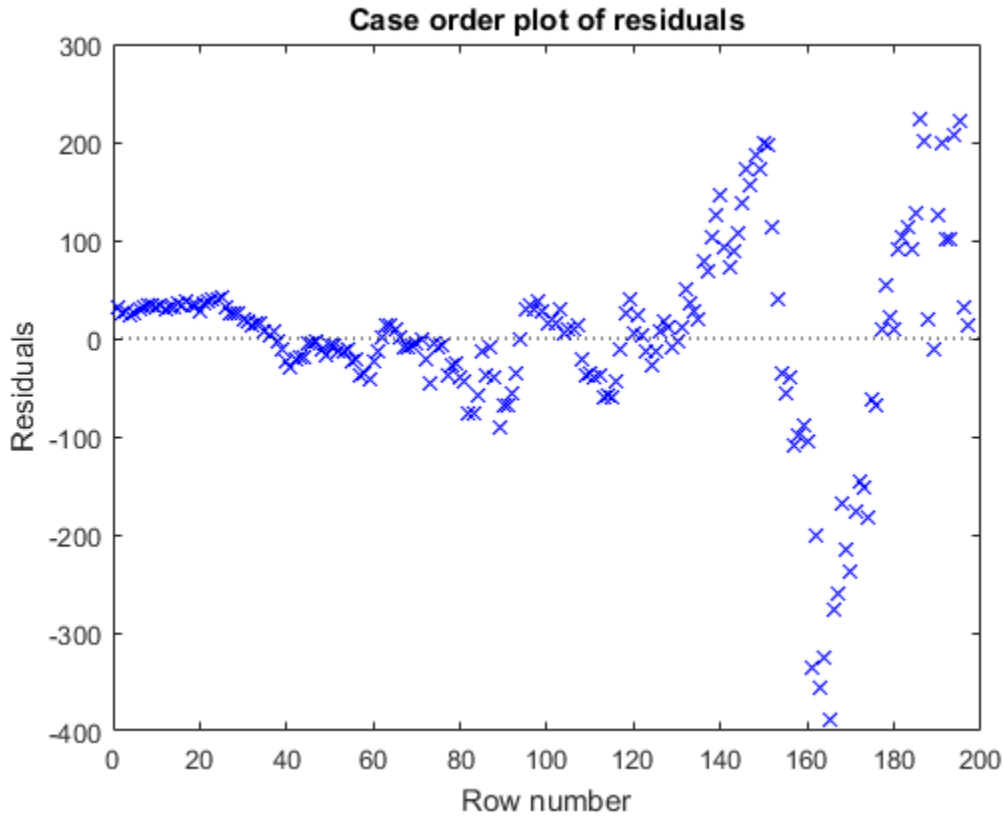
```
Mdl = fitlm(X,y);
```

`Mdl` is a LinearModel class model object.

Extract the residuals from the estimated linear model. Draw two histogram plots using the residuals: one with respect to fitted values in case order, and the other with respect to the previous residual.

```
res = Mdl.Residuals.Raw;
figure;
plotResiduals(Mdl,'lagged');
figure;
plotResiduals(Mdl,'caseorder');
```


Plot of residuals vs. lagged residuals
Residual(t)
Residual(t-1)
300
200
100
0
-100
-200
-300
-400
-400
-300
-200
-100
0
100
200
300



Because the scatter plot of residual vs. lagged residual forms a trend, autocorrelation exists in the residuals. Also, residuals on the extremes seem to flare out, which suggests the presence of heteroscedasticity.

Conduct Engle's ARCH test at 5% level of significance to assess whether the innovations are heteroscedastic.

```
[hARCH,pValueARCH] = archtest(res)
```

```
hARCH =

     1
```

```
pValueARCH =

     0
```

`hARCH = 1` suggests to reject the null hypothesis that the entire residual series has no conditional heteroscedasticity.

Apply the log transformation to all series that appear to grow exponentially to reduce the effects of heteroscedasticity.

```
y = log(y);
X = log(X);
```

To account for autocorrelation, create predictor variables for all exponential series by lagging them by one period.

```
LagMat = lagmatrix([X y],1);
X = [X(2:end,:)  LagMat(2:end,:)]; % Concatenate data and remove first row
fHIdx = fHIdx(2:end);
y = y(2:end);
```

Based on the residual diagnostics, choose this linear model for GDP

$$\mathrm{GDP}_t = \beta_0 + \beta_1 \mathrm{CPIAUCSL}_t + \beta_2 \mathrm{COE}_t + \beta_3 \mathrm{CPIAUCSL}_{t-1} + \beta_4 \mathrm{COE}_{t-1} + \beta_5 \mathrm{GDP}_{t-1} + \varepsilon_t.$$

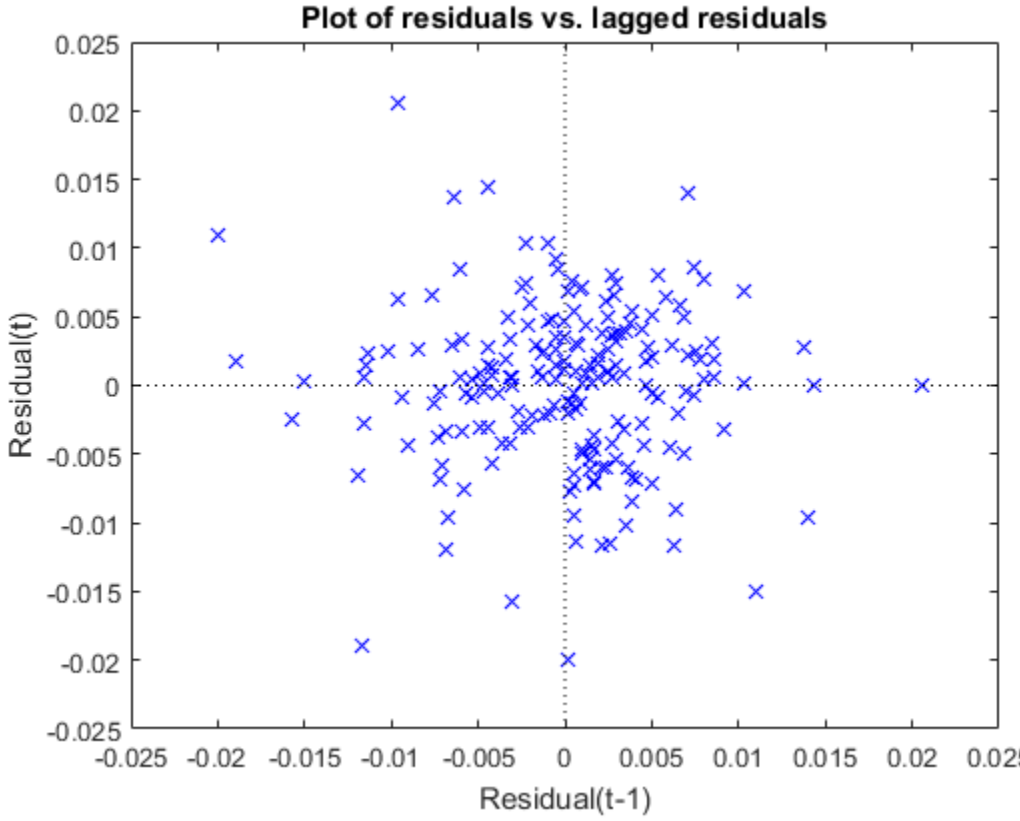
ε_t should be a Gaussian series of innovations with mean zero and constant variance σ^2.

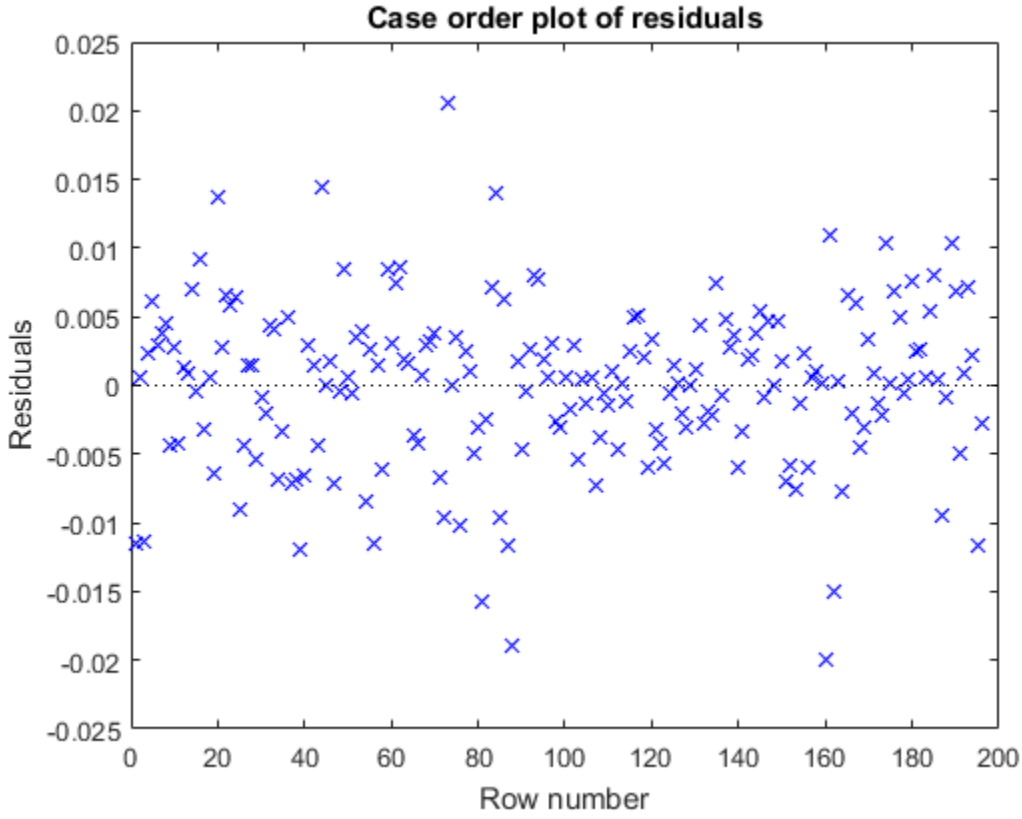
Diagnose the residuals again.

```
Mdl = fitlm(X,y);

res = Mdl.Residuals.Raw;
figure;
plotResiduals(Mdl,'lagged');
figure;
plotResiduals(Mdl,'caseorder');

[hARCH,pValueARCH] = archtest(res)

SubMdl = {fitlm(X(~fHIdx,:),y(~fHIdx)) fitlm(X(fHIdx,:),y(fHIdx))};
```

```
subRes = {SubMdl{1}.Residuals.Raw SubMdl{2}.Residuals.Raw};
[hVT2,pValueVT2] = vartest2(subRes{1},subRes{2})
```

```
hARCH =

     0


pValueARCH =

    0.2813


hVT2 =

     0


pValueVT2 =

    0.1645
```


Plot of residuals vs. lagged residuals
Residual(t)
Residual(t-1)
0.025
0.02
0.015
0.01
0.005
0
-0.005
-0.01
-0.015
-0.02
-0.025
-0.025 -0.02 -0.015 -0.01 -0.005 0 0.005 0.01 0.015 0.02 0.025



The residuals plots and tests suggest that the innovations are homoscedastic and uncorrelated.

Conduct a Kolmogorov-Smirnov test to assess whether the innovations are Gaussian.

```
[hKS,pValueKS] = kstest(res/std(res))
```

```
hKS =

     0


pValueKS =
```

```
    0.2347
```

`hKS = 0` suggests to not reject the null hypothesis that the innovations are Gaussian.

For the distributed lag model, the Chow test assumptions appear valid.

Conduct Chow Test

Treating 2007 and beyond as a post-recession regime, test whether the linear model is stable. Specify that the break point is the last quarter of 2006. Because the complementary subsample size is greater than the number of coefficients, conduct a break point test.

```
bp = find(~fHIdx,1,'last');
chowtest(X,y,bp,'Display','summary');
```

```
RESULTS SUMMARY

***************
Test 1

Sample size: 196
Breakpoint: 187

Test type: breakpoint
Coefficients tested: All

Statistic: 1.3741
Critical value: 2.1481

P value: 0.2272
Significance level: 0.0500

Decision: Fail to reject coefficient stability
```

The test fails to reject the stability of the linear model. Evidence is inefficient to infer a structural change between `Q4-2006` and `Q1-2007`.

See Also

`archtest` | `chowtest` | `fitlm` | LinearModel | `vartest2`

Related Examples

- "Power of the Chow Test" on page 3-123

Power of the Chow Test

This example shows how to estimate the power of a Chow test using a Monte Carlo simulation.

Introduction

Statistical power is the probability of rejecting the null hypothesis given that it is actually false. To estimate the power of a test:

1. Simulate many data sets from a model that typifies the alternative hypothesis.
2. Test each data set.
3. Estimate the power, which is the proportion of times the test rejects the null hypothesis.

The following can compromise the power of the Chow test:

- Linear model assumption departures
- Relatively large innovation variance
- Using the forecast test when the sample size of the complementary subsample is greater than the number of coefficients in the test [24].

Departures from model assumptions allow for an examination of the factors that most affect the power of the Chow test.

Consider the model

$$y = \begin{bmatrix} X1 & 0 \\ 0 & X2 \end{bmatrix} \begin{bmatrix} beta1 \\ beta2 \end{bmatrix} + innov$$

- `innov` is a vector of random Gaussian variates with mean zero and standard deviation `sigma`.
- `X1` and `X2` are the sets of predictor data for initial and complementary subsamples, respectively.
- `beta1` and `beta2` are the regression coefficient vectors for the initial and complementary subsamples, respectively.

Simulate Predictor Data

Specify four predictors, 50 observations, and a break point at period 44 for the simulated linear model.

```
numPreds = 4;
numObs = 50;
bp = 44;
rng(1); % For reproducibility
```

Form the predictor data by specifying means for the predictors, and then adding random, standard Gaussian noise to each of the means.

```
mu = [0 1 2 3];
X = repmat(mu,numObs,1) + randn(numObs,numPreds);
```

To indicate an intercept, add a column of ones to the predictor data.

```
X = [ones(numObs,1) X];
X1 = X(1:bp,:);        % Initial subsample predictors
X2 = X(bp+1:end,:);    % Complementary subsample predictors
```

Specify the true values of the regression coefficients.

```
beta1 = [1 2 3 4 5]'; % Initial subsample coefficients
```

Estimate Power for Small and Large Jump

Compare the power between the break point and forecast tests for jumps of different sizes small in the intercept and second regression coefficient. In this example, a small jump is a 10% increase in the current value, and a large jump is a 15% increase. Complementary subsample coefficients

```
beta2Small = beta1 + [beta1(1)*0.1 0 beta1(3)*0.1 0 0 ]';
beta2Large = beta1 + [beta1(1)*0.15 0 beta1(3)*0.15 0 0 ]';
```

Simulate 1000 response paths of the linear model for each of the small and large coefficient jumps. Specify that `sigma` is 0.2. Choose to test the intercept and the second regression coefficient.

```
M = 1000;

sigma = 0.2;
Coeffs = [true false true false false];
h1BP = nan(M,2); % Preallocation
h1F = nan(M,2);
for j = 1:M
    innovSmall = sigma*randn(numObs,1);
    innovLarge = sigma*randn(numObs,1);
    ySmall = [X1 zeros(bp,size(X2,2)); ...
```

```
        zeros(numObs - bp,size(X1,2)) X2]*[beta1; beta2Small] + innovSmall;
    yLarge = [X1 zeros(bp,size(X2,2)); ...
        zeros(numObs - bp,size(X1,2)) X2]*[beta1; beta2Large] + innovLarge;
    h1BP(j,1) = chowtest(X,ySmall,bp,'Intercept',false,'Coeffs',Coeffs,...
        'Display','off')';
    h1BP(j,2) = chowtest(X,yLarge,bp,'Intercept',false,'Coeffs',Coeffs,...
        'Display','off')';
    h1F(j,1) = chowtest(X,ySmall,bp,'Intercept',false,'Coeffs',Coeffs,...
        'Test','forecast','Display','off')';
    h1F(j,2) = chowtest(X,yLarge,bp,'Intercept',false,'Coeffs',Coeffs,...
        'Test','forecast','Display','off')';
end
```

Estimate the power by computing the proportion of times `chowtest` correctly rejected the null hypothesis of coefficient stability.

```
power1BP = mean(h1BP);
power1F = mean(h1F);
table(power1BP',power1F','RowNames',{'Small_Jump','Large_Jump'},...
    'VariableNames',{'Breakpoint','Forecast'})
```

```
ans =

                Breakpoint    Forecast
                __________    ________

    Small_Jump    0.717         0.645
    Large_Jump    0.966          0.94
```

In this scenario, the Chow test can detect a change in the coefficient with more power when the jump is larger. The break point test has greater power to detect the jump than the forecast test.

Estimate Power for Large Innovations Variance

Simulate 1000 response paths of the linear model for a large coefficient jump. Specify that `sigma` is 0.4. Choose to test the intercept and the second regression coefficient.

```
sigma = 0.4;
h2BP = nan(M,1);
h2F = nan(M,1);
for j = 1:M
    innov = sigma*randn(numObs,1);
```

```
    y = [X1 zeros(bp,size(X2,2)); ...
        zeros(numObs - bp,size(X1,2)) X2]*[beta1; beta2Large] + innov;
    h2BP(j) = chowtest(X,y,bp,'Intercept',false,'Coeffs',Coeffs,...
        'Display','off')';
    h2F(j) = chowtest(X,y,bp,'Intercept',false,'Coeffs',Coeffs,...
        'Test','forecast','Display','off')';
end

power2BP = mean(h2BP);
power2F = mean(h2F);
table([power1BP(2); power2BP],[power1F(2); power2F],...
    'RowNames',{'Small_sigma','Large_Sigma'},...
    'VariableNames',{'Breakpoint','Forecast'})


ans =

                  Breakpoint    Forecast
                  __________    ________

    Small_sigma    0.966          0.94
    Large_Sigma    0.418         0.352
```

For larger innovation variance, both Chow tests have difficulty detecting the large structural breaks in the intercept and second regression coefficient.

See Also

`chowtest`

Related Examples

- “Check Model Assumptions for Chow Test” on page 3-112

Time Series Regression Models

Time Series Regression Models

Time series regression models attempt to explain the current response using the response history (autoregressive dynamics) and the transfer of dynamics from relevant predictors (or otherwise). Theoretical frameworks for potential relationships among variables often permit different representations of the system.

Use time series regression models to analyze *time series data*, which are measurements that you take at successive time points. For example, use time series regression modeling to:

- Examine the linear effects of the current and past unemployment rates and past inflation rates on the current inflation rate.
- Forecast GDP growth rates by using an ARIMA model and include the CPI growth rate as a predictor.
- Determine how a unit increase in rainfall, amount of fertilizer, and labor affect crop yield.

You can start a time series analysis by building a design matrix (X_t), which can include current and past observations of predictors. You can also complement the regression component with an autoregressive (AR) component to account for the possibility of response (y_t) dynamics. For example, include past measurements of inflation rate in the regression component to explain the current inflation rate. AR terms account for dynamics unexplained by the regression component, which is necessarily underspecified in econometric applications. Also, the AR terms absorb residual autocorrelations, simplify innovation models, and generally improve forecast performance. Then, apply ordinary least squares (OLS) to the multiple linear regression (MLR) model:

$$y_t = X_t\beta + u_t.$$

If a residual analysis suggests classical linear model assumption departures such as that heteroscedasticity or autocorrelation (i.e., nonspherical errors), then:

- You can estimate robust *HAC* (heteroscedasticity and autocorrelation consistent) standard errors (for details, see `hac`).
- If you know the innovation covariance matrix (at least up to a scaling factor), then you can apply *generalized least squares* (GLS). Given that the innovation covariance matrix is correct, GLS effectively reduces the problem to a linear regression where the residuals have covariance *I*.

- If you do not know the structure of the innovation covariance matrix, but know the nature of the heteroscedasticity and autocorrelation, then you can apply *feasible generalized least squares* (FGLS). FGLS applies GLS iteratively, but uses the estimated residual covariance matrix. FGLS estimators are efficient under certain conditions. For details, see [1], Chapter 11.

There are time series models that model the dynamics more explicitly than MLR models. These models can account for AR and predictor effects as with MLR models, but have the added benefits of:

- Accounting for moving average (MA) effects. Include MA terms to reduce the number of AR lags, effectively reducing the number of observation required to initialize the model.
- Easily modeling seasonal effects. In order to model seasonal effects with an MLR model, you have to build an indicator design matrix.
- Modeling nonseasonal and seasonal integration for unit root nonstationary processes.

These models also differ from MLR in that they rely on distribution assumptions (i.e., they use maximum likelihood for estimation). Popular types of time series regression models include:

- *Autoregressive integrated moving average with exogenous* predictors (ARIMAX). This is an ARIMA model that linearly includes predictors (exogenous or otherwise). For details, see arima or "ARIMAX(p,D,q) Model" on page 5-58.
- *Regression model with ARIMA time series errors*. This is an MLR model where the unconditional disturbance process (u_t) is an ARIMA time series. In other words, you explicitly model u_t as a linear time series. For details, see regARIMA.
- *Distributed lag model* (DLM). This is an MLR model that includes the effects of predictors that persist over time. In other words, the regression component contains coefficients for contemporaneous and lagged values of predictors. Econometrics Toolbox does not contain functions that model DLMs explicitly, but you can use `regARIMA` or `fitlm` with an appropriately constructed predictor (design) matrix to analyze a DLM.
- *Transfer function* (autoregressive distributed lag) model. This model extends the distributed lag framework in that it includes autoregressive terms (lagged responses). Econometrics Toolbox does not contain functions that model DLMs explicitly, but you can use the arima functionality with an appropriately constructed predictor matrix to analyze an autoregressive DLM.

The choice you make on which model to use depends on your goals for the analysis, and the properties of the data.

References

[1] Greene, W. H. *Econometric Analysis*. 6th ed. Englewood Cliffs, NJ: Prentice Hall, 2008.

See Also

arima | `fitlm` | `hac` | regARIMA

More About

- "ARIMAX(p,D,q) Model" on page 5-58
- "Regression Models with Time Series Errors" on page 4-6

Regression Models with Time Series Errors

In this section...

What Are Regression Models with Time Series Errors?

Regression models with time series errors attempt to explain the mean behavior of a response series (y_t, $t = 1,...,T$) by accounting for linear effects of predictors (X_t) using a multiple linear regression (MLR). However, the errors (u_t), called *unconditional disturbances*, are time series rather than white noise, which is a departure from the linear model assumptions. Unlike the ARIMA model that includes exogenous predictors, regression models with time series errors preserve the sensitivity interpretation of the regression coefficients (β) [2].

These models are particularly useful for econometric data. Use these models to:

- Analyze the effects of a new policy on a market indicator (an intervention model).
- Forecast population size adjusting for predictor effects, such as expected prevalence of a disease.
- Study the behavior of a process adjusting for calender effects. For example, you can analyze traffic volume by adjusting for the effects of major holidays. For details, see [3].
- Estimate the trend by including time (t) in the model.
- Forecast total energy consumption accounting for current and past prices of oil and electricity (distributed lag model).

Use these tools in Econometrics Toolbox to:

- Specify a regression model with ARIMA errors (see regARIMA).
- Estimate parameters using a specified model, and response and predictor data (see estimate).
- Simulate responses using a model and predictor data (see simulate).
- Forecast responses using a model and future predictor data (see forecast).
- Infer residuals and estimated unconditional disturbances from a model using the model and predictor data (see infer).

- filter innovations through a model using the model and predictor data
- Generate impulse responses (see impulse).
- Compare a regression model with ARIMA errors to an ARIMAX model (see arima).

Conventions

A regression model with time series errors has the following form (in lag operator notation):

$$y_t = c + X_t\beta + u_t$$
$$a(L)A(L)(1-L)^D\left(1-L^s\right)u_t = b(L)B(L)\varepsilon_t,$$

where

- $t = 1,...,T$.
- y_t is the response series.
- X_t is row t of X, which is the matrix of concatenated predictor data vectors. That is, X_t is observation t of each predictor series.
- c is the regression model intercept.
- β is the regression coefficient.
- u_t is the disturbance series.
- ε_t is the innovations series.
- $L^j y_t = y_{t-j}$.
- $a(L) = \left(1 - a_1L - ... - a_pL^p\right)$, which is the degree p, nonseasonal autoregressive polynomial.
- $A(L) = \left(1 - A_1L - ... - A_{p_s}L^{p_s}\right)$, which is the degree p_s, seasonal autoregressive polynomial.
- $(1-L)^D$, which is the degree D, nonseasonal integration polynomial.
- $\left(1-L^s\right)$, which is the degree s, seasonal integration polynomial.

- $b(L) = \left(1 + b_1 L + \ldots + b_q L^q\right)$, which is the degree q, nonseasonal moving average polynomial.
- $B(L) = \left(1 + B_1 L + \ldots + B_{q_s} L^{q_s}\right)$, which is the degree q_s, seasonal moving average polynomial.

Following Box and Jenkins methodology, u_t is a stationary or unit root nonstationary, regular, linear time series. However, if u_t is unit root nonstationary, then you do not have to explicitly difference the series as they recommend in [1]. You can simply specify the seasonal and nonseasonal integration degree using the software. For details, see "Specify Regression Models with ARIMA Errors Using regARIMA" on page 4-10.

Another deviation from the Box and Jenkins methodology is that u_t does not have a constant term (conditional mean), and therefore its unconditional mean is 0. However, the regression model contains an intercept term, c.

Note: If the unconditional disturbance process is nonstationary (i.e., the nonseasonal or seasonal integration degree is greater than 0), then the regression intercept, c, is not identifiable. For details, see "Intercept Identifiability in Regression Models with ARIMA Errors" on page 4-130.

The software enforces stability and invertibility of the ARMA process. That is,

$$\psi(L) = \frac{b(L)B(L)}{a(L)A(L)} = 1 + \psi_1 L + \psi_2 L^2 + \ldots,$$

where the series $\{\psi_t\}$ must be absolutely summable. The conditions for $\{\psi_t\}$ to be absolutely summable are:

- $a(L)$ and $A(L)$ are *stable* (i.e., the eigenvalues of $a(L) = 0$ and $A(L) = 0$ lie inside the unit circle).
- $b(L)$ and $B(L)$ are *invertible* (i.e., their eigenvalues lie of $b(L) = 0$ and $B(L) = 0$ inside the unit circle).

The software uses maximum likelihood for parameter estimation. You can choose either a Gaussian or Student's t distribution for the innovations, ε_t.

The software treats predictors as nonstochastic variables for estimation and inference.

References

[1] Box, G. E. P., G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis: Forecasting and Control*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

[2] Hyndman, R. J. (2010, October). "The ARIMAX Model Muddle." *Rob J. Hyndman*. Retrieved February 7, 2013 from http://robjhyndman.com/researchtips/arimax/.

[3] Ruey, T. S. "Regression Models with Time Series Errors." *Journal of the American Statistical Association*. Vol. 79, Number 385, March 1984, pp. 118–124.

See Also

arima | estimate | filter | forecast | impulse | infer | regARIMA | simulate

Related Examples

- "Compare Alternative ARIMA Model Representations" on page 4-136
- "Intercept Identifiability in Regression Models with ARIMA Errors" on page 4-130

More About

- "ARIMA Model Including Exogenous Covariates" on page 5-58
- "Specify Regression Models with ARIMA Errors Using regARIMA" on page 4-10

Specify Regression Models with ARIMA Errors Using regARIMA

In this section...

Default Regression Model with ARIMA Errors Specifications

Regression models with ARIMA errors have the following form (in lag operator notation):

$$y_t = c + X_t\beta + u_t$$
$$a(L)A(L)(1-L)^D\left(1-L^s\right)u_t = b(L)B(L)\varepsilon_t,$$

where

- $t = 1,...,T$.
- y_t is the response series.
- X_t is row t of X, which is the matrix of concatenated predictor data vectors. That is, X_t is observation t of each predictor series.
- c is the regression model intercept.
- β is the regression coefficient.
- u_t is the disturbance series.
- ε_t is the innovations series.
- $L^j y_t = y_{t-j}$.
- $a(L) = \left(1 - a_1L - ... - a_pL^p\right)$, which is the degree p, nonseasonal autoregressive polynomial.
- $A(L) = \left(1 - A_1L - ... - A_{p_s}L^{p_s}\right)$, which is the degree p_s, seasonal autoregressive polynomial.

- $(1-L)^D$, which is the degree D, nonseasonal integration polynomial.
- $\left(1-L^s\right)$, which is the degree s, seasonal integration polynomial.
- $b(L)=\left(1+b_1L+\ldots+b_qL^q\right)$, which is the degree q, nonseasonal moving average polynomial.
- $B(L)=\left(1+B_1L+\ldots+B_{q_s}L^{q_s}\right)$, which is the degree q_s, seasonal moving average polynomial.

For simplicity, use the shorthand notation `Mdl = regARIMA(p,D,q)` to specify a regression model with ARIMA(p,D,q) errors, where `p`, `D`, and `q` are nonnegative integers. `Mdl` has the following default properties.

Property Name	Property Data Type
`AR`	Length `p` cell vector of `NaN`s
`Beta`	Empty vector `[]` of regression coefficients, corresponding to the predictor series
`D`	Nonnegative scalar, corresponding to D
`Distribution`	`Gaussian`, corresponding to the distribution of ε_t
`Intercept`	`NaN`, corresponding to c
`MA`	Length `q` cell vector of `NaN`s
`P`	Number of AR terms plus degree of integration, $p + D$
`Q`	Number of MA terms, q
`SAR`	Empty cell vector
`SMA`	Empty cell vector
`Variance`	`NaN`, corresponding to the variance of ε_t
`Seasonality`	`0`, corresponding to s

If you specify nonseasonal ARIMA errors, then

- The properties `D` and `Q` are the inputs `D` and `q`, respectively.
- Property `P` = `p` + `D`, which is the degree of the compound, nonseasonal autoregressive polynomial. In other words, `P` is the degree of the product of the nonseasonal autoregressive polynomial, $a(L)$ and the nonseasonal integration polynomial, $(1 - L)^D$.

The values of properties `P` and `Q` indicate how many presample observations the software requires to initialize the time series.

You can modify the properties of `Mdl` using dot notation. For example, `Mdl.Variance = 0.5` sets the innovation variance to 0.5.

For maximum flexibility in specifying a regression model with ARIMA errors, use name-value pair arguments to, for example, set each of the autoregressive parameters to a value, or specify multiplicative seasonal terms. For example, `Mdl = regARIMA('AR', {0.2 0.1})` defines a regression model with AR(2) errors, and the coefficients are a_1 = 0.2 and a_2 = 0.1.

Specify regARIMA Models Using Name-Value Pair Arguments

You can only specify the nonseasonal autoregressive and moving average polynomial degrees, and nonseasonal integration degree using the shorthand notation `regARIMA(p,D,q)`. Some tasks, such as forecasting and simulation, require you to specify values for parameters. You cannot specify parameter values using shorthand notation. For maximum flexibility, use name-value pair arguments to specify regression models with ARIMA errors.

The nonseasonal ARIMA error model might contain the following polynomials:

- The degree p autoregressive polynomial $a(L) = 1 - a_1L - a_2L_2 - ... - a_pL^p$. The eigenvalues of $a(L)$ must lie within the unit circle (i.e., $a(L)$ must be a stable polynomial).
- The degree q moving average polynomial $b(L) = 1 + b_1L + b_2L_2 + ... + b_qL^q$. The eigenvalues of $b(L)$ must lie within the unit circle (i.e., $b(L)$ must be an invertible polynomial).
- The degree D nonseasonal integration polynomial is $(1 - L)^D$.

The following table contains the name-value pair arguments that you use to specify the ARIMA error model (i.e., a regression model with ARIMA errors, but without a regression component and intercept):

$$y_t = u_t$$
$$a(L)(1-L)^D = b(L)\varepsilon_t.$$

Name-Value Pair Arguments for Nonseasonal ARIMA Error Models

Name	Corresponding Model Term(s) in Equation 4-2	When to Specify
`AR`	Nonseasonal AR coefficients: a_1, a_2,...,a_p	• To set equality constraints for the AR coefficients. For example, to specify the AR coefficients in the ARIMA error model $u_t = 0.8u_{t-1} - 0.2u_{t-2} + \varepsilon_t$, specify `'AR',{0.8,-0.2}`. • You only need to specify the nonzero elements of `AR`. If the nonzero coefficients are at nonconsecutive lags, specify the corresponding lags using `ARLags`. • The coefficients must correspond to a stable AR polynomial.
`ARLags`	Lags corresponding to nonzero, nonseasonal AR coefficients	• `ARLags` is not a model property. Use this argument as a shortcut for specifying `AR` when the nonzero AR coefficients correspond to nonconsecutive lags. For example, to specify nonzero AR coefficients at lags 1 and 12, e.g., $u_t = a_1u_{t-1} + a_2u_{t-12} + \varepsilon_t$, specify `'ARLags',[1,12]`. • Use `AR` and `ARLags` together to specify known nonzero AR coefficients at nonconsecutive lags. For example, if in the given AR(12) error model with a_1 = 0.6 and a_{12} = –0.3, then specify `'AR', {0.6,-0.3},'ARLags',[1,12]`.
`D`	Degree of nonseasonal differencing, D	• To specify a degree of nonseasonal differencing greater than zero. For example,

Name	Corresponding Model Term(s) in Equation 4-2	When to Specify
		to specify one degree of differencing, specify `'D',1`. • By default, `D` has value `0` (meaning no nonseasonal integration).
`Distribution`	Distribution of the innovation process, ε_t	• Use this argument to specify a Student's t distribution. By default, the innovation distribution is `Gaussian`. For example, to specify a t distribution with unknown degrees of freedom, specify `'Distribution','t'`. • To specify a t innovation distribution with known degrees of freedom, assign `Distribution` a structure with fields `Name` and `DoF`. For example, for a t distribution with nine degrees of freedom, specify `'Distribution',struct('Name','t','DoF',9)`.
`MA`	Nonseasonal MA coefficients: b_1, b_2,...,b_q	• To set equality constraints for the MA coefficients. For example, to specify the MA coefficients in the ARIMA error model $u_t = \varepsilon_t + 0.5\varepsilon_{t-1} + 0.2\varepsilon_{t-2}$, specify `'MA',{0.5,0.2}`. • You only need to specify the nonzero elements of `MA`. If the nonzero coefficients are at nonconsecutive lags, specify the corresponding lags using `MALags`. • The coefficients must correspond to an invertible MA polynomial.
`MALags`	Lags corresponding to nonzero, nonseasonal MA coefficients	• `MALags` is not a model property. • Use this argument as a shortcut for specifying `MA` when the nonzero MA coefficients correspond to nonconsecutive lags. For example, to specify nonzero MA coefficients at lags 1 and 4, e.g.,

Name	Corresponding Model Term(s) in Equation 4-2	When to Specify
		$u_t = \varepsilon_t + b_1\varepsilon_{t-1} + b_4\varepsilon_{t-4}$, specify `'MALags',[1,4]`. • Use `MA` and `MALags` together to specify known nonzero MA coefficients at nonconsecutive lags. For example, if in the given MA(4) error model $b_1 = 0.5$ and $b_4 = 0.2$, specify `'MA',{0.4,0.2},'MALags',[1,4]`.
`Variance`	Scalar variance, σ^2, of the innovation process, ε_t	To set equality constraints for σ^2. For example, for an ARIMA error model with known innovation variance 0.1, specify `'Variance',0.1`. By default, `Variance` has value `NaN`.

Use the name-value pair arguments in the following table in conjunction with those in Name-Value Pair Arguments for Nonseasonal ARIMA Error Models to specify the regression components of the regression model with ARIMA errors:

$$y_t = c + X_t\beta + u_t$$
$$a(L)(1-L)^D = b(L)\varepsilon_t.$$

Name-Value Pair Arguments for the Regression Component of the regARIMA Model

Name	Corresponding Model Term(s) in Equation 4-3	When to Specify
`Beta`	Regression coefficient values corresponding to the predictor series, β	• Use this argument to specify the values of the coefficients of the predictor series. For example, use `'Beta',[0.5 7 -2]` to specify $\beta = [0.5 \quad 7 \quad -2]'$. • By default, `Beta` is an empty vector, `[]`.

Name	Corresponding Model Term(s) in Equation 4-3	When to Specify
`Intercept`	Intercept term for the regression model, c	• To set equality constraints for c. For example, for a model with no intercept term, specify `'Intercept',0`. • By default, `Intercept` has value `NaN`.

If the time series has seasonality s, then

- The degree p_s seasonal autoregressive polynomial is $A(L) = 1 - A_1L - A_2L^2 - ... - A_{p_s}L^{p_s}$.
- The degree q_s seasonal moving average polynomial is $B(L)\ 1 + B_1L + B_2L^2 + ... + B_{q_s}L^{q_s}$.
- The degree s seasonal integration polynomial is $(1 - L^s)$.

Use the name-value pair arguments in the following table in conjunction with those in tables Name-Value Pair Arguments for Nonseasonal ARIMA Error Models and Name-Value Pair Arguments for the Regression Component of the regARIMA Model to specify the regression model with multiplicative seasonal ARIMA errors:

$$y_t = c + X_t\beta + u_t$$
$$\alpha(L)(1-L)^D A(L)(1-L^s) = b(L)B(L)\varepsilon_t.$$

Name-Value Pair Arguments for Seasonal ARIMA Models

Argument	Corresponding Model Term(s) in Equation 4-4	When to Specify
SAR	Seasonal AR coefficients: $A_1, A_2, ..., A_{p_s}$	• To set equality constraints for the seasonal AR coefficients. • Use `SARLags` to specify the lags of the nonzero seasonal AR coefficients. Specify the lags associated with the seasonal polynomials in the periodicity of the observed data (e.g., 4, 8,... for quarterly data, or 12, 24,... for monthly data), and not as multiples of the seasonality (e.g., 1, 2,...). For example, to specify the ARIMA error model

Argument	Corresponding Model Term(s) in Equation 4-4	When to Specify
		$(1-0.8L)(1-0.2L^{12})u_t = \varepsilon_t$, specify `'AR',0.8,'SAR',0.2,'SARLags',12`. • The coefficients must correspond to a stable seasonal AR polynomial.
`SARLags`	Lags corresponding to nonzero seasonal AR coefficients, in the periodicity of the responses	• `SARLags` is not a model property. • Use this argument when specifying `SAR` to indicate the lags of the nonzero seasonal AR coefficients. For example, to specify the ARIMA error model $(1-a_1L)(1-A_{12}L^{12})u_t = \varepsilon_t$, specify `'ARLags',1,'SARLags',12`.
`SMA`	Seasonal MA coefficients: $B_1, B_2,...,B_{q_s}$	• To set equality constraints for the seasonal MA coefficients. • Use `SMALags` to specify the lags of the nonzero seasonal MA coefficients. Specify the lags associated with the seasonal polynomials in the periodicity of the observed data (e.g., 4, 8,... for quarterly data, or 12, 24,... for monthly data), and not as multiples of the seasonality (e.g., 1, 2,...). For example, to specify the ARIMA error model $u_t = (1+0.6L)(1+0.2L^4)\varepsilon_t$, specify `'MA',0.6,'SMA',0.2,'SMALags',4`. • The coefficients must correspond to an invertible seasonal MA polynomial.
`SMALags`	Lags corresponding to the nonzero seasonal MA coefficients, in the periodicity of the responses	• `SMALags` is not a model property. • Use this argument when specifying `SMA` to indicate the lags of the nonzero seasonal

Argument	Corresponding Model Term(s) in Equation 4-4	When to Specify
		MA coefficients. For example, to specify the model $u_t = (1 + b_1 L)(1 + B_4 L^4)\varepsilon_t$, specify `'MALags',1,'SMALags',4`.
`Seasonality`	Seasonal periodicity, s	• To specify the degree of seasonal integration s in the seasonal differencing polynomial $\Delta_s = 1 - L^s$. For example, to specify the periodicity for seasonal integration of quarterly data, specify `'Seasonality',4`. • By default, `Seasonality` has value `0` (meaning no periodicity nor seasonal integration).

Note: You cannot assign values to the properties `P` and `Q`. For multiplicative ARIMA error models,

- `regARIMA` sets `P` equal to $p + D + p_s + s$.
- `regARIMA` sets `Q` equal to $q + q_s$

See Also

regARIMA

Related Examples

- "Specify the Default Regression Model with ARIMA Errors" on page 4-20
- "Modify regARIMA Model Properties" on page 4-22
- "Specify Regression Models with AR Errors" on page 4-29
- "Specify Regression Models with MA Errors" on page 4-35
- "Specify Regression Models with ARMA Errors" on page 4-42
- "Specify Regression Models with SARIMA Errors" on page 4-55
- "Specify the ARIMA Error Model Innovation Distribution" on page 4-69

More About

- "Regression Models with Time Series Errors" on page 4-6

Specify the Default Regression Model with ARIMA Errors

This example shows how to specify the default regression model with ARIMA errors using the shorthand ARIMA(p, D, q) notation corresponding to the following equation:

$$y_t = c + u_t$$
$$\left(1 - \phi_1 L - \phi_2 L^2 - \phi_3 L^3\right)(1 - L)^D u_t = \left(1 + \theta_1 L + \theta_2 L^2\right)\varepsilon_t.$$

Specify a regression model with ARIMA(3,1,2) errors.

```
Mdl = regARIMA(3,1,2)
```

```
Mdl =

    ARIMA(3,1,2) Error Model:
    -------------------------
    Distribution: Name = 'Gaussian'
       Intercept: NaN
               P: 4
               D: 1
               Q: 2
              AR: {NaN NaN NaN} at Lags [1 2 3]
             SAR: {}
              MA: {NaN NaN} at Lags [1 2]
             SMA: {}
        Variance: NaN
```

The model specification for `Mdl` appears in the Command Window. By default, `regARIMA` sets:

- The autoregressive (`AR`) parameter values to `NaN` at lags `[1 2 3]`
- The moving average (`MA`) parameter values to `NaN` at lags `[1 2]`
- The variance (`Variance`) of the innovation process, ε_t, to `NaN`
- The distribution (`Distribution`) of ε_t to `Gaussian`
- The regression model intercept to `NaN`

There is no regression component (`Beta`) by default.

`The property:`

- `P = p + D`, which represents the number of presample observations that the software requires to initialize the autoregressive component of the model to perform, for example, estimation.
- `D` represents the level of nonseasonal integration.
- `Q` represents the number of presample observations that the software requires to initialize the moving average component of the model to perform, for example, estimation.

Fit `Mdl` to data by passing it and the data into `estimate`. If you pass the predictor series into `estimate`, then `estimate` estimates `Beta` by default.

You can modify the properties of `Mdl` using dot notation.

References:

Box, G. E. P., G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis: Forecasting and Control*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

See Also

estimate | forecast | regARIMA | simulate

Related Examples

- “Specify Regression Models with ARIMA Errors Using regARIMA” on page 4-10
- “Modify regARIMA Model Properties” on page 4-22
- “Specify Regression Models with AR Errors” on page 4-29
- “Specify Regression Models with MA Errors” on page 4-35
- “Specify Regression Models with ARMA Errors” on page 4-42
- “Specify Regression Models with SARIMA Errors” on page 4-55
- “Specify the ARIMA Error Model Innovation Distribution” on page 4-69

More About

- “Regression Models with Time Series Errors” on page 4-6

Modify regARIMA Model Properties

In this section...

Modify Properties Using Dot Notation

If you create a regression model with ARIMA errors using `regARIMA`, then the software assigns values to all of its properties. To change any of these property values, you do not need to reconstruct the entire model. You can modify property values of an existing model using dot notation. To access the property, type the model name, then the property name, separated by `'|.|'` (a period).

Specify the regression model with ARIMA(3,1,2) errors

$$\begin{aligned} y_t &= c + u_t \\ \left(1 - \phi_1 L - \phi_2 L^2 - \phi_3 L^3\right)(1 - L)^D u_t &= \left(1 + \theta_1 L + \theta_2 L^2\right)\varepsilon_t. \end{aligned}$$

```
Mdl = regARIMA(3,1,2);
```

Use cell array notation to set the autoregressive and moving average parameters to values.

```
Mdl.AR = {0.2 0.1 0.05};
Mdl.MA = {0.1 -0.05}
```

```
Mdl =

    ARIMA(3,1,2) Error Model:
    -------------------------
    Distribution: Name = 'Gaussian'
       Intercept: NaN
               P: 4
               D: 1
               Q: 2
              AR: {0.2 0.1 0.05} at Lags [1 2 3]
             SAR: {}
              MA: {0.1 -0.05} at Lags [1 2]
             SMA: {}
```

```
        Variance: NaN
```

Use dot notation to display the autoregressive coefficients of `Mdl` in the Command Window.

```
ARCoeff = Mdl.AR
```

```
ARCoeff =

    [0.2000]    [0.1000]    [0.0500]
```

`ARCoeff` is a 1-by-3 cell array. Each, successive cell contains the next autoregressive lags.

You can also add more lag coefficients.

```
Mdl.MA = {0.1 -0.05 0.01}
```

```
Mdl =

    ARIMA(3,1,3) Error Model:
    -------------------------
    Distribution: Name = 'Gaussian'
       Intercept: NaN
               P: 4
               D: 1
               Q: 3
              AR: {0.2 0.1 0.05} at Lags [1 2 3]
             SAR: {}
              MA: {0.1 -0.05 0.01} at Lags [1 2 3]
             SMA: {}
        Variance: NaN
```

By default, the specification sets the new coefficient to the next, consecutive lag. The addition of the new coefficient increases `Q` by 1.

You can specify a lag coefficient to a specific lag term by using cell indexing.

```
Mdl.AR{12} = 0.01
```

```
Mdl =
```

```
    ARIMA(12,1,3) Error Model:
    --------------------------
    Distribution: Name = 'Gaussian'
       Intercept: NaN
               P: 13
               D: 1
               Q: 3
              AR: {0.2 0.1 0.05 0.01} at Lags [1 2 3 12]
             SAR: {}
              MA: {0.1 -0.05 0.01} at Lags [1 2 3]
             SMA: {}
        Variance: NaN
```

The autoregressive coefficient `0.01` is located at the 12th lag. Property `P` increases to 13 with the new specification.

Set the innovation distribution to the *t* distribution with `NaN` degrees of freedom.

```
Distribution = struct('Name','t','DoF',NaN);
Mdl.Distribution = Distribution
```

```
Mdl =

    ARIMA(12,1,3) Error Model:
    --------------------------
    Distribution: Name = 't', DoF = NaN
       Intercept: NaN
               P: 13
               D: 1
               Q: 3
              AR: {0.2 0.1 0.05 0.01} at Lags [1 2 3 12]
             SAR: {}
              MA: {0.1 -0.05 0.01} at Lags [1 2 3]
             SMA: {}
        Variance: NaN
```

If `DoF` is `NaN`, then `estimate` estimates the degrees of freedom. For other tasks, such as simulating or forecasting a model, you must specify a value for `DoF`.

To specify a regression coefficient, assign a vector to the property `Beta`.

```
Mdl.Beta = [1; 3; -5]
```

```
Mdl =

    Regression with ARIMA(12,1,3) Error Model:
    ----------------------------------------
    Distribution: Name = 't', DoF = NaN
       Intercept: NaN
            Beta: [1 3 -5]
               P: 13
               D: 1
               Q: 3
              AR: {0.2 0.1 0.05 0.01} at Lags [1 2 3 12]
             SAR: {}
              MA: {0.1 -0.05 0.01} at Lags [1 2 3]
             SMA: {}
        Variance: NaN
```

If you pass `Mdl` into `estimate` with the response data and three predictor series, then the software fixes the non-`|NaN|` parameters at their values, and estimate `Intercept`, `Variance`, and `DoF`. For example, if you want to simulate data from this model, then you must specify `Variance` and `DoF`.

Nonmodifiable Properties

Not all properties of a `regARIMA` model are modifiable. To change them directly, you must redefine the model using `regARIMA`. Nonmodifiable properties include:

- `P`, which is the compound autoregressive polynomial degree. The software determines `P` from p, d, p_s, and s. For details on notation, see "Regression Model with ARIMA Time Series Errors" on page 9-851.
- `Q`, which is the compound moving average degree. The software determines `Q` from q and q_s
- `DoF`, which is the degrees of freedom for models having a t-distributed innovation process

Though they are not explicitly properties, you cannot reassign or print the lag structure using `ARLags`, `MALags`, `SARLags`, or `SMALags`. Pass these and the lag structure into `regARIMA` as name-value pair arguments when you specify the model.

For example, specify a regression model with ARIMA(4,1) errors using `regARIMA`, where the autoregressive coefficients occur at lags 1 and 4.

```
Mdl = regARIMA('ARLags',[1 4],'MALags',1)
```

```
Mdl =

    ARIMA(4,0,1) Error Model:
    -------------------------
    Distribution: Name = 'Gaussian'
       Intercept: NaN
               P: 4
               D: 0
               Q: 1
              AR: {NaN NaN} at Lags [1 4]
             SAR: {}
              MA: {NaN} at Lags [1]
             SMA: {}
        Variance: NaN
```

You can produce the same results by specifying a regression model with ARMA(1,1) errors, then adding an autoregressive coefficient at the fourth lag.

```
Mdl = regARIMA(1,0,1);
Mdl.AR{4} = NaN
```

```
Mdl =

    ARIMA(4,0,1) Error Model:
    -------------------------
    Distribution: Name = 'Gaussian'
       Intercept: NaN
               P: 4
               D: 0
               Q: 1
              AR: {NaN NaN} at Lags [1 4]
             SAR: {}
              MA: {NaN} at Lags [1]
             SMA: {}
        Variance: NaN
```

To change the value of `DoF`, you must define a new structure for the distribution, and use dot notation to pass it into the model. For example, specify a regression model with AR(1) errors having t-distributed innovations.

```
Mdl = regARIMA('AR',0.5,'Distribution','t')
```

```
Mdl =

    ARIMA(1,0,0) Error Model:
    -------------------------
    Distribution: Name = 't', DoF = NaN
       Intercept: NaN
               P: 1
               D: 0
               Q: 0
              AR: {0.5} at Lags [1]
             SAR: {}
              MA: {}
             SMA: {}
        Variance: NaN
```

The value of `DoF` is `NaN` by default.

Specify that the *t* distribution has 10 degrees of freedom.

```
Distribution = struct('Name','t','DoF',10);
Mdl.Distribution = Distribution
```

```
Mdl =

    ARIMA(1,0,0) Error Model:
    -------------------------
    Distribution: Name = 't', DoF = 10
       Intercept: NaN
               P: 1
               D: 0
               Q: 0
              AR: {0.5} at Lags [1]
             SAR: {}
              MA: {}
             SMA: {}
        Variance: NaN
```

See Also

estimate | forecast | regARIMA | simulate

Related Examples

- “Specify Regression Models with ARIMA Errors Using regARIMA” on page 4-10

- “Specify the Default Regression Model with ARIMA Errors” on page 4-20
- “Specify Regression Models with AR Errors” on page 4-29
- “Specify Regression Models with MA Errors” on page 4-35
- “Specify Regression Models with ARMA Errors” on page 4-42
- “Specify Regression Models with SARIMA Errors” on page 4-55
- “Specify the ARIMA Error Model Innovation Distribution” on page 4-69

More About

- “Regression Models with Time Series Errors” on page 4-6

Specify Regression Models with AR Errors

In this section...

Default Regression Model with AR Errors

This example shows how to apply the shorthand `regARIMA(p,D,q)` syntax to specify a regression model with AR errors.

Specify the default regression model with AR(3) errors:

$$y_t = c + X_t\beta + u_t$$
$$u_t = a_1u_{t-1} + a_2u_{t-2} + a_3u_{t-3} + \varepsilon_t.$$

```
Mdl = regARIMA(3,0,0)
```

```
Mdl =

    ARIMA(3,0,0) Error Model:
    -------------------------
    Distribution: Name = 'Gaussian'
       Intercept: NaN
               P: 3
               D: 0
               Q: 0
              AR: {NaN NaN NaN} at Lags [1 2 3]
             SAR: {}
              MA: {}
             SMA: {}
        Variance: NaN
```

The software sets the innovation distribution to `Gaussian`, and each parameter to `NaN`. The AR coefficients are at lags 1 through 3.

Pass `Mdl` into `estimate` with data to estimate the parameters set to `NaN`. Though `Beta` is not in the display, if you pass a matrix of predictors (X_t) into `estimate`, then `estimate` estimates `Beta`. The `estimate` function infers the number of regression coefficients in `Beta` from the number of columns in X_t.

Tasks such as simulation and forecasting using `simulate` and `forecast` do not accept models with at least one `NaN` for a parameter value. Use dot notation to modify parameter values.

AR Error Model Without an Intercept

This example shows how to specify a regression model with AR errors without a regression intercept.

Specify the default regression model with AR(3) errors:

$$y_t = X_t\beta + u_t$$
$$u_t = a_1u_{t-1} + a_2u_{t-2} + a_3u_{t-3} + \varepsilon_t.$$

```
Mdl = regARIMA('ARLags',1:3,'Intercept',0)
```

```
Mdl =

    ARIMA(3,0,0) Error Model:
    -------------------------
    Distribution: Name = 'Gaussian'
       Intercept: 0
               P: 3
               D: 0
               Q: 0
              AR: {NaN NaN NaN} at Lags [1 2 3]
             SAR: {}
              MA: {}
             SMA: {}
        Variance: NaN
```

The software sets `Intercept` to 0, but all other parameters in `Mdl` are `|NaN|`s by default.

Since `Intercept` is not a `NaN`, it is an equality constraint during estimation. In other words, if you pass `Mdl` and data into `estimate`, then `estimate` sets `Intercept` to 0 during estimation.

You can modify the properties of `Mdl` using dot notation.

AR Error Model with Nonconsecutive Lags

This example shows how to specify a regression model with AR errors, where the nonzero AR terms are at nonconsecutive lags.

Specify the regression model with AR(4) errors:

$$y_t = c + X_t\beta + u_t$$
$$u_t = a_1 u_{t-1} + a_4 u_{t-4} + \varepsilon_t.$$

```
Mdl = regARIMA('ARLags',[1,4])
```

```
Mdl =

    ARIMA(4,0,0) Error Model:
    -------------------------
    Distribution: Name = 'Gaussian'
       Intercept: NaN
               P: 4
               D: 0
               Q: 0
              AR: {NaN NaN} at Lags [1 4]
             SAR: {}
              MA: {}
             SMA: {}
        Variance: NaN
```

The AR coefficients are at lags 1 and 4.

Verify that the AR coefficients at lags 2 and 3 are 0.

```
Mdl.AR
```

```
ans =
```

```
    [NaN]    [0]    [0]    [NaN]
```

The software displays a 1-by-4 cell array. Each consecutive cell contains the corresponding AR coefficient value.

Pass `Mdl` and data into `estimate`. The software estimates all parameters that have the value `NaN`. Then, `estimate` holds $a_2 = 0$ and $a_3 = 0$ during estimation.

Known Parameter Values for a Regression Model with AR Errors

This example shows how to specify values for all parameters of a regression model with AR errors.

Specify the regression model with AR(4) errors:

$$y_t = X_t \begin{bmatrix} -2 \\ 0.5 \end{bmatrix} + u_t$$
$$u_t = 0.2u_{t-1} + 0.1u_{t-4} + \varepsilon_t,$$

where ε_t is Gaussian with unit variance.

```
Mdl = regARIMA('AR',{0.2,0.1},'ARLags',[1,4],...
    'Constant',0,'Beta',[-2;0.5],'Variance',1)
```

```
Mdl =

    Regression with ARIMA(4,0,0) Error Model:
    ----------------------------------------
    Distribution: Name = 'Gaussian'
       Intercept: 0
            Beta: [-2 0.5]
               P: 4
               D: 0
               Q: 0
              AR: {0.2 0.1} at Lags [1 4]
             SAR: {}
              MA: {}
             SMA: {}
        Variance: 1
```

There are no NaN values in any Mdl properties, and therefore there is no need to estimate Mdl using estimate. However, you can simulate or forecast responses from Mdl using simulate or forecast.

Regression Model with AR Errors and t Innovations

This example shows how to set the innovation distribution of a regression model with AR errors to a t distribution.

Specify the regression model with AR(4) errors:

$$y_t = X_t \begin{bmatrix} -2 \\ 0.5 \end{bmatrix} + u_t$$
$$u_t = 0.2u_{t-1} + 0.1u_{t-4} + \varepsilon_t,$$

where ε_t has a t distribution with the default degrees of freedom and unit variance.

```
Mdl = regARIMA('AR',{0.2,0.1},'ARLags',[1,4],...
    'Constant',0,'Beta',[-2;0.5],'Variance',1,...
    'Distribution','t')
```

```
Mdl =

    Regression with ARIMA(4,0,0) Error Model:
    ----------------------------------------
    Distribution: Name = 't', DoF = NaN
       Intercept: 0
            Beta: [-2 0.5]
               P: 4
               D: 0
               Q: 0
              AR: {0.2 0.1} at Lags [1 4]
             SAR: {}
              MA: {}
             SMA: {}
        Variance: 1
```

The default degrees of freedom is NaN. If you don't know the degrees of freedom, then you can estimate it by passing Mdl and the data to estimate.

Specify a t_{10} distribution.

```
Mdl.Distribution = struct('Name','t','DoF',10)

Mdl =

    Regression with ARIMA(4,0,0) Error Model:
    ----------------------------------------
    Distribution: Name = 't', DoF = 10
       Intercept: 0
            Beta: [-2 0.5]
               P: 4
               D: 0
               Q: 0
              AR: {0.2 0.1} at Lags [1 4]
             SAR: {}
              MA: {}
             SMA: {}
        Variance: 1
```

You can simulate or forecast responses using `simulate` or `forecast` because `Mdl` is completely specified.

In applications, such as simulation, the software normalizes the random t innovations. In other words, `Variance` overrides the theoretical variance of the t random variable (which is `DoF`/(`DoF` - 2)), but preserves the kurtosis of the distribution.

See Also

estimate | forecast | regARIMA | simulate

Related Examples

- “Specify Regression Models with ARIMA Errors Using regARIMA” on page 4-10
- “Specify the Default Regression Model with ARIMA Errors” on page 4-20
- “Specify Regression Models with MA Errors” on page 4-35
- “Specify Regression Models with ARMA Errors” on page 4-42
- “Specify Regression Models with SARIMA Errors” on page 4-55
- “Specify the ARIMA Error Model Innovation Distribution” on page 4-69

More About

- “Regression Models with Time Series Errors” on page 4-6

Specify Regression Models with MA Errors

In this section...

Default Regression Model with MA Errors

This example shows how to apply the shorthand `regARIMA(p,D,q)` syntax to specify the regression model with MA errors.

Specify the default regression model with MA(2) errors:

$$y_t = c + X_t\beta + u_t$$
$$u_t = \varepsilon_t + b_1\varepsilon_{t-1} + b_2\varepsilon_{t-2}.$$

```
Mdl = regARIMA(0,0,2)
```

```
Mdl =

    ARIMA(0,0,2) Error Model:
    -------------------------
    Distribution: Name = 'Gaussian'
       Intercept: NaN
               P: 0
               D: 0
               Q: 2
              AR: {}
             SAR: {}
              MA: {NaN NaN} at Lags [1 2]
             SMA: {}
        Variance: NaN
```

The software sets each parameter to `NaN`, and the innovation distribution to `Gaussian`. The MA coefficients are at lags 1 and 2.

Pass `Mdl` into `estimate` with data to estimate the parameters set to `NaN`. Though `Beta` is not in the display, if you pass a matrix of predictors (X_t) into `estimate`, then `estimate` estimates `Beta`. The `estimate` function infers the number of regression coefficients in `Beta` from the number of columns in X_t.

Tasks such as simulation and forecasting using `simulate` and `forecast` do not accept models with at least one `NaN` for a parameter value. Use dot notation to modify parameter values.

MA Error Model Without an Intercept

This example shows how to specify a regression model with MA errors without a regression intercept.

Specify the default regression model with MA(2) errors:

$$y_t = X_t\beta + u_t$$
$$u_t = \varepsilon_t + b_1\varepsilon_{t-1} + b_2\varepsilon_{t-2}.$$

```
Mdl = regARIMA('MALags',1:2,'Intercept',0)
```

```
Mdl =

    ARIMA(0,0,2) Error Model:
    -------------------------
    Distribution: Name = 'Gaussian'
       Intercept: 0
               P: 0
               D: 0
               Q: 2
              AR: {}
             SAR: {}
              MA: {NaN NaN} at Lags [1 2]
             SMA: {}
        Variance: NaN
```

The software sets `Intercept` to 0, but all other parameters in `Mdl` are `NaN` values by default.

Since `Intercept` is not a `NaN`, it is an equality constraint during estimation. In other words, if you pass `Mdl` and data into `estimate`, then `estimate` sets `Intercept` to 0 during estimation.

You can modify the properties of `Mdl` using dot notation.

MA Error Model with Nonconsecutive Lags

This example shows how to specify a regression model with MA errors, where the nonzero MA terms are at nonconsecutive lags.

Specify the regression model with MA(12) errors:

$$y_t = c + X_t\beta + u_t$$
$$u_t = \varepsilon_t + b_1\varepsilon_{t-1} + b_{12}\varepsilon_{t-12}.$$

```
Mdl = regARIMA('MALags',[1, 12])
```

```
Mdl =

    ARIMA(0,0,12) Error Model:
    --------------------------
    Distribution: Name = 'Gaussian'
       Intercept: NaN
               P: 0
               D: 0
               Q: 12
              AR: {}
             SAR: {}
              MA: {NaN NaN} at Lags [1 12]
             SMA: {}
        Variance: NaN
```

The MA coefficients are at lags 1 and 12.

Verify that the MA coefficients at lags 2 through 11 are 0.

```
Mdl.MA'
```

```
ans =

    [NaN]
    [  0]
    [  0]
    [  0]
```

```
    [  0]
    [  0]
    [  0]
    [  0]
    [  0]
    [  0]
    [  0]
    [NaN]
```

After applying the transpose, the software displays a 12-by-1 cell array. Each consecutive cell contains the corresponding MA coefficient value.

Pass `Mdl` and data into `estimate`. The software estimates all parameters that have the value `NaN`. Then `estimate` holds $b_2 = b_3 = ... = b_{11} = 0$ during estimation.

Known Parameter Values for a Regression Model with MA Errors

This example shows how to specify values for all parameters of a regression model with MA errors.

Specify the regression model with MA(2) errors:

$$y_t = X_t \begin{bmatrix} 0.5 \\ -3 \\ 1.2 \end{bmatrix} + u_t$$
$$u_t = \varepsilon_t + 0.5\varepsilon_{t-1} - 0.1\varepsilon_{t-2},$$

where ε_t is Gaussian with unit variance.

```
Mdl = regARIMA('Intercept',0,'Beta',[0.5; -3; 1.2],...
    'MA',{0.5, -0.1},'Variance',1)
```

```
Mdl =

    Regression with ARIMA(0,0,2) Error Model:
    ----------------------------------------
    Distribution: Name = 'Gaussian'
       Intercept: 0
            Beta: [0.5 -3 1.2]
               P: 0
```

```
              D: 0
              Q: 2
             AR: {}
            SAR: {}
             MA: {0.5 -0.1} at Lags [1 2]
            SMA: {}
       Variance: 1
```

The parameters in `Mdl` do not contain `NaN` values, and therefore there is no need to estimate `Mdl` using `estimate`. However, you can simulate or forecast responses from `Mdl` using `simulate` or `forecast`.

Regression Model with MA Errors and t Innovations

This example shows how to set the innovation distribution of a regression model with MA errors to a t distribution.

Specify the regression model with MA(2) errors:

$$y_t = X_t \begin{bmatrix} 0.5 \\ -3 \\ 1.2 \end{bmatrix} + u_t$$
$$u_t = \varepsilon_t + 0.5\varepsilon_{t-1} - 0.1\varepsilon_{t-2},$$

where ε_t has a t distribution with the default degrees of freedom and unit variance.

```
Mdl = regARIMA('Intercept',0,'Beta',[0.5; -3; 1.2],...
    'MA',{0.5, -0.1},'Variance',1,'Distribution','t')
```

```
Mdl =

    Regression with ARIMA(0,0,2) Error Model:
    ----------------------------------------
    Distribution: Name = 't', DoF = NaN
       Intercept: 0
            Beta: [0.5 -3 1.2]
               P: 0
               D: 0
               Q: 2
              AR: {}
             SAR: {}
```

```
              MA: {0.5 -0.1} at Lags [1 2]
             SMA: {}
        Variance: 1
```

The default degrees of freedom is `NaN`. If you don't know the degrees of freedom, then you can estimate it by passing `Mdl` and the data to `estimate`.

Specify a t_{15} distribution.

```
Mdl.Distribution = struct('Name','t','DoF',15)
```

```
Mdl =

    Regression with ARIMA(0,0,2) Error Model:
    ----------------------------------------
    Distribution: Name = 't', DoF = 15
       Intercept: 0
            Beta: [0.5 -3 1.2]
               P: 0
               D: 0
               Q: 2
              AR: {}
             SAR: {}
              MA: {0.5 -0.1} at Lags [1 2]
             SMA: {}
        Variance: 1
```

You can simulate and forecast responses from by passing `Mdl` to `simulate` or `forecast` because `Mdl` is completely specified.

In applications, such as simulation, the software normalizes the random t innovations. In other words, `Variance` overrides the theoretical variance of the t random variable (which is `DoF`/(`DoF` - 2)), but preserves the kurtosis of the distribution.

See Also

estimate | forecast | regARIMA | simulate

Related Examples

- “Specify Regression Models with ARIMA Errors Using regARIMA” on page 4-10
- “Specify the Default Regression Model with ARIMA Errors” on page 4-20

- "Specify Regression Models with AR Errors" on page 4-29
- "Specify Regression Models with ARMA Errors" on page 4-42
- "Specify Regression Models with SARIMA Errors" on page 4-55
- "Specify the ARIMA Error Model Innovation Distribution" on page 4-69

More About

- "Regression Models with Time Series Errors" on page 4-6

Specify Regression Models with ARMA Errors

In this section...

Default Regression Model with ARMA Errors

This example shows how to apply the shorthand `regARIMA(p,D,q)` syntax to specify the regression model with ARMA errors.

Specify the default regression model with ARMA(3,2) errors:

$$y_t = c + X_t\beta + u_t$$
$$u_t = a_1u_{t-1} + a_2u_{t-2} + a_3u_{t-3} + \varepsilon_t + b_1\varepsilon_{t-1} + b_2\varepsilon_{t-2}.$$

```
Mdl = regARIMA(3,0,2)
```

```
Mdl =

    ARIMA(3,0,2) Error Model:
    -------------------------
    Distribution: Name = 'Gaussian'
       Intercept: NaN
               P: 3
               D: 0
               Q: 2
              AR: {NaN NaN NaN} at Lags [1 2 3]
             SAR: {}
              MA: {NaN NaN} at Lags [1 2]
             SMA: {}
        Variance: NaN
```

The software sets each parameter to `NaN`, and the innovation distribution to `Gaussian`. The AR coefficients are at lags 1 through 3, and the MA coefficients are at lags 1 and 2.

Pass `Mdl` into `estimate` with data to estimate the parameters set to `NaN`. The `regARIMA` model sets `Beta` to `[]` and does not display it. If you pass a matrix of predictors (X_t) into `estimate`, then `estimate` estimates `Beta`. The `estimate` function infers the number of regression coefficients in `Beta` from the number of columns in X_t.

Tasks such as simulation and forecasting using `simulate` and `forecast` do not accept models with at least one `NaN` for a parameter value. Use dot notation to modify parameter values.

ARMA Error Model Without an Intercept

This example shows how to specify a regression model with ARMA errors without a regression intercept.

Specify the default regression model with ARMA(3,2) errors:

$$y_t = X_t\beta + u_t$$
$$u_t = a_1u_{t-1} + a_2u_{t-2} + a_3u_{t-3} + \varepsilon_t + b_1\varepsilon_{t-1} + b_2\varepsilon_{t-2}.$$

```
Mdl = regARIMA('ARLags',1:3,'MALags',1:2,'Intercept',0)
```

```
Mdl =

    ARIMA(3,0,2) Error Model:
    -------------------------
    Distribution: Name = 'Gaussian'
       Intercept: 0
               P: 3
               D: 0
               Q: 2
              AR: {NaN NaN NaN} at Lags [1 2 3]
             SAR: {}
              MA: {NaN NaN} at Lags [1 2]
             SMA: {}
        Variance: NaN
```

The software sets `Intercept` to 0, but all other parameters in `Mdl` are `NaN` values by default.

Since `Intercept` is not a `NaN`, it is an equality constraint during estimation. In other words, if you pass `Mdl` and data into `estimate`, then `estimate` sets `Intercept` to 0 during estimation.

You can modify the properties of `Mdl` using dot notation.

ARMA Error Model with Nonconsecutive Lags

This example shows how to specify a regression model with ARMA errors, where the nonzero ARMA terms are at nonconsecutive lags.

Specify the regression model with ARMA(8,4) errors:

$$y_t = c + X_t\beta + u_t$$
$$u_t = a_1u_1 + a_4u_4 + a_8u_8 + \varepsilon_t + b_1\varepsilon_{t-1} + b_4\varepsilon_{t-4}.$$

```
Mdl = regARIMA('ARLags',[1,4,8],'MALags',[1,4])
```

```
Mdl =

    ARIMA(8,0,4) Error Model:
    -------------------------
    Distribution: Name = 'Gaussian'
       Intercept: NaN
               P: 8
               D: 0
               Q: 4
              AR: {NaN NaN NaN} at Lags [1 4 8]
             SAR: {}
              MA: {NaN NaN} at Lags [1 4]
             SMA: {}
        Variance: NaN
```

The AR coefficients are at lags 1, 4, and 8, and the MA coefficients are at lags 1 and 4. The software sets the interim lags to 0.

Pass `Mdl` and data into `estimate`. The software estimates all parameters that have the value `NaN`. Then `estimate` holds all interim lag coefficients to 0 during estimation.

Known Parameter Values for a Regression Model with ARMA Errors

This example shows how to specify values for all parameters of a regression model with ARMA errors.

Specify the regression model with ARMA(3,2) errors:

$$y_t = X_t \begin{bmatrix} 2.5 \\ -0.6 \end{bmatrix} + u_t$$
$$u_t = 0.7u_{t-1} - 0.3u_{t-2} + 0.1u_{t-3} + \varepsilon_t + 0.5\varepsilon_{t-1} + 0.2\varepsilon_{t-2},$$

where ε_t is Gaussian with unit variance.

```
Mdl = regARIMA('Intercept',0,'Beta',[2.5; -0.6],...
    'AR',{0.7, -0.3, 0.1},'MA',{0.5, 0.2},'Variance',1)
```

```
Mdl =

    Regression with ARIMA(3,0,2) Error Model:
    ----------------------------------------
    Distribution: Name = 'Gaussian'
       Intercept: 0
            Beta: [2.5 -0.6]
               P: 3
               D: 0
               Q: 2
              AR: {0.7 -0.3 0.1} at Lags [1 2 3]
             SAR: {}
              MA: {0.5 0.2} at Lags [1 2]
             SMA: {}
        Variance: 1
```

The parameters in `Mdl` do not contain `NaN` values, and therefore there is no need to estimate `Mdl` using `estimate`. However, you can simulate or forecast responses from `Mdl` using `simulate` or `forecast`.

Regression Model with ARMA Errors and t Innovations

This example shows how to set the innovation distribution of a regression model with ARMA errors to a t distribution.

Specify the regression model with ARMA(3,2) errors:

$$y_t = X_t \begin{bmatrix} 2.5 \\ -0.6 \end{bmatrix} + u_t$$
$$u_t = 0.7u_{t-1} - 0.3u_{t-2} + 0.1u_{t-3} + \varepsilon_t + 0.5\varepsilon_{t-1} + 0.2\varepsilon_{t-2},$$

where ε_t has a t distribution with the default degrees of freedom and unit variance.

```
Mdl = regARIMA('Intercept',0,'Beta',[2.5; -0.6],...
    'AR',{0.7, -0.3, 0.1},'MA',{0.5, 0.2},'Variance',1,...
    'Distribution','t')
```

```
Mdl =

    Regression with ARIMA(3,0,2) Error Model:
    ----------------------------------------
    Distribution: Name = 't', DoF = NaN
       Intercept: 0
            Beta: [2.5 -0.6]
               P: 3
               D: 0
               Q: 2
              AR: {0.7 -0.3 0.1} at Lags [1 2 3]
             SAR: {}
              MA: {0.5 0.2} at Lags [1 2]
             SMA: {}
        Variance: 1
```

The default degrees of freedom is `NaN`. If you don't know the degrees of freedom, then you can estimate it by passing `Mdl` and the data to `estimate`.

Specify a t_5 distribution.

```
Mdl.Distribution = struct('Name','t','DoF',5)
```

```
Mdl =

    Regression with ARIMA(3,0,2) Error Model:
    ----------------------------------------
    Distribution: Name = 't', DoF = 5
       Intercept: 0
            Beta: [2.5 -0.6]
               P: 3
               D: 0
               Q: 2
              AR: {0.7 -0.3 0.1} at Lags [1 2 3]
             SAR: {}
              MA: {0.5 0.2} at Lags [1 2]
             SMA: {}
```

```
    Variance: 1
```

You can simulate or forecast responses from `Mdl` using `simulate` or `forecast` because `Mdl` is completely specified.

In applications, such as simulation, the software normalizes the random t innovations. In other words, `Variance` overrides the theoretical variance of the t random variable (which is `DoF/(DoF - 2)`), but preserves the kurtosis of the distribution.

See Also

estimate | forecast | regARIMA | simulate

Related Examples

- "Specify Regression Models with ARIMA Errors Using regARIMA" on page 4-10
- "Specify the Default Regression Model with ARIMA Errors" on page 4-20
- "Specify Regression Models with AR Errors" on page 4-29
- "Specify Regression Models with MA Errors" on page 4-35
- "Specify Regression Models with ARIMA Errors" on page 4-48
- "Specify Regression Models with SARIMA Errors" on page 4-55
- "Specify the ARIMA Error Model Innovation Distribution" on page 4-69

More About

- "Regression Models with Time Series Errors" on page 4-6

Specify Regression Models with ARIMA Errors

In this section...

Default Regression Model with ARIMA Errors

This example shows how to apply the shorthand `regARIMA(p,D,q)` syntax to specify the regression model with ARIMA errors.

Specify the default regression model with ARIMA(3,1,2) errors:

$$y_t = c + X_t\beta + u_t$$
$$\left(1 - a_1L - a_2L^2 - a_3L^3\right)(1 - L)\,u_t = \left(1 + b_1L + b_2L^2\right)\varepsilon_t.$$

```
Mdl = regARIMA(3,1,2)
```

```
Mdl =

    ARIMA(3,1,2) Error Model:
    -------------------------
    Distribution: Name = 'Gaussian'
       Intercept: NaN
               P: 4
               D: 1
               Q: 2
              AR: {NaN NaN NaN} at Lags [1 2 3]
             SAR: {}
              MA: {NaN NaN} at Lags [1 2]
             SMA: {}
        Variance: NaN
```

The software sets each parameter to `NaN`, and the innovation distribution to `Gaussian`. The AR coefficients are at lags 1 through 3, and the MA coefficients are at lags 1 and

2. The property `P` = $p + D = 3 + 1 = 4$. Therefore, the software requires at least four presample values to initialize the time series.

Pass `Mdl` into `estimate` with data to estimate the parameters set to `NaN`. The `regARIMA` model sets `Beta` to `[]` and does not display it. If you pass a matrix of predictors (X_t) into `estimate`, then `estimate` estimates `Beta`. The `estimate` function infers the number of regression coefficients in `Beta` from the number of columns in X_t.

Tasks such as simulation and forecasting using `simulate` and `forecast` do not accept models with at least one `NaN` for a parameter value. Use dot notation to modify parameter values.

Be aware that the regression model intercept (`Intercept`) is not identifiable in regression models with ARIMA errors. If you want to `estimate` `Mdl`, then you must set `Intercept` to a value using, for example, dot notation. Otherwise, `estimate` might return a spurious estimate of `Intercept`.

ARIMA Error Model Without an Intercept

This example shows how to specify a regression model with ARIMA errors without a regression intercept.

Specify the default regression model with ARIMA(3,1,2) errors:

$$y_t = X_t\beta + u_t$$
$$\left(1 - a_1L - a_2L^2 - a_3L^3\right)(1 - L)\,u_t = \left(1 + b_1L + b_2L^2\right)\varepsilon_t.$$

```
Mdl = regARIMA('ARLags',1:3,'MALags',1:2,'D',1,'Intercept',0)
```

```
Mdl =

    ARIMA(3,1,2) Error Model:
    -------------------------
    Distribution: Name = 'Gaussian'
       Intercept: 0
               P: 4
               D: 1
               Q: 2
              AR: {NaN NaN NaN} at Lags [1 2 3]
```

```
             SAR: {}
              MA: {NaN NaN} at Lags [1 2]
             SMA: {}
        Variance: NaN
```

The software sets `Intercept` to 0, but all other parameters in `Mdl` are `NaN` values by default.

Since `Intercept` is not a `NaN`, it is an equality constraint during estimation. In other words, if you pass `Mdl` and data into `estimate`, then `estimate` sets `Intercept` to 0 during estimation.

In general, if you want to use `estimate` to estimate a regression models with ARIMA errors where $D > 0$ or $s > 0$, then you must set `Intercept` to a value before estimation.

You can modify the properties of `Mdl` using dot notation.

ARIMA Error Model with Nonconsecutive Lags

This example shows how to specify a regression model with ARIMA errors, where the nonzero AR and MA terms are at nonconsecutive lags.

Specify the regression model with ARIMA(8,1,4) errors:

$$y_t = X_t\beta + u_t$$
$$(1 - a_1L - a_4L^4 - a_8L^8)(1 - L)u_t = (1 + b_1L + b_4L^4)\varepsilon_t.$$

```
Mdl = regARIMA('ARLags',[1,4,8],'D',1,'MALags',[1,4],...
    'Intercept',0)
```

```
Mdl =

    ARIMA(8,1,4) Error Model:
    -------------------------
    Distribution: Name = 'Gaussian'
       Intercept: 0
               P: 9
               D: 1
               Q: 4
              AR: {NaN NaN NaN} at Lags [1 4 8]
             SAR: {}
```

```
            MA: {NaN NaN} at Lags [1 4]
           SMA: {}
      Variance: NaN
```

The AR coefficients are at lags 1, 4, and 8, and the MA coefficients are at lags 1 and 4. The software sets the interim lags to 0.

Pass `Mdl` and data into `estimate`. The software estimates all parameters that have the value `NaN`. Then `estimate` holds all interim lag coefficients to 0 during estimation.

Known Parameter Values for a Regression Model with ARIMA Errors

This example shows how to specify values for all parameters of a regression model with ARIMA errors.

Specify the regression model with ARIMA(3,1,2) errors:

$$y_t = X_t \begin{bmatrix} 2.5 \\ -0.6 \end{bmatrix} + u_t$$
$$\left(1 - 0.7L + 0.3L^2 - 0.1L^3\right)(1 - L)\,u_t = \left(1 + 0.5L + 0.2L^2\right)\varepsilon_t,$$

where ε_t is Gaussian with unit variance.

```
Mdl = regARIMA('Intercept',0,'Beta',[2.5; -0.6],...
    'AR',{0.7, -0.3, 0.1},'MA',{0.5, 0.2},...
    'Variance',1,'D',1)
```

```
Mdl =

    Regression with ARIMA(3,1,2) Error Model:
    ----------------------------------------
    Distribution: Name = 'Gaussian'
       Intercept: 0
            Beta: [2.5 -0.6]
               P: 4
               D: 1
               Q: 2
              AR: {0.7 -0.3 0.1} at Lags [1 2 3]
             SAR: {}
              MA: {0.5 0.2} at Lags [1 2]
             SMA: {}
```

```
        Variance: 1
```

The parameters in `Mdl` do not contain `NaN` values, and therefore there is no need to estimate it. However, you can simulate or forecast responses by passing `Mdl` to `simulate` or `forecast`.

Regression Model with ARIMA Errors and t Innovations

This example shows how to set the innovation distribution of a regression model with ARIMA errors to a t distribution.

Specify the regression model with ARIMA(3,1,2) errors:

$$y_t = X_t \begin{bmatrix} 2.5 \\ -0.6 \end{bmatrix} + u_t$$
$$\left(1 - 0.7L + 0.3L^2 - 0.1L^3\right)(1 - L)\,u_t = \left(1 + 0.5L + 0.2L^2\right)\varepsilon_t,$$

where ε_t has a t distribution with the default degrees of freedom and unit variance.

```
Mdl = regARIMA('Intercept',0,'Beta',[2.5; -0.6],...
    'AR',{0.7, -0.3, 0.1},'MA',{0.5, 0.2},'Variance',1,...
    'Distribution','t','D',1)
```

```
Mdl =

    Regression with ARIMA(3,1,2) Error Model:
    ----------------------------------------
    Distribution: Name = 't', DoF = NaN
       Intercept: 0
            Beta: [2.5 -0.6]
               P: 4
               D: 1
               Q: 2
              AR: {0.7 -0.3 0.1} at Lags [1 2 3]
             SAR: {}
              MA: {0.5 0.2} at Lags [1 2]
             SMA: {}
        Variance: 1
```

The default degrees of freedom is `NaN`. If you don't know the degrees of freedom, then you can estimate it by passing `Mdl` and the data to `estimate`.

Specify a t_{10} distribution.

```
Mdl.Distribution = struct('Name','t','DoF',10)
```

```
Mdl =

    Regression with ARIMA(3,1,2) Error Model:
    ----------------------------------------
    Distribution: Name = 't', DoF = 10
       Intercept: 0
            Beta: [2.5 -0.6]
               P: 4
               D: 1
               Q: 2
              AR: {0.7 -0.3 0.1} at Lags [1 2 3]
             SAR: {}
              MA: {0.5 0.2} at Lags [1 2]
             SMA: {}
        Variance: 1
```

You can simulate or forecast responses by passing `Mdl` to `simulate` or `forecast` because `Mdl` is completely specified.

In applications, such as simulation, the software normalizes the random t innovations. In other words, `Variance` overrides the theoretical variance of the t random variable (which is `DoF`/(`DoF` - 2)), but preserves the kurtosis of the distribution.

See Also

estimate | forecast | regARIMA | simulate

Related Examples

- "Specify Regression Models with ARIMA Errors Using regARIMA" on page 4-10
- "Specify the Default Regression Model with ARIMA Errors" on page 4-20
- "Specify Regression Models with AR Errors" on page 4-29
- "Specify Regression Models with MA Errors" on page 4-35
- "Specify Regression Models with ARMA Errors" on page 4-42
- "Specify Regression Models with SARIMA Errors" on page 4-55
- "Specify the ARIMA Error Model Innovation Distribution" on page 4-69

More About

- "Regression Models with Time Series Errors" on page 4-6

Specify Regression Models with SARIMA Errors

In this section...

SARMA Error Model Without an Intercept

This example shows how to specify a regression model with SARMA errors without a regression intercept.

Specify the default regression model with $\mathrm{SARMA}(1,1)\times(2,1,1)_4$ errors:

$$y_t = X_t\beta + u_t$$
$$(1-a_1L)\left(1-A_4L^4-A_8L^8\right)\left(1-L^4\right)u_t = (1+b_1L)\left(1+B_4L^4\right)\varepsilon_t.$$

```
Mdl = regARIMA('ARLags',1,'SARLags',[4, 8],...
    'Seasonality',4,'MALags',1,'SMALags',4,'Intercept',0)
```

```
Mdl =

    ARIMA(1,0,1) Error Model Seasonally Integrated with Seasonal AR(8) and MA(4):
    --------------------------------------------------------------------------------
    Distribution: Name = 'Gaussian'
       Intercept: 0
               P: 13
               D: 0
               Q: 5
              AR: {NaN} at Lags [1]
             SAR: {NaN NaN} at Lags [4 8]
              MA: {NaN} at Lags [1]
             SMA: {NaN} at Lags [4]
     Seasonality: 4
        Variance: NaN
```

The name-value pair argument:

- `'ARLags',1` specifies which lags have nonzero coefficients in the nonseasonal autoregressive polynomial, so $a(L) = (1 - a_1 L)$.
- `'SARLags',[4 8]` specifies which lags have nonzero coefficients in the seasonal autoregressive polynomial, so $A(L) = (1 - A_4 L^4 - A_8 L^8)$.
- `'MALags',1` specifies which lags have nonzero coefficients in the nonseasonal moving average polynomial, so $b(L) = (1 + b_1 L)$.
- `'SMALags',4` specifies which lags have nonzero coefficients in the seasonal moving average polynomial, so $B(L) = (1 + B_4 L^4)$.
- `'Seasonality',4` specifies the degree of seasonal integration and corresponds to $(1 - L^4)$.

The software sets `Intercept` to 0, but all other parameters in `Mdl` are `NaN` values by default.

Property `P` = $p + D + p_s$ + s = 1 + 0 + 8 + 4 = 13, and property `Q` = $q + q_s$ = 1 + 4 = 5. Therefore, the software requires at least 13 presample observation to initialize `Mdl`.

Since `Intercept` is not a `NaN`, it is an equality constraint during estimation. In other words, if you pass `Mdl` and data into `estimate`, then `estimate` sets `Intercept` to 0 during estimation.

You can modify the properties of `Mdl` using dot notation.

Be aware that the regression model intercept (`Intercept`) is not identifiable in regression models with ARIMA errors. If you want to estimate `Mdl`, then you must set `Intercept` to a value using, for example, dot notation. Otherwise, `estimate` might return a spurious estimate of `Intercept`.

Known Parameter Values for a Regression Model with SARIMA Errors

This example shows how to specify values for all parameters of a regression model with SARIMA errors.

Specify the regression model with $\text{SARIMA}(1,1,1) \times (1,1,0)_{12}$ errors:

$$y_t = X_t\beta + u_t$$
$$(1-0.2L)(1-L)\left(1-0.25L^{12}-0.1L^{24}\right)\left(1-L^{12}\right)u_t = (1+0.15L)\varepsilon_t,$$

where ε_t is Gaussian with unit variance.

```
Mdl = regARIMA('AR',0.2,'SAR',{0.25, 0.1},'SARLags',[12 24],...
    'D',1,'Seasonality',12,'MA',0.15,'Intercept',0,'Variance',1)
```

```
Mdl =

    ARIMA(1,1,1) Error Model Seasonally Integrated with Seasonal AR(24):
    ----------------------------------------------------------------------
    Distribution: Name = 'Gaussian'
       Intercept: 0
               P: 38
               D: 1
               Q: 1
              AR: {0.2} at Lags [1]
             SAR: {0.25 0.1} at Lags [12 24]
              MA: {0.15} at Lags [1]
             SMA: {}
     Seasonality: 12
        Variance: 1
```

The parameters in `Mdl` do not contain `NaN` values, and therefore there is no need to estimate `Mdl`. However, you can simulate or forecast responses by passing `Mdl` to `simulate` or `forecast`.

Regression Model with SARIMA Errors and t Innovations

This example shows how to set the innovation distribution of a regression model with SARIMA errors to a t distribution.

Specify the regression model with $\text{SARIMA}(1,1,1)\times(1,1,0)_{12}$ errors:

$$y_t = X_t\beta + u_t$$
$$(1-0.2L)(1-L)\left(1-0.25L^{12}-0.1L^{24}\right)\left(1-L^{12}\right)u_t = (1+0.15L)\varepsilon_t,$$

where ε_t has a t distribution with the default degrees of freedom and unit variance.

```
Mdl = regARIMA('AR',0.2,'SAR',{0.25, 0.1},'SARLags',[12 24],...
    'D',1,'Seasonality',12,'MA',0.15,'Intercept',0,...
```

```
  'Variance',1,'Distribution','t')

Mdl =

    ARIMA(1,1,1) Error Model Seasonally Integrated with Seasonal AR(24):
    ---------------------------------------------------------------------
    Distribution: Name = 't', DoF = NaN
       Intercept: 0
               P: 38
               D: 1
               Q: 1
              AR: {0.2} at Lags [1]
             SAR: {0.25 0.1} at Lags [12 24]
              MA: {0.15} at Lags [1]
             SMA: {}
     Seasonality: 12
        Variance: 1
```

The default degrees of freedom is `NaN`. If you don't know the degrees of freedom, then you can estimate it by passing `Mdl` and the data to `estimate`.

Specify a t_{10} distribution.

```
Mdl.Distribution = struct('Name','t','DoF',10)

Mdl =

    ARIMA(1,1,1) Error Model Seasonally Integrated with Seasonal AR(24):
    ---------------------------------------------------------------------
    Distribution: Name = 't', DoF = 10
       Intercept: 0
               P: 38
               D: 1
               Q: 1
              AR: {0.2} at Lags [1]
             SAR: {0.25 0.1} at Lags [12 24]
              MA: {0.15} at Lags [1]
             SMA: {}
     Seasonality: 12
        Variance: 1
```

You can simulate or forecast responses by passing `Mdl` to `simulate` or `forecast` because `Mdl` is completely specified.

In applications, such as simulation, the software normalizes the random t innovations. In other words, `Variance` overrides the theoretical variance of the t random variable (which is `DoF/(DoF - 2)`), but preserves the kurtosis of the distribution.

See Also

estimate | forecast | regARIMA | simulate

Related Examples

- "Specify Regression Models with ARIMA Errors Using regARIMA" on page 4-10
- "Specify the Default Regression Model with ARIMA Errors" on page 4-20
- "Specify Regression Models with AR Errors" on page 4-29
- "Specify Regression Models with MA Errors" on page 4-35
- "Specify Regression Models with ARMA Errors" on page 4-42
- "Specify a Regression Model with SARIMA Errors" on page 4-60
- "Specify the ARIMA Error Model Innovation Distribution" on page 4-69

More About

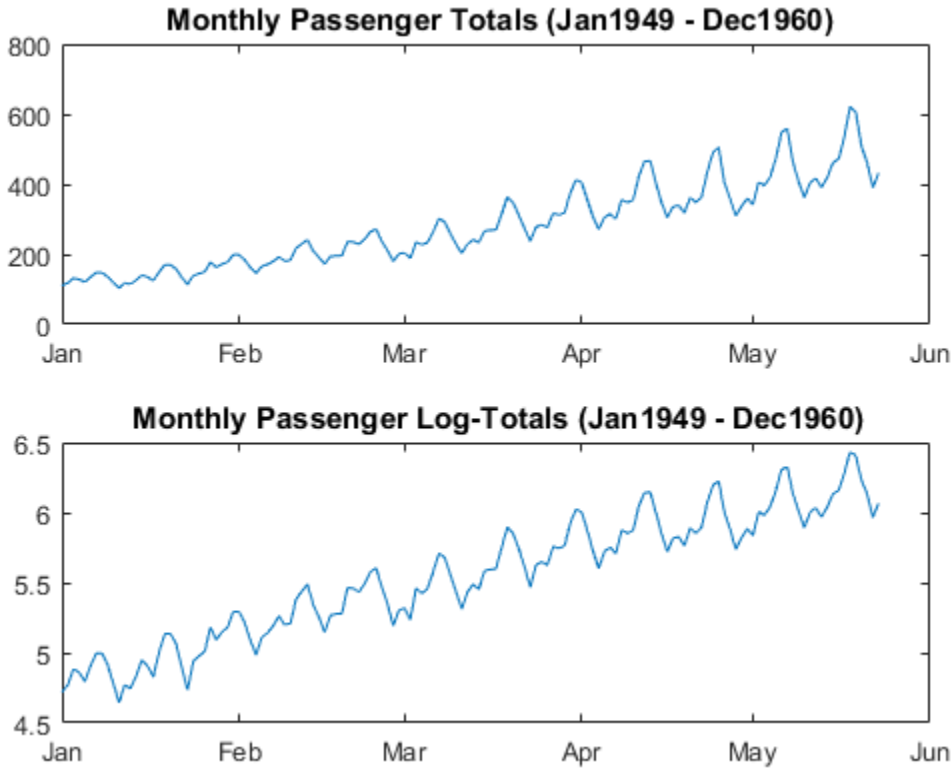
- "Regression Models with Time Series Errors" on page 4-6

Specify a Regression Model with SARIMA Errors

This example shows how to specify a regression model with multiplicative seasonal ARIMA errors.

Load the Airline data set from the MATLAB® root folder, and load the recession data set. Plot the monthly passenger totals and log-totals.

```
load(fullfile(matlabroot,'examples','econ','Data_Airline.mat'))
load Data_Recessions;

y = Data;
logY = log(y);

figure
subplot(2,1,1)
plot(y)
title('{\bf Monthly Passenger Totals (Jan1949 - Dec1960)}')
datetick
subplot(2,1,2)
plot(log(y))
title('{\bf Monthly Passenger Log-Totals (Jan1949 - Dec1960)}')
datetick
```



The log transformation seems to linearize the time series.

Construct this predictor, which is whether the country was in a recession during the sampled period. 0 means the country was not in a recession, and 1 means that it was in a recession.

```
X = zeros(numel(dates),1); % Preallocation
for j = 1:size(Recessions,1)
    X(dates >= Recessions(j,1) & dates <= Recessions(j,2)) = 1;
end
```

Fit the simple linear regression model,

$$y_t = c + X_t\beta + u_t$$

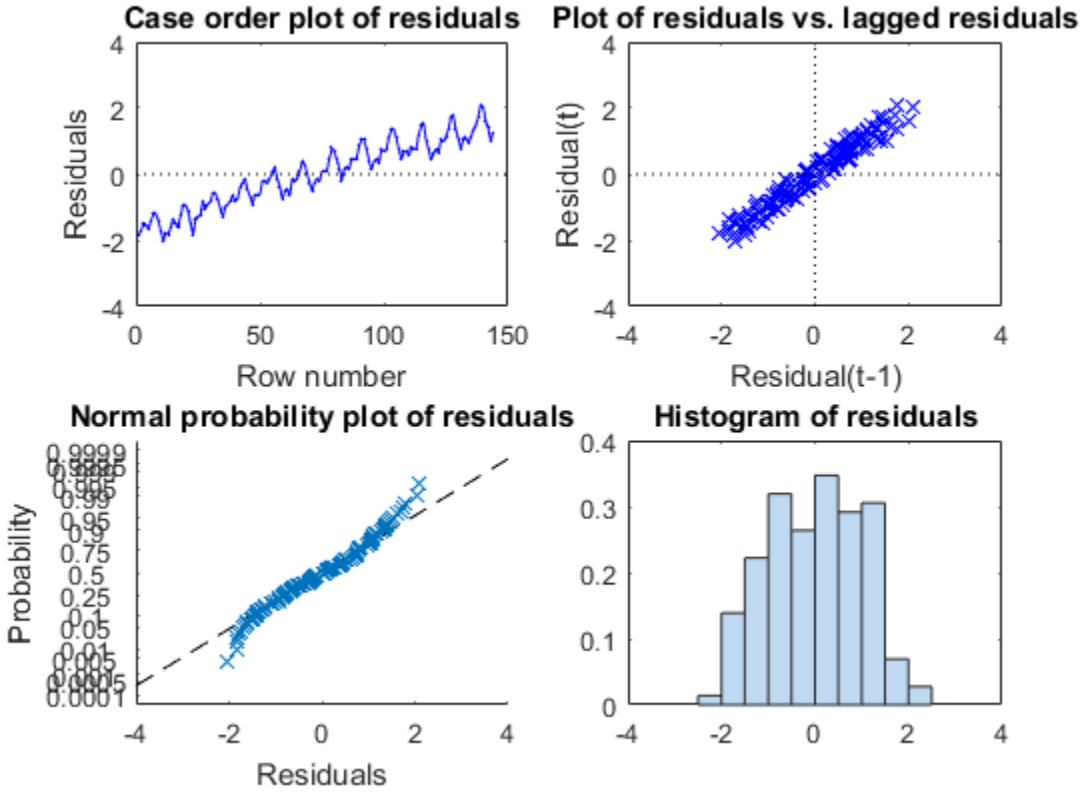
to the data.

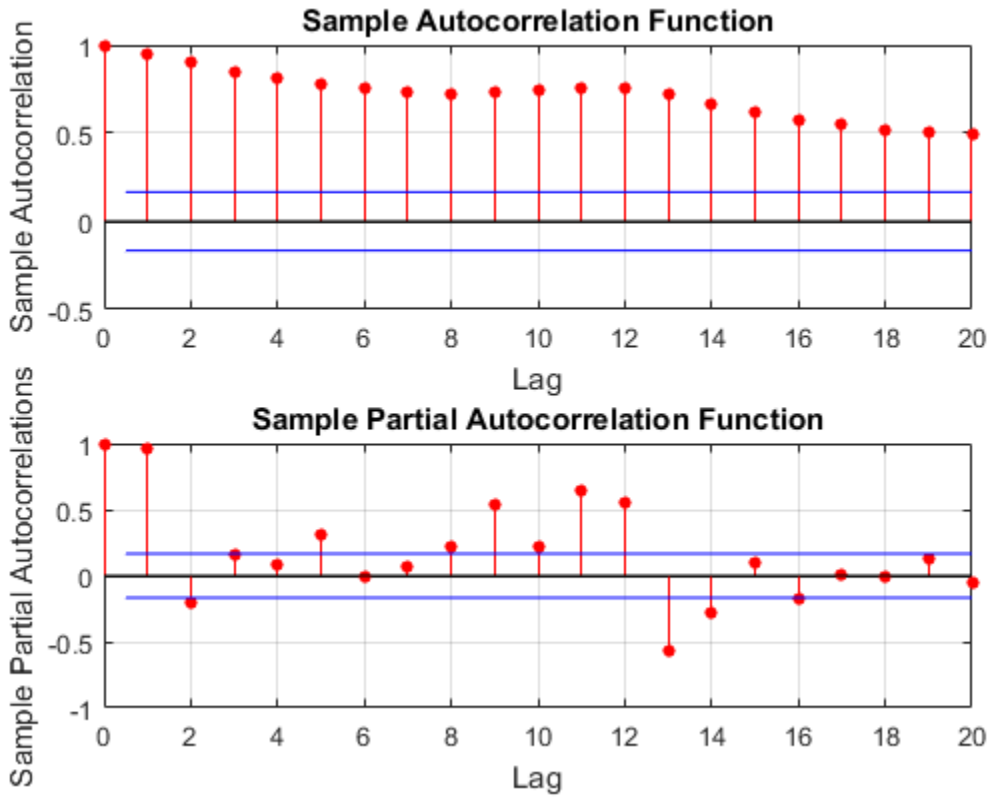
```
Fit = fitlm(X,logY);
```

`Fit` is a `LinearModel` that contains the least squares estimates.

Perform a residual diagnosis by plotting the residuals several ways.

```
figure
subplot(2,2,1)
plotResiduals(Fit,'caseorder','ResidualType','Standardized',...
    'LineStyle','-','MarkerSize',0.5)
subplot(2,2,2)
plotResiduals(Fit,'lagged','ResidualType','Standardized')
subplot(2,2,3)
plotResiduals(Fit,'probability','ResidualType','Standardized')
subplot(2,2,4)
plotResiduals(Fit,'histogram','ResidualType','Standardized')

r = Fit.Residuals.Standardized;
figure
subplot(2,1,1)
autocorr(r)
subplot(2,1,2)
parcorr(r)
```


Case order plot of residuals
Residuals
Row number
Plot of residuals vs. lagged residuals
Residual(t)
Residual(t-1)
Normal probability plot of residuals
Probability
Residuals
Histogram of residuals



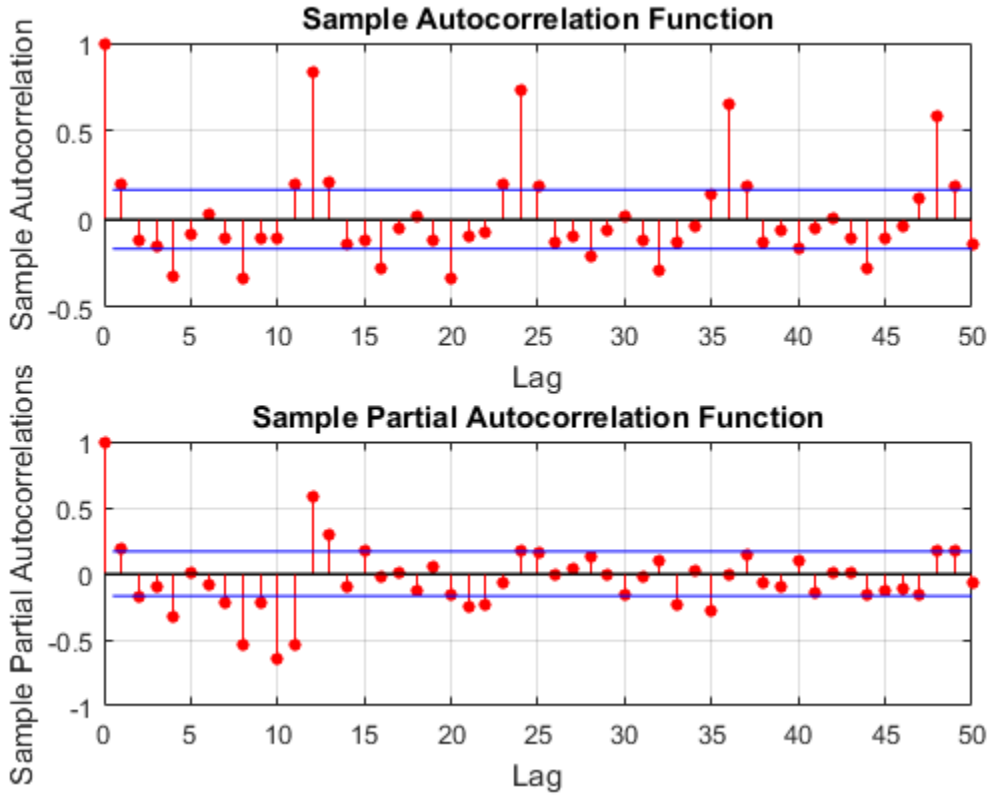
The residual plots indicate that the residuals are autocorrelated. The probability plot and histogram seem to indicate that the residuals are Gaussian.

The ACF of the residuals confirms that they are autocorrelated.

Take the 1st difference of the residuals and plot the ACF and PACF of the differenced residuals.

```
dR = diff(r);

figure
subplot(2,1,1)
autocorr(dR,50)
subplot(2,1,2)
```

```
parcorr(dR,50)
```

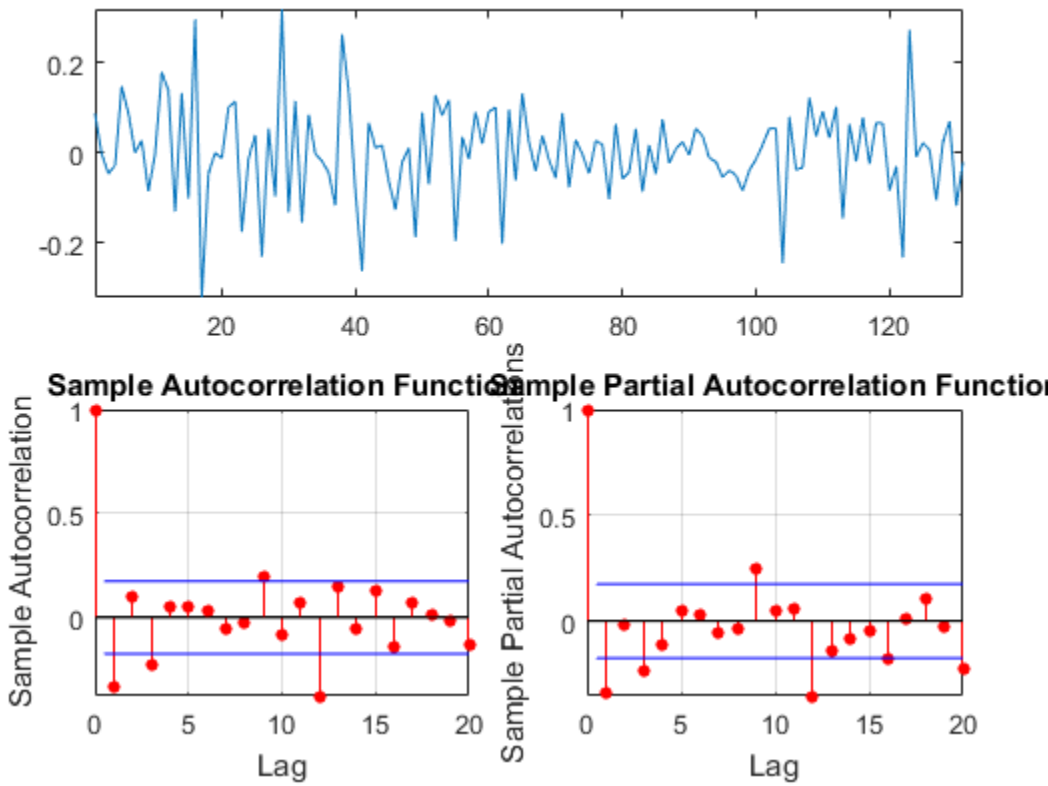


The ACF shows that there are significantly large autocorrelations, particularly at every 12th lag. This indicates that the residuals have 12th degree seasonal integration.

Take the first and 12th differences of the residuals. Plot the differenced residuals, and their ACF and PACF.

```
DiffPoly = LagOp([1 -1]);
SDiffPoly = LagOp([1 -1],'Lags',[0, 12]);
diffR = filter(DiffPoly*SDiffPoly,r);

figure
```

```
subplot(2,1,1)
plot(diffR)
axis tight
subplot(2,2,3)
autocorr(diffR)
axis tight
subplot(2,2,4)
parcorr(diffR)
axis tight
```



The residuals resemble white noise (with possible heteroscedasticity). According to Box and Jenkins (1994), Chapter 9, the ACF and PACF indicate that the unconditional disturbances are an $\mathrm{IMA}(0,1,1) \times (0,1,1)_{12}$ model.

Specify the regression model with $\mathrm{IMA}(0,1,1) \times (0,1,1)_{12}$ errors:

$$y_t = X_t\beta + u_t$$
$$(1-L)\left(1-L^{12}\right)u_t = (1+b_1L)\left(1+B_{12}L^{12}\right)\varepsilon_t.$$

```
Mdl = regARIMA('MALags',1,'D',1,'Seasonality',12,'SMALags',12)
```

```
Mdl =

    ARIMA(0,1,1) Error Model Seasonally Integrated with Seasonal MA(12):
    --------------------------------------------------------------------
    Distribution: Name = 'Gaussian'
       Intercept: NaN
               P: 13
               D: 1
               Q: 13
              AR: {}
             SAR: {}
              MA: {NaN} at Lags [1]
             SMA: {NaN} at Lags [12]
     Seasonality: 12
        Variance: NaN
```

`Mdl` is a regression model with $\mathrm{IMA}(0,1,1) \times (0,1,1)_{12}$ errors. By default, the innovations are Gaussian, and all parameters are `NaN`. The property:

- `P` = $p + D + sp_s + s = 0 + 1 + 0 + 12 = 13$.
- `Q` = $q + sq_s = 1 + 12 = 13$.

Therefore, the software requires at least 13 presample observations to initialize the model.

Pass `Mdl`, `y`, and `X` into `estimate` to estimate the model. In order to simulate or forecast responses using `simulate` or `forecast`, you need to set values to all parameters.

References:

Box, G. E. P., G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis: Forecasting and Control*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

See Also

estimate | forecast | LinearModel | regARIMA | simulate

Related Examples

- “Specify Regression Models with ARIMA Errors Using regARIMA” on page 4-10
- “Specify the Default Regression Model with ARIMA Errors” on page 4-20
- “Specify Regression Models with AR Errors” on page 4-29
- “Specify Regression Models with MA Errors” on page 4-35
- “Specify Regression Models with ARMA Errors” on page 4-42
- “Specify Regression Models with SARIMA Errors” on page 4-55
- “Specify the ARIMA Error Model Innovation Distribution” on page 4-69

More About

- “Regression Models with Time Series Errors” on page 4-6

Specify the ARIMA Error Model Innovation Distribution

In this section...

About the Innovation Process

A regression model with ARIMA errors has the following general form:

$$y_t = c + X_t\beta + u_t$$
$$a(L)A(L)(1-L)^D\left(1-L^s\right)u_t = b(L)B(L)\varepsilon_t,$$

where

- $t = 1,...,T$.
- y_t is the response series.
- X_t is row t of X, which is the matrix of concatenated predictor data vectors. That is, X_t is observation t of each predictor series.
- c is the regression model intercept.
- β is the regression coefficient.
- u_t is the disturbance series.
- ε_t is the innovations series.
- $L^j y_t = y_{t-j}$.
- $a(L) = \left(1 - a_1L - ... - a_pL^p\right)$, which is the degree p, nonseasonal autoregressive polynomial.
- $A(L) = \left(1 - A_1L - ... - A_{p_s}L^{p_s}\right)$, which is the degree p_s, seasonal autoregressive polynomial.
- $(1-L)^D$, which is the degree D, nonseasonal integration polynomial.

- $\left(1-L^s\right)$, which is the degree s, seasonal integration polynomial.
- $b(L)=\left(1+b_1L+...+b_qL^q\right)$, which is the degree q, nonseasonal moving average polynomial.
- $B(L)=\left(1+B_1L+...+B_{q_s}L^{q_s}\right)$, which is the degree q_s, seasonal moving average polynomial.

Suppose that the unconditional disturbance series (u_t) is a stationary stochastic processes. Then, you can express the second equation in Equation 4-5 as

$$u_t=a^{-1}(L)A^{-1}(L)(1-L)^{-D}(1-L^s)^{-1}b(L)B(L)\varepsilon_t=\Psi(L)\varepsilon_t,$$

where $\Psi(L)$ is an infinite degree lag operator polynomial [1].

The innovation process (ε_t) is an independent and identically distributed (iid), mean 0 process with a known distribution. Econometrics Toolbox generalizes the innovation process to $\varepsilon_t = \sigma z_t$, where z_t is a series of iid random variables with mean 0 and variance 1, and σ^2 is the constant variance of ε_t.

regARIMA models contain two properties that describe the distribution of ε_t:

- `Variance` stores σ^2.
- `Distribution` stores the parametric form of z_t.

Innovation Distribution Options

- The default value of `Variance` is `NaN`, meaning that the innovation variance is unknown. You can assign a positive scalar to `Variance` when you specify the model using the name-value pair argument `'Variance',sigma2` (where `sigma2` = σ^2), or by modifying an existing model using dot notation. Alternatively, you can estimate `Variance` using `estimate`.
- You can specify the following distributions for z_t (using name-value pair arguments or dot notation):
  - Standard Gaussian

- Standardized Student's t with degrees of freedom $\nu > 2$. Specifically,

$$z_t = T_\nu \sqrt{\frac{\nu - 2}{\nu}},$$

  where T_ν is a Student's t distribution with degrees of freedom $\nu > 2$.

The t distribution is useful for modeling innovations that are more extreme than expected under a Gaussian distribution. Such innovation processes have *excess kurtosis*, a more peaked (or heavier tailed) distribution than a Gaussian. Note that for $\nu > 4$, the kurtosis (fourth central moment) of T_ν is the same as the kurtosis of the Standardized Student's t (z_t), i.e., for a t random variable, the kurtosis is scale invariant.

Tip It is good practice to assess the distributional properties of the residuals to determine if a Gaussian innovation distribution (the default distribution) is appropriate for your model.

Specify the Innovation Distribution

`regARIMA` stores the distribution (and degrees of freedom for the t distribution) in the `Distribution` property. The data type of `Distribution` is a `struct` array with potentially two fields: `Name` and `DoF`.

- If the innovations are Gaussian, then the `Name` field is `Gaussian`, and there is no `DoF` field. `regARIMA` sets `Distribution` to `Gaussian` by default.
- If the innovations are t-distributed, then the `Name` field is `t` and the `DoF` field is `NaN` by default, or you can specify a scalar that is greater than 2.

To illustrate specifying the distribution, consider this regression model with AR(2) errors:

$$\begin{aligned} y_t &= c + X_t\beta + u_t \\ u_t &= \alpha_1 u_{t-1} + \alpha_2 u_{t-2} + \varepsilon_t \end{aligned}$$

```
Mdl = regARIMA(2,0,0);
Mdl.Distribution
```

```
ans =

    Name: 'Gaussian'
```

By default, `Distribution` property of `Mdl` is a `struct` array with the field `Name` having the value `Gaussian`.

If you want to specify a *t* innovation distribution, then you can either specify the model using the name-value pair argument `'Distribution','t'`, or use dot notation to modify an existing model.

Specify the model using the name-value pair argument.

```
Mdl = regARIMA('ARLags',1:2,'Distribution','t');
Mdl.Distribution
```

```
ans =

    Name: 't'
     DoF: NaN
```

If you use the name-value pair argument to specify the *t* innovation distribution, then the default degrees of freedom is `NaN`.

You can use dot notation to yield the same result.

```
Mdl = regARIMA(2,0,0);
Mdl.Distribution = 't'
```

```
Mdl =

    ARIMA(2,0,0) Error Model:
    -------------------------
    Distribution: Name = 't', DoF = NaN
       Intercept: NaN
               P: 2
               D: 0
               Q: 0
              AR: {NaN NaN} at Lags [1 2]
             SAR: {}
              MA: {}
```

```
           SMA: {}
      Variance: NaN
```

If the innovation distribution is t_{10}, then you can use dot notation to modify the `Distribution` property of the existing model `Mdl`. You cannot modify the fields of `Distribution` using dot notation, e.g., `Mdl.Distribution.DoF = 10` is not a value assignment. However, you can display the value of the fields using dot notation.

```
Mdl.Distribution = struct('Name','t','DoF',10)
tDistributionDoF = Mdl.Distribution.DoF
```

```
Mdl =

    ARIMA(2,0,0) Error Model:
    -------------------------
    Distribution: Name = 't', DoF = 10
       Intercept: NaN
               P: 2
               D: 0
               Q: 0
              AR: {NaN NaN} at Lags [1 2]
             SAR: {}
              MA: {}
             SMA: {}
        Variance: NaN

tDistributionDoF =

    10
```

Since the `DoF` field is not a `NaN`, it is an equality constraint when you estimate `Mdl` using `estimate`.

Alternatively, you can specify the t_{10} innovation distribution using the name-value pair argument.

```
Mdl = regARIMA('ARLags',1:2,'Constant',0,...
    'Distribution',struct('Name','t','DoF',10))
```

```
Mdl =
```

```
ARIMA(2,0,0) Error Model:
--------------------------
Distribution: Name = 't', DoF = 10
   Intercept: 0
           P: 2
           D: 0
           Q: 0
          AR: {NaN NaN} at Lags [1 2]
         SAR: {}
          MA: {}
         SMA: {}
    Variance: NaN
```

References

[1] Wold, H. *A Study in the Analysis of Stationary Time Series*. Uppsala, Sweden: Almqvist & Wiksell, 1938.

See Also

estimate | forecast | regARIMA | simulate | `struct`

Related Examples

- “Specify Regression Models with ARIMA Errors Using regARIMA” on page 4-10
- “Specify the Default Regression Model with ARIMA Errors” on page 4-20
- “Specify Regression Models with AR Errors” on page 4-29
- “Specify Regression Models with MA Errors” on page 4-35
- “Specify Regression Models with ARMA Errors” on page 4-42
- “Specify Regression Models with SARIMA Errors” on page 4-55

More About

- “Regression Models with Time Series Errors” on page 4-6

Impulse Response for Regression Models with ARIMA Errors

The general form of a regression model with ARIMA errors is:

$$\begin{aligned} y_t &= c + X_t\beta + u_t \\ \mathrm{H}(L)u_t &= \mathrm{N}(L)\varepsilon_t, \end{aligned}$$

where

- $t = 1,\ldots,T$.
- $H(L)$ is the compound autoregressive polynomial.
- $N(L)$ is the compound moving average polynomial.

Solve for u_t in the ARIMA error model to obtain

$$u_t = \mathrm{H}^{-1}(L)\mathrm{N}(L)\varepsilon_t = \psi(L)\varepsilon_t,$$

where $\psi(L) = 1 + \psi_1 L + \psi_2 L^2 + \ldots$ is an infinite degree polynomial.

The coefficient ψ_j is called a *dynamic multiplier* [1]. You can interpret ψ_j as the change in the future response (y_{t+j}) due to a one-time unit change in the current innovation (ε_t) and no changes in future innovations ($\varepsilon_{t+1},\varepsilon_{t+2},\ldots$). That is, the *impulse response function* is

$$\psi_j = \frac{\partial y_{t+j}}{\partial \varepsilon_t}.$$

Equation 4-7 implies that the regression intercept (c) and predictors (X_t) of Equation 4-6 do not impact the impulse response function. In other words, the impulse response function describes the change in the response that is solely due to the one-time unit shock of the innovation ε_t.

- If the series $\{\psi_j\}$ is absolutely summable, then Equation 4-6 is a stationary stochastic process [2].
- If the ARIMA error model is stationary, then the impact on the response due to a change in ε_t is not permanent. That is, the effect of the impulse decays to 0.
- If the ARIMA error model is nonstationary, then the impact on the response due to a change in ε_t persists.

References

[1] Hamilton, J. D. *Time Series Analysis*. Princeton, NJ: Princeton University Press, 1994.

[2] Wold, H. *A Study in the Analysis of Stationary Time Series*. Uppsala, Sweden: Almqvist & Wiksell, 1938.

Plot the Impulse Response of regARIMA Models

Impulse response functions help examine the effects of a unit innovation shock to future values of the response of a time series model, without accounting for the effects of exogenous predictors. For example, if an innovation shock to an aggregate output series, e.g., GDP, is persistent, then GDP is sensitive to such shocks. The examples below show how to plot impulse response functions for regression models with various ARIMA error model structures using impulse.

In this section...

Regression Model with AR Errors

This example shows how to plot the impulse response function for a regression model with AR errors.

Specify the regression model with AR(4) errors:

$$y_t = 2 + X_t \begin{bmatrix} 5 \\ -1 \end{bmatrix} + u_t$$
$$u_t = 0.9u_{t-1} - 0.8u_{t-2} + 0.75u_{t-3} - 0.6u_{t-4} + \varepsilon_t.$$

```
Mdl = regARIMA('Intercept',2,'Beta',[5; -1],'AR',...
    {0.9, -0.8, 0.75, -0.6})
```

```
Mdl =

    Regression with ARIMA(4,0,0) Error Model:
    ----------------------------------------
    Distribution: Name = 'Gaussian'
       Intercept: 2
            Beta: [5 -1]
               P: 4
               D: 0
               Q: 0
```

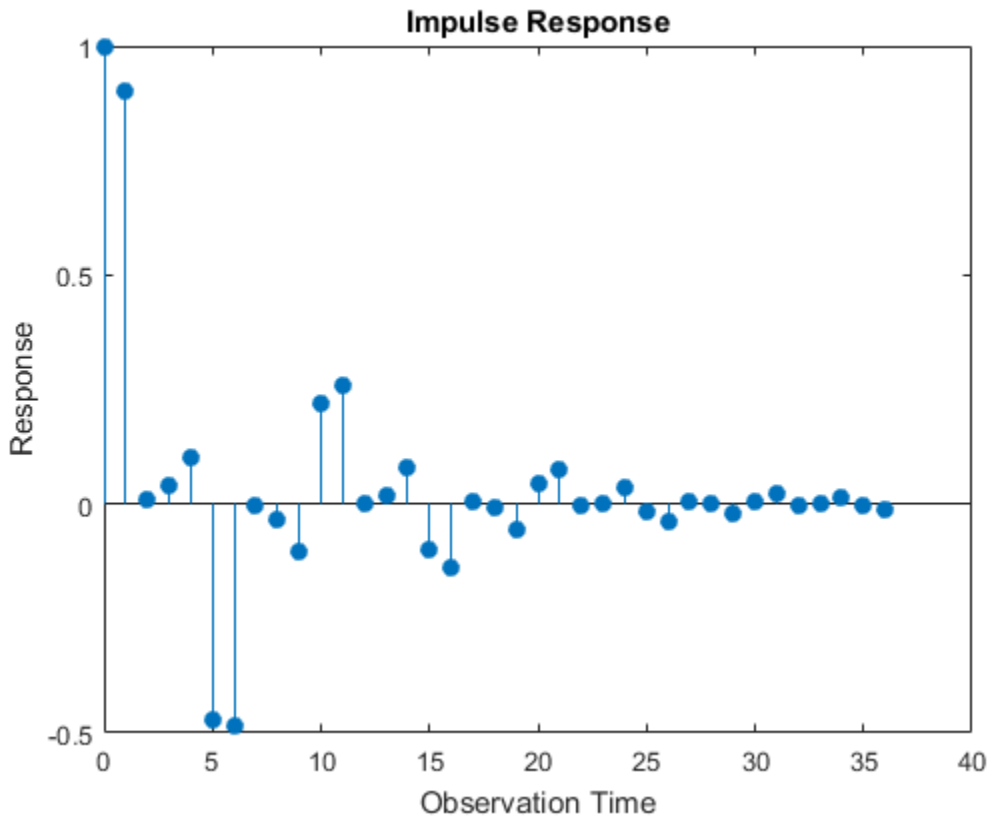
```
        AR: {0.9 -0.8 0.75 -0.6} at Lags [1 2 3 4]
       SAR: {}
        MA: {}
       SMA: {}
  Variance: NaN
```

The dynamic multipliers are absolutely summable because the autoregressive component is stable. Therefore, `Mdl` is stationary.

You do not need to specify the innovation variance.

Plot the impulse response function.

```
impulse(Mdl)
```



The impulse response decays to 0 since `Mdl` defines a stationary error process. The regression component does not impact the impulse responses.

Regression Model with MA Errors

This example shows how to plot a regression model with MA errors.

Specify the regression model with MA(10) errors:

$$y_t = 2 + X_t \begin{bmatrix} 5 \\ -1 \end{bmatrix} + u_t$$
$$u_t = \varepsilon_t + 0.5\varepsilon_{t-2} - 0.4\varepsilon_{t-4} - 0.3\varepsilon_{t-6} + 0.2\varepsilon_{t-8} - 0.1\varepsilon_{t-10}.$$

```
Mdl = regARIMA('Intercept',2,'Beta',[5; -1],...
    'MA',{0.5,-0.4,-0.3,0.2,-0.1},'MALags',[2 4 6 8 10])
```

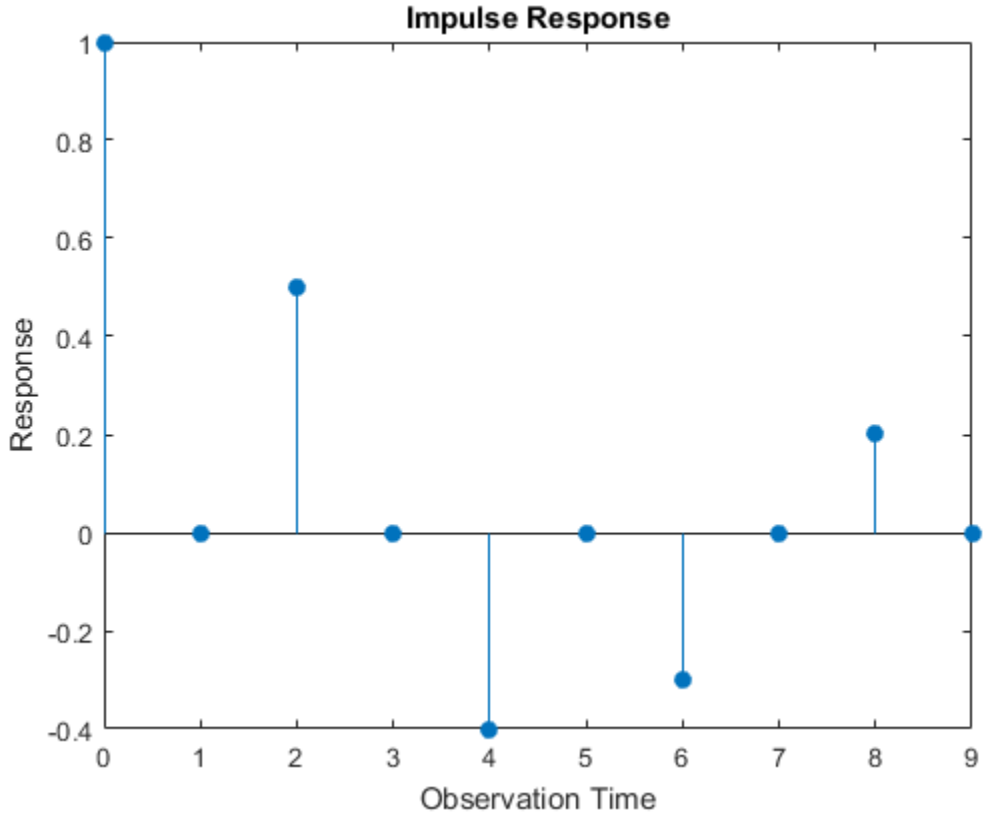
```
Mdl =

    Regression with ARIMA(0,0,10) Error Model:
    ----------------------------------------
    Distribution: Name = 'Gaussian'
       Intercept: 2
            Beta: [5 -1]
               P: 0
               D: 0
               Q: 10
              AR: {}
             SAR: {}
              MA: {0.5 -0.4 -0.3 0.2 -0.1} at Lags [2 4 6 8 10]
             SMA: {}
        Variance: NaN
```

The dynamic multipliers are absolutely summable because the moving average component is invertible. Therefore, `Mdl` is stationary.

You do not need to specify the innovation variance.

Plot the impulse response function for 10 responses.

```
impulse(Mdl,10)
```



The impulse response of an MA error model is simply the MA coefficients at their corresponding lags.

Regression Model with ARMA Errors

This example shows how to plot the impulse response function of a regression model with ARMA errors.

Specify the regression model with ARMA(4,10) errors:

$$y_t = 2 + X_t \begin{bmatrix} 5 \\ -1 \end{bmatrix} + u_t$$

$$\left(1 - 0.9L + 0.8L^2 - 0.75L^3 + 0.6L^4\right) u_t = \left(1 + 0.5L^2 - 0.4L^4 - 0.3L^6 + 0.2L^8 - 0.1L^{10}\right)$$

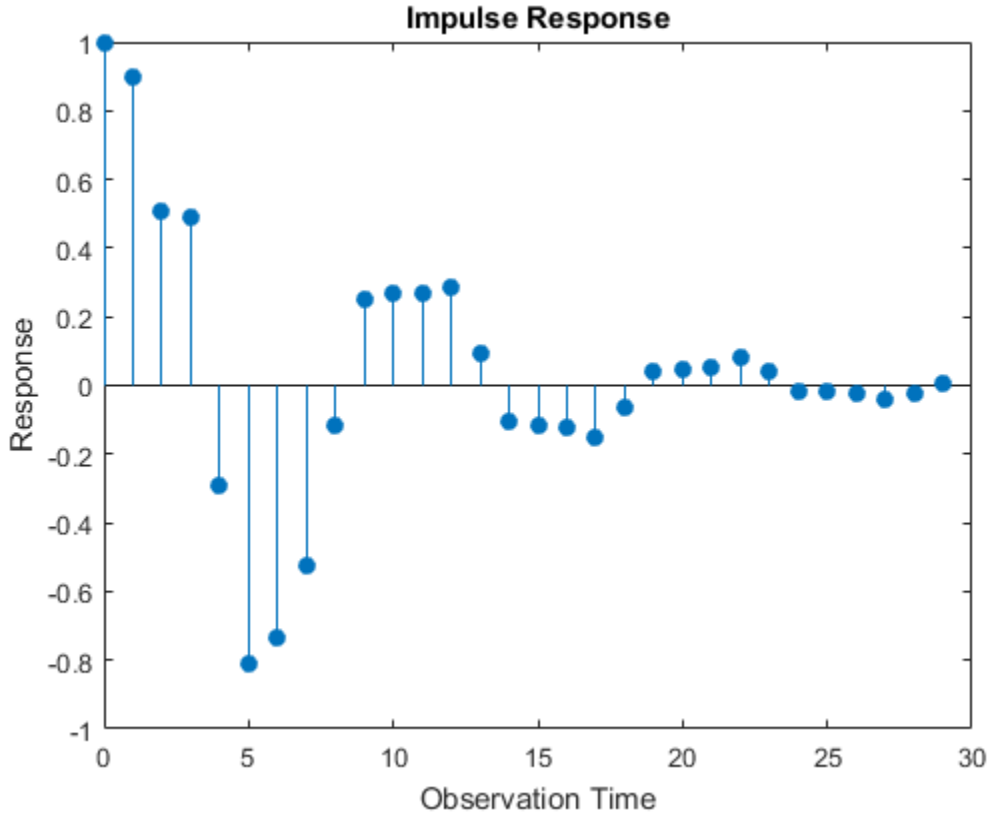
```
Mdl = regARIMA('Intercept',2,'Beta',[5; -1],...
    'AR',{0.9, -0.8, 0.75, -0.6},...
    'MA',{0.5, -0.4, -0.3, 0.2, -0.1},'MALags',[2 4 6 8 10])


Mdl =

    Regression with ARIMA(4,0,10) Error Model:
    ----------------------------------------
    Distribution: Name = 'Gaussian'
       Intercept: 2
            Beta: [5 -1]
               P: 4
               D: 0
               Q: 10
              AR: {0.9 -0.8 0.75 -0.6} at Lags [1 2 3 4]
             SAR: {}
              MA: {0.5 -0.4 -0.3 0.2 -0.1} at Lags [2 4 6 8 10]
             SMA: {}
        Variance: NaN
```

The dynamic multipliers are absolutely summable because the autoregressive component is stable, and the moving average component is invertible. Therefore, `Mdl` defines a stationary error process.

You do not need to specify the innovation variance.

Plot the first 30 impulse responses.

```
impulse(Mdl,30)
```



The impulse response decays to 0 since `Mdl` defines a stationary error process.

Regression Model with ARIMA Errors

This example shows how to plot the impulse response function of a regression model with ARIMA errors.

Specify the regression model with ARIMA(4,1,10) errors:

$$y_t = 2 + X_t \begin{bmatrix} 5 \\ -1 \end{bmatrix} + u_t$$

$$\left(1 - 0.9L + 0.8L^2 - 0.75L^3 + 0.6L^4\right)(1 - L)u_t = \left(1 + 0.5L^2 - 0.4L^4 - 0.3L^6 + 0.2L^8 - 0.1L^{10}\right)\varepsilon_t.$$

```
Mdl = regARIMA('Intercept',2,'Beta',[5; -1],...
    'AR',{0.9, -0.8, 0.75, -0.6},...
    'MA',{0.5, -0.4, -0.3, 0.2, -0.1},...
    'MALags',[2 4 6 8 10],'D',1)
```

```
Mdl =

    Regression with ARIMA(4,1,10) Error Model:
    ----------------------------------------
    Distribution: Name = 'Gaussian'
       Intercept: 2
            Beta: [5 -1]
               P: 5
               D: 1
               Q: 10
              AR: {0.9 -0.8 0.75 -0.6} at Lags [1 2 3 4]
             SAR: {}
              MA: {0.5 -0.4 -0.3 0.2 -0.1} at Lags [2 4 6 8 10]
             SMA: {}
        Variance: NaN
```

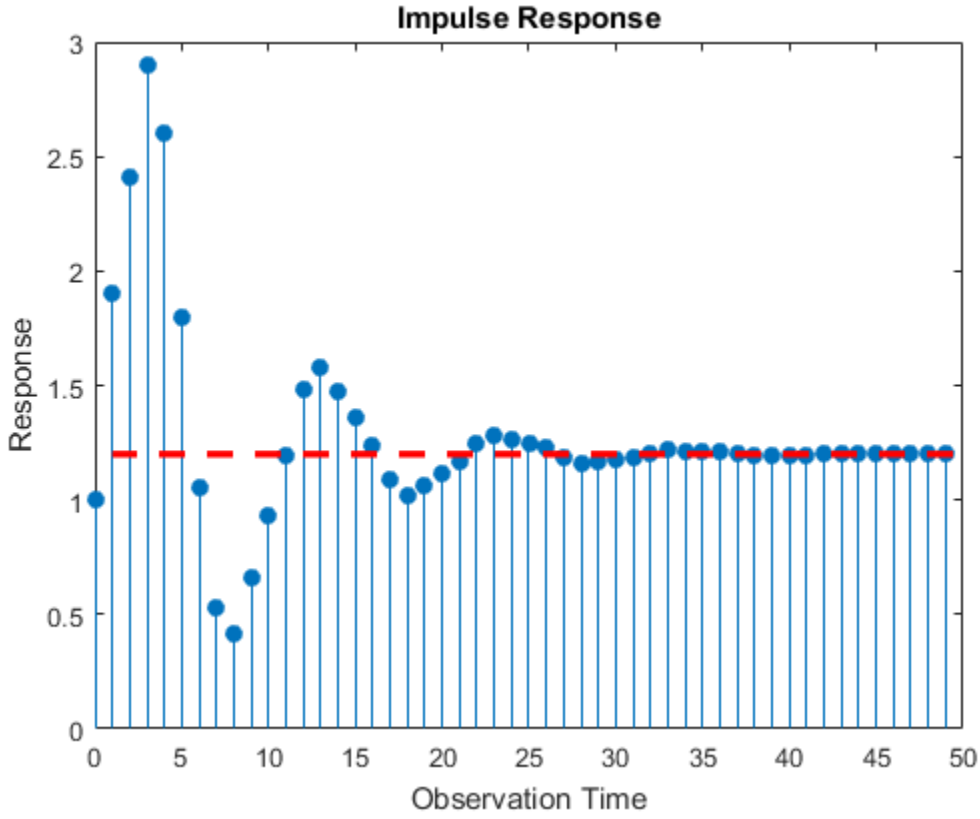
One of the roots of the compound autoregressive polynomial is 1, therefore `Mdl` defines a nonstationary error process.

You do not need to specify the innovation variance.

Plot the first impulse responses.

```
quot = sum([1,cell2mat(Mdl.MA)])/sum([1,-cell2mat(Mdl.AR)])

impulse(Mdl,50)
hold on
plot([1 50],[quot quot],'r--','Linewidth',2.5)
hold off
```

```
quot =

    1.2000
```



The impulse responses do not decay to 0. They settle at the quotient of the sums of the moving average and autoregressive polynomial coefficients (`quot`).

$$\texttt{quot} = \frac{1 + 0.5 - 0.4 - 0.3 + 0.2 - 0.1}{1 - 0.9 + 0.8 - 0.75 + 0.6} = 1.2.$$

See Also

impulse | regARIMA

Related Examples

- "Specify Regression Models with ARMA Errors" on page 4-42

- “Specify Regression Models with ARIMA Errors” on page 4-48

More About

- “Impulse Response for Regression Models with ARIMA Errors” on page 4-75

Maximum Likelihood Estimation of regARIMA Models

Innovation Distribution

For regression models with ARIMA time series errors in Econometrics Toolbox, $\varepsilon_t = \sigma z_t$, where:

- ε_t is the innovation corresponding to observation t.
- σ is the constant variance of the innovations. You can set its value using the `Variance` property of a `regARIMA` model.
- z_t is the innovation distribution. You can set the distribution using the `Distribution` property of a `regARIMA` model. Specify either a standard Gaussian (the default) or standardized Student's t with $\nu > 2$ or `NaN` degrees of freedom.

Note: If ε_t has a Student's t distribution, then

$$z_t = T_\nu \sqrt{\frac{\nu - 2}{\nu}},$$

where T_ν is a Student's t random variable with $\nu > 2$ degrees of freedom. Subsequently, z_t is t-distributed with mean 0 and variance 1, but has the same kurtosis as T_ν. Therefore, ε_t is t-distributed with mean 0, variance σ, and has the same kurtosis as T_ν.

`estimate` builds and optimizes the likelihood objective function based on ε_t by:

1. Estimating c and β using MLR
2. Inferring the unconditional disturbances from the estimated regression model, $\hat{u}_t = y_t - \hat{c} - X_t\hat{\beta}$
3. Estimating the ARIMA error model, $\hat{u}_t = \mathrm{H}^{-1}(L)\mathrm{N}(L)\varepsilon_t$, where $H(L)$ is the compound autoregressive polynomial and $N(L)$ is the compound moving average polynomial
4. Inferring the innovations from the ARIMA error model, $\hat{\varepsilon}_t = \hat{\mathrm{H}}^{-1}(L)\hat{\mathrm{N}}(L)\hat{u}_t$
5. Maximizing the loglikelihood objective function with respect to the free parameters

Note: If the unconditional disturbance process is nonstationary (i.e., the nonseasonal or seasonal integration degree is greater than 0), then the regression intercept, *c*, is not identifiable. `estimate` returns a `NaN` for *c* when it fits integrated models. For details, see "Intercept Identifiability in Regression Models with ARIMA Errors" on page 4-130.

`estimate` estimates all parameters in the `regARIMA` model set to `NaN`. `estimate` honors any equality constraints in the `regARIMA` model, i.e., `estimate` fixes the parameters at the values that you set during estimation.

Loglikelihood Functions

Given its history, the innovations are conditionally independent. Let H_t denote the history of the process available at time t, where $t = 1,...,T$. The likelihood function of the innovations is

$$f(\varepsilon_1,...,\varepsilon_T|H_{T-1}) = \prod_{t=1}^{T} f(\varepsilon_t|\mathrm{H}_{t-1}),$$

where f is the standard Gaussian or t probability density function.

The exact form of the loglikelihood objective function depends on the parametric form of the innovation distribution.

- If z_t is standard Gaussian, then the loglikelihood objective function is

$$logL = -\frac{T}{2}\log(2\pi) - \frac{T}{2}\log\sigma^2 - \frac{1}{2\sigma^2}\sum_{t=1}^{T}\varepsilon_t^2.$$

- If z_t is a standardized Student's t, then the loglikelihood objective function is

$$logL = T\log\left[\frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\Gamma\left(\frac{\nu}{2}\right)\sqrt{\pi(\nu-2)}}\right] - \frac{T}{2}\sigma^2 - \frac{\nu+1}{2}\sum_{t=1}^{T} log\left[1+\frac{\varepsilon_t^2}{\sigma^2(\nu-2)}\right].$$

`esitmate` performs covariance matrix estimation for maximum likelihood estimates using the outer product of gradients (OPG) method.

See Also

estimate | regARIMA

More About

- "regARIMA Model Estimation Using Equality Constraints" on page 4-89
- "Presample Values for regARIMA Model Estimation" on page 4-95
- "Initial Values for regARIMA Model Estimation" on page 4-98
- "Optimization Settings for regARIMA Model Estimation" on page 4-100

regARIMA Model Estimation Using Equality Constraints

`estimate` requires a `regARIMA` model and a vector of univariate response data to estimate a regression model with ARIMA errors. Without predictor data, the model specifies the parametric form of an intercept-only regression component with an ARIMA error model. This is not the same as a conditional mean model with a constant. For details, see "Compare Alternative ARIMA Model Representations" on page 4-136. If you specify a T-by-r matrix of predictor data, then `estimate` includes a linear regression component for the r series.

`estimate` returns fitted values for any parameters in the input model with `NaN` values. For example, if you specify a default `regARIMA` model and pass a T-by-r matrix of predictor data, then the software sets all parameters to `NaN` including the `r` regression coefficients, and estimates them all. If you specify non-`NaN` values for any parameters, then `estimate` views these values as equality constraints and honors them during estimation.

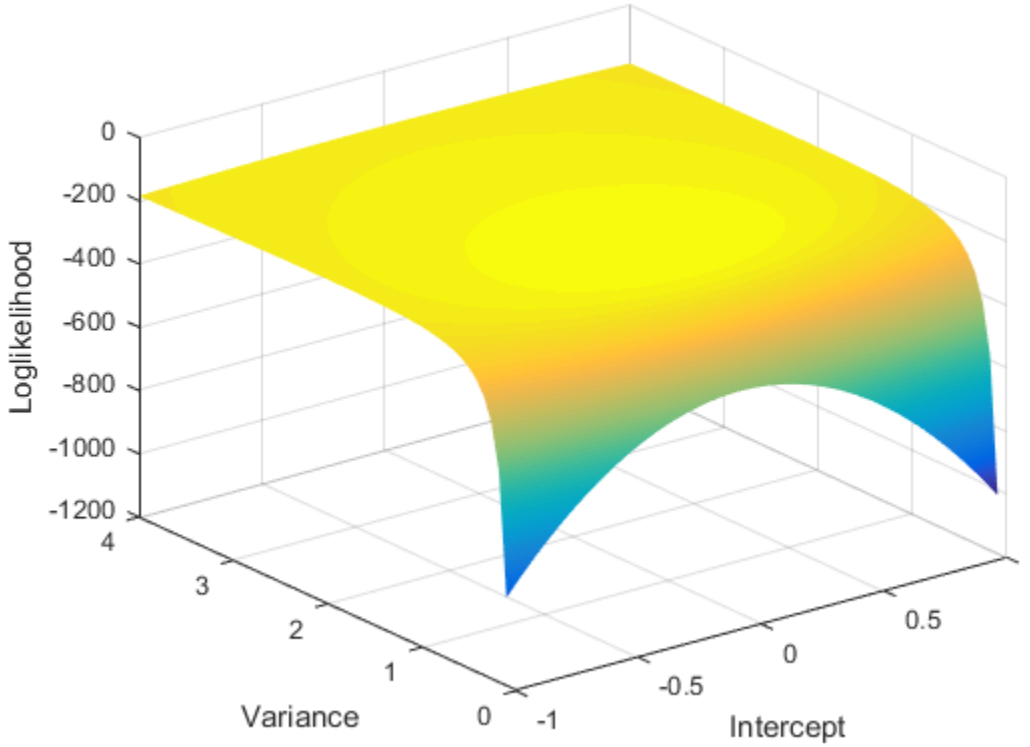
For example, suppose residual diagnostics from a linear regression suggest integrated unconditional disturbances. Since the regression intercept is unidentifiable in integrated models, you decide to set the intercept to 0. Specify `'Intercept',0` in the `regARIMA` model that you pass into `estimate`. The software views this non-`NaN` value as an equality constraint, and does not estimate the intercept, its standard error, and its covariance with other estimates. To illustrate further, suppose the true model for a response series y_t is

$$\begin{aligned} y_t &= 0 + u_t \\ u_t &= \varepsilon_t, \end{aligned}$$

where ε_t is Gaussian with variance 1. The loglikelihood function for a simulated data set from this model can resemble the surface in the following figure over a grid of variances and intercepts.

```
rng(1); % For reproducibility
e = randn(100,1);
Variance = 1;
Intercept = 0;
Mdl = regARIMA('Intercept',Intercept,'Variance',Variance);
y = filter(Mdl,e);

gridLength = 50;
intGrid1 = linspace(-1,1,50);
```

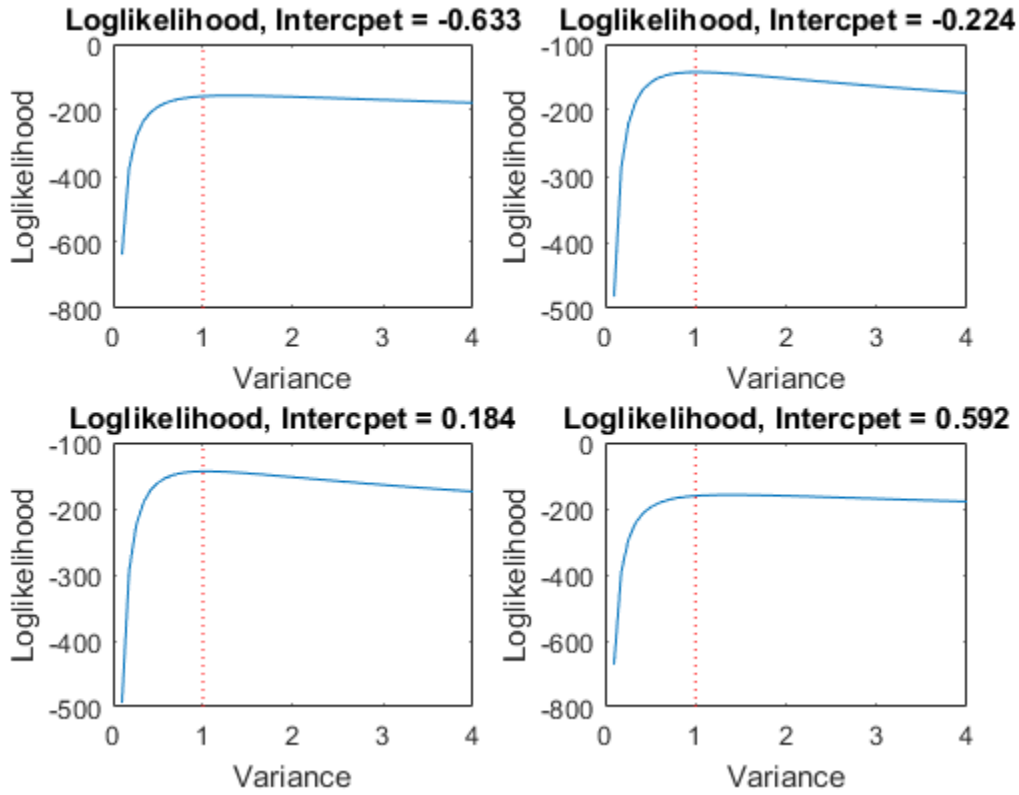
```
varGrid1 = linspace(0.1,4,50);
[varGrid2,intGrid2] = meshgrid(varGrid1,intGrid1);
LogLGrid = zeros(numel(varGrid1),numel(intGrid1));

for k1 = 1:numel(intGrid1)
    for k2 = 1:numel(varGrid1)
        ToEstMdl = regARIMA('Intercept',...
            intGrid1(k1),'Variance',varGrid1(k2));
        [~,~,LogLGrid(k1,k2)] = estimate(ToEstMdl,y);
    end
end

surf(intGrid2,varGrid2,LogLGrid) % 3D loglikelihood plot
xlabel 'Intercept';
ylabel 'Variance';
zlabel 'Loglikelihood';
shading interp
```



Notice that the maximum (darkest red region) occurs around `Intercept = 0` and `Variance = 1`. If you apply an equality constraint, then the optimizer views a two-dimensional slice (in this example) of the loglikelihood function at that constraint. The following plots display the loglikelihood at several different `Intercept` equality constraints.

```
intValue = [intGrid1(10), intGrid1(20),...
    intGrid1(30), intGrid1(40)];
for k = 1:4
    subplot(2,2,k)
%    plot(varGrid1,LogLGrid(find(intGrid2 == intValue(k))))
    plot(varGrid1,LogLGrid(intGrid2 == intValue(k)))
    title(sprintf('Loglikelihood, Intercpet = %.3f',intValue(k)))
    xlabel 'Variance';
```

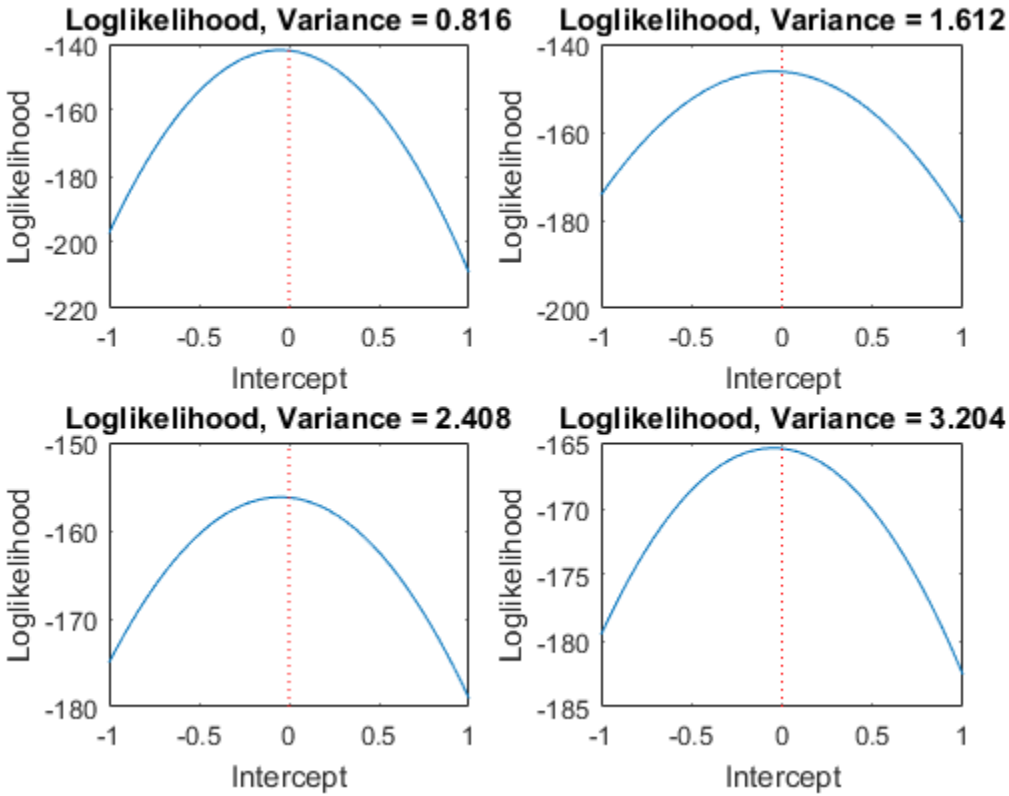
```
    ylabel 'Loglikelihood';
    hold on
    h1 = gca;
    plot([Variance Variance],h1.YLim,'r:')
    hold off
end
```



In each case, `Variance = 1` (its true value) occurs very close to the maximum of the loglikelihood function. Rather than constrain `Intercept`, the following plots display the likelihood function using several `Variance` equality constraints.

```
varValue = [varGrid1(10),varGrid1(20),varGrid1(30),varGrid1(40)];
for k = 1:4
```

```
    subplot(2,2,k)
%    plot(intGrid1,LogLGrid(find(varGrid2 == varValue(k))))
    plot(intGrid1,LogLGrid(varGrid2 == varValue(k)))
    title(sprintf('Loglikelihood, Variance = %.3f',varValue(k)))
    xlabel('Intercept')
    ylabel('Loglikelihood')
    hold on
    h2 = gca;
    plot([Intercept Intercept],h2.YLim,'r:')
    hold off
end
```



In each case, `Intercept` = 0 (its true value) occurs very close to the maximum of the loglikelihood function.

`estimate` also honors a subset of equality constraints while estimating all other parameters set to `NaN`. For example, suppose $r = 3$, and you know that $\beta_2 = 5$. Specify `Beta = [NaN; 5; NaN]` in the `regARIMA` model, and pass this model with the data to `estimate`.

`estimate` optionally returns the estimated variance-covariance matrix for the estimated parameters. The parameter order in this matrix is:

- Intercept
- Nonzero `AR` coefficients at positive lags
- Nonzero `SAR` coefficients at positive lags
- Nonzero `MA` coefficients at positive lags
- Nonzero `SMA` coefficients at positive lags
- Regression coefficients (when you specify `X` in `estimate`)
- Innovation variance
- Degrees of freedom for the t distribution

If any parameter known to the optimizer has an equality constraint, then the corresponding row and column of the variance-covariance matrix has all 0s.

In addition to your equality constraints, `estimate` sets any AR, MA, SAR, and SMA coefficient with an estimate less than 1e-12 in magnitude equal to 0.

See Also

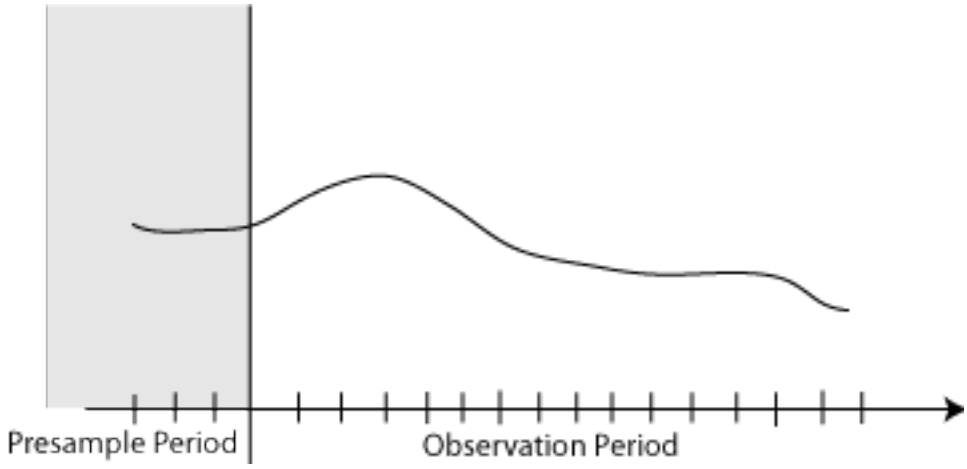
estimate | regARIMA

More About

- “Maximum Likelihood Estimation of regARIMA Models” on page 4-86
- “Presample Values for regARIMA Model Estimation” on page 4-95
- “Initial Values for regARIMA Model Estimation” on page 4-98
- “Optimization Settings for regARIMA Model Estimation” on page 4-100

Presample Values for regARIMA Model Estimation

Presample data comes from time points before the beginning of the observation period. In Econometrics Toolbox, you can specify your own presample data or use generated presample data.



In regression models with ARIMA errors, the distribution of the current innovation (ε_t) is conditional on *historic information* (H_t). Historic information can include past unconditional disturbances or past innovations, i.e., $H_t = \{u_{t-1},\varepsilon_{t-1},u_{t-2},\varepsilon_{t-2},\ldots,u_0,\varepsilon_0,u_{-1},\varepsilon_{-1},\ldots\}$. However, the software does not include past responses (y_t) nor past predictors (X_t) in H_t. For example, in a regression model with ARIMA(2,1,1) errors, you can write the error model in several ways:

- $\left(1-\phi_1 L-\phi_2 L^2\right)(1-L)u_t = (1+\theta_1 L)\varepsilon_t.$
- $\left(1-L-\phi_1\left(L-L^2\right)-\phi_2\left(L^2-L^3\right)\right)u_t = (1+\theta_1 L)\varepsilon_t.$
- $u_t = u_{t-1}+\phi_1(u_{t-1}-u_{t-2})+\phi_2(u_{t-2}-u_{t-3})+\varepsilon_t+\theta_1\varepsilon_{t-1}.$
- $\varepsilon_t = u_t-u_{t-1}-\phi_1(u_{t-1}-u_{t-2})-\phi_2(u_{t-2}-u_{t-3})-\theta_1\varepsilon_{t-1}.$

The last equation implies that:

- The first innovation in the series (ε_1) depends on the history $H_1 = \{u_{-2},u_{-1},u_0,\varepsilon_0\}$. H_1 is not observable nor inferable from the regression model.
- The second innovation in the series (ε_2) depends on the history $H_2 = \{u_{-1},u_0,u_1,\varepsilon_1\}$. The software can infer u_1 and ε_1, but not the others.
- The third innovation in the series (ε_3) depends on the history $H_3 = \{u_0,u_1,u_2,\varepsilon_2\}$. The software can infer u_1, u_2, and ε_1, but not u_0.
- The rest of the innovations depend on inferable unconditional disturbances and innovations.

Therefore, the software requires three presample unconditional disturbances to initialize the autoregressive portion, and one presample innovation to initialize the moving average portion.

The degrees of the compound autoregressive and moving average polynomials determine the number of past unconditional disturbances and innovations that ε_t depends on. The compound autoregressive polynomial includes the seasonal and nonseasonal autoregressive, and seasonal and nonseasonal integration polynomials. The compound moving average polynomial includes the seasonal and nonseasonal moving average polynomials. In the example, the degree of the compound autoregressive polynomial is `P` = 3, and the degree of the moving average polynomial is `Q` = 1. Therefore, the software requires three presample unconditional disturbances and one presample innovation.

If you do not have presample values (or do not supply them), then, by default, the software backcasts for the necessary presample unconditional disturbances, and sets the necessary presample innovations to 0.

Another option to obtain presample unconditional disturbances is to partition the data set into a presample portion and estimation portion:

1. Partition the data such that the presample portion contains at least `max(P,Q)` observations. The software uses the most recent `max(P,Q)` observations and ignores the rest.
2. For the presample portion, regress y_t onto X_t.
3. Infer the residuals from the regression model. These are the presample unconditional disturbances.
4. Pass the presample unconditional disturbances (`U0`) and the estimation portion of the data into `estimate`.

This option results in a loss of sample size. Note that when comparing multiple models using likelihood-based measures of fit (such as likelihood ratio tests or information

criteria), then the data must have the same estimation portions, and the presample portions must be of equal size.

If you plan on specifying presample values, then you must specify at least the number necessary to initialize the series.

You can specify both presample unconditional disturbances and innovations, one or the other, or neither.

More About

- "Maximum Likelihood Estimation of regARIMA Models" on page 4-86
- "regARIMA Model Estimation Using Equality Constraints" on page 4-89
- "Initial Values for regARIMA Model Estimation" on page 4-98
- "Optimization Settings for regARIMA Model Estimation" on page 4-100

Initial Values for regARIMA Model Estimation

`estimate` uses `fmincon` from Optimization Toolbox™ to minimize the negative loglikelihood objective function. `fmincon` requires initial (i.e., starting) values to begin the optimization process.

If you want to specify your own initial values, then use name-value pair arguments. For example, to specify 0.1 for the initial value of a nonseasonal AR coefficient of the error model, pass the name-value pair argument `'AR0',0.1` into `estimate`.

By default, `estimate` generates initial values using standard time series techniques. If you partially specify initial values (that is, specify initial values for some parameters), `estimate` honors the initial values that you set, and generates default initial values for the remaining parameters.

`estimate` enforces stability and invertibility for all seasonal and nonseasonal AR and MA lag operator polynomials of the error model. When you specify initial values for the AR and MA coefficients, it is possible that `estimate` cannot find initial values for the remaining coefficients that satisfy stability and invertibility. In this case, `estimate` honors your initial values, and sets the remaining initial coefficient values to 0.

The way `estimate` generates default initial values depends on the model.

- If the model contains a regression component and intercept, then `estimate` performs ordinary least squares (OLS). `estimate` uses the estimates for `Beta0` and `Intercept0`. Then, `estimate` infers the unconditional disturbances using the regression model. `estimate` uses the inferred unconditional disturbances and the ARIMA error model to gather the other initial values.

- If the model does not contain a regression component and an intercept, then the unconditional disturbance series is the response series. `estimate` uses the unconditional disturbances and the ARIMA error model to gather the other initial values.

This table summarizes the techniques that `estimate` uses to gather the remaining initial values.

	Technique to Generate Initial Values	
Parameter	**Error Model Does Not Contain MA Terms**	**Error Model Contains MA Terms**
AR	OLS	Solve the Yule-Walker equations [1].
MA	N/A	Solve the Yule-Walker equations [1].
Variance	Population variance of OLS residuals	Variance of inferred innovation process (using initial MA coefficients)

References

[1] Box, G. E. P., G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis: Forecasting and Control*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

See Also

estimate | `fmincon`

More About

- “Maximum Likelihood Estimation of regARIMA Models” on page 4-86
- “regARIMA Model Estimation Using Equality Constraints” on page 4-89
- “Presample Values for regARIMA Model Estimation” on page 4-95
- “Optimization Settings for regARIMA Model Estimation” on page 4-100

Optimization Settings for regARIMA Model Estimation

In this section...

Optimization Options

`estimate` maximizes the loglikelihood function using `fmincon` from Optimization Toolbox. `fmincon` has many optimization options, such as choice of optimization algorithm and constraint violation tolerance. Choose optimization options using `optimoptions`.

`estimate` uses the `fmincon` optimization options by default, with these exceptions. For details, see `fmincon` and `optimoptions` in Optimization Toolbox.

optimoptions Properties	Description	estimate Settings
`Algorithm`	Algorithm for minimizing the negative loglikelihood function	`'sqp'`
`Display`	Level of display for optimization progress	`'off'`
`Diagnostics`	Display for diagnostic information about the function to be minimized	`'off'`
`TolCon`	Termination tolerance on constraint violations	`1e-7`

If you want to use optimization options that differ from the default, then set your own using `optimoptions`.

For example, suppose that you want `estimate` to display optimization diagnostics. The best practice is to set the name-value pair argument `'Display','diagnostics'` in `estimate`. Alternatively, you can direct the optimizer to display optimization diagnostics.

Specify a regression model with AR(1) errors (`Mdl`) and simulate data from it.

```
Mdl = regARIMA('AR',0.5,'Intercept',0,'Variance',1);
rng(1); % For reproducibility
y = simulate(Mdl,25);
```

`Mdl` does not have a regression component. By default, `fmincon` does not display the optimization diagnostics. Use `optimoptions` to set it to display the optimization diagnostics, and set the other `fmincon` properties to the default settings of `estimate` listed in the previous table.

```
options = optimoptions(@fmincon,'Diagnostics','on','Algorithm',...
    'sqp','Display','off','TolCon',1e-7)
% @fmincon is the function handle for fmincon
```

```
options =

  fmincon options:

   Options used by current Algorithm ('sqp'):
   (Other available algorithms: 'active-set', 'interior-point', 'trust-region-reflectiv

   Set by user:
          Algorithm: 'sqp'
        Diagnostics: 'on'
            Display: 'off'
             TolCon: 1.0000e-07

   Default:
    DerivativeCheck: 'off'
      DiffMaxChange: Inf
      DiffMinChange: 0
     FinDiffRelStep: 'sqrt(eps)'
        FinDiffType: 'forward'
        FunValCheck: 'off'
         GradConstr: 'off'
            GradObj: 'off'
        MaxFunEvals: '100*numberOfVariables'
            MaxIter: 400
     ObjectiveLimit: -1.0000e+20
          OutputFcn: []
           PlotFcns: []
       ScaleProblem: 'none'
             TolFun: 1.0000e-06
               TolX: 1.0000e-06
           TypicalX: 'ones(numberOfVariables,1)'
```

```
     UseParallel: 0

Options not used by current Algorithm ('sqp')
Default:
 AlwaysHonorConstraints: 'bounds'
               HessFcn: []
              HessMult: []
           HessPattern: 'sparse(ones(numberOfVariables))'
               Hessian: 'not applicable'
      InitBarrierParam: 0.1000
  InitTrustRegionRadius: 'sqrt(numberOfVariables)'
            MaxPCGIter: 'max(1,floor(numberOfVariables/2))'
         MaxProjCGIter: '2*(numberOfVariables-numberOfEqualities)'
            MaxSQPIter: '10*max(numberOfVariables,numberOfInequalities+…'
      PrecondBandWidth: 0
        RelLineSrchBnd: []
 RelLineSrchBndDuration: 1
    SubproblemAlgorithm: 'ldl-factorization'
              TolConSQP: 1.0000e-06
                 TolPCG: 0.1000
              TolProjCG: 0.0100
           TolProjCGAbs: 1.0000e-10
```

The options that you set appear under the `Set by user:` heading. The properties under the `Default:` heading are other options that you can set.

Fit `Mdl` to `y` using the new optimization options.

```
ToEstMdl = regARIMA(1,0,0);
EstMdl = estimate(ToEstMdl,y,'Options',options);
```

```
_____________________________________________________________
   Diagnostic Information

Number of variables: 3

Functions
Objective:                              @(X)nLogLike(X,YData,XData,E,U,Mdl,AR.Lags,MA.Lag
Gradient:                               finite-differencing
Hessian:                                finite-differencing (or Quasi-Newton)
Nonlinear constraints:                  @(x)internal.econ.arimaNonLinearConstraints(x,Lag
Nonlinear constraints gradient:         finite-differencing
```

```
Constraints
Number of nonlinear inequality constraints: 1
Number of nonlinear equality constraints:   0

Number of linear inequality constraints:    0
Number of linear equality constraints:      0
Number of lower bound constraints:          3
Number of upper bound constraints:          3

Algorithm selected
   sqp


____________________________________________________________
   End diagnostic information

    ARIMA(1,0,0) Error Model:
    --------------------------
    Conditional Probability Distribution: Gaussian

                                Standard          t
     Parameter       Value        Error       Statistic
    -----------   -----------   ------------   -----------
    Intercept       -0.12097      0.447475      -0.270338
        AR{1}       0.463859      0.157813         2.9393
     Variance        1.23081      0.472745        2.60354
```

Note:

- `estimate` numerically maximizes the loglikelihood function potentially using equality, inequality, and lower and upper bound constraints. If you set `Algorithm` to anything other than `sqp`, then check that the algorithm supports similar constraints, such as `interior-point`. For example, `fmincon` sets `Algorithm` to `trust-region-reflective` by default. `trust-region-reflective` does not support inequality constraints. Therefore, if you do not change the default `Algorithm` property value of `fmincon`, then `estimate` displays a warning. During estimation, `fmincon` temporarily sets `Algorithm` to `active-set` by default to satisfy the constraints.
- `estimate` sets a constraint level of `TolCon` so constraints are not violated. Be aware that an estimate with an active constraint has unreliable standard errors since

variance-covariance estimation assumes the likelihood function is locally quadratic around the maximum likelihood estimate.

Constraints on Regression Models with ARIMA Errors

The software enforces these constraints while estimating a regression model with ARIMA errors:

- Stability of nonseasonal and seasonal AR operator polynomials
- Invertibility of nonseasonal and seasonal MA operator polynomials
- Innovation variance strictly greater than zero
- Degrees of freedom strictly greater than two for a *t* innovation distribution

See Also

estimate | `fmincon` | `optimoptions` | regARIMA

More About

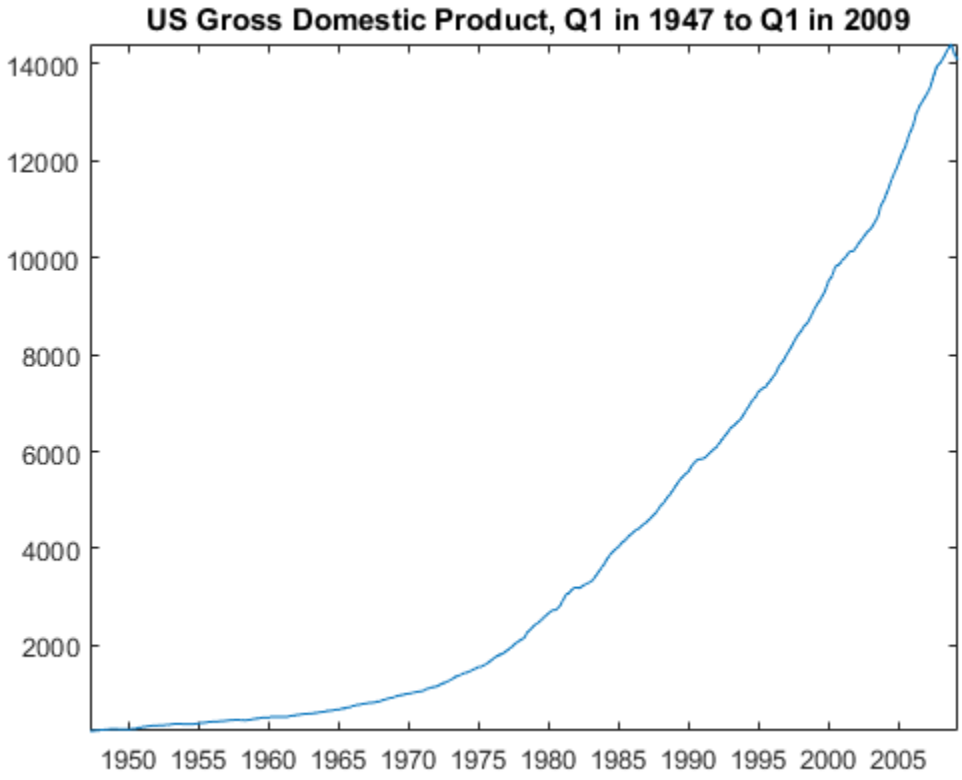
- "Maximum Likelihood Estimation of regARIMA Models" on page 4-86
- "regARIMA Model Estimation Using Equality Constraints" on page 4-89
- "Presample Values for regARIMA Model Estimation" on page 4-95
- "Initial Values for regARIMA Model Estimation" on page 4-98

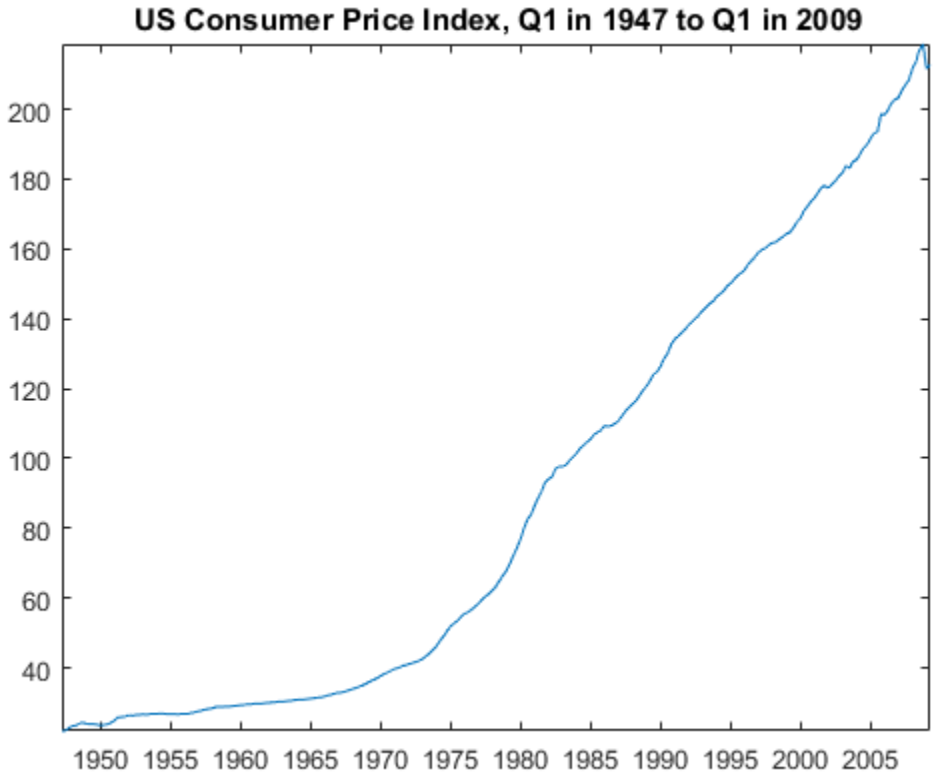
Estimate a Regression Model with ARIMA Errors

This example shows how to estimate the sensitivity of the US Gross Domestic Product (GDP) to changes in the Consumer Price Index (CPI) using `estimate`.

Load the US macroeconomic data set, `Data_USEconModel`. Plot the GDP and CPI.

```
load Data_USEconModel
gdp = DataTable.GDP;
cpi = DataTable.CPIAUCSL;

figure
plot(dates,gdp)
title('{\bf US Gross Domestic Product, Q1 in 1947 to Q1 in 2009}')
datetick
axis tight

figure
plot(dates,cpi)
title('{\bf US Consumer Price Index, Q1 in 1947 to Q1 in 2009}')
datetick
axis tight
```


US Gross Domestic Product, Q1 in 1947 to Q1 in 2009
14000
12000
10000
8000
6000
4000
2000
1950
1955
1960
1965
1970
1975
1980
1985
1990
1995
2000
2005

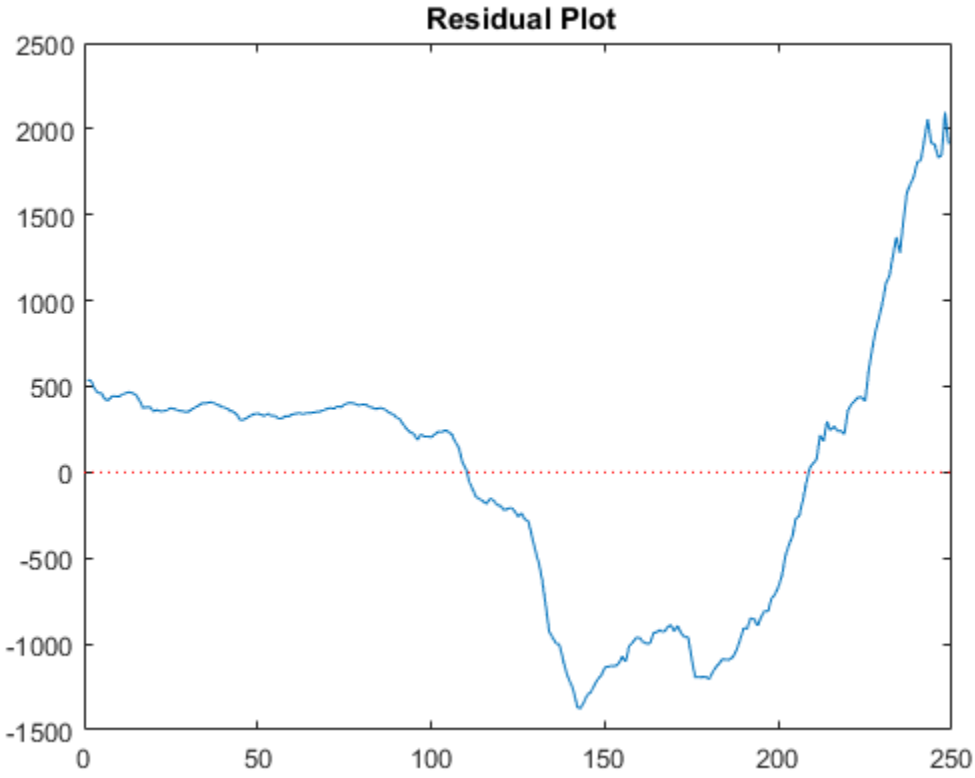


`gdp` and `cpi` seem to increase exponentially.

Regress `gdp` onto `cpi`. Plot the residuals.

```
XDes = [ones(length(cpi),1) cpi]; % Design matrix
beta = XDes\gdp;
u = gdp - XDes*beta; % Residuals

figure
plot(u)
h1 = gca;
hold on
plot(h1.XLim,[0 0],'r:')
title('{\bf Residual Plot}')
```

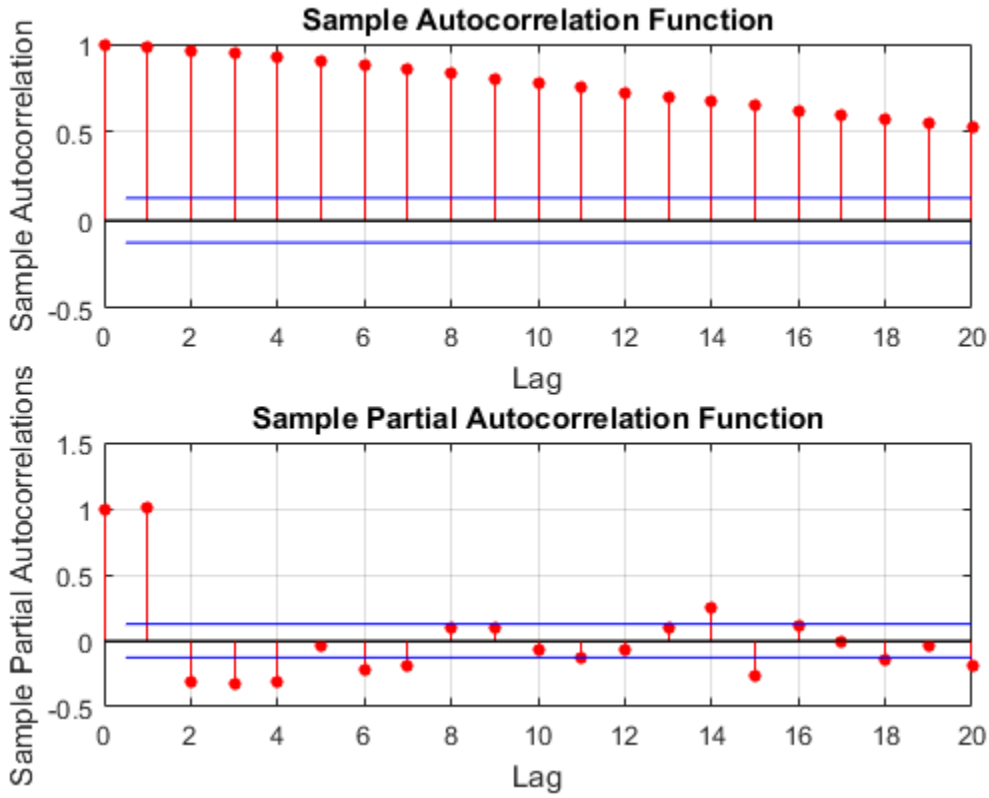
```
hold off
```



The pattern of the residuals suggests that the standard linear model assumption of uncorrelated errors is violated. The residuals appear autocorrelated.

Plot correlograms for the residuals.

```
figure
subplot(2,1,1)
autocorr(u)
subplot(2,1,2)
parcorr(u)
```

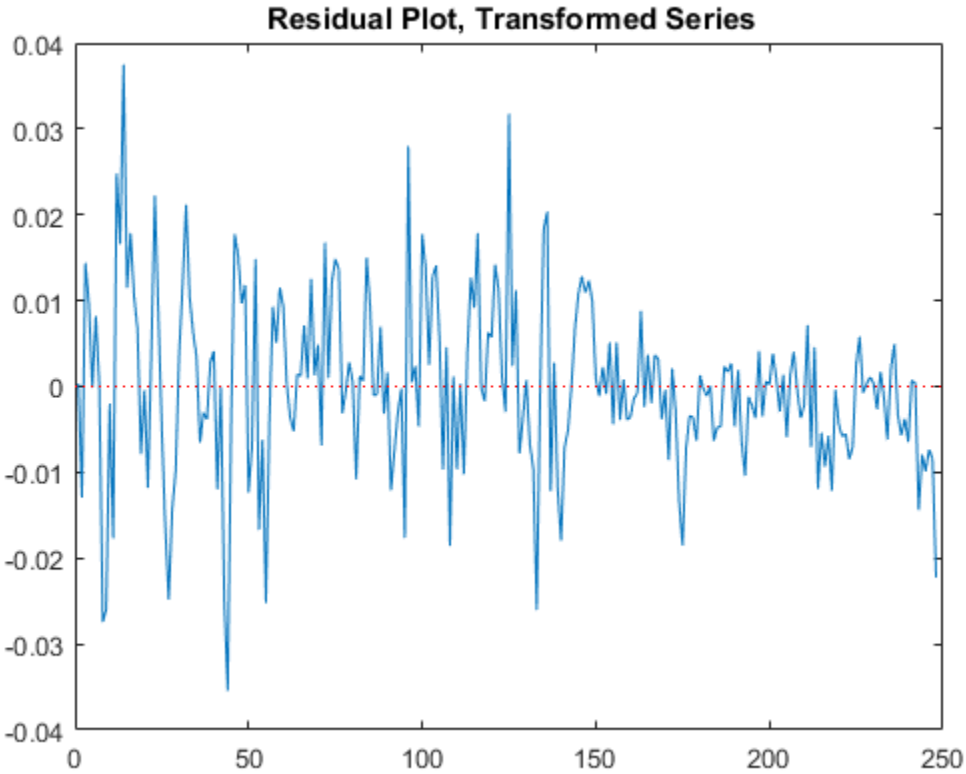


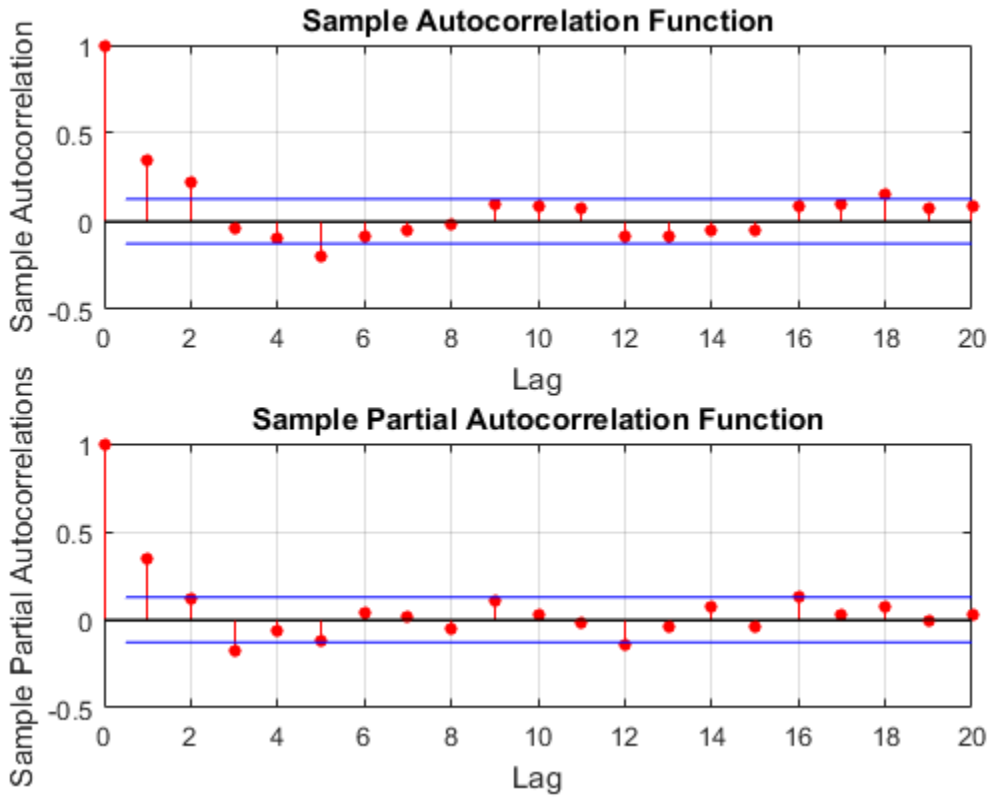
The autocorrelation function suggests that the residuals are a nonstationary process.

Apply the first difference to the logged series to stabilize the residuals.

```
dlGDP = diff(log(gdp));
dlCPI = diff(log(cpi));
dlXDes = [ones(length(dlCPI),1) dlCPI];
beta = dlXDes\dlGDP;
u = dlGDP - dlXDes*beta;

figure
plot(u);
h2 = gca;
hold on
```

```
plot(h2.XLim,[0 0],'r:')
title('{\bf Residual Plot, Transformed Series}')
hold off

figure
subplot(2,1,1)
autocorr(u)
subplot(2,1,2)
parcorr(u)
```





The residual plot from the transformed data suggests stabilized, albeit heteroscedastic, unconditional disturbances. The correlograms suggest that the unconditional disturbances follow an AR(1) process.

Specify the regression model with AR(1) errors:

$$\begin{aligned} \texttt{dlGDP} &= \texttt{Intercept} + \texttt{dlCPI}\beta + u_t \\ u_t &= \phi u_{t-1} + \varepsilon_t. \end{aligned}$$

```
Mdl = regARIMA('ARLags',1);
```

`estimate` estimates any parameter having a value of `NaN`.

Fit `Mdl` to the data.

```
EstMdl = estimate(Mdl,dlGDP,'X',dlCPI,'Display','params');
```

```
    Regression with ARIMA(1,0,0) Error Model:
    ----------------------------------------
    Conditional Probability Distribution: Gaussian

                                Standard          t
     Parameter       Value        Error       Statistic
    -----------   -----------   ------------   -----------
    Intercept      0.0127623    0.00134717        9.47336
        AR{1}       0.382447     0.0524938        7.28557
        Beta1       0.398902     0.0772861        5.16137
     Variance    9.01012e-05   5.94704e-06        15.1506
```

Alternatively, estimate the regression coefficients and Newey-West standard errors using `hac`.

```
hac(dlCPI,dlGDP,'intercept',true,'display','full');
```

```
Estimator type: HAC
Estimation method: BT
Bandwidth: 4.1963
Whitening order: 0
Effective sample size: 248
Small sample correction: on

Coefficient Estimates:

       |  Coeff    SE
-----------------------
 Const | 0.0115  0.0012
 x1    | 0.5421  0.1005

Coefficient Covariances:

       |  Const      x1
--------------------------
 Const |  0.0000  -0.0001
 x1    | -0.0001   0.0101
```

The intercept estimates are close, but the regression coefficient estimates corresponding to `dlCPI` are not. This is because `regARIMA` explicitly models for the autocorrelation of the disturbances. `hac` estimates the coefficients using ordinary least squares,

and returns standard errors that are robust to the residual autocorrelation and heteroscedasticity.

Assuming that the model is correct, the results suggest that an increase of one point in the CPI rate increases the GDP growth rate by 0.399 points. This effect is significant according to the t statistic.

From here, you can use `forecast` or `simulate` to obtain forecasts and forecast intervals for the GDP rate. You can also compare several models by computing their AIC statistics using `aicbic`.

See Also

`aicbic` | estimate | forecast | simulate

More About

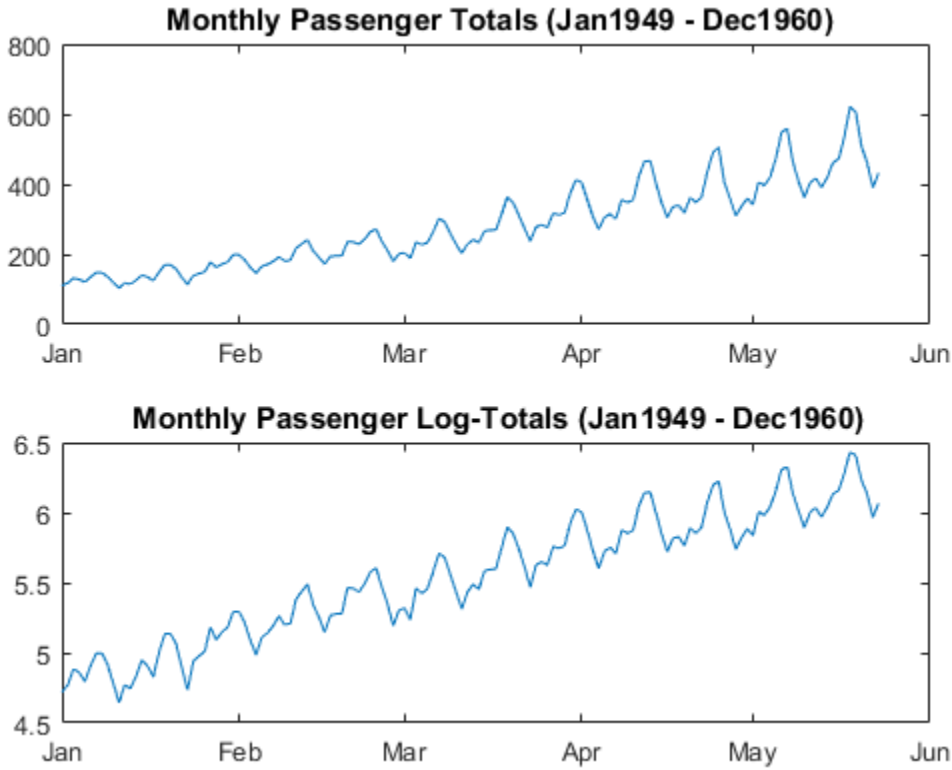
- "Maximum Likelihood Estimation of regARIMA Models" on page 4-86
- "Presample Values for regARIMA Model Estimation" on page 4-95
- "Initial Values for regARIMA Model Estimation" on page 4-98

Estimate a Regression Model with Multiplicative ARIMA Errors

This example shows how to fit a regression model with multiplicative ARIMA errors to data using `estimate`.

Load the Airline data set from the MATLAB® root folder, and load the Recession data set. Plot the monthly passenger totals and the log of the totals.

```
load(fullfile(matlabroot,'examples','econ','Data_Airline.mat'))
load Data_Recessions

y = Data;
logY = log(y);

figure
subplot(2,1,1)
plot(y)
title('{\bf Monthly Passenger Totals (Jan1949 - Dec1960)}')
datetick
subplot(2,1,2)
plot(log(y))
title('{\bf Monthly Passenger Log-Totals (Jan1949 - Dec1960)}')
datetick
```



The log transformation seems to linearize the time series.

Construct the predictor (X), which is whether the country was in a recession during the sampled period. A 0 in row t means the country was not in a recession in month t, and a 1 in row t means that it was in a recession in month t.

```
X = zeros(numel(dates),1); % Preallocation
for j = 1:size(Recessions,1)
    X(dates >= Recessions(j,1) & dates <= Recessions(j,2)) = 1;
end
```

Fit the simple linear regression model

$$y_t = c + X_t\beta + u_t$$

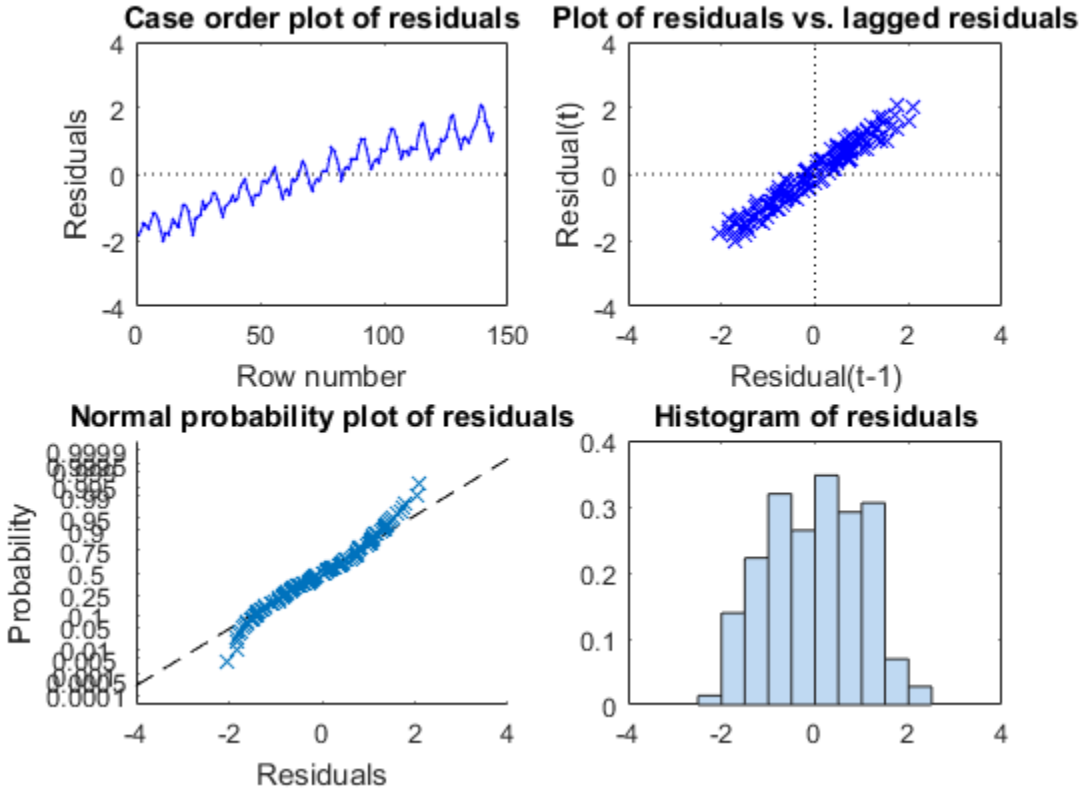
to the data.

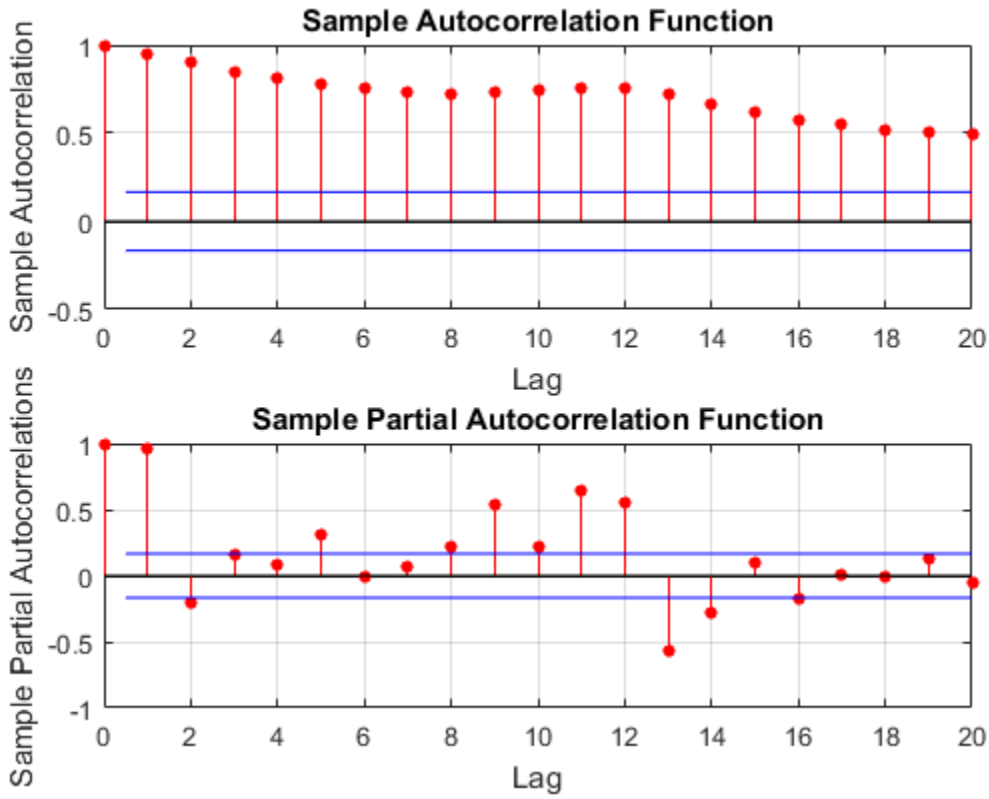
```
Fit = fitlm(X,logY);
```

`Fit` is a `LinearModel` that contains the least squares estimates.

Check for standard linear model assumption departures by plotting the residuals several ways.

```
figure
subplot(2,2,1)
plotResiduals(Fit,'caseorder','ResidualType','Standardized',...
    'LineStyle','-','MarkerSize',0.5)
subplot(2,2,2)
plotResiduals(Fit,'lagged','ResidualType','Standardized')
subplot(2,2,3)
plotResiduals(Fit,'probability','ResidualType','Standardized')
subplot(2,2,4)
plotResiduals(Fit,'histogram','ResidualType','Standardized')

r = Fit.Residuals.Standardized;
figure
subplot(2,1,1)
autocorr(r)
subplot(2,1,2)
parcorr(r)
```


Case order plot of residuals
Residuals
Row number
Plot of residuals vs. lagged residuals
Residual(t)
Residual(t-1)
Normal probability plot of residuals
Probability
Residuals
Histogram of residuals



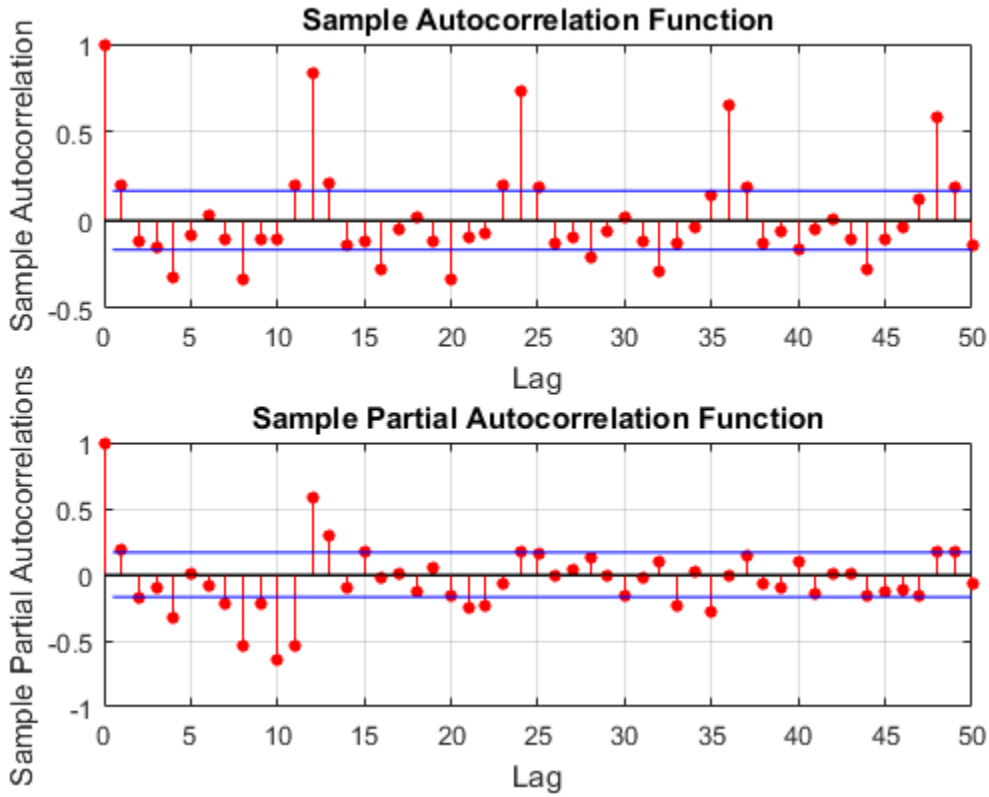
The residual plots indicate that the unconditional disturbances are autocorrelated. The probability plot and histogram seem to indicate that the unconditional disturbances are Gaussian.

The ACF of the residuals confirms that the unconditional disturbances are autocorrelated.

Take the 1st difference of the residuals and plot the ACF and PACF of the differenced residuals.

```
dR = diff(r);

figure
```

```
subplot(2,1,1)
autocorr(dR,50)
subplot(2,1,2)
parcorr(dR,50)
```

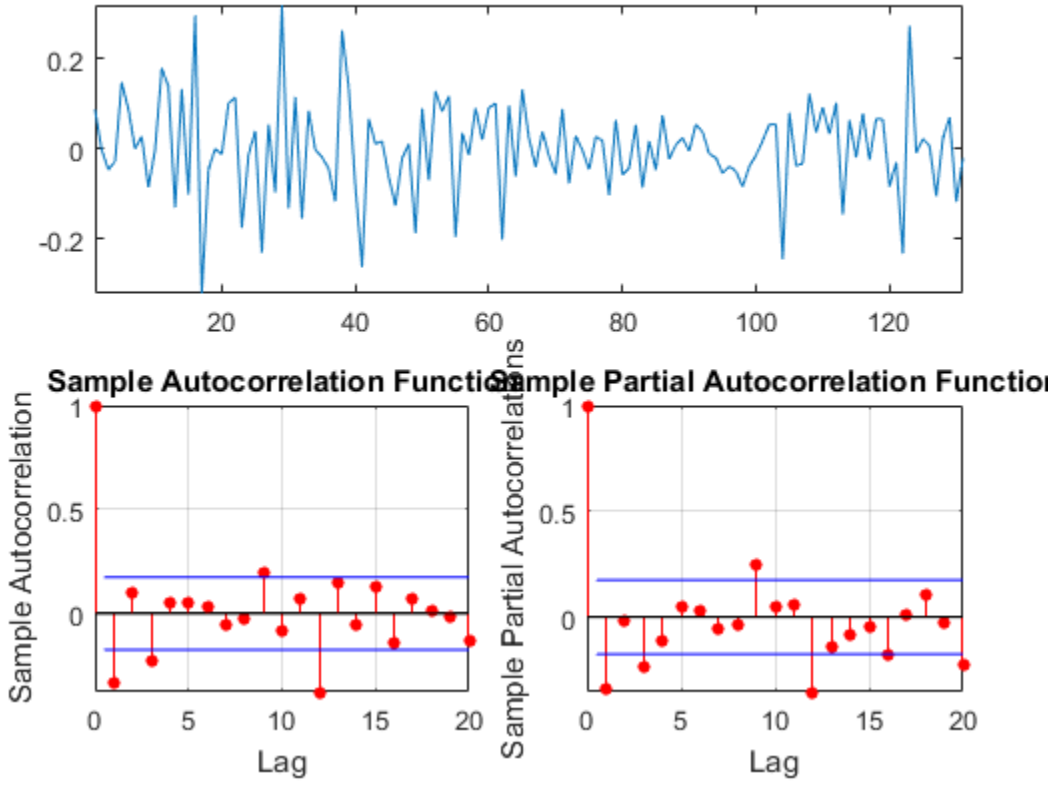


The ACF shows that there are significantly large autocorrelations, particularly at every 12th lag. This indicates that the unconditional disturbances have 12th degree seasonal integration.

Take the first and 12th differences of the residuals. Plot the differenced residuals, and their ACF and PACF.

```
DiffPoly = LagOp([1 -1]);
```

```
SDiffPoly = LagOp([1 -1],'Lags',[0, 12]);
diffR = filter(DiffPoly*SDiffPoly,r);

figure
subplot(2,1,1)
plot(diffR)
axis tight
subplot(2,2,3)
autocorr(diffR)
axis tight
subplot(2,2,4)
parcorr(diffR)
axis tight
```



The residuals resemble white noise (with possible heteroscedasticity). According to Box and Jenkins (1994), Chapter 9, the ACF and PACF indicate that the unconditional disturbances are an $\mathrm{IMA}(0,1,1)\times(0,1,1)_{12}$ model.

Specify the regression model with $\mathrm{IMA}(0,1,1)\times(0,1,1)_{12}$ errors:

$$\begin{aligned} y_t &= X_t\beta + u_t \\ (1-L)\left(1-L^{12}\right)u_t &= (1+b_1L)\left(1+B_{12}L^{12}\right)\varepsilon_t. \end{aligned}$$

```
Mdl = regARIMA('MALags',1,'D',1,'Seasonality',12,'SMALags',12)
```

```
Mdl =

    ARIMA(0,1,1) Error Model Seasonally Integrated with Seasonal MA(12):
    --------------------------------------------------------------------
    Distribution: Name = 'Gaussian'
       Intercept: NaN
               P: 13
               D: 1
               Q: 13
              AR: {}
             SAR: {}
              MA: {NaN} at Lags [1]
             SMA: {NaN} at Lags [12]
     Seasonality: 12
        Variance: NaN
```

Partition the data set into the presample and estimation sample so that you can initialize the series. P = Q = 13, so the presample should be at least 13 periods long.

```
preLogY = logY(1:13);   % Presample responses
estLogY = logY(14:end); % Estimation sample responses
preX = X(1:13);         % Presample predictors
estX = X(14:end);       % Estimation sample predictors
```

Obtain presample unconditional disturbances from a linear regression of the presample data.

```
PreFit = fitlm(preX,preLogY);...
    % Presample fit for presample residuals
EstFit = fitlm(estX,estLogY);...
    % Estimation sample fit for the intercept
```

```
U0 = PreFit.Residuals.Raw;
```

If the error model is integrated, then the regression model intercept is not identifiable. Set `Intercept` to the estimated intercept from a linear regression of the estimation sample data. Estimate the regression model with IMA errors.

```
Mdl.Intercept = EstFit.Coefficients{1,1};
EstMdl = estimate(Mdl,estLogY,'X',estX,'U0',U0);
```

```
    Regression with ARIMA(0,1,1) Error Model Seasonally Integrated with Seasonal MA(12)
    ------------------------------------------------------------------------------------
    Conditional Probability Distribution: Gaussian

                               Standard          t
     Parameter       Value       Error       Statistic
    -----------   -----------   ------------   -----------
    Intercept        5.57217         Fixed          Fixed
        MA{1}     -0.0253657      0.221971      -0.114275
      SMA{12}       -0.80255     0.0527051       -15.2272
        Beta1     0.00275883       0.10139      0.0272102
     Variance     0.00724633   0.000159736        45.3645
```

`MA{1}` and `Beta1` are not significantly different from 0. You can remove these parameters from the model, possibly add other parameters (e.g., AR parameters), and compare multiple model fits using `aicbic`. Note that the estimation and presample should be the same over competing models.

References:

Box, G. E. P., G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis: Forecasting and Control*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

See Also

`aicbic` | estimate | regARIMA

More About

- “Maximum Likelihood Estimation of regARIMA Models” on page 4-86
- “Presample Values for regARIMA Model Estimation” on page 4-95
- “Intercept Identifiability in Regression Models with ARIMA Errors” on page 4-130

Select a Regression Model with ARIMA Errors

Regression models with ARIMA time series errors contain two components: a regression model and an error model. Typically, the research emphasis is on the regression model. But, in order to properly select the predictors, you must properly model the error structure. The following steps outline the infinite loop that you might experience when selecting a regression model with ARIMA errors:

1. To determine the appropriate lags to include in the error model, you must infer the unconditional disturbances, u_t, where $t = 1,...,T$.
2. To properly infer u_t from the regression model, you must estimate the regression model including all appropriate predictors, X_t.
3. To determine the appropriate predictors, you must properly model the error structure, u_t. That is, you must determine the appropriate lags for the error model.

If econometric theory suggests that a particular regression model is appropriate, then fit the regression model over varying autoregressive and moving average degrees. Choose the model that yields the lowest information criterion. For example, see "Choose Lags for an ARMA Error Model" on page 4-125.

However, if you want statistical methods to choose both the regression and error models, then one way to choose an appropriate regression model with ARIMA errors (as recommended in [1]) is to:

1. Check each variable for stationarity. Transform or difference the nonstationary series to make them stationary. To maintain the interpretation of the relationships between the variables, transform or difference all variables the same way. For details, see "Data Transformations" on page 2-2.
2. Assume that the error model is AR(2) or an appropriate multiplicative seasonal AR(2) model. Estimate the regression model using `estimate` including all predictors and the possibly transformed or differenced data.
3. Infer u_t from the fitted regression model using `infer`.
4. Determine an appropriate ARIMA error model. For details, see "Box-Jenkins Methodology" on page 3-2 and "Autocorrelation and Partial Autocorrelation" on page 3-13.
5. Use the new ARIMA error model to reestimate the regression model with ARIMA errors.

6. Check that the innovations (ε_t) are a white noise sequence. For details, see "Residual Diagnostics" on page 3-90. If the innovations are not a white noise sequence, then choose a different ARIMA error model, reestimate the regression model with ARIMA errors, and recheck the innovations.
7. Compute information criteria for the final model using `aicbic`.
8. Perform the full procedure repeatedly using a subset of predictors for each trial. Choose the model with the lowest information criterion.

References

[1] Hyndman, R. J. and G. Athanasopoulos. "Dynamic Regression Models." *Forecasting: Principles and Practice*. April 2013. http://www.otexts.org/fpp/9/1.

See Also

`aicbic` | estimate | infer

Related Examples

- "Choose Lags for an ARMA Error Model" on page 4-125
- "Estimate a Regression Model with ARIMA Errors" on page 4-105

More About

- "Regression Models with Time Series Errors" on page 4-6
- "Box-Jenkins Methodology" on page 3-2
- "Data Transformations" on page 2-2
- "Autocorrelation and Partial Autocorrelation" on page 3-13
- "Residual Diagnostics" on page 3-90

Choose Lags for an ARMA Error Model

This example shows how to use Akaike Information Criterion (AIC) to select the nonseasonal autoregressive and moving average lag polynomial degrees for a regression model with ARMA errors.

Estimate several models by passing the data to `estimate`. Vary the autoregressive and moving average degrees p and q, respectively. Each fitted model contains an optimized loglikelihood objective function value, which you pass to `aicbic` to calculate AIC fit statistics. The AIC fit statistic penalizes the optimized loglikelihood function for complexity (i.e., for having more parameters). Assume that econometric theory dictates that the predictors of the regression model are appropriate.

Simulate response and predictor data for the regression model with ARMA errors:

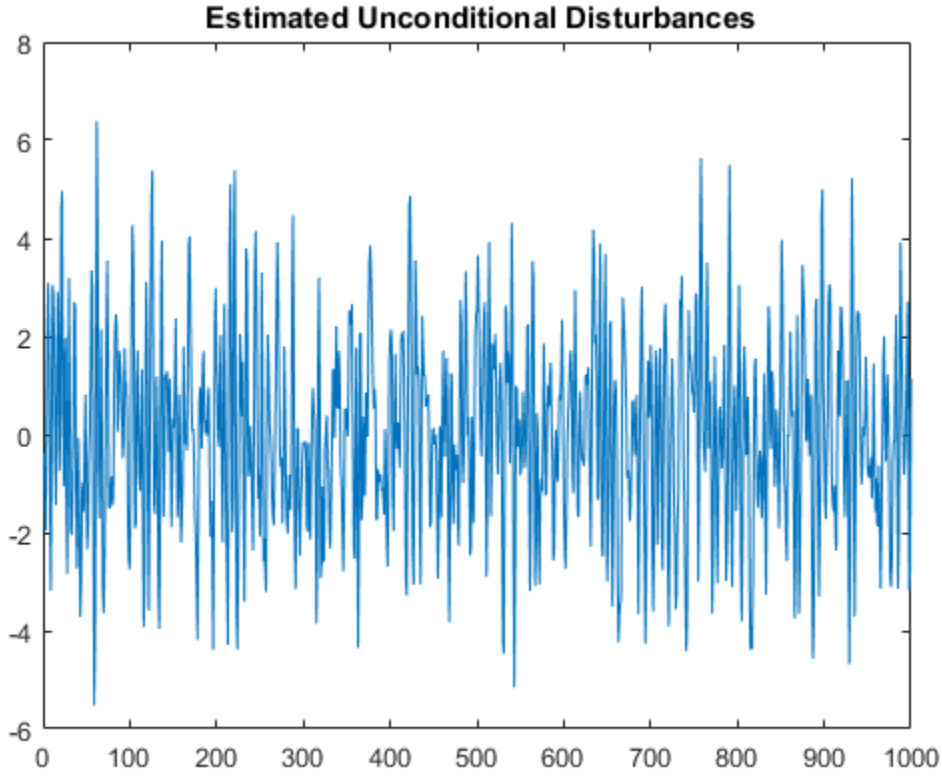
$$y_t = 2 + X_t \begin{bmatrix} -2 \\ 1.5 \end{bmatrix} + u_t$$
$$u_t = 0.75u_{t-1} - 0.5u_{t-2} + \varepsilon_t + 0.7\varepsilon_{t-1},$$

where ε_t is Gaussian with mean 0 and variance 1.

```
Mdl = regARIMA('Intercept',2,'Beta',[-2; 1.5],...
    'AR',{0.75, -0.5},'MA',0.7,'Variance',1);

rng(2);            % For reproducibility
X = randn(1000,2); % Predictors
y = simulate(Mdl,1000,'X',X);
```

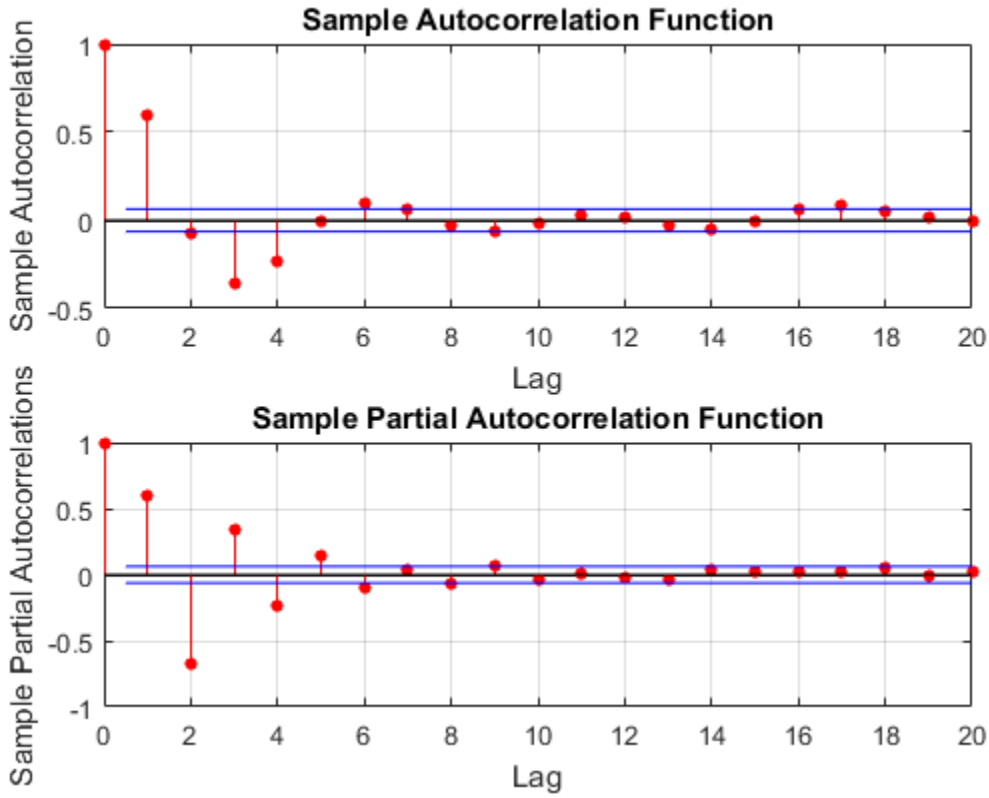
Regress the response onto the predictors. Plot the residuals (i.e., estimated unconditional disturbances).

```
Fit = fitlm(X,y);
u = Fit.Residuals.Raw;

figure
plot(u)
title('{\bf Estimated Unconditional Disturbances}')
```



Plot the ACF and PACF of the residuals.

```
figure
subplot(2,1,1)
autocorr(u)
subplot(2,1,2)
parcorr(u)
```



The ACF and PACF decay slowly, which indicates an ARMA process. It is difficult to use these correlograms to determine the lags. However, it seems reasonable that both polynomials should have four or fewer lags based on the lengths of the autocorrelations and partial autocorrelations.

To determine the number of AR and MA lags, define and estimate regression models with ARMA(p, q) errors by varying p = 1,..,4 and q = 1,...,4. Store the optimized loglikelihood objective function value for each model fit.

```
pMax = 4;
qMax = 4;
LogL = zeros(pMax,qMax);
SumPQ = LogL;
```

```
for p = 1:pMax
    for q = 1:qMax
        ToEstMdl = regARIMA(p,0,q);
        [~,~,LogL(p,q)] = estimate(ToEstMdl,y,'X',X,...
            'Display','off');
        SumPQ(p,q) = p+q;
    end
end
```

Calculate AIC for each model fit. The number of parameters is $p + q + 4$ (i.e., the intercept, two regression coefficients, and innovation variance).

```
logL = reshape(LogL,pMax*qMax,1);...
    % Elements taken column-wise
numParams = reshape(SumPQ,pMax*qMax,1) + 4;
aic = aicbic(logL,numParams);
AIC = reshape(aic,pMax,qMax)

minAIC = min(aic)
[bestP,bestQ] = find(AIC == minAIC)
```

```
AIC =

   1.0e+03 *

    3.1323    3.0195    2.9984    2.9462
    2.9280    2.9297    2.9314    2.9331
    2.9297    2.9305    2.9321    2.9345
    2.9314    2.9325    2.9343    2.9358


minAIC =

   2.9280e+03


bestP =

     2


bestQ =

     1
```

The best fitting model is the regression model with AR(2,1) errors because its corresponding AIC is the lowest.

See Also

`aicbic` | estimate

Related Examples

- “Estimate a Regression Model with ARIMA Errors” on page 4-105

More About

- “Select a Regression Model with ARIMA Errors” on page 4-123

Intercept Identifiability in Regression Models with ARIMA Errors

In this section...

Intercept Identifiability

A regression model with ARIMA errors has the following general form ($t = 1,...,T$)

$$y_t = c + X_t\beta + u_t$$
$$a(L)A(L)(1-L)^D\left(1-L^s\right)u_t = b(L)B(L)\varepsilon_t,$$

where

- $t = 1,...,T$.
- y_t is the response series.
- X_t is row t of X, which is the matrix of concatenated predictor data vectors. That is, X_t is observation t of each predictor series.
- c is the regression model intercept.
- β is the regression coefficient.
- u_t is the disturbance series.
- ε_t is the innovations series.
- $L^j y_t = y_{t-j}$.
- $a(L) = \left(1 - a_1L - ... - a_pL^p\right)$, which is the degree p, nonseasonal autoregressive polynomial.
- $A(L) = \left(1 - A_1L - ... - A_{p_s}L^{p_s}\right)$, which is the degree p_s, seasonal autoregressive polynomial.
- $(1-L)^D$, which is the degree D, nonseasonal integration polynomial.

- $\left(1-L^{s}\right)$, which is the degree s, seasonal integration polynomial.
- $b(L)=\left(1+b_{1}L+\ldots+b_{q}L^{q}\right)$, which is the degree q, nonseasonal moving average polynomial.
- $B(L)=\left(1+B_{1}L+\ldots+B_{q_{s}}L^{q_{s}}\right)$, which is the degree q_s, seasonal moving average polynomial.
- If you specify that $D = s = 0$ (i.e., you do not indicate seasonal or nonseasonal integration), then every parameter is identifiable. In other words, the likelihood objective function is sensitive to a change in a parameter, given the data.
- If you specify that $D > 0$ or $s > 0$, and you want to estimate the intercept, c, then c is not identifiable.

You can show that this is true.

- Consider Equation 4-8. Solve for u_t in the second equation and substitute it into the first.

  $$y_t = c + X_t\beta + \mathrm{H}^{-1}(L)\mathrm{N}(L)\varepsilon_t,$$

  where

  - $\mathrm{H}(L) = a(L)(1-L)^{D}A(L)(1-L^{s})$.
  - $\mathrm{N}(L) = b(L)B(L)$.
- The likelihood function is based on the distribution of ε_t. Solve for ε_t.

  $$\varepsilon_t = \mathrm{N}^{-1}(L)\mathrm{H}(L)y_t + \mathrm{N}^{-1}(L)\mathrm{H}(L)c + \mathrm{N}^{-1}(L)\mathrm{H}(L)X_t\beta.$$

- Note that $L^{j}c = c$. The constant term contributes to the likelihood as follows.

  $$\begin{aligned}\mathrm{N}^{-1}(L)\mathrm{H}(L)c &= \mathrm{N}^{-1}(L)a(L)A(L)(1-L)^{D}(1-L^{s})c\\ &= \mathrm{N}^{-1}(L)a(L)A(L)(1-L)^{D}(c-c)\\ &= 0\end{aligned}$$

or

$$\begin{aligned}\mathrm{N}^{-1}(L)\mathrm{H}(L)c &= \mathrm{N}^{-1}(L)a(L)A(L)(1-L^s)(1-L)^D c \\ &= \mathrm{N}^{-1}(L)a(L)A(L)(1-L^s)(1-L)^{D-1}(1-L)c \\ &= \mathrm{N}^{-1}(L)a(L)A(L)(1-L^s)(1-L)^{D-1}(c-c) \\ &= 0.\end{aligned}$$

Therefore, when the ARIMA error model is integrated, the likelihood objective function based on the distribution of ε_t is invariant to the value of c.

In general, the effective constant in the equivalent ARIMAX representation of a regression model with ARIMA errors is a function of the compound autoregressive coefficients and the original intercept c, and incorporates a nonlinear constraint. This constraint is seamlessly incorporated for applications such as Monte Carlo simulation of integrated models with nonzero intercepts. However, for estimation, the ARIMAX model is unable to identify the constant in the presence of an integrated polynomial, and this results in spurious or unusual parameter estimates.

You should exclude an intercept from integrated models in most applications.

Intercept Identifiability Illustration

As an illustration, consider the regression model with ARIMA(2,1,1) errors without predictors

$$\begin{aligned}y_t &= 0.5 + u_t \\ \left(1 - 0.8L + 0.4L^2\right)(1-L)u_t &= (1 + 0.3L)\varepsilon_t,\end{aligned}$$

or

$$\begin{aligned}y_t &= 0.5 + u_t \\ \left(1 - 1.8L + 1.2L^2 - 0.4L^3\right)u_t &= (1 + 0.3L)\varepsilon_t.\end{aligned}$$

You can rewrite Equation 4-10 using substitution and some manipulation

$$y_t = (1 - 1.8 + 1.2 - 0.4)0.5 + 1.8y_{t-1} - 1.2y_{t-2} + 0.4y_{t-3} + \varepsilon_t + 0.3\varepsilon_{t-1}.$$

Note that

$$(1 - 1.8 + 1.2 - 0.4)0.5 = 0(0.5) = 0.$$

Therefore, the regression model with ARIMA(2,1,1) errors in Equation 4-10 has an ARIMA(2,1,1) model representation

$$y_t = 1.8y_{t-1} - 1.2y_{t-2} + 0.4y_{t-3} + \varepsilon_t + 0.3\varepsilon_{t-1}.$$

You can see that the constant is not present in the model (which implies its value is 0), even though the value of the regression model with ARIMA errors intercept is 0.5.

You can also simulate this behavior. Start by specifying the regression model with ARIMA(2,1,1) errors in Equation 4-10.

```
Mdl = regARIMA('D',1,'AR',{0.8 -0.4},'MA',0.3,...
    'Intercept',0.5,'Variance', 0.2);
```

Simulate 1000 observations.

```
rng(1);
T = 1000;
y = simulate(Mdl, T);
```

Fit `Mdl` to the data.

```
ToEstMdl = regARIMA('ARLags',1:2,'MALags',1,'D',1);...
    % "Empty" model to pass into estimate
[EstMdl,EstParamCov] = estimate(ToEstMdl,y,'Display','params');
```

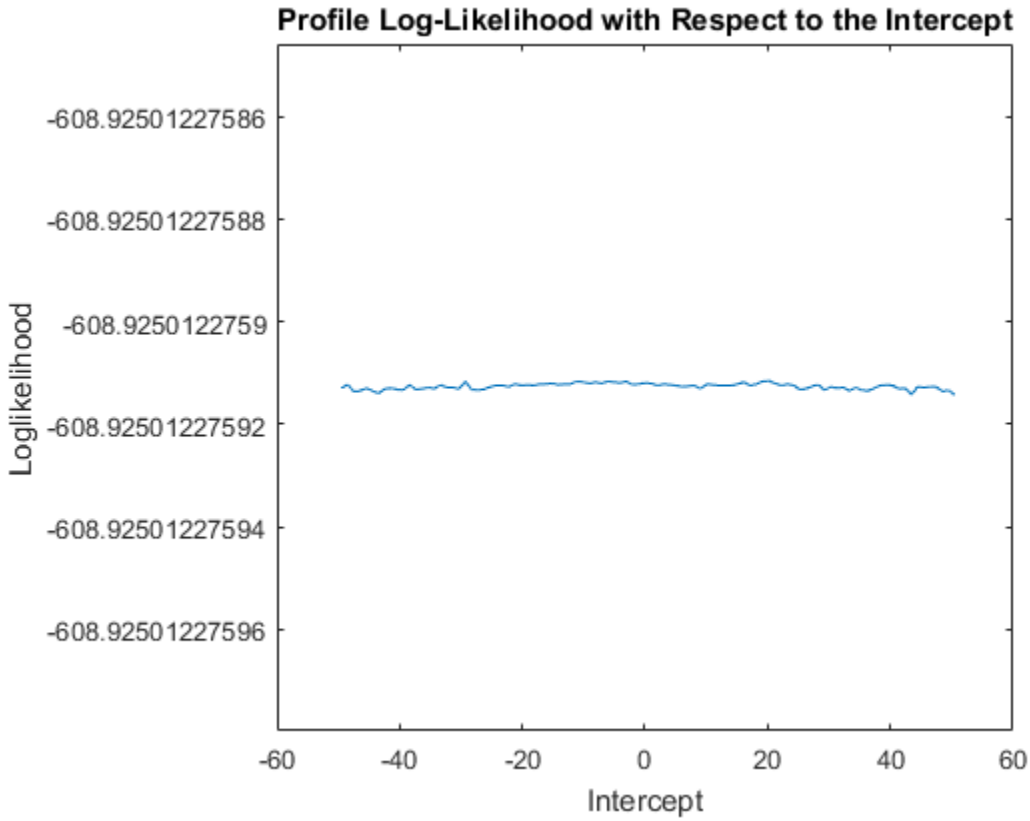
```
Warning: When ARIMA error model is integrated, the intercept is unidentifiable
and cannot be estimated; a NaN is returned.
```

```
ARIMA(2,1,1) Error Model:
--------------------------
Conditional Probability Distribution: Gaussian

                              Standard          t
 Parameter       Value          Error       Statistic
-----------   -----------   ------------   -----------
Intercept           NaN           NaN            NaN
    AR{1}      0.896466     0.0485066        18.4813
    AR{2}     -0.451015     0.0389158       -11.5895
    MA{1}      0.188039      0.054505        3.44994
 Variance      0.197893    0.00835124        23.6963
```

`estimate` displays a warning to inform you that the intercept is not identifiable, and sets its estimate, standard error, and *t*-statistic to `NaN`.

Plot the profile likelihood for the intercept.

```
c = linspace(Mdl.Intercept - 50,...
    Mdl.Intercept + 50,100); % Grid of intercepts
logL = nan(numel(c),1); % For preallocation

for i = 1:numel(logL)
    EstMdl.Intercept = c(i);
    [~,~,~,logL(i)] = infer(EstMdl,y);
end

figure
plot(c,logL)
title('Profile Log-Likelihood with Respect to the Intercept')
xlabel('Intercept')
ylabel('Loglikelihood')
```



The loglikelihood does not change over the grid of intercept values. The slight oscillation is a result of the numerical routine used by `infer`.

Related Examples

- “Estimate a Regression Model with ARIMA Errors” on page 4-105

Compare Alternative ARIMA Model Representations

In this section...

regARIMA to ARIMAX Model Conversion

ARIMAX models and regression models with ARIMA errors are closely related, and the choice of which to use is generally dictated by your goals for the analysis. If your objective is to fit a parsimonious model to data and forecast responses, then there is very little difference between the two models.

If you are more interested in preserving the usual interpretation of a regression coefficient as a measure of sensitivity, i.e., the effect of a unit change in a predictor variable on the response, then use a regression model with ARIMA errors. Regression coefficients in ARIMAX models do not possess that interpretation because of the dynamic dependence on the response [1].

Suppose that you have the parameter estimates from a regression model with ARIMA errors, and you want to see how the model structure compares to ARIMAX model. Or, suppose you want some insight as to the underlying relationship between the two models.

The ARIMAX model is ($t = 1,\ldots,T$):

$$\mathrm{H}(L)y_t = c + X_t\beta + \mathrm{N}(L)\varepsilon_t,$$

where

- y_t is the univariate response series.
- X_t is row t of X, which is the matrix of concatenated predictor series. That is, X_t is observation t of each predictor series.
- β is the regression coefficient.
- c is the regression model intercept.

- $\mathrm{H}(L) = \phi(L)(1-L)^D\Phi(L)(1-L^s) = 1-\eta_1 L-\eta_2 L^2-\ldots-\eta_P L^P$, which is the degree P lag operator polynomial that captures the combined effect of the seasonal and nonseasonal autoregressive polynomials, and the seasonal and nonseasonal integration polynomials. For more details on notation, see "Multiplicative ARIMA Model" on page 5-46.
- $\mathrm{N}(L) = \theta(L)\Theta(L) = 1+\nu_1 L+\nu_2 L^2+\ldots+\nu_Q L^Q$, which is the degree Q lag operator polynomial that captures the combined effect of the seasonal and nonseasonal moving average polynomials.
- ε_t is a white noise innovation process.

The regression model with ARIMA errors is (t = 1,...,T)

$$y_t = c + X_t\beta + u_t$$
$$A(L)u_t = B(L)\varepsilon_t,$$

where

- u_t is the unconditional disturbances process.
- $A(L) = \phi(L)(1-L)^D\Phi(L)(1-L^s) = 1-a_1 L-a_2 L^2-\ldots-a_P L^P$, which is the degree P lag operator polynomial that captures the combined effect of the seasonal and nonseasonal autoregressive polynomials, and the seasonal and nonseasonal integration polynomials.
- $B(L) = \theta(L)\Theta(L) = 1+b_1 L+b_2 L^2+\ldots+b_Q L^Q$, which is the degree Q lag operator polynomial that captures the combined effect of the seasonal and nonseasonal moving average polynomials.

The values of the variables defined in Equation 4-12 are not necessarily equivalent to the values of the variables in Equation 4-11, even though the notation might be similar.

Illustrate regARIMA to ARIMAX Model Conversion

Consider Equation 4-12, the regression model with ARIMA errors. Use the following operations to convert the regression model with ARIMA errors to its corresponding ARIMAX model.

1 Solve for u_t..

$$y_t = c + X_t\beta + u_t$$
$$u_t = \frac{B(L)}{A(L)}\varepsilon_t.$$

2 Substitute u_t into the regression equation.

$$y_t = c + X_t\beta + \frac{B(L)}{A(L)}\varepsilon_t$$
$$A(L)y_t = A(L)c + A(L)X_t\beta + B(L)\varepsilon_t.$$

3 Solve for y_t.

$$y_t = A(L)c + A(L)X_t\beta + \sum_{k=1}^{P} a_k y_{t-k} + B(L)\varepsilon_t$$
$$= A(L)c + Z_t\Gamma + \sum_{k=1}^{P} a_k y_{t-k} + B(L)\varepsilon_t.$$

In Equation 4-13,

- $A(L)c = (1 - a_1 - a_2 - ... - a_P)c$. That is, the constant in the ARIMAX model is the intercept in the regression model with ARIMA errors with a nonlinear constraint. Though applications, such as simulate, handle this constraint, estimate cannot incorporate such a constraint. In the latter case, the models are equivalent when you fix the intercept and constant to 0.
- In the term $A(L)X_t\beta$, the lag operator polynomial $A(L)$ filters the T-by-1 vector $X_t\beta$, which is the linear combination of the predictors weighted by the regression coefficients. This filtering process requires P presample observations of the predictor series.
- arima constructs the matrix Z_t as follows:
  - Each column of Z_t corresponds to each term in $A(L)$.
  - The first column of Z_t is the vector $X_t\beta$.
  - The second column of Z_t is a sequence of d_2 `NaN`s (d_2 is the degree of the second term in $A(L)$), followed by the product $L^{d_j}X_t\beta$. That is, the software attaches

d_2 NaNs at the beginning of the T-by-1 column, attaches $X_t\beta$ after the NaNs, but truncates the end of that product by d_2 observations.

- The jth column of Z_t is a sequence of d_j NaNs (d_j is the degree of the jth term in $A(L)$), followed by the product $L^{d_j}X_t\beta$. That is, the software attaches d_j NaNs at the beginning of the T-by-1 column, attaches $X_t\beta$ after the NaNs, but truncates the end of that product by d_j observations.

.

- $\Gamma = [1\ -\alpha_1\ -\alpha_2\ \ldots\ -\alpha_P]'$.

  The `arima` converter removes all zero-valued autoregressive coefficients of the difference equation. Subsequently, the `arima` converter does not associate zero-valued autoregressive coefficients with columns in Z_t, nor does it include corresponding, zero-valued coefficients in Γ.

4 Rewrite Equation 4-13,

$$y_t = (1 - \sum_{k=1}^{P} a_k)c + X_t\beta - \sum_{k=1}^{P} a_k X_{t-k}\beta + \sum_{k=1}^{P} a_k y_{t-k} + \varepsilon_t + \sum_{k=1}^{Q} \varepsilon_{t-k}.$$

For example, consider the following regression model whose errors are ARMA(2,1):

$$\begin{aligned} y_t &= 0.2 + 0.5X_t + u_t \\ \left(1 - 0.8L + 0.4L^2\right)u_t &= (1 + 0.3L)\varepsilon_t. \end{aligned}$$

The equivalent ARMAX model is:

$$\begin{aligned} y_t &= 0.12 + \left(0.5 - 0.4L + 0.2L^2\right)X_t + 0.8y_{t-1} - 0.4y_{t-2} + (1 + 0.3L)\varepsilon_t \\ &= 0.12 + Z_t\Gamma + 0.8y_{t-1} - 0.4y_{t-2} + (1 + 0.3L)\varepsilon_t, \end{aligned}$$

or

$$(1 - 0.8L + 0.4L^2)y_t = 0.12 + Z_t\Gamma + (1 + 0.3L)\varepsilon_t,$$

where $\Gamma = [1\ -0.8\ 0.4]'$ and

$$Z_t = 0.5\begin{bmatrix} x_1 & \text{NaN} & \text{NaN} \\ x_2 & x_1 & \text{NaN} \\ x_3 & x_2 & x_1 \\ \vdots & \vdots & \vdots \\ x_T & x_{T-1} & x_{T-2} \end{bmatrix}.$$

This model is not integrated because all of the eigenvalues associated with the AR polynomial are within the unit circle, but the predictors might affect the otherwise stable process. Also, you need presample predictor data going back at least 2 periods to, for example, fit the model to data.

You can illustrate this further through simulation and estimation.

1. Specify the regression model with ARIMA errors in Equation 4-14.

```
Mdl1 = regARIMA('Intercept',0.2,'AR',{0.8 -0.4},...
                'MA',0.3,'Beta',[0.3 -0.2],'Variance',0.2);
```

2. Generate presample observations and predictor data.

```
rng(1);   % For reproducibility
T = 100;
maxPQ = max(Mdl1.P,Mdl1.Q);
numObs  = T + maxPQ;...
    % Adjust number of observations to account for presample
X1 = randn(numObs,2); % Simulate predictor data
u0 = randn(maxPQ,1);  % Presample unconditional disturbances u(t)
e0 = randn(maxPQ,1);  % Presample innovations e(t)
```

3. Simulate data from `Mdl1`.

```
rng(100) % For reproducibility
[y1,e1,u1] = simulate(Mdl1,T,'U0',u0,...
    'E0',e0,'X',X1);
```

4. Convert `Mdl1` to an ARIMAX model.

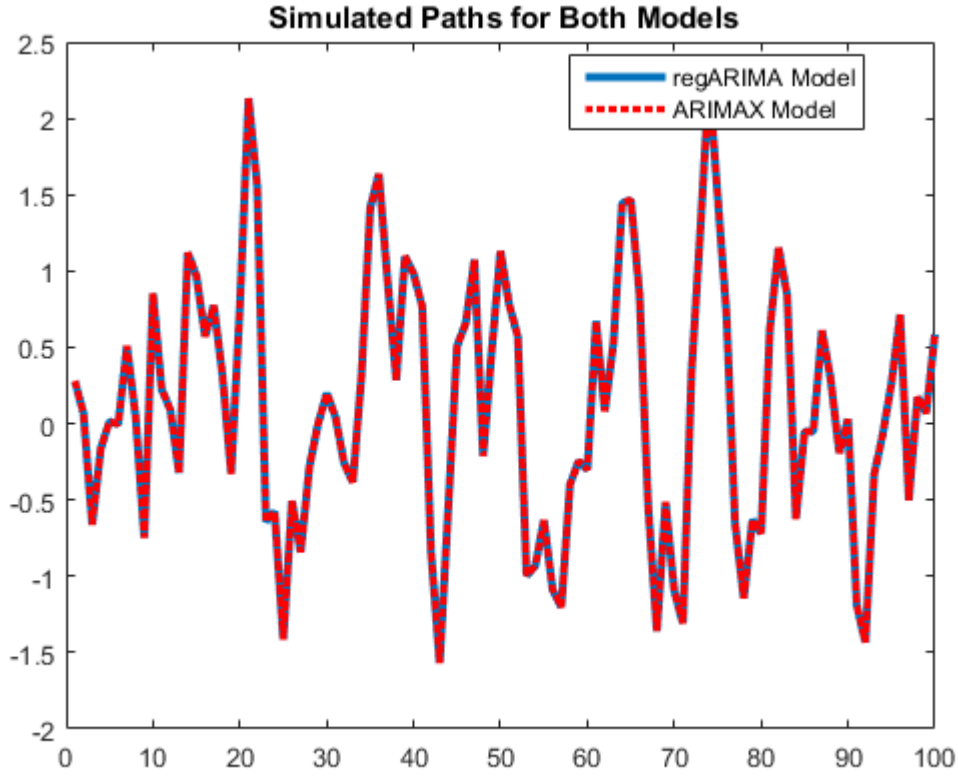
```
[Mdl2,X2] = arima(Mdl1,'X',X1);
Mdl2
```

```
Mdl2 =

    ARIMAX(2,0,1) Model:
```

```
---------------------
Distribution: Name = 'Gaussian'
           P: 2
           D: 0
           Q: 1
    Constant: 0.12
          AR: {0.8 -0.4} at Lags [1 2]
         SAR: {}
          MA: {0.3} at Lags [1]
         SMA: {}
        Beta: [1 -0.8 0.4]
    Variance: 0.2
```

5 Generate presample responses for the ARIMAX model to ensure consistency with `Mdl1`. Simulate data from `Mdl2`.

```
y0 = Mdl1.Intercept + X1(1:maxPQ,:)*Mdl1.Beta' + u0;
rng(100)
y2 = simulate(Mdl2,T,'Y0',y0,'E0',e0,'X',X2);

figure
plot(y1,'LineWidth',3)
hold on
plot(y2,'r:','LineWidth',2.5)
hold off
title('{\bf Simulated Paths for Both Models}')
legend('regARIMA Model','ARIMAX Model','Location','Best')
```



The simulated paths are equivalent because the `arima` converter enforces the nonlinear constraint when it converts the regression model intercept to the ARIMAX model constant.

6 Fit a regression model with ARIMA errors to the simulated data.

```
ToEstMdl1 = regARIMA('ARLags',[1 2],'MALags',1);
EstMdl1 = estimate(ToEstMdl1,y1,'E0',e0,'U0',u0,'X',X1);
```

```
Regression with ARIMA(2,0,1) Error Model:
----------------------------------------
Conditional Probability Distribution: Gaussian
```

```
                            Standard          t
 Parameter       Value        Error       Statistic
-----------   -----------   ------------   -----------
Intercept       0.140736      0.101405        1.38787
    AR{1}       0.830611      0.137504        6.04065
    AR{2}      -0.454025      0.116397       -3.90067
    MA{1}       0.428031      0.151453        2.82616
    Beta1       0.295519     0.0229383        12.8832
    Beta2      -0.176007     0.0306069       -5.75057
 Variance       0.182313     0.0277648        6.56633
```

7 Fit an ARIMAX model to the simulated data.

```
ToEstMdl2 = arima('ARLags',[1 2],'MALags',1);
EstMdl2 = estimate(ToEstMdl2,y2,'E0',e0,'Y0',...
    y0,'X',X2);
```

```
    ARIMAX(2,0,1) Model:
    --------------------
    Conditional Probability Distribution: Gaussian

                            Standard          t
     Parameter       Value        Error       Statistic
    -----------   -----------   ------------   -----------
     Constant      0.0849961     0.0642166        1.32359
        AR{1}       0.831361      0.136345        6.09748
        AR{2}      -0.455993       0.11788       -3.86828
        MA{1}          0.426      0.157526        2.70431
        Beta1        1.05303      0.136849        7.69485
        Beta2        -0.6904      0.192617       -3.58432
        Beta3       0.453993      0.153522        2.95718
     Variance       0.181119     0.0288359        6.28103
```

8 Convert `EstMdl1` to an ARIMAX model.

```
ConvertedMdl2 = arima(EstMdl1,'X',X1)
```

```
ConvertedMdl2 =

    ARIMAX(2,0,1) Model:
    --------------------
    Distribution: Name = 'Gaussian'
               P: 2
```

```
       D: 0
       Q: 1
Constant: 0.087737
      AR: {0.830611 -0.454025} at Lags [1 2]
     SAR: {}
      MA: {0.428031} at Lags [1]
     SMA: {}
    Beta: [1 -0.830611 0.454025]
Variance: 0.182313
```

The estimated ARIMAX model constant is not equivalent to the ARIMAX model constant converted from the regression model with ARIMA errors. In other words, `EstMdl2.Constant = 0.0849961` and `ConvertedMdl2.Constant = 0.087737`. This is because `estimate` does not enforce the nonlinear constraint that the `arima` converter enforces. As a result, the other estimates are not equivalent either, albeit close.

References

[1] Hyndman, R. J. (2010, October). "The ARIMAX Model Muddle." *Rob J. Hyndman*. Retrieved February 7, 2013 from http://robjhyndman.com/researchtips/arimax/.

See Also

arima | estimate | estimate

Related Examples

- "Estimate a Regression Model with ARIMA Errors" on page 4-105

Simulate Regression Models with ARMA Errors

In this section...

Simulate an AR Error Model

This example shows how to simulate sample paths from a regression model with AR errors without specifying presample disturbances.

Specify the regression model with AR(2) errors:

$$y_t = 2 + X_t \begin{bmatrix} -2 \\ 1.5 \end{bmatrix} + u_t$$
$$u_t = 0.75u_{t-1} - 0.5u_{t-2} + \varepsilon_t,$$

where ε_t is Gaussian with mean 0 and variance 1.

```
Beta = [-2; 1.5];
Intercept = 2;
a1 = 0.75;
a2 = -0.5;
Variance = 1;
Mdl = regARIMA('AR',{a1, a2},'Intercept',Intercept,...
    'Beta',Beta,'Variance',Variance);
```

Generate two length $T = 50$ predictor series by random selection from the standard Gaussian distribution.

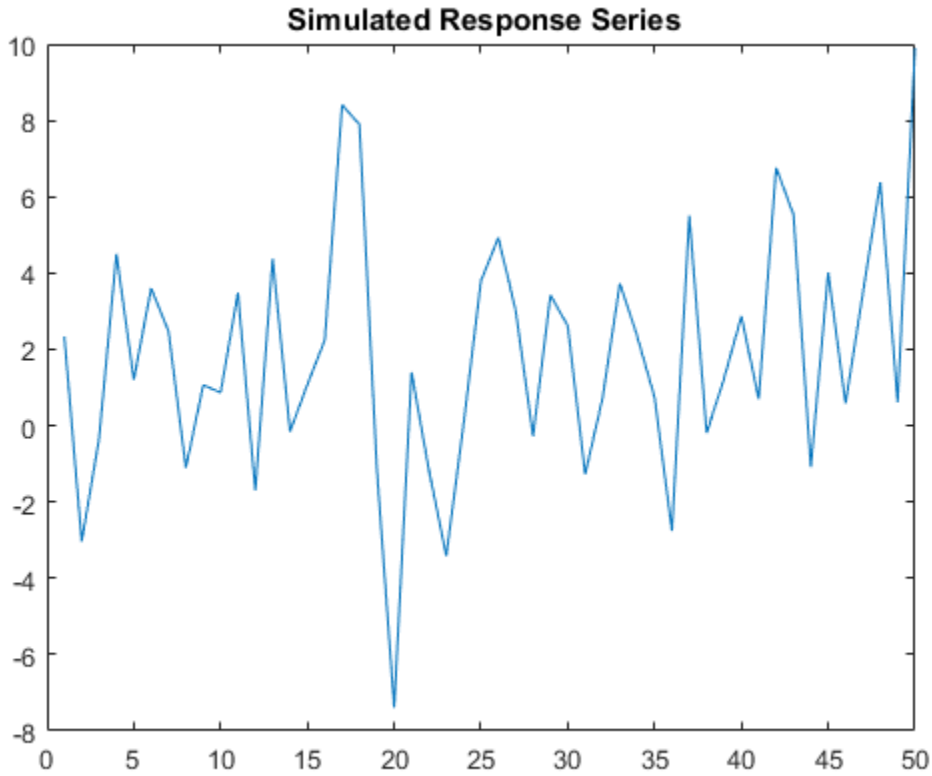
```
T = 50;
rng(1); % For reproducibility
X = randn(T,2);
```

The software treats the predictors as nonstochastic series.

Generate and plot one sample path of responses from `Mdl`.

```
rng(2);
ySim = simulate(Mdl,T,'X',X);
```

```
figure
plot(ySim)
title('{\bf Simulated Response Series}')
```

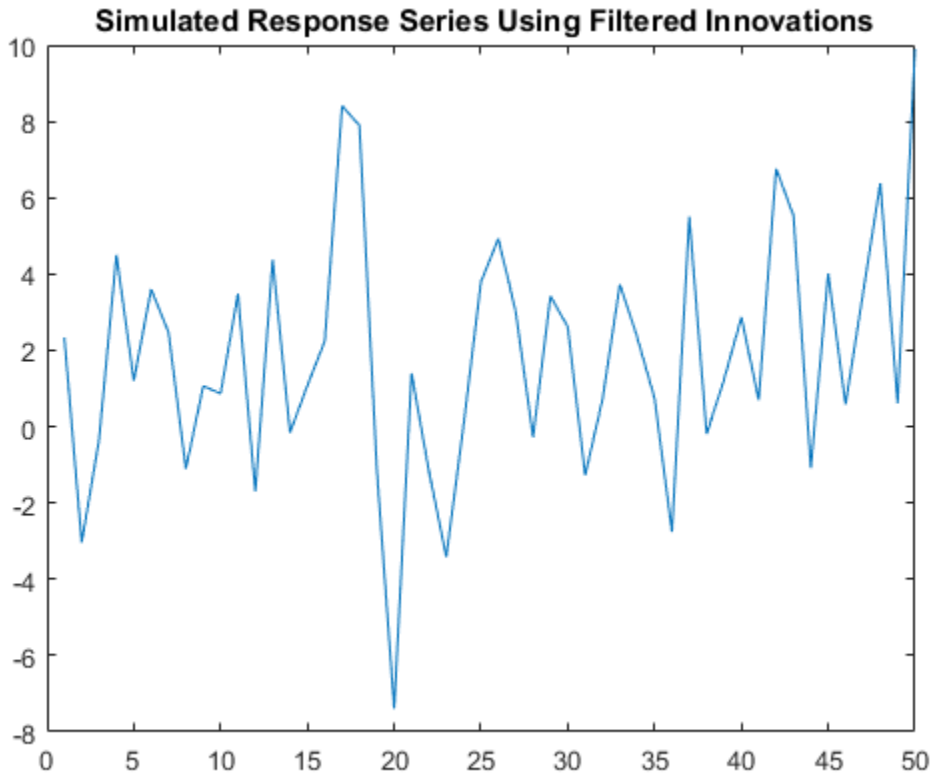


`simulate` requires `P` = 2 presample unconditional disturbances (u_t) to initialize the error series. Without them, as in this case, `simulate` sets the necessary presample unconditional disturbances to 0.

Alternatively, filter a random innovation series through `Mdl` using `filter`.

```
rng(2);
e = randn(T,1);
yFilter = filter(Mdl,e,'X',X);
```

```
figure
plot(yFilter)
title('{\bf Simulated Response Series Using Filtered Innovations}')
```

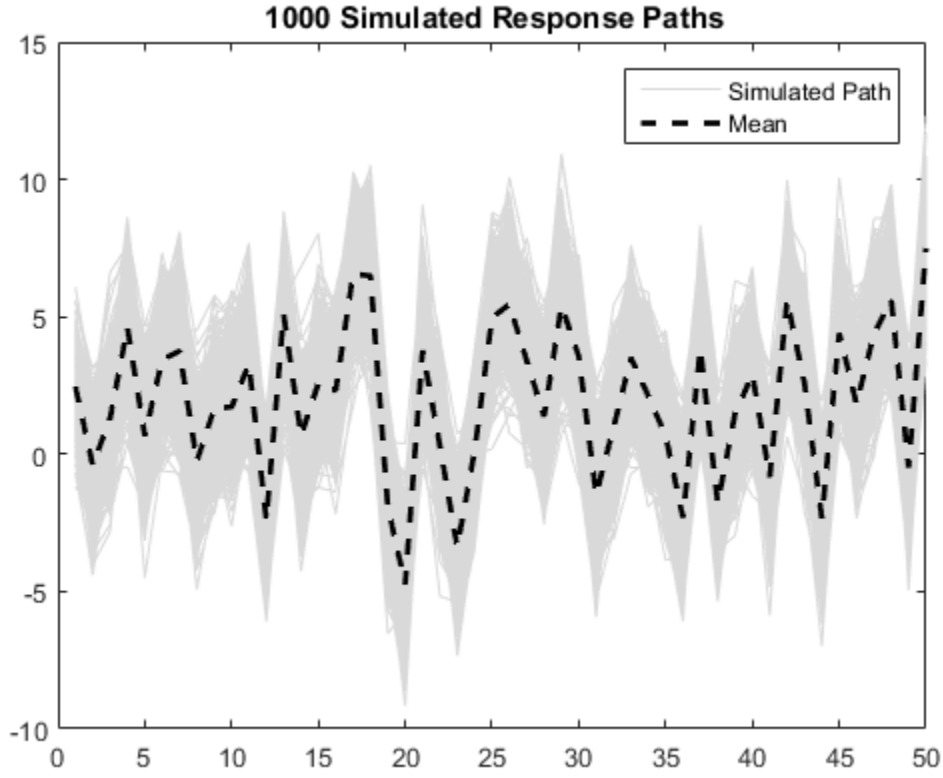


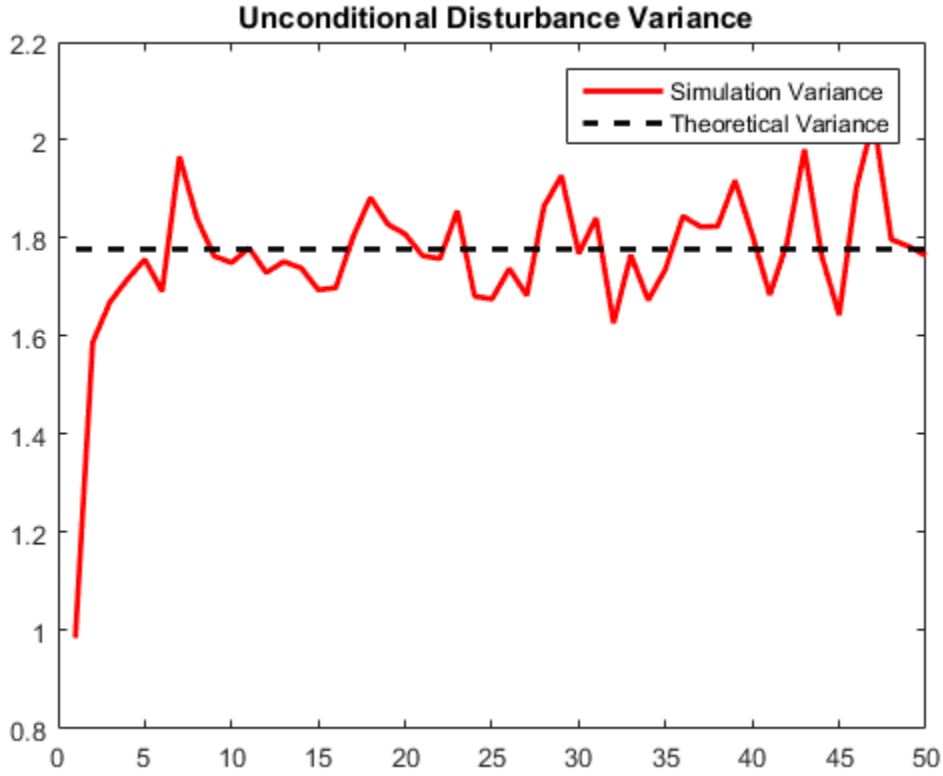
The plots suggest that the simulated responses and the responses generated from the filtered innovations are equivalent.

Simulate 1000 response paths from `Mdl`. Assess transient effects by plotting the unconditional disturbance (`U`) variances across the simulated paths at each period.

```
numPaths = 1000;
[Y,~,U] = simulate(Mdl,T,'NumPaths',numPaths,'X',X);

figure
```

```
h1 = plot(Y,'Color',[.85,.85,.85]);
title('{\bf 1000 Simulated Response Paths}')
hold on
h2 = plot(1:T,Intercept+X*Beta,'k--','LineWidth',2);
legend([h1(1),h2],'Simulated Path','Mean')
hold off

figure
h1 = plot(var(U,0,2),'r','LineWidth',2);
hold on
theoVarFix = ((1-a2)*Variance)/((1+a2)*((1-a2)^2-a1^2));
h2 = plot([1 T],[theoVarFix theoVarFix],'k--','LineWidth',2);
title('{\bf Unconditional Disturbance Variance}')
legend([h1,h2],'Simulation Variance','Theoretical Variance')
hold off
```


1000 Simulated Response Paths
Simulated Path
Mean
15
10
5
0
-5
-10
0
5
10
15
20
25
30
35
40
45
50



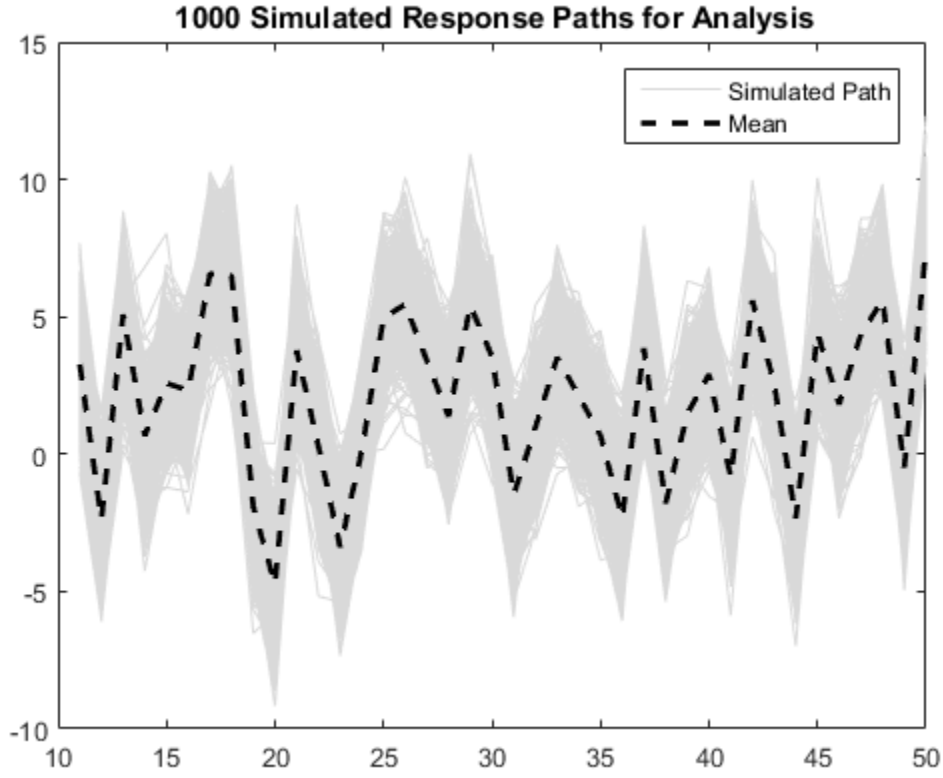
The simulated response paths follow their theoretical mean, $c + X\beta$, which is not constant over time (and might look nonstationary).

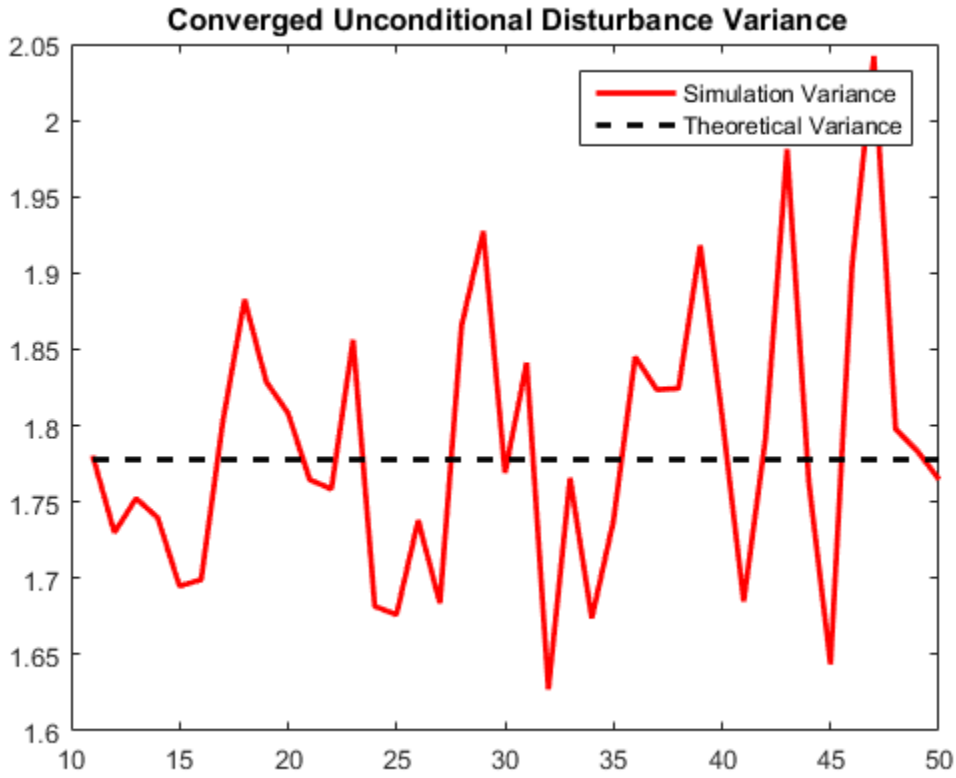
The variance of the process is not constant, but levels off at the theoretical variance by the 10th period. The theoretical variance of the AR(2) error model is

$$\frac{(1-a_2)\,\sigma_\varepsilon^2}{(1+a_2)\left[(1-a_2)^2-a_1^2\right]} = \frac{(1+0.5)}{(1-0.5)\left[(1+0.5)^2-0.75^2\right]} = 1.78$$

You can reduce transient effects is by partitioning the simulated data into a burn-in portion and a portion for analysis. Do not use the burn-in portion for analysis. Include enough periods in the burn-in portion to overcome the transient effects.

```
burnIn = 1:10;
notBurnIn = burnIn(end)+1:T;
Y = Y(notBurnIn,:);
X = X(notBurnIn,:);
U = U(notBurnIn,:);
figure
h1 = plot(notBurnIn,Y,'Color',[.85,.85,.85]);
hold on
h2 = plot(notBurnIn,Intercept+X*Beta,'k--','LineWidth',2);
title('{\bf 1000 Simulated Response Paths for Analysis}')
legend([h1(1),h2],'Simulated Path','Mean')
hold off

figure
h1 = plot(notBurnIn,var(U,0,2),'r','LineWidth',2);
hold on
h2 = plot([notBurnIn(1) notBurnIn(end)],...
    [theoVarFix theoVarFix],'k--','LineWidth',2);
title('{\bf Converged Unconditional Disturbance Variance}')
legend([h1,h2],'Simulation Variance','Theoretical Variance')
hold off
```


1000 Simulated Response Paths for Analysis
Simulated Path
Mean
15
10
5
0
-5
-10
10
15
20
25
30
35
40
45
50



Unconditional disturbance simulation variances fluctuate around the theoretical variance due to Monte Carlo sampling error. Be aware that the exclusion of the burn-in sample from analysis reduces the effective sample size.

Simulate an MA Error Model

This example shows how to simulate responses from a regression model with MA errors without specifying a presample.

Specify the regression model with MA(8) errors:

$$y_t = 2 + X_t \begin{bmatrix} -2 \\ 1.5 \end{bmatrix} + u_t$$
$$u_t = \varepsilon_t + 0.4\varepsilon_{t-1} - 0.3\varepsilon_{t-4} + 0.2\varepsilon_{t-8},$$

where ε_t is Gaussian with mean 0 and variance 0.5.

```
Beta = [-2; 1.5];
Intercept = 2;
b1 = 0.4;
b4 = -0.3;
b8 = 0.2;
Variance = 0.5;
Mdl = regARIMA('MA',{b1, b4, b8},'MALags',[1 4 8],...
    'Intercept',Intercept,'Beta',Beta,'Variance',Variance);
```

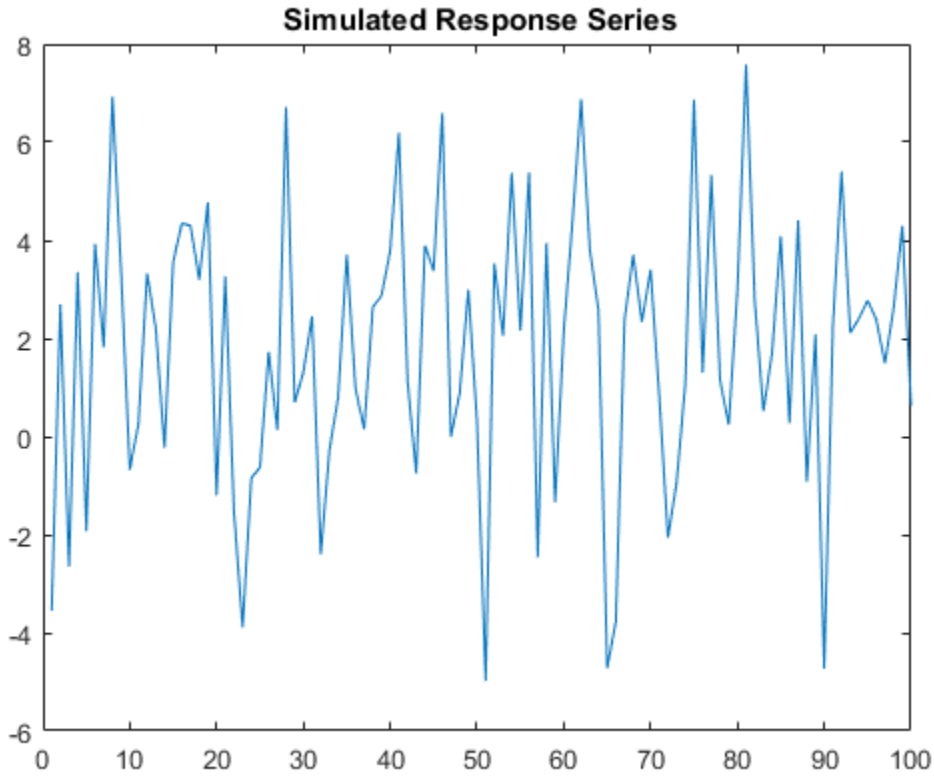
Generate two length $T = 100$ predictor series by random selection from the standard Gaussian distribution.

```
T = 100;
rng(4); % For reproducibility
X = randn(T,2);
```

The software treats the predictors as nonstochastic series.

Generate and plot one sample path of responses from `Mdl`.

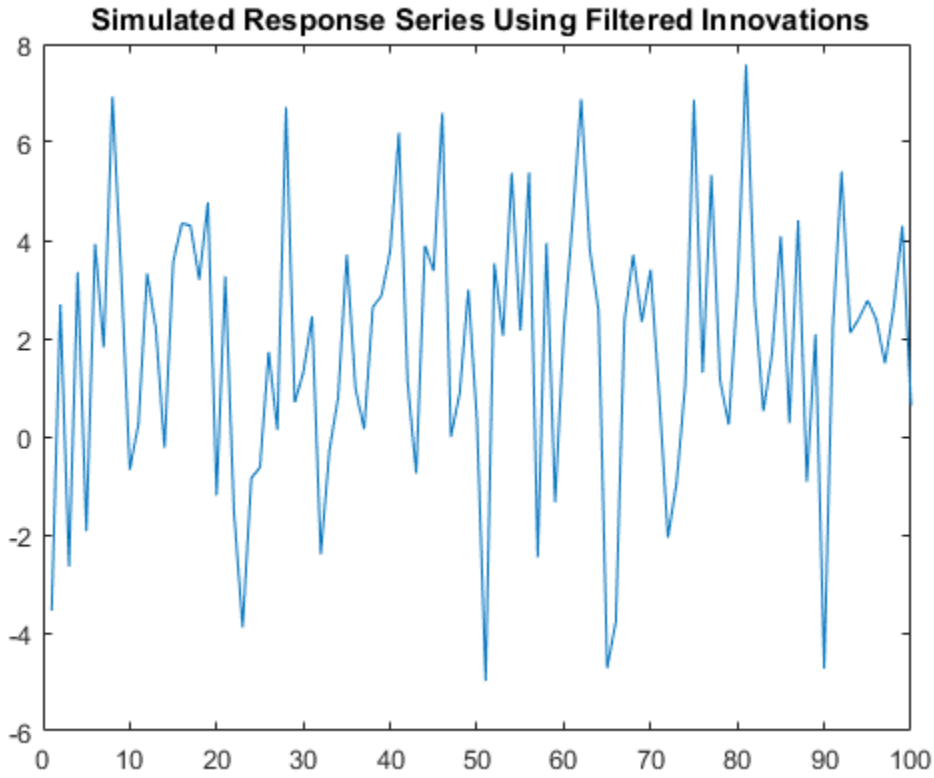
```
rng(5);
ySim = simulate(Mdl,T,'X',X);

figure
plot(ySim)
title('{\bf Simulated Response Series}')
```



`simulate` requires `Q = 8` presample innovations (ε_t) to initialize the error series. Without them, as in this case, `simulate` sets the necessary presample innovations to 0.

Alternatively, use `filter` to filter a random innovation series through `Mdl`.

```
rng(5);
e = randn(T,1);
yFilter = filter(Mdl,e,'X',X);

figure
plot(yFilter)
title('{\bf Simulated Response Series Using Filtered Innovations}')
```

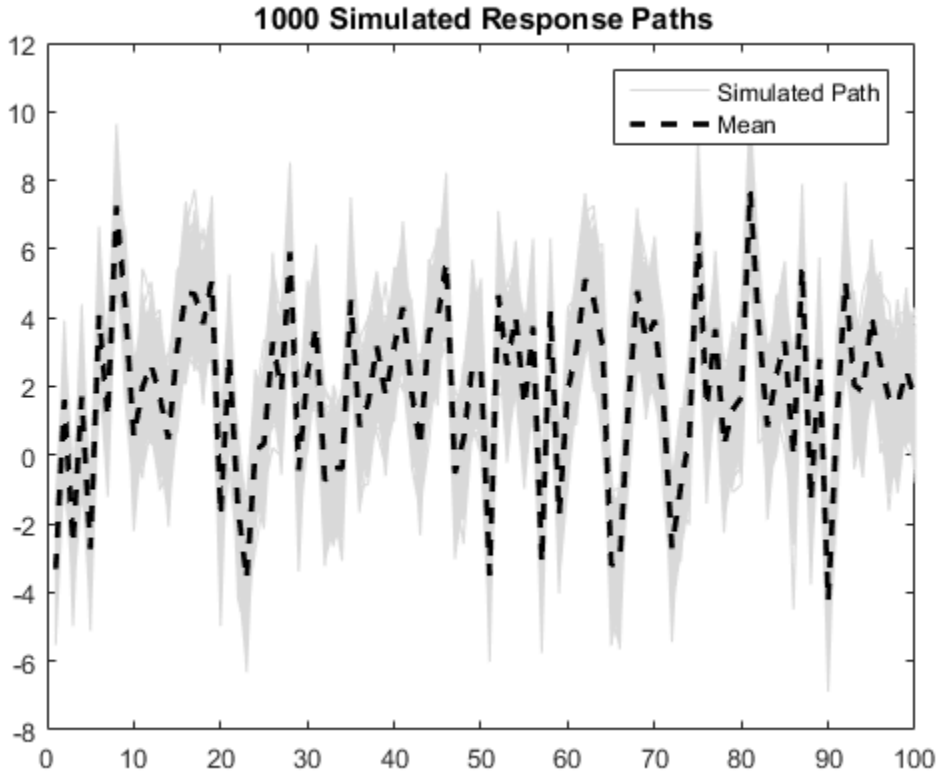


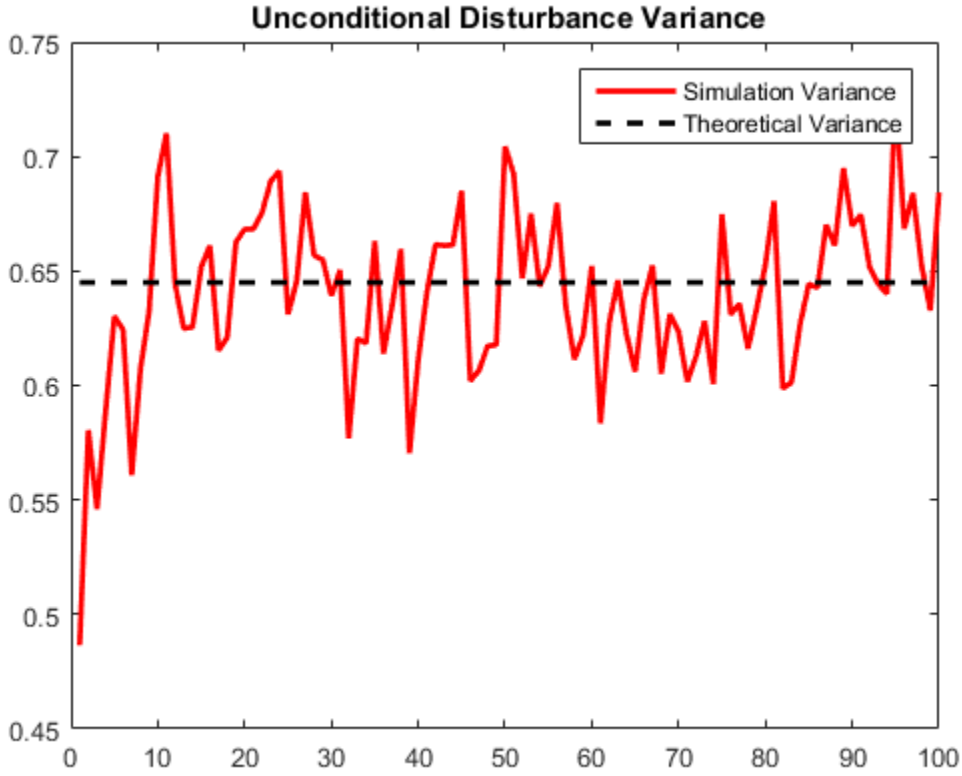
The plots suggest that the simulated responses and the responses generated from the filtered innovations are equivalent.

Simulate 1000 response paths from `Mdl`. Assess transient effects by plotting the unconditional disturbance (`U`) variances across the simulated paths at each period.

```
numPaths = 1000;
[Y,~,U] = simulate(Mdl,T,'NumPaths',numPaths,'X',X);

figure
h1 = plot(Y,'Color',[.85,.85,.85]);
title('{\bf 1000 Simulated Response Paths}')
hold on
h2 = plot(1:T,Intercept+X*Beta,'k--','LineWidth',2);
```

```
legend([h1(1),h2],'Simulated Path','Mean')
hold off

figure
h1 = plot(var(U,0,2),'r','LineWidth',2);
hold on
theoVarFix = (1+b1^2+b4^2+b8^2)*Variance;
h2 = plot([1 T],[theoVarFix theoVarFix],'k--','LineWidth',2);
title('{\bf Unconditional Disturbance Variance}')
legend([h1,h2],'Simulation Variance','Theoretical Variance')
hold off
```





The simulated paths follow their theoretical mean, $c + X\beta$, which is not constant over time (and might look nonstationary).

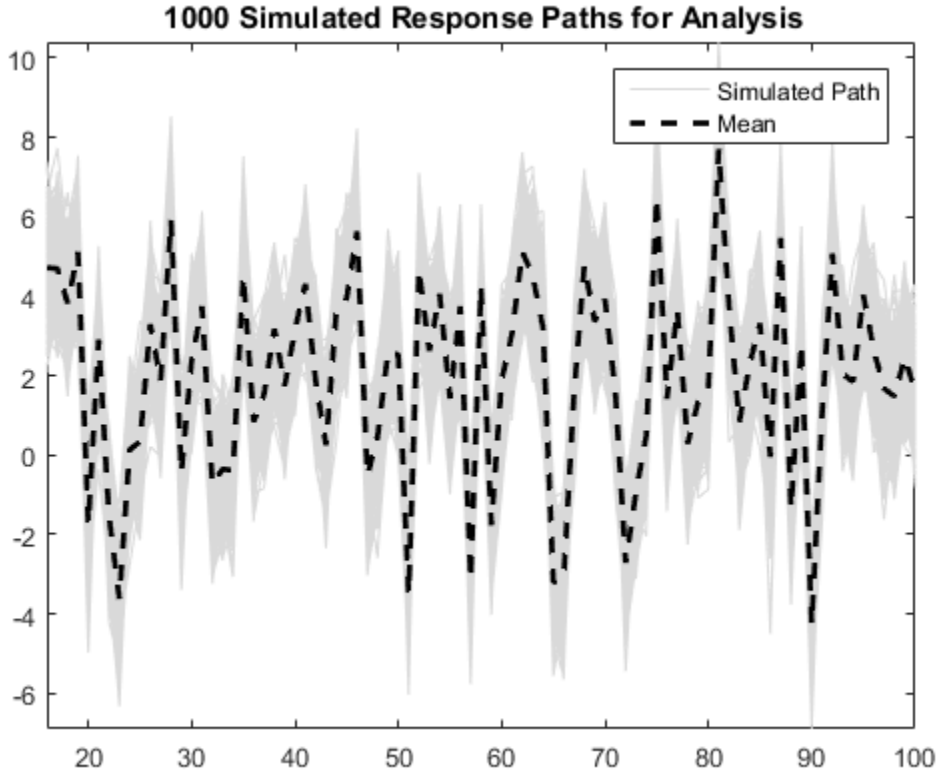
The variance of the process is not constant, but levels off at the theoretical variance by the 15th period. The theoretical variance of the MA(8) error model is

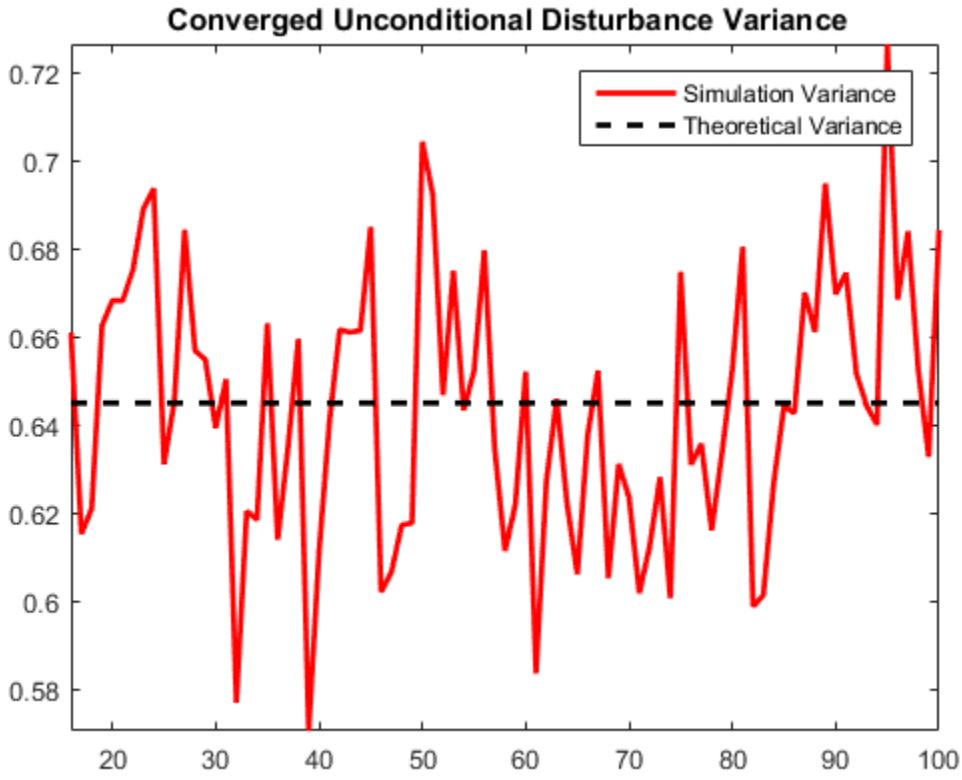
$$(1 + b_1^2 + b_4^2 + b_8^2)\sigma_\varepsilon^2 = \left(1 + 0.4^2 + (-0.3)^2 + 0.2^2\right)0.5 = 0.645.$$

You can reduce transient effects is by partitioning the simulated data into a burn-in portion and a portion for analysis. Do not use the burn-in portion for analysis. Include enough periods in the burn-in portion to overcome the transient effects.

```
burnIn = 1:15;
```

```
notBurnIn = burnIn(end)+1:T;
Y = Y(notBurnIn,:);
X = X(notBurnIn,:);
U = U(notBurnIn,:);
figure
h1 = plot(notBurnIn,Y,'Color',[.85,.85,.85]);
hold on
h2 = plot(notBurnIn,Intercept+X*Beta,'k--','LineWidth',2);
title('{\bf 1000 Simulated Response Paths for Analysis}')
legend([h1(1),h2],'Simulated Path','Mean')
axis tight
hold off

figure
h1 = plot(notBurnIn,var(U,0,2),'r','LineWidth',2);
hold on
h2 = plot([notBurnIn(1) notBurnIn(end)],...
    [theoVarFix theoVarFix],'k--','LineWidth',2);
title('{\bf Converged Unconditional Disturbance Variance}')
legend([h1,h2],'Simulation Variance','Theoretical Variance')
axis tight
hold off
```


1000 Simulated Response Paths for Analysis
Simulated Path
Mean
10
8
6
4
2
0
-2
-4
-6
20
30
40
50
60
70
80
90
100



Unconditional disturbance simulation variances fluctuate around the theoretical variance due to Monte Carlo sampling error. Be aware that the exclusion of the burn-in sample from analysis reduces the effective sample size.

Simulate an ARMA Error Model

This example shows how to simulate responses from a regression model with ARMA errors without specifying a presample.

Specify the regression model with ARMA(2,1) errors:

$$y_t = 2 + X_t \begin{bmatrix} -2 \\ 1.5 \end{bmatrix} + u_t$$
$$u_t = 0.9u_{t-1} - 0.1u_{t-2} + \varepsilon_t + 0.5\varepsilon_{t-1},$$

where ε_t is distributed with 15 degrees of freedom and variance 1.

```
Beta = [-2; 1.5];
Intercept = 2;
a1 = 0.9;
a2 = -0.1;
b1 = 0.5;
Variance = 1;
Distribution = struct('Name','t','DoF',15);
Mdl = regARIMA('AR',{a1, a2},'MA',b1,...
    'Distribution',Distribution,'Intercept',Intercept,...
    'Beta',Beta,'Variance',Variance);
```

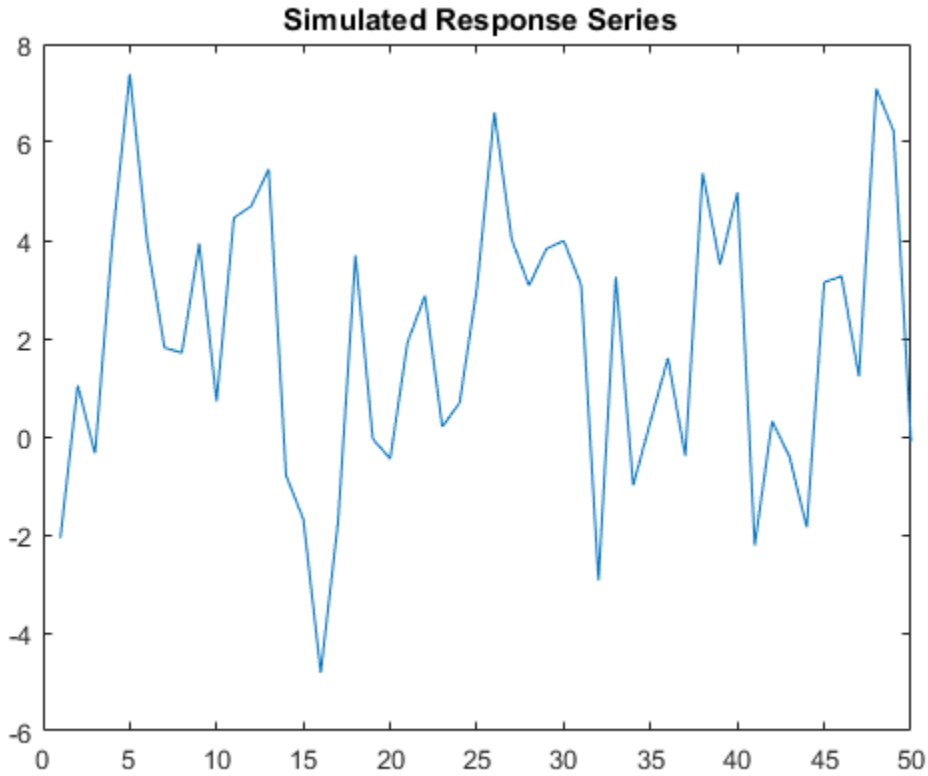
Generate two length $T = 50$ predictor series by random selection from the standard Gaussian distribution.

```
T = 50;
rng(6); % For reproducibility
X = randn(T,2);
```

The software treats the predictors as nonstochastic series.

Generate and plot one sample path of responses from `Mdl`.

```
rng(7);
ySim = simulate(Mdl,T,'X',X);

figure
plot(ySim)
title('{\bf Simulated Response Series}')
```



`simulate` requires:

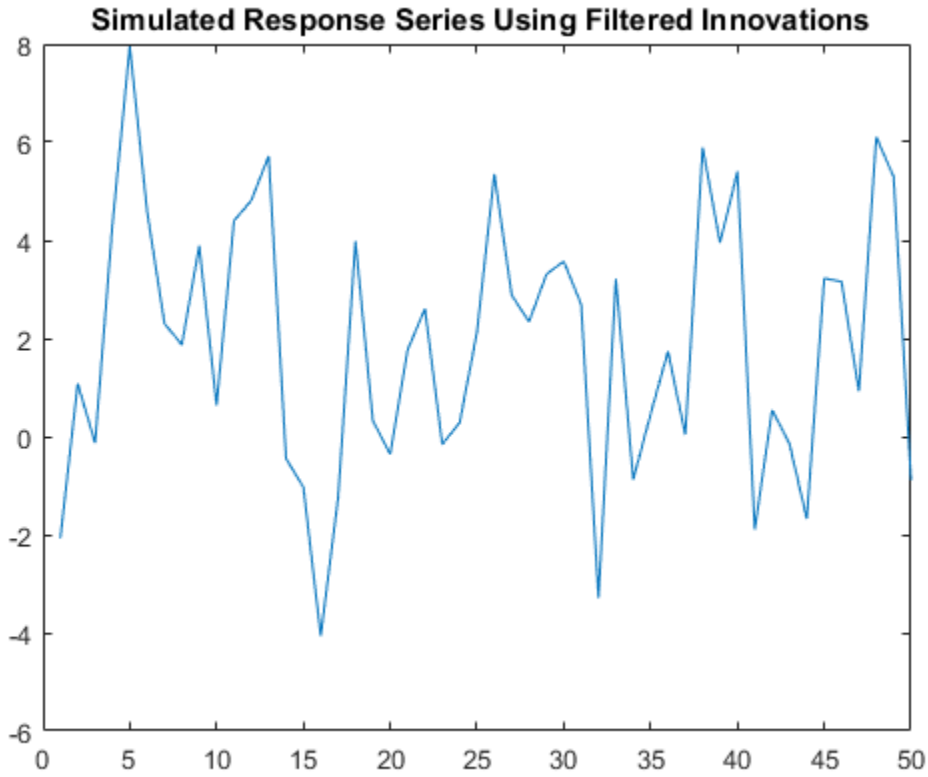
- `P = 2` presample unconditional disturbances to initialize the autoregressive component of the error series.
- `Q = 1` presample innovations to initialize the moving average component of the error series.

Without them, as in this case, `simulate` sets the necessary presample errors to 0.

Alternatively, use `filter` to filter a random innovation series through `Mdl`.

```
rng(7);
e = randn(T,1);
yFilter = filter(Mdl,e,'X',X);
```

```
figure
plot(yFilter)
title('{\bf Simulated Response Series Using Filtered Innovations}')
```

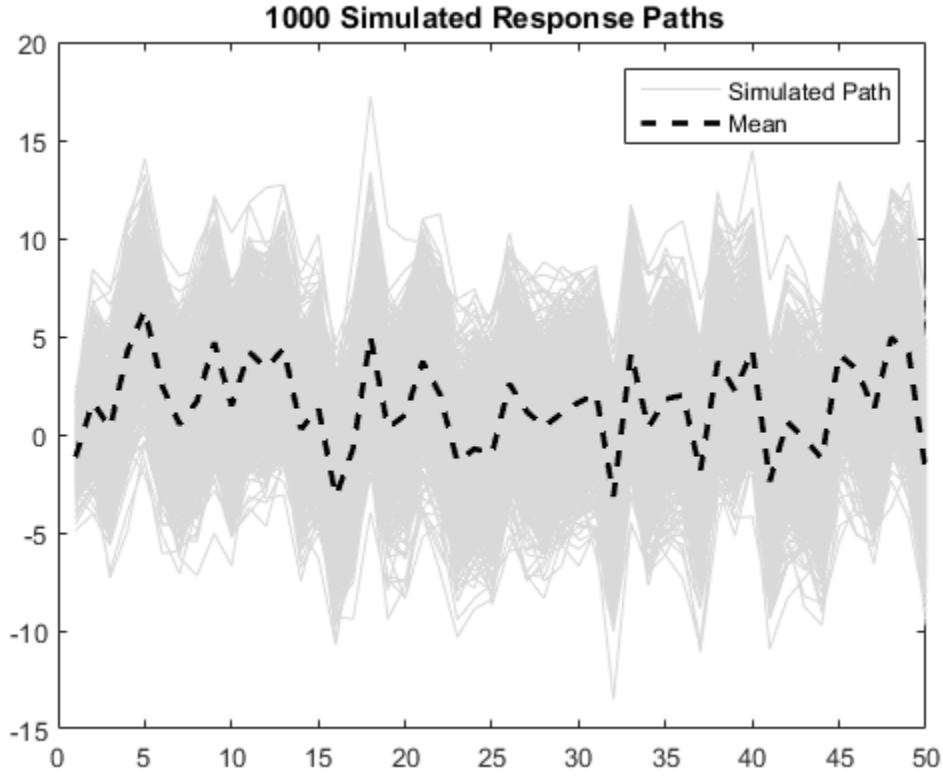


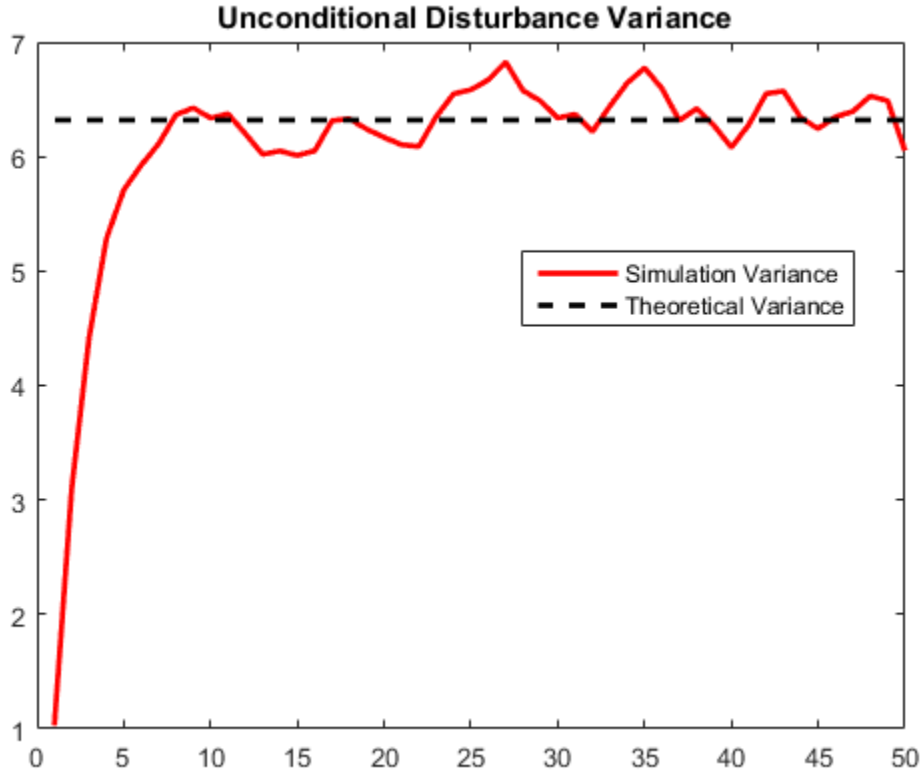
The plots suggest that the simulated responses and the responses generated from the filtered innovations are equivalent.

Simulate 1000 response paths from `Mdl`. Assess transient effects by plotting the unconditional disturbance (`U`) variances across the simulated paths at each period.

```
numPaths = 1000;
[Y,~,U] = simulate(Mdl,T,'NumPaths',numPaths,'X',X);
```

```
figure
h1 = plot(Y,'Color',[.85,.85,.85]);
title('{\bf 1000 Simulated Response Paths}')
hold on
h2 = plot(1:T,Intercept+X*Beta,'k--','LineWidth',2);
legend([h1(1),h2],'Simulated Path','Mean')
hold off

figure
h1 = plot(var(U,0,2),'r','LineWidth',2);
hold on
theoVarFix = Variance*(a1*b1*(1+a2)+(1-a2)*(1+a1*b1+b1^2))/...
    ((1+a2)*((1-a2)^2-a1^2));
h2 = plot([1 T],[theoVarFix theoVarFix],'k--','LineWidth',2);
title('{\bf Unconditional Disturbance Variance}')
legend([h1,h2],'Simulation Variance','Theoretical Variance',...
    'Location','Best')
hold off
```


1000 Simulated Response Paths
Simulated Path
Mean
20
15
10
5
0
-5
-10
-15
0
5
10
15
20
25
30
35
40
45
50



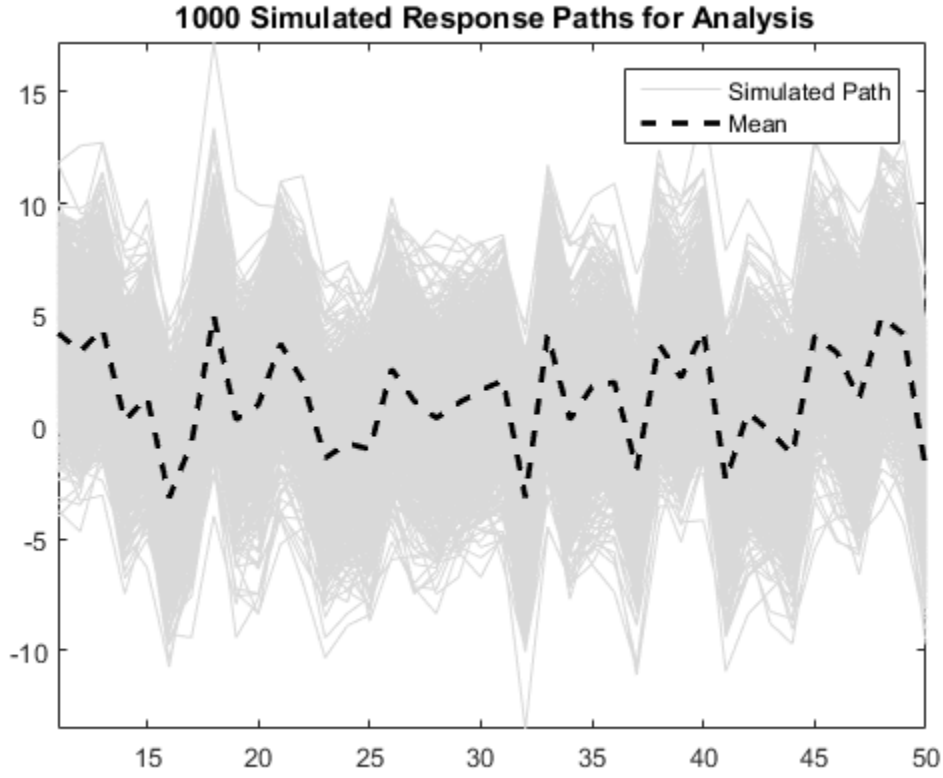
The simulated paths follow their theoretical mean, $c + X\beta$, which is not constant over time (and might look nonstationary).

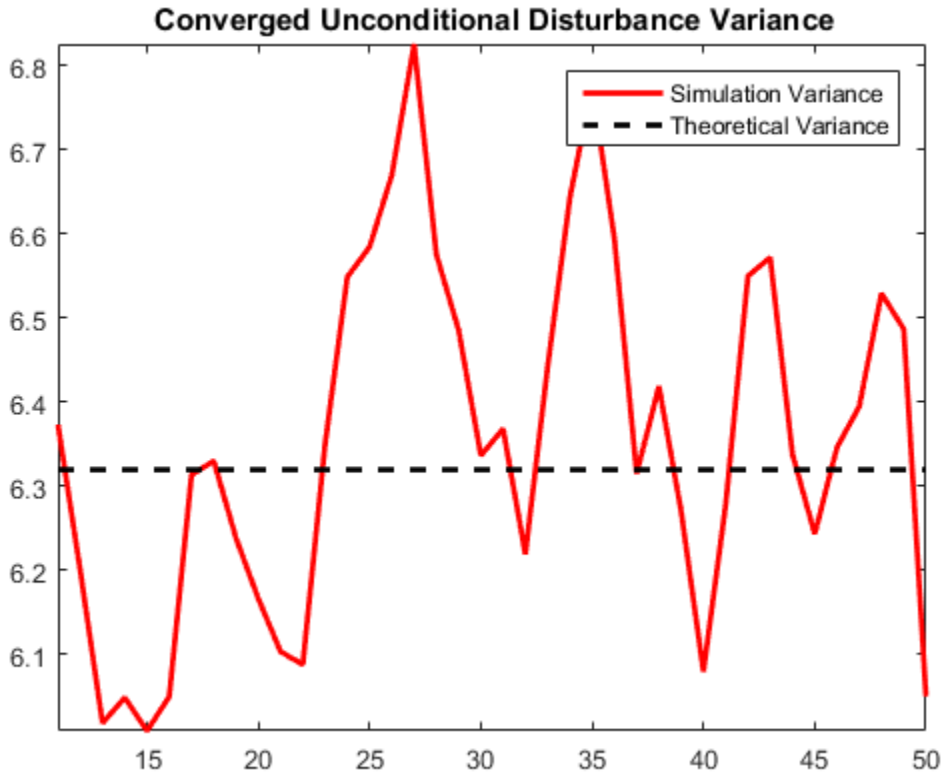
The variance of the process is not constant, but levels off at the theoretical variance by the 10th period. The theoretical variance of the ARMA(2,1) error model is:

$$\frac{\left[a_1 b_1 \left(1+a_2\right)+\left(1-a_2\right)\left(1+a_1 b_1+b_1^2\right)\right]}{\left(1+a_2\right)^2\left[\left(1-a_2\right)^2-a_1^2\right]}=\frac{\left[0.9(0.5)\left(1-0.1\right)+\left(1+0.1\right)\left(1+0.9(0.5)+0.5^2\right)\right]}{\left(1-0.1\right)^2\left[\left(1+0.1\right)^2-0.9^2\right]}=6.$$

You can reduce transient effects by partitioning the simulated data into a burn-in portion and a portion for analysis. Do not use the burn-in portion for analysis. Include enough periods in the burn-in portion to overcome the transient effects.

```
burnIn = 1:10;
notBurnIn = burnIn(end)+1:T;
Y = Y(notBurnIn,:);
X = X(notBurnIn,:);
U = U(notBurnIn,:);
figure
h1 = plot(notBurnIn,Y,'Color',[.85,.85,.85]);
hold on
h2 = plot(notBurnIn,Intercept+X*Beta,'k--','LineWidth',2);
title('{\bf 1000 Simulated Response Paths for Analysis}')
legend([h1(1),h2],'Simulated Path','Mean')
axis tight
hold off

figure
h1 = plot(notBurnIn,var(U,0,2),'r','LineWidth',2);
hold on
h2 = plot([notBurnIn(1) notBurnIn(end)],...
    [theoVarFix theoVarFix],'k--','LineWidth',2);
title('{\bf Converged Unconditional Disturbance Variance}')
legend([h1,h2],'Simulation Variance','Theoretical Variance')
axis tight
hold off
```


1000 Simulated Response Paths for Analysis
Simulated Path
Mean
15
10
5
0
-5
-10
15
20
25
30
35
40
45
50



Unconditional disturbance simulation variances fluctuate around the theoretical variance due to Monte Carlo sampling error. Be aware that the exclusion of the burn-in sample from analysis reduces the effective sample size.

Simulate Regression Models with Nonstationary Errors

In this section...

Simulate a Regression Model with Nonstationary Errors

This example shows how to simulate responses from a regression model with ARIMA unconditional disturbances, assuming that the predictors are white noise sequences.

Specify the regression model with ARIMA errors:

$$y_t = 3 + X_t \begin{bmatrix} 2 \\ -1.5 \end{bmatrix} + u_t$$

$$\Delta u_t = 0.5\Delta u_{t-1} + \varepsilon_t + 1.4\varepsilon_{t-1} + 0.8\varepsilon_{t-2},$$

where the innovations are Gaussian with variance 1.

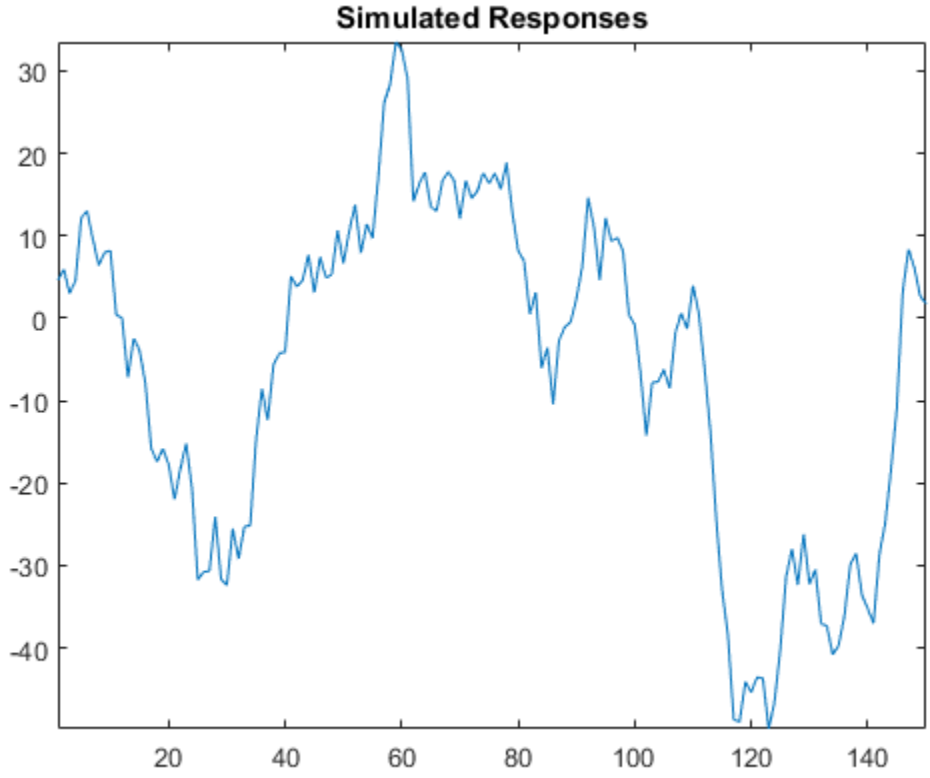
```
T = 150; % Sample size
Mdl = regARIMA('MA',{1.4,0.8},'AR',0.5,'Intercept',3,...
    'Variance',1,'Beta',[2;-1.5],'D',1);
```

Simulate two Gaussian predictor series with mean 0 and variance 1.

```
rng(1); % For reproducibility
X = randn(T,2);
```

Simulate and plot the response series.

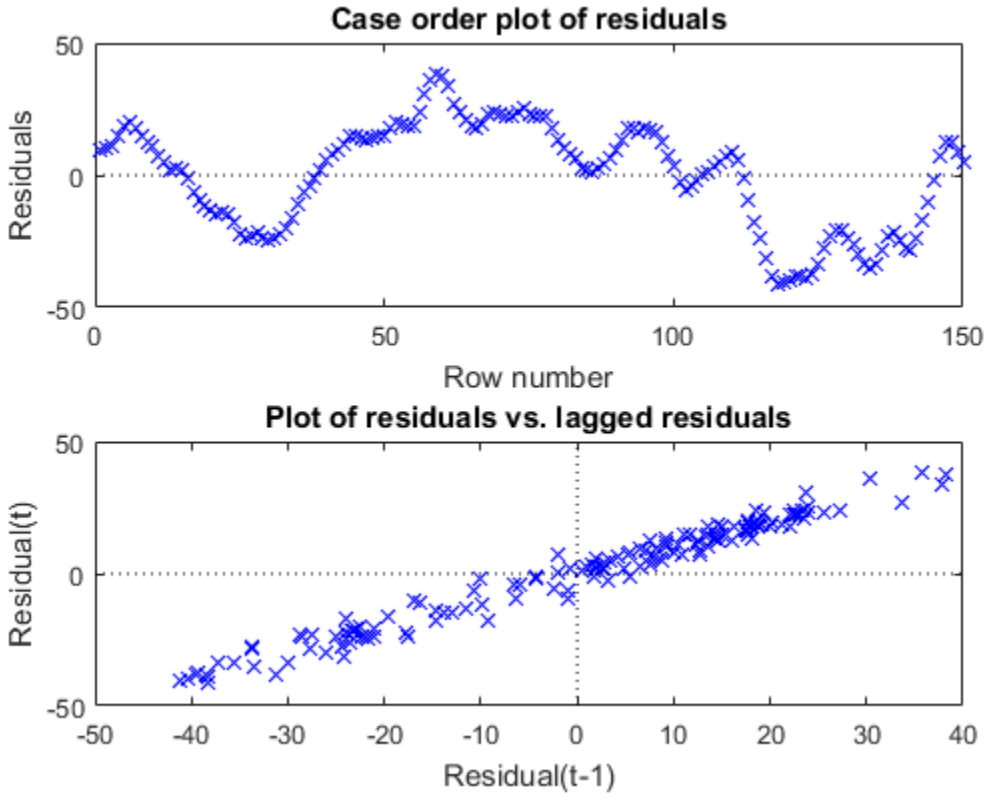
```
y = simulate(Mdl,T,'X',X);

figure;
plot(y);
title 'Simulated Responses';
axis tight;
```



Regress `y` onto `X`. Plot the residuals, and test them for a unit root.

```
RegMdl = fitlm(X,y);

figure;
subplot(2,1,1);
plotResiduals(RegMdl,'caseorder');
subplot(2,1,2);
plotResiduals(RegMdl,'lagged');

h = adftest(RegMdl.Residuals.Raw)


h =
```

0

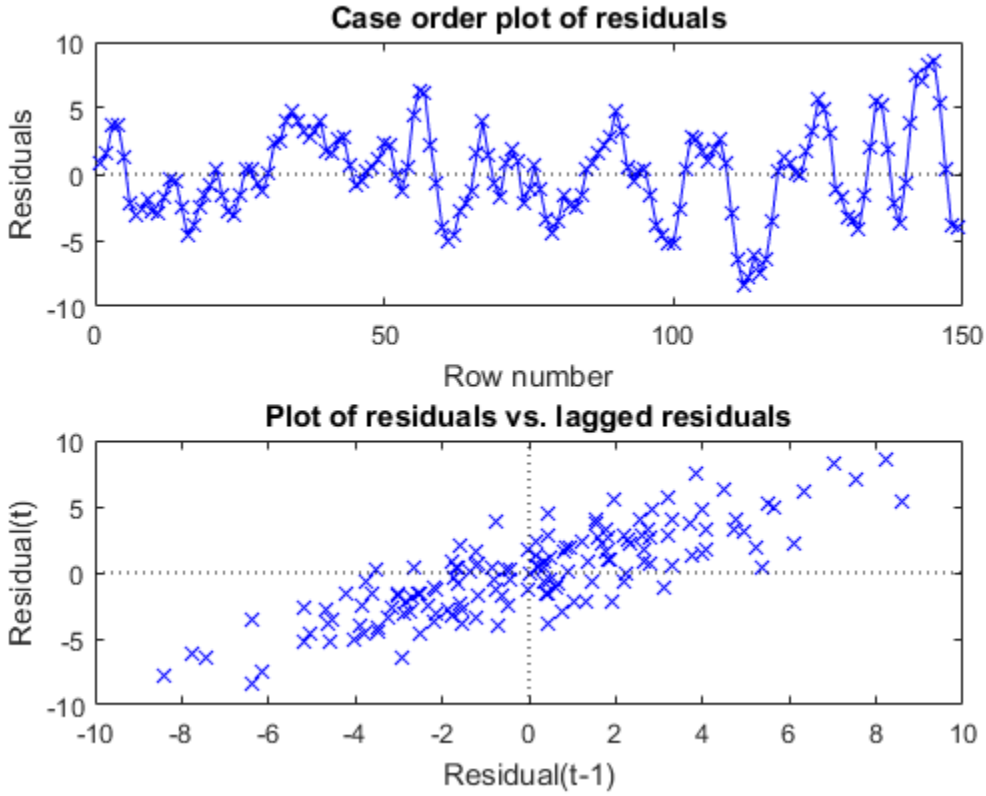


The residual plots indicate that they are autocorrelated and possibly nonstationary (as constructed). `h = 0` indicates that there is insufficient evidence to suggest that the residual series is not a unit root process.

Treat the nonstationary unconditional disturbances by transforming the data appropriately. In this case, difference the responses and predictors. Reestimate the regression model using the transformed responses, and plot the residuals.

```
dY = diff(y);
```

```
dX = diff(X);
dRegMdl = fitlm(dX,dY);

figure;
subplot(2,1,1);
plotResiduals(dRegMdl,'caseorder','LineStyle','-');
subplot(2,1,2);
plotResiduals(dRegMdl,'lagged');

h = adftest(dRegMdl.Residuals.Raw)


h =

     1
```



The residual plots indicate that they are still autocorrelated, but stationary. `h = 1` indicates that there is enough evidence to suggest that the residual series is not a unit root process.

Once the residuals appear stationary, you can determine the appropriate number of lags for the error model using Box and Jenkins methodology. Then, use `regARIMA` to completely model the regression model with ARIMA errors.

Simulate a Regression Model with Nonstationary Exponential Errors

This example shows how to simulate responses from a regression model with nonstationary, exponential, unconditional disturbances. Assume that the predictors are white noise sequences.

Specify the following ARIMA error model:

$$\Delta u_t = 0.9\Delta u_{t-1} + \varepsilon_t,$$

where the innovations are Gaussian with mean 0 and variance 0.05.

```
T = 50; % Sample size
MdlU = arima('AR',0.9,'Variance',0.05,'D',1,'Constant',0);
```

Simulate unconditional disturbances. Exponentiate the simulated errors.

```
rng(10); % For reproducibility
u = simulate(MdlU,T,'Y0',[0.5:1.5]');
expU = exp(u);
```

Simulate two Gaussian predictor series with mean 0 and variance 1.

```
X = randn(T,2);
```

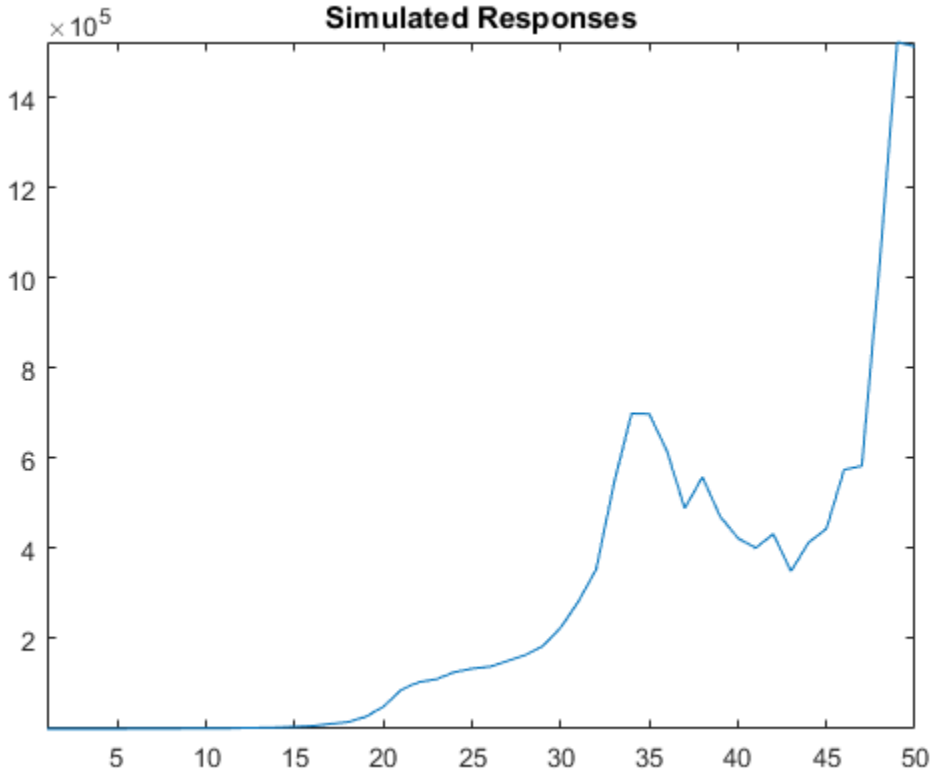
Generate responses from the regression model with time series errors:

$$y_t = 3 + X_t \begin{bmatrix} 2 \\ -1.5 \end{bmatrix} + e^{u_t}.$$

```
Beta = [2;-1.5];
Intercept = 3;
y = Intercept + X*Beta + expU;
```

Plot the responses.

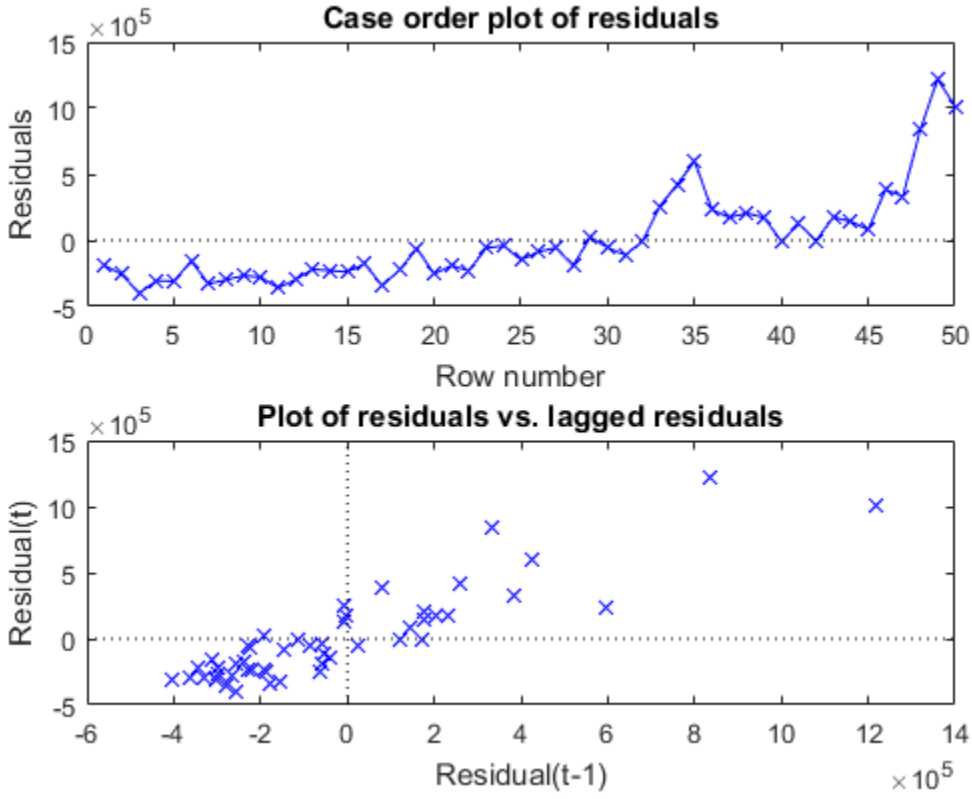
```
figure
plot(y)
title('Simulated Responses')
axis tight
```



The response series seems to grow exponentially (as constructed).

Regress `y` onto `X`. Plot the residuals.

```
RegMdl1 = fitlm(X,y);

figure
subplot(2,1,1)
plotResiduals(RegMdl1,'caseorder','LineStyle','-')
subplot(2,1,2)
plotResiduals(RegMdl1,'lagged')
```

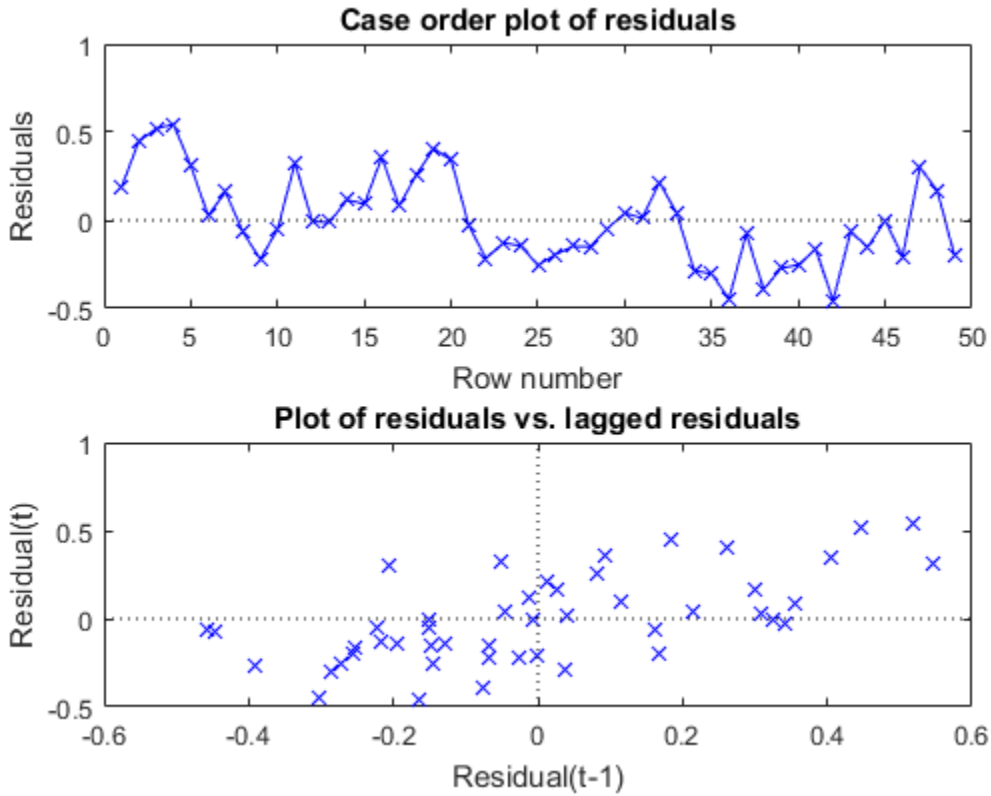


The residuals seem to grow exponentially, and seem autocorrelated (as constructed).

Treat the nonstationary unconditional disturbances by transforming the data appropriately. In this case, take the log of the response series. Difference the logged responses. It is recommended to transform the predictors the same way as the responses to maintain the original interpretation of their relationship. However, do not transform the predictors in this case because they contain negative values. Reestimate the regression model using the transformed responses, and plot the residuals.

```
dLogY = diff(log(y));
RegMdl2 = fitlm(X(2:end,:),dLogY);

figure
subplot(2,1,1)
```

```
plotResiduals(RegMdl2,'caseorder','LineStyle','-')
subplot(2,1,2)
plotResiduals(RegMdl2,'lagged')

h = adftest(RegMdl2.Residuals.Raw)


h =

     1
```



The residual plots indicate that they are still autocorrelated, but stationary. `h = 1` indicates that there is enough evidence to suggest that the residual series is not a unit root process.

Once the residuals appear stationary, you can determine the appropriate number of lags for the error model using Box and Jenkins methodology. Then, use `regARIMA` to completely model the regression model with ARIMA errors.

References

[1] Box, G. E. P., G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis: Forecasting and Control*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

See Also

regaARIMA

More About

- "Box-Jenkins Methodology" on page 3-2

Simulate Regression Models with Multiplicative Seasonal Errors

In this section...

Simulate a Regression Model with Stationary Multiplicative Seasonal Errors

This example shows how to simulate sample paths from a regression model with multiplicative seasonal ARIMA errors using `simulate`. The time series is monthly international airline passenger numbers from 1949 to 1960.

Load the airline and recessions data sets.

```
load(fullfile(matlabroot,'examples','econ','Data_Airline.mat'))
load Data_Recessions
```

Transform the airline data by applying the logarithm, and the 1st and 12th differences.

```
y = Data;
logY = log(y);
DiffPoly = LagOp([1 -1]);
SDiffPoly = LagOp([1 -1],'Lags',[0, 12]);
dLogY = filter(DiffPoly*SDiffPoly,logY);
```

Construct the predictor (`X`), which determines whether the country was in a recession during the sampled period. A 0 in row t means the country was not in a recession in month t, and a 1 in row t means that it was in a recession in month t.

```
X = zeros(numel(dates),1); % Preallocation
for j = 1:size(Recessions,1)
    X(dates >= Recessions(j,1) & dates <= Recessions(j,2)) = 1;
end
X = X(14:end); % Remove the first 14 observations for consistency
dates = dates(14:end);
```

Define index sets that partition the data into estimation and forecast samples.

```
nSim = 60; % Forecast period
```

```
T = length(dLogY);
estInds = 1:(T-nSim);
foreInds = (T-nSim+1):T;
```

Estimate the regression model with multiplicative seasonal errors:

$$y_t = c + X_t\beta + u_t$$

$$u_t = (1 + BL)(1 + B_{12}L^{12})\varepsilon_t.$$

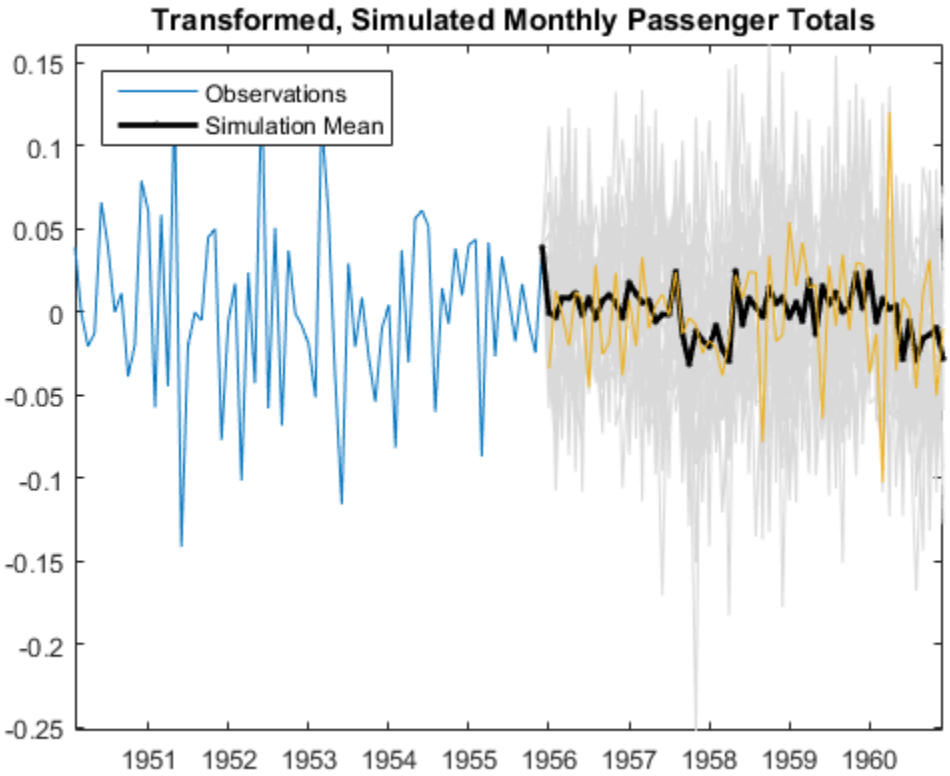
```
Mdl = regARIMA('MALags',1,'SMALags',12);
EstMdl = estimate(Mdl,dLogY(estInds),'X',X(estInds));
```

```
    Regression with ARIMA(0,0,1) Error Model with Seasonal MA(12):
    --------------------------------------------------------------
    Conditional Probability Distribution: Gaussian

                               Standard          t
     Parameter       Value       Error       Statistic
    -----------   -----------   ------------   -----------
    Intercept     0.00421457    0.00153333        2.74864
        MA{1}      -0.477125      0.120874       -3.94728
      SMA{12}      -0.741149      0.120416       -6.15492
        Beta1      -0.018912    0.00756648       -2.49945
     Variance     0.00166952   0.000311687        5.35639
```

Use the estimated coefficients of the model (contained in `EstMdl`) to simulate 25 realizations of airline passenger counts over the 60-month horizon. Infer the residuals, and use them as a presample.

```
[~,u0] = infer(EstMdl,dLogY(estInds),'X',X(estInds));
rng(5);
numPaths = 25;
dLogYSim = simulate(EstMdl,60,'numPaths',numPaths,'U0',u0,'X',X(foreInds));
meanDLogYSim = mean(dLogYSim,2);

figure
h1 = plot(dates(estInds),dLogY(estInds));
title('{\bf Transformed, Simulated Monthly Passenger Totals}')
hold on
plot(dates(foreInds),dLogYSim,'Color',[.85,.85,.85])
h2 = plot(dates(foreInds),meanDLogYSim,'k.-','LineWidth',2);
```

```
plot([dates(estInds(end)),dates(foreInds(1))],...
    [repmat(dLogY(estInds(end)),numPaths,1),dLogYSim(1,:)'],...
    'Color',[.85,.85,.85])
plot([dates(estInds(end)),dates(foreInds(1))],...
    [dLogY(estInds(end)),meanDLogYSim(1)],'k.-','LineWidth',2)
plot(dates(foreInds),dLogY(foreInds))
datetick
legend([h1,h2],'Observations','Simulation Mean','Location','NorthWest')
axis tight
hold off
```



The regression model with SMA errors seems to forecast the series well.

Check the predictive performance of the model by:

1. Varying the size of the forecast period
2. Estimating the prediction mean square error (PMSE)
3. Choosing the model with the lowest PMSE

Simulate a Regression Model with Nonstationary Multiplicative Seasonal Errors

This example shows how to simulate sample paths from a regression model with multiplicative seasonal ARIMA errors using `simulate`. The time series is monthly international airline passenger numbers from 1949 to 1960.

Load the airline and recessions data sets. Transform the response.

```
load(fullfile(matlabroot,'examples','econ','Data_Airline.mat'))
load Data_Recessions
y = log(Data);
```

Construct the predictor (X), which determines whether the country was in a recession during the sampled period. A 0 in row t means the country was not in a recession in month t, and a 1 in row t means that it was in a recession in month t.

```
X = zeros(numel(dates),1); % Preallocation
for j = 1:size(Recessions,1)
    X(dates >= Recessions(j,1) & dates <= Recessions(j,2)) = 1;
end
```

Define index sets that partition the data into estimation and forecast samples.

```
nSim = 60; % Forecast period
T = length(y);
estInds = 1:(T-nSim);
foreInds = (T-nSim+1):T;
```

Estimate the regression model with multiplicative seasonal errors:

$$y_t = X_t\beta + u_t$$

$$(1 - L)(1 - L^{12})u_t = (1 + BL)(1 + B_{12}L^{12})\varepsilon_t.$$

Set the regression model intercept to 0 since it is not identifiable in an integrated model.

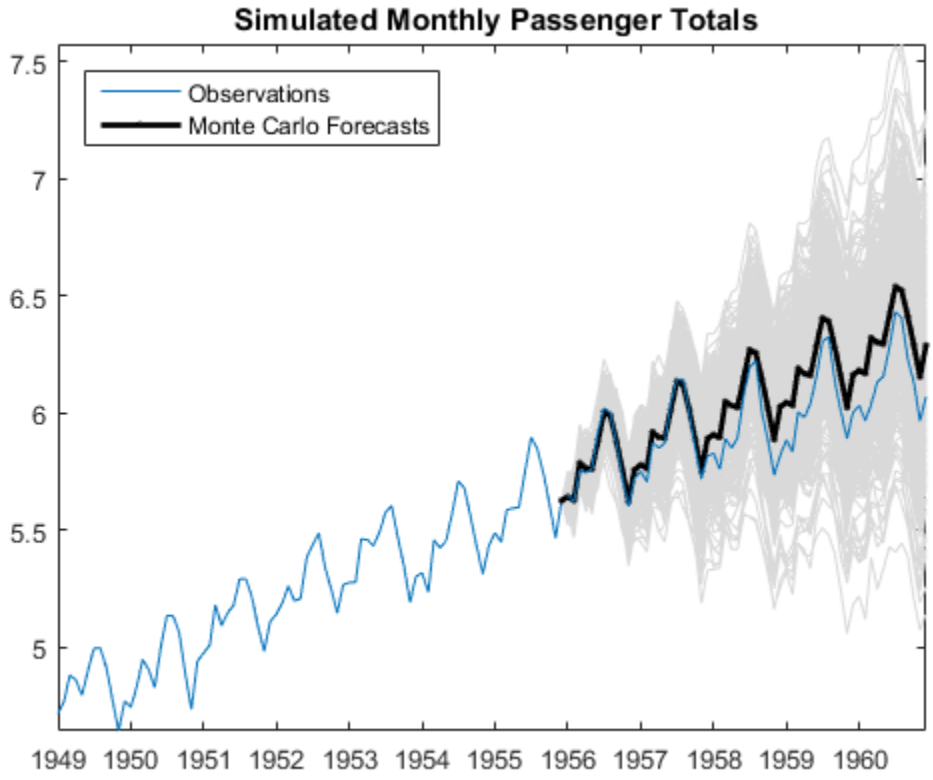
```
Mdl = regARIMA('D',1,'Seasonality',12,'MALags',1,'SMALags',12,...
```

```
    'Intercept',0);
EstMdl = estimate(Mdl,y(estInds),'X',X(estInds));
```

```
    Regression with ARIMA(0,1,1) Error Model Seasonally Integrated with Seasonal MA(12)
    ------------------------------------------------------------------------------------
    Conditional Probability Distribution: Gaussian

                                Standard          t
     Parameter       Value        Error       Statistic
    -----------   -----------   ------------   -----------
    Intercept               0         Fixed          Fixed
        MA{1}      -0.356617      0.103933       -3.43121
      SMA{12}      -0.677293      0.112935       -5.99718
        Beta1     0.00150984     0.0205331      0.0735321
     Variance     0.00151984   0.000214114        7.09828
```

Use the estimated coefficients of the model (contained in `EstMdl`), to simulate airline passenger counts over the 60-month horizon. Infer the residuals, and use them as a presample.

```
[e0,u0] = infer(EstMdl,y(estInds),'X',X(estInds));
rng(5);
numPaths = 500;
ySim = simulate(EstMdl,nSim,'numPaths',numPaths,'E0',e0,...
    'U0',u0,'X',X(foreInds));
meanYSim = mean(ySim,2);

figure
h1 = plot(dates(estInds),y(estInds));
title('{\bf Simulated Monthly Passenger Totals}')
hold on
plot(dates(foreInds),ySim,'Color',[.85,.85,.85])
h2 = plot(dates(foreInds),meanYSim,'k.-','LineWidth',2);
plot([dates(estInds(end)),dates(foreInds(1))],...
    [repmat(y(estInds(end)),numPaths,1),ySim(1,:)'],...
    'Color',[.85,.85,.85])
plot([dates(estInds(end)),dates(foreInds(1))],...
    [y(estInds(end)),meanYSim(1)],'k.-','LineWidth',2)
plot(dates(foreInds),y(foreInds))
datetick
legend([h1,h2],'Observations','Monte Carlo Forecasts',...
    'Location','NorthWest')
axis tight
hold off
```



The simulated forecasts show growth and seasonal periodicity similar to the observed series. The regression model with SMA errors seems to forecast the series well, albeit slightly overestimating.

Check the predictive performance of the model by:

1. Varying the size of the forecast period
2. Estimating the prediction mean square error (PMSE)
3. Choosing the model with the lowest PMSE

Monte Carlo Simulation of Regression Models with ARIMA Errors

In this section...

What Is Monte Carlo Simulation?

Monte Carlo simulation is the process of generating independent, random draws from a specified probabilistic model. When simulating time series models, one draw (or realization) is an entire sample path of specified length N, y_1, y_2,...,y_N. When you generate a large number of draws, say M, you generate M sample paths, each of length N.

Note: Some extensions of Monte Carlo simulation rely on generating dependent random draws, such as Markov Chain Monte Carlo (MCMC). The `simulate` function in Econometrics Toolbox generates independent realizations.

Some applications of Monte Carlo simulation are:

- Demonstrating theoretical results
- Forecasting future events
- Estimating the probability of future events

Generate Monte Carlo Sample Paths

The time series portion of the model specifies the dynamic evolution of the unconditional disturbance process over time through a conditional mean structure. To perform Monte Carlo simulation of regression models with ARIMA errors:

1. Specify presample innovations or unconditional disturbances (or use default presample data).
2. Generate an uncorrelated innovation series from a probability distribution.
3. Filter the innovations through the ARIMA error model to obtain the simulated unconditional disturbances.

4 Use the regression model, predictor data, and simulated unconditional disturbances to obtain the responses.

For example, consider simulating N responses from the regression model with ARMA(2,1) errors:

$$y_t = X_t\beta + u_t$$
$$u_t = \phi_1 u_{t-1} + \phi_2 u_{t-2} + \varepsilon_t + \theta_1\varepsilon_{t-1},$$

where ε_t is Gaussian with mean 0 and variance σ^2. Given presample unconditional disturbances (u_0 and u_{-1}) and innovations (ε_0), following these steps:

1 Generate N independent innovations from the Gaussian distribution:

$\{\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_N\}$.

2 Filter the innovations recursively to obtain the unconditional disturbances:

a $\hat{u}_1 = \phi_1 u_0 + \phi_2 u_{-1} + \hat{\varepsilon}_1 + \varepsilon_0$

b $\hat{u}_2 = \phi_1 \hat{u}_1 + \phi_2 u_0 + \hat{\varepsilon}_2 + \hat{\varepsilon}_1$

c $\hat{u}_3 = \phi_1 \hat{u}_2 + \phi_2 \hat{u}_1 + \hat{\varepsilon}_3 + \hat{\varepsilon}_2$

d ...

e $\hat{u}_N = \phi_1 \hat{u}_{N-1} + \phi_2 \hat{u}_{N-2} + \hat{\varepsilon}_N + \hat{\varepsilon}_{N-1}$.

3 Obtain simulated responses using the unconditional disturbances, regression model, and the predictors:

$\hat{y}_t = X_t\beta + \hat{u}_t$.

Econometrics Toolbox automates this process with simulate. Pass in a fully specified regression model with ARIMA errors (regARIMA), the number of responses to simulate, and, optionally, the number of paths and presample data, and `simulate` simulates the responses.

Note: Econometrics Toolbox treats the predictors in the regression model as fixed, nonstochastic series. Therefore, in order to generate Monte Carlo sample paths of the response, you need to know the values of the predictors.

Monte Carlo Error

Using many simulated paths, you can estimate various features of the model. However, Monte Carlo estimation is based on a finite number of simulations. Therefore, Monte Carlo estimates are subject to some amount of error. You can reduce the amount of Monte Carlo error in your simulation study by increasing the number of sample paths, M, that you generate from your model.

For example, to estimate the probability of a future event:

1. Generate M sample paths from your model.
2. Estimate the probability of the future event using the sample proportion of the event occurrence across M simulations,

$$\hat{p} = \frac{\#\ times\ event\ occurs\ in\ M\ draws}{M}.$$

3. Calculate the Monte Carlo standard error for the estimate,

$$se = \sqrt{\frac{\hat{p}(1-\hat{p})}{M}}.$$

You can reduce the Monte Carlo error of the probability estimate by increasing the number of realizations. If you know the desired precision of your estimate, you can solve for the number of realizations needed to achieve that level of precision.

See Also

regARIMA | simulate

Related Examples

- "Simulate Regression Models with ARMA Errors" on page 4-145
- "Simulate Regression Models with Nonstationary Errors" on page 4-171
- "Simulate Regression Models with Multiplicative Seasonal Errors" on page 4-181

More About

- "Presample Data for regARIMA Model Simulation" on page 4-191
- "Transient Effects in regARIMA Model Simulations" on page 4-192

- “Regression Models with Time Series Errors” on page 4-6

Presample Data for regARIMA Model Simulation

When simulating realizations from a regression model with ARIMA errors, the software requires presample unconditional disturbances and innovations to initialize the error process. The regARIMA model property `P` stores the number of presample unconditional disturbances that you need to initialize the simulation. The property `Q` stores the number of presample innovations that you need to initialize the simulation.

You can specify your own presample data, or let simulate generate presample data. If you let `simulate` generate default presample data, then `simulate` sets the required number of presample unconditional disturbances and presample innovations to 0.

`simulate` only accepts presample data for the error process, even if the response and predictors are time series.

See Also

regARIMA | simulate

Related Examples

- "Simulate Regression Models with ARMA Errors" on page 4-145
- "Simulate Regression Models with Nonstationary Errors" on page 4-171
- "Simulate Regression Models with Multiplicative Seasonal Errors" on page 4-181

More About

- "Monte Carlo Simulation of Regression Models with ARIMA Errors" on page 4-187
- "Transient Effects in regARIMA Model Simulations" on page 4-192
- "Regression Models with Time Series Errors" on page 4-6

Transient Effects in regARIMA Model Simulations

In this section...

What Are Transient Effects?

When you use automatically generated presample data, you often see transient effects at the beginning of the simulation. This is sometimes called a *burn-in period*. For stationary error processes, the impulse response function decays to zero over time. This means the starting point of the error simulation is eventually forgotten. To reduce transient effects, you can:

- *Oversample*: generate sample paths that are longer than needed, and discard the beginning samples that show transient effects.
- *Recycle*: use a first simulation to generate presample data for a second simulation.

If the model exhibits nonstationary errors, then the error process does not forget its starting point. By default, all realizations of nonstationary processes begin at zero. For a nonzero starting point, you need to specify your own presample data.

Illustration of Transient Effects on Regression

Transient effects in regression models with ARIMA errors can affect the regression coefficient estimates. The following examples illustrate the behavior of the regression line in models that ignore transient effects and models that account for them.

Transient Effects Are Randomly Spread

This example examines regression lines of regression models with ARMA errors when the transient effects are randomly spread with respect to the joint distribution of the predictor and response.

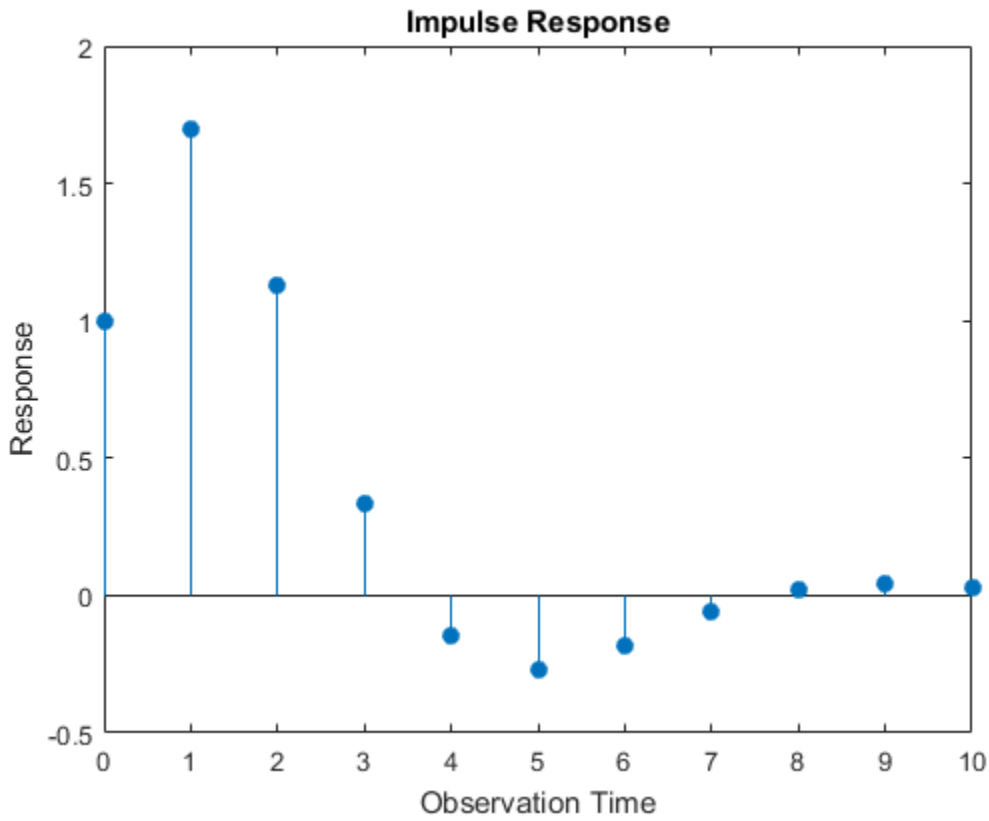
Specify the regression model with ARMA(2,1) errors:

$$y_t = 3 + 2X_t + u_t$$

$$u_t = 0.9u_{t-2} - 0.4u_{t-2} + \varepsilon_t + 0.8\varepsilon_{t-1},$$

where ε_t is Gaussian with mean 0 and variance 1. Plot the impulse response function.

```
Mdl = regARIMA('AR',{0.9,-0.4},'MA',{0.8},'Beta',2,...
    'Variance',1,'Intercept',3);

figure
impulse(Mdl)
```



The unconditional disturbances seem to settle after the 10th lag. Therefore, the transient effects end at the 10th lag.

Simulate a univariate, Gaussian predictor series with mean 0 and variance 1. Simulate 100 paths from `Mdl`.

```
rng(5);         % For reproducibility
T = 50;         % Sample size
numPaths = 100; % Number of paths

X = randn(T,1);                                % Full predictor series
Y = simulate(Mdl,T,'numPaths',numPaths,'X',X); % Full response series

endTrans = 10;
truncX = X((endTrans+1):end);   % Predictor without transient effects
truncY = Y((endTrans+1):end,:); % Response without transient effects
```

Fit the model to each simulated response path separately for the full and truncated series.

```
ToEstMdl = regARIMA(2,0,1); % Empty model for estimation
beta1 = zeros(2,numPaths);
beta2 = beta1;

for i = 1:numPaths
    EstMdl1 = estimate(ToEstMdl,Y(:,i),'X',X,'display','off');
    EstMdl2 = estimate(ToEstMdl,truncY(:,i),'X',truncX,'display','off');
    beta1(:,i) = [EstMdl1.Intercept; EstMdl1.Beta];
    beta2(:,i) = [EstMdl2.Intercept; EstMdl2.Beta];
end
```

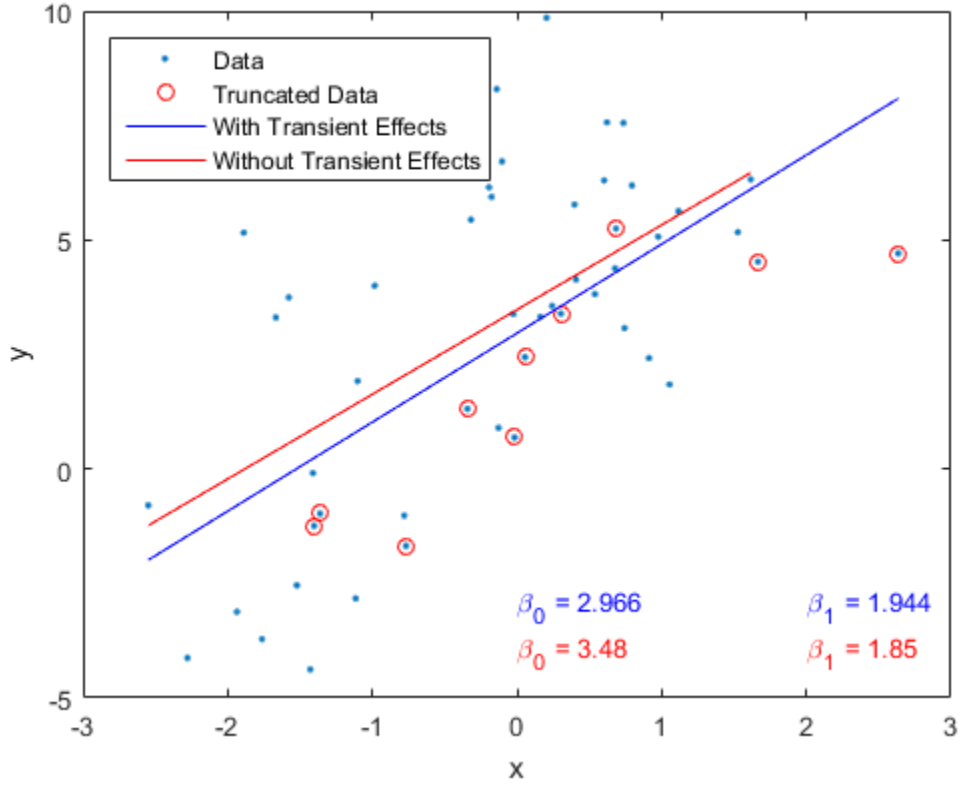
`beta1` is a 2-by- `numPaths` matrix containing the estimated intercepts and slopes for each simulated data set. `beta2` is a 2-by- `numPaths` matrix containing the estimated intercepts and slopes for the truncated, simulated data sets.

Compare the simulated regression lines between the full and truncated series. For one of the paths, plot the simulated data and its corresponding regression lines.

```
betaBar1 = mean(beta1,2);
betaBar2 = mean(beta2,2);
fprintf('Transient Effects  | Sim. Mean of Intercept    | Sim. Mean of Slope\n')
fprintf('=================================================================\n')
fprintf('Include            | %0.6g                   | %0.6g\n',betaBar1(1),betaBar1(2
```

```
fprintf('Without            | %0.6g                    | %0.6g\n',betaBar2(1),betaBar2(2
figure
plot(X,Y(:,1),'.')
hold on
plot(X(1:endTrans),Y(1:endTrans),'ro')
plot([min(X) max(X)],beta1(1,1) + beta1(2,1)*[min(X) max(X)],'b')
plot([min(truncX) max(truncX)],...
    beta2(1,1) + beta2(2,1)*[min(truncX) max(truncX)],'r')
legend('Data','Truncated Data','With Transient Effects',...
    'Without Transient Effects','Location','NorthWest')
xlabel('x')
ylabel('y')
text(0,-3,sprintf('\\beta_0 = %0.4g',beta1(1,1)),'Color',[0,0,1])
text(0,-4,sprintf('\\beta_0 = %0.4g',beta2(1,1)),'Color',[1,0,0])
text(2,-3,sprintf('\\beta_1 = %0.4g',beta1(2,1)),'Color',[0,0,1])
text(2,-4,sprintf('\\beta_1 = %0.4g',beta2(2,1)),'Color',[1,0,0])
hold off
```

```
Transient Effects  | Sim. Mean of Intercept    | Sim. Mean of Slope
=================================================================
Include            | 3.08619                   | 2.00098
Without            | 3.16408                   | 1.99455
```



The table in the Command Window displays the simulation averages of the intercept and slope of the regression model. The results suggest the regression line corresponding to the analysis including the full data set is parallel to the regression line corresponding to the truncated data set. In other words, the slope is mostly unaffected by accounting for transient effects, but the intercept is slightly affected.

Transient Effects Begin the Series

This example examines regression lines of regression models with ARMA errors when the transient effects occur at the beginning of each series.

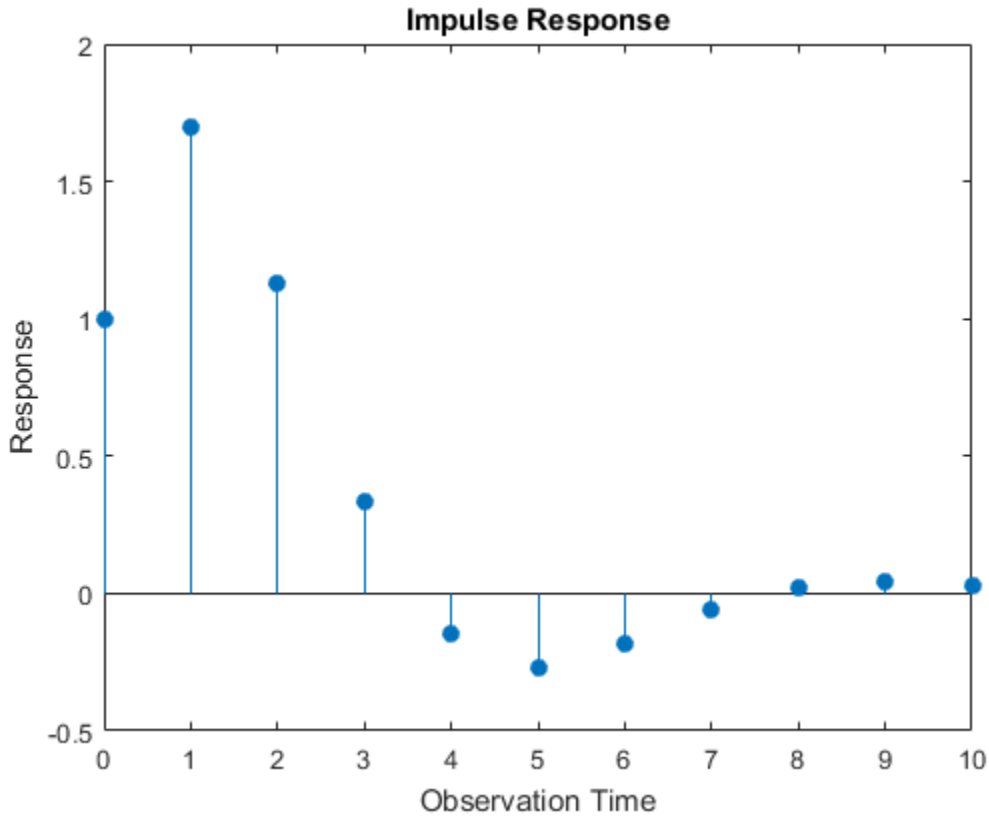
Specify the regression model with ARMA(2,1) errors:

$$y_t = 3 + 2X_t + u_t$$

$$u_t = 0.9u_{t-2} - 0.4u_{t-2} + \varepsilon_t + 0.8\varepsilon_{t-1},$$

where ε_t is Gaussian with mean 0 and variance 1. Plot the impulse response function.

```
Mdl = regARIMA('AR',{0.9,-0.4},'MA',{0.8},'Beta',2,...
    'Variance',1,'Intercept',3);

figure
impulse(Mdl)
```



The unconditional disturbances seem to settle at the 10th lag. Therefore, the transient effects end after the 10th lag.

Simulate a univariate, Gaussian predictor series with mean 0 and variance 1. Simulate 100 paths from `Mdl`. Truncate the response and predictor data sets to remove the transient effects.

```
rng(5);         % For reproducibility
T = 50;         % Sample size
numPaths = 100; % Number of paths

X = linspace(-3,3,T)' + randn(T,1)*0.1;        % Full predictor series
Y = simulate(Mdl,T,'numPaths',numPaths,'X',X); % Full response series

endTrans = 10;
truncX = X((endTrans+1):end);   % Predictor without transient effects
truncY = Y((endTrans+1):end,:); % Response without transient effects
```

Fit the model to each simulated response path separately for the full and truncated series.

```
ToEstMdl = regARIMA(2,0,1); % Empty model for estimation
beta1 = zeros(2,numPaths);
beta2 = beta1;

for i = 1:numPaths
    EstMdl1 = estimate(ToEstMdl,Y(:,i),'X',X,'display','off');
    EstMdl2 = estimate(ToEstMdl,truncY(:,i),'X',truncX,'display','off');
    beta1(:,i) = [EstMdl1.Intercept; EstMdl1.Beta];
    beta2(:,i) = [EstMdl2.Intercept; EstMdl2.Beta];
end
```

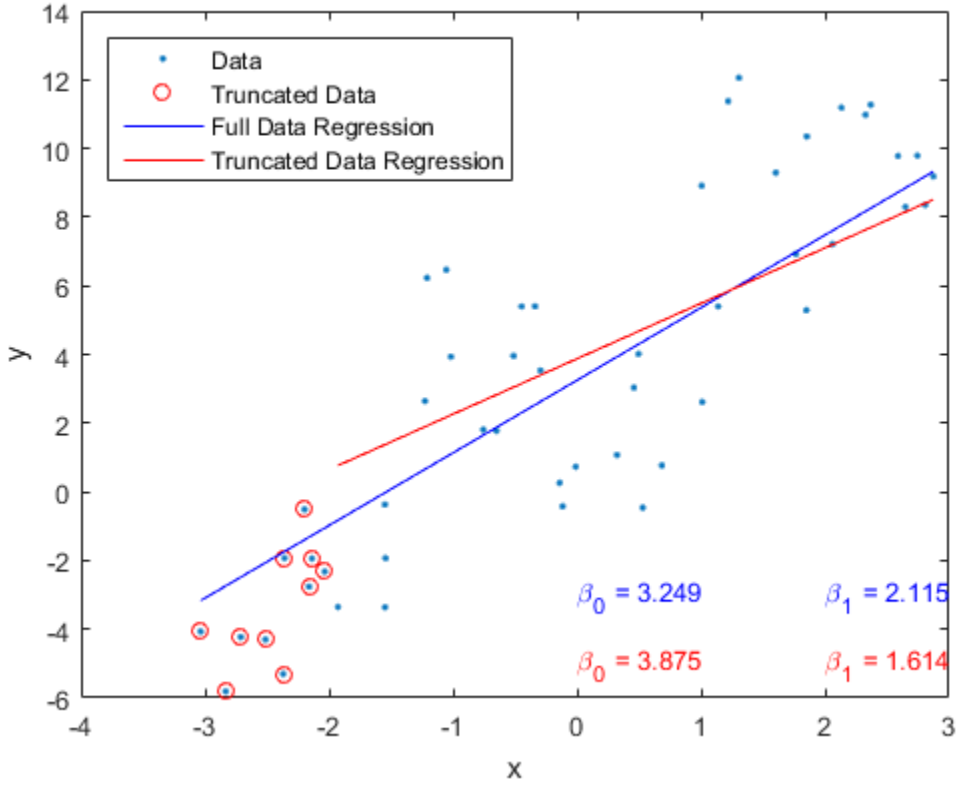
`beta1` is a 2-by- `numPaths` matrix containing the estimated intercepts and slopes for each simulated data set. `beta2` is a 2-by- `numPaths` matrix containing the estimated intercepts and slopes for the truncated, simulated data sets.

Compare the simulated regression lines between the full and truncated series. For one of the paths, plot the simulated data and its corresponding regression lines.

```
betaBar1 = mean(beta1,2);
betaBar2 = mean(beta2,2);
fprintf('Data      | Sim. Mean of Intercept    | Sim. Mean of Slope\n')
fprintf('=========================================================\n')
fprintf('Full      | %0.6g                     | %0.6g\n',betaBar1(1),betaBar1(2))
fprintf('Truncated | %0.6g                     | %0.6g\n',betaBar2(1),betaBar2(2))
figure
plot(X,Y(:,1),'.')
```

```
hold on
plot(X(1:endTrans),Y(1:endTrans),'ro')
plot([min(X) max(X)],beta1(1,1) + beta1(2,1)*[min(X) max(X)],'b')
plot([min(truncX) max(truncX)],...
    beta2(1,1) + beta2(2,1)*[min(truncX) max(truncX)],'r')
xlabel('x')
ylabel('y')
legend('Data','Truncated Data','Full Data Regression',...
    'Truncated Data Regression','Location','NorthWest')
text(0,-3,sprintf('\\beta_0 = %0.4g',beta1(1,1)),'Color',[0,0,1])
text(0,-5,sprintf('\\beta_0 = %0.4g',beta2(1,1)),'Color',[1,0,0])
text(2,-3,sprintf('\\beta_1 = %0.4g',beta1(2,1)),'Color',[0,0,1])
text(2,-5,sprintf('\\beta_1 = %0.4g',beta2(2,1)),'Color',[1,0,0])
hold off
```

```
Data      | Sim. Mean of Intercept    | Sim. Mean of Slope
=================================================================
Full      | 3.09312                   | 2.01796
Truncated | 3.14734                   | 1.98798
```



The table in the Command Window displays the simulation averages of the intercept and slope of the regression model. The results suggest that, on average, the regression lines corresponding to the full data and truncated data have slightly different intercepts and slopes. In other words, transient effects slightly affect regression estimates.

The plot displays the data and regression lines for one simulated path. The transient effects seem to affect the results more severely.

See Also

regARIMA | simulate

Related Examples

- "Simulate Regression Models with ARMA Errors" on page 4-145
- "Simulate Regression Models with Nonstationary Errors" on page 4-171
- "Simulate Regression Models with Multiplicative Seasonal Errors" on page 4-181

More About

- "Monte Carlo Simulation of Regression Models with ARIMA Errors" on page 4-187
- "Regression Models with Time Series Errors" on page 4-6

Forecast a Regression Model with ARIMA Errors

This example shows how to forecast a regression model with ARIMA(3,1,2) errors using `forecast` and `simulate`.

Simulate two Gaussian predictor series with mean 2 and variance 1.

```
rng(1);
T = 50; % Sample size
X = randn(T,2) + 2;
```

Specify the regression model with ARIMA(3,1,2) errors:

$$y_t = 3 + X_t \begin{bmatrix} -2 \\ 1.5 \end{bmatrix} + u_t$$

$$(1 - 0.9L + 0.5L^2 - 0.2L^3)(1 - L)u_t = (1 + 0.75L - 0.15L^2)\varepsilon_t,$$

where ε_t is Gaussian with mean 0 and variance 2.

```
Mdl = regARIMA('Intercept',3,'Beta',[-2;1.5],'AR',{0.9,-0.5,0.2},...
    'D',1','MA',{0.75,-0.15},'Variance',2);
```

`Mdl` is a fully specified regression model with ARIMA(3,1,2) errors. Methods such as `simulate` and `forecast` require a fully specified model.

Simulate 30 observations from `Mdl`.

```
[y,e,u] = simulate(Mdl,30,'X',X(1:30,:));
```

`y` contains the simulated responses. `e` and `u` contain the corresponding simulated innovations and unconditional disturbances, respecitvely. It is best practice to provide `forecast` with presample innovations and unconditional disturbances if they are available.

Compute MMSE forecasts for `Mdl` 20 periods into the future using `forecast`. Compute the corresponding 95% forecast intervals.

```
[yF,yMSE] = forecast(Mdl,20,'X0',X(1:30,:),'U0',u,...
```

```
    'EO',e,'XF',X(31:T,:));
yFCI = [yF,yF] + 1.96*[-sqrt(yMSE),sqrt(yMSE)];
```

`yFCI` is a 20-by-2 matrix containing the 20 forecast intervals. The first column of `yFCI` contains the lower bounds for the forecast intervals, and the second column contains the upper bounds.

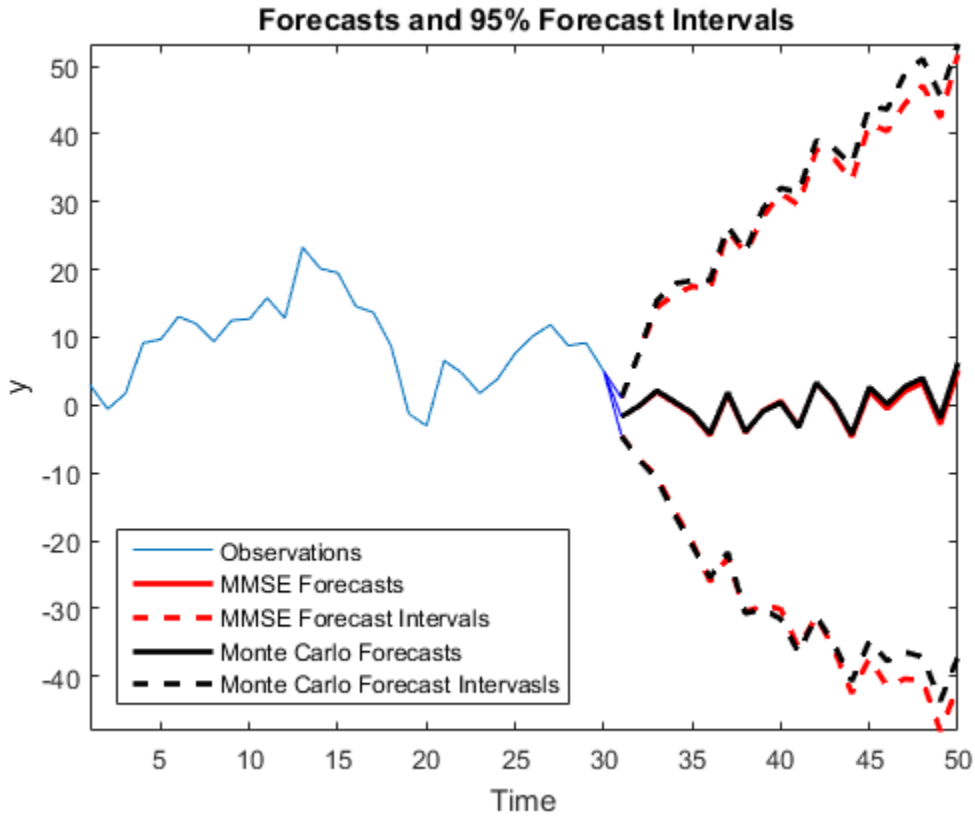
Forecast `Mdl` 20 periods into the future using Monte Carlo simulation. Compute the corresponding 95% forecast intervals

```
yMC = simulate(Mdl,20,'numPaths',1000,'X',X(31:T,:),'UO',u,'EO',e);
yMCBar = mean(yMC,2);
yMCCI = prctile(yMC,[2.5,97.5],2);
```

`yMCBar` is a 20-by-1 vector that contains the Monte Carlo forecasts over the forecast horizon. Like `yFCI`, `yMMCI` is a 20-by-2 matrix containing the forecast intervals, but based on the Monte Carlo simulation.

Plot the two forecast sets and their corresponding 95% forecast intervals.

```
figure
h1 = plot(1:30,y);
title('{\bf Forecasts and 95% Forecast Intervals}')
hold on
h2 = plot(31:50,yF,'r','LineWidth',2);
h3 = plot(31:50,yFCI,'r--','LineWidth',2);
h4 = plot(31:50,yMCBar,'k','LineWidth',2);
h5 = plot(31:50,yMCCI,'k--','LineWidth',2);
plot(30:31,[repmat(y(end),3,1),[yF(1),yFCI(1,:)]'],'b')
legend([h1,h2,h3(1),h4,h5(1)],'Observations','MMSE Forecasts',...
    'MMSE Forecast Intervals','Monte Carlo Forecasts',...
    'Monte Carlo Forecast Intervasls','Location','SouthWest')
xlabel('Time')
ylabel('y')
axis tight
hold off
```



The MMSE and Monte Carlo forecasts are virtually equivalent. There are minor discrepancies between the forecast intervals.

The width of the forecast intervals increases as time increases. This is a consequence of forecasting with integrated errors.

See Also

estimate | forecast | regARIMA

Related Examples

- "Forecast a Regression Model with Multiplicative Seasonal ARIMA Errors" on page 4-206

- "Verify Predictive Ability Robustness of a regARIMA Model" on page 4-212

More About

- "MMSE Forecasting Regression Models with ARIMA Errors" on page 4-215
- "Monte Carlo Forecasting of regARIMA Models" on page 4-220

Forecast a Regression Model with Multiplicative Seasonal ARIMA Errors

This example shows how to forecast a multiplicative seasonal ARIMA model using `forecast`. The response series is monthly international airline passenger numbers from 1949 to 1960.

Load the airline and recessions data sets. Transform the response.

```
load(fullfile(matlabroot,'examples','econ','Data_Airline.mat'))
load Data_Recessions
y = log(Data);
```

Construct the predictor (X), which determines whether the country was in a recession during the sampled period. A 0 in row t means the country was not in a recession in month t, and a 1 in row t means that it was in a recession in month t.

```
X = zeros(numel(dates),1); % Preallocation
for j = 1:size(Recessions,1)
    X(dates >= Recessions(j,1) & dates <= Recessions(j,2)) = 1;
end
```

Define index sets that partition the data into estimation and forecast samples.

```
nSim = 60; % Forecast period
T = length(y);
estInds = 1:(T-nSim);
foreInds = (T-nSim+1):T;
```

Estimate the regression model with multiplicative seasonal ARIMA$\{$ $(0,1,1)\times(0,1,1)_{12}$ errors:

$$y_t = X_t\beta + u_t$$

$$(1-L)(1-L^{12})u_t = (1+BL)(1+B_{12}L^{12})\varepsilon_t$$

Set the regression model intercept to 0 since it is not identifiable in a model with integrated errors.

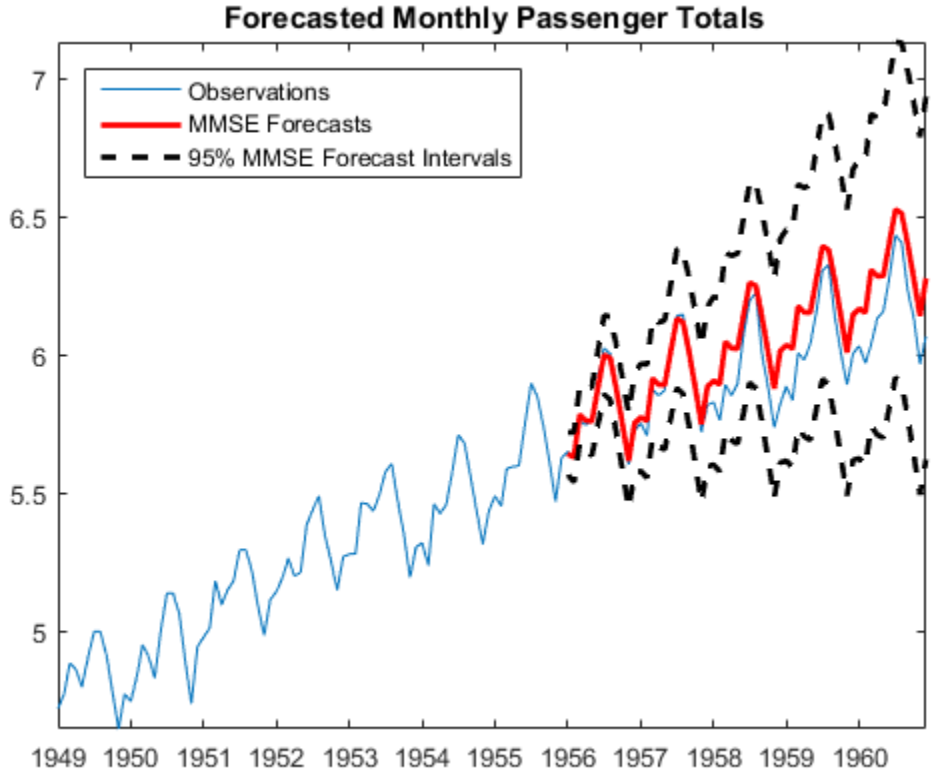
```
Mdl = regARIMA('D',1,'Seasonality',12,'MALags',1,'SMALags',12,...
```

```
    'Intercept',0);
EstMdl = estimate(Mdl,y(estInds),'X',X(estInds));
```

```
    Regression with ARIMA(0,1,1) Error Model Seasonally Integrated with Seasonal MA(12)
    ------------------------------------------------------------------------------------
    Conditional Probability Distribution: Gaussian

                                Standard          t
     Parameter       Value        Error       Statistic
    -----------   -----------   ------------   -----------
    Intercept               0         Fixed          Fixed
        MA{1}      -0.356617      0.103933       -3.43121
      SMA{12}      -0.677293      0.112935       -5.99718
        Beta1     0.00150984     0.0205331      0.0735321
     Variance     0.00151984   0.000214114        7.09828
```

Use the estimated coefficients of the model (contained in `EstMdl`), to generate MMSE forecasts and corresponding mean square errors over a 60-month horizon. Use the observed series as presample data. By default, `forecast` infers presample innovations and unconditional disturbances using the specified model and observations.

```
[YF,YMSE] = forecast(EstMdl,nSim,'XO',X(estInds),...
    'YO',y(estInds),'XF',X(foreInds));
ForecastInt = [YF,YF] + 1.96*[-sqrt(YMSE), sqrt(YMSE)];

figure
h1 = plot(dates,y);
title('{\bf Forecasted Monthly Passenger Totals}')
hold on
h2 = plot(dates(foreInds),YF,'Color','r','LineWidth',2);
h3 = plot(dates(foreInds),ForecastInt,'k--','LineWidth',2);
datetick
legend([h1,h2,h3(1)],'Observations','MMSE Forecasts',...
    '95% MMSE Forecast Intervals','Location','NorthWest')
axis tight
hold off
```

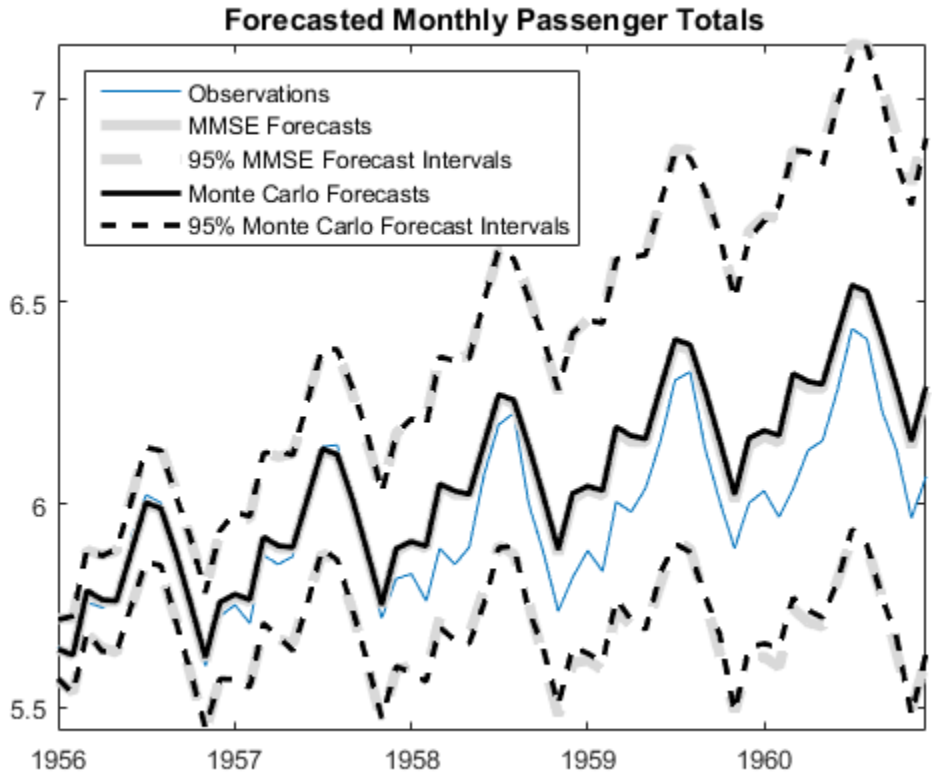


The regression model with SMA errors seems to forecast the series well, albeit slightly overestimating. Since the error process is nonstationary, the forecast intervals widen as time increases.

Compare the MMSE forecasts to Monte Carlo forecasts by simulating 500 sample paths from `EstMdl` over the forcast horizon.

```
[e0,u0] = infer(EstMdl,y(estInds),'X',X(estInds));
rng(5);
numPaths = 500;
ySim = simulate(EstMdl,nSim,'numPaths',numPaths,...
    'E0',e0,'U0',u0,'X',X(foreInds));
meanYSim = mean(ySim,2);
ForecastIntMC = [prctile(ySim,2.5,2),prctile(ySim,97.5,2)];
```

```
figure
h1 = plot(dates(foreInds),y(foreInds));
title('{\bf Forecasted Monthly Passenger Totals}')
hold on
h2 = plot(dates(foreInds),YF,'Color',[0.85,0.85,0.85],...
    'LineWidth',4);
h3 = plot(dates(foreInds),ForecastInt,'--','Color',...
    [0.85,0.85,0.85],'LineWidth',4);
h4 = plot(dates(foreInds),meanYSim,'k','LineWidth',2);
h5 = plot(dates(foreInds),ForecastIntMC,'k--','LineWidth',2);
datetick
legend([h1,h2,h3(1),h4,h5(1)],'Observations',...
    'MMSE Forecasts','95% MMSE Forecast Intervals',...
    'Monte Carlo Forecasts','95% Monte Carlo Forecast Intervals',...
    'Location','NorthWest')
axis tight
hold off
```



The MMSE forecasts and Monte Carlo mean forecasts are virtually indistinguishable. However, there are slight discrepancies between the theoretical 95% forecast intervals and the simulation-based 95% forecast intervals.

See Also

estimate | forecast | regARIMA | simulate

Related Examples

- "Forecast a Regression Model with ARIMA Errors" on page 4-202
- "Verify Predictive Ability Robustness of a regARIMA Model" on page 4-212

More About

- "MMSE Forecasting Regression Models with ARIMA Errors" on page 4-215
- "Monte Carlo Forecasting of regARIMA Models" on page 4-220

Verify Predictive Ability Robustness of a regARIMA Model

This example shows how to forecast a regression model with ARIMA errors, and how to check the model predictability robustness.

Load the Credit Defaults data set, assign the response (`IGD`) to `y` and the predictors `AGE`, `CPF`, and `SPR` to `X`. For illustration, specify that the response series is a regression model with AR(1) errors. To avoid distraction from the purpose of this example, assume that all predictor series are stationary.

```
load Data_CreditDefaults
y = Data(:,5);
X = Data(:,[1 3:4]);
T = size(X,1); % Sample size
Mdl = regARIMA(1,0,0);
```

Vary the validation sample size (`m`), and forecast responses from `Mdl` recursively. That is, for each validation sample size:

1. Fit the model to the data (`EstMdlY`).
2. Forecast responses from the estimated model (`yF`).
3. Compute the two performance statistics, root mean square error (RMSE) and root prediction mean square error (RPMSE).

```
m = 4:10;  % Validation sample lengths
rPMSE = m; % Preallocate rPMSE
rMSE = m;  % Preallocate rMSE

for k = 1:numel(m);
    yEst = y(1:(T-m(k)));    % Response data for estimation
    yVal = y((T-m(k)+1):T);  % Validation sample
    EstMdlY = estimate(Mdl,yEst,'X',X,'display','off');
    yHat = EstMdlY.Intercept + X(1:(T-m(k)),:)*EstMdlY.Beta';...
        % Estimation sample predicted values
    [e0,u0] = infer(EstMdlY,yEst,'X',X);
    yF = forecast(EstMdlY,m(k),'Y0',yEst,...
   'X0',X(1:T-m(k),:),'XF',X((T-m(k)+1):T,:));...
            % Validation sample predicted values
    rMSE(k) = sqrt(mean((yEst - yHat).^2));
    rPMSE(k) = sqrt(mean((yF - yVal).^2));
end
```

`rMSE` and `rPMSE` are vectors that contain the RMSE and RPMSE, respectively, for each validation sample.

Display the performance measures.

```
fprintf('\n m |  rMSE  |  rPMSE\n')
fprintf('====================\n')
for k = 1:length(m)
    fprintf('%2d | %0.4f | %0.4f\n',m(k),rMSE(k),rPMSE(k))
end
```

```
 m |  rMSE  |  rPMSE
====================
 4 | 0.0947 | 0.2274
 5 | 0.0808 | 0.1902
 6 | 0.0810 | 0.2036
 7 | 0.0714 | 0.1924
 8 | 0.0809 | 0.1532
 9 | 0.0720 | 0.1557
10 | 0.0899 | 0.1300
```

The predictive ability of this model is fairly robust because `rPMSE` changes slightly for increasing `m`. However, `rMSE` is less than `rPMSE` for all `m`. This signifies poor predictive ability.

Search for a better model by specifying, e.g., more AR or MA lags in the error model, and compare the PMSEs over these models. Choose the model with the lowest PMSE for a given validation sample size.

See Also

estimate | forecast | regARIMA

Related Examples

- "Verify Predictive Ability Robustness of a regARIMA Model" on page 4-212
- "Forecast a Regression Model with Multiplicative Seasonal ARIMA Errors" on page 4-206

More About

- "MMSE Forecasting Regression Models with ARIMA Errors" on page 4-215

- “Monte Carlo Forecasting of regARIMA Models” on page 4-220

MMSE Forecasting Regression Models with ARIMA Errors

In this section...

What Are MMSE Forecasts?

An objective of time series analysis is generating forecasts for responses over a future time horizon. That is, you can generate predictions for $y_{T+1},y_{T+2},\ldots,y_{T+h}$ given the following:

- An observed series $y_1, y_2,\ldots,y_T$
- A forecast horizon h
- Nonstochastic predictors $x_1,x_2,\ldots,x_T,\ldots,x_{T+h}$, where x_k is an r-vector containing the measurements of r predictors observed at time k
- A regression model with ARIMA errors

$$y_t = c + X_t\beta + u_t$$
$$\mathrm{H}(L)u_t = \mathrm{N}(L)\varepsilon_t,$$

where H(L) and N(L) are compound autoregressive and moving average lag operator polynomials (possibly containing integration), respectively.

Let $\hat{y}_{t+1}$ denote a forecast for the process at time $t + 1$, conditional on the history of the process up to time t (H_t), and assume that the predictors are fixed. The minimum mean square error (MMSE) forecast is the forecast $\hat{y}_{t+1}$ that minimizes expected square loss,

$$E\left(y_{t+1} - \hat{y}_{t+1} \mid H_t\right)^2.$$

Minimizing this loss function yields the MMSE forecast,

$$\hat{y}_{t+1} = E(y_{t+1} \mid H_t).$$

How forecast Generates MMSE Forecasts

forecast generates MMSE forecasts recursively. When you call `forecast`, you must specify a regARIMA model (`Mdl`) and the forecast horizon. You can also specify presample observations (`Y0`), predictors (`X0`), innovations (`E0`), and conditional disturbances (`U0`) using name-value pair arguments.

To begin forecasting y_t starting at time $T + 1$, use the last few observations of y_t and X_t as presample responses and predictors to initialize the forecast. Alternatively, you can specify presample unconditional disturbances or innovations.

However, when you specify presample data:

- If you provide presample predictor data (`X0`), then you must also provide predictor forecasts (`XF`). It is best practice to set `X0` to the same predictor matrix that estimates the parameters. If you do not provide presample and future predictors, then `forecast` ignores the regression component in the model.
- If the error process in `Mdl` contains a seasonal or nonseasonal autoregressive component, or seasonal or nonseasonal integration, then `forecast` requires a minimum of P presample unconditional disturbances to initialize the forecast. The property `P` of `Mdl` stores P.
- If the error process in `Mdl` contains a seasonal or nonseasonal moving average component, then `forecast` requires a minimum of Q presample innovations to initialize the forecast. The property `Q` of `Mdl` stores Q.
- If you provide a sufficient amount of presample unconditional disturbances, then `forecast` ignores `Y0` and `X0`. If you also do not provide `E0`, but provide enough presample unconditional disturbances, then `forecast` infers the required amount of presample innovations from the ARIMA error model and `U0`.
- If you provide a sufficient amount of presample responses and predictors (and do not provide `U0`), then `forecast` uses the regression model to infer the presample unconditional disturbances.
- If you do not provide presample observations, then `forecast` sets the required amount of presample unconditional disturbances and innovations to 0.
- If you provide an insufficient amount of presample observations, then `forecast` returns an error.

Consider generating forecasts from a regression model with ARMA(3,2) errors:

$$\begin{aligned} y_t &= c + X_t\beta + u_t \\ \left(1 - a_1L - a_2L^2 - a_3L^3\right)u_t &= \left(1 + b_1L + b_2L^2\right)\varepsilon_t \end{aligned}$$

or

$$a(L)u_t = b(L)\varepsilon_t,$$

where $a(L)$ and $B(L)$ are lag operator polynomials. The largest AR lag is 3, the largest MA lag is 2. This model does not contain any seasonal lags nor integration. Therefore, $P = 3$ and $Q = 2$. To forecast this model, you need three presample responses and predictors, or three presample unconditional disturbances, and two presample innovations.

Given presample unconditional disturbances (u_{T-2}, u_{T-1}, u_T), presample innovations $(\varepsilon_{T-1}, \varepsilon_T)$, and future predictors $(X_{T+1}, X_{T+2}, \ldots)$, you can forecast the model as follows:

- $\hat{u}_{T+1} = a_1u_T + a_2u_{T-1} + a_3u_{T-2} + b_1\varepsilon_T + b_2\varepsilon_{T-1}$
  $\hat{y}_{T+1} = c + X_{T+1}\beta + \hat{u}_{T+1}.$
- $\hat{u}_{T+2} = a_1\hat{u}_{T+1} + a_2u_T + a_3u_{T-1} + b_2\varepsilon_T$
  $\hat{y}_{T+2} = c + X_{T+2}\beta + \hat{u}_{T+2}.$
- $\hat{u}_{T+3} = a_1\hat{u}_{T+2} + a_2\hat{u}_{T+1} + a_3u_T$
  $\hat{y}_{T+3} = c + X_{T+3}\beta + \hat{u}_{T+3}.$

...

Note that:

- Future innovations take on their unconditional mean, 0.
- For stationary error processes, such as this one:
  - The forecasted unconditional disturbances converge to their unconditional mean,

$$E(u_t) = \frac{b(L)}{a(L)}E(\varepsilon_t) = 0.$$

- $c + X_t\beta$ governs the long-term behavior of the forecasted responses.

Forecast Error

The forecast error for an s-step ahead forecast of a regression model with ARIMA errors is

$$\begin{aligned} MSE &= E\left(y_{T+s} - \hat{y}_{T+s} \mid H_{T+s-1}\right)^2 \\ &= E\left(c + X_{T+s}\beta + u_{T+s} - c - X_{t+s}\beta - \hat{u}_{T+s} \mid H_{T+s-1}\right)^2 \\ &= E\left(u_{T+s} - \hat{u}_{T+s} \mid H_{T+s-1}\right)^2 \\ &= \frac{\mathrm{N}(L)}{\mathrm{H}(L)} E\left(\varepsilon_t^2 \mid H_{T+s-1}\right) \\ &= \psi(L)\sigma^2, \end{aligned}$$

where the dividend $\psi(L)$ is an infinite lag operator polynomial, and σ^2 is the innovation variance.

If the error process is stationary, then the coefficients of $\psi(L)$ are absolutely summable. Therefore, the MSE (mean square error) converges to the unconditional variance of the process [1].

If the error process is not stationary, then the MSE grows with increasing s.

References

[1] Box, G. E. P., G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis: Forecasting and Control*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

See Also

forecast | regARIMA

Related Examples

- "Verify Predictive Ability Robustness of a regARIMA Model" on page 4-212
- "Forecast a Regression Model with Multiplicative Seasonal ARIMA Errors" on page 4-206

- “Forecast a Regression Model with ARIMA Errors” on page 4-202

More About

- “Monte Carlo Forecasting of regARIMA Models” on page 4-220

Monte Carlo Forecasting of regARIMA Models

In this section...

Monte Carlo Forecasts

You can use Monte Carlo simulation to forecast an error process over a future time horizon. This is an alternative to minimum mean square error (MMSE) forecasting, which provides an analytical forecast solution. You can calculate MMSE forecasts using `forecast`.

To forecast a process using Monte Carlo simulation:

1. Fit a model to your observed series using `estimate`, or fully specify a `regARIMA` model.
2. Infer residuals (estimated innovations) and unconditional disturbances from the model using `infer` and the data. The inferred series are presample observations.
3. Generate many sample paths over the forecast horizon using `simulate` and the presample observations.

Advantage of Monte Carlo Forecasts

An advantage of Monte Carlo forecasting is that you obtain a complete distribution for future events, not just a point estimate and standard error. The simulation mean approximates the MMSE forecast. Use the 2.5th and 97.5th percentiles of the simulation realizations as endpoints for approximate 95% forecast intervals.

See Also

estimate | forecast | infer | regARIMA | simulate

Related Examples

- "Verify Predictive Ability Robustness of a regARIMA Model" on page 4-212
- "Forecast a Regression Model with Multiplicative Seasonal ARIMA Errors" on page 4-206

- “Forecast a Regression Model with ARIMA Errors” on page 4-202

More About

- “MMSE Forecasting Regression Models with ARIMA Errors” on page 4-215

Conditional Mean Models

Conditional Mean Models

In this section...

Unconditional vs. Conditional Mean

For a random variable y_t, the *unconditional mean* is simply the expected value, $E(y_t)$. In contrast, the *conditional mean* of y_t is the expected value of y_t given a conditioning set of variables, Ω_t. A *conditional mean model* specifies a functional form for $E(y_t \mid \Omega_t)$..

Static vs. Dynamic Conditional Mean Models

For a *static* conditional mean model, the conditioning set of variables is measured contemporaneously with the dependent variable y_t. An example of a static conditional mean model is the ordinary linear regression model. Given $\mathbf{x}_t$, a row vector of exogenous covariates measured at time t, and β, a column vector of coefficients, the conditional mean of y_t is expressed as the linear combination

$$E(y_t \mid x_t) = x_t'\beta$$

(that is, the conditioning set is $\Omega_t = x_t$).

In time series econometrics, there is often interest in the dynamic behavior of a variable over time. A *dynamic* conditional mean model specifies the expected value of y_t as a function of historical information. Let H_{t-1} denote the history of the process available at time t. A dynamic conditional mean model specifies the evolution of the conditional mean, $E(y_t \mid H_{t-1})$. Examples of historical information are:

- Past observations, $y_1, y_2,\ldots,y_{t-1}$
- Vectors of past exogenous variables, $x_1, x_2,\ldots, x_{t-1}$

- Past innovations, $\varepsilon_1,\varepsilon_2,\ldots,\varepsilon_{t-1}$

Conditional Mean Models for Stationary Processes

By definition, a covariance stationary stochastic process has an unconditional mean that is constant with respect to time. That is, if y_t is a stationary stochastic process, then $E(y_t) = \mu$ for all times t.

The constant mean assumption of stationarity does not preclude the possibility of a dynamic conditional expectation process. The serial autocorrelation between lagged observations exhibited by many time series suggests the expected value of y_t depends on historical information. By Wold's decomposition [1], you can write the conditional mean of any stationary process y_t as

$$E(y_t \mid H_{t-1}) = \mu + \sum_{i=1}^{\infty} \psi_i \varepsilon_{t-i},$$

where $\{\varepsilon_{t-i}\}$ are past observations of an uncorrelated innovation process with mean zero, and the coefficients ψ_i are absolutely summable. $E(y_t) = \mu$ is the constant unconditional mean of the stationary process.

Any model of the general linear form given by Equation 5-1 is a valid specification for the dynamic behavior of a stationary stochastic process. Special cases of stationary stochastic processes are the autoregressive (AR) model, moving average (MA) model, and the autoregressive moving average (ARMA) model.

References

[1] Wold, H. *A Study in the Analysis of Stationary Time Series*. Uppsala, Sweden: Almqvist & Wiksell, 1938.

See Also

arima

Related Examples

- "Specify Conditional Mean Models Using arima" on page 5-6

More About

Specify Conditional Mean Models Using arima

In this section...

Default ARIMA Model

The default ARIMA(p,D,q) model in Econometrics Toolbox is the nonseasonal model of the form

$$\Delta^D y_t = c + \phi_1 \Delta^D y_{t-1} + \ldots + \phi_p \Delta^D y_{t-p} + \theta_1 \varepsilon_{t-1} + \ldots + \theta_q \varepsilon_{t-q} + \varepsilon_t.$$

You can write this equation in condensed form using lag operator notation:

$$\phi(L)(1-L)^D y_t = c + \theta(L)\varepsilon_t$$

In either equation, the default innovation distribution is Gaussian with mean zero and constant variance.

You can specify a model of this form using the shorthand syntax `arima(p,D,q)`. For the input arguments `p`, `D`, and `q`, enter the number of nonseasonal AR terms (p), the order of nonseasonal integration (D), and the number of nonseasonal MA terms (q), respectively.

When you use this shorthand syntax, `arima` creates an `arima` model with these default property values.

Property Name	Property Data Type
`AR`	Cell vector of `NaN`s
`Beta`	Empty vector `[]` of regression coefficients corresponding to exogenous covariates
`Constant`	`NaN`
`D`	Degree of nonseasonal integration, D
`Distribution`	`'Gaussian'`
`MA`	Cell vector of `NaN`s
`P`	Number of AR terms plus degree of integration, $p + D$

Property Name	Property Data Type
Q	Number of MA terms, q
SAR	Cell vector of NaNs
SMA	Cell vector of NaNs
Variance	NaN

To assign nondefault values to any properties, you can modify the created model object using dot notation.

Notice that the inputs D and q are the values arima assigns to properties D and Q. However, the input argument p is not necessarily the value arima assigns to the model property P. P stores the number of presample observations needed to initialize the AR component of the model. For nonseasonal models, the required number of presample observations is $p + D$.

To illustrate, consider specifying the ARIMA(2,1,1) model

$$(1-\phi_1 L-\phi_2 L^2)(1-L)^1 y_t = c+(1+\theta_1 L)\varepsilon_t,$$

where the innovation process is Gaussian with (unknown) constant variance.

```
Mdl = arima(2,1,1)
```

```
Mdl =

    ARIMA(2,1,1) Model:
    --------------------
    Distribution: Name = 'Gaussian'
               P: 3
               D: 1
               Q: 1
        Constant: NaN
              AR: {NaN NaN} at Lags [1 2]
             SAR: {}
              MA: {NaN} at Lags [1]
             SMA: {}
        Variance: NaN
```

Notice that the model property P does not have value 2 (the AR degree). With the integration, a total of $p + D$ (here, 2 + 1 = 3) presample observations are needed to initialize the AR component of the model.

The created model, `Mdl`, has `NaN`s for all parameters. A `NaN` value signals that a parameter needs to be estimated or otherwise specified by the user. All parameters must be specified to forecast or simulate the model.

To estimate parameters, input the model object (along with data) to `estimate`. This returns a new fitted `arima` model object. The fitted model object has parameter estimates for each input `NaN` value.

Calling `arima` without any input arguments returns an ARIMA(0,0,0) model specification with default property values:

```
DefaultMdl = arima
```

```
DefaultMdl =

    ARIMA(0,0,0) Model:
    --------------------
    Distribution: Name = 'Gaussian'
               P: 0
               D: 0
               Q: 0
        Constant: NaN
              AR: {}
             SAR: {}
              MA: {}
             SMA: {}
        Variance: NaN
```

Specify Nonseasonal Models Using Name-Value Pairs

The best way to specify models to `arima` is using name-value pair arguments. You do not need, nor are you able, to specify a value for every model object property. `arima` assigns default values to any properties you do not (or cannot) specify.

In condensed, lag operator notation, nonseasonal ARIMA(p,D,q) models are of the form

$$\phi(L)(1-L)^{D}y_t = c+\theta(L)\varepsilon_t.$$

You can extend this model to an ARIMAX(p,D,q) model with the linear inclusion of exogenous variables. This model has the form

$$\phi(L)y_t = c^{*} + x_t^{'}\beta + \theta^{*}(L)\varepsilon_t,$$

where $c^* = c/(1–L)^D$ and $\theta^*(L) = \theta(L)/(1–L)^D$.

Tip If you specify a nonzero `D`, then Econometrics Toolbox differences the response series y_t *before* the predictors enter the model. You should preprocess the exogenous covariates x_t by testing for stationarity and differencing if any are unit root nonstationary. If any nonstationary exogenous covariate enters the model, then the false negative rate for significance tests of β can increase.

For the distribution of the innovations, ε_t, there are two choices:

- Independent and identically distributed (iid) Gaussian or Student's t with a constant variance, σ_ε^2.
- Dependent Gaussian or Student's t with a conditional variance process, σ_t^2. Specify the conditional variance model using a `garch`, `egarch`, or `gjr` model.

The `arima` default for the innovations is an iid Gaussian process with constant (scalar) variance.

In order to estimate, forecast, or simulate a model, you must specify the parametric form of the model (e.g., which lags correspond to nonzero coefficients, the innovation distribution) and any known parameter values. You can set any unknown parameters equal to `NaN`, and then input the model to `estimate` (along with data) to get estimated parameter values.

`arima` (and `estimate`) returns a model corresponding to the model specification. You can modify models to change or update the specification. Input models (with no `NaN` values) to `forecast` or `simulate` for forecasting and simulation, respectively. Here are some example specifications using name-value arguments.

Model	Specification
• $y_t = c + \phi_1 y_{t-1} + \varepsilon_t$ • $\varepsilon_t = \sigma_\varepsilon z_t$ • z_t Gaussian	`arima('AR',NaN)` or `arima(1,0,0)`
• $y_t = \varepsilon_t + \theta_1\varepsilon_{t-1} + \theta_2\varepsilon_{t-2}$	`arima('Constant',0,'MA',{NaN,NaN},...`

Model	Specification
• $\varepsilon_t = \sigma_\varepsilon z_t$ • z_t Student's t with unknown degrees of freedom	`'Distribution','t')`

• $(1-0.8L)(1-L)y_t = 0.2+(1+0.6L)\varepsilon_t$
• $\varepsilon_t = 0.1z_t$
• z_t Student's t with eight degrees of freedom
	`arima('Constant',0.2,'AR',0.8,'MA',0.6,...`
	`'Variance',0.1,'Distribution',...`
	`struct('Name','t','DoF',8))`

• $(1+0.5L)(1-L)^1\Delta y_t = x_t'\begin{bmatrix}-5\\2\end{bmatrix}+\varepsilon_t$ • $\varepsilon_t \sim N(0,1)$	`arima('AR',-0.5,'D',1,'Beta',[-5 2])`

You can specify the following name-value arguments to create nonseasonal `arima` models.

Name-Value Arguments for Nonseasonal ARIMA Models

Name	Corresponding Model Term(s) in Equation 5-2	When to Specify
`AR`	Nonseasonal AR coefficients, $\phi_1,\ldots,\phi_p$	To set equality constraints for the AR coefficients. For example, to specify the AR coefficients in the model $y_t = 0.8y_{t-1} - 0.2y_{t-2} + \varepsilon_t,$ specify `'AR',{0.8,-0.2}` You only need to specify the nonzero elements of `AR`. If the nonzero coefficients are at nonconsecutive lags, specify the corresponding lags using `ARLags`. Any coefficients you specify must correspond to a stable AR operator polynomial.
`ARLags`	Lags corresponding to nonzero,	`ARLags` is not a model property. Use this argument as a shortcut for specifying `AR` when the nonzero AR coefficients correspond to nonconsecutive lags. For example, to specify

Name	Corresponding Model Term(s) in Equation 5-2	When to Specify
	nonseasonal AR coefficients	nonzero AR coefficients at lags 1 and 12, e.g., $y_t = \phi_1 y_{t-1} + \phi_{12} y_{t-12} + \varepsilon_t$, specify `'ARLags',[1,12]`. Use `AR` and `ARLags` together to specify known nonzero AR coefficients at nonconsecutive lags. For example, if in the given AR(12) model $\phi_1 = 0.6$ and $\phi_{12} = -0.3$, specify `'AR',{0.6,-0.3},'ARLags',[1,12]`.
`Beta`	Values of the coefficients of the exogenous covariates	Use this argument to specify the values of the coefficients of the exogenous variables. For example, use `'Beta',[0.5 7 -2]` to specify $\beta = [0.5 \quad 7 \quad -2]'$. By default, `Beta` is an empty vector.
`Constant`	Constant term, c	To set equality constraints for c. For example, for a model with no constant term, specify `'Constant',0`. By default, `Constant` has value `NaN`.
`D`	Degree of nonseasonal differencing, D	To specify a degree of nonseasonal differencing greater than zero. For example, to specify one degree of differencing, specify `'D',1`. By default, `D` has value `0` (meaning no nonseasonal integration).
`Distribution`	Distribution of the innovation process	Use this argument to specify a Student's t innovation distribution. By default, the innovation distribution is Gaussian. For example, to specify a t distribution with unknown degrees of freedom, specify `'Distribution','t'`. To specify a t innovation distribution with known degrees of freedom, assign `Distribution` a data structure with fields `Name` and `DoF`. For example, for

Name	Corresponding Model Term(s) in Equation 5-2	When to Specify
		a t distribution with nine degrees of freedom, specify `'Distribution',struct('Name','t','DoF',9)`.
`MA`	Nonseasonal MA coefficients, $\theta_1,\ldots,\theta_q$	To set equality constraints for the MA coefficients. For example, to specify the MA coefficients in the model $$y_t = \varepsilon_t + 0.5\varepsilon_{t-1} + 0.2\varepsilon_{t-2},$$ specify `'MA',{0.5,0.2}`. You only need to specify the nonzero elements of `MA`. If the nonzero coefficients are at nonconsecutive lags, specify the corresponding lags using `MALags`. Any coefficients you specify must correspond to an invertible MA polynomial.
`MALags`	Lags corresponding to nonzero, nonseasonal MA coefficients	`MALags` is not a model property. Use this argument as a shortcut for specifying `MA` when the nonzero MA coefficients correspond to nonconsecutive lags. For example, to specify nonzero MA coefficients at lags 1 and 4, e.g., $$y_t = \varepsilon_t + \theta_1\varepsilon_{t-1} + \theta_4\varepsilon_{t-4},$$ specify `'MALags',[1,4]`. Use `MA` and `MALags` together to specify known nonzero MA coefficients at nonconsecutive lags. For example, if in the given MA(4) model $\theta_1 = 0.5$ and $\theta_4 = 0.2$, specify `'MA',{0.4,0.2},'MALags',[1,4]`.
`Variance`	• Scalar variance of the innovation process, σ_ε^2	• To set equality constraints for σ_ε^2. For example, for a model with known variance 0.1, specify `'Variance',0.1`. By default, `Variance` has value `NaN`.

Name	Corresponding Model Term(s) in Equation 5-2	When to Specify
	• Conditional variance process, σ_t^2	• To specify a conditional variance model, σ_t^2. Set `'Variance'` equal to a conditional variance model object, e.g., a `garch` model object.

Note: You cannot assign values to the properties `P` and `Q`. For nonseasonal models,

- `arima` sets `P` equal to $p + D$
- `arima` sets `Q` equal to q

Specify Multiplicative Models Using Name-Value Pairs

For a time series with periodicity s, define the degree p_s seasonal AR operator polynomial, $\Phi(L) = (1 - \Phi_1 L^{p_1} - \ldots - \Phi_{p_s} L^{p_s})$, and the degree q_s seasonal MA operator polynomial, $\Theta(L) = (1 + \Theta_1 L^{q_1} + \ldots + \Theta_{q_s} L^{q_s})$. Similarly, define the degree p nonseasonal AR operator polynomial, $\phi(L) = (1 - \phi_1 L - \ldots - \phi_p L^p)$, and the degree q nonseasonal MA operator polynomial,

$$\theta(L) = (1 + \theta_1 L + \ldots + \theta_q L^q).$$

A multiplicative ARIMA model with degree D nonseasonal integration and degree s seasonality is given by

$$\phi(L)\Phi(L)(1 - L)^D(1 - L^s)y_t = c + \theta(L)\Theta(L)\varepsilon_t.$$

The innovation series can be an independent or dependent Gaussian or Student's t process. The `arima` default for the innovation distribution is an iid Gaussian process with constant (scalar) variance.

In addition to the arguments for specifying nonseasonal models (described in Name-Value Arguments for Nonseasonal ARIMA Models), you can specify these name-value

arguments to create a multiplicative `arima` model. You can extend an ARIMAX model similarly to include seasonal effects.

Name-Value Arguments for Seasonal ARIMA Models

Argument	Corresponding Model Term(s) in Equation 5-5	When to Specify
`SAR`	Seasonal AR coefficients, $\Phi_1,\ldots,\Phi_{p_s}$	To set equality constraints for the seasonal AR coefficients. When specifying AR coefficients, use the sign opposite to what appears in Equation 5-5 (that is, use the sign of the coefficient as it would appear on the right side of the equation). Use `SARLags` to specify the lags of the nonzero seasonal AR coefficients. Specify the lags associated with the seasonal polynomials in the periodicity of the observed data (e.g., 4, 8,... for quarterly data, or 12, 24,... for monthly data), and not as multiples of the seasonality (e.g., 1, 2,...). For example, to specify the model $$(1-0.8L)(1-0.2L^{12})y_t=\varepsilon_t,$$ specify `'AR',0.8,'SAR',0.2,'SARLags',12`. Any coefficient values you enter must correspond to a stable seasonal AR polynomial.
`SARLags`	Lags corresponding to nonzero seasonal AR coefficients, in the periodicity of the observed series	`SARLags` is not a model property. Use this argument when specifying `SAR` to indicate the lags of the nonzero seasonal AR coefficients. For example, to specify the model $$(1-\phi L)(1-\Phi_{12}L^{12})y_t=\varepsilon_t,$$ specify `'ARLags',1,'SARLags',12`.

Argument	Corresponding Model Term(s) in Equation 5-5	When to Specify
`SMA`	Seasonal MA coefficients, $\Theta_1,\ldots,\Theta_{q_s}$	To set equality constraints for the seasonal MA coefficients. Use `SMALags` to specify the lags of the nonzero seasonal MA coefficients. Specify the lags associated with the seasonal polynomials in the periodicity of the observed data (e.g., 4, 8,... for quarterly data, or 12, 24,... for monthly data), and not as multiples of the seasonality (e.g., 1, 2,...). For example, to specify the model $$y_t = (1+0.6L)(1+0.2L^{12})\varepsilon_t,$$ specify `'MA',0.6,'SMA',0.2,'SMALags',12`. Any coefficient values you enter must correspond to an invertible seasonal MA polynomial.
`SMALags`	Lags corresponding to the nonzero seasonal MA coefficients, in the periodicity of the observed series	`SMALags` is not a model property. Use this argument when specifying `SMA` to indicate the lags of the nonzero seasonal MA coefficients. For example, to specify the model $$y_t = (1+\theta_1 L)(1+\Theta_4 L^4)\varepsilon_t,$$ specify `'MALags',1,'SMALags',4`.
`Seasonality`	Seasonal periodicity, s	To specify the degree of seasonal integration s in the seasonal differencing polynomial $\Delta_s = 1 - L^s$. For example, to specify the periodicity for seasonal integration of monthly data, specify `'Seasonality',12`. If you specify nonzero `Seasonality`, then the degree of the whole seasonal differencing polynomial is one. By default, `Seasonality` has

Argument	Corresponding Model Term(s) in Equation 5-5	When to Specify
		value `0` (meaning periodicity and no seasonal integration).

Note: You cannot assign values to the properties `P` and `Q`. For multiplicative ARIMA models,

- `arima` sets `P` equal to $p + D + p_s + s$
- `arima` sets `Q` equal to $q + q_s$

See Also

arima | estimate | forecast | simulate

Related Examples

- "AR Model Specifications" on page 5-21
- "MA Model Specifications" on page 5-29
- "ARMA Model Specifications" on page 5-37
- "ARIMA Model Specifications" on page 5-43
- "ARIMAX Model Specifications" on page 5-61
- "Multiplicative ARIMA Model Specifications" on page 5-48
- "Modify Properties of Conditional Mean Model Objects" on page 5-65
- "Specify Conditional Mean Model Innovation Distribution" on page 5-72
- "Model Seasonal Lag Effects Using Indicator Variables" on page 5-117

More About

- "Autoregressive Model" on page 5-18
- "Moving Average Model" on page 5-27
- "Autoregressive Moving Average Model" on page 5-34
- "ARIMA Model" on page 5-41
- "ARIMAX(p,D,q) Model" on page 5-58

- "ARIMA Model Including Exogenous Covariates" on page 5-58
- "Multiplicative ARIMA Model" on page 5-46

Autoregressive Model

In this section...

AR(*p*) Model

Many observed time series exhibit serial autocorrelation; that is, linear association between lagged observations. This suggests past observations might predict current observations. The autoregressive (AR) process models the conditional mean of y_t as a function of past observations, $y_{t-1}, y_{t-2}, \ldots, y_{t-p}$. An AR process that depends on p past observations is called an AR model of degree p, denoted by AR(p).

The form of the AR(p) model in Econometrics Toolbox is

$$y_t = c + \phi_1 y_{t-1} + \ldots + \phi_p y_{t-p} + \varepsilon_t,$$

where ε_t is an uncorrelated innovation process with mean zero.

In lag operator polynomial notation, $L^i y_t = y_{t-i}$. Define the degree p AR lag operator polynomial $\phi(L) = (1 - \phi_1 L - \ldots - \phi_p L^p)$. You can write the AR($p$) model as

$$\phi(L) y_t = c + \varepsilon_t.$$

The signs of the coefficients in the AR lag operator polynomial, $\phi(L)$, are opposite to the right side of Equation 5-6. When specifying and interpreting AR coefficients in Econometrics Toolbox, use the form in Equation 5-6.

Stationarity of the AR Model

Consider the AR(p) model in lag operator notation,

$$\phi(L) y_t = c + \varepsilon_t.$$

From this expression, you can see that

$$y_t = \mu + \phi^{-1}(L)\varepsilon_t = \mu + \psi(L)\varepsilon_t,$$

where

$$\mu = \frac{c}{\left(1 - \phi_1 - \ldots - \phi_p\right)}$$

is the unconditional mean of the process, and $\psi(L)$ is an infinite-degree lag operator polynomial, $(1 + \psi_1 L + \psi_2 L^2 + \ldots)$.

Note: The `Constant` property of an `arima` model object corresponds to c, and not the unconditional mean μ.

By Wold's decomposition [1], Equation 5-8 corresponds to a stationary stochastic process provided the coefficients ψ_i are absolutely summable. This is the case when the AR polynomial, $\phi(L)$, is *stable*, meaning all its roots lie outside the unit circle.

Econometrics Toolbox enforces stability of the AR polynomial. When you specify an AR model using `arima`, you get an error if you enter coefficients that do not correspond to a stable polynomial. Similarly, `estimate` imposes stationarity constraints during estimation.

References

[1] Wold, H. *A Study in the Analysis of Stationary Time Series*. Uppsala, Sweden: Almqvist & Wiksell, 1938.

See Also

arima | estimate

Related Examples

- "Specify Conditional Mean Models Using arima" on page 5-6

- "AR Model Specifications" on page 5-21
- "Plot the Impulse Response Function" on page 5-88

More About

- "Conditional Mean Models" on page 5-3
- "Autoregressive Moving Average Model" on page 5-34

AR Model Specifications

In this section...

Default AR Model

This example shows how to use the shorthand `arima(p,D,q)` syntax to specify the default AR(p) model,

$$y_t = c + \phi_1 y_{t-1} + \ldots + \phi_p y_{t-p} + \varepsilon_t.$$

By default, all parameters in the created model object have unknown values, and the innovation distribution is Gaussian with constant variance.

Specify the default AR(2) model:

```
model = arima(2,0,0)
```

```
model =

    ARIMA(2,0,0) Model:
    --------------------
    Distribution: Name = 'Gaussian'
               P: 2
               D: 0
               Q: 0
        Constant: NaN
              AR: {NaN NaN} at Lags [1 2]
             SAR: {}
              MA: {}
             SMA: {}
        Variance: NaN
```

The output shows that the created model object, `model`, has `NaN` values for all model parameters: the constant term, the AR coefficients, and the variance. You can modify the created model object using dot notation, or input it (along with data) to `estimate`.

AR Model with No Constant Term

This example shows how to specify an AR(p) model with constant term equal to zero. Use name-value syntax to specify a model that differs from the default model.

Specify an AR(2) model with no constant term,

$$y_t = \phi_1 y_{t-1} + \phi_2 y_{t-2} + \varepsilon_t,$$

where the innovation distribution is Gaussian with constant variance.

```
model = arima('ARLags',1:2,'Constant',0)
```

```
model =

    ARIMA(2,0,0) Model:
    --------------------
    Distribution: Name = 'Gaussian'
               P: 2
               D: 0
               Q: 0
        Constant: 0
              AR: {NaN NaN} at Lags [1 2]
             SAR: {}
              MA: {}
             SMA: {}
        Variance: NaN
```

The `ARLags` name-value argument specifies the lags corresponding to nonzero AR coefficients. The property `Constant` in the created model object is equal to `0`, as specified. The model object has default values for all other properties, including `NaN` values as placeholders for the unknown parameters: the AR coefficients and scalar variance.

You can modify the created model object using dot notation, or input it (along with data) to `estimate`.

AR Model with Nonconsecutive Lags

This example shows how to specify an AR(p) model with nonzero coefficients at nonconsecutive lags.

Specify an AR(4) model with nonzero AR coefficients at lags 1 and 4 (and no constant term),

$$y_t = 0.2 + 0.8y_{t-1} - 0.1y_{t-4} + \varepsilon_t,$$

where the innovation distribution is Gaussian with constant variance.

```
model = arima('ARLags',[1,4],'Constant',0)
```

```
model =

    ARIMA(4,0,0) Model:
    --------------------
    Distribution: Name = 'Gaussian'
               P: 4
               D: 0
               Q: 0
        Constant: 0
              AR: {NaN NaN} at Lags [1 4]
             SAR: {}
              MA: {}
             SMA: {}
        Variance: NaN
```

The output shows the nonzero AR coefficients at lags 1 and 4, as specified. The property `P` is equal to `4`, the number of presample observations needed to initialize the AR model. The unconstrained parameters are equal to `NaN`.

Display the value of `AR`:

```
model.AR
```

```
ans =

    [NaN]    [0]    [0]    [NaN]
```

The AR cell array returns four elements. The first and last elements (corresponding to lags 1 and 4) have value NaN, indicating these coefficients are nonzero and need to be estimated or otherwise specified by the user. arima sets the coefficients at interim lags equal to zero to maintain consistency with MATLAB® cell array indexing.

ARMA Model with Known Parameter Values

This example shows how to specify an ARMA(p, q) model with known parameter values. You can use such a fully specified model as an input to simulate or forecast.

Specify the ARMA(1,1) model

$$y_t = 0.3 + 0.7\phi y_{t-1} + \varepsilon_t + 0.4\varepsilon_{t-1},$$

where the innovation distribution is Student's t with 8 degrees of freedom, and constant variance 0.15.

```
tdist = struct('Name','t','DoF',8);
model = arima('Constant',0.3,'AR',0.7,'MA',0.4,...
      'Distribution',tdist,'Variance',0.15)
```

```
model =

    ARIMA(1,0,1) Model:
    --------------------
    Distribution: Name = 't', DoF = 8
               P: 1
               D: 0
               Q: 1
        Constant: 0.3
              AR: {0.7} at Lags [1]
             SAR: {}
              MA: {0.4} at Lags [1]
             SMA: {}
        Variance: 0.15
```

Because all parameter values are specified, the created model has no `NaN` values. The functions `simulate` and `forecast` don't accept input models with `NaN` values.

AR Model with a t Innovation Distribution

This example shows how to specify an AR(P) model with a Student's t innovation distribution.

Specify an AR(2) model with no constant term,

$$y_t = \phi_1 y_{t-1} + \phi_2 y_{t-2} + \varepsilon_t,$$

where the innovations follow a Student's t distribution with unknown degrees of freedom.

```
model = arima('Constant',0,'ARLags',1:2,'Distribution','t')
```

```
model =

    ARIMA(2,0,0) Model:
    --------------------
    Distribution: Name = 't', DoF = NaN
               P: 2
               D: 0
               Q: 0
        Constant: 0
              AR: {NaN NaN} at Lags [1 2]
             SAR: {}
              MA: {}
             SMA: {}
        Variance: NaN
```

The value of `Distribution` is a `struct` array with field `Name` equal to `'t'` and field `DoF` equal to `NaN`. The `NaN` value indicates the degrees of freedom are unknown, and need to be estimated using `estimate` or otherwise specified by the user.

See Also

arima | estimate | forecast | simulate | `struct`

Related Examples

- “Specify Conditional Mean Models Using arima” on page 5-6

- “Modify Properties of Conditional Mean Model Objects” on page 5-65
- “Specify Conditional Mean Model Innovation Distribution” on page 5-72

More About

- “Autoregressive Model” on page 5-18

Moving Average Model

In this section...

MA(*q*) Model

The moving average (MA) model captures serial autocorrelation in a time series y_t by expressing the conditional mean of y_t as a function of past innovations, $\varepsilon_{t-1}, \varepsilon_{t-2}, \ldots, \varepsilon_{t-q}$. An MA model that depends on q past innovations is called an MA model of degree q, denoted by MA(q).

The form of the MA(q) model in Econometrics Toolbox is

$$y_t = c + \varepsilon_t + \theta_1 \varepsilon_{t-1} + \ldots + \theta_q \varepsilon_{t-q},$$

where ε_t is an uncorrelated innovation process with mean zero. For an MA process, the unconditional mean of y_t is $\mu = c$.

In lag operator polynomial notation, $L^i y_t = y_{t-i}$. Define the degree q MA lag operator polynomial $\theta(L) = (1 + \theta_1 L + \ldots + \theta_q L^q)$. You can write the MA($q$) model as

$$y_t = \mu + \theta(L)\varepsilon_t.$$

Invertibility of the MA Model

By Wold's decomposition [1], an MA(q) process is always stationary because $\theta(L)$ is a finite-degree polynomial.

For a given process, however, there is no unique MA polynomial—there is always a *noninvertible* and *invertible* solution [2]. For uniqueness, it is conventional to impose invertibility constraints on the MA polynomial. Practically speaking, choosing the invertible solution implies the process is *causal*. An invertible MA process can be

expressed as an infinite-degree AR process, meaning only past events (not future events) predict current events. The MA operator polynomial $\theta(L)$ is invertible if all its roots lie outside the unit circle.

Econometrics Toolbox enforces invertibility of the MA polynomial. When you specify an MA model using `arima`, you get an error if you enter coefficients that do not correspond to an invertible polynomial. Similarly, `estimate` imposes invertibility constraints during estimation.

References

[1] Wold, H. *A Study in the Analysis of Stationary Time Series*. Uppsala, Sweden: Almqvist & Wiksell, 1938.

[2] Hamilton, J. D. *Time Series Analysis*. Princeton, NJ: Princeton University Press, 1994.

See Also

arima | estimate

Related Examples

- "Specify Conditional Mean Models Using arima" on page 5-6
- "MA Model Specifications" on page 5-29
- "Plot the Impulse Response Function" on page 5-88

More About

- "Conditional Mean Models" on page 5-3
- "Autoregressive Moving Average Model" on page 5-34

MA Model Specifications

In this section...

Default MA Model

This example shows how to use the shorthand `arima(p,D,q)` syntax to specify the default MA

$$y_t = c + \varepsilon_t + \theta_1\varepsilon_{t-1} + \ldots + \theta_q\varepsilon_{t-q}.$$

By default, all parameters in the created model object have unknown values, and the innovation distribution is Gaussian with constant variance.

Specify the default MA(3) model:

```
model = arima(0,0,3)
```

```
model =

    ARIMA(0,0,3) Model:
    --------------------
    Distribution: Name = 'Gaussian'
               P: 0
               D: 0
               Q: 3
        Constant: NaN
              AR: {}
             SAR: {}
              MA: {NaN NaN NaN} at Lags [1 2 3]
             SMA: {}
        Variance: NaN
```

The output shows that the created model object, `model`, has `NaN` values for all model parameters: the constant term, the MA coefficients, and the variance. You can modify the created model object using dot notation, or input it (along with data) to `estimate`.

MA Model with No Constant Term

This example shows how to specify an MA(q) model with constant term equal to zero. Use name-value syntax to specify a model that differs from the default model.

Specify an MA(2) model with no constant term,

$$y_t = \varepsilon_t + \theta_1\varepsilon_{t-1} + \theta_2\varepsilon_{t-2},$$

where the innovation distribution is Gaussian with constant variance.

```
model = arima('MALags',1:2,'Constant',0)
```

```
model =

    ARIMA(0,0,2) Model:
    --------------------
    Distribution: Name = 'Gaussian'
               P: 0
               D: 0
               Q: 2
        Constant: 0
              AR: {}
             SAR: {}
              MA: {NaN NaN} at Lags [1 2]
             SMA: {}
        Variance: NaN
```

The `MALags` name-value argument specifies the lags corresponding to nonzero MA coefficients. The property `Constant` in the created model object is equal to `0`, as specified. The model object has default values for all other properties, including `NaN` values as placeholders for the unknown parameters: the MA coefficients and scalar variance.

You can modify the created model variable, or input it (along with data) to `estimate`.

MA Model with Nonconsecutive Lags

This example shows how to specify an MA(q) model with nonzero coefficients at nonconsecutive lags.

Specify an MA(4) model with nonzero MA coefficients at lags 1 and 4 (an no constant term),

$$y_t = \varepsilon_t + \theta_1\varepsilon_{t-1} + \theta_{12}\varepsilon_{t-12},$$

where the innovation distribution is Gaussian with constant variance.

```
model = arima('MALags',[1,4],'Constant',0)
```

```
model =

    ARIMA(0,0,4) Model:
    -------------------
    Distribution: Name = 'Gaussian'
               P: 0
               D: 0
               Q: 4
        Constant: 0
              AR: {}
             SAR: {}
              MA: {NaN NaN} at Lags [1 4]
             SMA: {}
        Variance: NaN
```

The output shows the nonzero AR coefficients at lags 1 and 4, as specified. The property `Q` is equal to `4`, the number of presample innovations needed to initialize the MA model. The unconstrained parameters are equal to `NaN`.

Display the value of `MA`:

```
model.MA
```

```
ans =

    [NaN]    [0]    [0]    [NaN]
```

The MA cell array returns four elements. The first and last elements (corresponding to lags 1 and 4) have value NaN, indicating these coefficients are nonzero and need to be estimated or otherwise specified by the user. arima sets the coefficients at interim lags equal to zero to maintain consistency with MATLAB® cell array indexing.

MA Model with Known Parameter Values

This example shows how to specify an MA(q) model with known parameter values. You can use such a fully specified model as an input to simulate or forecast.

Specify the MA(4) model

$$y_t = 0.1 + \varepsilon_t + 0.7\varepsilon_{t-1} + 0.2\varepsilon_{t-4},$$

where the innovation distribution is Gaussian with constant variance 0.15.

```
model = arima('Constant',0.1,'MA',{0.7,0.2},...
      'MALags',[1,4],'Variance',0.15)
```

```
model =

    ARIMA(0,0,4) Model:
    --------------------
    Distribution: Name = 'Gaussian'
               P: 0
               D: 0
               Q: 4
        Constant: 0.1
              AR: {}
             SAR: {}
              MA: {0.7 0.2} at Lags [1 4]
             SMA: {}
        Variance: 0.15
```

Because all parameter values are specified, the created model object has no NaN values. The functions simulate and forecast don't accept input models with NaN values.

MA Model with a t Innovation Distribution

This example shows how to specify an MA(q) model with a Student's t innovation distribution.

Specify an MA(2) model with no constant term,

$$y_t = \varepsilon_t + \theta_1\varepsilon_{t-1} + \theta_2\varepsilon_{t-2},$$

where the innovation process follows a Student's t distribution with eight degrees of freedom.

```
tdist = struct('Name','t','DoF',8);
model = arima('Constant',0,'MALags',1:2,'Distribution',tdist)
```

```
model =

    ARIMA(0,0,2) Model:
    --------------------
    Distribution: Name = 't', DoF = 8
               P: 0
               D: 0
               Q: 2
        Constant: 0
              AR: {}
             SAR: {}
              MA: {NaN NaN} at Lags [1 2]
             SMA: {}
        Variance: NaN
```

The value of `Distribution` is a `struct` array with field `Name` equal to `'t'` and field `DoF` equal to `8`. When you specify the degrees of freedom, they aren't estimated if you input the model to `estimate`.

See Also

arima | estimate | forecast | simulate | `struct`

Related Examples

- "Specify Conditional Mean Models Using arima" on page 5-6
- "Modify Properties of Conditional Mean Model Objects" on page 5-65
- "Specify Conditional Mean Model Innovation Distribution" on page 5-72

More About

- "Moving Average Model" on page 5-27

Autoregressive Moving Average Model

In this section...

ARMA(*p*,*q*) Model

For some observed time series, a very high-order AR or MA model is needed to model the underlying process well. In this case, a combined autoregressive moving average (ARMA) model can sometimes be a more parsimonious choice.

An ARMA model expresses the conditional mean of y_t as a function of both past observations, $y_{t-1},\ldots,y_{t-p}$, and past innovations, $\varepsilon_{t-1},\ldots,\varepsilon_{t-q}$. The number of past observations that y_t depends on, p, is the AR degree. The number of past innovations that y_t depends on, q, is the MA degree. In general, these models are denoted by ARMA(p,q).

The form of the ARMA(p,q) model in Econometrics Toolbox is

$$y_t = c + \phi_1 y_{t-1} + \ldots + \phi_p y_{t-p} + \varepsilon_t + \theta_1 \varepsilon_{t-1} + \ldots + \theta_q \varepsilon_{t-q},$$

where ε_t is an uncorrelated innovation process with mean zero.

In lag operator polynomial notation, $L^i y_t = y_{t-i}$. Define the degree p AR lag operator polynomial $\phi(L) = (1 - \phi_1 L - \ldots - \phi_p L^p)$. Define the degree q MA lag operator polynomial $\theta(L) = (1 + \theta_1 L + \ldots + \theta_q L^q)$. You can write the ARMA(p,q) model as

$$\phi(L) y_t = c + \theta(L) \varepsilon_t.$$

The signs of the coefficients in the AR lag operator polynomial, $\phi(L)$, are opposite to the right side of Equation 5-10. When specifying and interpreting AR coefficients in Econometrics Toolbox, use the form in Equation 5-10.

Stationarity and Invertibility of the ARMA Model

Consider the ARMA(p,q) model in lag operator notation,

$$\phi(L)y_t = c + \theta(L)\varepsilon_t.$$

From this expression, you can see that

$$y_t = \mu + \frac{\theta(L)}{\phi(L)}\varepsilon_t = \mu + \psi(L)\varepsilon_t,$$

where

$$\mu = \frac{c}{\left(1 - \phi_1 - \ldots - \phi_p\right)}$$

is the unconditional mean of the process, and $\psi(L)$ is a rational, infinite-degree lag operator polynomial, $(1 + \psi_1 L + \psi_2 L^2 + \ldots)$.

Note: The `Constant` property of an `arima` model object corresponds to c, and not the unconditional mean μ.

By Wold's decomposition [1], Equation 5-12 corresponds to a stationary stochastic process provided the coefficients ψ_i are absolutely summable. This is the case when the AR polynomial, $\phi(L)$, is *stable*, meaning all its roots lie outside the unit circle. Additionally, the process is *causal* provided the MA polynomial is *invertible*, meaning all its roots lie outside the unit circle.

Econometrics Toolbox enforces stability and invertibility of ARMA processes. When you specify an ARMA model using `arima`, you get an error if you enter coefficients that do not correspond to a stable AR polynomial or invertible MA polynomial. Similarly, `estimate` imposes stationarity and invertibility constraints during estimation.

References

[1] Wold, H. *A Study in the Analysis of Stationary Time Series*. Uppsala, Sweden: Almqvist & Wiksell, 1938.

See Also

arima | estimate

Related Examples

- "Specify Conditional Mean Models Using arima" on page 5-6
- "ARMA Model Specifications" on page 5-37
- "Plot the Impulse Response Function" on page 5-88

More About

- "Conditional Mean Models" on page 5-3
- "Autoregressive Model" on page 5-18
- "Moving Average Model" on page 5-27
- "ARIMA Model" on page 5-41

ARMA Model Specifications

In this section...

Default ARMA Model

This example shows how to use the shorthand `arima(p,D,q)` syntax to specify the default ARMA(p, q) model,

$$y_t = 6 + 0.2y_{t-1} - 0.3y_{t-2} + 3x_t + \varepsilon_t + 0.1\varepsilon_{t-1}$$

By default, all parameters in the created model object have unknown values, and the innovation distribution is Gaussian with constant variance.

Specify the default ARMA(1,1) model:

```
model = arima(1,0,1)
```

```
model =

    ARIMA(1,0,1) Model:
    --------------------
    Distribution: Name = 'Gaussian'
               P: 1
               D: 0
               Q: 1
        Constant: NaN
              AR: {NaN} at Lags [1]
             SAR: {}
              MA: {NaN} at Lags [1]
             SMA: {}
        Variance: NaN
```

The output shows that the created model object, `model`, has `NaN` values for all model parameters: the constant term, the AR and MA coefficients, and the variance. You

can modify the created model object using dot notation, or input it (along with data) to `estimate`.

ARMA Model with No Constant Term

This example shows how to specify an ARMA(p, q) model with constant term equal to zero. Use name-value syntax to specify a model that differs from the default model.

Specify an ARMA(2,1) model with no constant term,

$$y_t = \phi_1 y_{t-1} + \phi_2 y_{t-2} + \varepsilon_t + \theta_1 \varepsilon_{t-1},$$

where the innovation distribution is Gaussian with constant variance.

```
model = arima('ARLags',1:2,'MALags',1,'Constant',0)
```

```
model =

    ARIMA(2,0,1) Model:
    --------------------
    Distribution: Name = 'Gaussian'
               P: 2
               D: 0
               Q: 1
        Constant: 0
              AR: {NaN NaN} at Lags [1 2]
             SAR: {}
              MA: {NaN} at Lags [1]
             SMA: {}
        Variance: NaN
```

The `ArLags` and `MaLags` name-value pair arguments specify the lags corresponding to nonzero AR and MA coefficients, respectively. The property `Constant` in the created model object is equal to `0`, as specified. The model has default values for all other properties, including `NaN` values as placeholders for the unknown parameters: the AR and MA coefficients, and scalar variance.

You can modify the created model using dot notation, or input it (along with data) to `estimate`.

ARMA Model with Known Parameter Values

This example shows how to specify an ARMA(p, q) model with known parameter values. You can use such a fully specified model as an input to `simulate` or `forecast`.

Specify the ARMA(1,1) model

$$y_t = 0.3 + 0.7\phi y_{t-1} + \varepsilon_t + 0.4\varepsilon_{t-1},$$

where the innovation distribution is Student's t with 8 degrees of freedom, and constant variance 0.15.

```
tdist = struct('Name','t','DoF',8);
model = arima('Constant',0.3,'AR',0.7,'MA',0.4,...
      'Distribution',tdist,'Variance',0.15)
```

```
model =

    ARIMA(1,0,1) Model:
    --------------------
    Distribution: Name = 't', DoF = 8
               P: 1
               D: 0
               Q: 1
        Constant: 0.3
              AR: {0.7} at Lags [1]
             SAR: {}
              MA: {0.4} at Lags [1]
             SMA: {}
        Variance: 0.15
```

Because all parameter values are specified, the created model has no `NaN` values. The functions `simulate` and `forecast` don't accept input models with `NaN` values.

See Also

arima | estimate | forecast | simulate | `struct`

Related Examples

- "Specify Conditional Mean Models Using arima" on page 5-6
- "Modify Properties of Conditional Mean Model Objects" on page 5-65
- "Specify Conditional Mean Model Innovation Distribution" on page 5-72

More About

- "Autoregressive Moving Average Model" on page 5-34

ARIMA Model

The autoregressive integrated moving average (ARIMA) process generates nonstationary series that are integrated of order *D*, denoted *I*(*D*). A nonstationary *I*(*D*) process is one that can be made stationary by taking *D* differences. Such processes are often called *difference-stationary* or *unit root* processes.

A series that you can model as a stationary ARMA(*p*,*q*) process after being differenced *D* times is denoted by ARIMA(*p*,*D*,*q*). The form of the ARIMA(*p*,*D*,*q*) model in Econometrics Toolbox is

$$\Delta^D y_t = c + \phi_1 \Delta^D y_{t-1} + \ldots + \phi_p \Delta^D y_{t-p} + \varepsilon_t + \theta_1 \varepsilon_{t-1} + \ldots + \theta_q \varepsilon_{t-q},$$

where $\Delta^D y_t$ denotes a *D*th differenced time series, and ε_t is an uncorrelated innovation process with mean zero.

In lag operator notation, $L^i y_t = y_{t-i}$. You can write the ARIMA(*p*,*D*,*q*) model as

$$\phi^*(L) y_t = \phi(L)(1-L)^D y_t = c + \theta(L)\varepsilon_t.$$

Here, $\phi^*(L)$ is an unstable AR operator polynomial with exactly *D* unit roots. You can factor this polynomial as $\phi(L)(1-L)^D$, where $\phi(L) = (1 - \phi_1 L - \ldots - \phi_p L^p)$ is a stable degree *p* AR lag operator polynomial (with all roots lying outside the unit circle). Similarly, $\theta(L) = (1 + \theta_1 L + \ldots + \theta_q L^q)$ is an invertible degree *q* MA lag operator polynomial (with all roots lying outside the unit circle).

The signs of the coefficients in the AR lag operator polynomial, $\phi(L)$, are opposite to the right side of Equation 5-13. When specifying and interpreting AR coefficients in Econometrics Toolbox, use the form in Equation 5-13.

Note: In the original Box-Jenkins methodology, you difference an integrated series until it is stationary before modeling. Then, you model the differenced series as a stationary ARMA(*p*,*q*) process [1]. Econometrics Toolbox fits and forecasts ARIMA(*p*,*D*,*q*) processes

directly, so you do not need to difference data before modeling (or backtransform forecasts).

References

[1] Box, G. E. P., G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis: Forecasting and Control*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

See Also

arima

Related Examples

- "Nonseasonal Differencing" on page 2-18
- "Specify Conditional Mean Models Using arima" on page 5-6
- "ARIMA Model Specifications" on page 5-43

More About

- "Trend-Stationary vs. Difference-Stationary Processes" on page 2-7
- "Autoregressive Moving Average Model" on page 5-34
- "Multiplicative ARIMA Model" on page 5-46

ARIMA Model Specifications

In this section...

Default ARIMA Model

This example shows how to use the shorthand `arima(p,D,q)` syntax to specify the default ARIMA(p, D, q) model,

$$\Delta^D y_t = c + \phi_1 \Delta^D y_{t-1} + \ldots + \phi_p \Delta^D y_{t-p} + \varepsilon_t + \theta_1 \varepsilon_{t-1} + \ldots + \theta_q \varepsilon_{t-q},$$

where $\Delta^D y_t$ is a D^{th} differenced time series. You can write this model in condensed form using lag operator notation:

$$\phi(L)(1-L)^D y_t = c + \theta(L)\varepsilon_t.$$

By default, all parameters in the created model object have unknown values, and the innovation distribution is Gaussian with constant variance.

Specify the default ARIMA(1,1,1) model:

```
model = arima(1,1,1)
```

```
model =

    ARIMA(1,1,1) Model:
    --------------------
    Distribution: Name = 'Gaussian'
               P: 2
               D: 1
               Q: 1
        Constant: NaN
              AR: {NaN} at Lags [1]
             SAR: {}
              MA: {NaN} at Lags [1]
             SMA: {}
```

```
        Variance: NaN
```

The output shows that the created model object, `model`, has `NaN` values for all model parameters: the constant term, the AR and MA coefficients, and the variance. You can modify the created model using dot notation, or input it (along with data) to `estimate`.

The property `P` has value 2 ($p + D$). This is the number of presample observations needed to initialize the AR model.

ARIMA Model with Known Parameter Values

This example shows how to specify an ARIMA(p, D, q) model with known parameter values. You can use such a fully specified model as an input to `simulate` or `forecast`.

Specify the ARIMA(2,1,1) model

$$\Delta y_t = 0.4 + 0.8\Delta y_{t-1} - 0.3\Delta y_{t-2} + \varepsilon_t + 0.5\varepsilon_{t-1},$$

where the innovation distribution is Student's t with 10 degrees of freedom, and constant variance 0.15.

```
tdist = struct('Name','t','DoF',10);
model = arima('Constant',0.4,'AR',{0.8,-0.3},'MA',0.5,...
      'D',1,'Distribution',tdist,'Variance',0.15)
```

```
model =

    ARIMA(2,1,1) Model:
    --------------------
    Distribution: Name = 't', DoF = 10
               P: 3
               D: 1
               Q: 1
        Constant: 0.4
              AR: {0.8 -0.3} at Lags [1 2]
             SAR: {}
              MA: {0.5} at Lags [1]
             SMA: {}
        Variance: 0.15
```

The name-value pair argument `D` specifies the degree of nonseasonal integration (D).

Because all parameter values are specified, the created model object has no `NaN` values. The functions `simulate` and `forecast` don't accept input models with `NaN` values.

See Also

arima | estimate | forecast | simulate

Related Examples

- "Specify Conditional Mean Models Using arima" on page 5-6
- "Modify Properties of Conditional Mean Model Objects" on page 5-65
- "Specify Conditional Mean Model Innovation Distribution" on page 5-72

More About

- "ARIMA Model" on page 5-41
- "Lag Operator Notation" on page 1-22

Multiplicative ARIMA Model

Many time series collected periodically (e.g., quarterly or monthly) exhibit a seasonal trend, meaning there is a relationship between observations made during the same period in successive years. In addition to this seasonal relationship, there can also be a relationship between observations made during successive periods. The multiplicative ARIMA model is an extension of the ARIMA model that addresses seasonality and potential seasonal unit roots [1].

In lag operator polynomial notation, $L^i y_t = y_{t-i}$. For a series with periodicity s, the multiplicative ARIMA$(p,D,q)\times(p_s,D_s,q_s)_s$ is given by

$$\phi(L)\Phi(L)(1-L)^D(1-L^s)^{D_s}y_t = c+\theta(L)\Theta(L)\varepsilon_t.$$

Here, the stable, degree p AR operator polynomial $\phi(L) = (1-\phi_1 L-\ldots-\phi_p L^p)$, and $\Phi(L)$ is a stable, degree p_s AR operator of the same form. Similarly, the invertible, degree q MA operator polynomial $\theta_q(L) = (1+\theta_1 L+\ldots+\theta_q L^q)$, and $\Theta(L)$ is an invertible, degree q_s MA operator of the same form.

When you specify a multiplicative ARIMA model using `arima`,

- Set the nonseasonal and seasonal AR coefficients with the opposite signs from their respective AR operator polynomials. That is, specify the coefficients as they would appear on the right side of Equation 5-15.
- Set the lags associated with the seasonal polynomials in the periodicity of the observed data (e.g., 4, 8,... for quarterly data, or 12, 24,... for monthly data), and not as multiples of the seasonality (e.g., 1, 2,...). This convention does not conform to standard Box and Jenkins notation, but is a more flexible approach for incorporating multiplicative seasonality.

The nonseasonal differencing operator, $(1-L)^D$ accounts for nonstationarity in observations made in successive periods. The seasonal differencing operator, $(1-L^s)^{D_s}$, accounts for nonstationarity in observations made in the same period in successive years. Econometrics Toolbox supports only the degrees of seasonal integration $D_s = 0$ or 1. When you specify $s \geq 0$, Econometrics Toolbox sets $D_s = 1$. $D_s = 0$ otherwise.

References

[1] Box, G. E. P., G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis: Forecasting and Control*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

See Also

arima

Related Examples

- "Nonseasonal and Seasonal Differencing" on page 2-23
- "Multiplicative ARIMA Model Specifications" on page 5-48
- "Specify Conditional Mean Models Using arima" on page 5-6
- "Specify Multiplicative ARIMA Model" on page 5-52
- "Model Seasonal Lag Effects Using Indicator Variables" on page 5-117

More About

- "Autoregressive Moving Average Model" on page 5-34
- "ARIMA Model" on page 5-41

Multiplicative ARIMA Model Specifications

In this section...

Seasonal ARIMA Model with No Constant Term

This example shows how to use `arima` to specify a multiplicative seasonal ARIMA model (for monthly data) with no constant term.

Specify a multiplicative seasonal ARIMA model with no constant term,

$$(1-\phi_1 L)(1-\Phi_{12}L^{12})(1-L)^1(1-L^{12})y_t = (1+\theta_1 L)(1+\Theta_{12}L^{12})\varepsilon_t,$$

where the innovation distribution is Gaussian with constant variance. Here, $(1-L)^1$ is the first degree nonseasonal differencing operator and $(1-L^{12})$ is the first degree seasonal differencing operator with periodicity 12.

```
model = arima('Constant',0,'ARLags',1,'SARLags',12,'D',1,...
            'Seasonality',12,'MALags',1,'SMALags',12)
```

```
model =

    ARIMA(1,1,1) Model Seasonally Integrated with Seasonal AR(12) and MA(12):
    --------------------------------------------------------------------------
    Distribution: Name = 'Gaussian'
               P: 26
               D: 1
               Q: 13
        Constant: 0
              AR: {NaN} at Lags [1]
             SAR: {NaN} at Lags [12]
              MA: {NaN} at Lags [1]
             SMA: {NaN} at Lags [12]
     Seasonality: 12
        Variance: NaN
```

The name-value pair argument `ARLags` specifies the lag corresponding to the nonseasonal AR coefficient, ϕ_1. `SARLags` specifies the lag corresponding to the seasonal

AR coefficient, here at lag 12. The nonseasonal and seasonal MA coefficients are specified similarly. `D` specifies the degree of nonseasonal integration. `Seasonality` specifies the periodicity of the time series, for example `Seasonality` = 12 indicates monthly data. Since `Seasonality` is greater than 0, the degree of seasonal integration D_s is one.

Whenever you include seasonal AR or MA polynomials (signaled by specifying `SAR` or `SMA`) in the model specification, `arima` incorporates them multiplicatively. `arima` sets the property `P` equal to $p + D + p_s + s$ (here, 1 + 1 + 12 + 12 = 26). Similarly, `arima` sets the property `Q` equal to $q + q_s$ (here, 1 + 12 = 13).

Display the value of `SAR`:

```
model.SAR
```

```
ans =

  Columns 1 through 11

    [0]    [0]    [0]    [0]    [0]    [0]    [0]    [0]    [0]    [0]    [0]

  Column 12

    [NaN]
```

The `SAR` cell array returns 12 elements, as specified by `SARLags`. `arima` sets the coefficients at interim lags equal to zero to maintain consistency with MATLAB® cell array indexing. Therefore, the only nonzero coefficient corresponds to lag 12.

All of the other elements in `model` have value `NaN`, indicating that these coefficients need to be estimated or otherwise specified by the user.

Seasonal ARIMA Model with Known Parameter Values

This example shows how to specify a multiplicative seasonal ARIMA model (for quarterly data) with known parameter values. You can use such a fully specified model as an input to `simulate` or `forecast`.

Specify the multiplicative seasonal ARIMA model

$$(1 - .5L)(1 + 0.7L^4)(1 - L)^1(1 - L^4)y_t = (1 + .3L)(1 - .2L^4)\varepsilon_t,$$

where the innovation distribution is Gaussian with constant variance 0.15. Here, $(1-L)^1$ is the nonseasonal differencing operator and $(1-L^4)$ is the first degree seasonal differencing operator with periodicity 4.

```
model = arima('Constant',0,'AR',{0.5},'SAR',-0.7,'SARLags',...
     4,'D',1,'Seasonality',4,'MA',0.3,'SMA',-0.2,...
     'SMALags',4,'Variance',0.15)
```

```
model =

    ARIMA(1,1,1) Model Seasonally Integrated with Seasonal AR(4) and MA(4):
    --------------------------------------------------------------------------
    Distribution: Name = 'Gaussian'
               P: 10
               D: 1
               Q: 5
        Constant: 0
              AR: {0.5} at Lags [1]
             SAR: {-0.7} at Lags [4]
              MA: {0.3} at Lags [1]
             SMA: {-0.2} at Lags [4]
     Seasonality: 4
        Variance: 0.15
```

The output specifies the nonseasonal and seasonal AR coefficients with opposite signs compared to the lag polynomials. This is consistent with the difference equation form of the model. The output specifies the lags of the seasonal AR and MA coefficients using `SARLags` and `SMALags`, respectively. `D` specifies the degree of nonseasonal integration. `Seasonality` = 4 specifies quarterly data with one degree of seasonal integration.

All of the parameters in the model have a value. Therefore, the model does not contain any `NaN` values. The functions `simulate` and `forecast` *do not* accept input models with `NaN` values.

See Also

arima | estimate | forecast | simulate

Related Examples

- "Specify Multiplicative ARIMA Model" on page 5-52
- "Modify Properties of Conditional Mean Model Objects" on page 5-65

- "Specify Conditional Mean Model Innovation Distribution" on page 5-72
- "Model Seasonal Lag Effects Using Indicator Variables" on page 5-117

More About

- "Specify Conditional Mean Models Using arima" on page 5-6
- "Multiplicative ARIMA Model" on page 5-46

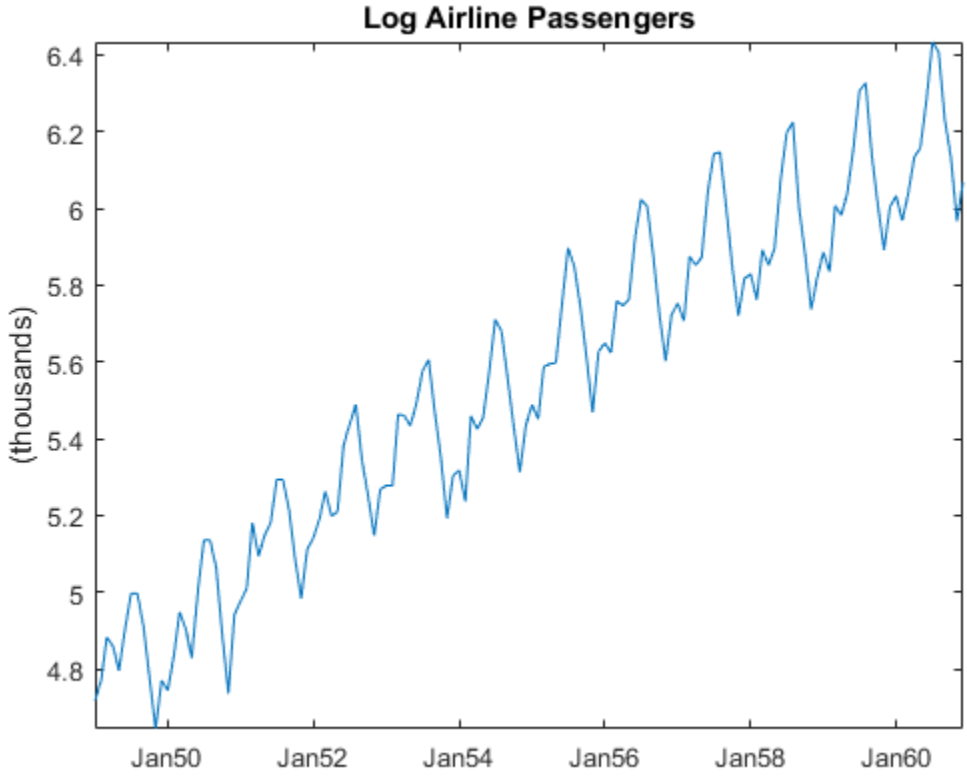
Specify Multiplicative ARIMA Model

This example shows how to specify a seasonal ARIMA model using `arima`. The time series is monthly international airline passenger numbers from 1949 to 1960.

Load the airline passenger data.

Load the airline data set, and then plot the natural log of the monthly passenger totals.

```
load(fullfile(matlabroot,'examples','econ','Data_Airline.mat'))
y = log(Data);
T = length(y);

figure
plot(dates,y)
xlim([1,T])
datetick('x','mmmyy')
axis tight
title('Log Airline Passengers')
ylabel('(thousands)')
```



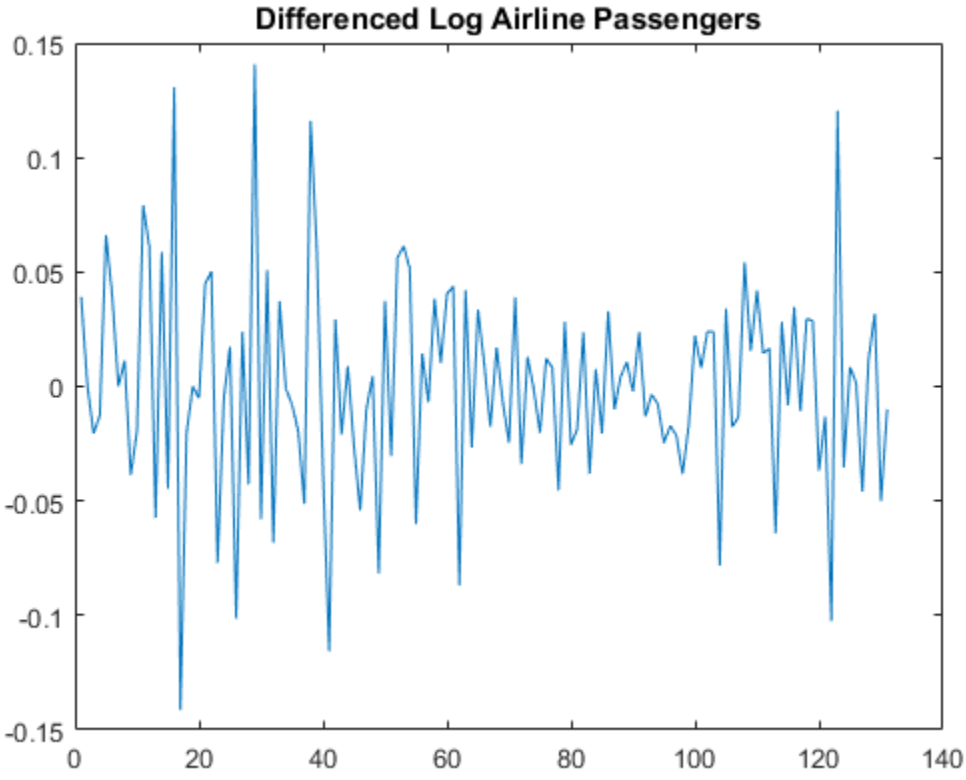
The data look nonstationary, with a linear trend and seasonal periodicity.

Plot the seasonally integrated series.

Calculate the differenced series, $(1-L)(1-L^{12})y_t$, where y_t is the original log-transformed data. Plot the differenced series.

```
A1 = LagOp({1,-1},'Lags',[0,1]);
A12 = LagOp({1,-1},'Lags',[0,12]);
dY = filter(A1*A12,y);

figure
plot(dY)
```

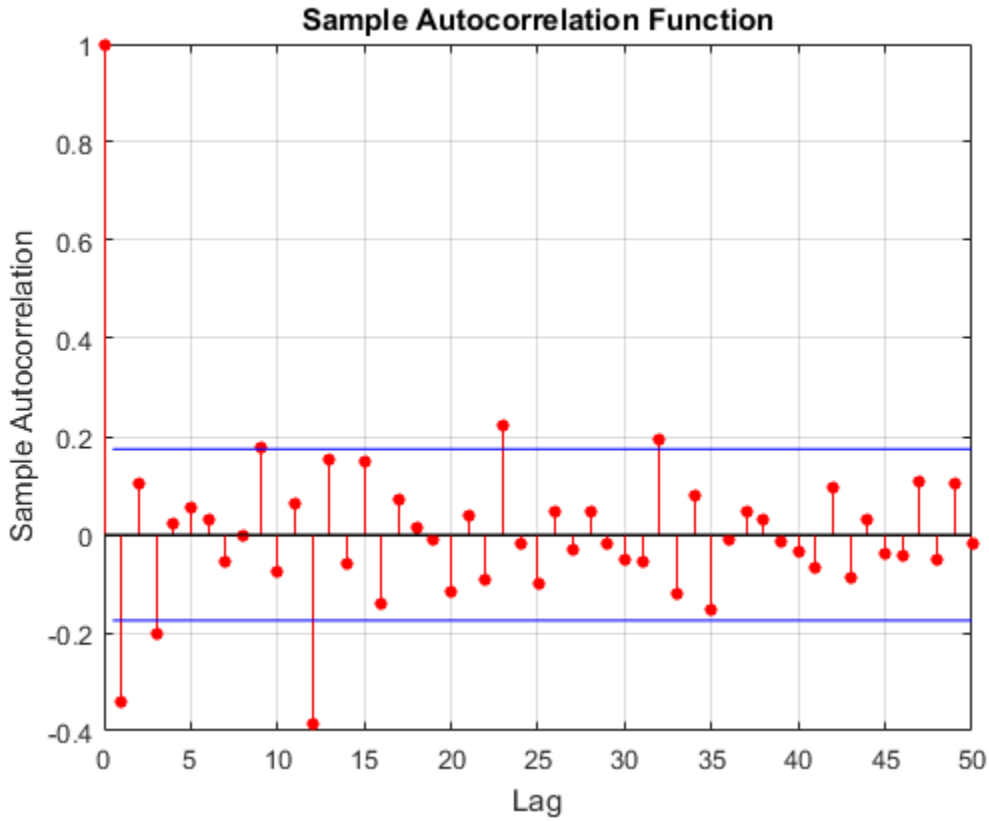
```
title('Differenced Log Airline Passengers')
```



The differenced series appears stationary.

Plot the sample autocorrelation function (ACF).

```
figure
autocorr(dY,50)
```



The sample ACF of the differenced series shows significant autocorrelation at lags that are multiples of 12. There is also potentially significant autocorrelation at smaller lags.

Specify a seasonal ARIMA model.

Box, Jenkins, and Reinsel suggest the multiplicative seasonal model,

$$(1-L)(1-L^{12})y_t = (1-\theta_1)(1-\Theta_{12})\varepsilon_t,$$

`for this data set (Box et al., 1994).`

Specify this model.

```
Mdl = arima('Constant',0,'D',1,'Seasonality',12,...
    'MALags',1,'SMALags',12)


Mdl =

    ARIMA(0,1,1) Model Seasonally Integrated with Seasonal MA(12):
    --------------------------------------------------------------
    Distribution: Name = 'Gaussian'
               P: 13
               D: 1
               Q: 13
        Constant: 0
              AR: {}
             SAR: {}
              MA: {NaN} at Lags [1]
             SMA: {NaN} at Lags [12]
     Seasonality: 12
        Variance: NaN
```

The property `P` is equal to `13`, corresponding to the sum of the nonseasonal and seasonal differencing degrees (1 + 12). The property `Q` is also equal to `13`, corresponding to the sum of the degrees of the nonseasonal and seasonal MA polynomials (1 + 12). Parameters that need to be estimated have value `NaN`.

References:

Box, G. E. P., G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis: Forecasting and Control*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

See Also

arima | LagOp | `autocorr` | filter

Related Examples

- “Estimate Multiplicative ARIMA Model” on page 5-113
- “Simulate Multiplicative ARIMA Models” on page 5-169
- “Forecast Multiplicative ARIMA Model” on page 5-192
- “Check Fit of Multiplicative ARIMA Model” on page 3-81
- “Model Seasonal Lag Effects Using Indicator Variables” on page 5-117

More About

- "Multiplicative ARIMA Model" on page 5-46

ARIMA Model Including Exogenous Covariates

In this section...

ARIMAX(*p*,*D*,*q*) Model

The autoregressive moving average model including exogenous covariates, ARMAX(*p*,*q*), extends the "Autoregressive Moving Average Model" on page 5-34 model by including the linear effect that one or more exogenous series has on the "Stationary Processes" on page 1-21 response series y_t. The general form of the ARMAX(*p*,*q*) model is

$$y_t = \sum_{i=1}^{p} \phi_i y_{t-i} + \sum_{k=1}^{r} \beta_k x_{tk} + \varepsilon_t + \sum_{j=1}^{q} \theta_j \varepsilon_{t-j},$$

and it has the following condensed form in lag operator notation:

$$\phi(L) y_t = c + x_t' \beta + \theta(L) \varepsilon_t.$$

In Equation 5-17, the vector x_t' holds the values of the *r* exogenous, time-varying predictors at time *t*, with coefficients denoted β.

You can use this model to check if a set of exogenous variables has an effect on a linear time series. For example, suppose you want to measure how the previous week's average price of oil, x_t, affects this week's United States exchange rate y_t. The exchange rate and the price of oil are time series, so an ARMAX model can be appropriate to study their relationships.

Conventions and Extensions of the ARIMAX Model

- ARMAX models have the same stationarity requirements as "Autoregressive Moving Average Model" on page 5-34. Specifically, the response series is *stable* if the roots of the homogeneous "Characteristic Equation" on page 1-23 of $\phi(L) = L^p - \phi_1 L^{p-1} - \phi_2 L^{p-2} - \ldots - \phi_p L^p = 0$ lie outside of the unit circle according to Wold's Decomposition [1].

If the response series y_t is not stable, then you can difference it to form a stationary "ARIMA Model" on page 5-41. Do this by specifying the degrees of integration `D`. Econometrics Toolbox enforces stability of the AR polynomial. When you specify an AR model using `arima`, the software displays an error if you enter coefficients that do not correspond to a stable polynomial. Similarly, `estimate` imposes stationarity constraints during estimation.

- The software differences the response series y_t *before* including the exogenous covariates if you specify the degree of integration `D`. In other words, the exogenous covariates enter a model with a *stationary response*. Therefore, the ARIMAX(p,D,q) model is

  $$\phi(L)y_t = c^* + x_t'\beta + \theta^*(L)\varepsilon_t,$$

  where $c^* = c/(1-L)^D$ and $\theta^*(L) = \theta(L)/(1-L)^D$. Subsequently, the interpretation of β has changed to the expected effect a unit increase in the predictor has on the *difference* between current and lagged values of the response (conditional on those lagged values).
- You should assess whether the predictor series x_t are stationary. Difference all predictor series that are not stationary with `diff` during the data preprocessing stage. If x_t is nonstationary, then a test for the significance of β can produce a false negative. The practical interpretation of β changes if you difference the predictor series.
- The software uses "Maximum Likelihood Estimation for Conditional Mean Models" on page 5-98 such as ARIMAX models. You can specify either a Gaussian or Student's t for the distribution of the innovations.
- You can include seasonal components in an ARIMAX model (see "Multiplicative ARIMA Model" on page 5-46) which creates a SARIMAX(p,D,q)(p_s,D_s,q_s)$_s$ model. Assuming that the response series y_t is stationary, the model has the form

  $$\phi(L)\Phi(L)y_t = c + x_t'\beta + \theta(L)\Theta(L)\varepsilon_t,$$

  where $\Phi(L)$ and $\Theta(L)$ are the seasonal lag polynomials. If y_t is not stationary, then you can specify degrees of nonseasonal or seasonal integration using `arima`. If you specify `Seasonality` ≥ 0, then the software applies degree one seasonal differencing (D_s = 1) to the response. Otherwise, $D_s = 0$. The software includes the exogenous covariates after it differences the response.

- The software treats the exogenous covariates as fixed during estimation and inference.

References

[1] Wold, H. *A Study in the Analysis of Stationary Time Series*. Uppsala, Sweden: Almqvist & Wiksell, 1938.

See Also

arima

Related Examples

- "Specify ARMAX Model Using Dot Notation" on page 5-62
- "Specify ARIMAX Model Using Name-Value Pairs" on page 5-61

More About

- "Autoregressive Moving Average Model" on page 5-34
- "Specify Conditional Mean Models Using arima" on page 5-6

ARIMAX Model Specifications

In this section...

Specify ARIMAX Model Using Name-Value Pairs

This example shows how to specify an ARIMAX model using `arima`.

Specify the ARIMAX(1,1,0) model that includes three predictors:

$$(1-0.1L)(1-L)^1 y_t = x_t' \begin{bmatrix} 3 & -2 & 5 \end{bmatrix}' + \varepsilon_t.$$

```
model = arima('AR',0.1,'D',1,'Beta',[3 -2 5])
```

```
model =

    ARIMAX(1,1,0) Model:
    --------------------
    Distribution: Name = 'Gaussian'
               P: 2
               D: 1
               Q: 0
        Constant: NaN
              AR: {0.1} at Lags [1]
             SAR: {}
              MA: {}
             SMA: {}
            Beta: [3 -2 5]
        Variance: NaN
```

The output shows that the ARIMAX model, `model`, has the following qualities:

- Property `P` in the output is the sum of the autoregressive lags and the degree of integration, i.e., `P = p + D = 2`.
- `Beta` contains three coefficients corresponding to the effect that the predictors have on the response.

- The rest of the properties are 0, `NaN`, or empty cells.

Be aware that if you specify nonzero `D` or `Seasonality`, then Econometrics Toolbox™ differences the response series y_t before the predictors enter the model. Therefore, the predictors enter a stationary model with respect to the response series y_t. You should preprocess the predictors x_t by testing for stationarity and differencing if any are unit root nonstationary. If any nonstationary predictor enters the model, then the false negative rate for significance tests of β can increase.

Specify ARMAX Model Using Dot Notation

This example shows how to specify a stationary ARMAX model using `arima`.

Specify the ARMAX(2,1) model

$$y_t = 6 + 0.2y_{t-1} - 0.3y_{t-2} + 3x_t + \varepsilon_t + 0.1\varepsilon_{t-1}$$

by including one stationary exogenous covariate in `arima`.

```
model = arima('AR',[0.2 -0.3],'MA',0.1,'Constant',6,'Beta',3)
```

```
model =

    ARIMAX(2,0,1) Model:
    --------------------
    Distribution: Name = 'Gaussian'
               P: 2
               D: 0
               Q: 1
        Constant: 6
              AR: {0.2 -0.3} at Lags [1 2]
             SAR: {}
              MA: {0.1} at Lags [1]
             SMA: {}
            Beta: [3]
        Variance: NaN
```

The output shows the model that you created, `model`, has `NaN` values or an empty cell (`{}`) for the `Variance`, `SAR`, and `SMA` properties. You can modify it using dot notation. For example, you can introduce another exogenous, stationary covariate, and specify that the variance of the innovations as 0.1:

$$y_t = 6 + 0.2y_{t-1} - 0.3y_{t-2} + x_t' \begin{bmatrix} 3 \\ -2 \end{bmatrix} + \varepsilon_t + 0.1\varepsilon_{t-1};\ \ \varepsilon_t \sim N(0, 0.1).$$

Modify `model`:

```
model.Beta=[3 -2];
model.Variance=0.1
```

```
model =

    ARIMAX(2,0,1) Model:
    --------------------
    Distribution: Name = 'Gaussian'
               P: 2
               D: 0
               Q: 1
        Constant: 6
              AR: {0.2 -0.3} at Lags [1 2]
             SAR: {}
              MA: {0.1} at Lags [1]
             SMA: {}
            Beta: [3 -2]
        Variance: 0.1
```

See Also

arima | estimate | forecast | simulate | `struct`

Related Examples

- "Specify Conditional Mean and Variance Models" on page 5-79
- "Specify Nonseasonal Models Using Name-Value Pairs" on page 5-8
- "Specify Multiplicative ARIMA Model" on page 5-52

More About

- "Specify Conditional Mean Models Using arima" on page 5-6
- "Modify Properties of Conditional Mean Model Objects" on page 5-65
- "Autoregressive Moving Average Model" on page 5-34
- "ARIMA Model" on page 5-41

- “ARIMA Model Including Exogenous Covariates” on page 5-58

Modify Properties of Conditional Mean Model Objects

In this section...

Dot Notation

A model created by `arima` has values assigned to all model properties. To change any of these property values, you do not need to reconstruct the whole model. You can modify property values of an existing model using dot notation. That is, type the model name, then the property name, separated by `'.'` (a period).

For example, start with this model specification:

```
Mdl = arima(2,0,0)
```

```
Mdl =

    ARIMA(2,0,0) Model:
    --------------------
    Distribution: Name = 'Gaussian'
               P: 2
               D: 0
               Q: 0
        Constant: NaN
              AR: {NaN NaN} at Lags [1 2]
             SAR: {}
              MA: {}
             SMA: {}
        Variance: NaN
```

Modify the model to remove the constant term:

```
Mdl.Constant = 0
```

```
Mdl =

    ARIMA(2,0,0) Model:
```

```
    --------------------
    Distribution: Name = 'Gaussian'
               P: 2
               D: 0
               Q: 0
        Constant: 0
              AR: {NaN NaN} at Lags [1 2]
             SAR: {}
              MA: {}
             SMA: {}
        Variance: NaN
```

The updated constant term now appears in the model output.

Be aware that every model property has a data type. Any modifications you make to a property value must be consistent with the data type of the property. For example, AR, MA, SAR, and SMA are all cell vectors. This mean you must index them using cell array syntax.

For example, start with the following model:

```
Mdl = arima(2,0,0)
```

```
Mdl =

    ARIMA(2,0,0) Model:
    --------------------
    Distribution: Name = 'Gaussian'
               P: 2
               D: 0
               Q: 0
        Constant: NaN
              AR: {NaN NaN} at Lags [1 2]
             SAR: {}
              MA: {}
             SMA: {}
        Variance: NaN
```

To modify the property value of AR, assign AR a cell array. Here, assign known AR coefficient values:

```
Mdl.AR  = {0.8,-0.4}
```

```
Mdl =

    ARIMA(2,0,0) Model:
    --------------------
    Distribution: Name = 'Gaussian'
               P: 2
               D: 0
               Q: 0
        Constant: NaN
              AR: {0.8 -0.4} at Lags [1 2]
             SAR: {}
              MA: {}
             SMA: {}
        Variance: NaN
```

The updated model now has AR coefficients with the specified equality constraints.

Similarly, the data type of `Distribution` is a data structure. The default data structure has only one field, `Name`, with value `'Gaussian'`.

```
Distribution = Mdl.Distribution
```

```
Distribution =

    Name: 'Gaussian'
```

To modify the innovation distribution, assign `Distribution` a new name or data structure. The data structure can have up to two fields, `Name` and `DoF`. The second field corresponds to the degrees of freedom for a Student's t distribution, and is only required if `Name` has the value `'t'`.

To specify a Student's t distribution with unknown degrees of freedom, enter:

```
Mdl.Distribution = 't'
```

```
Mdl =

    ARIMA(2,0,0) Model:
    --------------------
    Distribution: Name = 't', DoF = NaN
               P: 2
               D: 0
```

```
           Q: 0
    Constant: NaN
          AR: {0.8 -0.4} at Lags [1 2]
         SAR: {}
          MA: {}
         SMA: {}
    Variance: NaN
```

The updated model has a Student's t distribution with `NaN` degrees of freedom. To specify a t distribution with eight degrees of freedom, say:

```
Mdl.Distribution = struct('Name','t','DoF',8)
```

```
Mdl =

    ARIMA(2,0,0) Model:
    --------------------
    Distribution: Name = 't', DoF = 8
               P: 2
               D: 0
               Q: 0
        Constant: NaN
              AR: {0.8 -0.4} at Lags [1 2]
             SAR: {}
              MA: {}
             SMA: {}
        Variance: NaN
```

The degrees of freedom property of the model is updated. Note that the `DoF` field of `Distribution` is not directly assignable. For example, `Mdl.Distribution.DoF = 8` is not a valid assignment. However, you can get the individual fields:

```
Mdl.Distribution.DoF
```

```
ans =

     8
```

You can modify `Mdl` to include, for example, two coefficients $\beta_1 = 0.2$ and $\beta_2 = 4$ corresponding to two predictor series. Since `Beta` has not been specified yet, you have not seen it in the output. To include it, enter:

```
Mdl.Beta=[0.2 4]
```

```
Mdl =

    ARIMAX(2,0,0) Model:
    --------------------
    Distribution: Name = 't', DoF = 8
               P: 2
               D: 0
               Q: 0
        Constant: NaN
              AR: {0.8 -0.4} at Lags [1 2]
             SAR: {}
              MA: {}
             SMA: {}
            Beta: [0.2 4]
        Variance: NaN
```

Nonmodifiable Properties

Not all model properties are modifiable. You cannot change these properties in an existing model:

- `P`. This property updates automatically when any of p (degree of the nonseasonal AR operator), p_s (degree of the seasonal AR operator), D (degree of nonseasonal differencing), or s (degree of seasonal differencing) changes.
- `Q`. This property updates automatically when either q (degree of the nonseasonal MA operator), or q_s (degree of the seasonal MA operator) changes.

Not all name-value pair arguments you can use for model creation are properties of the created model. Specifically, you can specify the arguments `ARLags`, `MALags`, `SARLags`, and `SMALags` during model creation. These are not, however, properties of `arima` models. This means you cannot retrieve or modify them in an existing model.

The nonseasonal and seasonal AR and MA lags update automatically if you add any elements to (or remove from) the coefficient cell arrays `AR`, `MA`, `SAR`, or `SMA`.

For example, specify an AR(2) model:

```
Mdl = arima(2,0,0)
```

```
Mdl =

    ARIMA(2,0,0) Model:
    --------------------
    Distribution: Name = 'Gaussian'
               P: 2
               D: 0
               Q: 0
        Constant: NaN
              AR: {NaN NaN} at Lags [1 2]
             SAR: {}
              MA: {}
             SMA: {}
        Variance: NaN
```

The model output shows nonzero AR coefficients at lags 1 and 2.

Add a new AR term at lag 12:

```
Mdl.AR{12} = NaN
```

```
Mdl =

    ARIMA(12,0,0) Model:
    --------------------
    Distribution: Name = 'Gaussian'
               P: 12
               D: 0
               Q: 0
        Constant: NaN
              AR: {NaN NaN NaN} at Lags [1 2 12]
             SAR: {}
              MA: {}
             SMA: {}
        Variance: NaN
```

The three nonzero coefficients at lags 1, 2, and 12 now display in the model output. However, the cell array assigned to `AR` returns twelve elements:

```
Mdl.AR
```

```
ans =
```

```
Columns 1 through 10

  [NaN]    [NaN]    [0]    [0]    [0]    [0]    [0]    [0]    [0]    [0]

Columns 11 through 12

  [0]    [NaN]
```

AR has zero coefficients at all the interim lags to maintain consistency with traditional MATLAB® cell array indexing.

See Also

arima | `struct`

Related Examples

- "Specify Conditional Mean Models Using arima" on page 5-6
- "Specify Conditional Mean Model Innovation Distribution" on page 5-72

More About

- "Conditional Mean Models" on page 5-3

Specify Conditional Mean Model Innovation Distribution

In this section...

About the Innovation Process

You can express all stationary stochastic processes in the general linear form [1]

$$y_t = \mu + \varepsilon_t + \sum_{i=1}^{\infty} \psi_i \varepsilon_{t-i}.$$

The innovation process, ε_t, is an uncorrelated—but not necessarily independent—mean zero process with a known distribution.

In Econometrics Toolbox, the general form for the innovation process is $\varepsilon_t = \sigma_t z_t$. Here, z_t is an independent and identically distributed (iid) series with mean 0 and variance 1, and σ_t^2 is the variance of the innovation process at time t. Thus, ε_t is an uncorrelated series with mean 0 and variance σ_t^2.

`arima` model objects have two properties for storing information about the innovation process:

- `Variance` stores the form of σ_t^2
- `Distribution` stores the parametric form of the distribution of z_t

Choices for the Variance Model

- If $\sigma_t^2 = \sigma_\varepsilon^2$ for all times t, then ε_t is an independent process with constant variance, σ_ε^2.

  The default value for `Variance` is `NaN`, meaning constant variance with unknown value. You can alternatively assign `Variance` any positive scalar value, or estimate it using `estimate`.

- A time series can exhibit *volatility clustering*, meaning a tendency for large changes to follow large changes, and small changes to follow small changes. You can model this behavior with a conditional variance model—a dynamic model describing the evolution of the process variance, σ_t^2, conditional on past innovations and variances.

  Set `Variance` equal to one of the three conditional variance model objects available in Econometrics Toolbox (`garch`, `egarch`, or `gjr`). This creates a composite conditional mean and variance model variable.

Choices for the Innovation Distribution

The available distributions for z_t are:

- Standardized Gaussian
- Standardized Student's t with $\nu > 2$ degrees of freedom,

  $$z_t = \sqrt{\frac{\nu - 2}{\nu}} T_\nu,$$

  where T_ν follows a Student's t distribution with $\nu > 2$ degrees of freedom.

The t distribution is useful for modeling time series with more extreme values than expected under a Gaussian distribution. Series with larger values than expected under normality are said to have *excess kurtosis*.

Tip It is good practice to assess the distributional properties of model residuals to determine if a Gaussian innovation distribution (the default distribution) is appropriate for your data.

Specify the Innovation Distribution

The property `Distribution` in a model stores the distribution name (and degrees of freedom for the *t* distribution). The data type of `Distribution` is a `struct` array. For a Gaussian innovation distribution, the data structure has only one field: `Name`. For a Student's *t* distribution, the data structure must have two fields:

- `Name`, with value `'t'`
- `DoF`, with a scalar value larger than two (`NaN` is the default value)

If the innovation distribution is Gaussian, you do not need to assign a value to `Distribution`. `arima` creates the required data structure.

To illustrate, consider specifying an MA(2) model with an iid Gaussian innovation process:

```
Mdl = arima(0,0,2)
```

```
Mdl =

    ARIMA(0,0,2) Model:
    --------------------
    Distribution: Name = 'Gaussian'
               P: 0
               D: 0
               Q: 2
        Constant: NaN
              AR: {}
             SAR: {}
              MA: {NaN NaN} at Lags [1 2]
             SMA: {}
        Variance: NaN
```

The model output shows that `Distribution` is a `struct` array with one field, `Name`, with the value `'Gaussian'`.

When specifying a Student's *t* innovation distribution, you can specify the distribution with either unknown or known degrees of freedom. If the degrees of freedom are unknown, you can simply assign `Distribution` the value `'t'`. By default, the property `Distribution` has a data structure with field `Name` equal to `'t'`, and field `DoF` equal to `NaN`. When you input the model to `estimate`, the degrees of freedom are estimated along with any other unknown model parameters.

For example, specify an MA(2) model with an iid Student's t innovation distribution, with unknown degrees of freedom:

```
Mdl = arima('MALags',1:2,'Distribution','t')
```

```
Mdl =

    ARIMA(0,0,2) Model:
    --------------------
    Distribution: Name = 't', DoF = NaN
               P: 0
               D: 0
               Q: 2
        Constant: NaN
              AR: {}
             SAR: {}
              MA: {NaN NaN} at Lags [1 2]
             SMA: {}
        Variance: NaN
```

The output shows that `Distribution` is a data structure with two fields. Field `Name` has the value `'t'`, and field `DoF` has the value `NaN`.

If the degrees of freedom are known, and you want to set an equality constraint, assign a `struct` array to `Distribution` with fields `Name` and `DoF`. In this case, if the model is input to `estimate`, the degrees of freedom won't be estimated (the equality constraint is upheld).

Specify an MA(2) model with an iid Student's t innovation process with eight degrees of freedom:

```
Mdl = arima('MALags',1:2,'Distribution',struct('Name','t','DoF',8))
```

```
Mdl =

    ARIMA(0,0,2) Model:
    --------------------
    Distribution: Name = 't', DoF = 8
               P: 0
               D: 0
               Q: 2
        Constant: NaN
```

```
              AR: {}
             SAR: {}
              MA: {NaN NaN} at Lags [1 2]
             SMA: {}
        Variance: NaN
```

The output shows the specified innovation distribution.

Modify the Innovation Distribution

After a model exists in the Workspace, you can modify its `Distribution` property using dot notation. You cannot modify the fields of the `Distribution` data structure directly. For example, `Mdl.Distribution.DoF = 8` is not a valid assignment. However, you can get the individual fields.

Start with an MA(2) model:

```
Mdl = arima(0,0,2);
```

To change the distribution of the innovation process in an existing model to a Student's t distribution with unknown degrees of freedom, type:

```
Mdl.Distribution = 't'
```

```
Mdl =

    ARIMA(0,0,2) Model:
    --------------------
    Distribution: Name = 't', DoF = NaN
               P: 0
               D: 0
               Q: 2
        Constant: NaN
              AR: {}
             SAR: {}
              MA: {NaN NaN} at Lags [1 2]
             SMA: {}
        Variance: NaN
```

To change the distribution to a t distribution with known degrees of freedom, use a data structure:

```
Mdl.Distribution = struct('Name','t','DoF',8)
```

```
Mdl =

    ARIMA(0,0,2) Model:
    --------------------
    Distribution: Name = 't', DoF = 8
               P: 0
               D: 0
               Q: 2
        Constant: NaN
              AR: {}
             SAR: {}
              MA: {NaN NaN} at Lags [1 2]
             SMA: {}
        Variance: NaN
```

You can get the individual `Distribution` fields:

```
DistributionDoF = Mdl.Distribution.DoF
```

```
DistributionDoF =

     8
```

To change the innovation distribution from a Student's t back to a Gaussian distribution, type:

```
Mdl.Distribution = 'Gaussian'
```

```
Mdl =

    ARIMA(0,0,2) Model:
    --------------------
    Distribution: Name = 'Gaussian'
               P: 0
               D: 0
               Q: 2
        Constant: NaN
              AR: {}
             SAR: {}
              MA: {NaN NaN} at Lags [1 2]
             SMA: {}
```

```
    Variance: NaN
```

The `Name` field is updated to `'Gaussian'`, and there is no longer a `DoF` field.

References

[1] Wold, H. *A Study in the Analysis of Stationary Time Series*. Uppsala, Sweden: Almqvist and Wiksell, 1938.

See Also

arima | `egarch` | `garch` | `gjr` | `struct`

Related Examples

- "Specify Conditional Mean Models Using arima" on page 5-6
- "Modify Properties of Conditional Mean Model Objects" on page 5-65
- "Specify Conditional Mean and Variance Models" on page 5-79

More About

- "Conditional Mean Models" on page 5-3
- Using garch Objects
- Using egarch Objects
- Using gjr Objects

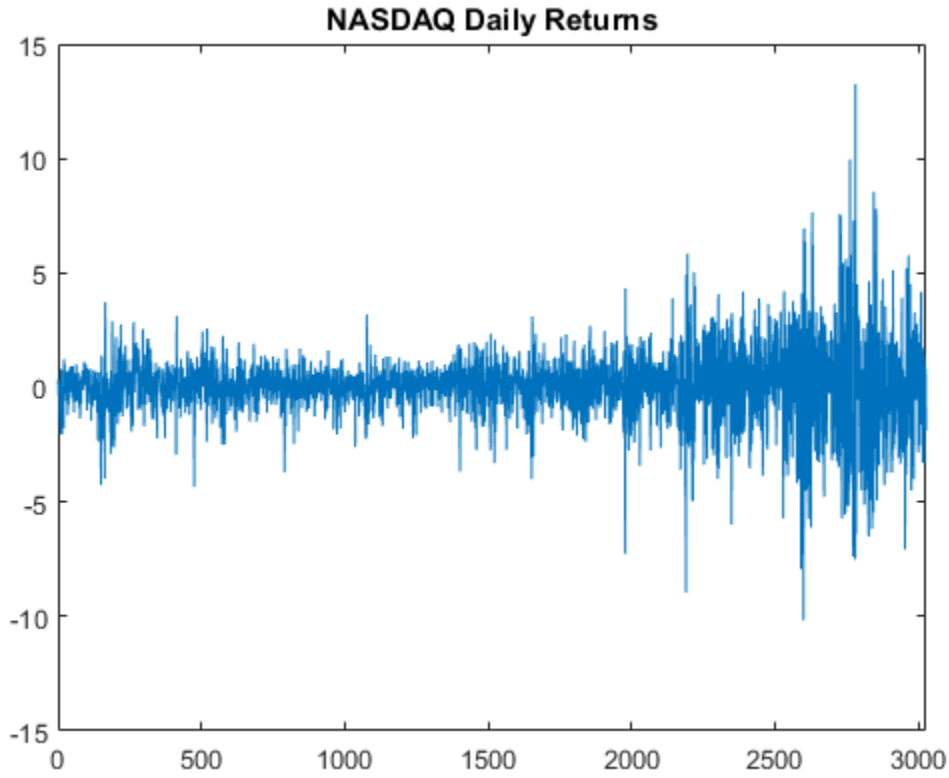
Specify Conditional Mean and Variance Models

This example shows how to specify a composite conditional mean and variance model using `arima`.

Load the Data

Load the NASDAQ data included with the toolbox. Convert the daily close composite index series to a percentage return series.

```
load Data_EquityIdx
nasdaq = DataTable.NASDAQ;
r = 100*price2ret(nasdaq);
T = length(r);

figure
plot(r)
xlim([0 T])
title('NASDAQ Daily Returns')
```



The returns appear to fluctuate around a constant level, but exhibit volatility clustering. Large changes in the returns tend to cluster together, and small changes tend to cluster together. That is, the series exhibits conditional heteroscedasticity.

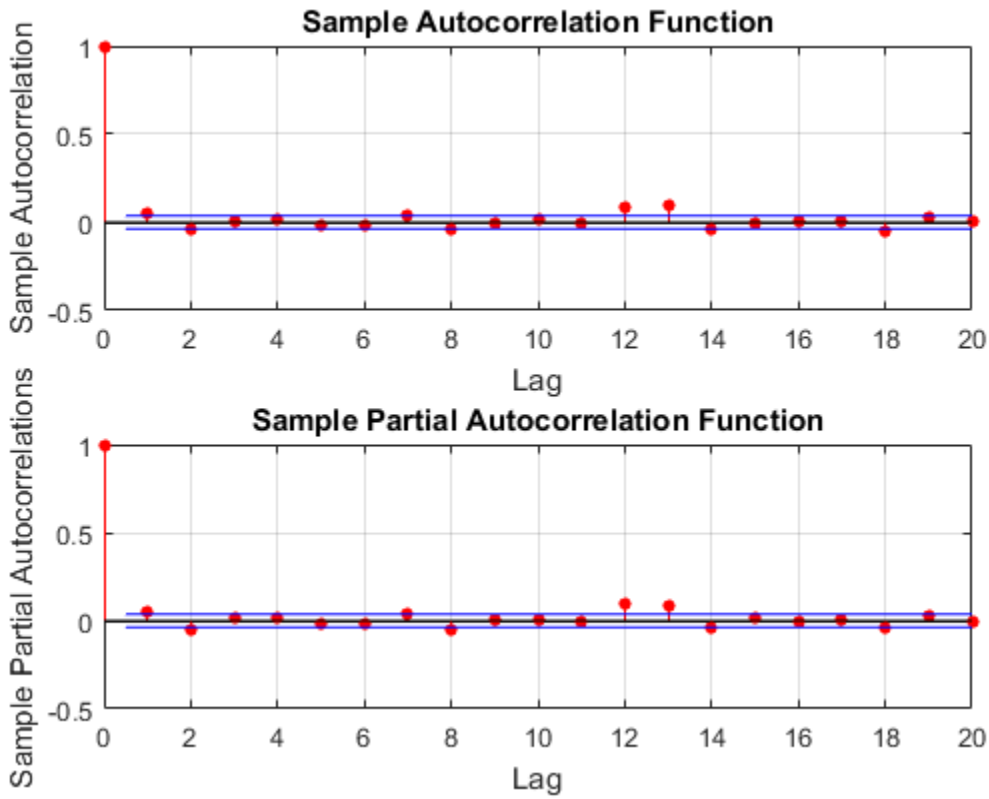
The returns are of relatively high frequency. Therefore, the daily changes can be small. For numerical stability, it is good practice to scale such data.

Check for Autocorrelation

Plot the sample autocorrelation function (ACF) and partial autocorrelation function (PACF) for the return series.

```
figure
subplot(2,1,1)
```

```
autocorr(r)
subplot(2,1,2)
parcorr(r)
```



The autocorrelation functions suggests there is significant autocorrelation at lag one.

Test the Significance of Autocorrelations

Conduct a Ljung-Box Q-test at lag 5.

```
[h,p] = lbqtest(r,'Lags',5)
```

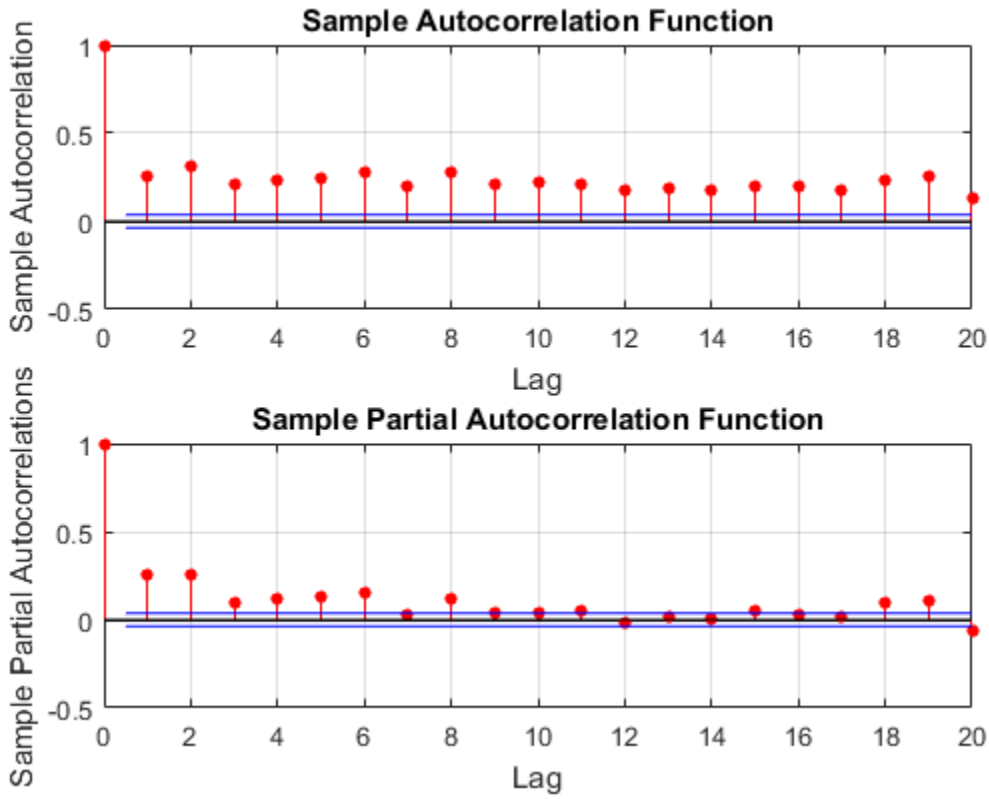
```
h =

     1


p =

    0.0120
```

The null hypothesis that all autocorrelations are 0 up to lag 5 is rejected (`h = 1`).

Check for Conditional Heteroscedasticity.

Plot the sample ACF and PACF of the squared return series.

```
figure
subplot(2,1,1)
autocorr(r.^2)
subplot(2,1,2)
parcorr(r.^2)
```



The autocorrelation functions show significant serial dependence, which suggests that the series is conditionally heteroscedastic.

Test for Significant ARCH Effects

Conduct an Engle's ARCH test. Test the null hypothesis of no conditional heteroscedasticity against the alternative hypothesis of an ARCH model with two lags (which is locally equivalent to a GARCH(1,1) model).

```
[h,p] = archtest(r-mean(r),'lags',2)
```

```
h =
```

```
     1

p =

     0
```

The null hypothesis is rejected in favor of the alternative hypothesis (`h = 1`).

Specify a Conditional Mean and Variance Model.

Specify an AR(1) model for the conditional mean of the NASDAQ returns, and a GARCH(1,1) model for the conditional variance. This is a model of the form

$$r_t = c + \phi_1 r_{t-1} + \varepsilon_t,$$

where $\varepsilon_t = \sigma_t z_t$,

$$\sigma_t^2 = \kappa + \gamma_1 \sigma_{t-1}^2 + \alpha_1 \varepsilon_{t-1}^2,$$

and z_t is an independent and identically distributed standardized Gaussian process.

```
Mdl = arima('ARLags',1,'Variance',garch(1,1))
```

```
Mdl =

    ARIMA(1,0,0) Model:
    -------------------
    Distribution: Name = 'Gaussian'
               P: 1
               D: 0
               Q: 0
        Constant: NaN
              AR: {NaN} at Lags [1]
             SAR: {}
              MA: {}
             SMA: {}
        Variance: [GARCH(1,1) Model]
```

The model output shows that a `garch` model is stored in the `Variance` property of the `arima` model, `Mdl`.

See Also

`archtest` | arima | `autocorr` | `garch` | `lbqtest` | `parcorr`

Related Examples

- "Estimate Conditional Mean and Variance Models" on page 5-129
- "Simulate Conditional Mean and Variance Models" on page 5-175
- "Forecast Conditional Mean and Variance Model" on page 5-197

More About

- "Multiplicative ARIMA Model" on page 5-46
- Using garch Objects

Impulse Response Function

The general linear model for a time series y_t is

$$y_t = \mu + \varepsilon_t + \sum_{i=1}^{\infty} \psi_i \varepsilon_{t-i} = \mu + \psi(L)\varepsilon_t,$$

where $\psi(L)$ denotes the infinite-degree lag operator polynomial $(1 + \psi_1 L + \psi_2 L^2 + \ldots)$.

The coefficients ψ_i are sometimes called *dynamic multipliers* [1]. You can interpret the coefficient ψ_j as the change in y_{t+j} due to a one-unit change in ε_t,

$$\frac{\partial y_{t+j}}{\partial \varepsilon_t} = \psi_j.$$

Provided the series $\{\psi_i\}$ is absolutely summable, Equation 5-19 corresponds to a stationary stochastic process [2]. For a stationary stochastic process, the impact on the process due to a change in ε_t is not permanent, and the effect of the impulse decays to zero. If the series $\{\psi_i\}$ is explosive, the process y_t is nonstationary. In this case, a one-unit change in ε_t permanently affects the process.

The series $\{\psi_i\}$ describes the change in future values y_{t+i} due to a one-unit impulse in the innovation ε_t, with no other changes to future innovations $\varepsilon_{t+1}, \varepsilon_{t+2}, \ldots$. As a result, $\{\psi_i\}$ is often called the *impulse response function*.

References

[1] Hamilton, J. D. *Time Series Analysis*. Princeton, NJ: Princeton University Press, 1994.

[2] Wold, H. *A Study in the Analysis of Stationary Time Series*. Uppsala, Sweden: Almqvist & Wiksell, 1938.

See Also

`armairf` | impulse

Related Examples

- “Plot the Impulse Response Function” on page 5-88

More About

- “Conditional Mean Models” on page 5-3

Plot the Impulse Response Function

In this section...

Moving Average Model

This example shows how to calculate and plot the impulse response function for a moving average (MA) model. The MA(q) model is given by

$$y_t = \mu + \theta(L)\varepsilon_t,$$

where $\theta(L)$ is a q-degree MA operator polynomial, $(1 + \theta_1 L + \ldots + \theta_q L^q)$.

The impulse response function for an MA model is the sequence of MA coefficients, $1, \theta_1, \ldots, \theta_q$.

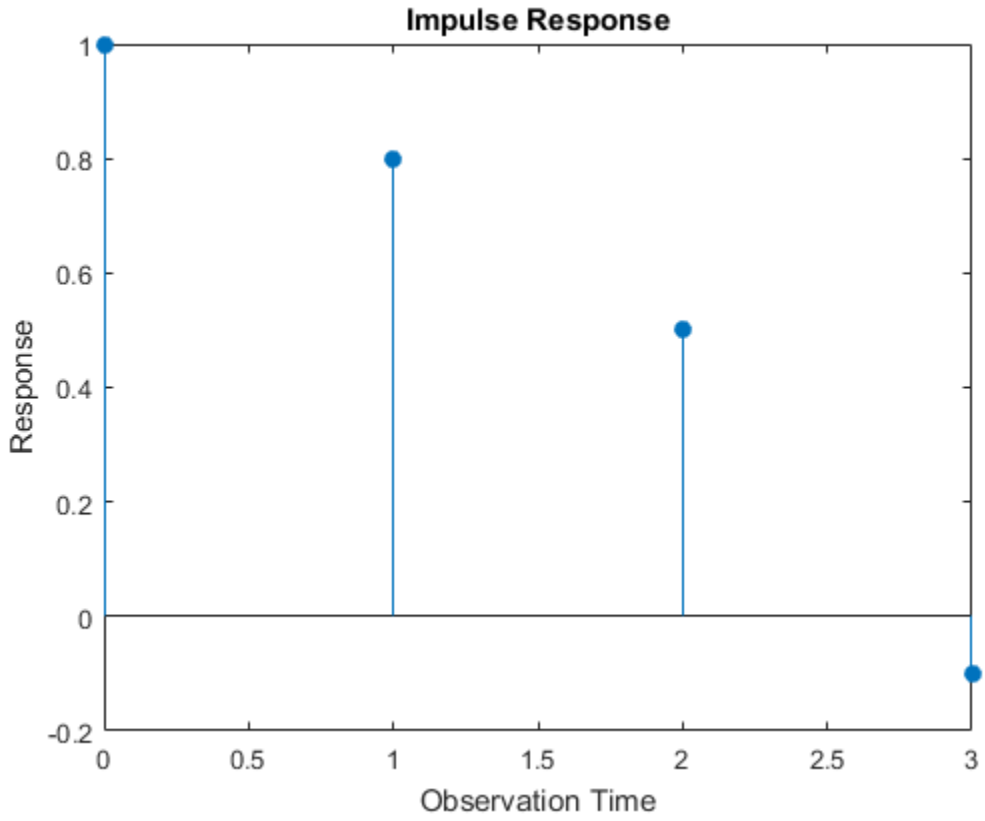
Step 1. Specify the MA model.

Specify a zero-mean MA(3) model with coefficients $\theta_1 = 0.8$, $\theta_2 = 0.5$, and $\theta_3 = -0.1$.

```
modelMA = arima('Constant',0,'MA',{0.8,0.5,-0.1});
```

Step 2. Plot the impulse response function.

```
impulse(modelMA)
```



For an MA model, the impulse response function cuts off after q periods. For this example, the last nonzero coefficient is at lag $q = 3$.

Autoregressive Model

This example shows how to compute and plot the impulse response function for an autoregressive (AR) model. The AR(p) model is given by

$$y_t = \mu + \phi(L)^{-1}\varepsilon_t,$$

where $\phi(L)$ is a p-degree AR operator polynomial, $(1 - \phi_1 L - \ldots - \phi_p L^p)$.

An AR process is stationary provided that the AR operator polynomial is stable, meaning all its roots lie outside the unit circle. In this case, the infinite-degree inverse polynomial, $\psi(L) = \phi(L)^{-1}$, has absolutely summable coefficients, and the impulse response function decays to zero.

Step 1. Specify the AR model.

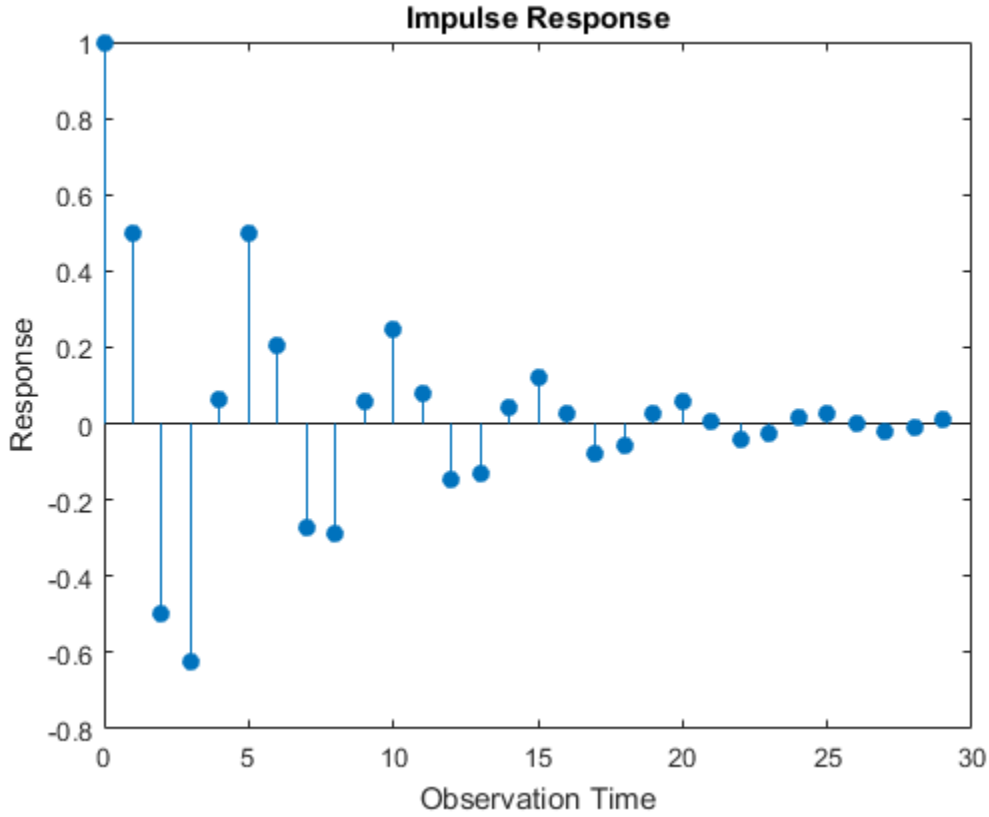
Specify an AR(2) model with coefficients $\phi_1 = 0.5$ and $\phi_2 = -0.75$.

```
modelAR = arima('AR',{0.5,-0.75});
```

Step 2. Plot the impulse response function.

Plot the impulse response function for 30 periods.

```
impulse(modelAR,30)
```



The impulse function decays in a sinusoidal pattern.

ARMA Model

This example shows how to plot the impulse response function for an autoregressive moving average (ARMA) model. The ARMA(p, q) model is given by

$$y_t = \mu + \frac{\theta(L)}{\phi(L)}\varepsilon_t,$$

where $\theta(L)$ is a q-degree MA operator polynomial, $(1 + \theta_1 L + \ldots + \theta_q L^q)$, and $\phi(L)$ is a p-degree AR operator polynomial, $(1 - \phi_1 L - \ldots - \phi_p L^p)$.

An ARMA process is stationary provided that the AR operator polynomial is stable, meaning all its roots lie outside the unit circle. In this case, the infinite-degree inverse polynomial, $\psi(L) = \theta(L)/\phi(L)$, has absolutely summable coefficients, and the impulse response function decays to zero.

Step 1. Specify an ARMA model.

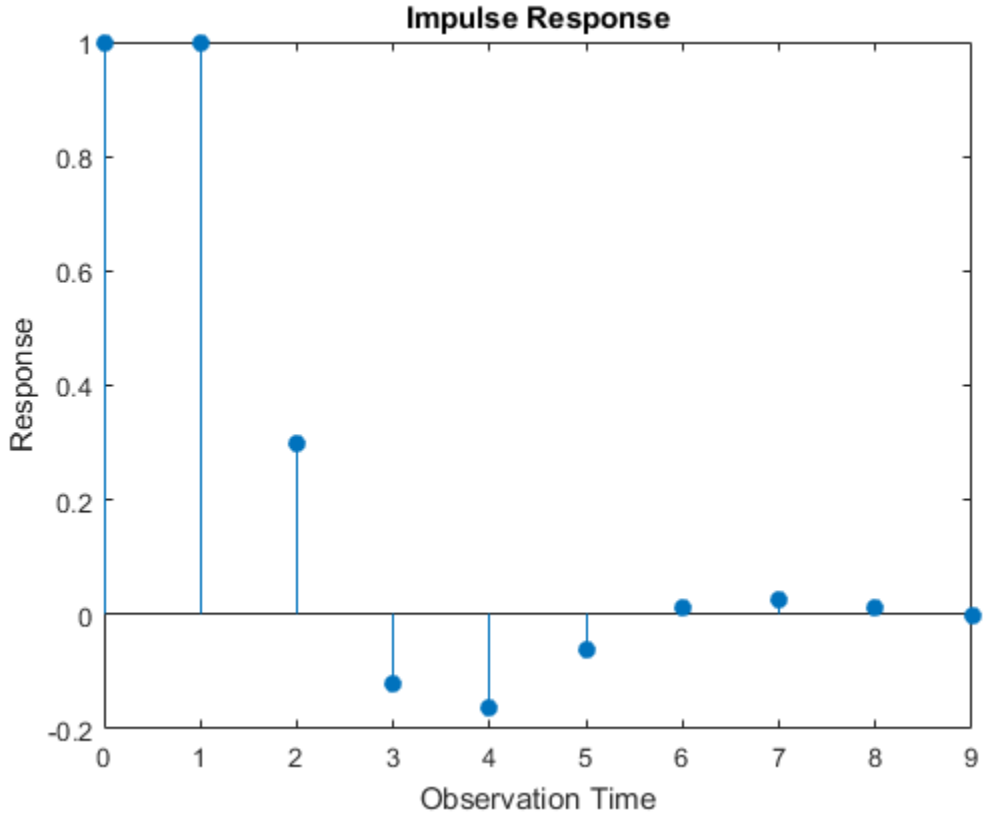
Specify an ARMA(2,1) model with coefficients $\phi_1 = 0.6$, $\phi_2 = -0.3$, and $\theta_1 = 0.4$.

```
modelARMA = arima('AR',{0.6,-0.3},'MA',0.4);
```

Step 2. Plot the impulse response function.

Plot the impulse response function for 10 periods.

```
impulse(modelARMA,10)
```



See Also

arima | `cell2mat` | impulse | isStable | LagOp | toCellArray

More About

- "Impulse Response Function" on page 5-86
- "Moving Average Model" on page 5-27
- "Autoregressive Model" on page 5-18
- "Autoregressive Moving Average Model" on page 5-34

Box-Jenkins Differencing vs. ARIMA Estimation

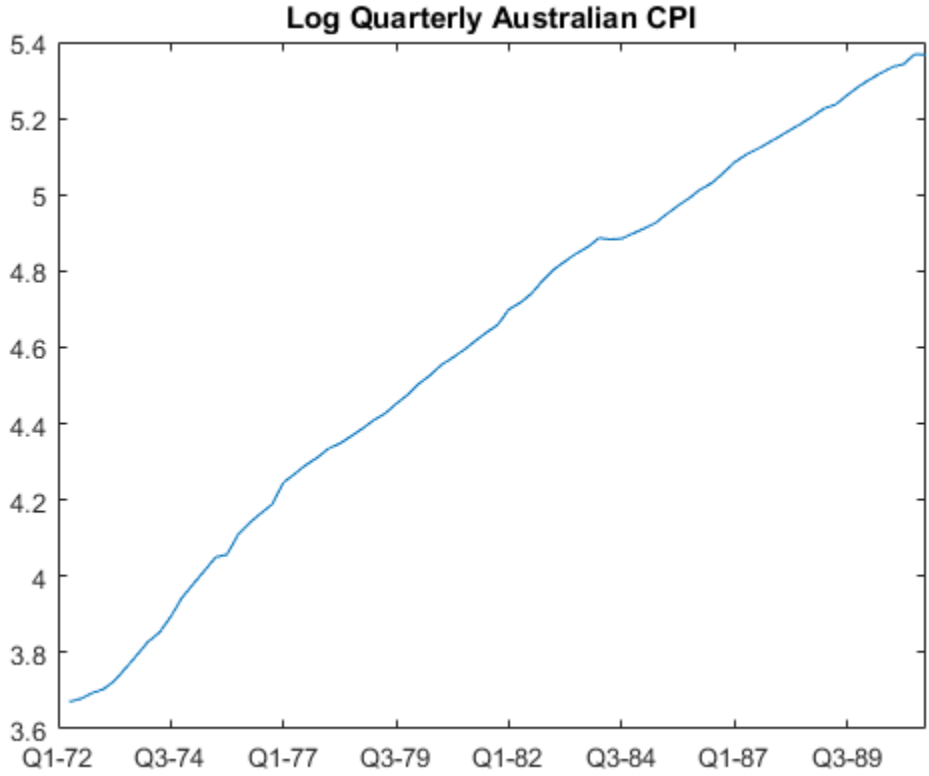
This example shows how to estimate an ARIMA model with nonseasonal integration using `estimate`. The series is not differenced before estimation. The results are compared to a Box-Jenkins modeling strategy, where the data are first differenced, and then modeled as a stationary ARMA model (Box et al., 1994).

The time series is the log quarterly Australian Consumer Price Index (CPI) measured from 1972 through 1991.

Load the Data

Load and plot the Australian CPI data.

```
load Data_JAustralian
y = DataTable.PAU;
T = length(y);

figure
plot(y);
h = gca;        % Define a handle for the current axes
h.XLim = [0,T]; % Set x-axis limits
h.XTickLabel = datestr(dates(1:10:T),17); % Label x-axis tick marks
title('Log Quarterly Australian CPI')
```



The series is nonstationary, with a clear upward trend. This suggests differencing the data before using a stationary model (as suggested by the Box-Jenkins methodology), or fitting a nonstationary ARIMA model directly.

Estimate an ARIMA Model

Specify an ARIMA(2,1,0) model, and estimate.

```
Mdl = arima(2,1,0);
EstMdl = estimate(Mdl,y);
```

```
    ARIMA(2,1,0) Model:
    --------------------
```

```
Conditional Probability Distribution: Gaussian

                           Standard          t
 Parameter       Value       Error       Statistic
-----------   -----------   -----------   -----------
 Constant      0.0100723    0.00328015       3.07069
    AR{1}       0.212059     0.0954278       2.22219
    AR{2}       0.337282      0.103781       3.24994
 Variance    9.23017e-05   1.11119e-05       8.30659
```

The estimated model is

$$\Delta y_t = 0.01 + 0.21\Delta y_{t-1} + 0.34\Delta y_{t-2} + \varepsilon_t,$$

where ε_t is normally distributed with standard deviation 0.01.

The signs of the estimated AR coefficients correspond to the AR coefficients on the right side of the model equation. In lag operator polynomial notation, the fitted model is

$$(1 - 0.21L - 0.34L^2)(1 - L)y_t = \varepsilon_t,$$

with the opposite sign on the AR coefficients.

Difference the Data Before Estimating

Take the first difference of the data. Estimate an AR(2) model using the differenced data.

```
dY = diff(y);
MdlAR = arima(2,0,0);
EstMdlAR = estimate(MdlAR,dY);
```

```
ARIMA(2,0,0) Model:
--------------------
Conditional Probability Distribution: Gaussian

                           Standard          t
 Parameter       Value       Error       Statistic
-----------   -----------   -----------   -----------
 Constant      0.0104289    0.00380427       2.74137
    AR{1}       0.201194      0.101463       1.98293
    AR{2}        0.32299      0.118035        2.7364
 Variance    9.42421e-05   1.16259e-05       8.10622
```

The parameter point estimates are very similar to those in `EstMdl`. The standard errors, however, are larger when the data is differenced before estimation.

Forecasts made using the estimated AR model (`EstMdlAR`) will be on the differenced scale. Forecasts made using the estimated ARIMA model (`EstMdl`) will be on the same scale as the original data.

References:

Box, G. E. P., G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis: Forecasting and Control*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

See Also

arima | estimate

Related Examples

- "Box-Jenkins Model Selection" on page 3-4
- "Infer Residuals for Diagnostic Checking" on page 5-140

More About

- "Box-Jenkins Methodology" on page 3-2
- "ARIMA Model" on page 5-41

Maximum Likelihood Estimation for Conditional Mean Models

Innovation Distribution

For conditional mean models in Econometrics Toolbox, the form of the innovation process is $\varepsilon_t = \sigma_t z_t$, where z_t can be standardized Gaussian or Student's t with $\nu > 2$ degrees of freedom. Specify your distribution choice in the `arima` model object `Distribution` property.

The innovation variance, σ_t^2, can be a positive scalar constant, or characterized by a conditional variance model. Specify the form of the conditional variance using the `Variance` property. If you specify a conditional variance model, the parameters of that model are estimated with the conditional mean model parameters simultaneously.

Given a stationary model,

$$y_t = \mu + \psi(L)\varepsilon_t,$$

applying an inverse filter yields a solution for the innovation ε_t

$$\varepsilon_t = \psi^{-1}(L)(y_t - \mu).$$

For example, for an AR(p) process,

$$\varepsilon_t = -c + \phi(L)y_t,$$

where $\phi(L) = (1 - \phi_1 L - \cdots - \phi_p L^p)$ is the degree p AR operator polynomial.

`estimate` uses maximum likelihood to estimate the parameters of an `arima` model. `estimate` returns fitted values for any parameters in the input model object equal to `NaN`. `estimate` honors any equality constraints in the input model object, and does not return estimates for parameters with equality constraints.

Loglikelihood Functions

Given the history of a process, innovations are conditionally independent. Let H_t denote the history of a process available at time t, $t = 1,...,N$. The likelihood function for the innovation series is given by

$$f(\varepsilon_1,\varepsilon_2,\ldots,\varepsilon_N \mid H_{N-1}) = \prod_{t=1}^{N} f(\varepsilon_t \mid H_{t-1}),$$

where f is a standardized Gaussian or t density function.

The exact form of the loglikelihood objective function depends on the parametric form of the innovation distribution.

- If z_t has a standard Gaussian distribution, then the loglikelihood function is

$$LLF = -\frac{N}{2}\log(2\pi) - \frac{1}{2}\sum_{t=1}^{N}\log\sigma_t^2 - \frac{1}{2}\sum_{t=1}^{N}\frac{\varepsilon_t^2}{\sigma_t^2}.$$

- If z_t has a standardized Student's t distribution with $\nu > 2$ degrees of freedom, then the loglikelihood function is

$$LLF = N\log\left[\frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\sqrt{\pi(\nu-2)}\Gamma\left(\frac{\nu}{2}\right)}\right] - \frac{1}{2}\sum_{t=1}^{N}\log\sigma_t^2 - \frac{\nu+1}{2}\sum_{t=1}^{N}\log\left[1+\frac{\varepsilon_t^2}{\sigma_t^2(\nu-2)}\right].$$

`estimate` performs covariance matrix estimation for maximum likelihood estimates using the outer product of gradients (OPG) method.

See Also

arima | estimate

Related Examples

- "Estimate Multiplicative ARIMA Model" on page 5-113
- "Estimate Conditional Mean and Variance Models" on page 5-129

More About

- "Conditional Mean Model Estimation with Equality Constraints" on page 5-101
- "Presample Data for Conditional Mean Model Estimation" on page 5-103
- "Initial Values for Conditional Mean Model Estimation" on page 5-106
- "Optimization Settings for Conditional Mean Model Estimation" on page 5-108
- "Maximum Likelihood Estimation for Conditional Variance Models" on page 6-62

Conditional Mean Model Estimation with Equality Constraints

For conditional mean model estimation, `estimate` requires an `arima` model and a vector of univariate time series data. The model specifies the parametric form of the conditional mean model that `estimate` estimates. `estimate` returns fitted values for any parameters in the input model with `NaN` values. If you pass a `T×r` exogenous covariate matrix in the `X` argument, then `estimate` returns `r` regression estimates . If you specify non-`NaN` values for any parameters, `estimate` views these values as equality constraints and honors them during estimation.

For example, suppose you are estimating a model without a constant term. Specify `'Constant',0` in the model you pass into `estimate`. `estimate` views this non-`NaN` value as an equality constraint, and does not estimate the constant term. `estimate` also honors all specified equality constraints while estimating parameters without equality constraints. You can set a subset of regression coefficients to a constant and estimate the rest. For example, suppose your model is called `model`. If your model has three exogenous covariates, and you want to estimate two of them and set the other to one to 5, then specify `model.Beta = [NaN 5 NaN]`.

`estimate` optionally returns the variance-covariance matrix for estimated parameters. The parameter order in this matrix is:

- Constant
- Nonzero AR coefficients at positive lags (`AR`)
- Nonzero seasonal AR coefficients at positive lags (`SAR`)
- Nonzero MA coefficients at positive lags (`MA`)
- Nonzero seasonal MA coefficients at positive lags (`SMA`)
- Regression coefficients (when you specify `X`)
- Variance parameters (scalar for constant-variance models, vector of additional parameters otherwise)
- Degrees of freedom (t innovation distribution only)

If any parameter known to the optimizer has an equality constraint, then the corresponding row and column of the variance-covariance matrix has all 0s.

In addition to user-specified equality constraints, `estimate` sets any AR or MA coefficient with an estimate less than `1e-12` in magnitude equal to 0.

See Also

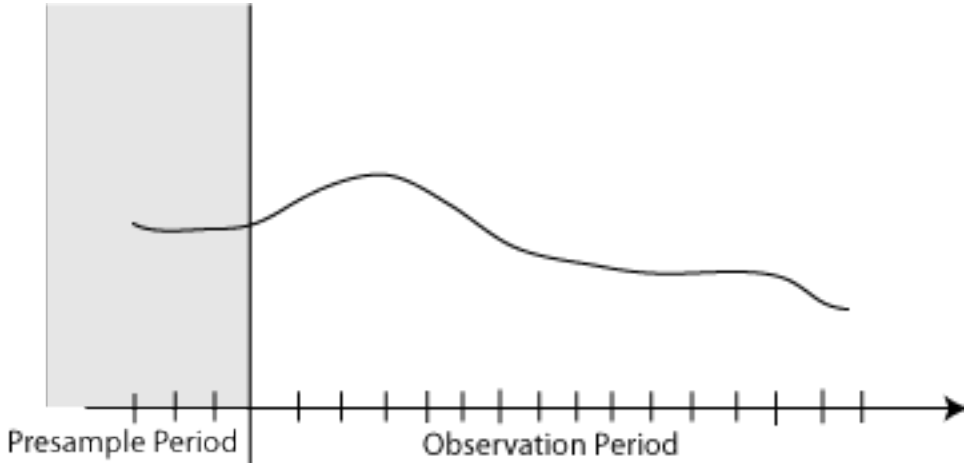
arima | estimate

More About

- "Maximum Likelihood Estimation for Conditional Mean Models" on page 5-98
- "Presample Data for Conditional Mean Model Estimation" on page 5-103
- "Initial Values for Conditional Mean Model Estimation" on page 5-106
- "Optimization Settings for Conditional Mean Model Estimation" on page 5-108

Presample Data for Conditional Mean Model Estimation

Presample data comes from time points before the beginning of the observation period. In Econometrics Toolbox, you can specify your own presample data or use generated presample data.



In a conditional mean model, the distribution of ε_t is conditional on historical information. Historical information includes past responses, $y_1, y_2, \ldots, y_{t-1}$, past innovations, $\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_{t-1}$, and, if you include them in the model, past and present exogenous covariates, $x_1, x_2, \ldots, x_{t-1}, x_t$.

The number of past responses and innovations that a current innovation depends on is determined by the degree of the AR or MA operators, and any differencing. For example, in an AR(2) model, each innovation depends on the two previous responses,

$$\varepsilon_t = y_t - c - \phi_1 y_{t-1} - \phi_2 y_{t-2}.$$

In ARIMAX models, the current innovation also depends on the *current value* of the exogenous covariate (unlike distributed lag models). For example, in an ARX(2) model with one exogenous covariate, each innovation depends on the previous two responses and the current value of the covariate,

$$\varepsilon_t = y_t - c - \phi_1 y_{t-1} - \phi_2 y_{t-2} + x_t.$$

In general, the likelihood contribution of the first few innovations is conditional on historical information that might not be observable. How do you estimate the parameters without all the data? In the ARX(2) example, ε_2 explicitly depends on y_1, y_0, and x_2, and ε_1 explicitly depends on y_0, y_{-1}, and x_1. Implicitly, ε_2 depends on x_1 and x_0, and ε_1 depends on x_0 and x_{-1}. However, you cannot observe y_0, y_{-1}, x_0, and x_{-1}.

The amount of presample data that you need to initialize a model depends on the degree of the model. The property `P` of an `arima` model specifies the number of presample responses and exogenous data that you need to initialize the AR portion of a conditional mean model. For example, `P = 2` in an ARX(2) model. Therefore, you need two responses and two data points from *each* exogenous covariate series to initialize the model.

One option is to use the first `P` data from the response and exogenous covariate series as your presample, and then fit your model to the remaining data. This results in some loss of sample size. If you plan to compare multiple potential models, be aware that you can only use likelihood-based measures of fit (including the likelihood ratio test and information criteria) to compare models fit to the same data (of the same sample size). If you specify your own presample data, then you must use the largest required number of presample responses across all models that you want to compare.

The property `Q` of an `arima` model specifies the number of presample innovations needed to initialize the MA portion of a conditional mean model. You can get presample innovations by dividing your data into two parts. Fit a model to the first part, and infer the innovations. Then, use the inferred innovations as presample innovations for estimating the second part of the data.

For a model with both an autoregressive and moving average component, you can specify both presample responses and innovations, one or the other, or neither.

By default, `estimate` generates automatic presample response and innovation data. The software:

- Generates presample responses by backward forecasting.
- Sets presample innovations to zero.
- Does *not* generate presample exogenous data. One option is to backward forecast each exogenous series to generate a presample during data preprocessing.

See Also

arima | estimate

Related Examples

- "Estimate Multiplicative ARIMA Model" on page 5-113

More About

- "Maximum Likelihood Estimation for Conditional Mean Models" on page 5-98
- "Conditional Mean Model Estimation with Equality Constraints" on page 5-101
- "Initial Values for Conditional Mean Model Estimation" on page 5-106
- "Optimization Settings for Conditional Mean Model Estimation" on page 5-108

Initial Values for Conditional Mean Model Estimation

The `estimate` method for `arima` models uses `fmincon` from Optimization Toolbox to perform maximum likelihood estimation. This optimization function requires initial (or, starting) values to begin the optimization process.

If you want to specify your own initial values, then use name-value arguments. For example, specify initial values for nonseasonal AR coefficients using the name-value argument `AR0`.

Alternatively, you can let `estimate` choose default initial values. Default initial values are generated using standard time series techniques. If you partially specify initial values (that is, specify initial values for some parameters), `estimate` honors the initial values that you set, and generates default initial values for the remaining parameters.

When you generate initial values, `estimate` enforces stability and invertibility for all AR and MA lag operator polynomials. When you specify initial values for the AR and MA coefficients, it is possible that `estimate` cannot find initial values for the remaining coefficients that satisfy stability and invertibility. In this case, `estimate` keeps the user-specified initial values, and sets the remaining initial coefficient values to 0.

This table summarizes the techniques `estimate` uses to generate default initial values. The software uses the methods in this table and the main data set to generate initial values. If you specify seasonal or nonseasonal integration in the model, then `estimate` differences the response series before initial values are generated. Here, AR coefficients and MA coefficients include both nonseasonal and seasonal AR and MA coefficients.

		Technique to Generate Initial Values	
	Parameter	**Regression Coefficients Present**	**Regression Coefficient Not Present**
MA Terms Not in Model	AR coefficients	Ordinary least squares (OLS)	OLS
	Constant	OLS constant	OLS constant
	Regression coefficients	OLS	N/A
	Constant variance	Population variance of OLS residuals	Population variance of OLS residuals

		Technique to Generate Initial Values	
	Parameter	**Regression Coefficients Present**	**Regression Coefficient Not Present**
MA Terms in Model	AR coefficients	OLS	Solve Yule-Walker equations, as described in Box, Jenkins, and Reinsel [1].
	Constant	OLS constant	Mean of AR-filtered series (using initial AR coefficients)
	Regression coefficients	OLS	N/A
	Constant variance	Population variance of OLS residuals	Variance of inferred innovation process (using initial MA coefficients)
	MA coefficients	Solve modified Yule-Walker equations, as described in Box, Jenkins, and Reinsel [1].	

For details about how `estimate` initializes conditional variance model parameters, see "Initial Values for Conditional Variance Model Estimation" on page 6-69.

References

[1] Box, G. E. P., G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis: Forecasting and Control*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

See Also

arima | estimate | `fmincon`

More About

- "Maximum Likelihood Estimation for Conditional Mean Models" on page 5-98
- "Conditional Mean Model Estimation with Equality Constraints" on page 5-101
- "Presample Data for Conditional Mean Model Estimation" on page 5-103
- "Optimization Settings for Conditional Mean Model Estimation" on page 5-108
- "Initial Values for Conditional Variance Model Estimation" on page 6-69

Optimization Settings for Conditional Mean Model Estimation

In this section...

Optimization Options

`estimate` maximizes the loglikelihood function using `fmincon` from Optimization Toolbox. `fmincon` has many optimization options, such as choice of optimization algorithm and constraint violation tolerance. Choose optimization options using `optimoptions`.

`estimate` uses the `fmincon` optimization options by default, with these exceptions. For details, see `fmincon` and `optimoptions` in Optimization Toolbox.

optimoptions Properties	Description	estimate Settings
`Algorithm`	Algorithm for minimizing the negative loglikelihood function	`'sqp'`
`Display`	Level of display for optimization progress	`'off'`
`Diagnostics`	Display for diagnostic information about the function to be minimized	`'off'`
`TolCon`	Termination tolerance on constraint violations	`1e-7`

If you want to use optimization options that differ from the default, then set your own using `optimoptions`.

For example, suppose that you want `estimate` to display optimization diagnostics. The best practice is to set the name-value pair argument `'Display','diagnostics'` in `estimate`. Alternatively, you can direct the optimizer to display optimization diagnostics.

Define an AR(1) model `Mdl` and simulate data from it.

```
Mdl = arima('AR',0.5,'Constant',0,'Variance',1);
rng(1); % For reproducibility
y = simulate(Mdl,25);
```

By default, `fmincon` does not display the optimization diagnostics. Use `optimoptions` to set it to display the optimization diagnostics, and set the other `fmincon` properties to the default settings of `estimate` listed in the previous table.

```
options = optimoptions(@fmincon,'Diagnostics','on',...
    'Algorithm','sqp','Display','off','TolCon',1e-7)
% @fmincon is the function handle for fmincon
```

```
options =

  fmincon options:

   Options used by current Algorithm ('sqp'):
   (Other available algorithms: 'active-set', 'interior-point', 'trust-region-reflectiv

   Set by user:
          Algorithm: 'sqp'
        Diagnostics: 'on'
            Display: 'off'
             TolCon: 1.0000e-07

   Default:
    DerivativeCheck: 'off'
      DiffMaxChange: Inf
      DiffMinChange: 0
     FinDiffRelStep: 'sqrt(eps)'
        FinDiffType: 'forward'
        FunValCheck: 'off'
         GradConstr: 'off'
            GradObj: 'off'
        MaxFunEvals: '100*numberOfVariables'
            MaxIter: 400
     ObjectiveLimit: -1.0000e+20
          OutputFcn: []
           PlotFcns: []
       ScaleProblem: 'none'
             TolFun: 1.0000e-06
               TolX: 1.0000e-06
```

```
          TypicalX: 'ones(numberOfVariables,1)'
       UseParallel: 0

   Options not used by current Algorithm ('sqp')
   Default:
    AlwaysHonorConstraints: 'bounds'
                   HessFcn: []
                  HessMult: []
               HessPattern: 'sparse(ones(numberOfVariables))'
                   Hessian: 'not applicable'
         InitBarrierParam: 0.1000
     InitTrustRegionRadius: 'sqrt(numberOfVariables)'
                MaxPCGIter: 'max(1,floor(numberOfVariables/2))'
              MaxProjCGIter: '2*(numberOfVariables-numberOfEqualities)'
                 MaxSQPIter: '10*max(numberOfVariables,numberOfInequalities+…'
        PrecondBandWidth: 0
          RelLineSrchBnd: []
    RelLineSrchBndDuration: 1
       SubproblemAlgorithm: 'ldl-factorization'
                 TolConSQP: 1.0000e-06
                    TolPCG: 0.1000
                 TolProjCG: 0.0100
              TolProjCGAbs: 1.0000e-10
```

The options that you set appear under the `Set by user:` heading. The properties under the `Default:` heading are other options that you can set.

Fit `Mdl` to `y` using the new optimization options.

```
ToEstMdl = arima(1,0,0);
EstMdl = estimate(ToEstMdl,y,'Options',options);
```

```
____________________________________________________________
   Diagnostic Information

Number of variables: 3

Functions
Objective:                              @(X)nLogLike(X,YData,XData,E,V,Mdl,AR.Lags,MA.Lag
Gradient:                               finite-differencing
Hessian:                                finite-differencing (or Quasi-Newton)
Nonlinear constraints:                  @(x)internal.econ.arimaNonLinearConstraints(x,Lag
```

```
Nonlinear constraints gradient:       finite-differencing

Constraints
Number of nonlinear inequality constraints: 1
Number of nonlinear equality constraints:   0

Number of linear inequality constraints:    0
Number of linear equality constraints:      0
Number of lower bound constraints:          3
Number of upper bound constraints:          3

Algorithm selected
   sqp


____________________________________________________________
   End diagnostic information

    ARIMA(1,0,0) Model:
    --------------------
    Conditional Probability Distribution: Gaussian

                                Standard          t
     Parameter       Value        Error       Statistic
    -----------   -----------   ------------   -----------
     Constant     -0.0648568      0.234565      -0.276498
        AR{1}       0.463859      0.157813         2.9393
     Variance        1.23081      0.472745        2.60354
```

Note:

- `estimate` numerically maximizes the loglikelihood function potentially using equality, inequality, and lower and upper bound constraints. If you set `Algorithm` to anything other than `sqp`, then check that the algorithm supports similar constraints, such as `interior-point`. For example, `fmincon` sets `Algorithm` to `trust-region-reflective` by default. `trust-region-reflective` does not support inequality constraints. Therefore, if you do not change the default `Algorithm` property value of `fmincon`, then `estimate` displays a warning. During estimation, `fmincon` temporarily sets `Algorithm` to `active-set` by default to satisfy the constraints.
- `estimate` sets a constraint level of `TolCon` so constraints are not violated. Be aware that an estimate with an active constraint has unreliable standard errors since

variance-covariance estimation assumes the likelihood function is locally quadratic around the maximum likelihood estimate.

Conditional Mean Model Constraints

The software enforces these constraints while estimating an ARIMA model:

- Stability of nonseasonal and seasonal AR operator polynomials
- Invertibility of nonseasonal and seasonal MA operator polynomials
- Innovation variance strictly greater than zero
- Degrees of freedom strictly greater than two for a *t* innovation distribution

See Also

arima | estimate | `fmincon` | `optimoptions`

More About

- "Maximum Likelihood Estimation for Conditional Mean Models" on page 5-98
- "Conditional Mean Model Estimation with Equality Constraints" on page 5-101
- "Presample Data for Conditional Mean Model Estimation" on page 5-103
- "Initial Values for Conditional Mean Model Estimation" on page 5-106

Estimate Multiplicative ARIMA Model

This example shows how to estimate a multiplicative seasonal ARIMA model using `estimate`. The time series is monthly international airline passenger numbers from 1949 to 1960.

Load the Data and Specify the model.

Load the airline data set.

```
load(fullfile(matlabroot,'examples','econ','Data_Airline.mat'))
y = log(Data);
T = length(y);

Mdl = arima('Constant',0,'D',1,'Seasonality',12,...
    'MALags',1,'SMALags',12);
```

Estimate the Model Using Presample Data.

Use the first 13 observations as presample data, and the remaining 131 observations for estimation.

```
y0 = y(1:13);
[EstMdl,EstParamCov] = estimate(Mdl,y(14:end),'Y0',y0)
```

```
    ARIMA(0,1,1) Model Seasonally Integrated with Seasonal MA(12):
    --------------------------------------------------------------
    Conditional Probability Distribution: Gaussian

                               Standard          t
     Parameter       Value       Error       Statistic
    -----------   -----------   ------------   -----------
     Constant             0         Fixed          Fixed
        MA{1}      -0.377161     0.0734258       -5.13662
       SMA{12}     -0.572379     0.0939327         -6.0935
     Variance     0.00138874   0.000152417          9.1115

EstMdl =

    ARIMA(0,1,1) Model Seasonally Integrated with Seasonal MA(12):
    --------------------------------------------------------------
    Distribution: Name = 'Gaussian'
```

```
               P: 13
               D: 1
               Q: 13
        Constant: 0
              AR: {}
             SAR: {}
              MA: {-0.377161} at Lags [1]
             SMA: {-0.572379} at Lags [12]
     Seasonality: 12
        Variance: 0.00138874

EstParamCov =

         0         0         0         0
         0    0.0054   -0.0015   -0.0000
         0   -0.0015    0.0088    0.0000
         0   -0.0000    0.0000    0.0000
```

The fitted model is

$$\Delta\Delta_{12}y_t = (1 - 0.38L)(1 - 0.57L^{12})\varepsilon_t,$$

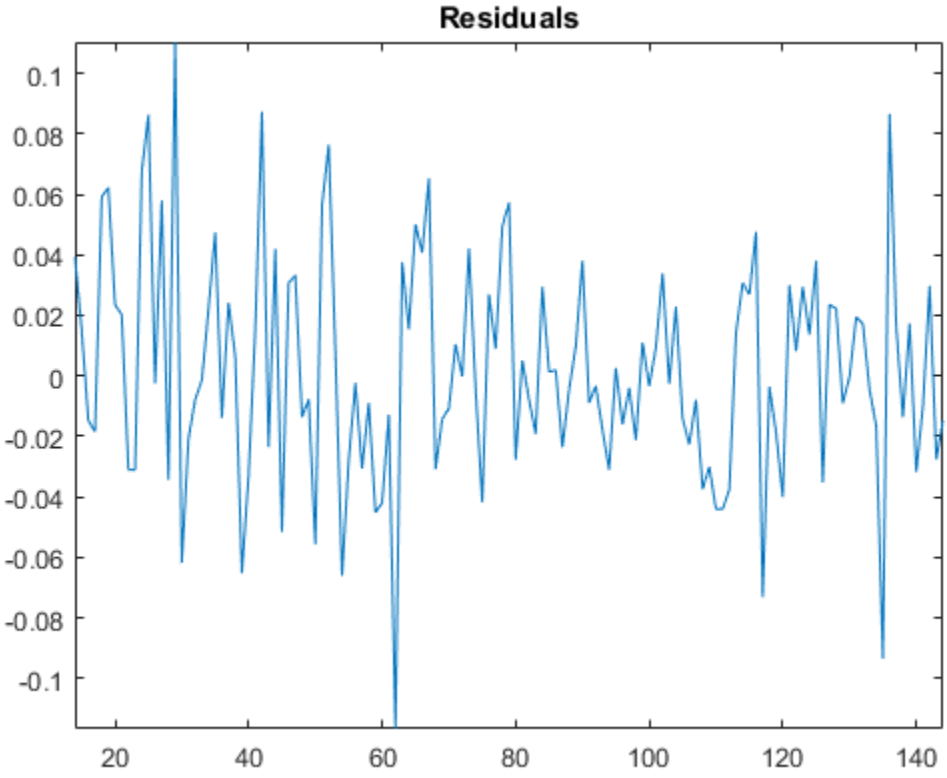
with innovation variance 0.0014.

Notice that the model constant is not estimated, but remains fixed at zero. There is no corresponding standard error or t statistic for the constant term. The row (and column) in the variance-covariance matrix corresponding to the constant term has all zeros.

Infer the Residuals.

Infer the residuals from the fitted model.

```
res = infer(EstMdl,y(14:end),'Y0',y0);

figure
plot(14:T,res)
xlim([0,T])
title('Residuals')
axis tight
```



When you use the first 13 observations as presample data, residuals are available from time 14 onward.

References:

Box, G. E. P., G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis: Forecasting and Control*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

See Also

arima | estimate | infer

Related Examples

- “Specify Multiplicative ARIMA Model” on page 5-52
- “Simulate Multiplicative ARIMA Models” on page 5-169
- “Forecast Multiplicative ARIMA Model” on page 5-192
- “Check Fit of Multiplicative ARIMA Model” on page 3-81

More About

- “Conditional Mean Model Estimation with Equality Constraints” on page 5-101
- “Presample Data for Conditional Mean Model Estimation” on page 5-103

Model Seasonal Lag Effects Using Indicator Variables

This example shows how to estimate a seasonal ARIMA model:

- Model the seasonal effects using a multiplicative seasonal model.
- Use indicator variables as a regression component for the seasonal effects, called seasonal dummies.

Subsequently, their forecasts show that the methods produce similar results. The time series is monthly international airline passenger numbers from 1949 to 1960.

Step 1. Load the data.

Load the data set `Data_Airline`, and plot the natural log of the monthly passenger totals counts.

```
load(fullfile(matlabroot,'examples','econ','Data_Airline.mat'))
dat = log(Data); % Transform to logarithmic scale
T = size(dat,1);
y = dat(1:103); % estimation sample
```

`y` is the part of `dat` used for estimation, and the rest of `dat` is the holdout sample to compare the two models' forecasts.

Step 2. Define and fit the model specifying seasonal lags.

Create an $\mathrm{ARIMA}(0,1,1)(0,1,1)_{12}$ model

$$(1-L)(1-L^{12})y_t = (1+\theta_1 L)(1+\Theta_{12}L^{12})\varepsilon_t,$$

where ε_t is an independent and identically distributed normally distributed series with mean 0 and variance σ^2. Use `estimate` to fit `model1` to `y`.

```
model1 = arima('MALags', 1, 'D', 1, 'SMALags', 12,...
'Seasonality',12, 'Constant', 0);
fit1 = estimate(model1,y);
```

```
ARIMA(0,1,1) Model Seasonally Integrated with Seasonal MA(12):
--------------------------------------------------------------
Conditional Probability Distribution: Gaussian
```

```
                                Standard          t
    Parameter       Value         Error       Statistic
    -----------   -----------   ------------   -----------
    Constant            0         Fixed          Fixed
       MA{1}      -0.357317      0.088031       -4.05899
     SMA{12}      -0.614685     0.0962493       -6.38639
    Variance     0.00130504   0.000152696        8.54666
```

The fitted model is

$$(1-L)(1-L^{12})y_t = (1-0.357L)(1-0.615L^{12})\varepsilon_t,$$

where ε_t is an iid normally distributed series with mean 0 and variance 0.0013.

Step 3. Define and fit the model using seasonal dummies.

Create an ARIMAX(0,1,1) model with period 12 seasonal differencing and a regression component,

$$(1-L)(1-L^{12})y_t = (1-0.357L)(1-0.615L^{12})\varepsilon_t,$$

$\{x_t; t = 1, ..., T\}$ is a series of T column vectors having length 12 that indicate in which month observation t was measured. A 1 in row i of x_t indicates that the observation was measured in month i, the rest of the elements are 0s.

Note that if you include an additive constant in the model, then the T rows of the design matrix X are composed of the row vectors $\begin{bmatrix} 1 & x_t' \end{bmatrix}$. Therefore, X is rank deficient, and one regression coefficient is not identifiable. A constant is left out of this example to avoid distraction from the main purpose. Format the in-sample X matrix

```
X  = dummyvar(repmat((1:12)', 12, 1));
% Format the presample X matrix
X0 = [zeros(1,11) 1 ; dummyvar((1:12)')];
model2 = arima('MALags', 1, 'D', 1, 'Seasonality',...
   12, 'Constant', 0);
fit2   = estimate(model2,y, 'X', [X0 ; X]);
```

```
    ARIMAX(0,1,1) Model Seasonally Integrated:
    ------------------------------------------
    Conditional Probability Distribution: Gaussian

                                    Standard          t
```

```
    Parameter       Value          Error       Statistic
-----------   -----------   ------------   -----------
    Constant             0          Fixed          Fixed
       MA{1}      -0.407106      0.0843875       -4.82425
       Beta1    -0.00257697      0.0251683       -0.10239
       Beta2     -0.0057769      0.0318848       -0.18118
       Beta3    -0.00220339      0.0305268      -0.0721787
       Beta4    0.000947372      0.0198667       0.0476864
       Beta5     -0.0012146      0.0179806      -0.0675506
       Beta6      0.00486998       0.018374        0.265047
       Beta7    -0.00879439      0.0152852       -0.575354
       Beta8      0.00483464      0.0124836        0.387279
       Beta9      0.00143697      0.0182453       0.0787582
      Beta10      0.00927403      0.0147513        0.628693
      Beta11      0.00736654         0.0105        0.701577
      Beta12    0.000988407      0.0142945       0.0691458
    Variance      0.00177152   0.000246566         7.18475
```

The fitted model is

$$(1-L)(1-L^{12})y_t = x_t'\hat{\beta} + (1-0.407L)\varepsilon_t,$$

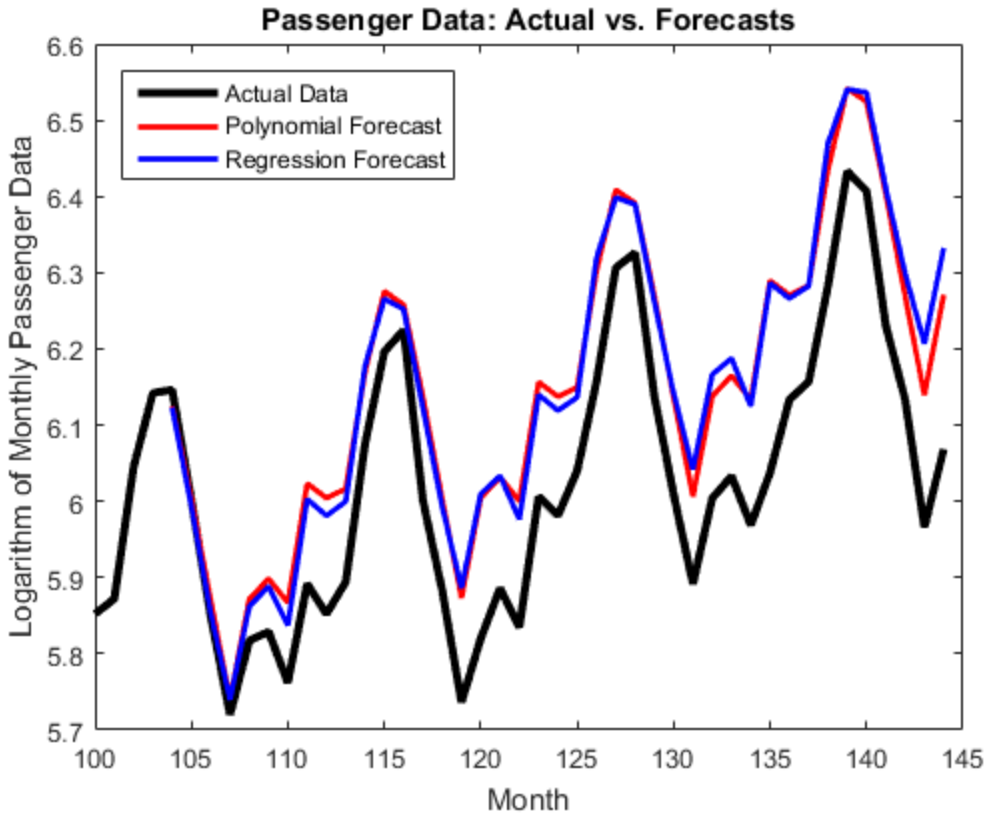
where ε_t is an iid normally distributed series with mean 0 and variance 0.0017 and $\hat{\beta}$ is a column vector with the values `Beta1` - `Beta12`. Note that the estimates `MA{1}` and `Variance` between `model1` and `model2` are not equal.

Step 4. Forecast using both models.

Use `forecast` to forecast both models 41 periods into the future from July 1957. Plot the holdout sample using these forecasts.

```
yF1 = forecast(fit1,41,'Y0',y);
yF2 = forecast(fit2,41,'Y0',y,'X0',X(1:103,:),...
   'XF',X(104:end,:));
l1 = plot(100:T,dat(100:end),'k','LineWidth',3);
hold on
l2 = plot(104:144,yF1,'-r','LineWidth',2);
l3 = plot(104:144,yF2,'-b','LineWidth',2);
hold off
title('Passenger Data: Actual vs. Forecasts')
xlabel('Month')
ylabel('Logarithm of Monthly Passenger Data')
legend({'Actual Data','Polynomial Forecast',...
```

```
'Regression Forecast'},'Location','NorthWest')
```



Though they overpredict the holdout observations, the forecasts of both models are almost equivalent. One main difference between the models is that `model1` is more parsimonious than `model2`.

References:

Box, G. E. P., G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis: Forecasting and Control*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

See Also

arima | `dummyvar` | estimate | forecast

Related Examples

- "Specify Multiplicative ARIMA Model" on page 5-52
- "Estimate Multiplicative ARIMA Model" on page 5-113
- "Forecast Multiplicative ARIMA Model" on page 5-192
- "Check Fit of Multiplicative ARIMA Model" on page 3-81
- "Forecast IGD Rate Using ARIMAX Model" on page 5-122

More About

- "Multiplicative ARIMA Model" on page 5-46
- "ARIMA Model Including Exogenous Covariates" on page 5-58
- "Conditional Mean Model Estimation with Equality Constraints" on page 5-101
- "MMSE Forecasting of Conditional Mean Models" on page 5-182

Forecast IGD Rate Using ARIMAX Model

This example shows how to forecast an ARIMAX model two ways.

Step 1. Load the data.

Load the Credit Defaults data set, assign the response IGD to `Y` and the predictors AGE, CPF, and SPR to the matrix `X`, and obtain the sample size `T`. To avoid distraction from the purpose of this example, assume that all predictors are stationary.

```
load Data_CreditDefaults
X = Data(:,[1 3:4]);
T = size(X,1);
Y = Data(:,5);
```

Step 2. Process response and predictor data.

Divide the response and predictor data into estimation and holdout series. Assume that each predictor series is AR(1), and fit each one to that model. Forecast the predictor series over a 10-year horizon.

```
Yest = Y(2:(T-10)) % Response data for estimation;
Xest = X(1:(T-10),:) % Predictor data for estimation;

modelX = arima(1,0,0); % Model for the predictors
X1fit = estimate(modelX,Xest(:,1),'print',false);
X2fit = estimate(modelX,Xest(:,2),'print',false);
X3fit = estimate(modelX,Xest(:,3),'print',false);

X1fore = forecast(X1fit,10,'Y0',Xest(:,1));
X2fore = forecast(X2fit,10,'Y0',Xest(:,2));
X3fore = forecast(X3fit,10,'Y0',Xest(:,3));
XF = [X1fore X2fore X3fore];
```

```
Yest =

    0.1496
         0
         0
    0.1429
    0.1382
    0.2005
         0
         0
```

```
    0.0522
    0.0466


Xest =

    3.6184   -0.0571    1.9700
    6.2827    3.4798    2.3100
    8.3958    3.5782    2.9000
    5.4114   -0.8488    2.4100
   13.1429    3.2469    1.5000
    7.8093   -4.1607    1.9200
    7.5535   -6.3518    2.3200
    7.7062    6.6135    2.2800
    5.4306    9.7126    2.0700
    5.3236    6.5485    1.9400
    9.2351    2.6272    1.2700
```

`XF` holds the forecasts for the three predictor series.

Step 3. Estimate response model and infer residuals.

Assume that the response series is ARX(1), and fit it to that model including the predictor series. Infer the residuals `Eest` from the fitted response model `Yfit`.

```
modelY = arima(1,0,0);
Yfit = estimate(modelY,Yest,'X',Xest,...
'print',false,'Y0',Y(1));
Eest = infer(Yfit,Yest,'Y0',Y(1),'X',Xest);
```

Step 5. MMSE forecast responses.

Forecast responses using the MMSE method at a 10-year horizon. Calculate prediction intervals for the forecasts assuming that they are normally distributed.

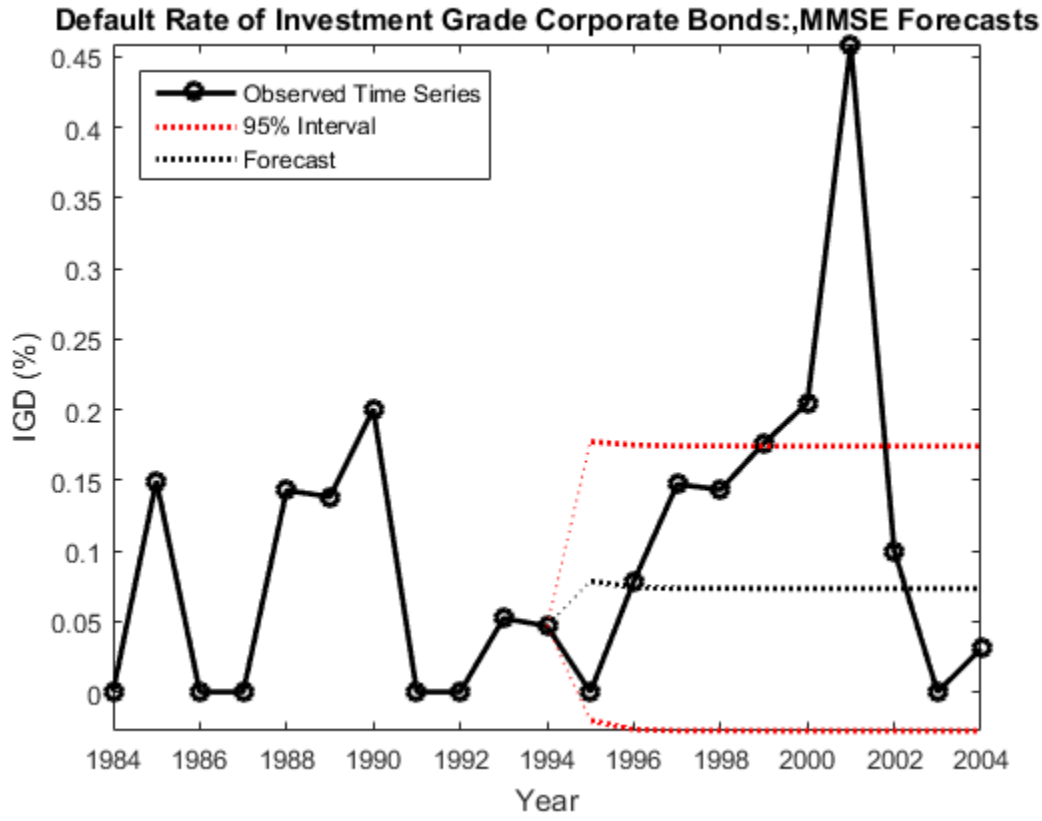
```
[Yfore,YMSE] = forecast(Yfit,10,'Y0',Y(1:(T-10)),...
   'X0',Xest,'XF',XF);
cil = Yfore - 1.96*sqrt(YMSE);
ciu = Yfore + 1.96*sqrt(YMSE);
```

Step 6. Plot MMSE forecasted responses.

Plot the response series using their MMSE forecasts and prediction intervals.

```
figure
```

```
l1 = plot(dates,Y,'ko-','LineWidth',2);
xlabel('Year')
ylabel('IGD (%)')
hold on
l2 = plot(dates((T-9):T),cil,'r:','LineWidth',2);
plot(dates((T-9):T),ciu,'r:','LineWidth',2)
l3 = plot(dates((T-9):T),Yfore,'k:','LineWidth',2);

plot(dates([T-10 T-9]),[Y(T-10) Yfore(1)],'k:')
plot(dates([T-10 T-9]),[Y(T-10) cil(1)],'r:')
plot(dates([T-10 T-9]),[Y(T-10) ciu(1)],'r:')

legend([l1 l2 l3],'Observed Time Series','95% Interval',...
   'Forecast','Location','NorthWest')
title('Default Rate of Investment Grade Corporate Bonds:,MMSE Forecasts')
axis tight
hold off
```



The forecasts seem reasonable, but there are outlying observations in 2000 and 2001.

Step 7. Monte Carlo forecast responses.

Forecast responses using the Monte Carlo method at a 10-year horizon by simulating 100 paths using the model `Yfit`. Set the estimation responses to `Y0` and the inferred residuals to `E0` as preforecast data. Set the forecasted predictors (`XF`) to `X`. Calculate simulation statistics.

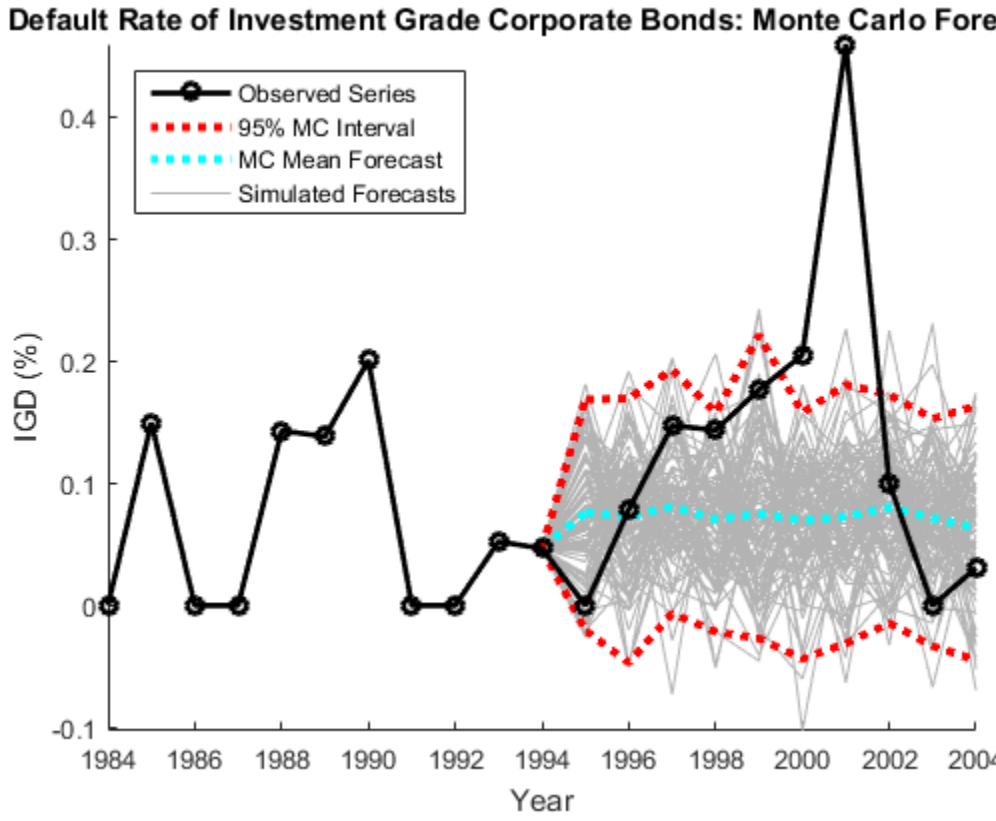
```
nsim = 100;
rng(1);
Ymcfore = simulate(Yfit,10,'NumPaths',nsim,'Y0',...
   Y(1:(T-10)),'E0',Eest,'X',XF);
Ymcforebar = mean(Ymcfore,2);
```

```
mc_cil = quantile(Ymcfore',0.025);
mc_ciu = quantile(Ymcfore',0.975);
```

Step 8. Plot Monte Carlo forecasted responses.

Plot the response series with their MMSE forecasts and prediction intervals.

```
figure
xlabel('Year')
ylabel('IGD (%)')

hold on
l4 = plot(dates((T-9):T),Ymcfore(:,1),'Color',[0.7 0.7 0.7]);
plot(dates((T-9):T),Ymcfore,'Color',[0.7 0.7 0.7]);
l2 = plot(dates((T-9):T),mc_cil,'r:','LineWidth',3);
plot(dates((T-9):T),mc_ciu,'r:','LineWidth',3)
l3 = plot(dates((T-9):T),Ymcforebar,'c:','LineWidth',3);

plot(dates([T-10 T-9]),[repmat(Y(T-10),1,nsim);...
   Ymcfore(1,:)],'Color',[0.7 0.7 0.7])
plot(dates([T-10 T-9]),[Y(T-10) Ymcforebar(1)],'c:',...
   'LineWidth',3)
plot(dates([T-10 T-9]),[Y(T-10) mc_cil(1)],'r:','LineWidth',3)
plot(dates([T-10 T-9]),[Y(T-10) mc_ciu(1)],'r:','LineWidth',3)
l1 = plot(dates,Y,'ko-','LineWidth',2);

legend([l1 l2 l3 l4],'Observed Series','95% MC Interval',...
   'MC Mean Forecast','Simulated Forecasts','Location',...
   'NorthWest')
title('Default Rate of Investment Grade Corporate Bonds: Monte Carlo Forecasts')
axis tight
hold off
```



The Monte Carlo forecasts and prediction intervals resemble those from the MMSE forecasting.

References:

Helwege, J., and P. Kleiman. "Understanding Aggregate Default Rates of High Yield Bonds." *Current Issues in Economics and Finance*. Vol. 2, Number. 6, 1996, pp. 1-6.

Loeffler, G., and P. N. Posch. *Credit Risk Modeling Using Excel and VBA*. West Sussex, England: Wiley Finance, 2007.

See Also

arima | estimate | forecast | infer | simulate

Related Examples

- “Specify Multiplicative ARIMA Model” on page 5-52
- “Estimate Multiplicative ARIMA Model” on page 5-113
- “Simulate Multiplicative ARIMA Models” on page 5-169
- “Model Seasonal Lag Effects Using Indicator Variables” on page 5-117
- “Check Fit of Multiplicative ARIMA Model” on page 3-81
- “Model Seasonal Lag Effects Using Indicator Variables” on page 5-117

More About

- “MMSE Forecasting of Conditional Mean Models” on page 5-182
- “Monte Carlo Forecasting of Conditional Mean Models” on page 5-181

Estimate Conditional Mean and Variance Models

This example shows how to estimate a composite conditional mean and variance model using `estimate`.

Load the Data and Specify the Model.

Load the NASDAQ data included with the toolbox. Convert the daily close composite index series to a return series. For numerical stability, convert the returns to percentage returns. Specify an AR(1) and GARCH(1,1) composite model. This is a model of the form

$$r_t = c + \phi_1 r_{t-1} + \varepsilon_t,$$

where $\varepsilon_t = \sigma_t z_t$,

$$\sigma_t^2 = \kappa + \gamma_1 \sigma_{t-1}^2 + \alpha_1 \varepsilon_{t-1}^2,$$

and z_t is an independent and identically distributed standardized Gaussian process.

```
load Data_EquityIdx
nasdaq = DataTable.NASDAQ;
r = 100*price2ret(nasdaq);
T = length(r);

Mdl = arima('ARLags',1,'Variance',garch(1,1))
```

```
Mdl =

    ARIMA(1,0,0) Model:
    --------------------
    Distribution: Name = 'Gaussian'
               P: 1
               D: 0
               Q: 0
        Constant: NaN
              AR: {NaN} at Lags [1]
             SAR: {}
              MA: {}
             SMA: {}
        Variance: [GARCH(1,1) Model]
```

Estimate the Model Parameters Without Using Presample Data.

Fit the model, `Mdl`, to the return series, `r`, using `estimate`. Use the presample observations that `estimate` automatically generates.

```
EstMdl = estimate(Mdl,r);
```

```
ARIMA(1,0,0) Model:
--------------------
Conditional Probability Distribution: Gaussian

                            Standard          t
 Parameter       Value          Error       Statistic
-----------   -----------   ------------   -----------
 Constant       0.072632     0.0180473        4.02454
    AR{1}       0.138157     0.0198931          6.945


GARCH(1,1) Conditional Variance Model:
--------------------------------------
Conditional Probability Distribution: Gaussian

                            Standard          t
 Parameter       Value          Error       Statistic
-----------   -----------   ------------   -----------
 Constant      0.0223769    0.00332007        6.73988
 GARCH{1}        0.87312    0.00910186        95.9276
  ARCH{1}       0.118649    0.00871697        13.6112
```

The estimation display shows the five estimated parameters and their corresponding standard errors (the AR(1) conditional mean model has two parameters, and the GARCH(1,1) conditional variance model has three parameters).

The fitted model (`EstMdl`) is

$$r_t = 7.1 \times 10^{-4} + 0.14 r_{t-1} + \varepsilon_t,$$

where $\varepsilon_t = \sigma_t z_t$ and

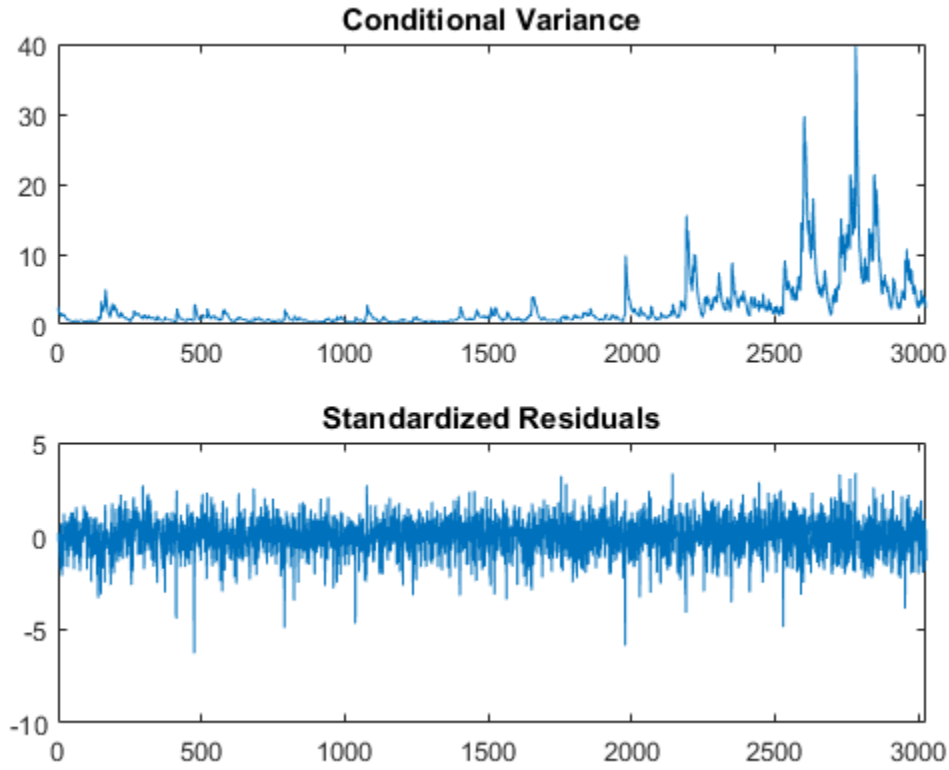
$$\sigma_t^2 = 2.3 \times 10^{-6} + 0.87\sigma_{t-1}^2 + 0.12\varepsilon_{t-1}^2.$$

All t statistics are greater than two, suggesting all parameters are statistically significant.

Infer the Conditional Variances and Residuals.

Infer and plot the conditional variances and standardized residuals. Also output the loglikelihood objective function value.

```
[res,v,logL] = infer(EstMdl,r);

figure
subplot(2,1,1)
plot(v)
xlim([0,T])
title('Conditional Variance')

subplot(2,1,2)
plot(res./sqrt(v))
xlim([0,T])
title('Standardized Residuals')
```



The conditional variances increase after observation 2000. This corresponds to the increased volatility seen in the original return series.

The standardized residuals have more large values (larger than 2 or 3 in absolute value) than expected under a standard normal distribution. This suggests a Student's t distribution might be more appropriate for the innovation distribution.

Fit a Model With a t-Innovation Distribution.

Modify the model so that it has a Student's t-innovation distribution. Fit the modified model to the NASDAQ return series. Specify an initial value for the variance model constant term.

```
MdlT = Mdl;
```

```
MdlT.Distribution = 't';
EstMdlT = estimate(MdlT,r,'Variance0',{'Constant0',0.001});
```

```
    ARIMA(1,0,0) Model:
    --------------------
    Conditional Probability Distribution: t

                               Standard          t
     Parameter       Value       Error       Statistic
    -----------   -----------   ------------   -----------
     Constant       0.093488     0.0166938        5.60018
        AR{1}       0.139107     0.0188565        7.37713
          DoF        7.47747      0.882611        8.47199


    GARCH(1,1) Conditional Variance Model:
    ----------------------------------------
    Conditional Probability Distribution: t

                               Standard          t
     Parameter       Value       Error       Statistic
    -----------   -----------   ------------   -----------
     Constant      0.0112456    0.00363047        3.09756
     GARCH{1}       0.907662     0.0105156        86.3156
      ARCH{1}      0.0898971     0.0108354        8.29661
          DoF        7.47747      0.882611        8.47199
```

The coefficient estimates change slightly when the t distribution is used for the innovations. The second model fit (`EstMdlT`) has one additional parameter estimate, the t distribution degrees of freedom. The estimated degrees of freedom are relatively small (about 8), indicating significant departure from normality.

Compare the Model Fits.

Compare the two model fits (Gaussian and t-innovation distribution) using the Akaike information criterion (AIC) and Bayesian information criterion (BIC). First, obtain the loglikelihood objective function value for the second fit.

```
[resT,vT,logLT] = infer(EstMdlT,r);
[aic,bic] = aicbic([logL,logLT],[5,6],T)
```

```
aic =
```

```
   1.0e+03 *

    9.4929    9.3807

bic =

   1.0e+03 *

    9.5230    9.4168
```

The second model has six parameters compared to five in the first model (because of the t distribution degrees of freedom). Despite this, both information criteria favor the model with the Student's t distribution. The AIC and BIC values are smaller for the t innovation distribution.

See Also

`aicbic` | arima | estimate | infer

Related Examples

- "Specify Conditional Mean and Variance Models" on page 5-79
- "Specify Conditional Mean Model Innovation Distribution" on page 5-72

More About

- "Initial Values for Conditional Mean Model Estimation" on page 5-106
- "Information Criteria" on page 3-63

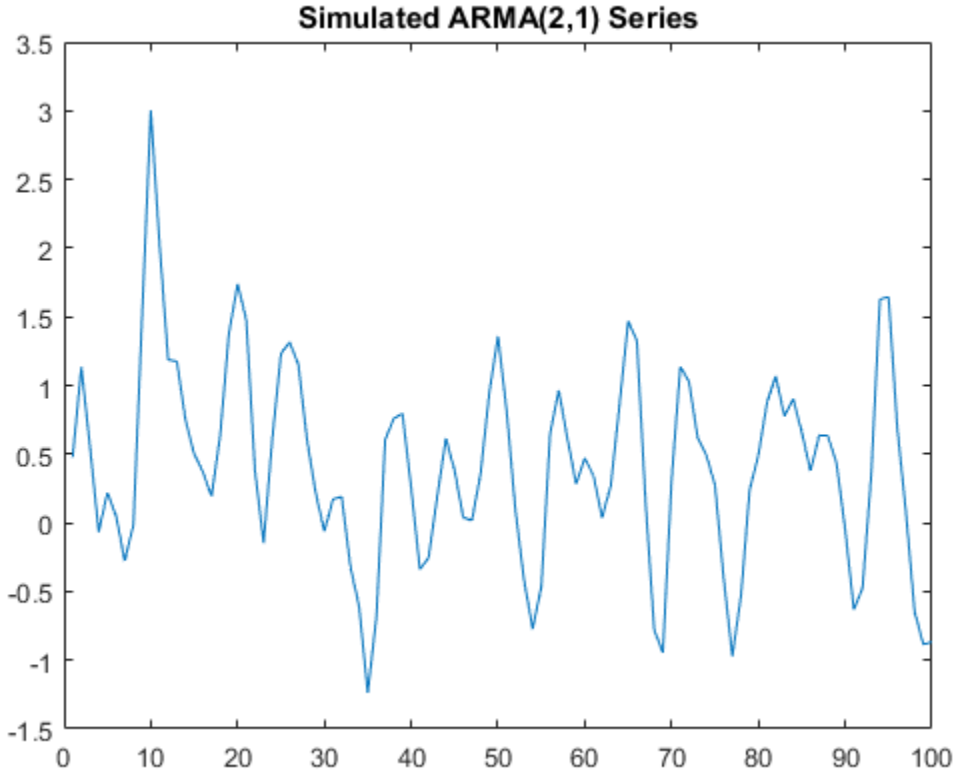
Choose ARMA Lags Using BIC

This example shows how to use the Bayesian information criterion (BIC) to select the degrees p and q of an ARMA model. Estimate several models with different p and q values. For each estimated model, output the loglikelihood objective function value. Input the loglikelihood value to `aicbic` to calculate the BIC measure of fit (which penalizes for complexity).

Step 1. Simulate an ARMA time series.

Simulate an ARMA(2,1) time series with 100 observations.

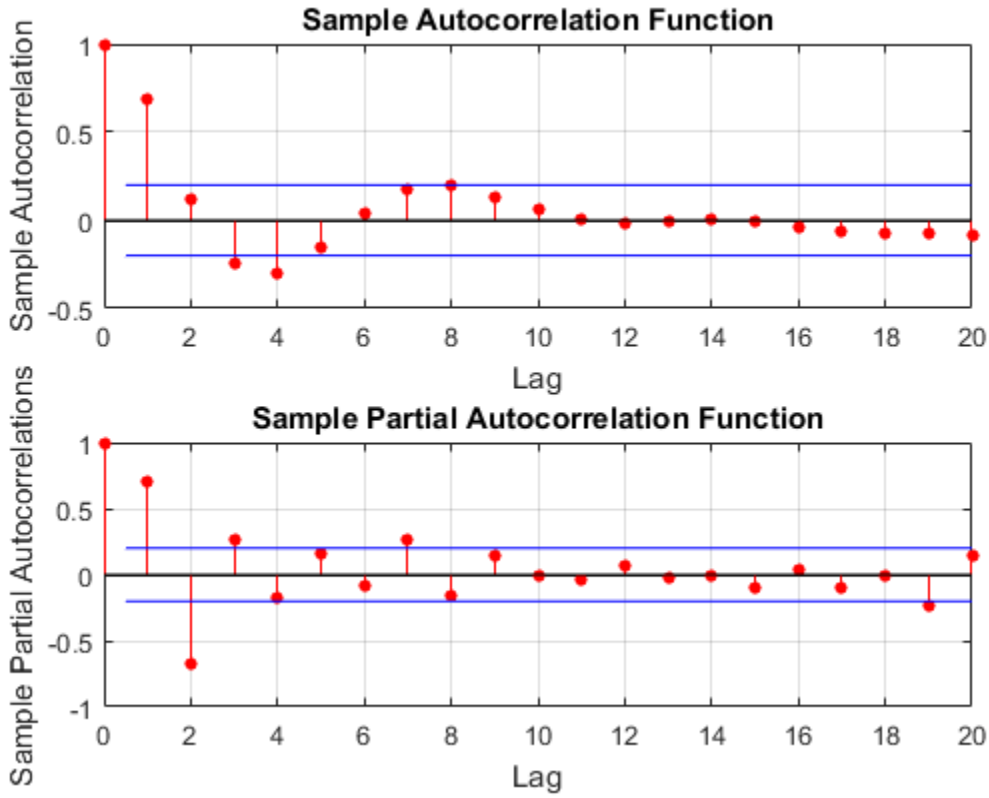
```
modSim = arima('Constant',0.2,'AR',{0.75,-0.4},...
               'MA',0.7,'Variance',0.1);
rng('default')
Y = simulate(modSim,100);

figure
plot(Y)
xlim([0,100])
title('Simulated ARMA(2,1) Series')
```



Step 2: Plot the sample ACF and PACF.

Plot the sample autocorrelation function (ACF) and partial autocorrelation function (PACF) for the simulated data.

```
figure
subplot(2,1,1)
autocorr(Y)
subplot(2,1,2)
parcorr(Y)
```



Both the sample ACF and PACF decay relatively slowly. This is consistent with an ARMA model. The ARMA lags cannot be selected solely by looking at the ACF and PACF, but it seems no more than four AR or MA terms are needed.

Step 3. Fit ARMA(p,q) models.

To identify the best lags, fit several models with different lag choices. Here, fit all combinations of $p = 1,...,4$ and $q = 1,...,4$ (a total of 16 models). Store the loglikelihood objective function and number of coefficients for each fitted model.

```
LOGL = zeros(4,4); %Initialize
PQ = zeros(4,4);
for p = 1:4
    for q = 1:4
```

```
        mod = arima(p,0,q);
        [fit,~,logL] = estimate(mod,Y,'print',false);
        LOGL(p,q) = logL;
        PQ(p,q) = p+q;
    end
end
```

Step 4: Calculate the BIC.

Calculate the BIC for each fitted model. The number of parameters in a model is $p + q + 1$ (for the AR and MA coefficients, and constant term). The number of observations in the data set is 100.

```
LOGL = reshape(LOGL,16,1);
PQ = reshape(PQ,16,1);
[~,bic] = aicbic(LOGL,PQ+1,100);
reshape(bic,4,4)
```

```
ans =

  108.6241  105.9489  109.4164  113.8443
   99.1639  101.5886  105.5203  109.4348
  102.9094  106.0305  107.6489   99.6794
  107.4045  100.7072  102.5746  102.0209
```

In the output BIC matrix, the rows correspond to the AR degree (p) and the columns correspond to the MA degree (q). The smallest value is best.

The smallest BIC value is `99.1639` in the (2,1) position. This corresponds to an ARMA(2,1) model, matching the model that generated the data.

See Also

`aicbic` | arima | `autocorr` | estimate | `parcorr` | simulate

Related Examples

- “Detect Autocorrelation” on page 3-18
- “Estimate Conditional Mean and Variance Models” on page 5-129

More About

- “Autoregressive Moving Average Model” on page 5-34

- "Information Criteria" on page 3-63

Infer Residuals for Diagnostic Checking

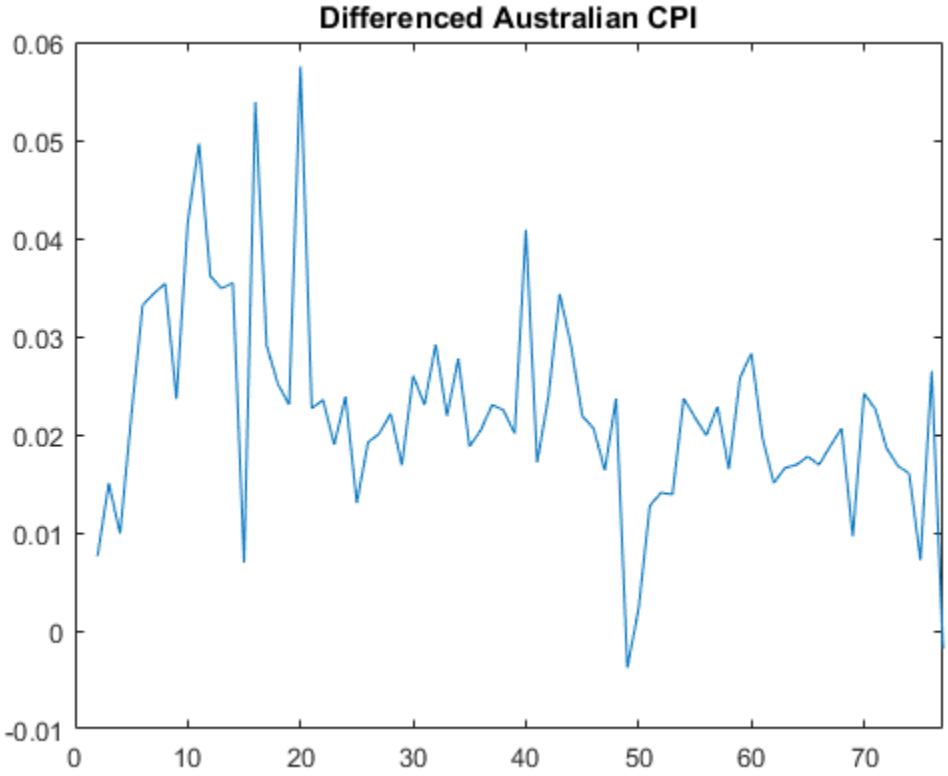
This example shows how to infer residuals from a fitted ARIMA model. Diagnostic checks are performed on the residuals to assess model fit.

The time series is the log quarterly Australian Consumer Price Index (CPI) measured from 1972 to 1991.

Load the Data.

Load the Australian CPI data. Take first differences, then plot the series.

```
load Data_JAustralian
y = DataTable.PAU;
T = length(y);
dY = diff(y);

figure
plot(2:T,dY)
xlim([0,T])
title('Differenced Australian CPI')
```

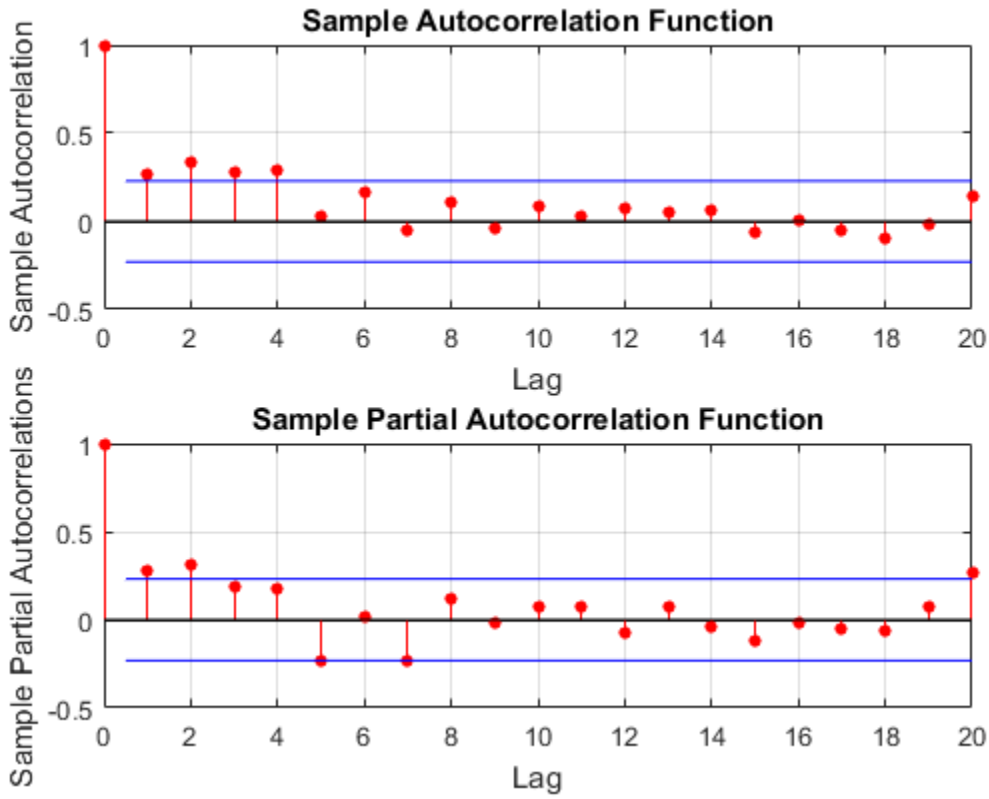


The differenced series looks relatively stationary.

Plot the Sample ACF and PACF.

Plot the sample autocorrelation function (ACF) and partial autocorrelation function (PACF) to look for autocorrelation in the differenced series.

```
figure
subplot(2,1,1)
autocorr(dY)
subplot(2,1,2)
parcorr(dY)
```



The sample ACF decays more slowly than the sample PACF. The latter cuts off after lag 2. This, along with the first-degree differencing, suggests an ARIMA(2,1,0) model.

Estimate an ARIMA(2,1,0) Model.

Specify, and then estimate, an ARIMA(2,1,0) model. Infer the residuals for diagnostic checking.

```
Mdl = arima(2,1,0);
EstMdl = estimate(Mdl,y);
[res,~,logL] = infer(EstMdl,y);
```

```
    ARIMA(2,1,0) Model:
```

```
--------------------
Conditional Probability Distribution: Gaussian

                           Standard          t
 Parameter       Value       Error       Statistic
-----------   -----------   ------------   -----------
 Constant      0.0100723    0.00328015        3.07069
    AR{1}       0.212059     0.0954278        2.22219
    AR{2}       0.337282      0.103781        3.24994
 Variance    9.23017e-05   1.11119e-05        8.30659
```

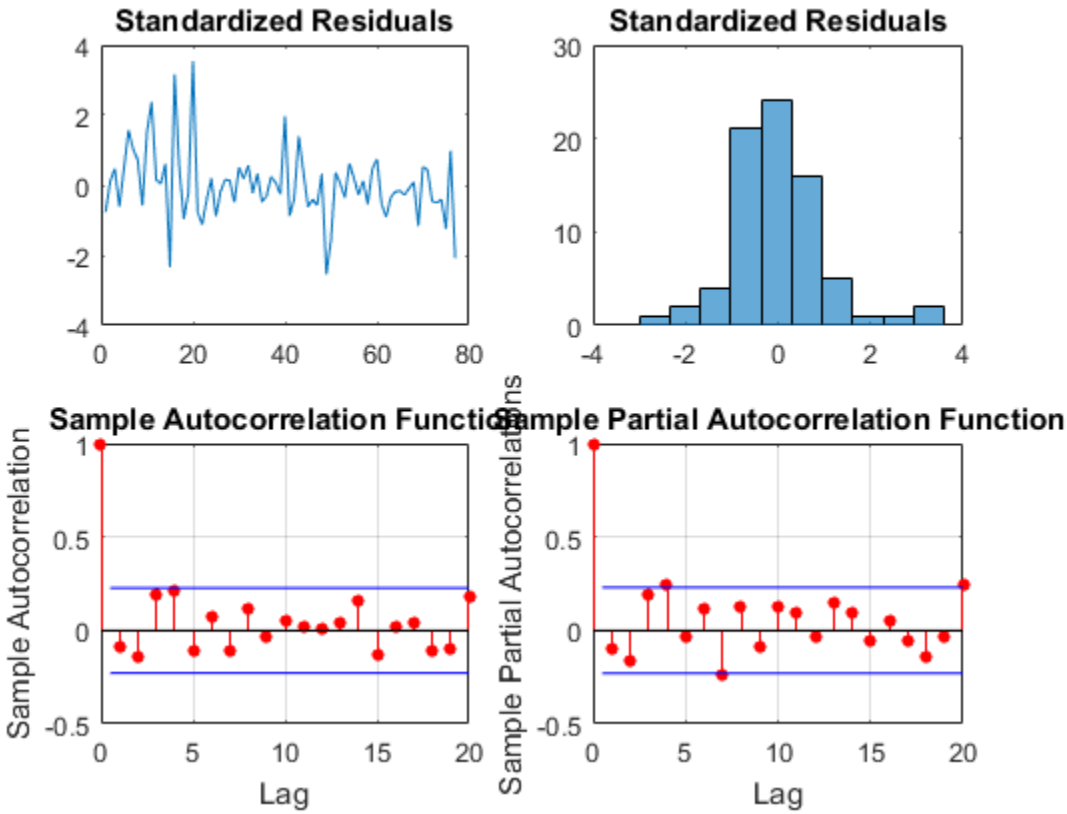
Notice that the model is fit to the original series, and not the differenced series. The model to be fit, `Mdl`, has property `D` equal to `1`. This accounts for the one degree of differencing.

This specification assumes a Gaussian innovation distribution. `infer` returns the value of the loglikelihood objective function (`logL`) along with the residuals (`res`).

Perform Residual Diagnostic Checks.

Standardize the inferred residuals, and check for normality and any unexplained autocorrelation.

```
stdr = res/sqrt(EstMdl.Variance);

figure
subplot(2,2,1)
plot(stdr)
title('Standardized Residuals')
subplot(2,2,2)
histogram(stdr,10)
title('Standardized Residuals')
subplot(2,2,3)
autocorr(stdr)
subplot(2,2,4)
parcorr(stdr)
```



The residuals appear uncorrelated and approximately normally distributed. There is some indication that there is an excess of large residuals.

Modify the Innovation Distribution.

To explore possible excess kurtosis in the innovation process, fit an ARIMA(2,1,0) model with a Student's t distribution to the original series. Return the value of the loglikelihood objective function so you can use the Bayesian information criterion (BIC) to compare the fit of the two models.

```
MdlT = Mdl;
MdlT.Distribution = 't';
[EstMdlT,~,logLT] = estimate(MdlT,y);
[~,bic] = aicbic([logLT,logL],[5,4],T)
```

```
    ARIMA(2,1,0) Model:
    --------------------
    Conditional Probability Distribution: t

                                Standard          t
     Parameter       Value        Error       Statistic
    -----------   -----------   ------------   -----------
     Constant     0.00997449    0.00161524        6.17524
        AR{1}       0.326887     0.0755033        4.32943
        AR{2}       0.187194     0.0746907        2.50625
     Variance    0.000247205   0.000746201       0.331284
          DoF        2.25938      0.955621         2.3643

bic =

 -492.5317 -479.4691
```

The models with the t-innovation distribution (`MdlT` and `EstMdlT`) have one extra parameter (the degrees of freedom of the t distribution).

According to the BIC, the ARIMA(2,1,0) model with a Student's t innovation distribution is the better choice because it has a smaller (more negative) BIC value.

See Also

`aicbic` | arima | estimate | infer

Related Examples

- “Box-Jenkins Differencing vs. ARIMA Estimation” on page 5-94
- “Specify Conditional Mean Model Innovation Distribution” on page 5-72

More About

- “Information Criteria” on page 3-63
- “Goodness of Fit” on page 3-88
- “Residual Diagnostics” on page 3-90

Monte Carlo Simulation of Conditional Mean Models

In this section...

What Is Monte Carlo Simulation?

Monte Carlo simulation is the process of generating independent, random draws from a specified probabilistic model. When simulating time series models, one draw (or realization) is an entire sample path of specified length N, y_1, y_2,...,y_N. When you generate a large number of draws, say M, you generate M sample paths, each of length N.

Note: Some extensions of Monte Carlo simulation rely on generating dependent random draws, such as Markov Chain Monte Carlo (MCMC). The `simulate` function in Econometrics Toolbox generates independent realizations.

Some applications of Monte Carlo simulation are:

- Demonstrating theoretical results
- Forecasting future events
- Estimating the probability of future events

Generate Monte Carlo Sample Paths

Conditional mean models specify the dynamic evolution of a process over time through the conditional mean structure. To perform Monte Carlo simulation of conditional mean models:

1. Specify presample data (or use default presample data).
2. Generate an uncorrelated innovation series from the innovation distribution that you specified.
3. Generate responses by recursively applying the specified AR and MA polynomial operators. The AR polynomial operator can include differencing.

For example, consider an AR(2) process,

$$y_t = c + \phi_1 y_{t-1} + \phi_2 y_{t-2} + \varepsilon_t.$$

Given presample responses y_0 and y_{-1}, and simulated innovations $\varepsilon_1,\ldots,\varepsilon_N$, realizations of the process are recursively generated:

- $y_1 = c + \phi_1 y_0 + \phi_2 y_{-1} + \varepsilon_1$
- $y_2 = c + \phi_1 y_1 + \phi_2 y_0 + \varepsilon_2$
- $y_3 = c + \phi_1 y_2 + \phi_2 y_1 + \varepsilon_3$

⋮

- $y_N = c + \phi_1 y_{N-1} + \phi_2 y_{N-2} + \varepsilon_N$

For an MA(12) process, e.g.,

$$y_t = c + \varepsilon_t + \theta_1 \varepsilon_{t-1} + \theta_{12} \varepsilon_{t-12},$$

you need 12 presample innovations to initialize the simulation. By default, `simulate` sets presample innovations equal to zero. The remaining N innovations are randomly sampled from the innovation process.

Monte Carlo Error

Using many simulated paths, you can estimate various features of the model. However, Monte Carlo estimation is based on a finite number of simulations. Therefore, Monte Carlo estimates are subject to some amount of error. You can reduce the amount of Monte Carlo error in your simulation study by increasing the number of sample paths, M, that you generate from your model.

For example, to estimate the probability of a future event:

1. Generate M sample paths from your model.
2. Estimate the probability of the future event using the sample proportion of the event occurrence across M simulations,

$$\hat{p} = \frac{\#\ times\ event\ occurs\ in\ M\ draws}{M}.$$

3 Calculate the Monte Carlo standard error for the estimate,

$$se = \sqrt{\frac{\hat{p}(1-\hat{p})}{M}}.$$

You can reduce the Monte Carlo error of the probability estimate by increasing the number of realizations. If you know the desired precision of your estimate, you can solve for the number of realizations needed to achieve that level of precision.

See Also

arima | simulate

Related Examples

- "Simulate Stationary Processes" on page 5-151
- "Simulate Trend-Stationary and Difference-Stationary Processes" on page 5-163
- "Simulate Multiplicative ARIMA Models" on page 5-169
- "Simulate Conditional Mean and Variance Models" on page 5-175
- "Forecast IGD Rate Using ARIMAX Model" on page 5-122

More About

- "Presample Data for Conditional Mean Model Simulation" on page 5-149
- "Transient Effects in Conditional Mean Model Simulations" on page 5-150
- "Monte Carlo Forecasting of Conditional Mean Models" on page 5-181

Presample Data for Conditional Mean Model Simulation

When simulating realizations from ARIMA processes, you need presample responses and presample innovations to initialize the conditional mean model. The number of presample responses you need to initialize a simulation are stored in the `arima` model property `P`. The number of presample innovations you need to initialize a simulation are stored in the property `Q`. You can specify your own presample data or let `simulate` generate presample data.

If you let `simulate` generate default presample data, then:

- For stationary processes, `simulate` sets presample responses to the unconditional mean of the process.
- For nonstationary processes, `simulate` sets presample responses to 0.
- Presample innovations are set to 0.

If you specify a matrix of exogenous covariates, then `simulate` sets the presample responses to 0.

See Also

arima | simulate

Related Examples

- “Simulate Stationary Processes” on page 5-151
- “Simulate Trend-Stationary and Difference-Stationary Processes” on page 5-163
- “Simulate Multiplicative ARIMA Models” on page 5-169
- “Simulate Conditional Mean and Variance Models” on page 5-175

More About

- “Monte Carlo Simulation of Conditional Mean Models” on page 5-146
- “Transient Effects in Conditional Mean Model Simulations” on page 5-150
- “Monte Carlo Forecasting of Conditional Mean Models” on page 5-181

Transient Effects in Conditional Mean Model Simulations

When you use automatically generated presample data, you often see some transient effects at the beginning of the simulation. This is sometimes called a *burn-in period*. For stationary processes, the impulse response function decays to zero over time. This means the starting point of the simulation is eventually forgotten. To reduce transient effects, you can:

- *Oversample*: generate sample paths longer than needed, and discard the beginning samples that show transient effects.
- *Recycle*: use a first simulation to generate presample data for a second simulation.

For nonstationary processes, the starting point is never forgotten. By default, all realizations of nonstationary processes begin at zero. For a nonzero starting point, you need to specify your own presample data.

Related Examples

- "Simulate Stationary Processes" on page 5-151
- "Simulate Trend-Stationary and Difference-Stationary Processes" on page 5-163

More About

- "Monte Carlo Simulation of Conditional Mean Models" on page 5-146
- "Presample Data for Conditional Mean Model Simulation" on page 5-149
- "Monte Carlo Forecasting of Conditional Mean Models" on page 5-181

Simulate Stationary Processes

In this section...

Simulate an AR Process

This example shows how to simulate sample paths from a stationary AR(2) process without specifying presample observations.

Step 1. Specify a model.

Specify the AR(2) model

$$y_t = 0.5 + 0.7y_{t-1} + 0.25y_{t-2} + \varepsilon_t,$$

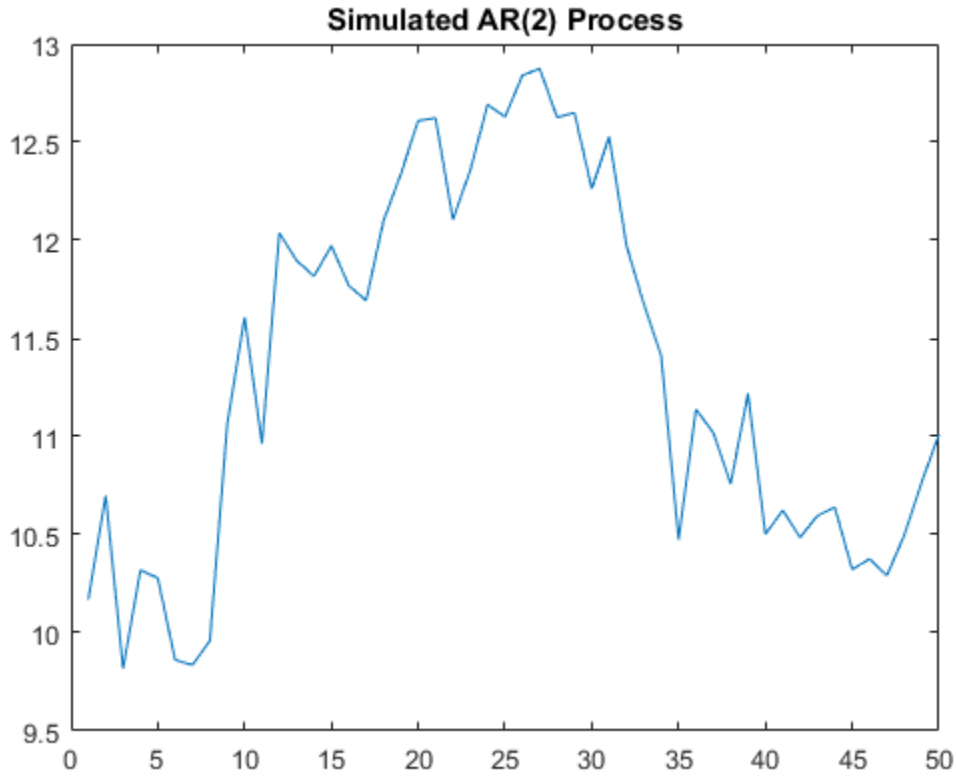
where the innovation process is Gaussian with variance 0.1.

```
model = arima('Constant',0.5,'AR',{0.7,0.25},'Variance',.1);
```

Step 2. Generate one sample path.

Generate one sample path (with 50 observations) from the specified model, and plot.

```
rng('default')
Y = simulate(model,50);

figure
plot(Y)
xlim([0,50])
title('Simulated AR(2) Process')
```



Because presample data was not specified, `simulate` sets the two required presample observations equal to the unconditional mean of the process,

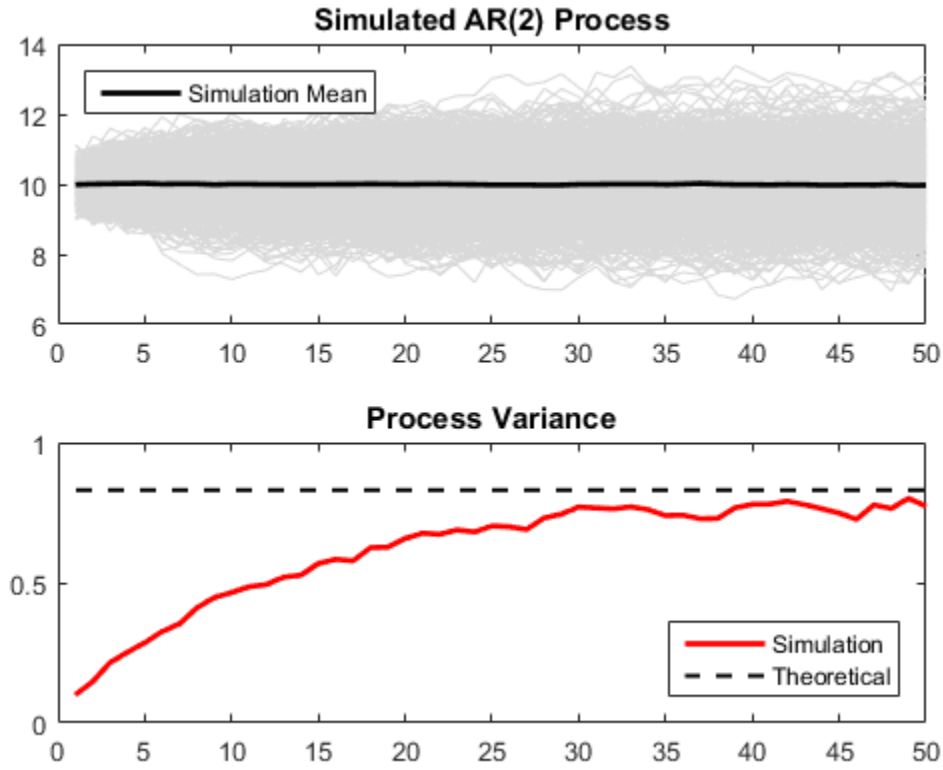
$$\frac{c}{(1-\phi_1-\phi_2)} = \frac{0.5}{(1-0.7-0.25)} = 10.$$

Step 3. Generate many sample paths.

Generate 1000 sample paths, each with 50 observations.

```
rng('default')
Y = simulate(model,50,'NumPaths',1000);

figure
```

```
subplot(2,1,1)
plot(Y,'Color',[.85,.85,.85])
title('Simulated AR(2) Process')
hold on
h=plot(mean(Y,2),'k','LineWidth',2);
legend(h,'Simulation Mean','Location','NorthWest')
hold off
subplot(2,1,2)
plot(var(Y,0,2),'r','LineWidth',2)
title('Process Variance')
hold on
plot(1:50,.83*ones(50,1),'k--','LineWidth',1.5)
legend('Simulation','Theoretical',...
       'Location','SouthEast')
hold off
```



The simulation mean is constant over time. This is consistent with the definition of a stationary process. The process variance is not constant over time, however. There are transient effects at the beginning of the simulation due to the absence of presample data.

The simulated variance approaches the theoretical variance,

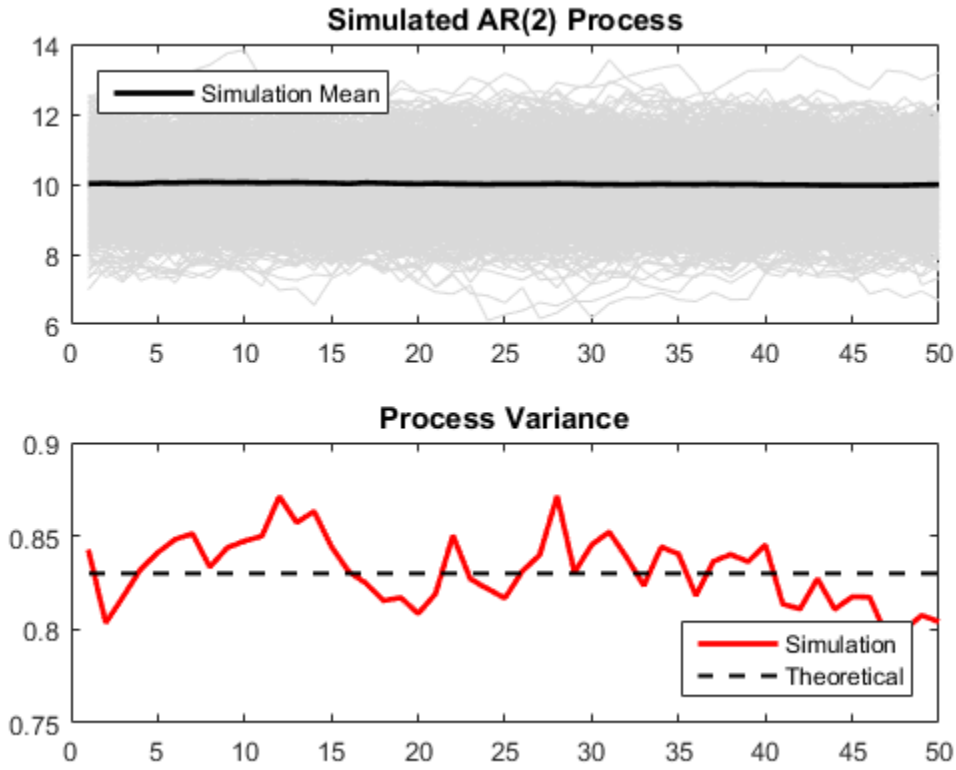
$$\frac{(1-\phi_2)}{(1+\phi_2)}\frac{\sigma_\varepsilon^2}{(1-\phi_2)^2-\phi_1^2} = \frac{(1-.25)}{(1+.25)}\frac{0.1}{(1-.25)^2-0.7^2} = 0.83,$$

by around the 50th observation.

Step 4. Oversample the process.

To reduce transient effects, one option is to oversample the process. For example, to sample 50 observations, you can generate paths with more than 50 observations, and discard all but the last 50 observations as burn-in. Here, simulate paths of length 150, and discard the first 100 observations.

```
rng('default')
Y = simulate(model,150,'NumPaths',1000);
Y = Y(101:end,:);

figure
subplot(2,1,1)
plot(Y,'Color',[.85,.85,.85])
title('Simulated AR(2) Process')
hold on
h=plot(mean(Y,2),'k','LineWidth',2);
legend(h,'Simulation Mean','Location','NorthWest')
hold off
subplot(2,1,2)
plot(var(Y,0,2),'r','LineWidth',2)
xlim([0,50])
title('Process Variance')
hold on
plot(1:50,.83*ones(50,1),'k--','LineWidth',1.5)
legend('Simulation','Theoretical',...
       'Location','SouthEast')
hold off
```



The realizations now look like draws from a stationary stochastic process. The simulation variance fluctuates (due to Monte Carlo error) around the theoretical variance.

Simulate an MA Process

This example shows how to simulate sample paths from a stationary MA(12) process without specifying presample observations.

Step 1. Specify a model.

Specify the MA(12) model

$$y_t = 0.5 + \varepsilon_t + 0.8\varepsilon_{t-1} + 0.2\varepsilon_{t-12},$$

where the innovation distribution is Gaussian with variance 0.2.

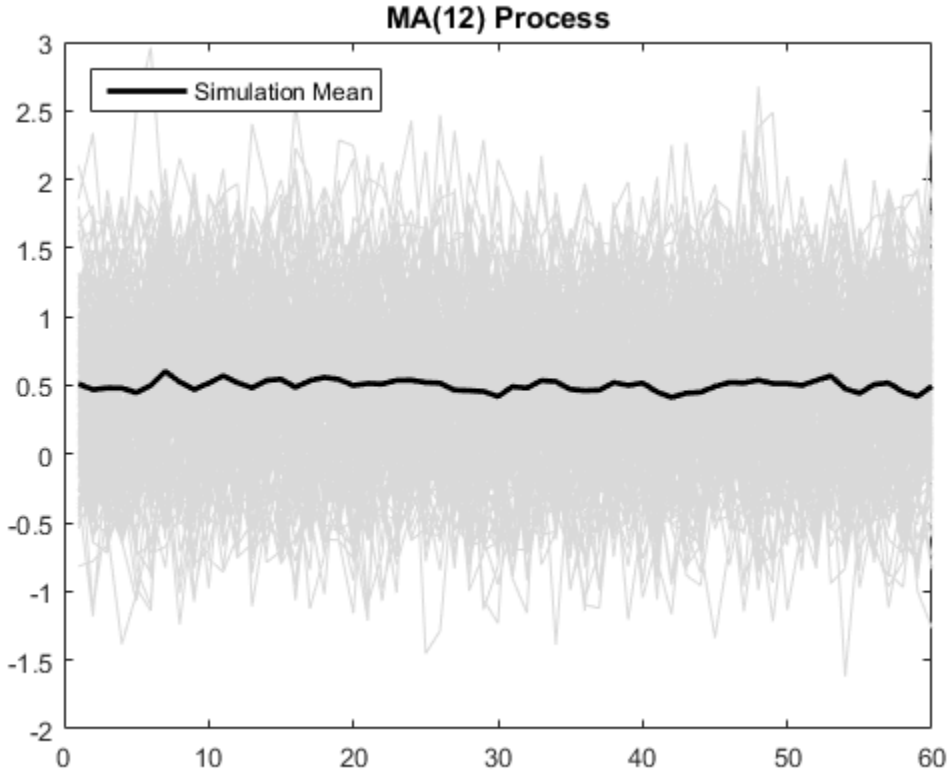
```
model = arima('Constant',0.5,'MA',{0.8,0.2},...
              'MALags',[1,12],'Variance',0.2);
```

Step 2. Generate sample paths.

Generate 200 sample paths, each with 60 observations.

```
rng('default')
Y = simulate(model,60,'NumPaths',200);

figure
plot(Y,'Color',[.85,.85,.85])
hold on
h = plot(mean(Y,2),'k','LineWidth',2)
legend(h,'Simulation Mean','Location','NorthWest')
title('MA(12) Process')
hold off
```

```
h =

  Line with properties:

            Color: [0 0 0]
        LineStyle: '-'
        LineWidth: 2
           Marker: 'none'
       MarkerSize: 6
  MarkerFaceColor: 'none'
            XData: [1x60 double]
            YData: [1x60 double]
            ZData: [1x0 double]

  Use GET to show all properties
```



For an MA process, the constant term is the unconditional mean. The simulation mean is around 0.5, as expected.

Step 3. Plot the simulation variance.

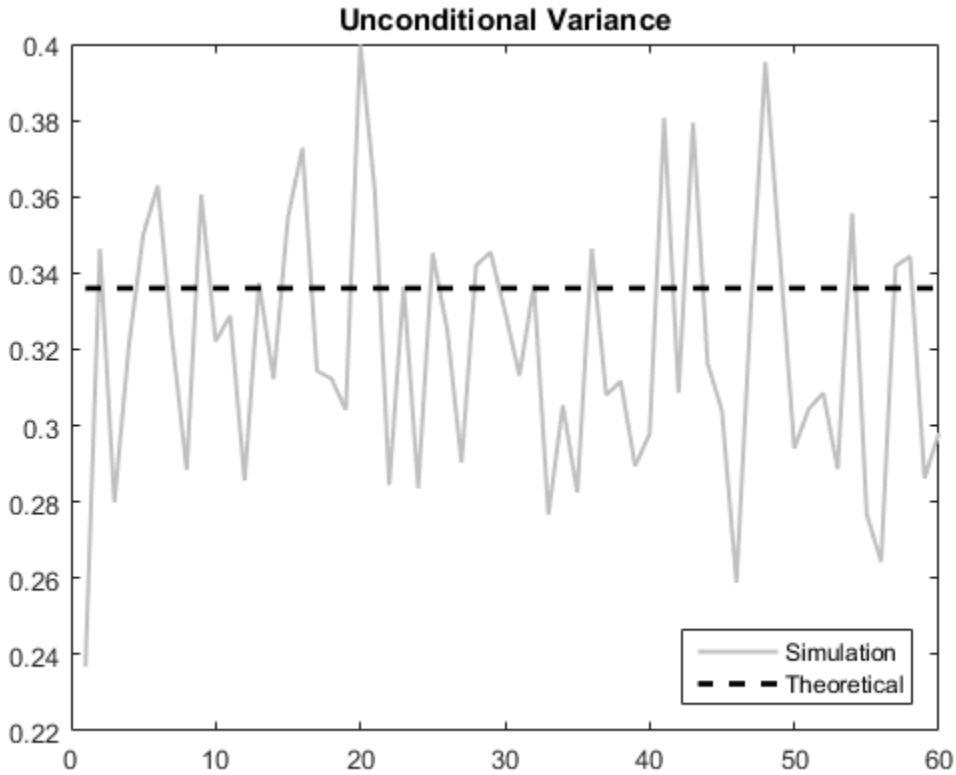
The unconditional variance for the model is

$$(1+\theta_1^2+\theta_{12}^2)\sigma_\varepsilon^2 = (1+0.8^2+0.2^2)\times 0.2 = 0.336.$$

Because the model is stationary, the unconditional variance should be constant across all times. Plot the simulation variance, and compare it to the theoretical variance.

```
figure
```

```
plot(var(Y,0,2),'Color',[.75,.75,.75],'LineWidth',1.5)
xlim([0,60])
title('Unconditional Variance')
hold on
plot(1:60,.336*ones(60,1),'k--','LineWidth',2)
legend('Simulation','Theoretical',...
       'Location','SouthEast')
hold off
```

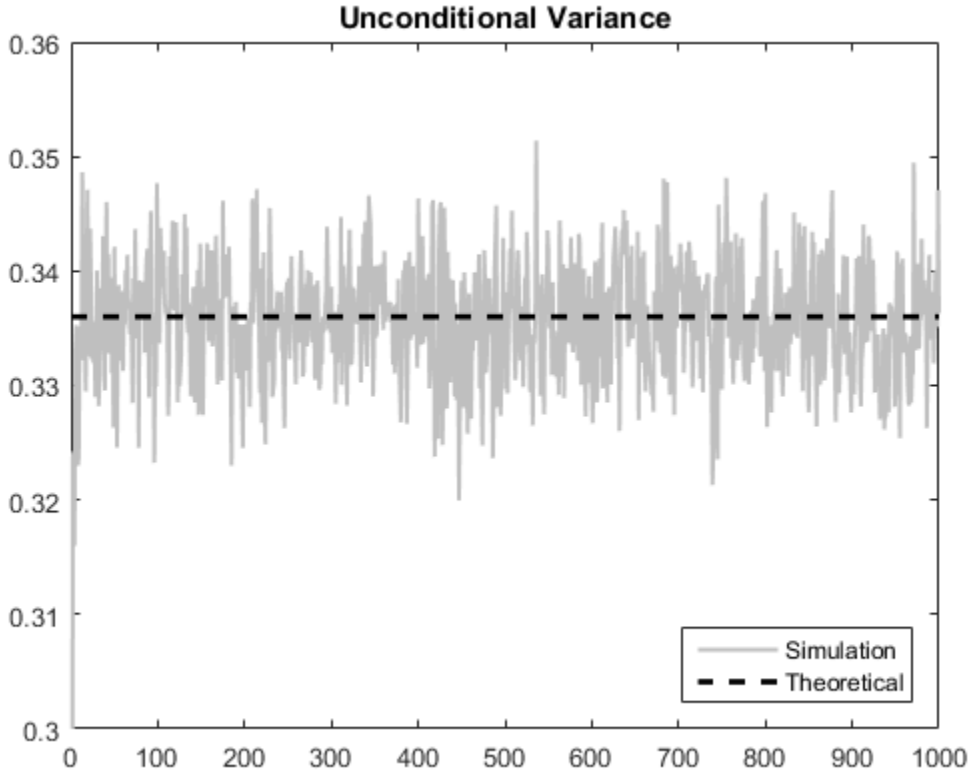


There appears to be a short burn-in period at the beginning of the simulation. During this time, the simulation variance is lower than expected. Afterwards, the simulation variance fluctuates around the theoretical variance.

Step 4. Generate more sample paths.

Simulate 10,000 paths from the model, each with length 1000. Look at the simulation variance.

```
rng('default')
YM = simulate(model,1000,'NumPaths',10000);
figure
plot(var(YM,0,2),'Color',[.75,.75,.75],'LineWidth',1.5)
ylim([0.3,0.36])
title('Unconditional Variance')
hold on
plot(1:1000,.336*ones(1000,1),'k--','LineWidth',2)
legend('Simulation','Theoretical',...
       'Location','SouthEast')
hold off
```



The Monte Carlo error is reduced when more realizations are generated. There is much less variability in the simulation variance, which tightly fluctuates around the theoretical variance.

See Also

arima | simulate

Related Examples

- "Simulate Trend-Stationary and Difference-Stationary Processes" on page 5-163
- "Simulate Multiplicative ARIMA Models" on page 5-169
- "Simulate Conditional Mean and Variance Models" on page 5-175

More About

- "Autoregressive Model" on page 5-18
- "Moving Average Model" on page 5-27
- "Monte Carlo Simulation of Conditional Mean Models" on page 5-146
- "Transient Effects in Conditional Mean Model Simulations" on page 5-150

Simulate Trend-Stationary and Difference-Stationary Processes

This example shows how to simulate trend-stationary and difference-stationary processes. The simulation results illustrate the distinction between these two nonstationary process models.

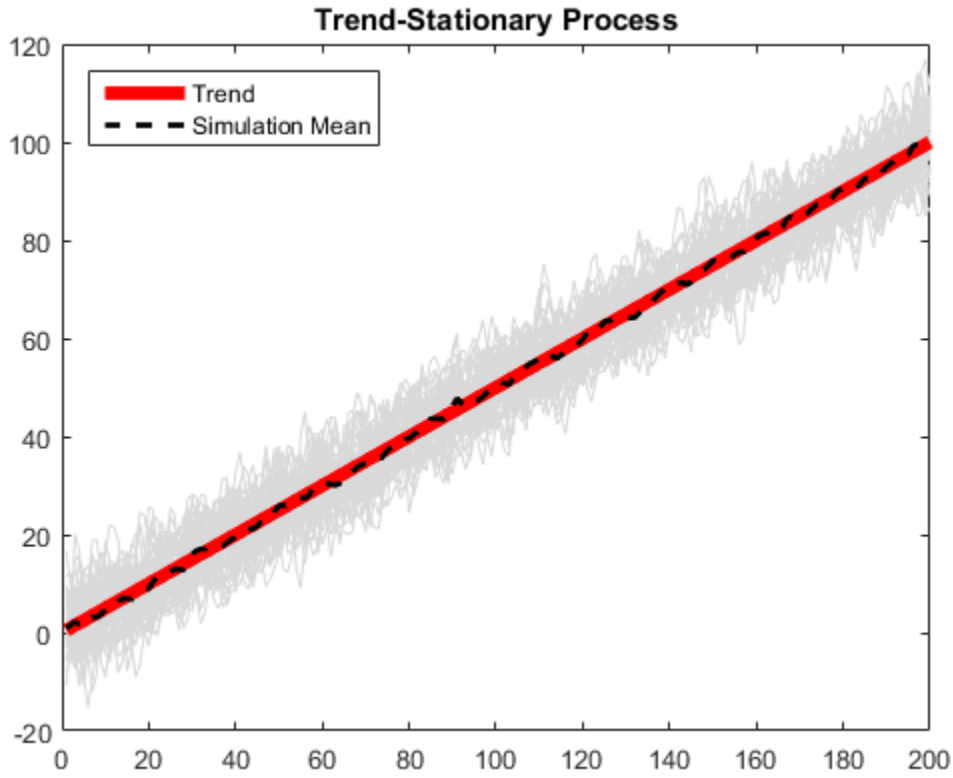
Step 1. Generate realizations from a trend-stationary process.

Specify the trend-stationary process

$$y_t = 0.5t + \varepsilon_t + 1.4\varepsilon_{t-1} + 0.8\varepsilon_{t-2},$$

where the innovation process is Gaussian with variance 8. After specifying the model, simulate 50 sample paths of length 200. Use 100 burn-in simulations.

```
t = [1:200]';
trend = 0.5*t;

model = arima('Constant',0,'MA',{1.4,0.8},'Variance',8);
rng('default')
u = simulate(model,300,'NumPaths',50);

Yt = repmat(trend,1,50) + u(101:300,:);


figure
plot(Yt,'Color',[.85,.85,.85])
hold on
h1=plot(t,trend,'r','LineWidth',5);
xlim([0,200])
title('Trend-Stationary Process')
h2=plot(mean(Yt,2),'k--','LineWidth',2);
legend([h1,h2],'Trend','Simulation Mean',...
       'Location','NorthWest')
hold off
```



The sample paths fluctuate around the theoretical trend line with constant variance. The simulation mean is close to the true trend line.

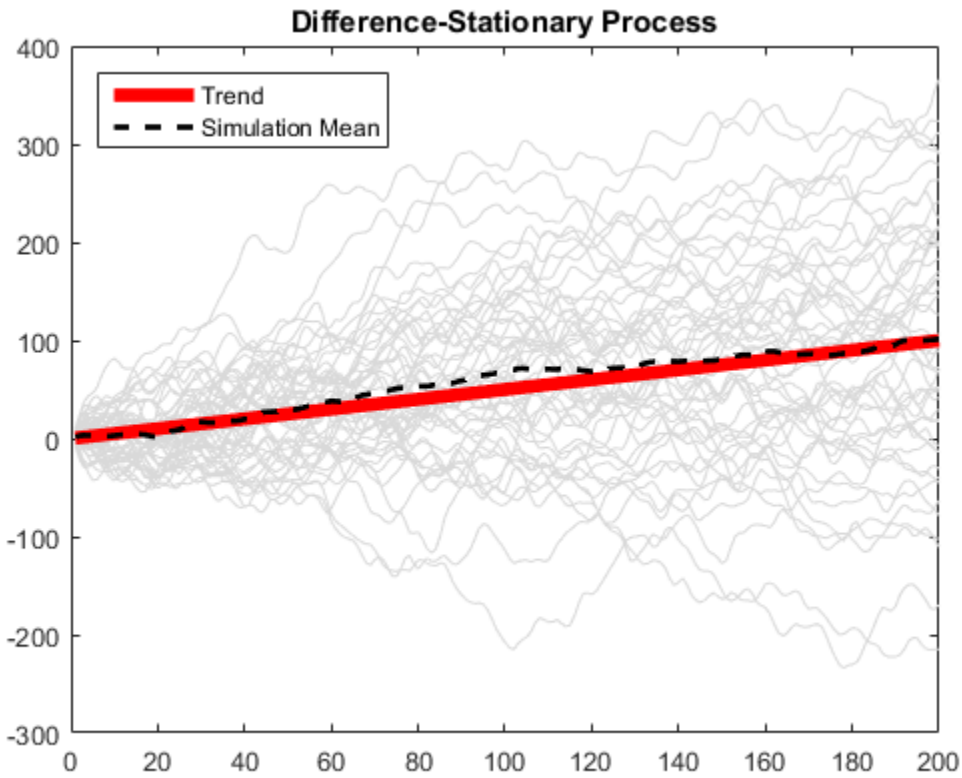
Step 2. Generate realizations from a difference-stationary process.

Specify the difference-stationary model

$$\Delta y_t = 0.5 + \varepsilon_t + 1.4\varepsilon_{t-1} + 0.8\varepsilon_{t-1},$$

where the innovation distribution is Gaussian with variance 8. After specifying the model, simulate 50 sample paths of length 200. No burn-in is needed because all sample paths should begin at zero. This is the `simulate` default starting point for nonstationary processes with no presample data.

```
model = arima('Constant',0.5,'D',1,'MA',{1.4,0.8},...
              'Variance',8);
rng('default')
Yd = simulate(model,200,'NumPaths',50);
figure
plot(Yd,'Color',[.85,.85,.85])
hold on
h1=plot(t,trend,'r','LineWidth',5);
xlim([0,200])
title('Difference-Stationary Process')
h2=plot(mean(Yd,2),'k--','LineWidth',2);
legend([h1,h2],'Trend','Simulation Mean',...
       'Location','NorthWest')
hold off
```

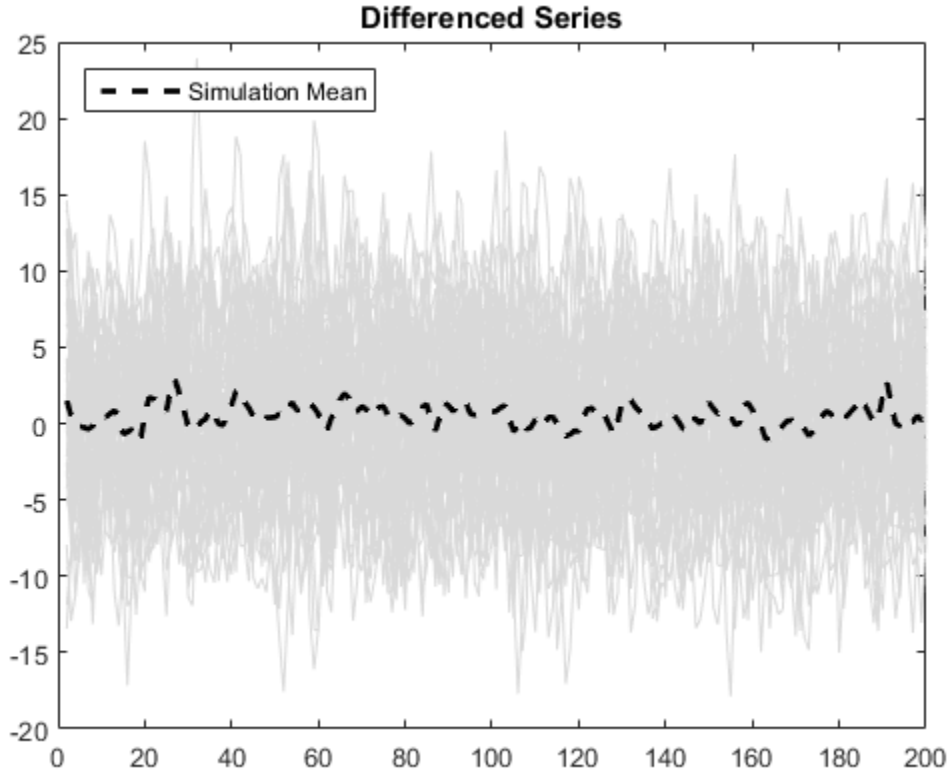


The simulation average is close to the trend line with slope 0.5. The variance of the sample paths grows over time.

Step 3. Difference the sample paths.

A difference-stationary process is stationary when differenced appropriately. Take the first differences of the sample paths from the difference-stationary process, and plot the differenced series. One observation is lost as a result of the differencing.

```
diffY = diff(Yd,1,1);

figure
plot(2:200,diffY,'Color',[.85,.85,.85])
xlim([0,200])
title('Differenced Series')
hold on
h = plot(2:200,mean(diffY,2),'k--','LineWidth',2);
legend(h,'Simulation Mean','Location','NorthWest')
hold off
```



The differenced series looks stationary, with the simulation mean fluctuating around zero.

See Also

arima | simulate

Related Examples

- "Simulate Stationary Processes" on page 5-151

More About

- "Trend-Stationary vs. Difference-Stationary Processes" on page 2-7

Simulate Multiplicative ARIMA Models

This example shows how to simulate sample paths from a multiplicative seasonal ARIMA model using `simulate`. The time series is monthly international airline passenger numbers from 1949 to 1960.

Load the Data and Estimate a Model.

Load the data set `Data_Airline`.

```
load(fullfile(matlabroot,'examples','econ','Data_Airline.mat'))
y = log(Data);
T = length(y);

Mdl = arima('Constant',0,'D',1,'Seasonality',12,...
    'MALags',1,'SMALags',12);
EstMdl = estimate(Mdl,y);
res = infer(EstMdl,y);
```

```
    ARIMA(0,1,1) Model Seasonally Integrated with Seasonal MA(12):
    --------------------------------------------------------------
    Conditional Probability Distribution: Gaussian

                               Standard          t
     Parameter       Value       Error       Statistic
    -----------   -----------   ------------   -----------
     Constant             0         Fixed          Fixed
        MA{1}      -0.377162     0.0667944       -5.64661
      SMA{12}      -0.572378     0.0854395       -6.69923
     Variance     0.00126337    0.00012395        10.1926
```

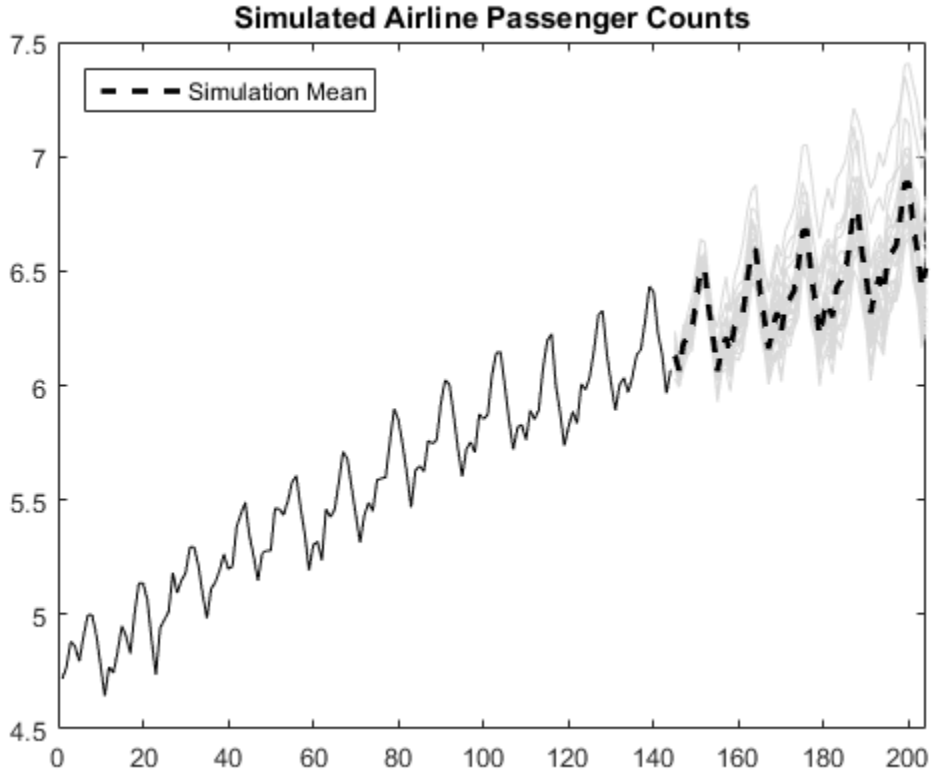
Simulate Airline Passenger Counts.

Use the fitted model to simulate 25 realizations of airline passenger counts over a 60-month (5-year) horizon. Use the observed series and inferred residuals as presample data.

```
rng 'default'
Ysim = simulate(EstMdl,60,'NumPaths',25,'Y0',y,'E0',res);
mn = mean(Ysim,2);

figure
```

```
plot(y,'k')
hold on
plot(T+1:T+60,Ysim,'Color',[.85,.85,.85]);
h = plot(T+1:T+60,mn,'k--','LineWidth',2)
xlim([0,T+60])
title('Simulated Airline Passenger Counts')
legend(h,'Simulation Mean','Location','NorthWest')
hold off
```

```
h =

  Line with properties:

              Color: [0 0 0]
          LineStyle: '--'
          LineWidth: 2
             Marker: 'none'
         MarkerSize: 6
    MarkerFaceColor: 'none'
              XData: [1x60 double]
              YData: [1x60 double]
              ZData: [1x0 double]

  Use GET to show all properties
```



The simulated forecasts show growth and seasonal periodicity similar to the observed series.

Estimate the Probability of a Future Event.

Use simulations to estimate the probability that log airline passenger counts will meet or exceed the value 7 sometime during the next 5 years. Calculate the Monte Carlo error associated with the estimated probability.

```
rng default
Ysim = simulate(EstMdl,60,'NumPaths',1000,'Y0',y,'E0',res);

g7 = sum(Ysim >= 7) > 0;
phat = mean(g7)
```

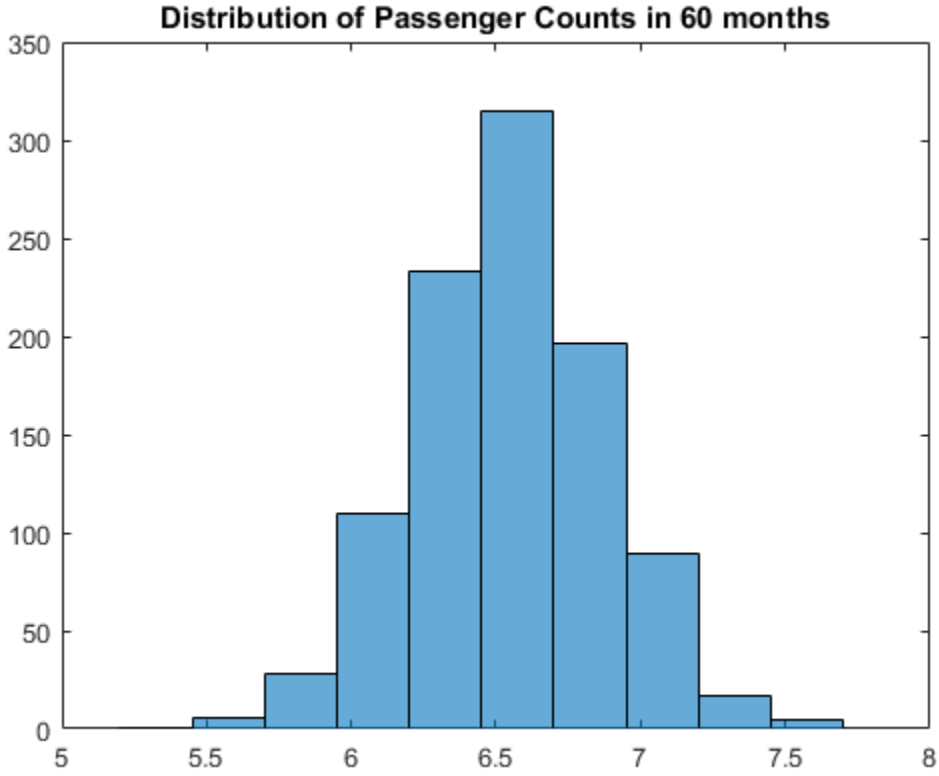
```
err = sqrt(phat*(1-phat)/1000)


phat =

    0.3910


err =

    0.0154
```

There is approximately a 39% chance that the (log) number of airline passengers will meet or exceed 7 in the next 5 years. The Monte Carlo standard error of the estimate is about 0.02.

Plot the Distribution of Passengers at a Future Time.

Use the simulations to plot the distribution of (log) airline passenger counts 60 months into the future.

```
figure
histogram(Ysim(60,:),10)
title('Distribution of Passenger Counts in 60 months')
```



See Also

arima | estimate | infer | simulate

Related Examples

- "Specify Multiplicative ARIMA Model" on page 5-52
- "Estimate Multiplicative ARIMA Model" on page 5-113
- "Forecast Multiplicative ARIMA Model" on page 5-192
- "Check Fit of Multiplicative ARIMA Model" on page 3-81

More About

- "Monte Carlo Simulation of Conditional Mean Models" on page 5-146
- "Presample Data for Conditional Mean Model Simulation" on page 5-149
- "Monte Carlo Forecasting of Conditional Mean Models" on page 5-181

Simulate Conditional Mean and Variance Models

This example shows how to simulate responses and conditional variances from a composite conditional mean and variance model.

Load the Data and Fit a Model

Load the NASDAQ data included with the toolbox. Fit a conditional mean and variance model to the daily returns. Scale the returns to percentage returns for numerical stability

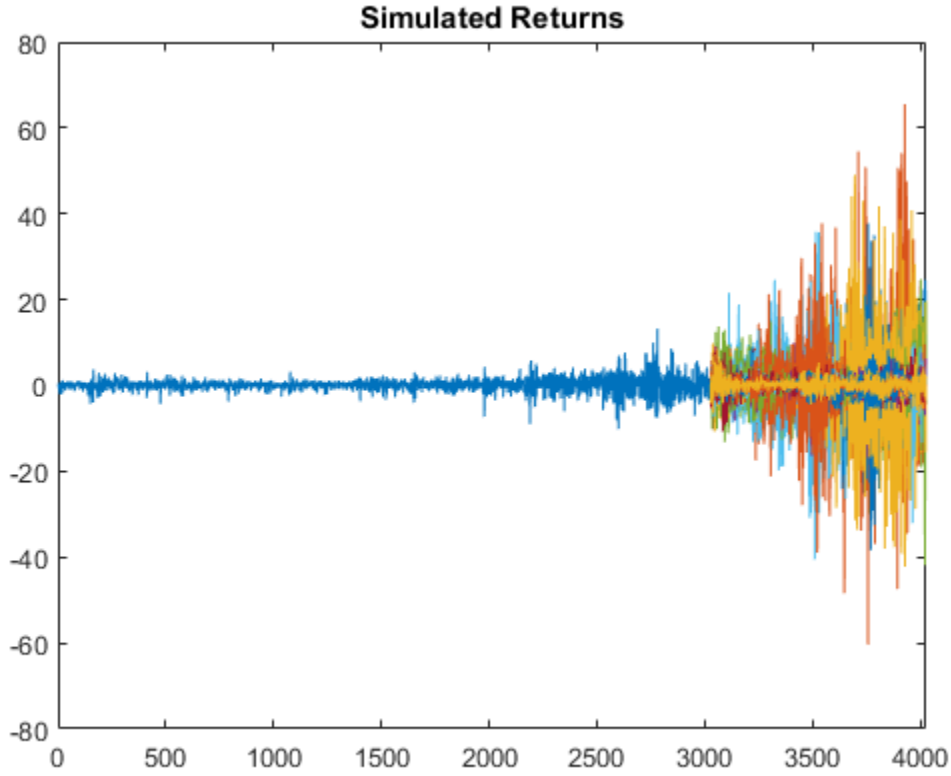
```
load Data_EquityIdx
nasdaq = DataTable.NASDAQ;
r = 100*price2ret(nasdaq);
T = length(r);

Mdl = arima('ARLags',1,'Variance',garch(1,1),...
    'Distribution','t');
EstMdl = estimate(Mdl,r,'VarianceO',{'ConstantO',0.001});
[eO,vO] = infer(EstMdl,r);
```

```
    ARIMA(1,0,0) Model:
    --------------------
    Conditional Probability Distribution: t

                               Standard          t
     Parameter       Value       Error       Statistic
    -----------   -----------   ------------   -----------
     Constant       0.093488     0.0166938        5.60018
        AR{1}       0.139107     0.0188565        7.37713
          DoF        7.47747      0.882611        8.47199


    GARCH(1,1) Conditional Variance Model:
    ----------------------------------------
    Conditional Probability Distribution: t

                               Standard          t
     Parameter       Value       Error       Statistic
    -----------   -----------   ------------   -----------
     Constant      0.0112456    0.00363047        3.09756
     GARCH{1}       0.907662     0.0105156        86.3156
      ARCH{1}      0.0898971     0.0108354        8.29661
          DoF        7.47747      0.882611        8.47199
```

Simulate Returns, Innovations, and Conditional Variances

Use `simulate` to generate 100 sample paths for the returns, innovations, and conditional variances for a 1000-period future horizon. Use the observed returns and inferred residuals and conditional variances as presample data.

```
rng 'default';
[y,e,v] = simulate(EstMdl,1000,'NumPaths',100,...
    'Y0',r,'E0',e0,'V0',v0);

figure
plot(r)
hold on
plot(T+1:T+1000,y)
xlim([0,T+1000])
title('Simulated Returns')
hold off
```

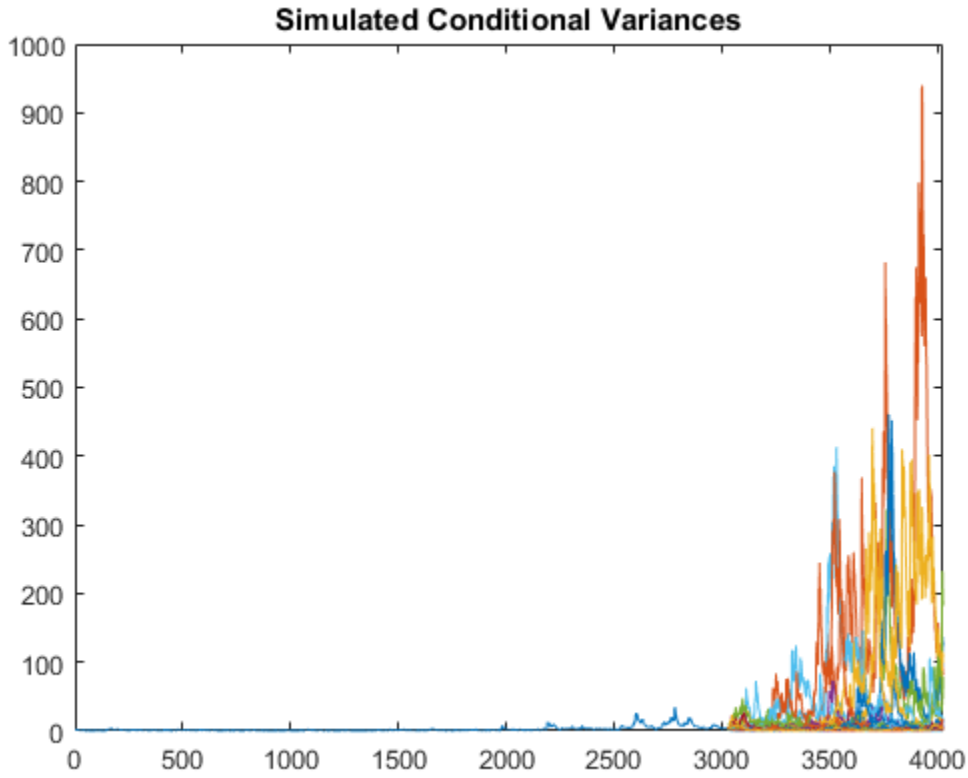


The simulation shows increased volatility over the forecast horizon.

Plot Conditional Variances

Plot the inferred and simulated conditional variances.

```
figure
plot(v0)
hold on
plot(T+1:T+1000,v)
xlim([0,T+1000])
title('Simulated Conditional Variances')
hold off
```

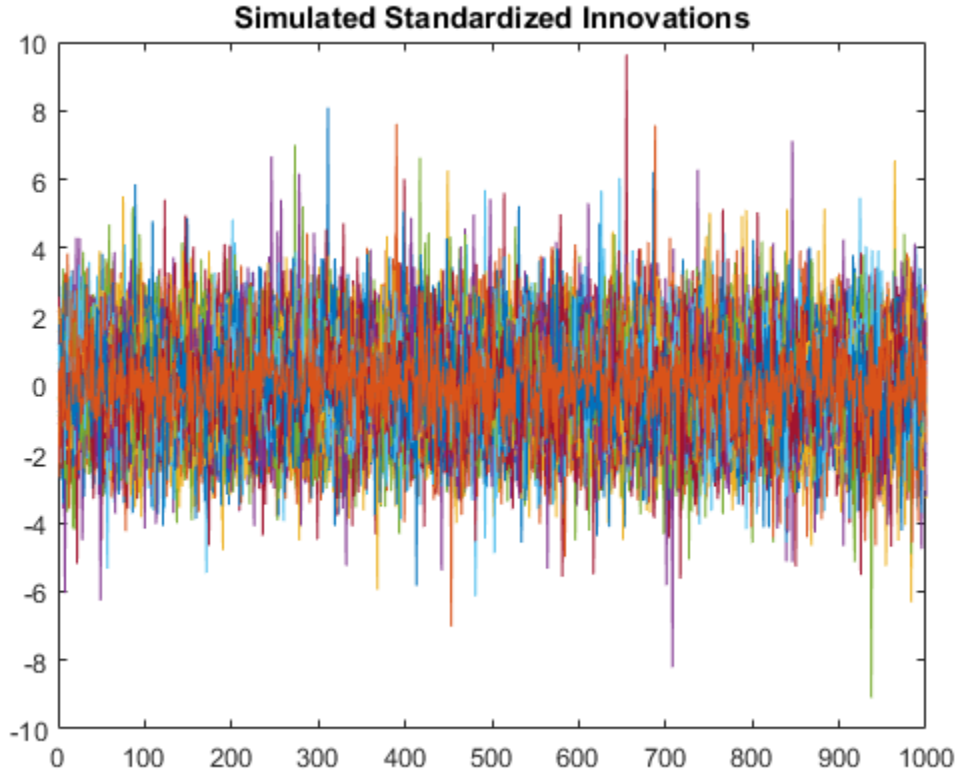


The increased volatility in the simulated returns is due to larger conditional variances over the forecast horizon.

Plot Standardized Innovations

Standardize the innovations using the square root of the conditional variance process. Plot the standardized innovations over the forecast horizon.

```
figure
plot(e./sqrt(v))
xlim([0,1000])
title('Simulated Standardized Innovations')
```



The fitted model assumes the standardized innovations follow a standardized Student's t distribution. Thus, the simulated innovations have more larger values than would be expected from a Gaussian innovation distribution.

See Also

arima | estimate | infer | simulate

Related Examples

- "Specify Conditional Mean and Variance Models" on page 5-79
- "Estimate Conditional Mean and Variance Models" on page 5-129
- "Forecast Conditional Mean and Variance Model" on page 5-197

More About

- "Monte Carlo Simulation of Conditional Mean Models" on page 5-146
- "Presample Data for Conditional Mean Model Simulation" on page 5-149
- "Monte Carlo Forecasting of Conditional Mean Models" on page 5-181

Monte Carlo Forecasting of Conditional Mean Models

Monte Carlo Forecasts

You can use Monte Carlo simulation to forecast a process over a future time horizon. This is an alternative to minimum mean square error (MMSE) forecasting, which provides an analytical forecast solution. You can calculate MMSE forecasts using `forecast`.

To forecast a process using Monte Carlo simulations:

- Fit a model to your observed series using `estimate`.
- Use the observed series and any inferred residuals and conditional variances (calculated using `infer`) for presample data.
- Generate many sample paths over the desired forecast horizon using `simulate`.

Advantage of Monte Carlo Forecasting

An advantage of Monte Carlo forecasting is that you obtain a complete *distribution* for future events, not just a point estimate and standard error. The simulation mean approximates the MMSE forecast. Use the 2.5th and 97.5th percentiles of the simulation realizations as endpoints for approximate 95% forecast intervals.

See Also

arima | estimate | forecast | simulate

Related Examples

- “Simulate Multiplicative ARIMA Models” on page 5-169
- “Simulate Conditional Mean and Variance Models” on page 5-175

More About

- “Monte Carlo Simulation of Conditional Mean Models” on page 5-146
- “Presample Data for Conditional Mean Model Simulation” on page 5-149
- “MMSE Forecasting of Conditional Mean Models” on page 5-182

MMSE Forecasting of Conditional Mean Models

In this section...

What are MMSE Forecasts?

A common objective of time series modeling is generating forecasts for a process over a future time horizon. That is, given an observed series $y_1, y_2,...,y_N$ and a forecast horizon h, generate predictions for $y_{N+1}, y_{N+2},\ldots,y_{N+h}$.

Let $\hat{y}_{t+1}$ denote a forecast for the process at time $t + 1$, conditional on the history of the process up to time t, H_t, and the exogenous covariate series up to time $t + 1$, X_{t+1}, if a regression component is included in the model. The minimum mean square error (MMSE) forecast is the forecast $\hat{y}_{t+1}$ that minimizes expected square loss,

$$E\left(y_{t+1} - \hat{y}_{t+1} \mid H_t, X_{t+1}\right)^2.$$

Minimizing this loss function yields the MMSE forecast,

$$\hat{y}_{t+1} = E(y_{t+1} \mid H_t, X_{t+1}).$$

How forecast Generates MMSE Forecasts

The `forecast` method generates MMSE forecasts recursively. When you call `forecast`, you can specify presample observations (`Y0`), innovations (`E0`), conditional variances (`V0`), and exogenous covariate data (`X0`) using name-value arguments. If you include presample exogenous covariate data, then you must also specify exogenous covariate forecasts (`XF`).

To begin forecasting from the end of an observed series, say `Y`, use the last few observations of `Y` as presample responses `Y0` to initialize the forecast. There are several points to keep in mind when you specify presample data:

- The minimum number of responses needed to initialize forecasting is stored in the property `P` of an `arima` model. If you provide too few presample observations, `forecast` returns an error.
- If you do not provide any presample responses, then `forecast` assigns default values:
  - For models that are stationary *and* do not contain a regression component, all presample observations are set to the unconditional mean of the process.
  - For nonstationary models *or* models with a regression component, all presample observations are set to zero.
- If you forecast a model with an MA component, then `forecast` requires presample innovations. The number of innovations needed is stored in the property `Q` of an `arima` model. If you also have a conditional variance model, you must additionally account for any presample innovations it requires. If you specify presample innovations, but not enough, `forecast` returns an error.
- If you forecast a model with a regression component, then `forecast` requires presample exogenous covariate data. The number of presample exogenous covariate data needed is at least the number of presample responses minus `P`. If you provide presample exogenous covariate data, but not enough, then `forecast` returns an error.
- If you do not specify any presample innovations, but specify sufficient presample *responses* (at least `P + Q`) and exogenous covariate data (at least the number of presample responses minus `P`), then `forecast` automatically infers presample innovations. In general, the longer the presample response series you provide, the better the inferred presample innovations will be. If you provide presample responses and exogenous covariate data, but not enough, `forecast` sets presample innovations equal to zero.

Consider generating forecasts for an AR(2) process,

$$y_t = c + \phi_1 y_{t-1} + \phi_2 y_{t-2} + \varepsilon_t.$$

Given presample observations y_{N-1} and y_N, forecasts are recursively generated as follows:

- $\hat{y}_{N+1} = c + \phi_1 y_N + \phi_2 y_{N-1}$
- $\hat{y}_{N+2} = c + \phi_1 \hat{y}_{N+1} + \phi_2 y_N$

- $\hat{y}_{N+3} = c + \phi_1 \hat{y}_{N+2} + \phi_2 \hat{y}_{N+1}$

⋮

For a stationary AR process, this recursion converges to the unconditional mean of the process,

$$\mu = \frac{c}{(1 - \phi_1 - \phi_2)}.$$

For an MA(12) process, e.g.,

$$y_t = \mu + \varepsilon_t + \theta_1 \varepsilon_{t-1} + \ldots + \theta_{12} \varepsilon_{t-12},$$

you need 12 presample innovations to initialize the forecasts. All innovations from time $N + 1$ and greater are set to their expectation, zero. Thus, for an MA(12) process, the forecast for any time more than 12 steps in the future is the unconditional mean, μ.

Forecast Error

The forecast mean square error for an s-step ahead forecast is given by

$$MSE = E\left(y_{t+s} - \hat{y}_{t+s} \mid H_{t+s-1}, X_{t+s}\right)^2.$$

Consider a conditional mean model given by

$$y_t = \mu + x_t' \beta + \psi(L)\varepsilon_t,$$

where $\psi(L) = 1 + \psi_1 L + \psi_2 L^2 + \ldots$. Sum the variances of the lagged innovations to get the s-step MSE,

$$(1 + \psi_1^2 + \psi_2^2 + \ldots + \psi_{s-1}^2)\sigma_\varepsilon^2,$$

where σ_ε^2 denotes the innovation variance.

For stationary processes, the coefficients of the infinite lag operator polynomial are absolutely summable, and the *MSE* converges to the unconditional variance of the process.

For nonstationary processes, the series does not converge, and the forecast error grows over time.

See Also

arima | forecast

Related Examples

- "Forecast Multiplicative ARIMA Model" on page 5-192
- "Convergence of AR Forecasts" on page 5-186

More About

- "Monte Carlo Forecasting of Conditional Mean Models" on page 5-181

Convergence of AR Forecasts

This example shows how to forecast a stationary AR(12) process using `forecast`. Evaluate the asymptotic convergence of the forecasts, and compare forecasts made with and without using presample data.

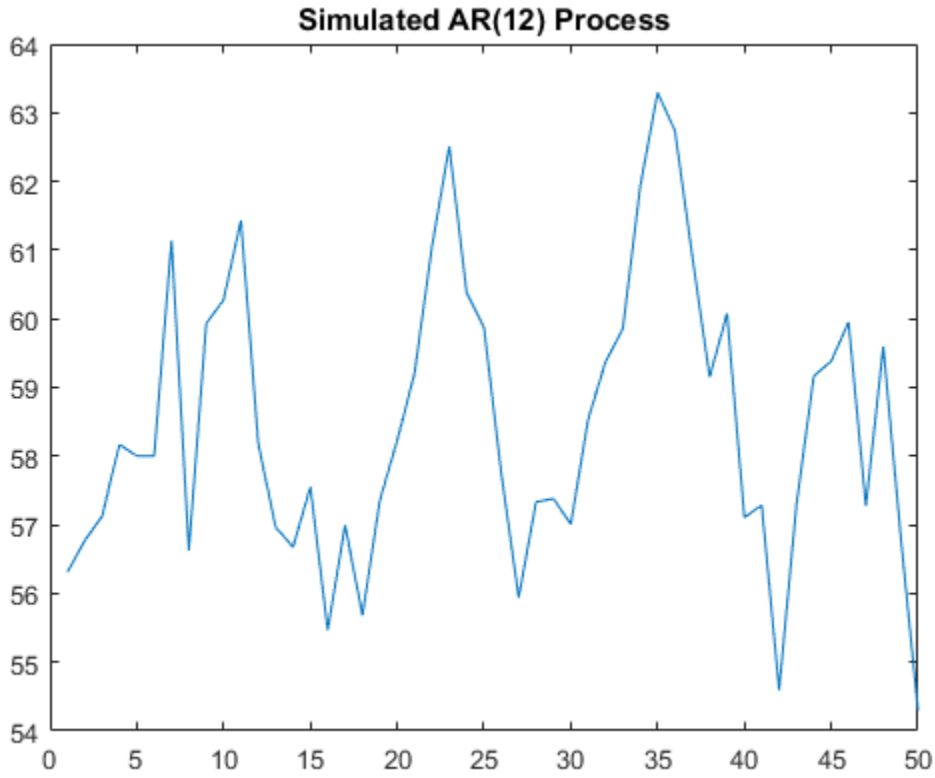
Step 1. Specify an AR(12) model.

Specify the model

$$y_t = 3 + 0.7y_{t-1} + 0.25y_{t-12} + \varepsilon_t,$$

where the innovations are Gaussian with variance 2. Generate a realization of length 300 from the process. Discard the first 250 observations as burn-in.

```
model = arima('Constant',3,'AR',{0.7,0.25},'ARLags',[1,12],...
              'Variance',2);

rng('default')
Y = simulate(model,300);
Y = Y(251:300);

figure
plot(Y)
xlim([0,50])
title('Simulated AR(12) Process')
```

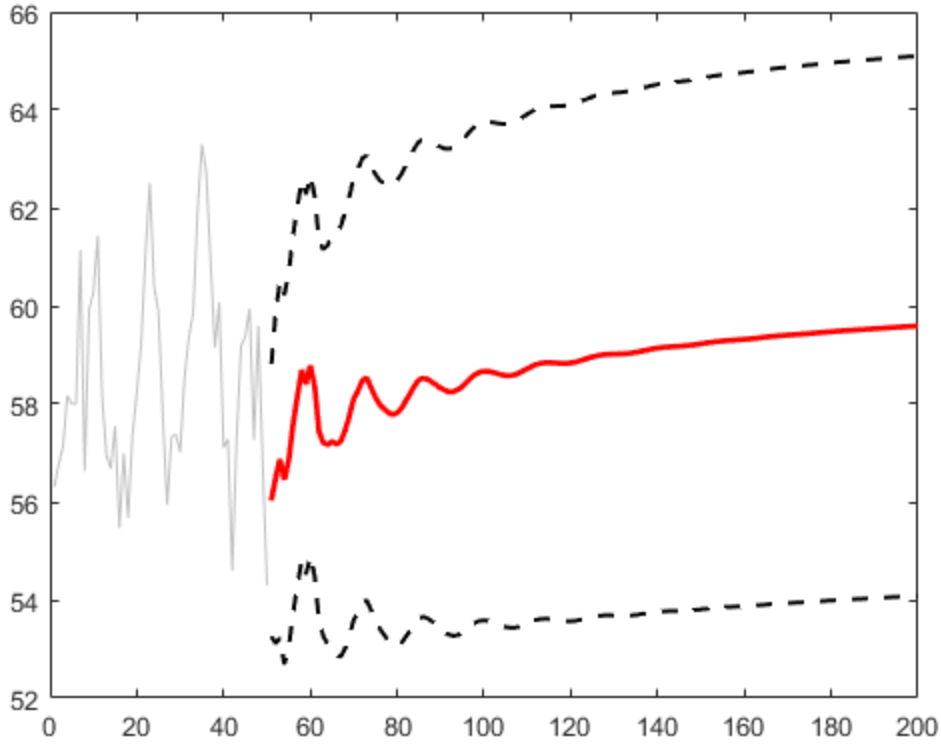


Step 2. Forecast the process using presample data.

Generate forecasts (and forecast errors) for a 150-step time horizon. Use the simulated series as presample data.

```
[Yf,YMSE] = forecast(model,150,'Y0',Y);
upper = Yf + 1.96*sqrt(YMSE);
lower = Yf - 1.96*sqrt(YMSE);

figure
plot(Y,'Color',[.75,.75,.75])
hold on
plot(51:200,Yf,'r','LineWidth',2)
plot(51:200,[upper,lower],'k--','LineWidth',1.5)
```

```
xlim([0,200])
hold off
```



The MMSE forecast sinusoidally decays, and begins converging to the unconditional mean, given by

$$\mu = \frac{c}{(1-\phi_1-\phi_{12})} = \frac{3}{(1-0.7-0.25)} = 60.$$

Step 3. Calculate the asymptotic variance.

The MSE of the process converges to the unconditional variance of the process ($\sigma_\varepsilon^2 = 2$). You can calculate the variance using the impulse response function. The impulse response function is based on the infinite-degree MA representation of the AR(2) process.

The last few values of YMSE show the convergence toward the unconditional variance.

```
ARpol = LagOp({1,-.7,-.25},'Lags',[0,1,12]);
IRF = cell2mat(toCellArray(1/ARpol));
sig2e = 2;

variance = sum(IRF.^2)*sig2e % Display the variance
YMSE(145:end) % Display the forecast MSEs
```

```
variance =

    7.9938


ans =

    7.8870
    7.8899
    7.8926
    7.8954
    7.8980
    7.9006
```

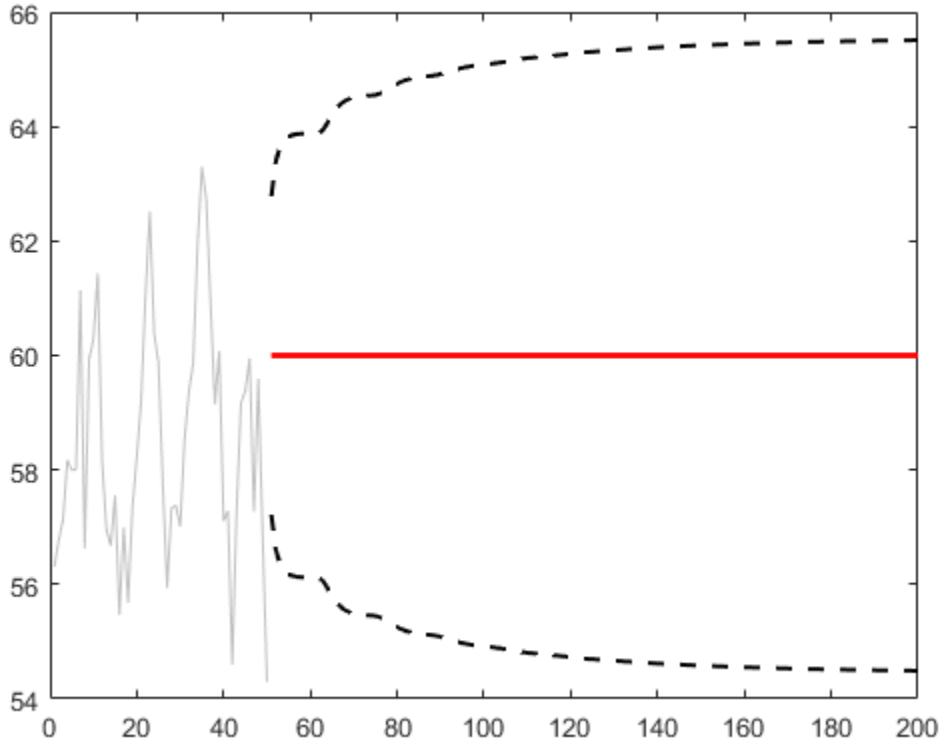
Convergence is not reached within 150 steps, but the forecast MSE is approaching the theoretical unconditional variance.

Step 4. Forecast without using presample data.

Repeat the forecasting without using any presample data.

```
[Yf2,YMSE2] = forecast(model,150);
upper2 = Yf2 + 1.96*sqrt(YMSE2);
lower2 = Yf2 - 1.96*sqrt(YMSE2);

YMSE2(145:end) % Display the forecast MSEs

figure
plot(Y,'Color',[.75,.75,.75])
hold on
plot(51:200,Yf2,'r','LineWidth',2)
plot(51:200,[upper2,lower2],'k--','LineWidth',1.5)
xlim([0,200])
hold off
```

```
ans =

    7.8870
    7.8899
    7.8926
    7.8954
    7.8980
    7.9006
```



The convergence of the forecast MSE is the same without using presample data. However, all MMSE forecasts are the unconditional mean. This is because `forecast`

initializes the AR model with the unconditional mean when you do not provide presample data.

See Also

arima | forecast | LagOp | simulate | toCellArray

Related Examples

- "Simulate Stationary Processes" on page 5-151
- "Forecast Multiplicative ARIMA Model" on page 5-192

More About

- "MMSE Forecasting of Conditional Mean Models" on page 5-182
- "Autoregressive Model" on page 5-18

Forecast Multiplicative ARIMA Model

This example shows how to forecast a multiplicative seasonal ARIMA model using `forecast`. The time series is monthly international airline passenger numbers from 1949 to 1960.

Load the Data and Estimate a Model.

Load the data set `Data_Airline`.

```
load(fullfile(matlabroot,'examples','econ','Data_Airline.mat'))
y = log(Data);
T = length(y);

Mdl = arima('Constant',0,'D',1,'Seasonality',12,...
    'MALags',1,'SMALags',12);
EstMdl = estimate(Mdl,y);
```

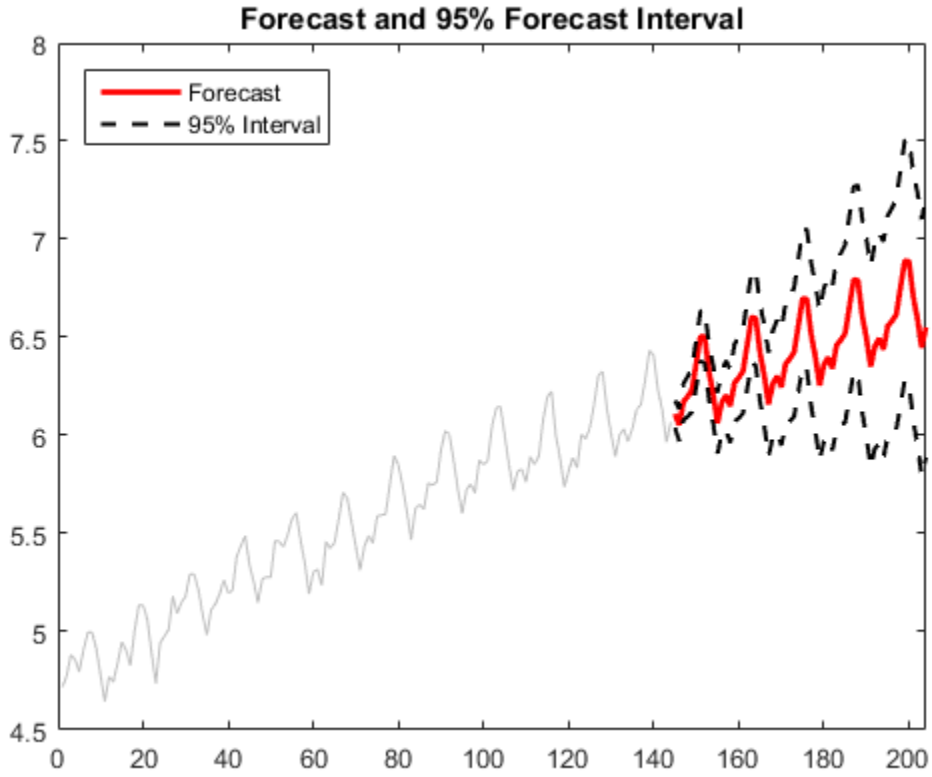
```
    ARIMA(0,1,1) Model Seasonally Integrated with Seasonal MA(12):
    --------------------------------------------------------------
    Conditional Probability Distribution: Gaussian

                               Standard          t
     Parameter       Value       Error       Statistic
    -----------   -----------   ------------   -----------
     Constant             0         Fixed          Fixed
        MA{1}      -0.377162     0.0667944       -5.64661
      SMA{12}      -0.572378     0.0854395       -6.69923
     Variance     0.00126337    0.00012395        10.1926
```

Forecast Airline Passenger Counts.

Use the fitted model to generate MMSE forecasts and corresponding mean square errors over a 60-month (5-year) horizon. Use the observed series as presample data. By default, `forecast` infers presample innovations using the specified model and observations.

```
[yF,yMSE] = forecast(EstMdl,60,'Y0',y);
upper = yF + 1.96*sqrt(yMSE);
lower = yF - 1.96*sqrt(yMSE);

figure
plot(y,'Color',[.75,.75,.75])
```

```
hold on
h1 = plot(T+1:T+60,yF,'r','LineWidth',2);
h2 = plot(T+1:T+60,upper,'k--','LineWidth',1.5);
plot(T+1:T+60,lower,'k--','LineWidth',1.5)
xlim([0,T+60])
title('Forecast and 95% Forecast Interval')
legend([h1,h2],'Forecast','95% Interval','Location','NorthWest')
hold off
```

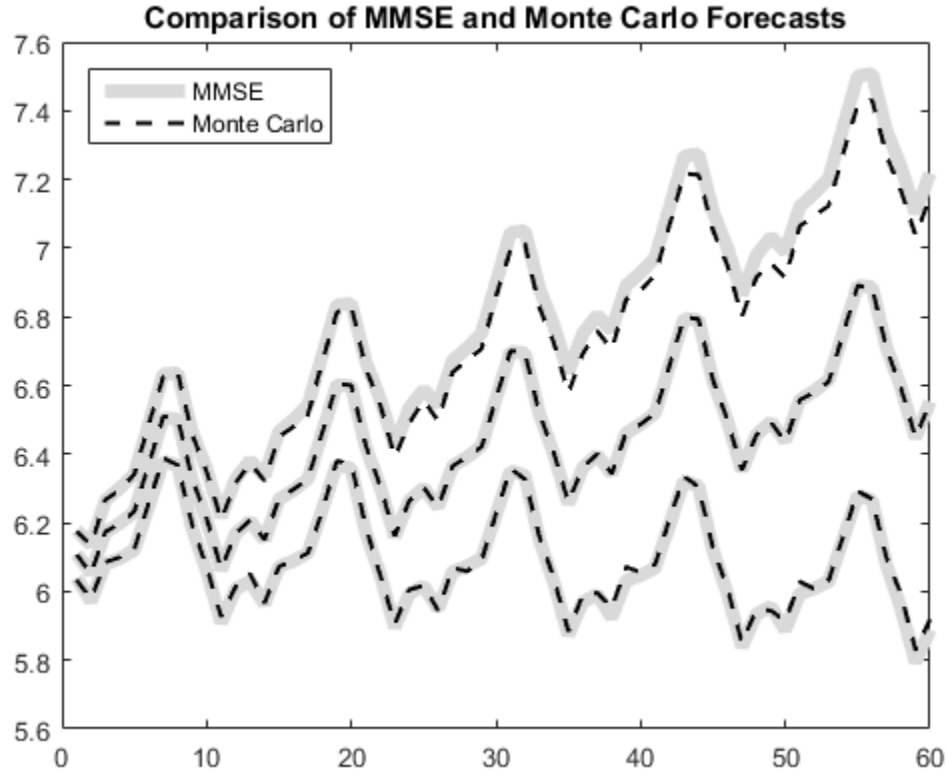


The MMSE forecast shows airline passenger counts continuing to grow over the forecast horizon. The confidence bounds show that a decline in passenger counts is plausible, however. Because this is a nonstationary process, the width of the forecast intervals grows over time.

Compare MMSE and Monte Carlo Forecasts.

Simulate 500 sample paths over the same forecast horizon. Compare the simulation mean to the MMSE forecast.

```
rng 'default';
res = infer(EstMdl,y);
Ysim = simulate(EstMdl,60,'NumPaths',500,'Y0',y,'E0',res);

yBar = mean(Ysim,2);
simU = prctile(Ysim,97.5,2);
simL = prctile(Ysim,2.5,2);

figure
h1 = plot(yF,'Color',[.85,.85,.85],'LineWidth',5);
hold on
h2 = plot(yBar,'k--','LineWidth',1.5);
xlim([0,60])
plot([upper,lower],'Color',[.85,.85,.85],'LineWidth',5)
plot([simU,simL],'k--','LineWidth',1.5)
title('Comparison of MMSE and Monte Carlo Forecasts')
legend([h1,h2],'MMSE','Monte Carlo','Location','NorthWest')
hold off
```



The MMSE forecast and simulation mean are virtually indistinguishable. There are slight discrepancies between the theoretical 95% forecast intervals and the simulation-based 95% forecast intervals.

See Also

arima | estimate | forecast | infer | simulate

Related Examples

- "Specify Multiplicative ARIMA Model" on page 5-52
- "Estimate Multiplicative ARIMA Model" on page 5-113
- "Simulate Multiplicative ARIMA Models" on page 5-169

- “Model Seasonal Lag Effects Using Indicator Variables” on page 5-117
- “Check Fit of Multiplicative ARIMA Model” on page 3-81

More About

- “MMSE Forecasting of Conditional Mean Models” on page 5-182
- “Monte Carlo Forecasting of Conditional Mean Models” on page 5-181

Forecast Conditional Mean and Variance Model

This example shows how to forecast responses and conditional variances from a composite conditional mean and variance model.

Step 1. Load the data and fit a model.

Load the NASDAQ data included with the toolbox. Fit a conditional mean and variance model to the data.

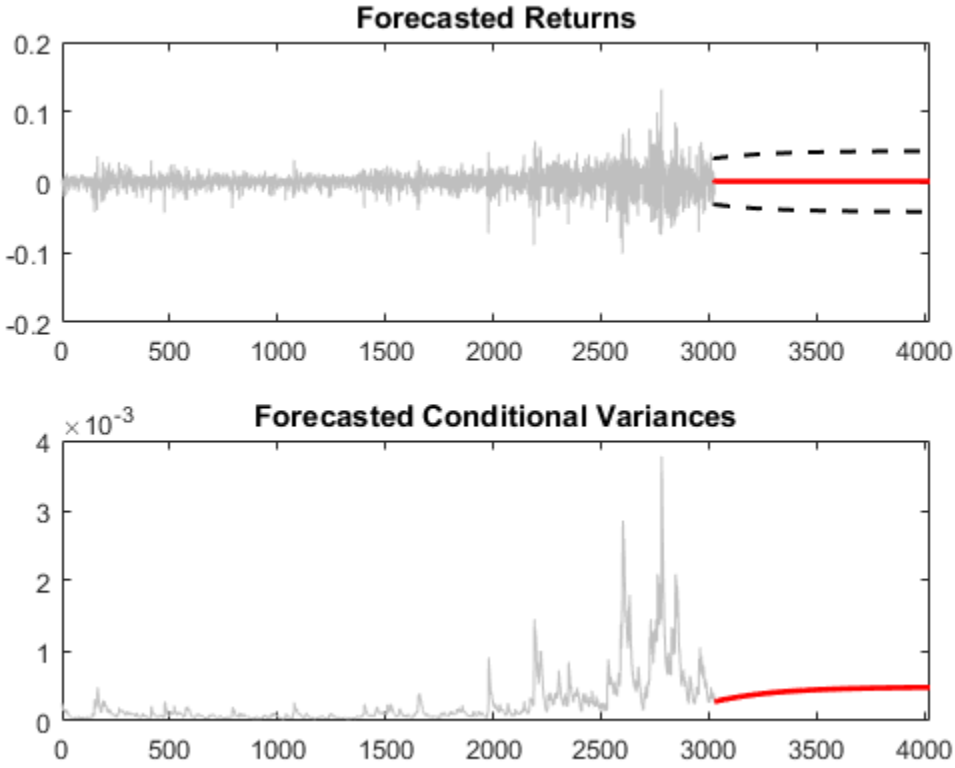
```
load Data_EquityIdx
nasdaq = DataTable.NASDAQ;
r = price2ret(nasdaq);
N = length(r);

model = arima('ARLags',1,'Variance',garch(1,1),...
              'Distribution','t');
fit = estimate(model,r,'Variance0',{'Constant0',0.001});
[E0,V0] = infer(fit,r);
```

```
    ARIMA(1,0,0) Model:
    --------------------
    Conditional Probability Distribution: t

                               Standard          t
     Parameter       Value       Error       Statistic
    -----------   -----------   ------------   -----------
     Constant     0.00103605   0.000170541        6.07506
        AR{1}       0.144925     0.0193368        7.49478
          DoF        7.43163      0.911017        8.15751


    GARCH(1,1) Conditional Variance Model:
    ----------------------------------------
    Conditional Probability Distribution: t

                               Standard          t
     Parameter       Value       Error       Statistic
    -----------   -----------   ------------   -----------
     Constant    1.68497e-06   6.57095e-07        2.56427
     GARCH{1}       0.890486      0.012054        73.8747
      ARCH{1}       0.106033      0.012524        8.46645
          DoF        7.43163      0.911017        8.15751
```

Step 2. Forecast returns and conditional variances.

Use `forecast` to compute MMSE forecasts of the returns and conditional variances for a 1000-period future horizon. Use the observed returns and inferred residuals and conditional variances as presample data.

```
[Y,YMSE,V] = forecast(fit,1000,'Y0',r,'E0',E0,'V0',V0);
upper = Y + 1.96*sqrt(YMSE);
lower = Y - 1.96*sqrt(YMSE);

figure
subplot(2,1,1)
plot(r,'Color',[.75,.75,.75])
hold on
plot(N+1:N+1000,Y,'r','LineWidth',2)
plot(N+1:N+1000,[upper,lower],'k--','LineWidth',1.5)
xlim([0,N+1000])
title('Forecasted Returns')
hold off
subplot(2,1,2)
plot(V0,'Color',[.75,.75,.75])
hold on
plot(N+1:N+1000,V,'r','LineWidth',2);
xlim([0,N+1000])
title('Forecasted Conditional Variances')
hold off
```



The conditional variance forecasts converge to the asymptotic variance of the GARCH conditional variance model. The forecasted returns converge to the estimated model constant (the unconditional mean of the AR conditional mean model).

See Also

arima | estimate | forecast | `garch` | infer

Related Examples

- "Specify Conditional Mean and Variance Models" on page 5-79
- "Estimate Conditional Mean and Variance Models" on page 5-129
- "Model Seasonal Lag Effects Using Indicator Variables" on page 5-117

- "Simulate Conditional Mean and Variance Models" on page 5-175

More About

- "MMSE Forecasting of Conditional Mean Models" on page 5-182
- Using garch Objects

Conditional Variance Models

Conditional Variance Models

In this section...

General Conditional Variance Model Definition

Consider the time series

$$y_t = \mu + \varepsilon_t,$$

where $\varepsilon_t = \sigma_t z_t$. Here, z_t is an independent and identically distributed series of standardized random variables. Econometrics Toolbox supports standardized Gaussian and standardized Student's t innovation distributions. The constant term, μ, is a mean offset.

A *conditional variance model* specifies the dynamic evolution of the innovation variance,

$$\sigma_t^2 = Var(\varepsilon_t \mid H_{t-1}),$$

where H_{t-1} is the history of the process. The history includes:

- Past variances, $\sigma_1^2, \sigma_2^2, \ldots, \sigma_{t-1}^2$
- Past innovations, $\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_{t-1}$

Conditional variance models are appropriate for time series that do not exhibit significant autocorrelation, but are serially dependent. The innovation series $\varepsilon_t = \sigma_t z_t$ is uncorrelated, because:

- $E(\varepsilon_t) = 0$.
- $E(\varepsilon_t\varepsilon_{t-h}) = 0$ for all t and $h \neq 0$.

However, if σ_t^2 depends on σ_{t-1}^2, for example, then ε_t depends on ε_{t-1}, even though they are uncorrelated. This kind of dependence exhibits itself as autocorrelation in the squared innovation series, ε_t^2.

Tip For modeling time series that are both autocorrelated and serially dependent, you can consider using a composite conditional mean and variance model.

Two characteristics of financial time series that conditional variance models address are:

- *Volatility clustering*. Volatility is the conditional standard deviation of a time series. Autocorrelation in the conditional variance process results in volatility clustering. The GARCH model and its variants model autoregression in the variance series.
- *Leverage effects*. The volatility of some time series responds more to large decreases than to large increases. This asymmetric clustering behavior is known as the leverage effect. The EGARCH and GJR models have leverage terms to model this asymmetry.

GARCH Model

The *generalized autoregressive conditional heteroscedastic* (GARCH) model is an extension of Engle's ARCH model for variance heteroscedasticity [1]. If a series exhibits volatility clustering, this suggests that past variances might be predictive of the current variance.

The GARCH(P,Q) model is an autoregressive moving average model for conditional variances, with P GARCH coefficients associated with lagged variances, and Q ARCH coefficients associated with lagged squared innovations. The form of the GARCH(P,Q) model in Econometrics Toolbox is

$$y_t = \mu + \varepsilon_t,$$

where $\varepsilon_t = \sigma_t z_t$ and

$$\sigma_t^2 = \kappa + \gamma_1\sigma_{t-1}^2 + \ldots + \gamma_P\sigma_{t-P}^2 + \alpha_1\varepsilon_{t-1}^2 + \ldots + \alpha_Q\varepsilon_{t-Q}^2.$$

Note: The `Constant` property of a `garch` model corresponds to κ, and the `Offset` property corresponds to μ.

For stationarity and positivity, the GARCH model has the following constraints:

- $\kappa > 0$
- $\gamma_i \geq 0,\ \alpha_j \geq 0$
- $\sum_{i=1}^{P}\gamma_i + \sum_{j=1}^{Q}\alpha_j < 1$

To specify Engle's original ARCH(Q) model, use the equivalent GARCH(0,Q) specification.

EGARCH Model

The exponential GARCH (EGARCH) model is a GARCH variant that models the logarithm of the conditional variance process. In addition to modeling the logarithm, the EGARCH model has additional leverage terms to capture asymmetry in volatility clustering.

The EGARCH(P,Q) model has P GARCH coefficients associated with lagged log variance terms, Q ARCH coefficients associated with the magnitude of lagged standardized innovations, and Q leverage coefficients associated with signed, lagged standardized innovations. The form of the EGARCH(P,Q) model in Econometrics Toolbox is

$$y_t = \mu + \varepsilon_t,$$

where $\varepsilon_t = \sigma_t z_t$ and

$$\log\sigma_t^2 = \kappa + \sum_{i=1}^{P}\gamma_i \log\sigma_{t-i}^2 + \sum_{j=1}^{Q}\alpha_j\left[\frac{|\varepsilon_{t-j}|}{\sigma_{t-j}} - E\left\{\frac{|\varepsilon_{t-j}|}{\sigma_{t-j}}\right\}\right] + \sum_{j=1}^{Q}\xi_j\left(\frac{\varepsilon_{t-j}}{\sigma_{t-j}}\right).$$

Note: The `Constant` property of an `egarch` model corresponds to κ, and the `Offset` property corresponds to μ.

The form of the expected value terms associated with ARCH coefficients in the EGARCH equation depends on the distribution of z_t:

- If the innovation distribution is Gaussian, then

$$E\left\{\frac{|\varepsilon_{t-j}|}{\sigma_{t-j}}\right\} = E\{|z_{t-j}|\} = \sqrt{\frac{2}{\pi}}.$$

- If the innovation distribution is Student's t with $\nu > 2$ degrees of freedom, then

$$E\left\{\frac{|\varepsilon_{t-j}|}{\sigma_{t-j}}\right\} = E\{|z_{t-j}|\} = \sqrt{\frac{\nu-2}{\pi}}\frac{\Gamma\left(\frac{\nu-1}{2}\right)}{\Gamma\left(\frac{\nu}{2}\right)}.$$

The toolbox treats the EGARCH(P,Q) model as an ARMA model for $\log\sigma_t^2$. Thus, to ensure stationarity, all roots of the GARCH coefficient polynomial, $(1-\gamma_1 L - \ldots - \gamma_P L^P)$, must lie outside the unit circle.

The EGARCH model is unique from the GARCH and GJR models because it models the logarithm of the variance. By modeling the logarithm, positivity constraints on the model parameters are relaxed. However, forecasts of conditional variances from an EGARCH model are biased, because by Jensen's inequality,

$$E(\sigma_t^2) \geq \exp\{E(\log\sigma_t^2)\}.$$

An EGARCH(1,1) specification will be complex enough for most applications. For an EGARCH(1,1) model, the GARCH and ARCH coefficients are expected to be positive, and the leverage coefficient is expected to be negative; large unanticipated downward shocks should increase the variance. If you get signs opposite to those expected, you might encounter difficulties inferring volatility sequences and forecasting (a negative ARCH coefficient can be particularly problematic). In this case, an EGARCH model might not be the best choice for your application.

GJR Model

The GJR model is a GARCH variant that includes leverage terms for modeling asymmetric volatility clustering. In the GJR formulation, large negative changes are more likely to be clustered than positive changes. The GJR model is named for Glosten, Jagannathan, and Runkle [2]. Close similarities exist between the GJR model and the threshold GARCH (TGARCH) model—a GJR model is a recursive equation for the variance process, and a TGARCH is the same recursion applied to the standard deviation process.

The GJR(P,Q) model has P GARCH coefficients associated with lagged variances, Q ARCH coefficients associated with lagged squared innovations, and Q leverage coefficients associated with the square of negative lagged innovations. The form of the GJR(P,Q) model in Econometrics Toolbox is

$$y_t = \mu + \varepsilon_t,$$

where $\varepsilon_t = \sigma_t z_t$ and

$$\sigma_t^2 = \kappa + \sum_{i=1}^{P} \gamma_i \sigma_{t-i}^2 + \sum_{j=1}^{Q} \alpha_j \varepsilon_{t-j}^2 + \sum_{j=1}^{Q} \xi_j I\left[\varepsilon_{t-j} < 0\right] \varepsilon_{t-j}^2.$$

The indicator function $I\left[\varepsilon_{t-j} < 0\right]$ equals 1 if $\varepsilon_{t-j} < 0$, and 0 otherwise. Thus, the leverage coefficients are applied to negative innovations, giving negative changes additional weight.

Note: The `Constant` property of a `gjr` model corresponds to κ, and the `Offset` property corresponds to μ.

For stationarity and positivity, the GJR model has the following constraints:

- $\kappa > 0$
- $\gamma_i \geq 0,\ \alpha_j \geq 0$
- $\alpha_j + \xi_j \geq 0$

- $\sum_{i=1}^{P}\gamma_i + \sum_{j=1}^{Q}\alpha_j + \frac{1}{2}\sum_{j=1}^{Q}\xi_j < 1$

The GARCH model is nested in the GJR model. If all leverage coefficients are zero, then the GJR model reduces to the GARCH model. This means you can test a GARCH model against a GJR model using the likelihood ratio test.

References

[1] Engle, Robert F. "Autoregressive Conditional Heteroskedasticity with Estimates of the Variance of United Kingdom Inflation." *Econometrica*. Vol. 50, 1982, pp. 987–1007.

[2] Glosten, L. R., R. Jagannathan, and D. E. Runkle. "On the Relation between the Expected Value and the Volatility of the Nominal Excess Return on Stocks." *The Journal of Finance*. Vol. 48, No. 5, 1993, pp. 1779–1801.

Related Examples

- "Specify GARCH Models Using garch" on page 6-8
- "Specify EGARCH Models Using egarch" on page 6-19
- "Specify GJR Models Using gjr" on page 6-31
- "Specify Conditional Mean and Variance Models" on page 5-79
- "Assess EGARCH Forecast Bias Using Simulations" on page 6-104

More About

- Using egarch Objects
- Using gjr Objects
- Using garch Objects
- "Conditional Mean Models" on page 5-3

Specify GARCH Models Using garch

In this section...

Default GARCH Model

The default GARCH(P,Q) model in Econometrics Toolbox is of the form

$$\varepsilon_t = \sigma_t z_t,$$

with Gaussian innovation distribution and

$$\sigma_t^2 = \kappa + \gamma_1 \sigma_{t-1}^2 + \ldots + \gamma_P \sigma_{t-P}^2 + \alpha_1 \varepsilon_{t-1}^2 + \ldots + \alpha_Q \varepsilon_{t-Q}^2.$$

The default model has no mean offset, and the lagged variances and squared innovations are at consecutive lags.

You can specify a model of this form using the shorthand syntax `garch(P,Q)`. For the input arguments `P` and `Q`, enter the number of lagged conditional variances (GARCH terms), P, and lagged squared innovations (ARCH terms), Q, respectively. The following restrictions apply:

- P and Q must be nonnegative integers.
- If P is zero, the GARCH(P,Q) model reduces to an ARCH(Q) model.

- If $P > 0$, then you must also specify $Q > 0$.

When you use this shorthand syntax, `garch` creates a `garch` model with these default property values.

Property	Default Value
`P`	Number of GARCH terms, P
`Q`	Number of ARCH terms, Q
`Offset`	`0`
`Constant`	`NaN`
`GARCH`	Cell vector of `NaN`s
`ARCH`	Cell vector of `NaN`s
`Distribution`	`'Gaussian'`

To assign nondefault values to any properties, you can modify the created model using dot notation.

To illustrate, consider specifying the GARCH(1,1) model

$$\varepsilon_t = \sigma_t z_t,$$

with Gaussian innovation distribution and

$$\sigma_t^2 = \kappa + \gamma_1 \sigma_{t-1}^2 + \alpha_1 \varepsilon_{t-1}^2.$$

```
Mdl = garch(1,1)
```

```
Mdl =

    GARCH(1,1) Conditional Variance Model:
    ----------------------------------------
    Distribution: Name = 'Gaussian'
```

```
              P: 1
              Q: 1
       Constant: NaN
          GARCH: {NaN} at Lags [1]
           ARCH: {NaN} at Lags [1]
```

The created model, `Mdl`, has `NaN`s for all model parameters. A `NaN` value signals that a parameter needs to be estimated or otherwise specified by the user. All parameters must be specified to forecast or simulate the model.

To estimate parameters, input the model (along with data) to `estimate`. This returns a new fitted `garch` model. The fitted model has parameter estimates for each input `NaN` value.

Calling `garch` without any input arguments returns a GARCH(0,0) model specification with default property values:

```
DefaultMdl = garch
```

```
DefaultMdl =

    GARCH(0,0) Conditional Variance Model:
    --------------------------------------
    Distribution: Name = 'Gaussian'
              P: 0
              Q: 0
       Constant: NaN
          GARCH: {}
           ARCH: {}
```

Specify Default GARCH Model

This example shows how to use the shorthand `garch(P,Q)` syntax to specify the default GARCH(P, Q) model, $\varepsilon_t = \sigma_t z_t$ with Gaussian innovation distribution and

$$\sigma_t^2 = \kappa + \gamma_1\sigma_{t-1}^2 + \ldots + \gamma_P\sigma_{t-P}^2 + \alpha_1\varepsilon_{t-1}^2 + \ldots + \alpha_Q\varepsilon_{t-Q}^2.$$

By default, all parameters in the created model have unknown values.

Specify the default GARCH(1,1) model.

```
Mdl = garch(1,1)
```

```
Mdl =

    GARCH(1,1) Conditional Variance Model:
    ----------------------------------------
    Distribution: Name = 'Gaussian'
               P: 1
               Q: 1
        Constant: NaN
           GARCH: {NaN} at Lags [1]
            ARCH: {NaN} at Lags [1]
```

The output shows that the created model, `Mdl`, has `NaN` values for all model parameters: the constant term, the GARCH coefficient, and the ARCH coefficient. You can modify the created model using dot notation, or input it (along with data) to `estimate`.

Using Name-Value Pair Arguments

The most flexible way to specify GARCH models is using name-value pair arguments. You do not need, nor are you able, to specify a value for every model property. `garch` assigns default values to any properties you do not (or cannot) specify.

The general GARCH(P,Q) model is of the form

$$y_t = \mu + \varepsilon_t,$$

where $\varepsilon_t = \sigma_t z_t$ and

$$\sigma_t^2 = \kappa + \gamma_1\sigma_{t-1}^2 + \ldots + \gamma_P\sigma_{t-P}^2 + \alpha_1\varepsilon_{t-1}^2 + \ldots + \alpha_Q\varepsilon_{t-Q}^2.$$

The innovation distribution can be Gaussian or Student's t. The default distribution is Gaussian.

In order to estimate, forecast, or simulate a model, you must specify the parametric form of the model (e.g., which lags correspond to nonzero coefficients, the innovation

distribution) and any known parameter values. You can set any unknown parameters equal to `NaN`, and then input the model to `estimate` (along with data) to get estimated parameter values.

`garch` (and `estimate`) returns a model corresponding to the model specification. You can modify models to change or update the specification. Input models (with no `NaN` values) to `forecast` or `simulate` for forecasting and simulation, respectively. Here are some example specifications using name-value arguments.

Model	Specification
• $y_t = \varepsilon_t$ • $\varepsilon_t = \sigma_t z_t$ • z_t Gaussian • $\sigma_t^2 = \kappa + \gamma_1\sigma_{t-1}^2 + \alpha_1\varepsilon_{t-1}^2$	`garch('GARCH',NaN,'ARCH',NaN)` or `garch(1,1)`
• $y_t = \mu + \varepsilon_t$ • $\varepsilon_t = \sigma_t z_t$ • z_t Student's t with unknown degrees of freedom • $\sigma_t^2 = \kappa + \gamma_1\sigma_{t-1}^2 + \alpha_1\varepsilon_{t-1}^2$	`garch('Offset',NaN,'GARCH',NaN,'ARCH',NaN,...` `'Distribution','t')`
• $y_t = \varepsilon_t$ • $\varepsilon_t = \sigma_t z_t$ • z_t Student's t with eight degrees of freedom • $\sigma_t^2 = 0.1 + 0.6\sigma_{t-1}^2 + 0.3\varepsilon_{t-1}^2$	`garch('Constant',0.1,'GARCH',0.6,'ARCH',0.3,...` `'Distribution',struct('Name','t','DoF',8))`

Here is a full description of the name-value arguments you can use to specify GARCH models.

Note: You cannot assign values to the properties `P` and `Q`. `garch` sets these properties equal to the largest GARCH and ARCH lags, respectively.

Name-Value Arguments for GARCH Models

Name	Corresponding GARCH Model Term(s)	When to Specify
`Offset`	Mean offset, μ	To include a nonzero mean offset. For example, `'Offset',0.3`. If you plan to estimate the offset term, specify `'Offset',NaN`. By default, `Offset` has value `0` (meaning, no offset).
`Constant`	Constant in the conditional variance model, κ	To set equality constraints for κ. For example, if a model has known constant 0.1, specify `'Constant',0.1`. By default, `Constant` has value `NaN`.
`GARCH`	GARCH coefficients, $\gamma_1,\ldots,\gamma_P$	To set equality constraints for the GARCH coefficients. For example, to specify the GARCH coefficient in the model $\varepsilon_t = 0.7\sigma_{t-1}^2 + 0.25\varepsilon_{t-1}^2,$ specify `'GARCH',0.7`. You only need to specify the nonzero elements of `GARCH`. If the nonzero coefficients are at nonconsecutive lags, specify the corresponding lags using `GARCHLags`. Any coefficients you specify must satisfy all stationarity and positivity constraints.
`GARCHLags`	Lags corresponding to nonzero GARCH coefficients	`GARCHLags` is not a model property. Use this argument as a shortcut for specifying `GARCH` when the nonzero GARCH coefficients correspond to nonconsecutive lags. For example, to specify nonzero GARCH coefficients at lags 1 and 3, e.g., $\sigma_t^2 = \gamma_1\sigma_{t-1}^2 + \gamma_3\sigma_{t-3}^2 + \alpha_1\varepsilon_{t-1}^2,$ specify `'GARCHLags',[1,3]`. Use `GARCH` and `GARCHLags` together to specify known nonzero GARCH coefficients at nonconsecutive lags. For example, if in the given GARCH(3,1) model $\gamma_1 = 0.3$ and

Name	Corresponding GARCH Model Term(s)	When to Specify
		$\gamma_3 = 0.1$, specify `'GARCH',{0.3,0.1},'GARCHLags',[1,3]`.
`ARCH`	ARCH coefficients, $\alpha_1,\ldots,\alpha_Q$	To set equality constraints for the ARCH coefficients. For example, to specify the ARCH coefficient in the model $$\varepsilon_t = 0.7\sigma_{t-1}^2 + 0.25\varepsilon_{t-1}^2,$$ specify `'ARCH',0.25`. You only need to specify the nonzero elements of `ARCH`. If the nonzero coefficients are at nonconsecutive lags, specify the corresponding lags using `ARCHLags`. Any coefficients you specify must satisfy all stationarity and positivity constraints.
`ARCHLags`	Lags corresponding to nonzero ARCH coefficients	`ARCHLags` is not a model property. Use this argument as a shortcut for specifying `ARCH` when the nonzero ARCH coefficients correspond to nonconsecutive lags. For example, to specify nonzero ARCH coefficients at lags 1 and 3, e.g., $$\sigma_t^2 = \gamma_1\sigma_{t-1}^2 + \alpha_1\varepsilon_{t-1}^2 + \alpha_3\varepsilon_{t-3}^2,$$ specify `'ARCHLags',[1,3]`. Use `ARCH` and `ARCHLags` together to specify known nonzero ARCH coefficients at nonconsecutive lags. For example, if in the above model $\alpha_1 = 0.4$ and $\alpha_3 = 0.2$, specify `'ARCH',{0.4,0.2},'ARCHLags',[1,3]`.
`Distribution`	Distribution of the innovation process	Use this argument to specify a Student's t innovation distribution. By default, the innovation distribution is Gaussian. For example, to specify a t distribution with unknown degrees of freedom, specify `'Distribution','t'`. To specify a t innovation distribution with known degrees of freedom, assign `Distribution` a data

Name	Corresponding GARCH Model Term(s)	When to Specify
		structure with fields `Name` and `DoF`. For example, for a t distribution with nine degrees of freedom, specify `'Distribution',struct('Name','t','DoF',9)`.

Specify GARCH Model with Mean Offset

This example shows how to specify a GARCH(P, Q) model with a mean offset. Use name-value pair arguments to specify a model that differs from the default model.

Specify a GARCH(1,1) model with a mean offset,

$$y_t = \mu + \varepsilon_t,$$

where $\varepsilon_t = \sigma_t z_t$ and

$$\sigma_t^2 = \kappa + \gamma_1 \sigma_{t-1}^2 + \alpha_1 \varepsilon_{t-1}^2.$$

```
Mdl = garch('Offset',NaN,'GARCHLags',1,'ARCHLags',1)
```

```
Mdl =

    GARCH(1,1) Conditional Variance Model with Offset:
    --------------------------------------------------
    Distribution: Name = 'Gaussian'
               P: 1
               Q: 1
        Constant: NaN
           GARCH: {NaN} at Lags [1]
            ARCH: {NaN} at Lags [1]
          Offset: NaN
```

The mean offset appears in the output as an additional parameter to be estimated or otherwise specified.

Specify GARCH Model with Known Parameter Values

This example shows how to specify a GARCH model with known parameter values. You can use such a fully specified model as an input to `simulate` or `forecast`.

Specify the GARCH(1,1) model

$$\sigma_t^2 = 0.1 + 0.7\sigma_{t-1}^2 + 0.2\varepsilon_{t-1}^2$$

with a Gaussian innovation distribution.

```
Mdl = garch('Constant',0.1,'GARCH',0.7,'ARCH',0.2)
```

```
Mdl =

    GARCH(1,1) Conditional Variance Model:
    --------------------------------------
    Distribution: Name = 'Gaussian'
               P: 1
               Q: 1
        Constant: 0.1
           GARCH: {0.7} at Lags [1]
            ARCH: {0.2} at Lags [1]
```

Because all parameter values are specified, the created model has no `NaN` values. The functions `simulate` and `forecast` don't accept input models with `NaN` values.

Specify GARCH Model with t Innovation Distribution

This example shows how to specify a GARCH model with a Student's t innovation distribution.

Specify a GARCH(1,1) model with a mean offset,

$$y_t = \mu + \varepsilon_t,$$

where $\varepsilon_t = \sigma_t z_t$ and

$$\sigma_t^2 = \kappa + \gamma_1\sigma_{t-1}^2 + \alpha_1\varepsilon_{t-1}^2.$$

Assume z_t follows a Student's t innovation distribution with eight degrees of freedom.

```
tdist = struct('Name','t','DoF',8);
Mdl = garch('Offset',NaN,'GARCHLags',1,'ARCHLags',1,...
              'Distribution',tdist)
```

```
Mdl =

    GARCH(1,1) Conditional Variance Model with Offset:
    --------------------------------------------------
    Distribution: Name = 't', DoF = 8
               P: 1
               Q: 1
        Constant: NaN
           GARCH: {NaN} at Lags [1]
            ARCH: {NaN} at Lags [1]
          Offset: NaN
```

The value of `Distribution` is a `struct` array with field `Name` equal to `'t'` and field `DoF` equal to `8`. When you specify the degrees of freedom, they aren't estimated if you input the model to `estimate`.

Specify GARCH Model with Nonconsecutive Lags

This example shows how to specify a GARCH model with nonzero coefficients at nonconsecutive lags.

Specify a GARCH(3,1) model with nonzero GARCH coefficients at lags 1 and 3. Include a mean offset.

```
Mdl = garch('Offset',NaN,'GARCHLags',[1,3],'ARCHLags',1)
```

```
Mdl =

    GARCH(3,1) Conditional Variance Model with Offset:
    --------------------------------------------------
    Distribution: Name = 'Gaussian'
               P: 3
               Q: 1
        Constant: NaN
           GARCH: {NaN NaN} at Lags [1 3]
            ARCH: {NaN} at Lags [1]
          Offset: NaN
```

The unknown nonzero GARCH coefficients correspond to lagged variances at lags 1 and 3. The output shows only nonzero coefficients.

Display the value of `GARCH`.

```
Mdl.GARCH

ans =

    [NaN]    [0]    [NaN]
```

The GARCH cell array returns three elements. The first and third elements have value NaN, indicating these coefficients are nonzero and need to be estimated or otherwise specified. By default, garch sets the interim coefficient at lag 2 equal to zero to maintain consistency with MATLAB® cell array indexing.

See Also

estimate | forecast | garch | simulate | struct

Related Examples

- "Modify Properties of Conditional Variance Models" on page 6-42
- "Specify the Conditional Variance Model Innovation Distribution" on page 6-48
- "Specify Conditional Variance Model For Exchange Rates" on page 6-53
- "Specify Conditional Mean and Variance Models" on page 5-79

More About

- Using garch Objects
- "GARCH Model" on page 6-3

Specify EGARCH Models Using egarch

In this section...

Default EGARCH Model

The default EGARCH(P,Q) model in Econometrics Toolbox is of the form

$$\varepsilon_t = \sigma_t z_t,$$

with Gaussian innovation distribution and

$$\log \sigma_t^2 = \kappa + \sum_{i=1}^{P} \gamma_i \log \sigma_{t-i}^2 + \sum_{j=1}^{Q} \alpha_j \left[\frac{|\varepsilon_{t-j}|}{\sigma_{t-j}} - E\left\{ \frac{|\varepsilon_{t-j}|}{\sigma_{t-j}} \right\} \right] + \sum_{j=1}^{Q} \xi_j \left(\frac{\varepsilon_{t-j}}{\sigma_{t-j}} \right).$$

The default model has no mean offset, and the lagged log variances and standardized innovations are at consecutive lags.

You can specify a model of this form using the shorthand syntax `egarch(P,Q)`. For the input arguments `P` and `Q`, enter the number of lagged log variances (GARCH terms), P, and lagged standardized innovations (ARCH and leverage terms), Q, respectively. The following restrictions apply:

- P and Q must be nonnegative integers.
- If $P > 0$, then you must also specify $Q > 0$.

When you use this shorthand syntax, `egarch` creates an `egarch` model with these default property values.

Property	Default Value
`P`	Number of GARCH terms, P
`Q`	Number of ARCH and leverage terms, Q
`Offset`	`0`
`Constant`	`NaN`
`GARCH`	Cell vector of `NaN`s
`ARCH`	Cell vector of `NaN`s
`Leverage`	Cell vector of `NaN`s
`Distribution`	`'Gaussian'`

To assign nondefault values to any properties, you can modify the created model using dot notation.

To illustrate, consider specifying the EGARCH(1,1) model

$$\varepsilon_t = \sigma_t z_t,$$

with Gaussian innovation distribution and

$$\log\sigma_t^2 = \kappa + \gamma_1 \log\sigma_{t-1}^2 + \alpha_1\left[\frac{|\varepsilon_{t-1}|}{\sigma_{t-1}} - E\left\{\frac{|\varepsilon_{t-1}|}{\sigma_{t-1}}\right\}\right] + \xi_1\left(\frac{\varepsilon_{t-1}}{\sigma_{t-1}}\right).$$

```
Mdl = egarch(1,1)
```

```
Mdl =

    EGARCH(1,1) Conditional Variance Model:
    ----------------------------------------
    Distribution: Name = 'Gaussian'
               P: 1
               Q: 1
        Constant: NaN
           GARCH: {NaN} at Lags [1]
            ARCH: {NaN} at Lags [1]
        Leverage: {NaN} at Lags [1]
```

The created model, `Mdl`, has `NaN`s for all model parameters. A `NaN` value signals that a parameter needs to be estimated or otherwise specified by the user. All parameters must be specified to forecast or simulate the model

To estimate parameters, input the model (along with data) to `estimate`. This returns a new fitted `egarch` model. The fitted model has parameter estimates for each input `NaN` value.

Calling `egarch` without any input arguments returns an EGARCH(0,0) model specification with default property values:

```
DefaultMdl = egarch
```

```
DefaultMdl =

    EGARCH(0,0) Conditional Variance Model:
    ----------------------------------------
    Distribution: Name = 'Gaussian'
               P: 0
               Q: 0
        Constant: NaN
           GARCH: {}
            ARCH: {}
        Leverage: {}
```

Specify Default EGARCH Model

This example shows how to use the shorthand `egarch(P,Q)` syntax to specify the default EGARCH(P, Q) model, $\varepsilon_t = \sigma_t z_t$ with a Gaussian innovation distribution and

$$\log \sigma_t^2 = \kappa + \sum_{i=1}^{P} \gamma_i \log \sigma_{t-i}^2 + \sum_{j=1}^{Q} \alpha_j \left[\frac{|\varepsilon_{t-j}|}{\sigma_{t-j}} - E\left\{ \frac{|\varepsilon_{t-j}|}{\sigma_{t-j}} \right\} \right] + \sum_{j=1}^{Q} \xi_j \left(\frac{\varepsilon_{t-j}}{\sigma_{t-j}} \right).$$

By default, all parameters in the created model have unknown values.

Specify the default EGARCH(1,1) model:

```
Mdl = egarch(1,1)
```

```
Mdl =
```

```
EGARCH(1,1) Conditional Variance Model:
----------------------------------------
Distribution: Name = 'Gaussian'
           P: 1
           Q: 1
    Constant: NaN
       GARCH: {NaN} at Lags [1]
        ARCH: {NaN} at Lags [1]
    Leverage: {NaN} at Lags [1]
```

The output shows that the created model, `Mdl`, has `NaN` values for all model parameters: the constant term, the GARCH coefficient, the ARCH coefficient, and the leverage coefficient. You can modify the created model using dot notation, or input it (along with data) to `estimate`.

Using Name-Value Pair Arguments

The most flexible way to specify EGARCH models is using name-value pair arguments. You do not need, nor are you able, to specify a value for every model property. `egarch` assigns default values to any model properties you do not (or cannot) specify.

The general EGARCH(P,Q) model is of the form

$$y_t = \mu + \varepsilon_t,$$

where $\varepsilon_t = \sigma_t z_t$ and

$$\log \sigma_t^2 = \kappa + \sum_{i=1}^{P} \gamma_i \log \sigma_{t-i}^2 + \sum_{j=1}^{Q} \alpha_j \left[\frac{|\varepsilon_{t-j}|}{\sigma_{t-j}} - E\left\{ \frac{|\varepsilon_{t-j}|}{\sigma_{t-j}} \right\} \right] + \sum_{j=1}^{Q} \xi_j \left(\frac{\varepsilon_{t-j}}{\sigma_{t-j}} \right).$$

The innovation distribution can be Gaussian or Student's t. The default distribution is Gaussian.

In order to estimate, forecast, or simulate a model, you must specify the parametric form of the model (e.g., which lags correspond to nonzero coefficients, the innovation distribution) and any known parameter values. You can set any unknown parameters equal to `NaN`, and then input the model to `estimate` (along with data) to get estimated parameter values.

`egarch` (and `estimate`) returns a model corresponding to the model specification. You can modify models to change or update the specification. Input models (with no `NaN`

values) to `forecast` or `simulate` for forecasting and simulation, respectively. Here are some example specifications using name-value arguments.

Model	Specification
• $y_t = \varepsilon_t$ • $\varepsilon_t = \sigma_t z_t$ • z_t Gaussian • $\log\sigma_t^2 = \kappa + \gamma_1 \log\sigma_{t-1}^2 + \ldots$ $\alpha_1\left[\frac{\lvert\varepsilon_{t-1}\rvert}{\sigma_{t-1}} - E\left\{\frac{\lvert\varepsilon_{t-1}\rvert}{\sigma_{t-1}}\right\}\right] + \xi_1\left(\frac{\varepsilon_{t-1}}{\sigma_{t-1}}\right)$	`egarch('GARCH',NaN,'ARCH',NaN,...` `'Leverage',NaN)` or `egarch(1,1)`

• $y_t = \mu + \varepsilon_t$
• $\varepsilon_t = \sigma_t z_t$
• z_t Student's t with unknown degrees of freedom
• $\log\sigma_t^2 = \kappa + \gamma_1 \log\sigma_{t-1}^2 + \ldots$ $\alpha_1\left[\frac{\lvert\varepsilon_{t-1}\rvert}{\sigma_{t-1}} - E\left\{\frac{\lvert\varepsilon_{t-1}\rvert}{\sigma_{t-1}}\right\}\right] + \xi_1\left(\frac{\varepsilon_{t-1}}{\sigma_{t-1}}\right)$
	`egarch('Offset',NaN,'GARCH',NaN,...`
	`'ARCH',NaN,'Leverage',NaN,...`
	`'Distribution','t')`

• $y_t = \varepsilon_t$
• $\varepsilon_t = \sigma_t z_t$
• z_t Student's t with eight degrees of freedom
• $\log\sigma_t^2 = -0.1 + 0.4\log\sigma_{t-1}^2 + \ldots$ $0.3\left[\frac{\lvert\varepsilon_{t-1}\rvert}{\sigma_{t-1}} - E\left\{\frac{\lvert\varepsilon_{t-1}\rvert}{\sigma_{t-1}}\right\}\right] - 0.1\left(\frac{\varepsilon_{t-1}}{\sigma_{t-1}}\right)$
	`egarch('Constant',-0.1,'GARCH',0.4,...`
	`'ARCH',0.3,'Leverage',-0.1,...`
	`'Distribution',struct('Name','t','DoF',8))`

Here is a full description of the name-value arguments you can use to specify EGARCH models.

Note: You cannot assign values to the properties `P` and `Q`. `egarch` sets `P` equal to the largest GARCH lag, and `Q` equal to the largest lag with a nonzero standardized innovation coefficient, including ARCH and leverage coefficients.

Name-Value Arguments for EGARCH Models

Name	Corresponding EGARCH Model Term(s)	When to Specify
`Offset`	Mean offset, μ	To include a nonzero mean offset. For example, `'Offset',0.2`. If you plan to estimate the offset term, specify `'Offset',NaN`. By default, `Offset` has value `0` (meaning, no offset).
`Constant`	Constant in the conditional variance model, κ	To set equality constraints for κ. For example, if a model has known constant –0.1, specify `'Constant',-0.1`. By default, `Constant` has value `NaN`.
`GARCH`	GARCH coefficients, $\gamma_1,\ldots,\gamma_P$	To set equality constraints for the GARCH coefficients. For example, to specify an EGARCH(1,1) model with $\gamma_1 = 0.6$, specify `'GARCH',0.6`. You only need to specify the nonzero elements of `GARCH`. If the nonzero coefficients are at nonconsecutive lags, specify the corresponding lags using `GARCHLags`. Any coefficients you specify must satisfy all stationarity constraints.
`GARCHLags`	Lags corresponding to nonzero GARCH coefficients	`GARCHLags` is not a model property. Use this argument as a shortcut for specifying `GARCH` when the nonzero GARCH coefficients correspond to nonconsecutive lags. For example, to specify nonzero GARCH coefficients at lags 1 and 3, e.g., nonzero γ_1 and γ_3, specify `'GARCHLags',[1,3]`. Use `GARCH` and `GARCHLags` together to specify known nonzero GARCH coefficients at nonconsecutive lags. For example, if $\gamma_1 = 0.3$ and $\gamma_3 = 0.1$, specify `'GARCH',{0.3,0.1},'GARCHLags',[1,3]`

Name	Corresponding EGARCH Model Term(s)	When to Specify
`ARCH`	ARCH coefficients, $\alpha_1,\ldots,\alpha_Q$	To set equality constraints for the ARCH coefficients. For example, to specify an EGARCH(1,1) model with $\alpha_1 = 0.3$, specify `'ARCH',0.3`. You only need to specify the nonzero elements of `ARCH`. If the nonzero coefficients are at nonconsecutive lags, specify the corresponding lags using `ARCHLags`.
`ARCHLags`	Lags corresponding to nonzero ARCH coefficients	`ARCHLags` is not a model property. Use this argument as a shortcut for specifying `ARCH` when the nonzero ARCH coefficients correspond to nonconsecutive lags. For example, to specify nonzero ARCH coefficients at lags 1 and 3, e.g., nonzero α_1 and α_3, specify `'ARCHLags',[1,3]`. Use `ARCH` and `ARCHLags` together to specify known nonzero ARCH coefficients at nonconsecutive lags. For example, if $\alpha_1 = 0.4$ and $\alpha_3 = 0.2$, specify `'ARCH', {0.4,0.2},'ARCHLags',[1,3]`
`Leverage`	Leverage coefficients, $\xi_1,\ldots,\xi_Q$	To set equality constraints for the leverage coefficients. For example, to specify an EGARCH(1,1) model with $\xi_1 = -0.1$, specify `'Leverage',-0.1`. You only need to specify the nonzero elements of `Leverage`. If the nonzero coefficients are at nonconsecutive lags, specify the corresponding lags using `LeverageLags`.
`LeverageLags`	Lags corresponding to nonzero leverage coefficients	`LeverageLags` is not a model property. Use this argument as a shortcut for specifying `Leverage` when the nonzero leverage coefficients correspond to nonconsecutive lags. For example, to specify nonzero leverage coefficients at lags 1 and 3, e.g., nonzero ξ_1 and ξ_3, specify `'LeverageLags',[1,3]`.

Name	Corresponding EGARCH Model Term(s)	When to Specify
		Use `Leverage` and `LeverageLags` together to specify known nonzero leverage coefficients at nonconsecutive lags. For example, if $\xi_1 = -0.2$ and $\xi_3 = -0.1$, specify `'Leverage',{-0.2,-0.1},'LeverageLags',[1,3]`.
`Distribution`	Distribution of the innovation process	Use this argument to specify a Student's t innovation distribution. By default, the innovation distribution is Gaussian. For example, to specify a t distribution with unknown degrees of freedom, specify `'Distribution','t'`. To specify a t innovation distribution with known degrees of freedom, assign `Distribution` a data structure with fields `Name` and `DoF`. For example, for a t distribution with nine degrees of freedom, specify `'Distribution',struct('Name','t','DoF',9)`.

Specify EGARCH Model with Mean Offset

This example shows how to specify an EGARCH(P, Q) model with a mean offset. Use name-value pair arguments to specify a model that differs from the default model.

Specify an EGARCH(1,1) model with a mean offset,

$$y_t = \mu + \varepsilon_t,$$

where $\varepsilon_t = \sigma_t z_t$ and

$$\log \sigma_t^2 = \kappa + \gamma_1 \log \sigma_{t-1}^2 + \alpha_1 \left[\frac{|\varepsilon_{t-1}|}{\sigma_{t-1}} - E\left\{ \frac{|\varepsilon_{t-1}|}{\sigma_{t-1}} \right\} \right] + \xi_1 \left(\frac{\varepsilon_{t-1}}{\sigma_{t-1}} \right).$$

```
Mdl = egarch('Offset',NaN,'GARCHLags',1,'ARCHLags',1,...
    'LeverageLags',1)
```

```
Mdl =

    EGARCH(1,1) Conditional Variance Model with Offset:
    ---------------------------------------------------
    Distribution: Name = 'Gaussian'
               P: 1
               Q: 1
        Constant: NaN
           GARCH: {NaN} at Lags [1]
            ARCH: {NaN} at Lags [1]
        Leverage: {NaN} at Lags [1]
          Offset: NaN
```

The mean offset appears in the output as an additional parameter to be estimated or otherwise specified.

Specify EGARCH Model with Nonconsecutive Lags

This example shows how to specify an EGARCH model with nonzero coefficients at nonconsecutive lags.

Specify an EGARCH(3,1) model with nonzero GARCH terms at lags 1 and 3. Include a mean offset.

```
Mdl = egarch('Offset',NaN,'GARCHLags',[1,3],'ARCHLags',1,...
    'LeverageLags',1)
```

```
Mdl =

    EGARCH(3,1) Conditional Variance Model with Offset:
    ---------------------------------------------------
    Distribution: Name = 'Gaussian'
               P: 3
               Q: 1
        Constant: NaN
           GARCH: {NaN NaN} at Lags [1 3]
            ARCH: {NaN} at Lags [1]
        Leverage: {NaN} at Lags [1]
          Offset: NaN
```

The unknown nonzero GARCH coefficients correspond to lagged log variances at lags 1 and 3. The output shows only the nonzero coefficients.

Display the value of GARCH:

```
Mdl.GARCH
```

```
ans =

    [NaN]    [0]    [NaN]
```

The GARCH cell array returns three elements. The first and third elements have value NaN, indicating these coefficients are nonzero and need to be estimated or otherwise specified. By default, egarch sets the interim coefficient at lag 2 equal to zero to maintain consistency with MATLAB® cell array indexing.

Specify EGARCH Model with Known Parameter Values

This example shows how to specify an EGARCH model with known parameter values. You can use such a fully specified model as an input to simulate or forecast.

Specify the EGARCH(1,1) model

$$\log\sigma_t^2 = 0.1 + 0.6\log\sigma_{t-1}^2 + 0.2\left[\frac{|\varepsilon_{t-1}|}{\sigma_{t-1}} - E\left\{\frac{|\varepsilon_{t-1}|}{\sigma_{t-1}}\right\}\right] - 0.1\left(\frac{\varepsilon_{t-1}}{\sigma_{t-1}}\right)$$

with a Gaussian innovation distribution.

```
Mdl = egarch('Constant',0.1,'GARCH',0.6,'ARCH',0.2,...
    'Leverage',-0.1)
```

```
Mdl =

    EGARCH(1,1) Conditional Variance Model:
    ----------------------------------------
    Distribution: Name = 'Gaussian'
               P: 1
               Q: 1
        Constant: 0.1
           GARCH: {0.6} at Lags [1]
            ARCH: {0.2} at Lags [1]
        Leverage: {-0.1} at Lags [1]
```

Because all parameter values are specified, the created model has no `NaN` values. The functions `simulate` and `forecast` don't accept input models with `NaN` values.

Specify EGARCH Model with t Innovation Distribution

This example shows how to specify an EGARCH model with a Student's t innovation distribution.

Specify an EGARCH(1,1) model with a mean offset,

$$y_t = \mu + \varepsilon_t,$$

where $\varepsilon_t = \sigma_t z_t$ and

$$\log \sigma_t^2 = \kappa + \gamma_1 \log \sigma_{t-1}^2 + \alpha_1 \left[\frac{|\varepsilon_{t-1}|}{\sigma_{t-1}} - E\left\{ \frac{|\varepsilon_{t-1}|}{\sigma_{t-1}} \right\} \right] + \xi_1 \left(\frac{\varepsilon_{t-1}}{\sigma_{t-1}} \right).$$

Assume z_t follows a Student's t innovation distribution with 10 degrees of freedom.

```
tDist = struct('Name','t','DoF',10);
Mdl = egarch('Offset',NaN,'GARCHLags',1,'ARCHLags',1,...
    'LeverageLags',1,'Distribution',tDist)
```

```
Mdl =

    EGARCH(1,1) Conditional Variance Model with Offset:
    ---------------------------------------------------
    Distribution: Name = 't', DoF = 10
               P: 1
               Q: 1
        Constant: NaN
           GARCH: {NaN} at Lags [1]
            ARCH: {NaN} at Lags [1]
        Leverage: {NaN} at Lags [1]
          Offset: NaN
```

The value of `Distribution` is a `struct` array with field `Name` equal to `'t'` and field `DoF` equal to `10`. When you specify the degrees of freedom, they aren't estimated if you input the model to `estimate`.

See Also

`egarch` | `estimate` | `forecast` | `simulate` | `struct`

Related Examples

- "Modify Properties of Conditional Variance Models" on page 6-42
- "Specify the Conditional Variance Model Innovation Distribution" on page 6-48
- "Specify Conditional Mean and Variance Models" on page 5-79

More About

- Using egarch Objects
- "EGARCH Model" on page 6-4

Specify GJR Models Using gjr

In this section...

Default GJR Model

The default GJR(P,Q) model in Econometrics Toolbox is of the form

$$\varepsilon_t = \sigma_t z_t,$$

with Gaussian innovation distribution and

$$\sigma_t^2 = \kappa + \sum_{i=1}^{P} \gamma_i \sigma_{t-i}^2 + \sum_{j=1}^{Q} \alpha_j \varepsilon_{t-j}^2 + \sum_{j=1}^{Q} \xi_j I\left[\varepsilon_{t-j} < 0\right] \varepsilon_{t-j}^2.$$

The indicator function $I\left[\varepsilon_{t-j} < 0\right]$ equals 1 if $\varepsilon_{t-j} < 0$ and 0 otherwise. The default model has no mean offset, and the lagged variances and squared innovations are at consecutive lags.

You can specify a model of this form using the shorthand syntax `gjr(P,Q)`. For the input arguments `P` and `Q`, enter the number of lagged variances (GARCH terms), P, and lagged squared innovations (ARCH and leverage terms), Q, respectively. The following restrictions apply:

- P and Q must be nonnegative integers.
- If $P > 0$, then you must also specify $Q > 0$

When you use this shorthand syntax, `gjr` creates a `gjr` model with these default property values.

Property	Default Value
`P`	Number of GARCH terms, P
`Q`	Number of ARCH and leverage terms, Q
`Offset`	`0`
`Constant`	`NaN`
`GARCH`	Cell vector of `NaN`s
`ARCH`	Cell vector of `NaN`s
`Leverage`	Cell vector of `NaN`s
`Distribution`	`'Gaussian'`

To assign nondefault values to any properties, you can modify the created model using dot notation.

To illustrate, consider specifying the GJR(1,1) model

$$\varepsilon_t = \sigma_t z_t,$$

with Gaussian innovation distribution and

$$\sigma_t^2 = \kappa + \gamma_1 \sigma_{t-1}^2 + \alpha_1 \varepsilon_{t-1}^2 + \xi_1 I\left[\varepsilon_{t-1} < 0\right] \varepsilon_{t-1}^2.$$

```
Mdl = gjr(1,1)
```

```
Mdl =

    GJR(1,1) Conditional Variance Model:
    ------------------------------------
    Distribution: Name = 'Gaussian'
               P: 1
               Q: 1
        Constant: NaN
           GARCH: {NaN} at Lags [1]
```

```
           ARCH: {NaN} at Lags [1]
      Leverage: {NaN} at Lags [1]
```

The created model, `Mdl`, has `NaN`s for all model parameters. A `NaN` value signals that a parameter needs to be estimated or otherwise specified by the user. All parameters must be specified to forecast or simulate the model.

To estimate parameters, input the model (along with data) to `estimate`. This returns a new fitted `gjr` model. The fitted model has parameter estimates for each input `NaN` value.

Calling `gjr` without any input arguments returns a GJR(0,0) model specification with default property values:

```
DefaultMdl = gjr
```

```
DefaultMdl =

    GJR(0,0) Conditional Variance Model:
    ------------------------------------
    Distribution: Name = 'Gaussian'
               P: 0
               Q: 0
        Constant: NaN
           GARCH: {}
            ARCH: {}
        Leverage: {}
```

Specify Default GJR Model

This example shows how to use the shorthand `gjr(P,Q)` syntax to specify the default GJR(P, Q) model, $\varepsilon_t = \sigma_t z_t$ with a Gaussian innovation distribution and

$$\sigma_t^2 = \kappa + \sum_{i=1}^{P} \gamma_i \sigma_{t-i}^2 + \sum_{j=1}^{Q} \alpha_j \varepsilon_{t-j}^2 + \sum_{j=1}^{Q} \xi_j I\left[\varepsilon_{t-j} < 0\right] \varepsilon_{t-j}^2.$$

By default, all parameters in the created model have unknown values.

Specify the default GJR(1,1) model:

```
Mdl = gjr(1,1)
```

```
Mdl =

    GJR(1,1) Conditional Variance Model:
    ------------------------------------
    Distribution: Name = 'Gaussian'
               P: 1
               Q: 1
        Constant: NaN
           GARCH: {NaN} at Lags [1]
            ARCH: {NaN} at Lags [1]
        Leverage: {NaN} at Lags [1]
```

The output shows that the created model, `Mdl`, has `NaN` values for all model parameters: the constant term, the GARCH coefficient, the ARCH coefficient, and the leverage coefficient. You can modify the created model using dot notation, or input it (along with data) to `estimate`.

Using Name-Value Pair Arguments

The most flexible way to specify GJR models is using name-value pair arguments. You do not need, nor are you able, to specify a value for every model property. `gjr` assigns default values to any model properties you do not (or cannot) specify.

The general GJR(P,Q) model is of the form

$$y_t = \mu + \varepsilon_t,$$

where $\varepsilon_t = \sigma_t z_t$ and

$$\sigma_t^2 = \kappa + \sum_{i=1}^{P} \gamma_i \sigma_{t-i}^2 + \sum_{j=1}^{Q} \alpha_j \varepsilon_{t-j}^2 + \sum_{j=1}^{Q} \xi_j I\left[\varepsilon_{t-j} < 0\right] \varepsilon_{t-j}^2.$$

The innovation distribution can be Gaussian or Student's t. The default distribution is Gaussian.

In order to estimate, forecast, or simulate a model, you must specify the parametric form of the model (e.g., which lags correspond to nonzero coefficients, the innovation

distribution) and any known parameter values. You can set any unknown parameters equal to `NaN`, and then input the model to `estimate` (along with data) to get estimated parameter values.

`gjr` (and `estimate`) returns a model corresponding to the model specification. You can modify models to change or update the specification. Input models (with no `NaN` values) to `forecast` or `simulate` for forecasting and simulation, respectively. Here are some example specifications using name-value arguments.

Model	Specification
• $y_t = \varepsilon_t$ • $\varepsilon_t = \sigma_t z_t$ • z_t Gaussian • $\sigma_t^2 = \kappa + \gamma_1\sigma_{t-1}^2 + \alpha_1\varepsilon_{t-1}^2 + \xi_1 I[\varepsilon_{t-1} < 0]\varepsilon_{t-1}^2$	`gjr('GARCH',NaN,'ARCH',NaN,...` `'Leverage',NaN)` or `gjr(1,1)`

• $y_t = \mu + \varepsilon_t$
• $\varepsilon_t = \sigma_t z_t$
• z_t Student's t with unknown degrees of freedom
• $\sigma_t^2 = \kappa + \gamma_1\sigma_{t-1}^2 + \alpha_1\varepsilon_{t-1}^2 + \xi_1 I[\varepsilon_{t-1} < 0]\varepsilon_{t-1}^2$
	`gjr('Offset',NaN,'GARCH',NaN,...`
	`'ARCH',NaN,'Leverage',NaN,...`
	`'Distribution','t')`

• $y_t = \varepsilon_t$
• $\varepsilon_t = \sigma_t z_t$
• z_t Student's t with eight degrees of freedom
• $\sigma_t^2 = 0.1 + 0.6\sigma_{t-1}^2 + 0.3\varepsilon_{t-1}^2 + 0.05I[\varepsilon_{t-1} < 0]\varepsilon_{t-1}^2$
	`gjr('Constant',0.1,'GARCH',0.6,...`
	`'ARCH',0.3,'Leverage',0.05,...`
	`'Distribution',...`
	`struct('Name','t','DoF',8))`

Here is a full description of the name-value arguments you can use to specify GJR models.

Note: You cannot assign values to the properties `P` and `Q`. `egarch` sets `P` equal to the largest GARCH lag, and `Q` equal to the largest lag with a nonzero squared innovation coefficient, including ARCH and leverage coefficients.

Name-Value Arguments for GJR Models

Name	Corresponding GJR Model Term(s)	When to Specify
`Offset`	Mean offset, μ	To include a nonzero mean offset. For example, `'Offset',0.2`. If you plan to estimate the offset term, specify `'Offset',NaN`. By default, `Offset` has value `0` (meaning, no offset).
`Constant`	Constant in the conditional variance model, κ	To set equality constraints for κ. For example, if a model has known constant 0.1, specify `'Constant',0.1`. By default, `Constant` has value `NaN`.
`GARCH`	GARCH coefficients, $\gamma_1,\ldots,\gamma_P$	To set equality constraints for the GARCH coefficients. For example, to specify a GJR(1,1) model with $\gamma_1 = 0.6$, specify `'GARCH',0.6`. You only need to specify the nonzero elements of `GARCH`. If the nonzero coefficients are at nonconsecutive lags, specify the corresponding lags using `GARCHLags`. Any coefficients you specify must satisfy all stationarity constraints.
`GARCHLags`	Lags corresponding to the nonzero GARCH coefficients	`GARCHLags` is not a model property. Use this argument as a shortcut for specifying `GARCH` when the nonzero GARCH coefficients correspond to nonconsecutive lags. For example, to specify nonzero GARCH coefficients at lags 1 and 3, e.g., nonzero γ_1 and γ_3, specify `'GARCHLags',[1,3]`. Use `GARCH` and `GARCHLags` together to specify known nonzero GARCH coefficients at nonconsecutive lags. For example, if $\gamma_1 = 0.3$ and $\gamma_3 = 0.1$, specify `'GARCH',{0.3,0.1},'GARCHLags',[1,3]`
`ARCH`	ARCH coefficients, $\alpha_1,\ldots,\alpha_Q$	To set equality constraints for the ARCH coefficients. For example, to specify a GJR(1,1) model with $\alpha_1 = 0.3$, specify `'ARCH',0.3`. You only need to specify the nonzero elements of `ARCH`. If the nonzero coefficients are at nonconsecutive lags, specify the corresponding lags using `ARCHLags`.

Name	Corresponding GJR Model Term(s)	When to Specify
`ARCHLags`	Lags corresponding to nonzero ARCH coefficients	`ARCHLags` is not a model property. Use this argument as a shortcut for specifying `ARCH` when the nonzero ARCH coefficients correspond to nonconsecutive lags. For example, to specify nonzero ARCH coefficients at lags 1 and 3, e.g., nonzero α_1 and α_3, specify `'ARCHLags',[1,3]`. Use `ARCH` and `ARCHLags` together to specify known nonzero ARCH coefficients at nonconsecutive lags. For example, if $\alpha_1 = 0.4$ and $\alpha_3 = 0.2$, specify `'ARCH',{0.4,0.2},'ARCHLags',[1,3]`
`Leverage`	Leverage coefficients, $\xi_1,\ldots,\xi_Q$	To set equality constraints for the leverage coefficients. For example, to specify a GJR(1,1) model with $\xi_1 = 0.1$ specify `'Leverage',0.1`. You only need to specify the nonzero elements of `Leverage`. If the nonzero coefficients are at nonconsecutive lags, specify the corresponding lags using `LeverageLags`.
`LeverageLags`	Lags corresponding to nonzero leverage coefficients	`LeverageLags` is not a model property. Use this argument as a shortcut for specifying `Leverage` when the nonzero leverage coefficients correspond to nonconsecutive lags. For example, to specify nonzero leverage coefficients at lags 1 and 3, e.g., nonzero ξ_1 and ξ_3, specify `'LeverageLags',[1,3]`. Use `Leverage` and `LeverageLags` together to specify known nonzero leverage coefficients at nonconsecutive lags. For example, if $\xi_1 = 0.1$ and $\xi_3 = 0.05$, specify `'Leverage',{0.1,0.05},'LeverageLags',[1,3]`.
`Distribution`	Distribution of the innovation process	Use this argument to specify a Student's *t* innovation distribution. By default, the innovation distribution is Gaussian. For example, to specify a *t* distribution with unknown degrees of freedom, specify `'Distribution','t'`.

Name	Corresponding GJR Model Term(s)	When to Specify
		To specify a t innovation distribution with known degrees of freedom, assign `Distribution` a data structure with fields `Name` and `DoF`. For example, for a t distribution with nine degrees of freedom, specify `'Distribution',struct('Name','t','DoF',9)`.

Specify GJR Model with Mean Offset

This example shows how to specify a GJR(P, Q) model with a mean offset. Use name-value pair arguments to specify a model that differs from the default model.

Specify a GJR(1,1) model with a mean offset,

$$y_t = \mu + \varepsilon_t,$$

where $\varepsilon_t = \sigma_t z_t$ and

$$\sigma_t^2 = \kappa + \gamma_1 \sigma_{t-1}^2 + \alpha_1 \varepsilon_{t-1}^2 + \xi_1 I\left[\varepsilon_{t-1} < 0\right] \varepsilon_{t-1}^2.$$

```
Mdl = gjr('Offset',NaN,'GARCHLags',1,'ARCHLags',1,...
    'LeverageLags',1)
```

```
Mdl =

    GJR(1,1) Conditional Variance Model with Offset:
    ------------------------------------------------
    Distribution: Name = 'Gaussian'
               P: 1
               Q: 1
        Constant: NaN
           GARCH: {NaN} at Lags [1]
            ARCH: {NaN} at Lags [1]
        Leverage: {NaN} at Lags [1]
          Offset: NaN
```

The mean offset appears in the output as an additional parameter to be estimated or otherwise specified.

Specify GJR Model with Nonconsecutive Lags

This example shows how to specify a GJR model with nonzero coefficients at nonconsecutive lags.

Specify a GJR(3,1) model with nonzero GARCH terms at lags 1 and 3. Include a mean offset.

```
Mdl = gjr('Offset',NaN,'GARCHLags',[1,3],'ARCHLags',1,...
    'LeverageLags',1)
```

```
Mdl =

    GJR(3,1) Conditional Variance Model with Offset:
    ------------------------------------------------
    Distribution: Name = 'Gaussian'
               P: 3
               Q: 1
        Constant: NaN
           GARCH: {NaN NaN} at Lags [1 3]
            ARCH: {NaN} at Lags [1]
        Leverage: {NaN} at Lags [1]
          Offset: NaN
```

The unknown nonzero GARCH coefficients correspond to lagged variances at lags 1 and 3. The output shows only the nonzero coefficients.

Display the value of `GARCH`:

```
Mdl.GARCH
```

```
ans =

    [NaN]    [0]    [NaN]
```

The `GARCH` cell array returns three elements. The first and third elements have value `NaN`, indicating these coefficients are nonzero and need to be estimated or otherwise

specified. By default, `gjr` sets the interim coefficient at lag 2 equal to zero to maintain consistency with MATLAB® cell array indexing.

Specify GJR Model with Known Parameter Values

This example shows how to specify a GJR model with known parameter values. You can use such a fully specified model as an input to `simulate` or `forecast`.

Specify the GJR(1,1) model

$$\sigma_t^2 = 0.1 + 0.6\sigma_{t-1}^2 + 0.2\varepsilon_{t-1}^2 + 0.1I\left[\varepsilon_{t-1} < 0\right]\varepsilon_{t-1}^2$$

with a Gaussian innovation distribution.

```
Mdl = gjr('Constant',0.1,'GARCH',0.6,'ARCH',0.2,...
    'Leverage',0.1)
```

```
Mdl =

    GJR(1,1) Conditional Variance Model:
    ------------------------------------
    Distribution: Name = 'Gaussian'
               P: 1
               Q: 1
        Constant: 0.1
           GARCH: {0.6} at Lags [1]
            ARCH: {0.2} at Lags [1]
        Leverage: {0.1} at Lags [1]
```

Because all parameter values are specified, the created model has no `NaN` values. The functions `simulate` and `forecast` don't accept input models with `NaN` values.

Specify GJR Model with t Innovation Distribution

This example shows how to specify a GJR model with a Student's t innovation distribution.

Specify a GJR(1,1) model with a mean offset,

$$y_t = \mu + \varepsilon_t,$$

where $\varepsilon_t = \sigma_t z_t$ and

$$\sigma_t^2 = \kappa + \gamma_1 \sigma_{t-1}^2 + \alpha_1 \varepsilon_{t-1}^2 + \xi_1 I\left[\varepsilon_{t-1} < 0\right] \varepsilon_{t-1}^2.$$

Assume z_t follows a Student's t innovation distribution with 10 degrees of freedom.

```
tDist = struct('Name','t','DoF',10);
Mdl = gjr('Offset',NaN,'GARCHLags',1,'ARCHLags',1,...
              'LeverageLags',1,'Distribution',tDist)
```

```
Mdl =

    GJR(1,1) Conditional Variance Model with Offset:
    ------------------------------------------------
    Distribution: Name = 't', DoF = 10
               P: 1
               Q: 1
        Constant: NaN
           GARCH: {NaN} at Lags [1]
            ARCH: {NaN} at Lags [1]
        Leverage: {NaN} at Lags [1]
          Offset: NaN
```

The value of `Distribution` is a `struct` array with field `Name` equal to `'t'` and field `DoF` equal to `10`. When you specify the degrees of freedom, they aren't estimated if you input the model to `estimate`.

See Also

`estimate` | `forecast` | `gjr` | `simulate` | `struct`

Related Examples

- "Specify the Conditional Variance Model Innovation Distribution" on page 6-48
- "Modify Properties of Conditional Variance Models" on page 6-42

More About

- Using gjr Objects
- "GJR Model" on page 6-6

Modify Properties of Conditional Variance Models

In this section...

Dot Notation

A model created by `garch`, `egarch`, or `gjr` has values assigned to all model properties. To change any of these property values, you do not need to reconstruct the whole model. You can modify property values of an existing model using dot notation. That is, type the model name, then the property name, separated by `'.'` (a period).

For example, start with this model specification:

```
Mdl = garch(1,1)
```

```
Mdl =

    GARCH(1,1) Conditional Variance Model:
    --------------------------------------
    Distribution: Name = 'Gaussian'
               P: 1
               Q: 1
        Constant: NaN
           GARCH: {NaN} at Lags [1]
            ARCH: {NaN} at Lags [1]
```

The default model has no mean offset, so the `Offset` property does not appear in the model output. The property exists, however:

```
Offset = Mdl.Offset
```

```
Offset =

     0
```

Modify the model to add an unknown mean offset term:

```
Mdl.Offset = NaN
```

```
Mdl =

    GARCH(1,1) Conditional Variance Model with Offset:
    ----------------------------------------------------
    Distribution: Name = 'Gaussian'
               P: 1
               Q: 1
        Constant: NaN
           GARCH: {NaN} at Lags [1]
            ARCH: {NaN} at Lags [1]
          Offset: NaN
```

`Offset` now appears in the model output, with the updated nonzero value.

Be aware that every model property has a data type. Any modifications you make to a property value must be consistent with the data type of the property. For example, `GARCH` and `ARCH` (and `Leverage` for `egarch` and `gjr` models) are all cell vectors. This means you must index them using cell array syntax.

For example, start with the following model:

```
GJRMdl = gjr(1,1)
```

```
GJRMdl =

    GJR(1,1) Conditional Variance Model:
    --------------------------------------
    Distribution: Name = 'Gaussian'
               P: 1
               Q: 1
        Constant: NaN
           GARCH: {NaN} at Lags [1]
            ARCH: {NaN} at Lags [1]
        Leverage: {NaN} at Lags [1]
```

To modify the property value of `GARCH`, assign `GARCH` a cell array. Here, assign known GARCH coefficient values:

```
GJRMdl.GARCH = {0.6,0.2}
```

```
GJRMdl =

    GJR(2,1) Conditional Variance Model:
    ------------------------------------
    Distribution: Name = 'Gaussian'
               P: 2
               Q: 1
        Constant: NaN
           GARCH: {0.6 0.2} at Lags [1 2]
            ARCH: {NaN} at Lags [1]
        Leverage: {NaN} at Lags [1]
```

The updated model now has two GARCH terms (at lags 1 and 2) with the specified equality constraints.

Similarly, the data type of `Distribution` is a data structure. The default data structure has only one field, `Name`, with value `'Gaussian'`.

```
Distribution = GJRMdl.Distribution
```

```
Distribution =

    Name: 'Gaussian'
```

To modify the innovation distribution, assign `Distribution` a new name or data structure. The data structure can have up to two fields, `Name` and `DoF`. The second field corresponds to the degrees of freedom for a Student's t distribution, and is only required if `Name` has the value `'t'`.

To specify a Student's t distribution with unknown degrees of freedom, enter:

```
GJRMdl.Distribution = 't'
```

```
GJRMdl =

    GJR(2,1) Conditional Variance Model:
    ------------------------------------
    Distribution: Name = 't', DoF = NaN
               P: 2
               Q: 1
        Constant: NaN
```

```
           GARCH: {0.6 0.2} at Lags [1 2]
            ARCH: {NaN} at Lags [1]
        Leverage: {NaN} at Lags [1]
```

The updated model has a Student's t distribution with `NaN` degrees of freedom. To specify a t distribution with eight degrees of freedom, say:

```
GJRMdl.Distribution = struct('Name','t','DoF',8)
```

```
GJRMdl =

    GJR(2,1) Conditional Variance Model:
    ----------------------------------------
    Distribution: Name = 't', DoF = 8
               P: 2
               Q: 1
        Constant: NaN
           GARCH: {0.6 0.2} at Lags [1 2]
            ARCH: {NaN} at Lags [1]
        Leverage: {NaN} at Lags [1]
```

The degrees of freedom property in the model is updated. Note that the `DoF` field of `Distribution` is not directly assignable. For example, `GJRMdl.Distribution.DoF = 8` is not a valid assignment. However, you can get the individual fields:

```
DistributionDoF = GJRMdl.Distribution.DoF
```

```
DistributionDoF =

     8
```

Nonmodifiable Properties

Not all model properties are modifiable. You cannot change these properties in an existing model:

- `P`. This property updates automatically when the lag corresponding to the largest nonzero GARCH term changes.
- `Q`. This property updates automatically when the lag corresponding to the largest nonzero ARCH or leverage term changes.

Not all name-value pair arguments you can use for model creation are properties of the created model. Specifically, you can specify the arguments `GARCHLags` and `ARCHLags` (and `LeverageLags` for EGARCH and GJR models) during model creation. These are not, however, properties of `garch`, `egarch`, or `gjr` model. This means you cannot retrieve or modify them in an existing model.

The ARCH, GARCH, and leverage lags update automatically if you add any elements to (or remove from) the coefficient cell arrays `GARCH`, `ARCH`, or `Leverage`.

For example, specify an EGARCH(1,1) model:

```
Mdl = egarch(1,1)
```

```
Mdl =

    EGARCH(1,1) Conditional Variance Model:
    ----------------------------------------
    Distribution: Name = 'Gaussian'
               P: 1
               Q: 1
        Constant: NaN
           GARCH: {NaN} at Lags [1]
            ARCH: {NaN} at Lags [1]
        Leverage: {NaN} at Lags [1]
```

The model output shows nonzero GARCH, ARCH, and leverage coefficients at lag 1.

Add a new GARCH coefficient at lag 3:

```
Mdl.GARCH{3} = NaN
```

```
Mdl =

    EGARCH(3,1) Conditional Variance Model:
    ----------------------------------------
    Distribution: Name = 'Gaussian'
               P: 3
               Q: 1
        Constant: NaN
           GARCH: {NaN NaN} at Lags [1 3]
            ARCH: {NaN} at Lags [1]
        Leverage: {NaN} at Lags [1]
```

The nonzero GARCH coefficients at lags 1 and 3 now display in the model output. However, the cell array assigned to GARCH returns three elements:

```
garchCoefficients = Mdl.GARCH
```

```
garchCoefficients =

    [NaN]    [0]    [NaN]
```

GARCH has a zero coefficient at lag 2 to maintain consistency with traditional MATLAB® cell array indexing.

See Also

egarch | garch | gjr

Related Examples

- "Specify GARCH Models Using garch" on page 6-8
- "Specify EGARCH Models Using egarch" on page 6-19
- "Specify GJR Models Using gjr" on page 6-31
- "Specify the Conditional Variance Model Innovation Distribution" on page 6-48

More About

- Using garch Objects
- Using egarch Objects
- Using gjr Objects
- "GARCH Model" on page 6-3
- "EGARCH Model" on page 6-4
- "GJR Model" on page 6-6

Specify the Conditional Variance Model Innovation Distribution

In Econometrics Toolbox, the general form of the innovation process is $\varepsilon_t = \sigma_t z_t$. A conditional variance model specifies the parametric form of the conditional variance process. The innovation distribution corresponds to the distribution of the independent and identically distributed (iid) process z_t.

For the distribution of z_t, you can choose a standardized Gaussian or standardized Student's t distribution with $\nu > 2$ degrees of freedom. Note that if z_t follows a standardized t distribution, then

$$z_t = \sqrt{\frac{\nu - 2}{\nu}} T_\nu ,$$

where T_ν follows a Student's t distribution with $\nu > 2$ degrees of freedom.

The t distribution is useful for modeling time series with more extreme values than expected under a Gaussian distribution. Series with larger values than expected under normality are said to have *excess kurtosis*.

Tip It is good practice to assess the distributional properties of model residuals to determine if a Gaussian innovation distribution (the default distribution) is appropriate for your data.

The property `Distribution` in a model stores the distribution name (and degrees of freedom for the t distribution). The data type of `Distribution` is a `struct` array. For a Gaussian innovation distribution, the data structure has only one field: `Name`. For a Student's t distribution, the data structure must have two fields:

- `Name`, with value `'t'`
- `DoF`, with a scalar value larger than two (`NaN` is the default value)

If the innovation distribution is Gaussian, you do not need to assign a value to `Distribution`. `garch`, `egarch`, and `gjr` create the required data structure.

To illustrate, consider specifying a GARCH(1,1) model:

```
Mdl = garch(1,1)
```

```
Mdl =

    GARCH(1,1) Conditional Variance Model:
    ----------------------------------------
    Distribution: Name = 'Gaussian'
               P: 1
               Q: 1
        Constant: NaN
           GARCH: {NaN} at Lags [1]
            ARCH: {NaN} at Lags [1]
```

The model output shows that `Distribution` is a `struct` array with one field, `Name`, with the value `'Gaussian'`.

When specifying a Student's *t* innovation distribution, you can specify the distribution with either unknown or known degrees of freedom. If the degrees of freedom are unknown, you can simply assign `Distribution` the value `'t'`. By default, the property `Distribution` has a data structure with field `Name` equal to `'t'`, and field `DoF` equal to `NaN`. When you input the model to `estimate`, the degrees of freedom are estimated along with any other unknown model parameters.

For example, specify a GJR(2,1) model with an iid Student's *t* innovation distribution, with unknown degrees of freedom:

```
GJRMdl = gjr('GARCHLags',1:2,'ARCHLags',1,'LeverageLags',1,...
             'Distribution','t')
```

```
GJRMdl =

    GJR(2,1) Conditional Variance Model:
    ----------------------------------------
    Distribution: Name = 't', DoF = NaN
               P: 2
               Q: 1
        Constant: NaN
           GARCH: {NaN NaN} at Lags [1 2]
            ARCH: {NaN} at Lags [1]
        Leverage: {NaN} at Lags [1]
```

The output shows that `Distribution` is a data structure with two fields. Field `Name` has the value `'t'`, and field `DoF` has the value `NaN`.

If the degrees of freedom are known, and you want to set an equality constraint, assign a `struct` array to `Distribution` with fields `Name` and `DoF`. In this case, if the model is input to `estimate`, the degrees of freedom won't be estimated (the equality constraint is upheld).

Specify a GARCH(1,1) model with an iid Student's t distribution with eight degrees of freedom:

```
GARCHMdl = garch('GARCHLags',1,'ARCHLags',1,...
              'Distribution',struct('Name','t','DoF',8))
```

```
GARCHMdl =

    GARCH(1,1) Conditional Variance Model:
    --------------------------------------
    Distribution: Name = 't', DoF = 8
               P: 1
               Q: 1
        Constant: NaN
           GARCH: {NaN} at Lags [1]
            ARCH: {NaN} at Lags [1]
```

The output shows the specified innovation distribution.

After a model exists in the workspace, you can modify its `Distribution` property using dot notation. You cannot modify the fields of the `Distribution` data structure directly. For example, `GARCHMdl.Distribution.DoF = 8` is not a valid assignment. However, you can get the individual fields.

To change the distribution of the innovation process in an existing model to a Student's t distribution with unknown degrees of freedom, type:

```
Mdl.Distribution = 't';
```

To change the distribution to a t distribution with known degrees of freedom, use a data structure:

```
Mdl.Distribution = struct('Name','t','DoF',8);
```

You can get the individual `Distribution` fields:

```
tDoF = Mdl.Distribution.DoF
```

```
tDoF =

     8
```

To change the innovation distribution from a Student's t back to a Gaussian distribution, type:

```
Mdl.Distribution = 'Gaussian'
```

```
Mdl =

    GARCH(1,1) Conditional Variance Model:
    --------------------------------------
    Distribution: Name = 'Gaussian'
                     P: 1
                     Q: 1
                Constant: NaN
                   GARCH: {NaN} at Lags [1]
                    ARCH: {NaN} at Lags [1]
```

The `Name` field is updated to `'Gaussian'`, and there is no longer a `DoF` field.

See Also

`egarch` | `garch` | `gjr`

Related Examples

- "Specify GARCH Models Using garch" on page 6-8
- "Specify EGARCH Models Using egarch" on page 6-19
- "Specify GJR Models Using gjr" on page 6-31
- "Modify Properties of Conditional Variance Models" on page 6-42

More About

- Using garch Objects
- Using egarch Objects
- Using gjr Objects
- "GARCH Model" on page 6-3

- "EGARCH Model" on page 6-4
- "GJR Model" on page 6-6

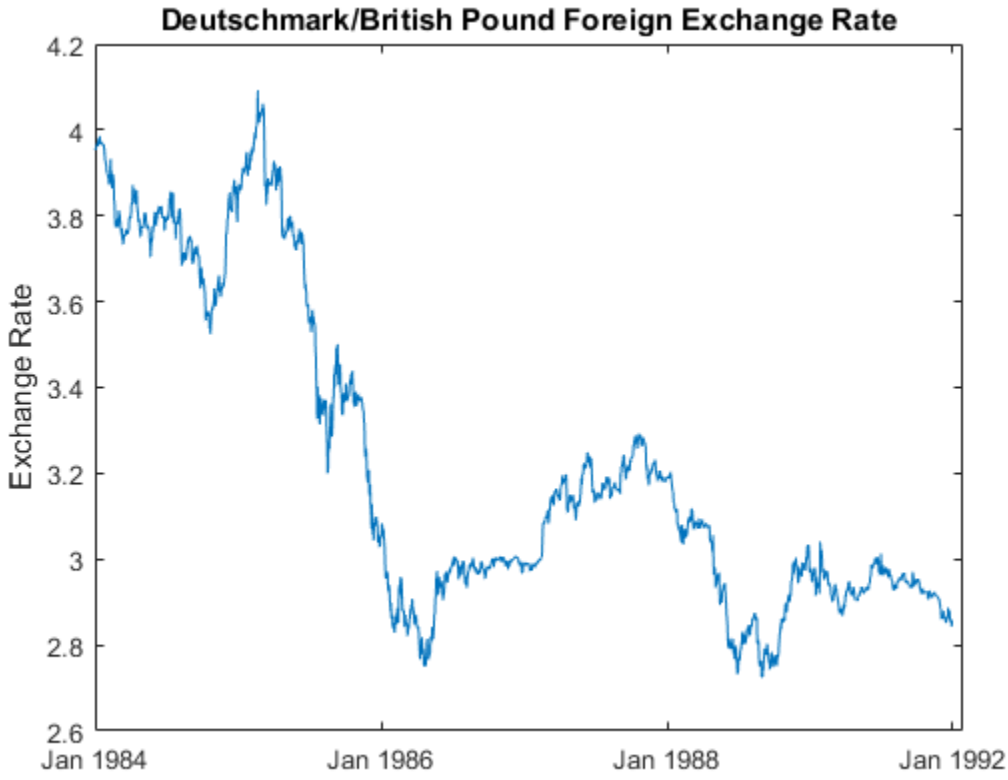
Specify Conditional Variance Model For Exchange Rates

This example shows how to specify a conditional variance model for daily Deutschmark/British pound foreign exchange rates observed from January 1984 to December 1991.

Load the Data.

Load the exchange rate data included with the toolbox.

```
load Data_MarkPound
y = Data;
T = length(y);

figure
plot(y)
h = gca;
h.XTick = [1 659 1318 1975];
h.XTickLabel = {'Jan 1984','Jan 1986','Jan 1988',...
     'Jan 1992'};
ylabel 'Exchange Rate';
title 'Deutschmark/British Pound Foreign Exchange Rate';
```



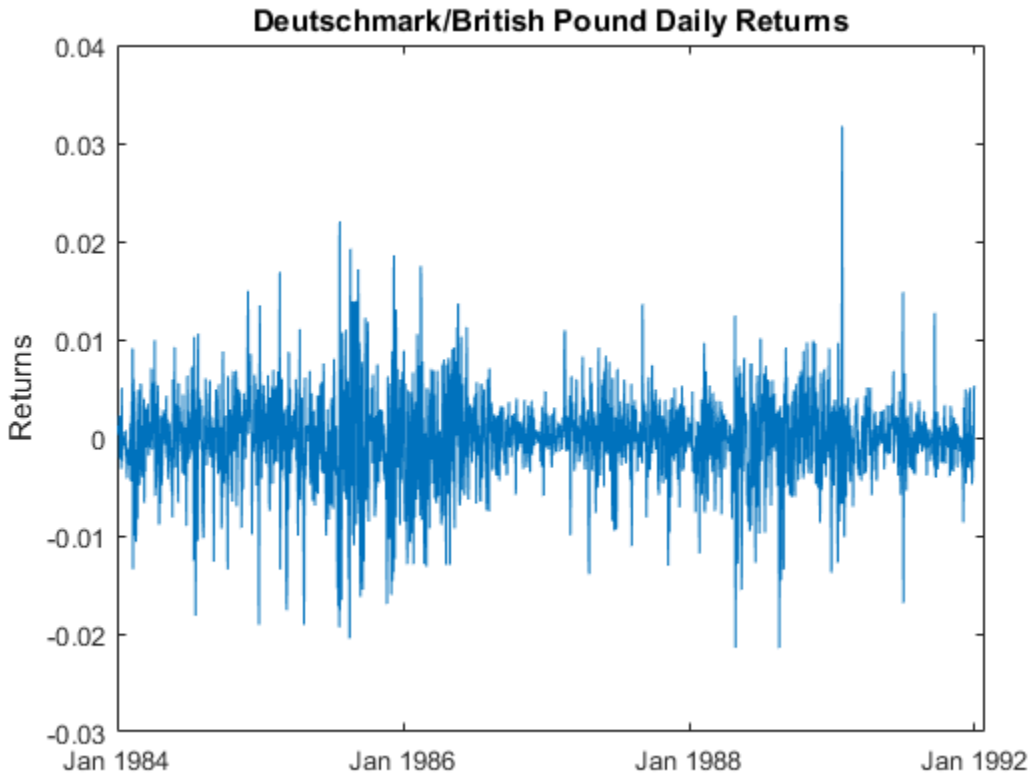
The exchange rate looks nonstationary (it does not appear to fluctuate around a fixed level).

Calculate the Returns.

Convert the series to returns. This results in the loss of the first observation.

```
r = price2ret(y);

figure
plot(2:T,r)
h2 = gca;
h2.XTick = [1 659 1318 1975];
h2.XTickLabel = {'Jan 1984','Jan 1986','Jan 1988',...
```

```
     'Jan 1992'};
ylabel 'Returns';
title 'Deutschmark/British Pound Daily Returns';
```



The returns series fluctuates around a common level, but exhibits volatility clustering. Large changes in the returns tend to cluster together, and small changes tend to cluster together. That is, the series exhibits conditional heteroscedasticity.

The returns are of relatively high frequency. Therefore, the daily changes can be small. For numerical stability, it is good practice to scale such data. In this case, scale the returns to percentage returns.

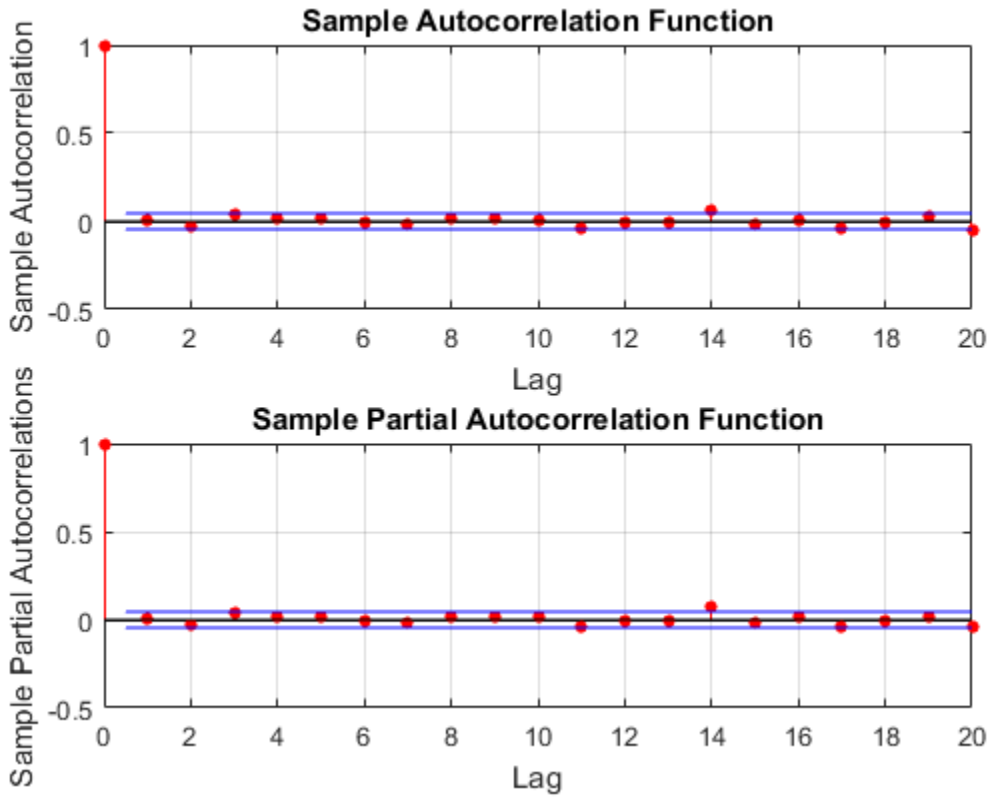
```
r = 100*r;
```

Check for Autocorrelation.

Check the returns series for autocorrelation. Plot the sample ACF and PACF, and conduct a Ljung-Box Q-test.

```
figure
subplot(2,1,1)
autocorr(r)
subplot(2,1,2)
parcorr(r)

[h,p] = lbqtest(r,[5 10 15])
```

```
h =

     0     0     0


p =

    0.3982    0.7278    0.2109
```



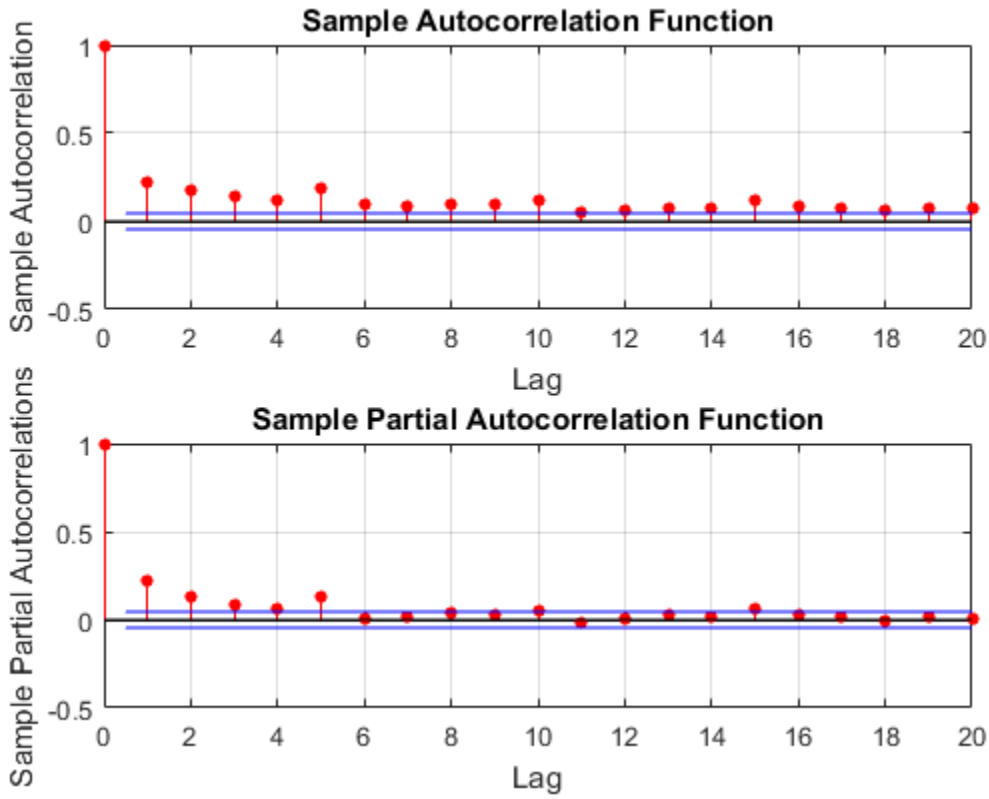
The sample ACF and PACF show virtually no significant autocorrelation. The Ljung-Box Q-test null hypothesis that all autocorrelations up to the tested lags are zero is not rejected for tests at lags 5, 10, and 15. This suggests that a conditional mean model is not needed for this returns series.

Check for Conditional Heteroscedasticity.

Check the return series for conditional heteroscedasticity. Plot the sample ACF and PACF of the squared returns series (after centering). Conduct Engle's ARCH test with a two-lag ARCH model alternative.

```
figure
subplot(2,1,1)
autocorr((r-mean(r)).^2)
```

```
subplot(2,1,2)
parcorr((r-mean(r)).^2)

[h,p] = archtest(r-mean(r),'lags',2)


h =

     1


p =

     0
```



The sample ACF and PACF of the squared returns show significant autocorrelation. This suggests a GARCH model with lagged variances and lagged squared innovations might be appropriate for modeling this series. Engle's ARCH test rejects the null hypothesis (`h = 1`) of no ARCH effects in favor of the alternative ARCH model with two lagged squared innovations. An ARCH model with two lagged innovations is locally equivalent to a GARCH(1,1) model.

Specify a GARCH(1,1) Model.

Based on the autocorrelation and conditional heteroscedasticity specification testing, specify the GARCH(1,1) model with a mean offset:

$$y_t = \mu + \varepsilon_t,$$

with $\varepsilon_t = \sigma_t z_t$ and

$$\sigma_t^2 = \kappa + \gamma_1 \sigma_{t-1}^2 + \alpha_1 \varepsilon_{t-1}^2.$$

Assume a Gaussian innovation distribution.

```
Mdl = garch('Offset',NaN,'GARCHLags',1,'ARCHLags',1)
```

```
Mdl =

    GARCH(1,1) Conditional Variance Model with Offset:
    --------------------------------------------------
    Distribution: Name = 'Gaussian'
               P: 1
               Q: 1
        Constant: NaN
           GARCH: {NaN} at Lags [1]
            ARCH: {NaN} at Lags [1]
          Offset: NaN
```

The created model, `Mdl`, has `NaN` values for all unknown parameters in the specified GARCH(1,1) model.

You can pass the GARCH model `Mdl` and `r` into `estimate` to estimate the parameters.

See Also

`archtest` | `autocorr` | `garch` | `lbqtest` | `parcorr`

Related Examples

- "Likelihood Ratio Test for Conditional Variance Models" on page 6-83
- "Simulate Conditional Variance Model" on page 6-111
- "Forecast a Conditional Variance Model" on page 6-126

More About

- Using garch Objects
- "GARCH Model" on page 6-3
- "Autocorrelation and Partial Autocorrelation" on page 3-13

- "Ljung-Box Q-Test" on page 3-16
- "Engle's ARCH Test" on page 3-25

Maximum Likelihood Estimation for Conditional Variance Models

In this section...

Innovation Distribution

For conditional variance models, the innovation process is $\varepsilon_t = \sigma_t z_t$, where z_t follows a standardized Gaussian or Student's t distribution with $\nu > 2$ degrees of freedom. Specify your distribution choice in the model property `Distribution`.

The innovation variance, σ_t^2, can follow a GARCH, EGARCH, or GJR conditional variance process.

If the model includes a mean offset term, then

$$\varepsilon_t = y_t - \mu.$$

The `estimate` function for `garch`, `egarch`, and `gjr` models estimates parameters using maximum likelihood estimation. `estimate` returns fitted values for any parameters in the input model equal to `NaN`. `estimate` honors any equality constraints in the input model, and does not return estimates for parameters with equality constraints.

Loglikelihood Functions

Given the history of a process, innovations are conditionally independent. Let H_t denote the history of a process available at time t, $t = 1,...,N$. The likelihood function for the innovation series is given by

$$f(\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_N \mid H_{N-1}) = \prod_{t=1}^{N} f(\varepsilon_t \mid H_{t-1}),$$

where f is a standardized Gaussian or t density function.

The exact form of the loglikelihood objective function depends on the parametric form of the innovation distribution.

- If z_t has a standard Gaussian distribution, then the loglikelihood function is

$$LLF = -\frac{N}{2}\log(2\pi) - \frac{1}{2}\sum_{t=1}^{N}\log\sigma_t^2 - \frac{1}{2}\sum_{t=1}^{N}\frac{\varepsilon_t^2}{\sigma_t^2}.$$

- If z_t has a standardized Student's t distribution with $\nu > 2$ degrees of freedom, then the loglikelihood function is

$$LLF = N\log\left[\frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\sqrt{\pi(\nu-2)}\Gamma\left(\frac{\nu}{2}\right)}\right] - \frac{1}{2}\sum_{t=1}^{N}\log\sigma_t^2 - \frac{\nu+1}{2}\sum_{t=1}^{N}\log\left[1+\frac{\varepsilon_t^2}{\sigma_t^2(\nu-2)}\right].$$

`estimate` performs covariance matrix estimation for maximum likelihood estimates using the outer product of gradients (OPG) method.

See Also

`estimate`

Related Examples

- "Likelihood Ratio Test for Conditional Variance Models" on page 6-83
- "Compare Conditional Variance Models Using Information Criteria" on page 6-87

More About

- Using garch Objects
- Using egarch Objects
- Using gjr Objects
- "Conditional Variance Model Estimation with Equality Constraints" on page 6-65
- "Presample Data for Conditional Variance Model Estimation" on page 6-67

- "Initial Values for Conditional Variance Model Estimation" on page 6-69
- "Optimization Settings for Conditional Variance Model Estimation" on page 6-71

Conditional Variance Model Estimation with Equality Constraints

For conditional variance model estimation, the required inputs for `estimate` are a model and a vector of univariate time series data. The model specifies the parametric form of the conditional variance model being estimated. `estimate` returns fitted values for any parameters in the input model with `NaN` values. If you specify non-`NaN` values for any parameters, `estimate` views these values as equality constraints and honors them during estimation.

For example, suppose you are estimating a model with a mean offset known to be 0.3. To indicate this, specify `'Offset',0.3` in the model you input to `estimate`. `estimate` views this non-`NaN` value as an equality constraint, and does not estimate the mean offset. `estimate` also honors all specified equality constraints during estimation of the parameters without equality constraints.

`estimate` optionally returns the variance-covariance matrix for estimated parameters. The parameters in the variance-covariance matrix are ordered as follows:

- Constant
- Nonzero GARCH coefficients at positive lags
- Nonzero ARCH coefficients at positive lags
- Nonzero leverage coefficients at positive lags (EGARCH and GJR models only)
- Degrees of freedom (t innovation distribution only)
- Offset (models with nonzero offset only)

If any parameter known to the optimizer has an equality constraint, the corresponding row and column of the variance-covariance matrix has all zeros.

In addition to user-specified equality constraints, note that `estimate` sets any GARCH, ARCH, or leverage coefficient with an estimate less than `1e-12` in magnitude equal to zero.

See Also

`estimate`

More About

- Using garch Objects

- Using egarch Objects
- Using gjr Objects
- “Maximum Likelihood Estimation for Conditional Variance Models” on page 6-62
- “Presample Data for Conditional Variance Model Estimation” on page 6-67
- “Initial Values for Conditional Variance Model Estimation” on page 6-69
- “Optimization Settings for Conditional Variance Model Estimation” on page 6-71

Presample Data for Conditional Variance Model Estimation

Presample data is data from time points before the beginning of the observation period. In Econometrics Toolbox, you can specify your own presample data or use automatically generated presample data.

In a conditional variance model, the current value of the innovation conditional variance, σ_t^2, depends on historical information. Historical information includes past conditional variances, $\sigma_1^2,\sigma_2^2,\ldots,\sigma_{t-1}^2$, and past innovations, $\varepsilon_1,\varepsilon_2,\ldots,\varepsilon_{t-1}$.

The number of past variances and innovations that a current conditional variance depends on is determined by the degree of the conditional variance model. For example, in a GARCH(1,1) model, each conditional variance depends on one lagged variance and one lagged squared innovation,

$$\sigma_t^2 = \kappa + \gamma_1\sigma_{t-1}^2 + \alpha_1\varepsilon_{t-1}^2.$$

In general, difficulties arise at the beginning of the series because the likelihood contribution of the first few innovations is conditional on historical information that is not observed. In the GARCH(1,1) example, σ_1^2 depends on σ_0^2 and ε_0. These values are not observed.

For the GARCH(P,Q) and GJR(P,Q) models, P presample variances and Q presample innovations are needed to initialize the variance equation. For an EGARCH(P,Q) model, max(P,Q) presample variances and Q presample innovations are needed to initialize the variance equation.

If you want to specify your own presample variances and innovations to `estimate`, use the name-value arguments `V0` and `E0`, respectively.

By default, `estimate` generates automatic presample data as follows. For GARCH and GJR models:

- Presample innovations are set to an estimate of the unconditional standard deviation of the innovation series. If there is a mean offset term, presample innovations are specified as the sample standard deviation of the offset-adjusted series. If there is no mean offset, presample innovations are specified as the square root of the sample mean of the squared response series.

- Presample variances are set to an estimate of the unconditional variance of the innovation series. If there is a mean offset term, the presample innovations are specified as the sample mean of the squared offset-adjusted series. If there is no mean offset, presample variances are specified as the sample mean of the squared response series.

For EGARCH models:

- Presample variances are computed as for GARCH and GJR models.
- Presample innovations are set to zero.

See Also

`estimate`

More About

- Using garch Objects
- Using egarch Objects
- Using gjr Objects
- “Maximum Likelihood Estimation for Conditional Variance Models” on page 6-62
- “Conditional Variance Model Estimation with Equality Constraints” on page 6-65
- “Initial Values for Conditional Variance Model Estimation” on page 6-69
- “Optimization Settings for Conditional Variance Model Estimation” on page 6-71

Initial Values for Conditional Variance Model Estimation

The `estimate` function for conditional variance models uses `fmincon` from Optimization Toolbox to perform maximum likelihood estimation. This optimization function requires initial (or, starting) values to begin the optimization process.

If you want to specify your own initial values, use name-value arguments. For example, specify initial values for GARCH coefficients using the name-value argument `GARCH0`.

Alternatively, you can let `estimate` choose default initial values. Default initial values are generated using standard time series techniques. If you partially specify initial values (that is, specify initial values for some parameters), `estimate` honors the initial values you do specify, and generates default initial values for the remaining parameters.

When generating initial values, `estimate` enforces any stationarity and positivity constraints for the conditional variance model being estimated. The techniques `estimate` uses to generate default initial values are as follows:

- For the GARCH and GJR models, the model is transformed to an equivalent ARMA model for the squared, offset-adjusted response series. Note that the GJR model is treated like a GARCH model, with all leverage coefficients equal to zero. The initial ARMA values are solved for using the modified Yule-Walker equations as described in Box, Jenkins, and Reinsel [1]. The initial GARCH and ARCH starting values are calculated by transforming the ARMA starting values back to the original GARCH (or GJR) representation.
- For the EGARCH model, the initial GARCH coefficient values are found by viewing the model as an equivalent ARMA model for the squared, offset-adjusted log response series. The initial GARCH values are solved for using Yule-Walker equations as described in Box, Jenkins, and Reinsel [1]. For the other coefficients, the first nonzero ARCH coefficient is set to a small positive value, and the first nonzero leverage coefficient is set to a small negative value (consistent with the expected signs of these coefficients).

References

[1] Box, G. E. P., G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis: Forecasting and Control*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

See Also

`estimate` | `fmincon`

More About

- Using garch Objects
- Using egarch Objects
- Using gjr Objects
- "Maximum Likelihood Estimation for Conditional Variance Models" on page 6-62
- "Conditional Variance Model Estimation with Equality Constraints" on page 6-65
- "Presample Data for Conditional Variance Model Estimation" on page 6-67
- "Optimization Settings for Conditional Variance Model Estimation" on page 6-71

Optimization Settings for Conditional Variance Model Estimation

In this section...

Optimization Options

`estimate` maximizes the loglikelihood function using `fmincon` from Optimization Toolbox. `fmincon` has many optimization options, such as choice of optimization algorithm and constraint violation tolerance. Choose optimization options using `optimoptions`.

`estimate` uses the `fmincon` optimization options by default, with these exceptions. For details, see `fmincon` and `optimoptions` in Optimization Toolbox.

optimoptions Properties	Description	estimate Settings
`Algorithm`	Algorithm for minimizing the negative loglikelihood function	`'sqp'`
`Display`	Level of display for optimization progress	`'off'`
`Diagnostics`	Display for diagnostic information about the function to be minimized	`'off'`
`TolCon`	Termination tolerance on constraint violations	`1e-7`

If you want to use optimization options that differ from the default, then set your own using `optimoptions`.

For example, suppose that you want `estimate` to display optimization diagnostics. The best practice is to set the name-value pair argument `'Display','diagnostics'` in `estimate`. Alternatively, you can direct the optimizer to display optimization diagnostics.

Define a GARCH(1,1) model (`Mdl`) and simulate data from it.

```
Mdl = garch('ARCH',0.2,'GARCH',0.5,'Constant',0.5);
rng(1);
y = simulate(Mdl,500);
```

`Mdl` does not have a regression component. By default, `fmincon` does not display the optimization diagnostics. Use `optimoptions` to set it to display the optimization diagnostics, and set the other `fmincon` properties to the default settings of `estimate` listed in the previous table.

```
options = optimoptions(@fmincon,'Diagnostics','on','Algorithm',...
    'sqp','Display','off','TolCon',1e-7)
% @fmincon is the function handle for fmincon


options =

  fmincon options:

   Options used by current Algorithm ('sqp'):
   (Other available algorithms: 'active-set', 'interior-point', 'trust-region-reflectiv

   Set by user:
          Algorithm: 'sqp'
        Diagnostics: 'on'
            Display: 'off'
             TolCon: 1.0000e-07

   Default:
    DerivativeCheck: 'off'
      DiffMaxChange: Inf
      DiffMinChange: 0
     FinDiffRelStep: 'sqrt(eps)'
        FinDiffType: 'forward'
        FunValCheck: 'off'
         GradConstr: 'off'
            GradObj: 'off'
        MaxFunEvals: '100*numberOfVariables'
            MaxIter: 400
     ObjectiveLimit: -1.0000e+20
          OutputFcn: []
           PlotFcns: []
       ScaleProblem: 'none'
             TolFun: 1.0000e-06
               TolX: 1.0000e-06
           TypicalX: 'ones(numberOfVariables,1)'
```

```
        UseParallel: 0

   Options not used by current Algorithm ('sqp')
   Default:
    AlwaysHonorConstraints: 'bounds'
                   HessFcn: []
                  HessMult: []
                HessPattern: 'sparse(ones(numberOfVariables))'
                    Hessian: 'not applicable'
          InitBarrierParam: 0.1000
     InitTrustRegionRadius: 'sqrt(numberOfVariables)'
                MaxPCGIter: 'max(1,floor(numberOfVariables/2))'
              MaxProjCGIter: '2*(numberOfVariables-numberOfEqualities)'
                 MaxSQPIter: '10*max(numberOfVariables,numberOfInequalities+…'
          PrecondBandWidth: 0
            RelLineSrchBnd: []
    RelLineSrchBndDuration: 1
       SubproblemAlgorithm: 'ldl-factorization'
                 TolConSQP: 1.0000e-06
                    TolPCG: 0.1000
                  TolProjCG: 0.0100
               TolProjCGAbs: 1.0000e-10
```

The options that you set appear under the `Set by user:` heading. The properties under the `Default:` heading are other options that you can set.

Fit `Mdl` to `y` using the new optimization options.

```
ToEstMdl = garch(1,1);
EstMdl = estimate(ToEstMdl,y,'Options',options);
```

```
_____________________________________________________________
   Diagnostic Information

Number of variables: 3

Functions
Objective:                              @(X)Mdl.nLogLikeGaussian(X,V,E,Lags,1,maxPQ,T,nar
Gradient:                               finite-differencing
Hessian:                                finite-differencing (or Quasi-Newton)
```

```
Constraints
Nonlinear constraints:                do not exist

Number of linear inequality constraints:    1
Number of linear equality constraints:      0
Number of lower bound constraints:          3
Number of upper bound constraints:          3

Algorithm selected
   sqp


____________________________________________________________
   End diagnostic information

    GARCH(1,1) Conditional Variance Model:
    ----------------------------------------
    Conditional Probability Distribution: Gaussian

                               Standard          t
     Parameter       Value       Error       Statistic
    -----------   -----------   ------------   -----------
     Constant       0.431451      0.465646       0.926565
     GARCH{1}       0.314347      0.249922        1.25778
      ARCH{1}       0.571428      0.326773         1.7487
```

Note:

- `estimate` numerically maximizes the loglikelihood function potentially using equality, inequality, and lower and upper bound constraints. If you set `Algorithm` to anything other than `sqp`, then check that the algorithm supports similar constraints, such as `interior-point`. For example, `fmincon` sets `Algorithm` to `trust-region-reflective` by default. `trust-region-reflective` does not support inequality constraints. Therefore, if you do not change the default `Algorithm` property value of `fmincon`, then `estimate` displays a warning. During estimation, `fmincon` temporarily sets `Algorithm` to `active-set` by default to satisfy the constraints.
- `estimate` sets a constraint level of `TolCon` so constraints are not violated. Be aware that an estimate with an active constraint has unreliable standard errors since variance-covariance estimation assumes the likelihood function is locally quadratic around the maximum likelihood estimate.

Conditional Variance Model Constraints

The software enforces these constraints while estimating a GARCH model:

- Covariance-stationarity,

$$\sum_{i=1}^{P}\gamma_i+\sum_{j=1}^{Q}\alpha_j<1$$

- Positivity of GARCH and ARCH coefficients
- Model constant strictly greater than zero
- For a t innovation distribution, degrees of freedom strictly greater than two

For GJR models, the constraints enforced during estimation are:

- Covariance-stationarity constraint,

$$\sum_{i=1}^{P}\gamma_i+\sum_{j=1}^{Q}\alpha_j+\frac{1}{2}\sum_{j=1}^{Q}\xi_j<1$$

- Positivity constraints on the GARCH and ARCH coefficients
- Positivity on the sum of ARCH and leverage coefficients,

$$\alpha_j+\xi_j\geq 0,\ \ j=1,\ldots,Q$$

- Model constant strictly greater than zero
- For a t innovation distribution, degrees of freedom strictly greater than two

For EGARCH models, the constraints enforced during estimation are:

- Stability of the GARCH coefficient polynomial
- For a t innovation distribution, degrees of freedom strictly greater than two

See Also

`estimate` | `fmincon` | `optimoptions`

More About

- Using garch Objects

- Using egarch Objects
- Using gjr Objects
- “Maximum Likelihood Estimation for Conditional Variance Models” on page 6-62
- “Conditional Variance Model Estimation with Equality Constraints” on page 6-65
- “Presample Data for Conditional Variance Model Estimation” on page 6-67
- “Initial Values for Conditional Variance Model Estimation” on page 6-69

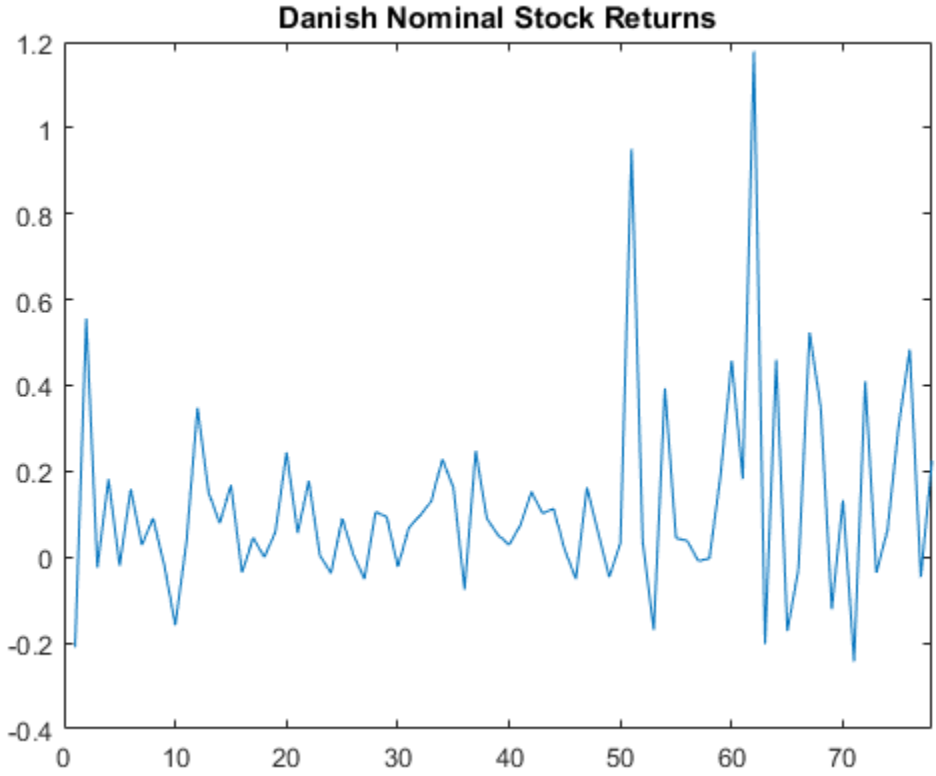
Infer Conditional Variances and Residuals

This example shows how to infer conditional variances from a fitted conditional variance model. Standardized residuals are computed using the inferred conditional variances to check the model fit.

Step 1. Load the data.

Load the Danish nominal stock return data included with the toolbox.

```
load Data_Danish
y = DataTable.RN;
T = length(y);

figure
plot(y)
xlim([0,T])
title('Danish Nominal Stock Returns')
```



The return series appears to have a nonzero mean offset and volatility clustering.

Step 2. Fit an EGARCH(1,1) model.

Specify, and then fit an EGARCH(1,1) model to the nominal stock returns series. Include a mean offset, and assume a Gaussian innovation distribution.

```
Mdl = egarch('Offset',NaN,'GARCHLags',1,...
    'ARCHLags',1,'LeverageLags',1);
EstMdl = estimate(Mdl,y);
```

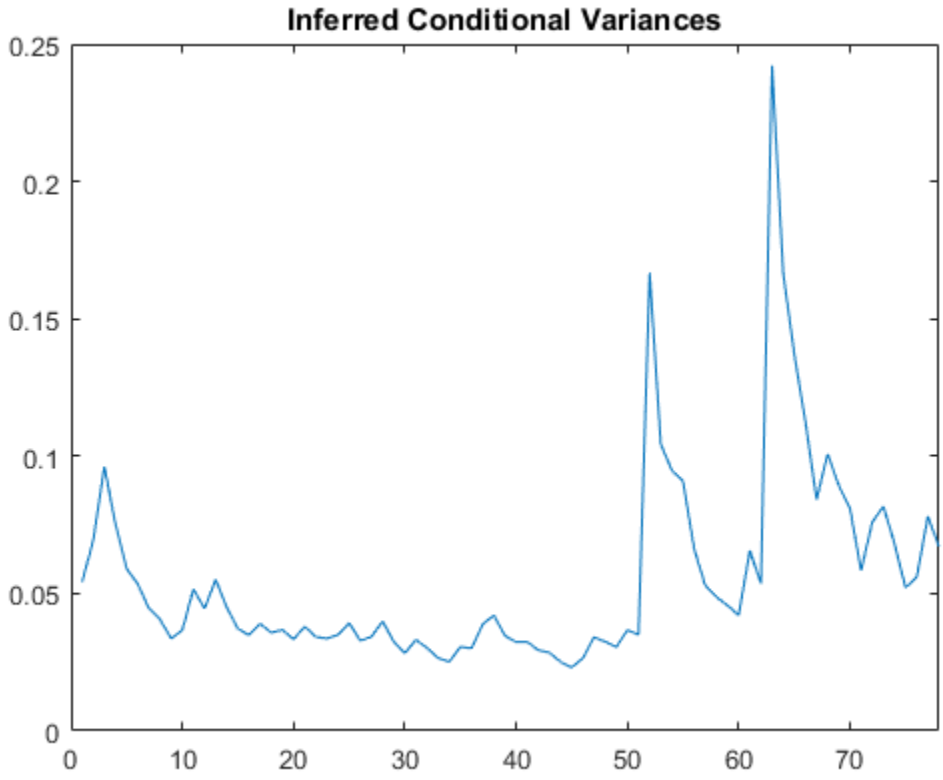
```
    EGARCH(1,1) Conditional Variance Model:
    ----------------------------------------
```

```
    Conditional Probability Distribution: Gaussian

                                Standard          t
     Parameter       Value        Error       Statistic
    -----------   -----------   ------------   -----------
     Constant       -0.62723      0.744007      -0.843043
     GARCH{1}       0.774189       0.23628        3.27658
      ARCH{1}       0.386361      0.373606        1.03414
  Leverage{1}    -0.00249918       0.19222     -0.0130016
       Offset        0.10325     0.0377269        2.73676
```

Step 3. Infer the conditional variances.

Infer the conditional variances using the fitted model.

```
v = infer(EstMdl,y);

figure
plot(v)
xlim([0,T])
title('Inferred Conditional Variances')
```



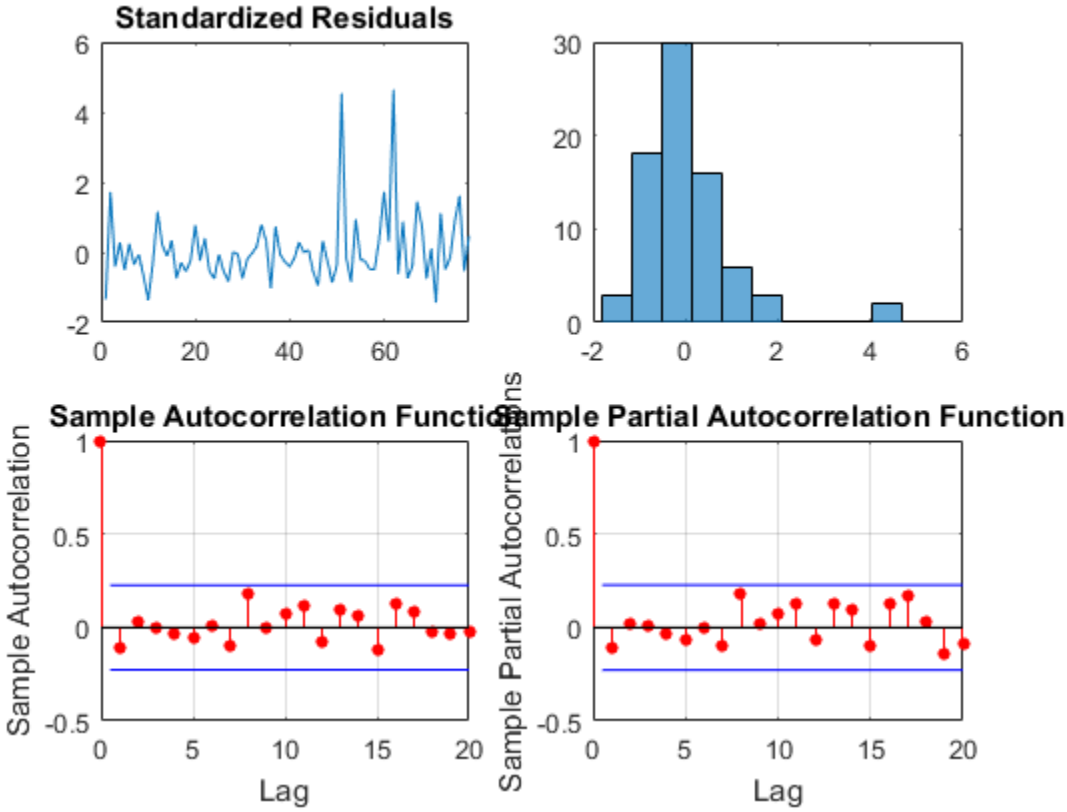
The inferred conditional variances show increased volatility at the end of the return series.

Step 4. Compute the standardized residuals.

Compute the standardized residuals for the model fit. Subtract the estimated mean offset, and divide by the square root of the conditional variance process.

```
res = (y-EstMdl.Offset)./sqrt(v);

figure
subplot(2,2,1)
plot(res)
xlim([0,T])
```

```
title('Standardized Residuals')

subplot(2,2,2)
histogram(res,10)

subplot(2,2,3)
autocorr(res)

subplot(2,2,4)
parcorr(res)
```



The standardized residuals exhibit no residual autocorrelation. There are a few residuals larger than expected for a Gaussian distribution, but the normality assumption is not unreasonable.

See Also

`autocorr` | `egarch` | `estimate` | `infer` | `parcorr`

Related Examples

- “Specify Conditional Variance Model For Exchange Rates” on page 6-53
- “Likelihood Ratio Test for Conditional Variance Models” on page 6-83

More About

- Using egarch Objects
- “Goodness of Fit” on page 3-88
- “Residual Diagnostics” on page 3-90

Likelihood Ratio Test for Conditional Variance Models

This example shows how to estimate a conditional variance model using `estimate`. Fit two competing models to the data, and then compare their fit using a likelihood ratio test.

Step 1. Load the data and specify a GARCH model.

Load the Deutschmark/British pound foreign exchange rate data included with the toolbox, and convert it to returns. Specify a GARCH(1,1) model with a mean offset to estimate.

```
load Data_MarkPound
r = price2ret(Data);
T = length(r);
Mdl = garch('Offset',NaN,'GARCHLags',1,'ARCHLags',1);
```

Step 2. Estimate the GARCH model parameters.

Fit the specified GARCH(1,1) model to the returns series using `estimate`. Return the value of the loglikelihood objective function.

```
[EstMdl,~,logL] = estimate(Mdl,r);
```

```
    GARCH(1,1) Conditional Variance Model:
    ----------------------------------------
    Conditional Probability Distribution: Gaussian

                               Standard          t
     Parameter       Value       Error       Statistic
    -----------   -----------   ------------   -----------
     Constant    1.07568e-06   3.57247e-07        3.01104
     GARCH{1}       0.806079     0.0132728        60.7317
      ARCH{1}       0.153097     0.0115308        13.2772
       Offset     -6.134e-05   8.28711e-05      -0.740185
```

The estimation output shows the four estimated parameters and corresponding standard errors. The t statistic for the mean offset is not greater than two in magnitude, suggesting this parameter is not statistically significant.

Step 3. Fit a GARCH model without a mean offset.

Specify a second model without a mean offset, and fit it to the returns series.

```
Mdl2 = garch(1,1);
[EstMdl2,~,logL2] = estimate(Mdl2,r);
```

```
    GARCH(1,1) Conditional Variance Model:
    ----------------------------------------
    Conditional Probability Distribution: Gaussian

                                Standard          t
     Parameter       Value        Error       Statistic
    -----------   -----------   ------------   -----------
     Constant    1.05346e-06   3.50483e-07        3.00575
     GARCH{1}       0.806576     0.0129095        62.4794
      ARCH{1}       0.154357     0.0115746        13.3358
```

All the t statistics for the new fitted model are greater than two in magnitude.

Step 4. Conduct a likelihood ratio test.

Compare the fitted models `EstMdl` and `EstMdl2` using the likelihood ratio test. The number of restrictions for the test is one (only the mean offset was excluded in the second model).

```
[h,p] = lratiotest(logL,logL2,1)
```

```
h =

     0


p =

    0.4534
```

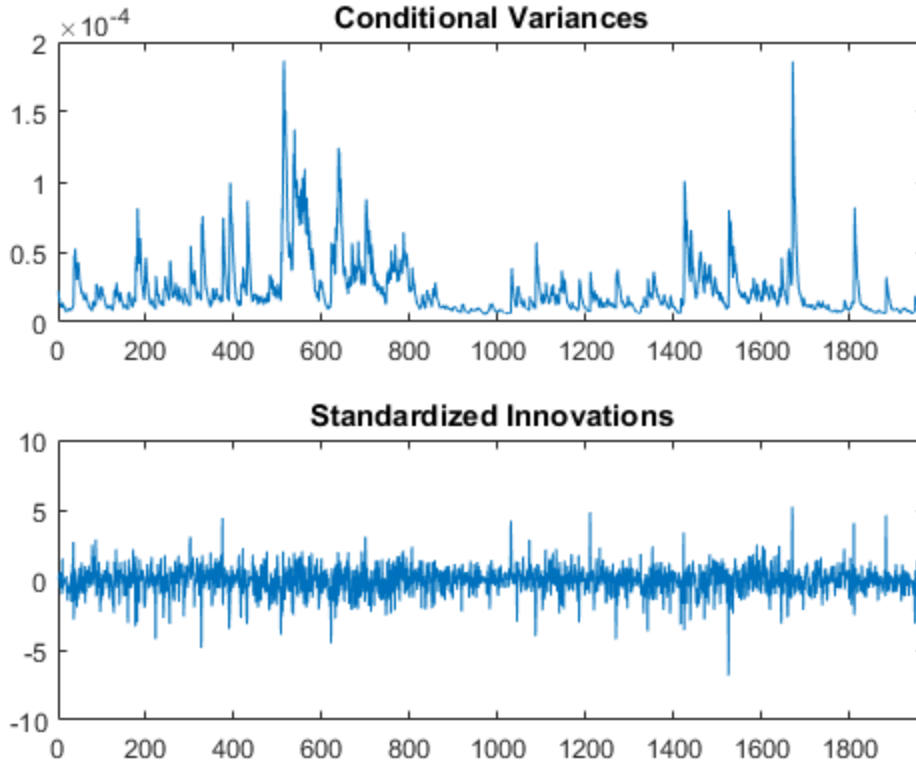
The null hypothesis of the restricted model is not rejected in favor of the larger model (`h = 0`). The model without a mean offset is the more parsimonious choice.

Step 5. Infer the conditional variances and standardized innovations.

Infer and plot the conditional variances and standardized innovations for the fitted model without a mean offset (`EstMdl2`).

```
v = infer(EstMdl2,r);
```

```
inn = r./sqrt(v);

figure
subplot(2,1,1)
plot(v)
xlim([0,T])
title('Conditional Variances')

subplot(2,1,2)
plot(inn)
xlim([0,T])
title('Standardized Innovations')
```



The inferred conditional variances show the periods of high volatility.

See Also

`estimate` | `garch` | `infer` | `lratiotest`

Related Examples

- "Specify Conditional Variance Model For Exchange Rates" on page 6-53
- "Simulate Conditional Variance Model" on page 6-111
- "Forecast a Conditional Variance Model" on page 6-126

More About

- Using garch Objects
- "Maximum Likelihood Estimation for Conditional Variance Models" on page 6-62
- "Model Comparison Tests" on page 3-65

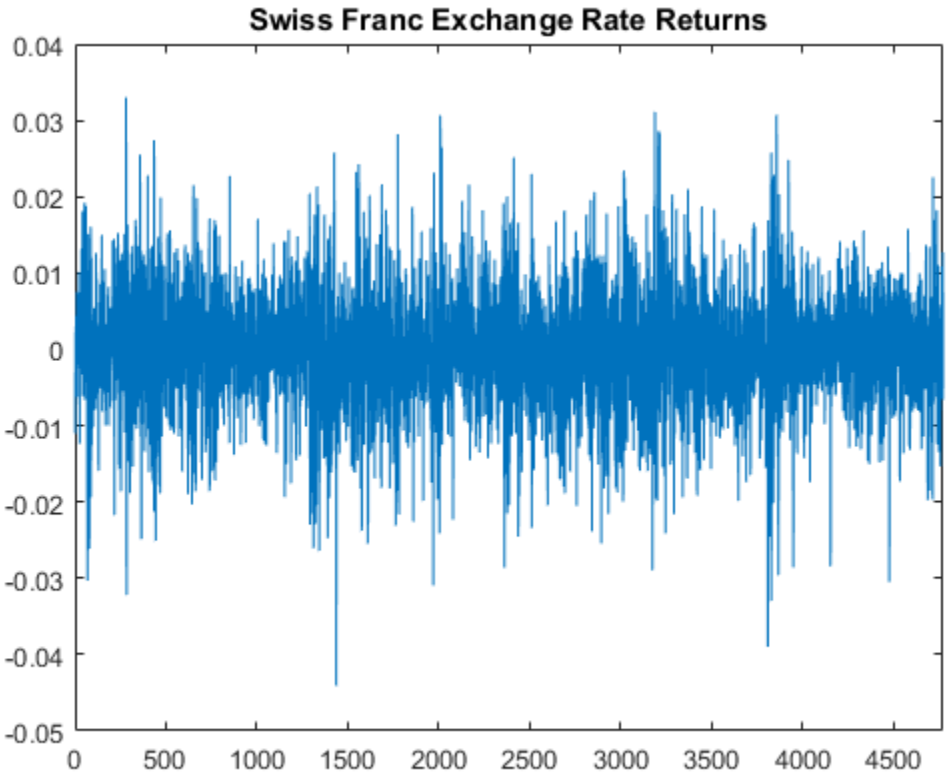
Compare Conditional Variance Models Using Information Criteria

This example shows how to specify and fit a GARCH, EGARCH, and GJR model to foreign exchange rate returns. Compare the fits using AIC and BIC.

Step 1. Load the data.

Load the foreign exchange rate data included with the toolbox. Convert the Swiss franc exchange rate to returns.

```
load Data_FXRates
y = DataTable.CHF;
r = price2ret(y);
T = length(r);

logL = zeros(1,3); % Preallocate
numParams = logL;  % Preallocate

figure
plot(r)
xlim([0,T])
title('Swiss Franc Exchange Rate Returns')
```



The returns series appears to exhibit some volatility clustering.

Step 2. Fit a GARCH(1,1) model.

Specify, and then fit a GARCH(1,1) model to the returns series. Return the value of the loglikelihood objective function.

```
Mdl1 = garch(1,1);
[EstMdl1,EstParamCov1,logL(1)] = estimate(Mdl1,r);
numParams(1) = sum(any(EstParamCov1)); % Number of fitted parameters
```

```
    GARCH(1,1) Conditional Variance Model:
    ----------------------------------------
```

```
    Conditional Probability Distribution: Gaussian

                               Standard          t
     Parameter       Value       Error       Statistic
    -----------   -----------   ------------   -----------
     Constant    1.62565e-06    4.3693e-07        3.72061
     GARCH{1}       0.913868    0.00687116        133.001
      ARCH{1}      0.0584481    0.00499475        11.7019
```

Step 3. Fit an EGARCH(1,1) model.

Specify, and then fit an EGARCH(1,1) model to the returns series. Return the value of the loglikelihood objective function.

```
Mdl2 = egarch(1,1);
[EstMdl2,EstParamCov2,logL(2)] = estimate(Mdl2,r);
numParams(2) = sum(any(EstParamCov2));
```

```
    EGARCH(1,1) Conditional Variance Model:
    --------------------------------------
    Conditional Probability Distribution: Gaussian

                               Standard          t
     Parameter       Value       Error       Statistic
    -----------   -----------   ------------   -----------
     Constant      -0.292496     0.0459408       -6.36681
     GARCH{1}       0.969758    0.00467846        207.281
      ARCH{1}       0.122915     0.0120521        10.1986
  Leverage{1}     -0.0132285    0.00494973       -2.67256
```

Step 4. Fit a GJR(1,1) model.

Specify, and then fit a GJR(1,1) model to the returns series. Return the value of the loglikelihood objective function.

```
Mdl3 = gjr(1,1);
[EstMdl3,EstParamCov3,logL(3)] = estimate(Mdl3,r);
numParams(3) = sum(any(EstParamCov3));
```

```
    GJR(1,1) Conditional Variance Model:
    -----------------------------------
    Conditional Probability Distribution: Gaussian
```

```
                           Standard          t
    Parameter       Value      Error       Statistic
    -----------   -----------   ------------   -----------
     Constant    1.70866e-06   4.50865e-07        3.78974
     GARCH{1}       0.911387    0.00722421        126.157
      ARCH{1}      0.0589006    0.00686708        8.57724
  Leverage{1}     0.00131757    0.00728035       0.180976
```

The leverage term in the GJR model is not statistically significant.

Step 5. Compare the model fits using AIC and BIC.

Calculate the AIC and BIC values for the GARCH, EGARCH, and GJR model fits. The GARCH model has three parameters; the EGARCH and GJR models each have four parameters.

```
[aic,bic] = aicbic(logL,numParams,T)
```

```
aic =

   1.0e+04 *

   -3.3329   -3.3321   -3.3327


bic =

   1.0e+04 *

   -3.3309   -3.3295   -3.3301
```

The GARCH(1,1) and EGARCH(1,1) models are not nested, so you cannot compare them by conducting a likelihood ratio test. The GARCH(1,1) is nested in the GJR(1,1) model, however, so you could use a likelihood ratio test to compare these models.

Using AIC and BIC, the GARCH(1,1) model has slightly smaller (more negative) AIC and BIC values. Thus, the GARCH(1,1) model is the preferred model according to these criteria.

See Also

`aicbic` | `estimate` | `garch`

Related Examples

- "Specify Conditional Variance Model For Exchange Rates" on page 6-53
- "Likelihood Ratio Test for Conditional Variance Models" on page 6-83

More About

- Using garch Objects
- Using egarch Objects
- Using gjr Objects
- "Maximum Likelihood Estimation for Conditional Variance Models" on page 6-62
- "Information Criteria" on page 3-63

Monte Carlo Simulation of Conditional Variance Models

In this section...

What Is Monte Carlo Simulation?

Monte Carlo simulation is the process of generating independent, random draws from a specified probabilistic model. When simulating time series models, one draw (or realization) is an entire sample path of specified length N, y_1, y_2,...,y_N. When you generate a large number of draws, say M, you generate M sample paths, each of length N.

Note: Some extensions of Monte Carlo simulation rely on generating dependent random draws, such as Markov Chain Monte Carlo (MCMC). The `simulate` function in Econometrics Toolbox generates independent realizations.

Some applications of Monte Carlo simulation are:

- Demonstrating theoretical results
- Forecasting future events
- Estimating the probability of future events

Generate Monte Carlo Sample Paths

Conditional variance models specify the dynamic evolution of the variance of a process over time. Perform Monte Carlo simulation of conditional variance models by:

1. Specifying any required presample data (or use default presample data).
2. Generating the next conditional variance recursively using the specified conditional variance model.
3. Simulating the next innovation from the innovation distribution (Gaussian or Student's t) using the current conditional variance.

For example, consider a GARCH(1,1) process without a mean offset, $\varepsilon_t = \sigma_t z_t$, where z_t either follows a standardized Gaussian or Student's t distribution and

$$\sigma_t^2 = \kappa + \gamma_1 \sigma_{t-1}^2 + \alpha_1 \varepsilon_{t-1}^2.$$

Suppose that the innovation distribution is Gaussian.

Given presample variance σ_0^2 and presample innovation ε_0, realizations of the conditional variance and innovation process are recursively generated:

- $\sigma_1^2 = \kappa + \gamma_1 \sigma_0^2 + \alpha_1 \varepsilon_0^2$
- Sample ε_1 from a Gaussian distribution with variance σ_1^2
- $\sigma_2^2 = \kappa + \gamma_1 \sigma_1^2 + \alpha_1 \varepsilon_1^2$
- Sample ε_2 from a Gaussian distribution with variance σ_2^2

⋮

- $\sigma_N^2 = \kappa + \gamma_1 \sigma_{N-1}^2 + \alpha_1 \varepsilon_{N-1}^2$
- Sample ε_N from a Gaussian distribution with variance σ_N^2

Random draws are generated from EGARCH and GJR models similarly, using the corresponding conditional variance equations.

Monte Carlo Error

Using many simulated paths, you can estimate various features of the model. However, Monte Carlo estimation is based on a finite number of simulations. Therefore, Monte Carlo estimates are subject to some amount of error. You can reduce the amount of Monte Carlo error in your simulation study by increasing the number of sample paths, *M*, that you generate from your model.

For example, to estimate the probability of a future event:

1. Generate *M* sample paths from your model.

2 Estimate the probability of the future event using the sample proportion of the event occurrence across M simulations,

$$\hat{p} = \frac{\#\ times\ event\ occurs\ in\ M\ draws}{M}.$$

3 Calculate the Monte Carlo standard error for the estimate,

$$se = \sqrt{\frac{\hat{p}(1-\hat{p})}{M}}.$$

You can reduce the Monte Carlo error of the probability estimate by increasing the number of realizations. If you know the desired precision of your estimate, you can solve for the number of realizations needed to achieve that level of precision.

See Also

`simulate`

Related Examples

- "Simulate Conditional Variance Model" on page 6-111
- "Simulate GARCH Models" on page 6-97
- "Assess EGARCH Forecast Bias Using Simulations" on page 6-104

More About

- Using garch Objects
- Using egarch Objects
- Using gjr Objects
- "Presample Data for Conditional Variance Model Simulation" on page 6-95
- "Monte Carlo Forecasting of Conditional Variance Models" on page 6-115

Presample Data for Conditional Variance Model Simulation

When simulating realizations from GARCH, EGARCH, or GJR processes, you need presample conditional variances and presample innovations to initialize the variance equation.

For the GARCH(P,Q) and GJR(P,Q) models, P presample variances and Q presample innovations are needed. For an EGARCH(P,Q) model, max(P,Q) presample variances and Q presample innovations are needed.

You can either specify your own presample data, or let `simulate` automatically generate presample data.

If you let `simulate` generate presample data, then:

- Presample variances are set to the theoretical unconditional variance of the model being simulated.
- Presample innovations are random draws from the innovation distribution with the theoretical unconditional variance.

If you are specifying your own presample data, note that `simulate` assumes presample innovations with mean zero. If your observed series is an innovation series plus an offset, subtract the offset from the observed series before using it as a presample innovation series.

When you specify adequate presample variances and innovations, the first conditional variance in the simulation recursion is the same for all sample paths. The first simulated innovations are random, however, because they are random draws from the innovation distribution (with common variance, specified by the presample variances and innovations).

See Also

`simulate`

Related Examples

- “Simulate Conditional Variance Model” on page 6-111
- “Simulate GARCH Models” on page 6-97
- “Assess EGARCH Forecast Bias Using Simulations” on page 6-104

More About

- Using garch Objects
- Using egarch Objects
- Using gjr Objects
- “Monte Carlo Simulation of Conditional Variance Models” on page 6-92
- “Monte Carlo Forecasting of Conditional Variance Models” on page 6-115

Simulate GARCH Models

This example shows how to simulate from a GARCH process with and without specifying presample data. The sample unconditional variances of the Monte Carlo simulations approximate the theoretical GARCH unconditional variance.

Step 1. Specify a GARCH model.

Specify a GARCH(1,1) model $\varepsilon_t = \sigma_t z_t$, where the distribution of z_t is Gaussian and

$$\sigma_t^2 = 0.01 + 0.7\sigma_{t-1}^2 + 0.25\varepsilon_{t-1}^2.$$

```
Mdl = garch('Constant',0.01,'GARCH',0.7,'ARCH',0.25)
```

```
Mdl =

    GARCH(1,1) Conditional Variance Model:
    --------------------------------------
    Distribution: Name = 'Gaussian'
               P: 1
               Q: 1
        Constant: 0.01
           GARCH: {0.7} at Lags [1]
            ARCH: {0.25} at Lags [1]
```

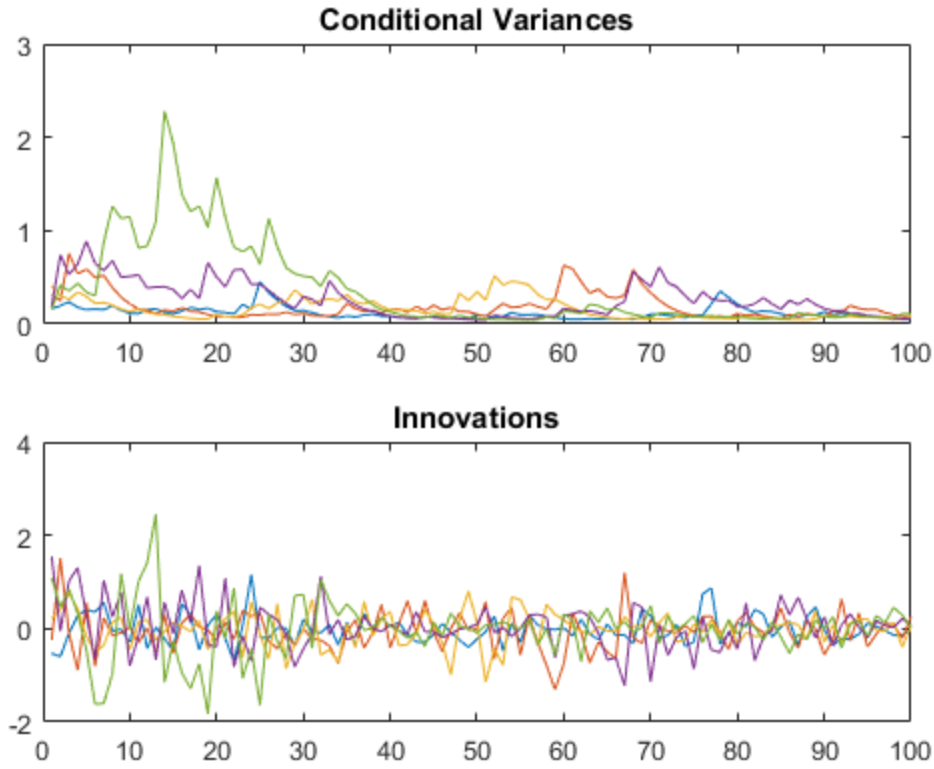
Step 2. Simulate from the model without using presample data.

Simulate five paths of length 100 from the GARCH(1,1) model, without specifying any presample innovations or conditional variances. Display the first conditional variance for each of the five sample paths. The model being simulated does not have a mean offset, so the response series is an innovation series.

```
rng default; % For reproducibility
[Vn,Yn] = simulate(Mdl,100,'NumPaths',5);

Vn(1,:) % Display variances

figure
subplot(2,1,1)
plot(Vn)
xlim([0,100])
title('Conditional Variances')
```

```
subplot(2,1,2)
plot(Yn)
xlim([0,100])
title('Innovations')
```

```
ans =

    0.1645    0.3182    0.4051    0.1872    0.1551
```



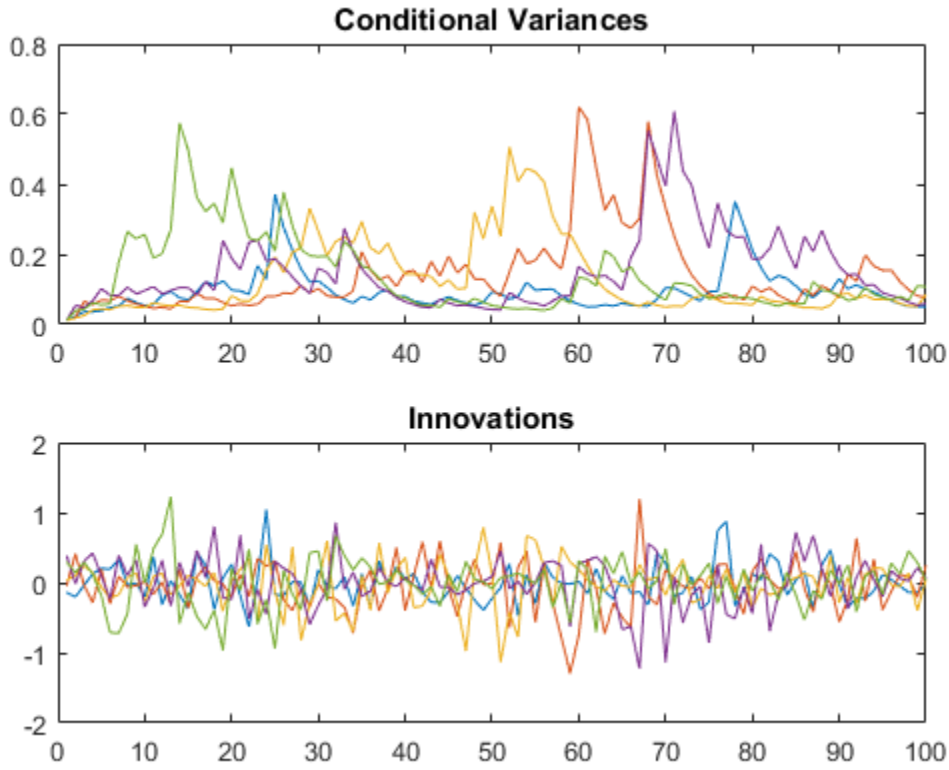
The starting conditional variances are different for each realization because no presample data was specified.

Step 3. Simulate from the model using presample data.

Simulate five paths of length 100 from the model, specifying the one required presample innovation and conditional variance. Display the first conditional variance for each of the five sample paths.

```
rng default;
[Vw,Yw] = simulate(Mdl,100,'NumPaths',5,...
    'E0',0.05,'V0',0.001);

Vw(1,:)

figure
subplot(2,1,1)
plot(Vw)
xlim([0,100])
title('Conditional Variances')

subplot(2,1,2)
plot(Yw)
xlim([0,100])
title('Innovations')
```

```
ans =

    0.0113    0.0113    0.0113    0.0113    0.0113
```



All five sample paths have the same starting conditional variance, calculated using the presample data.

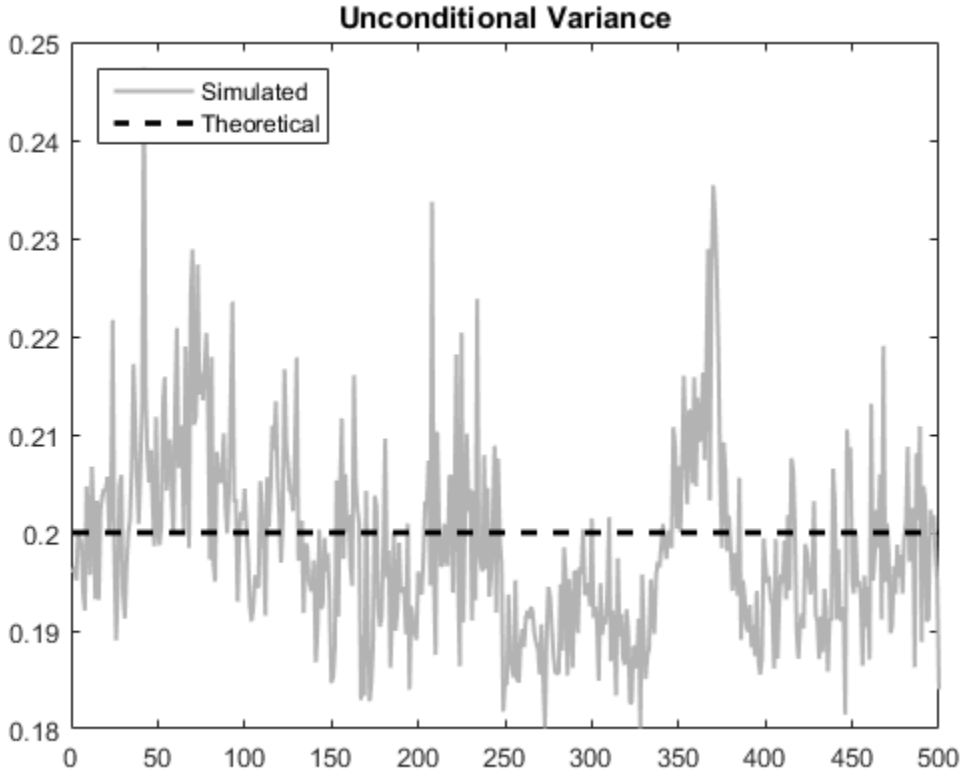
Note that even with the same starting variance, the realizations of the innovation series have different starting points. This is because each ε_1 is a random draw from a Gaussian distribution with mean 0 and variance $\sigma_1 = 0.0113$.

Step 4. Look at the unconditional variances.

Simulate 10,000 sample paths of length 500 from the specified GARCH model. Plot the sample unconditional variances of the Monte Carlo simulations, and compare them to the theoretical unconditional variance,

$$\sigma_\varepsilon^2 = \frac{\kappa}{(1-\gamma_1-\alpha_1)} = \frac{0.01}{(1-0.7-0.25)} = 0.2.$$

```
sig2 = 0.01/(1-0.7-0.25);

rng default;
[V,Y] = simulate(Mdl,500,'NumPaths',10000);

figure
plot(var(Y,0,2),'Color',[.7,.7,.7],'LineWidth',1.5)
xlim([0,500])
hold on
plot(1:500,ones(500,1)*sig2,'k--','LineWidth',2)
legend('Simulated','Theoretical','Location','NorthWest')
title('Unconditional Variance')
hold off
```



The simulated unconditional variances fluctuate around the theoretical unconditional variance.

See Also

`garch` | `simulate`

Related Examples

- "Specify GARCH Models Using garch" on page 6-8
- "Assess EGARCH Forecast Bias Using Simulations" on page 6-104

More About

- Using garch Objects
- "Monte Carlo Simulation of Conditional Variance Models" on page 6-92
- "Presample Data for Conditional Variance Model Simulation" on page 6-95

Assess EGARCH Forecast Bias Using Simulations

This example shows how to simulate an EGARCH process. Simulation-based forecasts are compared to minimum mean square error (MMSE) forecasts, showing the bias in MMSE forecasting of EGARCH processes.

Step 1. Specify an EGARCH model.

Specify an EGARCH(1,1) process with constant $\kappa = 0.01$, GARCH coefficient $\gamma_1 = 0.7$, ARCH coefficient $\alpha_1 = 0.3$ and leverage coefficient $\xi_1 = -0.1$.

```
Mdl = egarch('Constant',0.01,'GARCH',0.7,...
    'ARCH',0.3,'Leverage',-0.1)
```

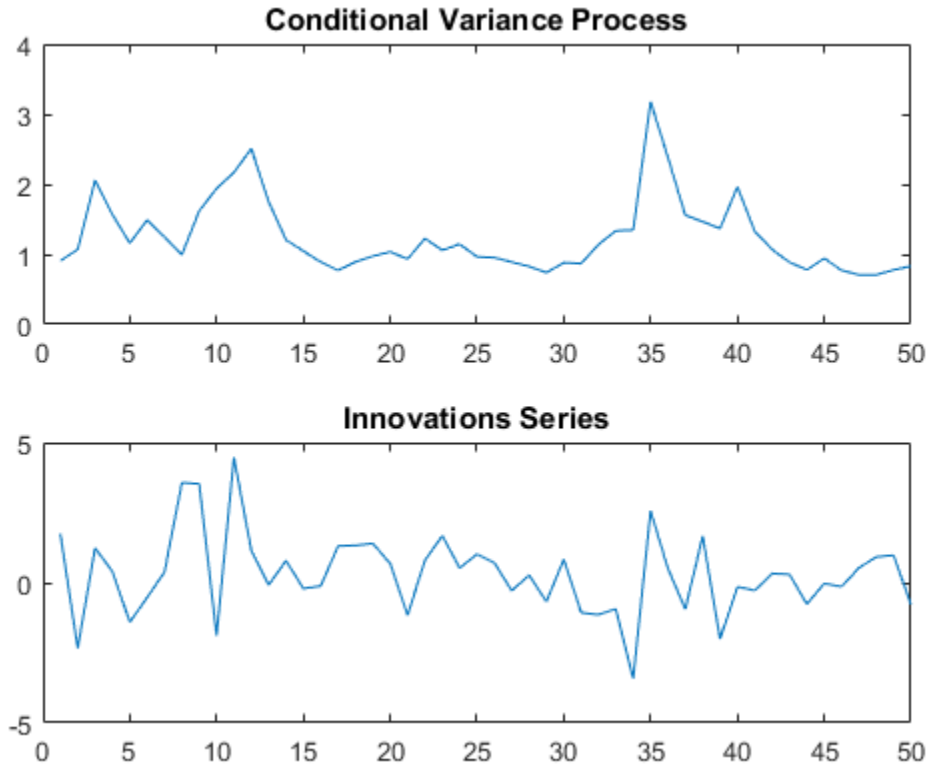
```
Mdl =

    EGARCH(1,1) Conditional Variance Model:
    ----------------------------------------
    Distribution: Name = 'Gaussian'
               P: 1
               Q: 1
        Constant: 0.01
           GARCH: {0.7} at Lags [1]
            ARCH: {0.3} at Lags [1]
        Leverage: {-0.1} at Lags [1]
```

Step 2. Simulate one realization.

Simulate one realization of length 50 from the EGARCH conditional variance process and corresponding innovation series.

```
rng default; % For reproducibility

[v,y] = simulate(Mdl,50);

figure
subplot(2,1,1)
plot(v)
xlim([0,50])
title('Conditional Variance Process')

subplot(2,1,2)
```

```
plot(y)
xlim([0,50])
title('Innovations Series')
```

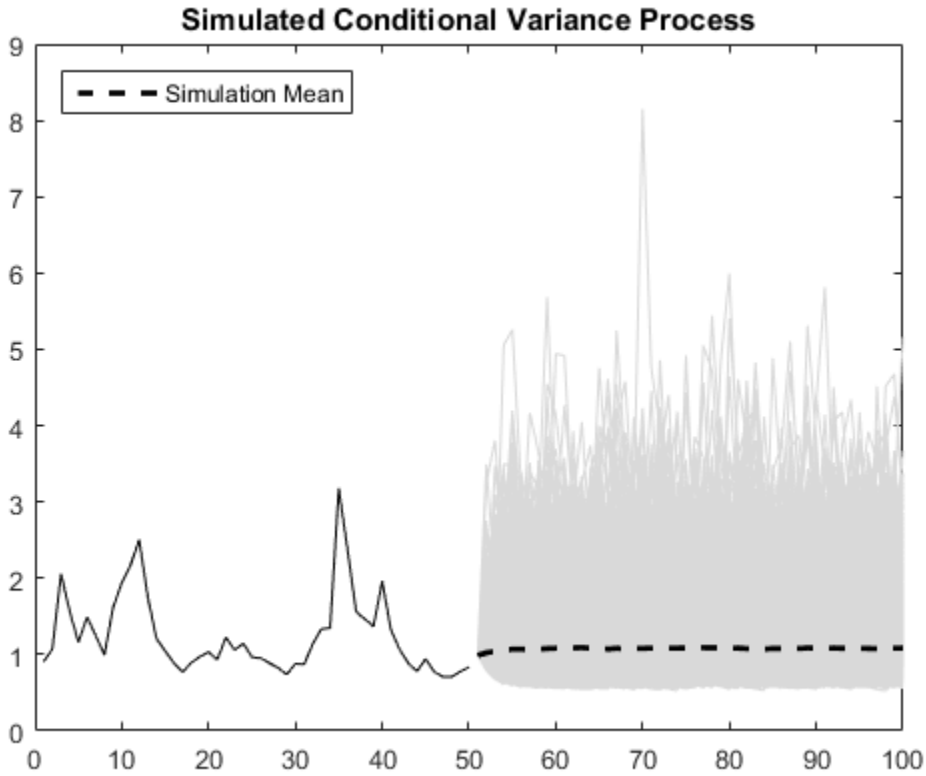


Step 3. Simulate multiple realizations.

Using the conditional variance and innovation realizations generated in Step 2 as presample data, simulate 5000 realizations of the EGARCH process for 50 future time steps. Plot the simulation mean of the forecasted conditional variance process.

```
rng default; % For reproducibility
[Vsim,Ysim] = simulate(Mdl,50,'NumPaths',5000,...
                       'E0',y,'V0',v);
```

```
figure
plot(v,'k')
hold on
plot(51:100,Vsim,'Color',[.85,.85,.85])
xlim([0,100])
h = plot(51:100,mean(Vsim,2),'k--','LineWidth',2);
title('Simulated Conditional Variance Process')
legend(h,'Simulation Mean','Location','NorthWest')
hold off
```



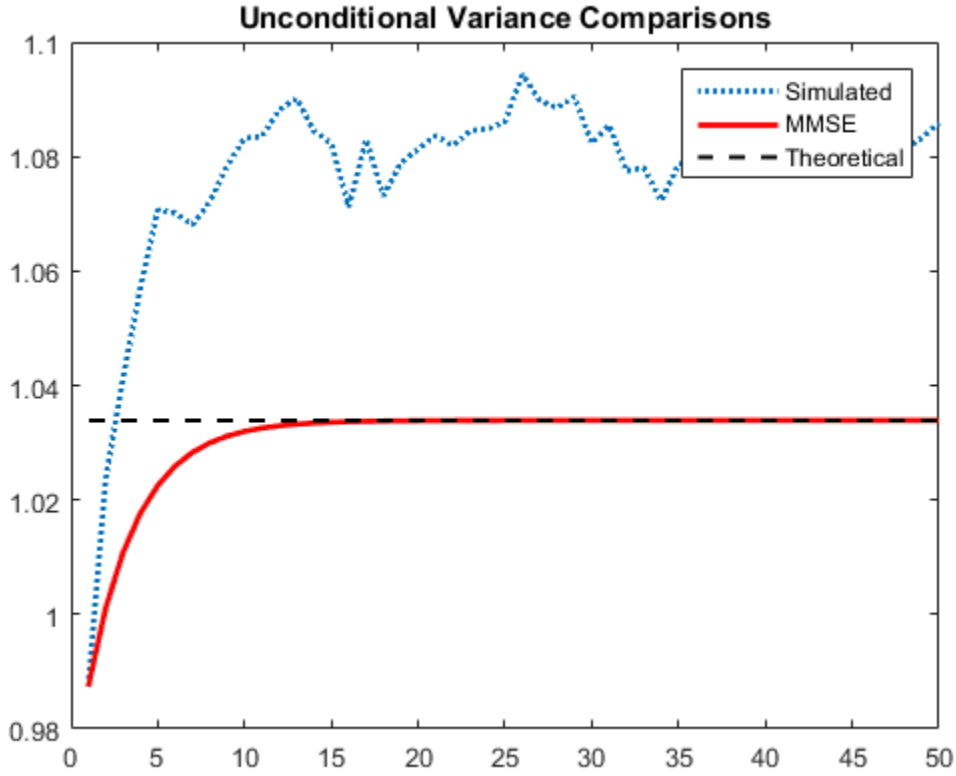
Step 4: Compare simulated and MMSE conditional variance forecasts.

Compare the simulation mean variance, the MMSE variance forecast, and the exponentiated, theoretical unconditional log variance.

The exponentiated, theoretical unconditional log variance for the specified EGARCH(1,1) model is

$$\sigma_{\varepsilon}^2 = \exp\left\{\frac{\kappa}{(1-\gamma_1)}\right\} = \exp\left\{\frac{0.01}{(1-0.7)}\right\} = 1.0339.$$

```
sim = mean(Vsim,2);
fcast = forecast(Mdl,50,'Y0',y,'V0',v);
sig2 = exp(0.01/(1-0.7));

figure
plot(sim,':','LineWidth',2)
hold on
plot(fcast,'r','LineWidth',2)
plot(ones(50,1)*sig2,'k--','LineWidth',1.5)
legend('Simulated','MMSE','Theoretical')
title('Unconditional Variance Comparisons')
hold off
```



The MMSE and exponentiated, theoretical log variance are biased relative to the unconditional variance (by about 4%) because by Jensen's inequality,

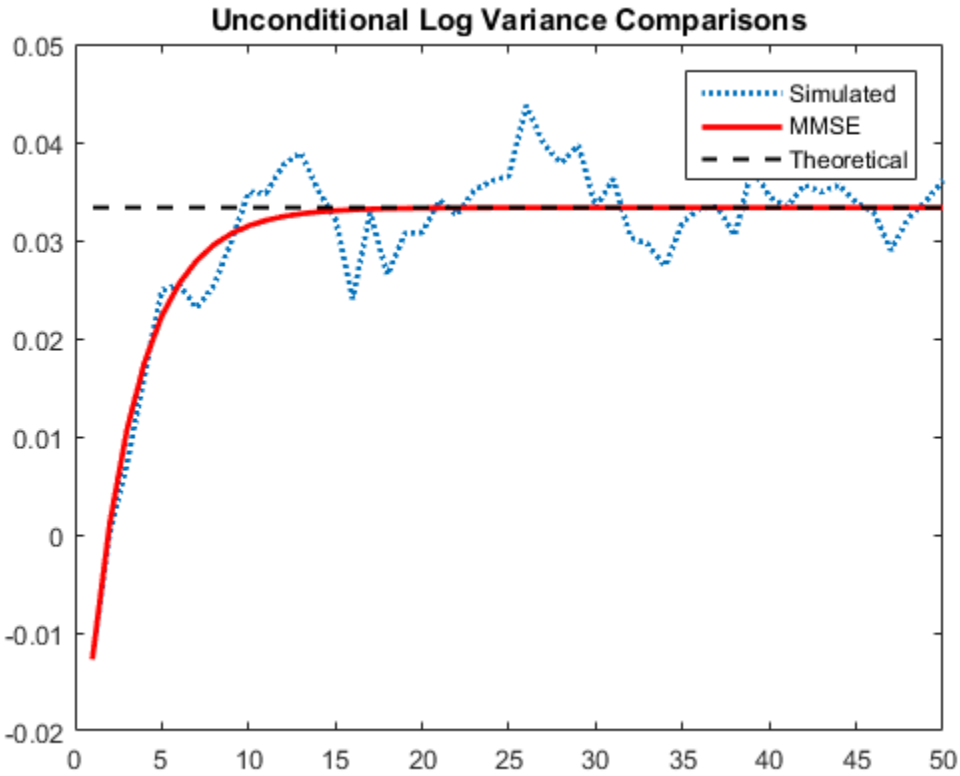
$$E(\sigma_t^2) \geq \exp\left\{E(\log \sigma_t^2)\right\}.$$

Step 5. Compare simulated and MMSE log conditional variance forecasts.

Compare the simulation mean log variance, the log MMSE variance forecast, and the theoretical, unconditional log variance.

```
logsim = mean(log(Vsim),2);
logsig2 = 0.01/(1-0.7);
```

```
figure
plot(logsim,':','LineWidth',2)
hold on
plot(log(fcast),'r','LineWidth',2)
plot(ones(50,1)*logsig2,'k--','LineWidth',1.5)
legend('Simulated','MMSE','Theoretical')
title('Unconditional Log Variance Comparisons')
hold off
```



The MMSE forecast of the unconditional log variance is unbiased.

See Also

`egarch` | `forecast` | `simulate`

Related Examples

- "Specify EGARCH Models Using egarch" on page 6-19

More About

- Using egarch Objects
- "EGARCH Model" on page 6-4
- "Monte Carlo Simulation of Conditional Variance Models" on page 6-92
- "Monte Carlo Forecasting of Conditional Variance Models" on page 6-115
- "MMSE Forecasting of Conditional Variance Models" on page 6-117

Simulate Conditional Variance Model

This example shows how to simulate a conditional variance model using `simulate`.

Step 1. Load the data and specify the model.

Load the Deutschmark/British pound foreign exchange rate data included with the toolbox, and convert to returns. Specify and fit a GARCH(1,1) model.

```
load Data_MarkPound
r = price2ret(Data);
T = length(r);
Mdl = garch(1,1);
EstMdl = estimate(Mdl,r);
v0 = infer(EstMdl,r);
```

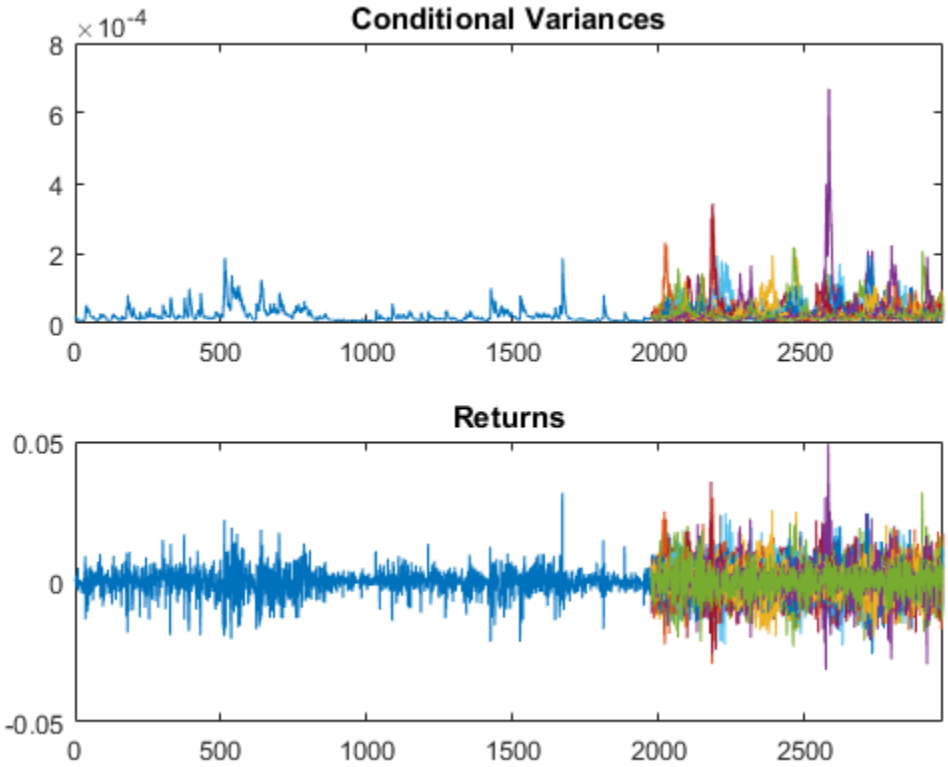
```
    GARCH(1,1) Conditional Variance Model:
    ----------------------------------------
    Conditional Probability Distribution: Gaussian

                               Standard          t
     Parameter       Value       Error       Statistic
    -----------   -----------   ------------   -----------
     Constant    1.05346e-06   3.50483e-07        3.00575
     GARCH{1}       0.806576     0.0129095        62.4794
      ARCH{1}       0.154357     0.0115746        13.3358
```

Step 2. Simulate foreign exchange rate returns.

Use the fitted model to simulate 25 realizations of foreign exchange rate returns and conditional variances over a 1000-period forecast horizon. Use the observed returns and inferred conditional variances as presample innovations and variances, respectively.

```
rng default; % For reproducibility
[V,Y] = simulate(EstMdl,1000,'NumPaths',25,...
    'E0',r,'V0',v0);

figure
subplot(2,1,1)
plot(v0)
hold on
plot(T+1:T+1000,V)
xlim([0,T+1000])
title('Conditional Variances')
```

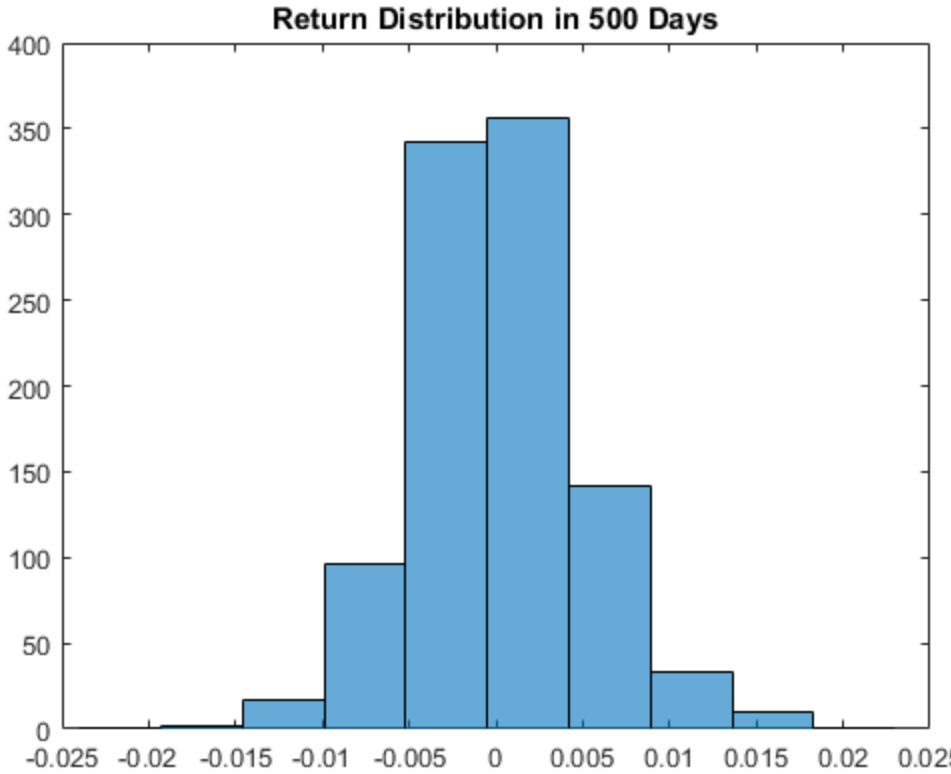
```
hold off

subplot(2,1,2)
plot(r)
hold on
plot(T+1:T+1000,Y)
xlim([0,T+1000])
title('Returns')
hold off
```



Step 3. Plot the returns distribution at a future time.

Use simulations to generate a forecast distribution of foreign exchange returns 500 days into the future. Generate 1000 sample paths to estimate the distribution.

```
rng default; % For reproducibility
[V,Y] = simulate(EstMdl,500,'NumPaths',1000,...
    'E0',r-EstMdl.Offset,'V0',v0);

figure
histogram(Y(500,:),10)
title('Return Distribution in 500 Days')
```



See Also

estimate | forecast | garch | simulate

Related Examples

- “Specify Conditional Variance Model For Exchange Rates” on page 6-53
- “Likelihood Ratio Test for Conditional Variance Models” on page 6-83
- “Forecast a Conditional Variance Model” on page 6-126

More About

- Using garch Objects
- “Monte Carlo Simulation of Conditional Variance Models” on page 6-92
- “Presample Data for Conditional Variance Model Simulation” on page 6-95
- “Monte Carlo Forecasting of Conditional Variance Models” on page 6-115

Monte Carlo Forecasting of Conditional Variance Models

In this section...

Monte Carlo Forecasts

You can use Monte Carlo simulation to forecast a process over a future time horizon. This is an alternative to minimum mean square error (MMSE) forecasting, which provides an analytical forecast solution. You can calculate MMSE forecasts using `forecast`.

To forecast a process using Monte Carlo simulations:

- Fit a model to your observed series using `estimate`.
- Use the observed series and any inferred residuals and conditional variances (calculated using `infer`) for presample data.
- Generate many sample paths over the desired forecast horizon using `simulate`.

Advantage of Monte Carlo Forecasting

An advantage of Monte Carlo forecasting is that you obtain a complete *distribution* for future events, not just a point estimate and standard error. The simulation mean approximates the MMSE forecast. Use the 2.5th and 97.5th percentiles of the simulation realizations as endpoints for approximate 95% forecast intervals.

See Also

`estimate` | `forecast` | `simulate`

Related Examples

- "Simulate Conditional Variance Model" on page 6-111
- "Assess EGARCH Forecast Bias Using Simulations" on page 6-104
- "Forecast a Conditional Variance Model" on page 6-126

More About

- Using garch Objects

- Using egarch Objects
- Using gjr Objects
- “Monte Carlo Simulation of Conditional Variance Models” on page 6-92
- “MMSE Forecasting of Conditional Variance Models” on page 6-117

MMSE Forecasting of Conditional Variance Models

In this section...

What Are MMSE Forecasts?

A common objective of conditional variance modeling is generating forecasts for the conditional variance process over a future time horizon. That is, given the conditional variance process $\sigma_1^2,\sigma_2^2,\ldots,\sigma_N^2$ and a forecast horizon h, generate predictions for $\sigma_{N+1}^2,\sigma_{N+2}^2,\ldots,\sigma_{N+h}^2$.

Let $\hat{\sigma}_{t+1}^2$ denote a forecast for the variance at time $t + 1$, conditional on the history of the process up to time t, H_t. The minimum mean square error (MMSE) forecast is the forecast $\hat{\sigma}_{t+1}^2$ that minimizes the conditional expected square loss,

$$E(\sigma_{t+1}^2 - \sigma_{t+1}^2 \mid H_t).$$

Minimizing this loss function yields the MMSE forecast,

$$\hat{\sigma}_{t+1}^2 = E(\sigma_{t+1}^2 \mid H_t) = E(\varepsilon_{t+1}^2 \mid H_t).$$

EGARCH MMSE Forecasts

For the EGARCH model, the MMSE forecast is found for the log conditional variance,

$$\log \sigma_{t+1}^2 = E(\log \sigma_{t+1}^2 \mid H_t).$$

For conditional variance forecasts of EGARCH processes, `forecast` returns the exponentiated MMSE log conditional variance forecast,

$$\hat{\sigma}^2_{t+1} = \exp\{\log \hat{\sigma}^2_{t+1}\}.$$

This results in a slight forecast bias because of Jensen's inequality,

$$E(\sigma^2_{t+1}) \geq \exp\{E(\log \sigma^2_{t+1})\}.$$

As an alternative to MMSE forecasting, you can conduct Monte Carlo simulations to forecast EGARCH processes. Monte Carlo simulations yield unbiased forecasts for EGARCH models. However, Monte Carlo forecasts are subject to Monte Carlo error (which you can reduce by increasing the simulation sample size).

How forecast Generates MMSE Forecasts

The `forecast` function generates MMSE forecasts recursively. When you call `forecast`, you can specify presample responses (`Y0`) and presample conditional variances (`V0`) using name-value arguments. If the model being forecasted includes a mean offset—signaled by a nonzero `Offset` property—`forecast` subtracts the offset term from the presample responses to create presample innovations.

To begin forecasting from the end of an observed series, say `Y`, use the last few observations of `Y` as presample responses `Y0` to initialize the forecast. The minimum number of presample responses needed to initialize forecasting is stored in the property `Q` of a model.

When specifying presample conditional variances `V0`, the minimum number of presample conditional variances needed to initialize forecasting is stored in the property `P` for GARCH(P,Q) and GJR(P,Q) models. For EGARCH(P,Q) models, the minimum number of presample conditional variances needed to initialize forecasting is max(P,Q).

Note that for all variance models, if you supply at least max(P,Q) + P presample response observations (`Y0`), `forecast` infers any needed presample conditional variances (`V0`) for you. If you supply presample observations, but less than max(P,Q) + P, `forecast` sets any needed presample conditional variances equal to the unconditional variance of the model.

If you do not provide any presample innovations, then for GARCH and GJR models, `forecast` sets any necessary presample innovations equal to the unconditional standard deviation of the model. For EGARCH models, `forecast` sets the presample innovations equal to zero.

GARCH Model

The `forecast` function generates MMSE forecasts for GARCH models recursively.

Consider generating forecasts for a GARCH(1,1) model, $\varepsilon_t = \sigma_t z_t$, where

$$\sigma_t^2 = \kappa + \gamma_1 \sigma_{t-1}^2 + \alpha_1 \varepsilon_{t-1}^2.$$

Given presample innovation ε_T and presample conditional variance σ_T^2, forecasts are recursively generated as follows:

- $\hat{\sigma}_{T+1}^2 = \kappa + \gamma_1 \sigma_T^2 + \alpha_1 \varepsilon_T^2$
- $\hat{\sigma}_{T+2}^2 = \kappa + \gamma_1 \hat{\sigma}_{T+1}^2 + \alpha_1 \hat{\sigma}_{T+1}^2$
- $\hat{\sigma}_{T+3}^2 = \kappa + \gamma_1 \hat{\sigma}_{T+2}^2 + \alpha_1 \hat{\sigma}_{T+2}^2$

⋮

Note that innovations are forecasted using the identity

$$E(\varepsilon_{t+1}^2 \mid H_t) = E(\sigma_{t+1}^2 \mid H_t) = \sigma_{t+1}^2.$$

This recursion converges to the unconditional variance of the process,

$$\sigma_\varepsilon^2 = \frac{\kappa}{(1 - \gamma_1 - \alpha_1)}.$$

GJR Model

The `forecast` function generates MMSE forecasts for GJR models recursively.

Consider generating forecasts for a GJR(1,1) model, $\varepsilon_t = \sigma_t z_t$, where $\sigma_t^2 = \kappa + \gamma_1 \sigma_{t-1}^2 + \alpha_1 \varepsilon_{t-1}^2 + \xi_1 I[\varepsilon_{t-1} < 0] \varepsilon_{t-1}^2$. Given presample innovation ε_T and presample conditional variance σ_T^2, forecasts are recursively generated as follows:

- $\hat{\sigma}^2_{T+1} = \kappa + \gamma_1\hat{\sigma}^2_T + \alpha_1\varepsilon^2_T + \xi_1 I[\varepsilon_T < 0]\varepsilon^2_T$
- $\hat{\sigma}^2_{T+2} = \kappa + \gamma_1\hat{\sigma}^2_{T+1} + \alpha_1\hat{\sigma}^2_{T+1} + \frac{1}{2}\xi_1\hat{\sigma}^2_{T+1}$
- $\hat{\sigma}^2_{T+3} = \kappa + \gamma_1\hat{\sigma}^2_{T+2} + \alpha_1\hat{\sigma}^2_{T+2} + \frac{1}{2}\xi_1\hat{\sigma}^2_{T+2}$

⋮

Note that the expected value of the indicator is 1/2 for an innovation process with mean zero, and that innovations are forecasted using the identity

$$E(\varepsilon^2_{t+1} \mid H_t) = E(\sigma^2_{t+1} \mid H_t) = \sigma^2_{t+1}.$$

This recursion converges to the unconditional variance of the process,

$$\sigma^2_\varepsilon = \frac{\kappa}{\left(1 - \gamma_1 - \alpha_1 - \frac{1}{2}\xi_1\right)}.$$

EGARCH Model

The `forecast` function generates MMSE forecasts for EGARCH models recursively. The forecasts are initially generated for the log conditional variances, and then exponentiated to forecast the conditional variances. This results in a slight forecast bias.

Consider generating forecasts for an EGARCH(1,1) model, $\varepsilon_t = \sigma_t z_t$, where

$$\log\sigma^2_t = \kappa + \gamma_1 \log\sigma^2_{t-1} + \alpha_1\left[\left|\frac{\varepsilon_{t-1}}{\sigma_{t-1}}\right| - E\left\{\left|\frac{\varepsilon_{t-1}}{\sigma_{t-1}}\right|\right\}\right] + \xi_1\frac{\varepsilon_{t-1}}{\sigma_{t-1}}.$$

The form of the expected value term depends on the choice of innovation distribution, Gaussian or Student's *t*. Given presample innovation ε_T and presample conditional variance σ^2_T, forecasts are recursively generated as follows:

- $$\log\sigma_{T+1}^2 = \kappa + \gamma_1 \log\sigma_T^2 + \alpha_1\left[\left|\frac{\varepsilon_T}{\sigma_T}\right| - E\left\{\left|\frac{\varepsilon_T}{\sigma_T}\right|\right\}\right] + \xi_1\frac{\varepsilon_T}{\sigma_T}$$
- $$\log\sigma_{T+2}^2 = \kappa + \gamma_1 \log\sigma_{T+1}^2$$
- $$\log\sigma_{T+3}^2 = \kappa + \gamma_1 \log\sigma_{T+2}^2$$

⋮

Notice that future absolute standardized innovations and future innovations are each replaced by their expected value. This means that both the ARCH and leverage terms are zero for all forecasts that are conditional on future innovations. This recursion converges to the unconditional log variance of the process,

$$\log\sigma_\varepsilon^2 = \frac{\kappa}{(1-\gamma_1)}.$$

`forecast` returns the exponentiated forecasts, $\exp\{\log\sigma_{T+1}^2\}, \exp\{\log\sigma_{T+2}^2\}, \ldots,$ which have limit

$$\exp\left\{\frac{\kappa}{(1-\gamma_1)}\right\}.$$

See Also

`forecast`

Related Examples

- “Assess EGARCH Forecast Bias Using Simulations” on page 6-104
- “Forecast a Conditional Variance Model” on page 6-126
- “Forecast GJR Models” on page 6-123

More About

- Using garch Objects

- Using egarch Objects
- Using gjr Objects
- “Monte Carlo Forecasting of Conditional Variance Models” on page 6-115

Forecast GJR Models

This example shows how to generate MMSE forecasts of a GJR model using `forecast`.

Step 1. Specify a GJR model.

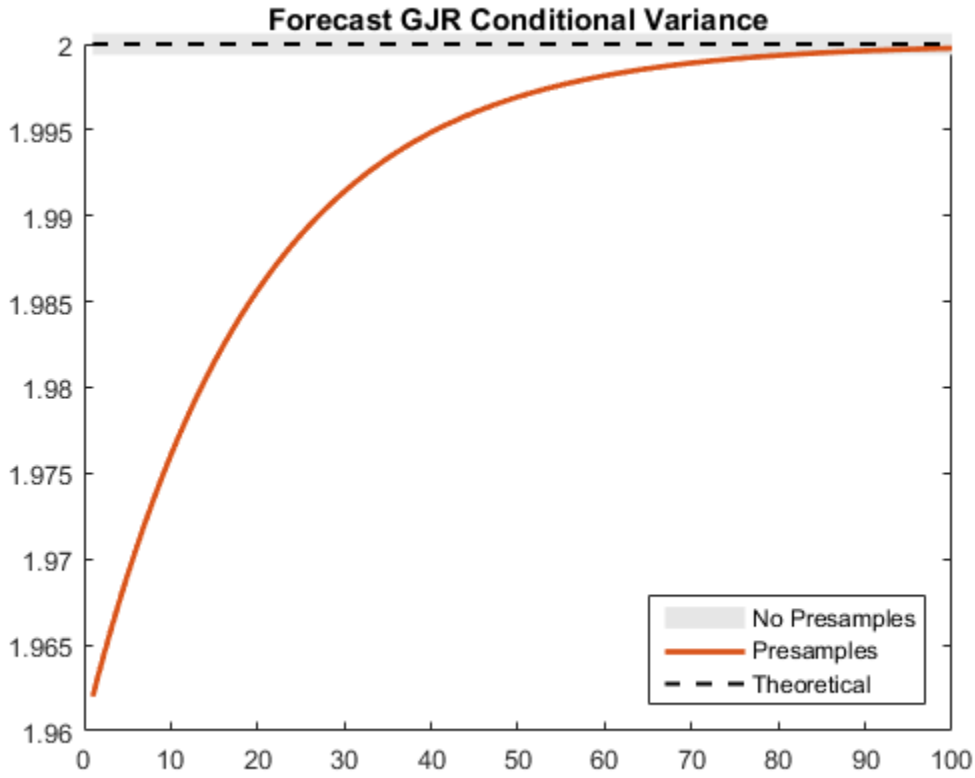
Specify a GJR(1,1) model without a mean offset and $\kappa = 0.1$, $\gamma_1 = 0.7$, $\alpha_1 = 0.2$ and $\xi_1 = 0.1$.

```
Mdl = gjr('Constant',0.1,'GARCH',0.7,...
    'ARCH',0.2,'Leverage',0.1);
```

Step 2. Generate MMSE forecasts.

Generate forecasts for a 100-period horizon with and without specifying a presample innovation and conditional variance. Plot the forecasts along with the theoretical unconditional variance of the model.

```
v1 = forecast(Mdl,100);
v2 = forecast(Mdl,100,'Y0',1.4,'V0',2.1);
denom = 1-Mdl.GARCH{1}-Mdl.ARCH{1}-0.5*Mdl.Leverage{1};
sig2 = Mdl.Constant/denom;

figure
plot(v1,'Color',[.9,.9,.9],'LineWidth',8)
hold on
plot(v2,'LineWidth',2)
plot(ones(100,1)*sig2,'k--','LineWidth',1.5)
xlim([0,100])
title('Forecast GJR Conditional Variance')
legend('No Presamples','Presamples','Theoretical',...
        'Location','SouthEast')
hold off

v2(1) % Display forecasted conditional variance


ans =

    1.9620
```



The forecasts generated without using presample data are equal to the theoretical unconditional variance. In the absence of presample data, `forecast` uses the unconditional variance for any required presample innovations and conditional variances.

In this example, for the given presample innovation and conditional variance, the starting forecast is

$$\hat{\sigma}^2_{t+1} = \kappa + \gamma_1\sigma^2_t + \alpha_1\varepsilon^2_t = 0.1 + 0.7(2.1) + 0.2(1.4^2) = 1.962.$$

The leverage term is not included in the forecast since the presample innovation was positive (thus, the negative-innovation indicator is zero).

See Also

`forecast` | `gjr`

Related Examples

- "Specify GJR Models Using gjr" on page 6-31
- "Forecast a Conditional Variance Model" on page 6-126

More About

- Using gjr Objects
- "GJR Model" on page 6-6
- "MMSE Forecasting of Conditional Variance Models" on page 6-117

Forecast a Conditional Variance Model

This example shows how to forecast a conditional variance model using `forecast`.

Load the data and specify the model.

Load the Deutschmark/British pound foreign exchange rate data included with the toolbox, and convert to returns. For numerical stability, convert returns to percentage returns.

```
load Data_MarkPound
r  = price2ret(Data);
pR = 100*r;
T  = length(r);
```

Specify and fit a GARCH(1,1) model.

```
Mdl = garch(1,1);
EstMdl = estimate(Mdl,pR);
```

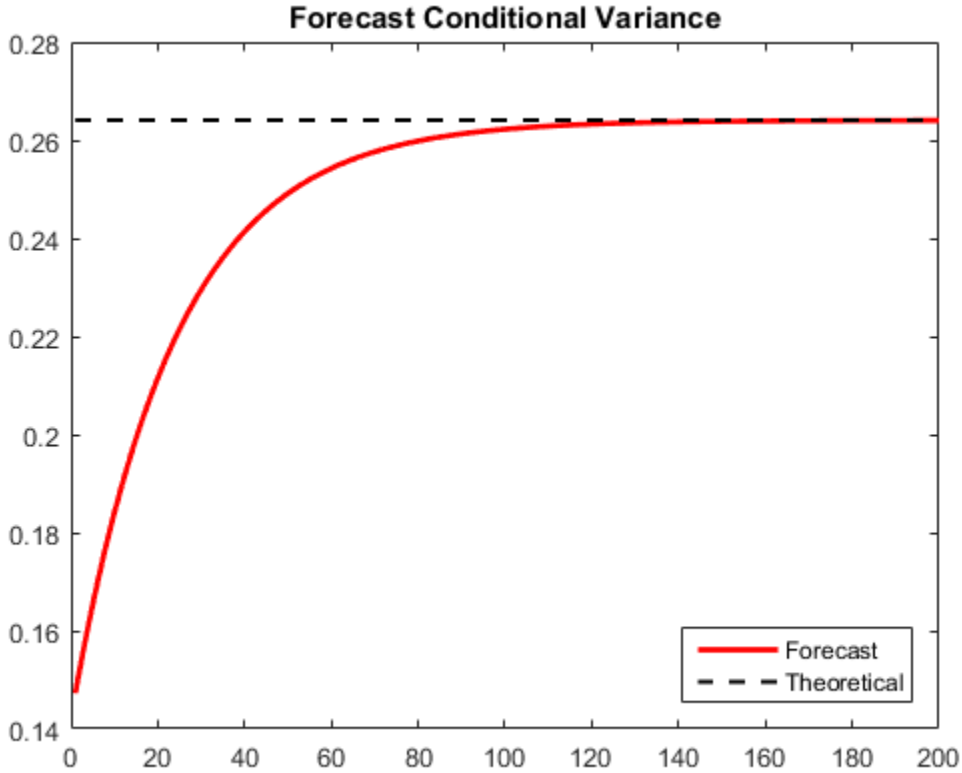
```
    GARCH(1,1) Conditional Variance Model:
    ----------------------------------------
    Conditional Probability Distribution: Gaussian

                               Standard          t
     Parameter       Value       Error       Statistic
    -----------   -----------   ------------   -----------
     Constant       0.010868    0.00129723        8.37786
     GARCH{1}       0.804518     0.0160384        50.1619
      ARCH{1}       0.154325     0.0138523        11.1408
```

Generate MMSE forecasts.

Use the fitted model to generate MMSE forecasts over a 200-period horizon. Use the observed return series as presample data. By default, `forecast` infers the corresponding presample conditional variances. Compare the asymptote of the variance forecast to the theoretical unconditional variance of the GARCH(1,1) model.

```
v = forecast(EstMdl,200,'Y0',pR);
sig2 = EstMdl.Constant/(1-EstMdl.GARCH{1}-EstMdl.ARCH{1});

figure
```

```
plot(v,'r','LineWidth',2)
hold on
plot(ones(200,1)*sig2,'k--','LineWidth',1.5)
xlim([0,200])
title('Forecast Conditional Variance')
legend('Forecast','Theoretical','Location','SouthEast')
hold off
```



The MMSE forecasts converge to the theoretical unconditional variance after about 160 steps.

See Also

`estimate` | `forecast` | `garch`

Related Examples

- "Specify Conditional Variance Model For Exchange Rates" on page 6-53
- "Likelihood Ratio Test for Conditional Variance Models" on page 6-83
- "Simulate Conditional Variance Model" on page 6-111

More About

- Using garch Objects
- "MMSE Forecasting of Conditional Variance Models" on page 6-117

Converting from GARCH Functions to Model Objects

In R2014a, `arima`, `garch`, `egarch`, and `gjr` models and associated functionality replace the `garchfit`, `garchinfer`, `garchplot`, `garchpred`, `garchsim`, `garchcount`, `garchdisp`, `garchget`, and `garchset` functions. If you use the older GARCH data analysis functions, then you will receive an error. Use the information on this page to help you convert from the older to the newer functionality.

Suppose that you want to analyze a univariate series `y` using an ARIMA(3,4) model with GARCH(1,2) innovations, and you have presample observations (`y0`), innovations (`e0`), and conditional standard deviations (`sigma0`). This table shows both ways to complete such an analysis. For examples, see Related Examples.

Step	Old Functionality	New Functionality

Specify a compound ARIMA-GARCH model	`Mdl = garchset('R',3,'M',4,...`
	`'P',1,'Q',2);`
		`VarMdl = garch(1,2);`
		`Mdl.Variance = VarMdl;`
		`Mdl = arima(3,0,4);`

Retrieve model properties	`garchget(Mdl,'K')`	`Mdl.Variance.Constant`
Set equality constraints	`Mdl = garchset(Mdl,...` `'K',0.75,'FixK',1);`	`Mdl.Variance.Constant = 0.75;`

Estimate parameters	`EstMdl = garchfit(Mdl,y,[],...`
	`e0,sigma0,y0);`
		`[EstMdl,EstParamCov] = ...`
		`estimate(Mdl,y,'E0',e0,...`
		`'V0',sigma0.^2,'Y0',y0)`

Count the number of fitted parameters	`numParams = garchcount(...` `EstMdl);`	`numParams = sum(any(...` `EstParamCov));`

Infer conditional variances (`sigma2`) and obtain the loglikelihood (`logL`)	`[e,sigma,logL] = ...`
	`garchinfer(EstMdl,y,...`
	`[],e0,sigma0,y0);`
	`sigma2 = sigma.^2;`
		`[e,sigma2,logL] = infer(...`
		`EstMdl,y,'E0',e0,...`
		`'V0',sigma0.^2,'Y0',y0);`

Simulate observations	`simY = garchsim(EstMdl,100);`	`simY = simulate(EstMdl,100);`

Filter disturbances	`e = randn(100,1);`
	`simY = garchsim(EstMdl,[],...`
	`[],e);`
		`e = randn(100,1);`
		`simY = filter(EstMdl,e);`

Forecast observations	`foreY = garchpred(EstMdl,y,...` `15);`	`foreY = forecast(EstMdl,15,...` `'Y0',y);`

Though similar, the input and output structure of the two functionalities differ in some ways. This example shows how to determine some of the differences between the two, and might help you through the conversion. This example does not show how to reproduce equivalent results between the models, because, in general, the estimates between the two functionalities might differ.

Suppose that you want to analyze a univariate series. You suspect that the model is either an ARIMA(2,1)/GARCH(1,1) or ARIMA(1,1)/GARCH(1,1) model, and want to test which model fits to the data better. Variables representing the new functionality have the suffix `1` (e.g., `Mdl1`), and those of the older functionality have suffix `2` (e.g., `Mdl2`).

1. Simulate the data from an ARIMA(2,1) model with GARCH(1,1) innovations.

```
% New way
VarMdl1 = garch('GARCH',0.3,'ARCH',0.2,'Constant',0.75);
Mdl1 = arima('AR',{0.5,0.3},'MA',-0.7,'Constant',0,'Variance',VarMdl1);
[y1,e1,v1] = simulate(Mdl1,100);

% Old way
Mdl2 = garchset('AR',[0.5,0.3],'MA',-0.7,'C',0,...
     'GARCH',0.3,'ARCH',0.2,'K',0.75);
[e2,sd2,y2] = garchsim(Mdl2,100);
```

   The differences are:

   - `Mdl1` is an `object` data type, and `Mdl2` is a `struct` data type.
   - `simulate` returns conditional variances, whereas `garchsim` returns conditional standard deviations.
   - With the new functionality, you must:
     - Specify multiple coefficient values using a cell array.
     - Specify the variance model using a `garch`, `egarch`, or `gjr` model.

2. Specify template models to use for estimation.

```
% New way
ToEstVarMdl1 = garch(1,1);
ToEstMdl11 = arima('ARLags',1,'MALags',1,'Variance',ToEstVarMdl1);
ToEstMdl21 = arima('ARLags',1:2,'MALags',1,'Variance',ToEstVarMdl1);

% Old way
ToEstMdl12 = garchset('R',1,'M',1,'P',1,'Q',1,'Display','off');
ToEstMdl22 = garchset('R',2,'M',1,'P',1,'Q',1,'Display','off');
```

The new functionality has the name-value pair arguments `'ARLags'` and `'MALags'` to set the polynomial terms of the model. You must specify each term order individually, which allows for a more flexible specification. The models `ToEstMdl1` and `ToEstMdl2` have properties `P` and `Q` corresponding to the autoregressive and moving average orders of the model.

3 Fit each model to the data.

```
% New way
logL1 = [0;0];       % Preallocate
numParams1 = logL1;  % Preallocate
[EstMdl11,EstParamCov11,logl11] = estimate(ToEstMdl11,...
    y1,'Display','off');
[EstMdl21,EstParamCov21,logl21] = estimate(ToEstMdl21,...
    y1,'Display','off');

% Old way
logL2 = [0;0];       % Preallocate
numParams2 = logL2;  % Preallocate
[EstMdl12,~,logl12] = garchfit(ToEstMdl12,y2);
[EstMdl22,~,logl22] = garchfit(ToEstMdl22,y2);
```

The `estimate` function:

- Returns the estimated parameter covariance matrix rather than just the standard errors.
- Lets you decide to see estimates and optimization information, rather than setting it when you specify the model.

4 Count the number of fitted parameters in the estimated model.

```
% New way
numParams11 = sum(any(EstParamCov11));
numParams21 = sum(any(EstParamCov21));

% Old way
numParams12 = garchcount(EstMdl12);
numParams22 = garchcount(EstMdl22);
```

The new functionality does not contain a function that counts the number of fitted parameters in an estimated model. However, if a parameter is fixed during estimation, then the software sets all variances and covariances of the corresponding parameter estimate to `0`. The new way to count fitted parameters uses this feature.

5 Assess which model is more appropriate using information criteria.

```
aic1 = aicbic(logL1,numParams1);
aic2 = aicbic(logL2,numParams2);
```

See Also

arima | `egarch` | `estimate` | `filter` | `forecast` | `garch` | `gjr` | `infer` | `print` | `simulate`

Related Examples

- "Specify Conditional Mean Models Using arima" on page 5-6
- "Specify Conditional Variance Model For Exchange Rates" on page 6-53
- "Specify Conditional Mean and Variance Models" on page 5-79
- "Modify Properties of Conditional Mean Model Objects" on page 5-65
- "Modify Properties of Conditional Variance Models" on page 6-42
- "Modify Properties of Conditional Mean Model Objects" on page 5-65
- "Modify Properties of Conditional Variance Models" on page 6-42
- "Estimate Conditional Mean and Variance Models" on page 5-129
- "Infer Residuals for Diagnostic Checking" on page 5-140
- "Infer Conditional Variances and Residuals" on page 6-77
- "Simulate GARCH Models" on page 6-97
- "Forecast a Conditional Variance Model" on page 6-126

More About

- Using garch Objects
- Using egarch Objects
- Using gjr Objects

Multivariate Time Series Models

Vector Autoregressive (VAR) Models

In this section...

Types of VAR Models

The multivariate time series models used in Econometrics Toolbox functions are based on linear, autoregressive models. The basic models are:

Model Name	Abbreviation	Equation
Vector Autoregressive	VAR(p)	$\boldsymbol{y}_t = a + \sum_{i=1}^{p} A_i \boldsymbol{y}_{t-i} + \varepsilon_t$
Vector Moving Average	VMA(q)	$\boldsymbol{y}_t = a + \sum_{j=1}^{q} B_j \varepsilon_{t-j} + \varepsilon_t$
Vector Autoregressive Moving Average	VARMA(p, q)	$\boldsymbol{y}_t = a + \sum_{i=1}^{p} A_i \boldsymbol{y}_{t-i} + \sum_{j=1}^{q} B_j \varepsilon_{t-j} + \varepsilon_t$
Vector Autoregressive Moving Average with eXogenous inputs	VARMAX(p, q, r)	$\boldsymbol{y}_t = a + X_t \cdot b + \sum_{i=1}^{p} A_i \boldsymbol{y}_{t-i} + \sum_{j=1}^{q} B_j \varepsilon_{t-j} + \varepsilon_t$
Structural Vector Autoregressive Moving Average with eXogenous inputs	SVARMAX(p, q, r)	$A_0 \boldsymbol{y}_t = a + X_t \cdot b + \sum_{i=1}^{p} A_i \boldsymbol{y}_{t-i} + \sum_{j=1}^{q} B_j \varepsilon_{t-j} + B_0 \varepsilon_t$

The following variables appear in the equations:

- y_t is the vector of *response* time series variables at time t. y_t has n elements.

- a is a constant vector of offsets, with n elements.
- A_i are n-by-n matrices for each i. The A_i are autoregressive matrices. There are p autoregressive matrices.
- ε_t is a vector of serially uncorrelated innovations, vectors of length n. The ε_t are multivariate normal random vectors with a covariance matrix Q, where Q is an identity matrix, unless otherwise specified.
- B_j are n-by-n matrices for each j. The B_j are moving average matrices. There are q moving average matrices.
- X_t is an n-by-r matrix representing exogenous terms at each time t. r is the number of exogenous series. Exogenous terms are data (or other unmodeled inputs) in addition to the response time series y_t.
- b is a constant vector of regression coefficients of size r. So the product $X_t \cdot b$ is a vector of size n.

Generally, the time series y_t and X_t are observable. In other words, if you have data, it represents one or both of these series. You do not always know the offset a, coefficient b, autoregressive matrices A_i, and moving average matrices B_j. You typically want to fit these parameters to your data. See the `vgxvarx` function reference page for ways to estimate unknown parameters. The innovations ε_t are not observable, at least in data, though they can be observable in simulations.

Lag Operator Representation

There is an equivalent representation of the linear autoregressive equations in terms of lag operators. The lag operator L moves the time index back by one: $Ly_t = y_{t-1}$. The operator L^m moves the time index back by m: $L^m y_t = y_{t-m}$.

In lag operator form, the equation for a SVARMAX(p, q, r) model becomes

$$\left(A_0 - \sum_{i=1}^{p} A_i L^i\right) y_t = a + X_t b + \left(B_0 + \sum_{j=1}^{q} B_j L^j\right) \varepsilon_t.$$

This equation can be written as

$$A(L) y_t = a + X_t b + B(L) \varepsilon_t,$$

where

$$A(L) = A_0 - \sum_{i=1}^{p} A_i L^i \text{ and } B(L) = B_0 + \sum_{j=1}^{q} B_j L^j.$$

Stable and Invertible Models

A VAR model is *stable* if

$$\det\left(I_n - A_1 z - A_2 z^2 - \ldots - A_p z^p \right) \neq 0 \text{ for } |z| \leq 1,$$

This condition implies that, with all innovations equal to zero, the VAR process converges to a as time goes on. See Lütkepohl [74] Chapter 2 for a discussion.

A VMA model is *invertible* if

$$\det\left(I_n + B_1 z + B_2 z^2 + \ldots + B_q z^q \right) \neq 0 \text{ for } |z| \leq 1.$$

This condition implies that the pure VAR representation of the process is stable. For an explanation of how to convert between VAR and VMA models, see "Changing Model Representations" on page 7-23. See Lütkepohl [74] Chapter 11 for a discussion of invertible VMA models.

A VARMA model is stable if its VAR part is stable. Similarly, a VARMA model is invertible if its VMA part is invertible.

There is no well-defined notion of stability or invertibility for models with exogenous inputs (e.g., VARMAX models). An exogenous input can destabilize a model.

Building VAR Models

To understand a multiple time series model, or multiple time series data, you generally perform the following steps:

1. Import and preprocess data.
2. Specify a model.
   a. "Specifying Models" on page 7-14 to set up a model using `vgxset`:

- "Specification Structures with Known Parameters" on page 7-15 to specify a model with known parameters
- "Specification Structures with No Parameter Values" on page 7-16 to specify a model when you want MATLAB to estimate the parameters
- "Specification Structures with Selected Parameter Values" on page 7-17 to specify a model where you know some parameters, and want MATLAB to estimate the others

b "Determining an Appropriate Number of Lags" on page 7-19 to determine an appropriate number of lags for your model

3 Fit the model to data.

a "Fitting Models to Data" on page 7-24 to use `vgxvarx` to estimate the unknown parameters in your models. This can involve:

- "Changing Model Representations" on page 7-23 to change your model to a type that `vgxvarx` handles
- Estimating "Structural Matrices" on page 7-22

4 Analyze and forecast using the fitted model. This can involve:

a "Examining the Stability of a Fitted Model" on page 7-25 to determine whether your model is stable and invertible.

b "VAR Model Forecasting" on page 7-39 to forecast directly from models or to forecast using a Monte Carlo simulation.

c "Calculating Impulse Responses" on page 7-40 to calculate impulse responses, which give forecasts based on an assumed change in an input to a time series.

d Compare the results of your model's forecasts to data held out for forecasting. For an example, see "VAR Model Case Study" on page 7-89.

Your application need not involve all of the steps in this workflow. For example, you might not have any data, but want to simulate a parameterized model. In that case, you would perform only steps 2 and 4 of the generic workflow.

You might iterate through some of these steps.

See Also

`vgxget` | `vgxpred` | `vgxset` | `vgxsim` | `vgxvarx`

Related Examples

- "Fit a VAR Model" on page 7-33
- "Implement Seemingly Unrelated Regression Analyses" on page 7-64
- "Estimate the Capital Asset Pricing Model Using SUR" on page 7-74
- "Forecast a VAR Model" on page 7-50
- "Forecast a VAR Model Using Monte Carlo Simulation" on page 7-53
- "Simulate Responses of Estimated VARX Model" on page 7-80
- "VAR Model Case Study" on page 7-89

More About

- "Multivariate Time Series Data Structures" on page 7-8
- "Multivariate Time Series Model Creation" on page 7-14
- "VAR Model Estimation" on page 7-22
- "VAR Model Forecasting, Simulation, and Analysis" on page 7-39
- "Multivariate Time Series Models with Regression Terms" on page 7-57
- "Cointegration and Error Correction Analysis" on page 7-108

Multivariate Time Series Data Structures

In this section...

Multivariate Time Series Data

Often, the first step in creating a multiple time series model is to obtain data. There are two types of multiple time series data:

- **Response data**. Response data corresponds to y_t in the multiple time series models defined in "Types of VAR Models" on page 7-3.
- **Exogenous data**. Exogenous data corresponds to X_t in the multiple time series models defined in "Types of VAR Models" on page 7-3. For details and examples on structuring preparing exogenous data for multivariate model analysis, see "Multivariate Time Series Models with Regression Terms" on page 7-57.

Before using Econometrics Toolbox functions with the data, put the data into the required form. Use standard MATLAB commands, or preprocess the data with a spreadsheet program, database program, PERL, or other utilities.

There are several freely available sources of data sets, such as the St. Louis Federal Reserve Economics Database (known as FRED): http://research.stlouisfed.org/fred2/. If you have a license, you can use Datafeed Toolbox™ functions to access data from various sources.

Response Data Structure

Response data for multiple time series models must be in the form of a matrix. Each row of the matrix represents one time, and each column of the matrix represents one time series. The earliest data is the first row, the latest data is the last row. The data represents y_t in the notation of "Types of VAR Models" on page 7-3. If there are T times and n time series, put the data in the form of a T-by-n matrix:

$$\begin{bmatrix} Y1_1 & Y2_1 & \cdots & Yn_1 \\ Y1_2 & Y2_2 & \cdots & Yn_2 \\ \vdots & \vdots & \ddots & \vdots \\ Y1_T & Y2_T & \cdots & Yn_T \end{bmatrix}$$

$Y1_t$ represents time series 1,..., Yn_t represents time series n, $1 \le t \le T$.

Multiple Paths

Response time series data can have an extra dimension corresponding to separate, independent paths. For this type of data, use a three-dimensional array `Y(t,j,p)`, where:

- `t` is the time index of an observation, $1 \le$ `t` $\le$ `T`.
- `j` is the index of a time series, $1 \le$ `j` $\le$ `n`.
- `p` is the path index, $1 \le$ `p` $\le$ `P`.

For any path `p`, `Y(t,j,p)` is a time series.

Example: Response Data Structure

The file `Data_USEconModel` ships with Econometrics Toolbox software. It contains time series from the St. Louis Federal Reserve Economics Database (known as FRED).

Enter

```
load Data_USEconModel
```

to load the data into your MATLAB workspace. The following items load into the workspace:

- `Data`, a 249-by-14 matrix containing the 14 time series,
- `DataTable`, a 249-by-14 tabular array that packages the data,
- `dates`, a 249-element vector containing the dates for `Data`,
- `Description`, a character array containing a description of the data series and the key to the labels for each series,
- `series`, a 1-by-14 cell array of labels for the time series.

Examine the data structures:

```
firstPeriod = dates(1)
lastPeriod = dates(end)
```

```
firstPeriod =

      711217


lastPeriod =

      733863
```

- `dates` is a vector containing MATLAB serial date numbers, the number of days since the putative date January 1, 0000. (This "date" is not a real date, but is convenient for making date calculations; for more information, see "Date Formats" in the Financial Toolbox™ User's Guide.)
- The `Data` matrix contains 14 columns. These represent the time series labeled by the cell vector of strings `series`.

FRED Series	Description
COE	Paid compensation of employees in $ billions
CPIAUCSL	Consumer Price Index
FEDFUNDS	Effective federal funds rate
GCE	Government consumption expenditures and investment in $ billions
GDP	Gross Domestic Product
GDPDEF	Gross domestic product in $ billions
GDPI	Gross private domestic investment in $ billions
GS10	Ten-year treasury bond yield
HOANBS	Non-farm business sector index of hours worked
M1SL	M1 money supply (narrow money)
M2SL	M2 money supply (broad money)
PCEC	Personal consumption expenditures in $ billions
TB3MS	Three-month treasury bill yield
UNRATE	Unemployment rate

`DataTable` is a tabular array containing the same data as in `Data`, but you can call variables using the tabular array and the name of the variable. Use dot notation to access a variable, for example, `DataTable.UNRATE` calls the unemployment rate time series.

Data Preprocessing

Your data might have characteristics that violate assumptions for linear multiple time series models. For example, you can have data with exponential growth, or data from multiple sources at different periodicities. You must preprocess your data to convert it into an acceptable form for analysis.

- For time series with exponential growth, you can preprocess the data by taking the logarithm of the growing series. In some cases you then difference the result. For an example, see "VAR Model Case Study" on page 7-89.
- For data from multiple sources, you must decide how best to fill in missing values. Commonly, you take the missing values as unchanged from the previous value, or as interpolated from neighboring values.

Note: If you take a difference of a series, the series becomes 1 shorter. If you take a difference of only some time series in a data set, truncate the other series so that all have the same length, or pad the differenced series with initial values.

Testing Data for Stationarity

You can test each time series data column for stability using unit root tests. For details, see "Unit Root Nonstationarity" on page 3-34.

Partitioning Response Data

To fit a lagged model to data, partition the response data in up to three sections:

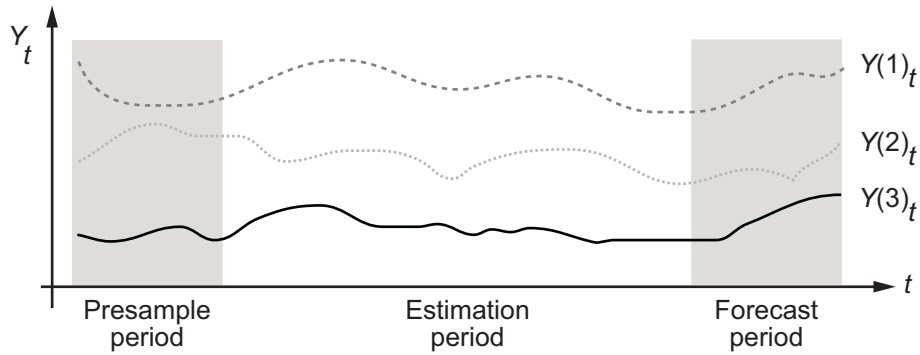
- Presample data
- Estimation data
- Forecast data

The purpose of presample data is to provide initial values for lagged variables. When trying to fit a model to the estimation data, you need to access earlier times. For

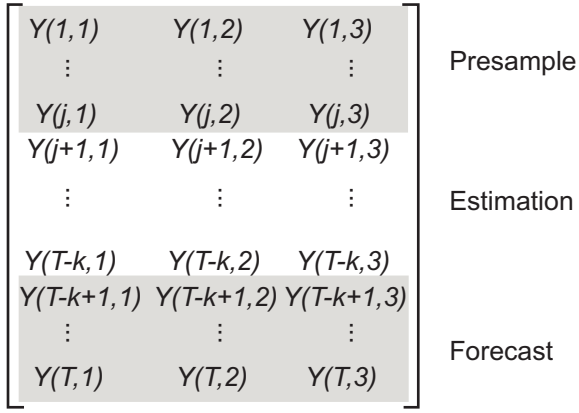
example, if the maximum lag in a model is 4, and if the earliest time in the estimation data is 50, then the model needs to access data at time 46 when fitting the observations at time 50. `vgxvarx` assumes the value 0 for any data that is not part of the presample data.

The estimation data contains the observations y_t. Use the estimation data to fit the model.

Use the forecast data for comparing fitted model predictions against data. You do not have to have a forecast period. Use one to validate the predictive power of a fitted model.



The following figure shows how to arrange the data in the data matrix, with j presample rows and k forecast rows.



See Also

`vgxget` | `vgxset` | `vgxvarx`

Related Examples

- "Fit a VAR Model" on page 7-33
- "Forecast a VAR Model" on page 7-50
- "Forecast a VAR Model Using Monte Carlo Simulation" on page 7-53
- "VAR Model Case Study" on page 7-89

More About

- "Vector Autoregressive (VAR) Models" on page 7-3
- "Multivariate Time Series Model Creation" on page 7-14
- "VAR Model Estimation" on page 7-22
- "VAR Model Forecasting, Simulation, and Analysis" on page 7-39

Multivariate Time Series Model Creation

In this section...

Models for Multiple Time Series

Econometrics Toolbox functions require a model specification structure as an input before they simulate, calibrate, forecast, or perform other calculations. You can specify a model with or without time series data. If you have data, you can fit models to the data as described in "VAR Model Estimation" on page 7-22. If you do not have data, you can specify a model with parameters you provide, as described in "Specification Structures with Known Parameters" on page 7-15.

Specifying Models

Create an Econometrics Toolbox multiple time series model specification structure using the `vgxset` function. Use this structure for calibrating, simulating, predicting, analyzing, and displaying multiple time series.

There are three ways to create model structures using the `vgxset` function:

- "Specification Structures with Known Parameters" on page 7-15. Use this method when you know the values of all relevant parameters of your model.
- "Specification Structures with No Parameter Values" on page 7-16. Use this method when you know the size, type, and number of lags in your model, but do not know the values of any of the AR or MA coefficients, or the value of your Q matrix.
- "Specification Structures with Selected Parameter Values" on page 7-17. Use this method when you know the size, type, and number of lags in your model, and also know some, but not all, of the values of AR or MA coefficients. This method includes the case when you want certain parameters to be zero. You can specify as many

parameters as you like. For example, you can specify certain parameters, request that `vgxvarx` estimate others, and specify other parameters with `[]` to indicate default values.

Specification Structures with Known Parameters

If you know the values of model parameters, create a model structure with the parameters. The following are the name-value pairs you can pass to `vgxset` for known parameter values:

Model Parameters

Name	Value
`a`	An `n`-vector of offset constants. The default is empty.
`AR0`	An `n`-by-`n` invertible matrix representing the zero-lag structural AR coefficients. The default is empty, which means an identity matrix.
`AR`	An `nAR`-element cell array of `n`-by-`n` matrices of AR coefficients. The default is empty.
`MA0`	An `n`-by-`n` invertible matrix representing the zero-lag structural MA coefficients. The default is empty, which means an identity matrix.
`MA`	An `nMA`-element cell array of `n`-by-`n` matrices of MA coefficients. The default is empty.
`b`	An `nX`-vector of regression coefficients. The default is empty.
`Q`	An `n`-by-`n` symmetric innovations covariance matrix. The default is empty, which means an identity matrix.
`ARlag`	A monotonically increasing `nAR`-vector of AR lags. The default is empty, which means all the lags from `1` to `p`, the maximum AR lag.
`MAlag`	A monotonically increasing `nMA`-vector of MA lags. The default is empty, which means all the lags from `1` to `q`, the maximum MA lag.

`vgxset` infers the model dimensionality, given by n, p, and q in "Types of VAR Models" on page 7-3, from the input parameters. These parameters are `n`, `nAR`, and `nMA` respectively in `vgxset` syntax. For more information, see "Specification Structures with No Parameter Values" on page 7-16.

The `ARlag` and `MAlag` vectors allow you to specify which lags you want to include. For example, to specify AR lags 1 and 3 without lag 2, set `ARlag` to `[1 3]`. This setting

corresponds to nAR = 2 for two specified lags, even though this is a third order model, since the maximum lag is 3.

The following example shows how to create a model structure when you have known parameters. Consider a VAR(1) model:

$$y_t = a + \begin{bmatrix} .5 & 0 & 0 \\ .1 & .1 & .3 \\ 0 & .2 & .3 \end{bmatrix} y_{t-1} + \varepsilon_t,$$

Specifically, $a = [0.05, 0, -.05]'$ and w_t are distributed as standard three-dimensional normal random variables.

Create a model specification structure with vgxset:

```
A1 = [.5 0 0;.1 .1 .3;0 .2 .3];
Q = eye(3);
Mdl = vgxset('a',[.05;0;-.05],'AR',{A1},'Q',Q)
```

```
Mdl =

       Model: 3-D VAR(1) with Additive Constant
           n: 3
         nAR: 1
         nMA: 0
          nX: 0
           a: [0.05 0 -0.05] additive constants
          AR: {1x1 cell} stable autoregressive process
           Q: [3x3] covariance matrix
```

vgxset identifies this model as a stable VAR(1) model with three dimensions and additive constants.

Specification Structures with No Parameter Values

By default, vgxvarx fits all unspecified additive (a), AR, regression coefficients (b), and Q parameters. You must specify the number of time series and the type of model you want vgxvarx to fit. The following are the name-value pairs you can pass to vgxset for unknown parameter values:

Model Orders

Name	Value
`n`	A positive integer specifying the number of time series. The default is `1`.
`nAR`	A nonnegative integer specifying the number of AR lags (corresponds to p in "Types of VAR Models" on page 7-3). The default is `0`.
`nMA`	A nonnegative integer specifying the number of MA lags (corresponds to q in "Types of VAR Models" on page 7-3). The default is `0`. Currently, `vgxvarx` cannot fit `MA` matrices. Therefore, specifying an `nMA` greater than 0 does not yield estimated `MA` matrices.
`nX`	A nonnegative integer specifying the number regression parameters (corresponds to r in "Types of VAR Models" on page 7-3). The default is `0`.
`Constant`	Additive offset logical indicator. The default is `false`.

The following example shows how to specify the model in "Specification Structures with Known Parameters" on page 7-15, but without explicit parameters.

```
Mdl = vgxset('n',3,'nAR',1,'Constant',true)
```

```
Mdl =

       Model: 3-D VAR(1) with Additive Constant
           n: 3
         nAR: 1
         nMA: 0
          nX: 0
           a: []
          AR: {}
           Q: []
```

Specification Structures with Selected Parameter Values

You can create a model structure with some known parameters, and have `vgxvarx` fit the unknown parameters to data. Here are the name-value pairs you can pass to `vgxset` for requested parameter values:

Model Parameter Estimation

Name	Value
asolve	An n-vector of additive offset logical indicators. The default is empty, which means true(n,1).
ARsolve	An nAR-element cell array of n-by-n matrices of AR logical indicators. The default is empty, which means an nAR-element cell array of true(n).
AR0solve	An n-by-n matrix of AR0 logical indicators. The default is empty, which means false(n).
MAsolve	An nMA-element cell array of n-by-n matrices of MA logical indicators. The default is empty, which means false(n).
MA0solve	An n-by-n matrix of MA0 logical indicators. The default is empty, which means false(n).
bsolve	An nX-vector of regression logical indicators. The default is empty, which means true(n,1).
Qsolve	An n-by-n symmetric covariance matrix logical indicator. The default is empty, which means true(n), unless the 'CovarType' option of vgxvarx overrides it.

Specify a logical 1 (true) for every parameter that you want vgxvarx to estimate.

Currently, vgxvarx cannot fit the AR0, MA0, or MA matrices. Therefore, vgxvarx ignores the AR0solve, MA0solve, and MAsolve indicators. However, you can examine the Example_StructuralParams.m file for an approach to estimating the AR0 and MA0 matrices. Enter help Example_StructuralParams at the MATLAB command line for information. See Lütkepohl [74] Chapter 9 for algorithms for estimating structural models.

Currently, vgxvarx also ignores the Qsolve matrix. vgxvarx can fit either a diagonal or a full Q matrix; see vgxvarx.

This example shows how to specify the model in "Specification Structures with Known Parameters" on page 7-15, but with requested AR parameters with a diagonal autoregressive structure. The dimensionality of the model is known, as is the parameter vector a, but the autoregressive matrix A1 and covariance matrix Q are not known.

```
Mdl = vgxset('ARsolve',{logical(eye(3))},'a',...
    [.05;0;-.05])
```

```
Mdl =

      Model: 3-D VAR(1) with Additive Constant
          n: 3
        nAR: 1
        nMA: 0
         nX: 0
          a: [0.05 0 -0.05] additive constants
         AR: {}
    ARsolve: {1x1 cell of logicals} autoregressive lag indicators
          Q: []
```

Displaying and Changing a Specification Structure

After you set up a model structure, you can examine it in several ways:

- Call the `vgxdisp` function.
- Double-click the structure in the MATLAB Workspace browser.
- Call the `vgxget` function.
- Enter `Spec` at the MATLAB command line, where *`Spec`* is the name of the model structure.
- Enter `Spec.ParamName` at the MATLAB command line, where *`Spec`* is the name of the model structure, and *`ParamName`* is the name of the parameter you want to examine.

You can change any part of a model structure named, for example, `Spec`, using the `vgxset` as follows:

```
Spec = vgxset(Spec,'ParamName',value,...)
```

This syntax changes only the `'ParamName'` parts of the `Spec` structure.

Determining an Appropriate Number of Lags

There are two Econometrics Toolbox functions that can help you determine an appropriate number of lags for your models:

- The `lratiotest` function performs likelihood ratio tests to help identify the appropriate number of lags.

- The `aicbic` function calculates the Akaike and Bayesian information criteria to determine the minimal appropriate number of required lags.

Example: Using the Likelihood Ratio Test to Calculate the Minimal Requisite Lag

`lratiotest` requires inputs of the loglikelihood of an unrestricted model, the loglikelihood of a restricted model, and the number of degrees of freedom (DoF). DoF is the difference in the number of active parameters between the unrestricted and restricted models. `lratiotest` returns a Boolean: `1` means reject the restricted model in favor of the unrestricted model, `0` means there is insufficient evidence to reject the restricted model.

In the context of determining an appropriate number of lags, the restricted model has fewer lags, and the unrestricted model has more lags. Otherwise, test models with the same type of fitted parameters (for example, both with full `Q` matrices, or both with diagonal `Q` matrices).

- Obtain the loglikelihood (LLF) of a model as the third output of `vgxvarx`:

  ```
  [EstSpec,EstStdErrors,LLF,W] = vgxvarx(...)
  ```

- Obtain the number of active parameters in a model as the second output of `vgxcount`:

  ```
  [NumParam,NumActive] = vgxcount(Spec)
  ```

For example, suppose you have four fitted models with varying lag structures. The models have loglikelihoods `LLF1`, `LLF2`, `LLF3`, and `LLF4`, and active parameter counts `n1p`, `n2p`, `n3p`, and `n4p`. Suppose model 4 has the largest number of lags. Perform likelihood ratio tests of models 1, 2, and 3 against model 4, as follows:

```
reject1 = lratiotest(LLF4,LLF1,n4p - n1p)
reject2 = lratiotest(LLF4,LLF2,n4p - n2p)
reject3 = lratiotest(LLF4,LLF3,n4p - n3p)
```

If `reject1 = 1`, you reject model 1 in favor of model 4, and similarly for models 2 and 3. If any of the models have `rejectI = 0`, you have an indication that you can use fewer lags than in model 4.

Example: Using Akaike Information Criterion to Calculate the Minimal Requisite Lag

`aicbic` requires inputs of the loglikelihood of a model and of the number of active parameters in the model. It returns the value of the Akaike information criterion. Lower

values are better than higher values. `aicbic` accepts vectors of loglikelihoods and vectors of active parameters, and returns a vector of Akaike information criteria, which makes it easy to find the minimum.

- Obtain the loglikelihood (LLF) of a model as the third output of `vgxvarx`:

  ```
  [EstSpec,EstStdErrors,LLF,W] = vgxvarx(...)
  ```

- Obtain the number of active parameters in a model as the second output of `vgxcount`:

  ```
  [NumParam,NumActive] = vgxcount(Spec)
  ```

For example, suppose you have four fitted models with varying lag structures. The models have loglikelihoods `LLF1`, `LLF2`, `LLF3`, and `LLF4`, and active parameter counts `n1p`, `n2p`, `n3p`, and `n4p`. Calculate the Akaike information criteria as follows:

```
AIC = aicbic([LLF1 LLF2 LLF3 LLF4],[n1p n2p n3p n4p])
```

The most suitable model has the lowest value of the Akaike information criterion.

See Also

`var2vec` | `vec2var` | `vgxar` | `vgxma` | `vgxqual` | `vgxvarx`

Related Examples

- "Fit a VAR Model" on page 7-33
- "Fit a VARMA Model" on page 7-35
- "Convert a VARMA Model to a VMA Model" on page 7-29
- "Convert a VARMA Model to a VAR Model" on page 7-27
- "Forecast a VAR Model" on page 7-50
- "VAR Model Case Study" on page 7-89

More About

- "Vector Autoregressive (VAR) Models" on page 7-3

VAR Model Estimation

In this section...

Preparing Models for Fitting

To create a model of multiple time series data, decide on a parametric form of the model, and fit parameters to the data. When you have a calibrated (fitted) model, check if the model fits the data adequately.

To fit a model to data, you must have:

- Time series data, as described in "Multivariate Time Series Data" on page 7-8
- At least one time series model specification structure, as described in "Multivariate Time Series Model Creation" on page 7-14

There are several Econometrics Toolbox functions that aid these tasks, including:

- `vgxvarx`, which fits VARX models.
- `vgxar` and `vgxma`, which convert models to pure AR or MA models; `vgxar` enables you to fit VARMA models with `vgxvarx`, as described in "Fit a VARMA Model" on page 7-35
- `lratiotest`, `lmtest`, `waldtest`, and `aicbic`, which can help determine the number of lags to include in a model.
- `vgxqual`, which examines the stability of models, as described in "Examining the Stability of a Fitted Model" on page 7-25.
- `vgxpred`, which creates forecasts that can be used to check the adequacy of the fit, as described in "VAR Model Forecasting, Simulation, and Analysis" on page 7-39

Structural Matrices

The structural matrices in SVARMAX models are the A_0 and B_0 matrices. See "Types of VAR Models" on page 7-3 for definitions of these terms. Currently,

vgxvarx cannot fit these matrices to data. However, you can examine the Example_StructuralParams.m file for an approach to estimating the AR0 and MA0 matrices. Enter help Example_StructuralParams at the MATLAB command line for information. See Lütkepohl [74] Chapter 9 for algorithms for estimating structural models.

Changing Model Representations

You can convert a VARMA model to an equivalent VAR model using the vgxar function. (See "Types of VAR Models" on page 7-3 for definitions of these terms.) Similarly, you can convert a VARMA model to an equivalent VMA model using the vgxma function. These functions are useful in the following situations:

- Calibration of models

  The vgxvarx function can calibrate only VAR and VARX models. So to calibrate a VARMA model, you could first convert it to a VAR model. However, you can ask vgxvarx to ignore VMA terms and fit just the VAR structure. See "Fit a VARMA Model" on page 7-35 for a comparison of conversion versus no conversion.

- Forecasting

  It is straightforward to generate forecasts for VMA models. In fact, vgxpred internally converts models to VMA models to calculate forecast statistics.

- Analyzing models

  Sometimes it is easier to define your model using one structure, but you want to analyze it using a different structure.

The algorithm for conversion between models involves series that are, in principle, infinite. The vgxar and vgxma functions truncate these series to the maximum of nMA and nAR, introducing an inaccuracy. You can specify that the conversion give more terms, or give terms to a specified accuracy. See [74] for more information on these transformations.

For model conversion examples, see "Convert a VARMA Model to a VAR Model" on page 7-27 and "Convert a VARMA Model to a VMA Model" on page 7-29.

Conversion Types and Accuracy

Some conversions occur when explicitly requested, such as those initiated by calls to vgxar and vgxma. Other conversions occur automatically as needed for calculations.

For example, vgxpred internally converts models to VMA models to calculate forecast statistics.

By default, conversions give terms up to the largest lag present in the model. However, for more accuracy in conversion, you can specify that the conversion use more terms. You can also specify that it continue until a residual term is below a threshold you set. The syntax is

```
SpecAR = vgxar(Spec,nAR,ARlag,Cutoff)
SpecMA = vgxma(Spec,nMA,MAlag,Cutoff)
```

- nMA and nAR represent the number of terms in the series.
- ARlag and MAlag are vectors of particular lags that you want in the converted model.
- Cutoff is a positive parameter that truncates the series if the norm of a converted term is smaller than Cutoff. Cutoff is 0 by default.

For details, see the function reference pages for vgxar and vgxma.

Fitting Models to Data

The vgxvarx function performs parameter estimation. vgxvarx only estimates parameters for VAR and VARX models. In other words, vgxvarx does not estimate moving average matrices, which appear, for example, in VMA and VARMA models. For definitions of these terms, see “Types of VAR Models” on page 7-3. For an example of fitting a VAR model to data, see “Fit a VAR Model” on page 7-33.

The vgxar function converts a VARMA model to a VAR model. Currently, it does not handle VARMAX models.

You have two choices in fitting parameters to a VARMA model or VARMAX model:

- Set the vgxvarx 'IgnoreMA' parameter to 'yes' (the default is 'no'). In this case vgxvarx ignores VMA parameters, and fits the VARX parameters.
- Convert a VARMA model to a VAR model using vgxar. Then fit the resulting VAR model using vgxvarx.

Each of these options is effective on some data. Try both if you have VMA terms in your model. See “Fit a VARMA Model” on page 7-35 for an example showing both options.

How vgxvarx Works

`vgxvarx` finds maximum likelihood estimators of `AR` and `Q` matrices and the `a` and `b` vectors if present. For VAR models and if the response series do not contain `NaN` values, `vgxvarx` uses a direct solution algorithm that requires no iterations. For VARX models or if the response data contain missing values, `vgxvarx` optimizes the likelihood using the expectation-conditional-maximization (ECM) algorithm. The iterations usually converge quickly, unless two or more exogenous data streams are proportional to each other. In that case, there is no unique maximum likelihood estimator, and the iterations might not converge. You can set the maximum number of iterations with the `MaxIter` parameter, which has a default value of 1000. `vgxvarx` does not support exogenous series containing `NaN` values.

`vgxvarx` calculates the loglikelihood of the data, giving it as an output of the fitted model. Use this output in testing the quality of the model. For example, see "Determining an Appropriate Number of Lags" on page 7-19 and "Examining the Stability of a Fitted Model" on page 7-25.

Examining the Stability of a Fitted Model

When you enter the name of a fitted model at the command line, you obtain a summary. This summary contains a report on the stability of the VAR part of the model, and the invertibility of the VMA part. You can also find whether a model is stable or invertible by entering:

```
[isStable,isInvertible] = vgxqual(Spec)
```

This gives a Boolean value of `1` for `isStable` if the model is stable, and a Boolean value of `1` for `isInvertible` if the model is invertible. This stability or invertibility does not take into account exogenous terms, which can disrupt the stability of a model.

Stable, invertible models give reliable results, while unstable or noninvertible ones might not.

Stability and invertibility are equivalent to all eigenvalues of the associated lag operators having modulus less than 1. In fact `vgxqual` evaluates these quantities by calculating eigenvalues. For more information, see the `vgxqual` function reference page or Hamilton [52]

See Also

`aicbic` | `lratiotest` | `var2vec` | `vec2var` | `vgxar` | `vgxinfer` | `vgxma` | `vgxqual` | `vgxvarx`

Related Examples

- "Fit a VAR Model" on page 7-33
- "Fit a VARMA Model" on page 7-35
- "Convert a VARMA Model to a VMA Model" on page 7-29
- "Convert a VARMA Model to a VAR Model" on page 7-27
- "Forecast a VAR Model" on page 7-50
- "VAR Model Case Study" on page 7-89

More About

- "Vector Autoregressive (VAR) Models" on page 7-3

Convert a VARMA Model to a VAR Model

This example creates a VARMA model, then converts it to a pure VAR model.

Create a VARMA model specification structure.

```
A1 = [.2 -.1 0;.1 .2 .05;0 .1 .3];
A2 = [.3 0 0;.1 .4 .1;0 0 .2];
A3 = [.4 .1 -.1;.2 -.5 0;.05 .05 .2];
MA1 = .2*eye(3);
MA2 = [.3 .2 .1;.2 .4 0;.1 0 .5];
Spec = vgxset('AR',{A1,A2,A3},'MA',{MA1,MA2})
```

```
Spec =

      Model: 3-D VARMA(3,2) with No Additive Constant
          n: 3
        nAR: 3
        nMA: 2
         nX: 0
         AR: {3x1 cell} stable autoregressive process
         MA: {2x1 cell} invertible moving average process
          Q: []
```

Convert the structure to a pure VAR structure:

```
SpecAR = vgxar(Spec)
```

```
SpecAR =

      Model: 3-D VAR(3) with No Additive Constant
          n: 3
        nAR: 3
        nMA: 0
         nX: 0
         AR: {3x1 cell} unstable autoregressive process
          Q: []
```

The converted process is unstable; see the AR row. An unstable model could yield inaccurate predictions. Taking more terms in the series gives a stable model:

```
specAR4 = vgxar(Spec,4)


specAR4 =

      Model: 3-D VAR(4) with No Additive Constant
          n: 3
        nAR: 4
        nMA: 0
         nX: 0
         AR: {4x1 cell} stable autoregressive process
          Q: []
```

See Also

`var2vec` | `vec2var` | `vgxar` | `vgxma` | `vgxvarx`

Related Examples

- "Fit a VARMA Model" on page 7-35
- "Convert a VARMA Model to a VMA Model" on page 7-29
- "Forecast a VAR Model" on page 7-50
- "VAR Model Case Study" on page 7-89

More About

- "Vector Autoregressive (VAR) Models" on page 7-3
- "VAR Model Estimation" on page 7-22

Convert a VARMA Model to a VMA Model

This example uses a VARMA model and converts it to a pure VMA model.

Create a VARMA model specification structure.

```
A1 = [.2 -.1 0;.1 .2 .05;0 .1 .3];
A2 = [.3 0 0;.1 .4 .1;0 0 .2];
A3 = [.4 .1 -.1;.2 -.5 0;.05 .05 .2];
MA1 = .2*eye(3);
MA2 = [.3 .2 .1;.2 .4 0;.1 0 .5];
Spec = vgxset('AR',{A1,A2,A3},'MA',{MA1,MA2})
```

```
Spec =

       Model: 3-D VARMA(3,2) with No Additive Constant
           n: 3
         nAR: 3
         nMA: 2
          nX: 0
          AR: {3x1 cell} stable autoregressive process
          MA: {2x1 cell} invertible moving average process
           Q: []
```

Convert the structure to a pure VAR structure:

```
SpecAR = vgxar(Spec)
```

```
SpecAR =

       Model: 3-D VAR(3) with No Additive Constant
           n: 3
         nAR: 3
         nMA: 0
          nX: 0
          AR: {3x1 cell} unstable autoregressive process
           Q: []
```

Convert the model specification structure `Spec` to a pure MA structure:

```
SpecMA = vgxma(Spec)
```

```
SpecMA =

      Model: 3-D VMA(3) with No Additive Constant
          n: 3
        nAR: 0
        nMA: 3
         nX: 0
         MA: {3x1 cell} non-invertible moving average process
          Q: []
```

Notice that the pure VMA process has more MA terms than the original process. The number is the maximum of `nMA` and `nAR`, and `nAR = 3`.

The converted VMA model is not invertible; see the `MA` row. A noninvertible model can yield inaccurate predictions. Taking more terms in the series results in an invertible model.

```
specMA4 = vgxma(Spec,4)
```

```
specMA4 =

      Model: 3-D VMA(4) with No Additive Constant
          n: 3
        nAR: 0
        nMA: 4
         nX: 0
         MA: {4x1 cell} invertible moving average process
          Q: []
```

Converting the resulting VMA model to a pure VAR model results in the same VAR(3) model as `SpecAR`.

```
SpecAR2 = vgxar(SpecMA);
vgxdisp(SpecAR,SpecAR2)
```

```
  Model 1: 3-D VAR(3) with No Additive Constant
           Conditional mean is not AR-stable and is MA-invertible
  Model 2: 3-D VAR(3) with No Additive Constant
           Conditional mean is not AR-stable and is MA-invertible
       Parameter        Model 1        Model 2
  -------------- -------------- --------------
```

AR(1)(1,1)	0.4	0.4
(1,2)	-0.1	-0.1
(1,3)	-0	-0
(2,1)	0.1	0.1
(2,2)	0.4	0.4
(2,3)	0.05	0.05
(3,1)	-0	-0
(3,2)	0.1	0.1
(3,3)	0.5	0.5
AR(2)(1,1)	0.52	0.52
(1,2)	0.22	0.22
(1,3)	0.1	0.1
(2,1)	0.28	0.28
(2,2)	0.72	0.72
(2,3)	0.09	0.09
(3,1)	0.1	0.1
(3,2)	-0.02	-0.02
(3,3)	0.6	0.6
AR(3)(1,1)	0.156	0.156
(1,2)	-0.004	-0.004
(1,3)	-0.18	-0.18
(2,1)	0.024	0.024
(2,2)	-0.784	-0.784
(2,3)	-0.038	-0.038
(3,1)	-0.01	-0.01
(3,2)	0.014	0.014
(3,3)	-0.17	-0.17
Q(:,:)	[]	[]

See Also

`var2vec` | `vec2var` | `vgxar` | `vgxma` | `vgxvarx`

Related Examples

- "Fit a VARMA Model" on page 7-35
- "Convert a VARMA Model to a VAR Model" on page 7-27
- "Forecast a VAR Model" on page 7-50
- "VAR Model Case Study" on page 7-89

More About

- "Vector Autoregressive (VAR) Models" on page 7-3

- "VAR Model Estimation" on page 7-22

Fit a VAR Model

This example uses two series: the consumer price index (CPI) and the unemployment rate (UNRATE) from the data set `Data_USEconmodel`.

Obtain the two time series, and convert them for stationarity:

```
load Data_USEconModel
cpi = DataTable.CPIAUCSL;
cpi = log(cpi);
dCPI = diff(cpi);
unem = DataTable.UNRATE;
Y = [dCPI,unem(2:end)];
```

Create a VAR model:

```
Spec = vgxset('n',2,'nAR',4,'Constant',true)
```

```
Spec =

      Model: 2-D VAR(4) with Additive Constant
          n: 2
        nAR: 4
        nMA: 0
         nX: 0
          a: []
         AR: {}
          Q: []
```

Fit the model to the data using `vgxvarx`:

```
[EstSpec,EstStdErrors,logL,W] = vgxvarx(Spec,Y);
vgxdisp(EstSpec)
```

```
  Model  : 2-D VAR(4) with Additive Constant
           Conditional mean is AR-stable and is MA-invertible
  a Constant:
       0.00184568
         0.315567
  AR(1) Autoregression Matrix:
         0.308635     -0.0032011
        -4.48152         1.34343
```

```
AR(2) Autoregression Matrix:
      0.224224     0.00124669
       7.19015       -0.26822
AR(3) Autoregression Matrix:
      0.353274     0.00287036
       1.48726      -0.227145
AR(4) Autoregression Matrix:
     -0.0473456   -0.000983119
       8.63672      0.0768312
Q Innovations Covariance:
   3.51443e-05   -0.000186967
  -0.000186967       0.116697
```

See Also

`vgxinfer` | `vgxpred` | `vgxvarx`

Related Examples

- "Fit a VARMA Model" on page 7-35
- "Forecast a VAR Model" on page 7-50
- "VAR Model Case Study" on page 7-89

More About

- "Vector Autoregressive (VAR) Models" on page 7-3
- "VAR Model Estimation" on page 7-22

Fit a VARMA Model

This example uses artificial data to generate a time series, then shows two ways of fitting a VARMA model to the series.

Specify the model:

```
AR1 = [.3 -.1 .05;.1 .2 .1;-.1 .2 .4];
AR2 = [.1 .05 .001;.001 .1 .01;-.01 -.01 .2];
Q = [.2 .05 .02;.05 .3 .1;.02 .1 .25];
MA1 = [.5 .2 .1;.1 .6 .2;0 .1 .4];
MA2 = [.2 .1 .1; .05 .1 .05;.02 .04 .2];
Spec = vgxset('AR',{AR1,AR2},'Q',Q,'MA',{MA1,MA2})
```

```
Spec =

      Model: 3-D VARMA(2,2) with No Additive Constant
          n: 3
        nAR: 2
        nMA: 2
         nX: 0
         AR: {2x1 cell} stable autoregressive process
         MA: {2x1 cell} invertible moving average process
          Q: [3x3] covariance matrix
```

Generate a time series using `vgxsim`:

```
YF = [100 50 20;110 52 22;119 54 23]; % starting values
rng(1);                               % For reproducibility
Y = vgxsim(Spec,190,[],YF);
```

Fit the data to a VAR model by calling `vgxvarx` with the `'IgnoreMA'` option:

```
estSpec = vgxvarx(Spec,Y(3:end,:),[],Y(1:2,:),'IgnoreMA','yes');
```

Compare the estimated model with the original:

```
vgxdisp(Spec,estSpec)
```

```
  Model 1: 3-D VARMA(2,2) with No Additive Constant
           Conditional mean is AR-stable and is MA-invertible
  Model 2: 3-D VAR(2) with No Additive Constant
           Conditional mean is AR-stable and is MA-invertible
       Parameter        Model 1        Model 2
```

AR(1)(1,1)	0.3	0.723964
(1,2)	-0.1	0.119695
(1,3)	0.05	0.10452
(2,1)	0.1	0.0828041
(2,2)	0.2	0.788177
(2,3)	0.1	0.299648
(3,1)	-0.1	-0.138715
(3,2)	0.2	0.397231
(3,3)	0.4	0.748157
AR(2)(1,1)	0.1	-0.126833
(1,2)	0.05	-0.0690256
(1,3)	0.001	-0.118524
(2,1)	0.001	0.0431623
(2,2)	0.1	-0.265387
(2,3)	0.01	-0.149646
(3,1)	-0.01	0.107702
(3,2)	-0.01	-0.304243
(3,3)	0.2	0.0165912
MA(1)(1,1)	0.5	
(1,2)	0.2	
(1,3)	0.1	
(2,1)	0.1	
(2,2)	0.6	
(2,3)	0.2	
(3,1)	0	
(3,2)	0.1	
(3,3)	0.4	
MA(2)(1,1)	0.2	
(1,2)	0.1	
(1,3)	0.1	
(2,1)	0.05	
(2,2)	0.1	
(2,3)	0.05	
(3,1)	0.02	
(3,2)	0.04	
(3,3)	0.2	
Q(1,1)	0.2	0.193553
Q(2,1)	0.05	0.0408221
Q(2,2)	0.3	0.252461
Q(3,1)	0.02	0.00690626
Q(3,2)	0.1	0.0922074
Q(3,3)	0.25	0.306271

The estimated `Q` matrix is close to the original `Q` matrix. However, the estimated AR terms are not close to the original AR terms. Specifically, nearly all the AR(2) coefficients are the opposite sign, and most AR(1) coefficients are off by about a factor of 2.

Alternatively, before fitting the model, convert it to a pure AR model. To do this, specify the model and generate a time series as above. Then, convert the model to a pure AR model:

```
specAR = vgxar(Spec);
```

Fit the converted model to the data:

```
estSpecAR = vgxvarx(specAR,Y(3:end,:),[],Y(1:2,:));
```

Compare the fitted model to the original model:

```
vgxdisp(specAR,estSpecAR)
```

```
  Model 1: 3-D VAR(2) with No Additive Constant
           Conditional mean is AR-stable and is MA-invertible
  Model 2: 3-D VAR(2) with No Additive Constant
           Conditional mean is AR-stable and is MA-invertible
       Parameter        Model 1        Model 2
  -------------- -------------- --------------
      AR(1)(1,1)            0.8       0.723964
           (1,2)            0.1       0.119695
           (1,3)           0.15        0.10452
           (2,1)            0.2      0.0828041
           (2,2)            0.8       0.788177
           (2,3)            0.3       0.299648
           (3,1)           -0.1      -0.138715
           (3,2)            0.3       0.397231
           (3,3)            0.8       0.748157
      AR(2)(1,1)          -0.13      -0.126833
           (1,2)          -0.09     -0.0690256
           (1,3)         -0.114      -0.118524
           (2,1)         -0.129      0.0431623
           (2,2)          -0.35      -0.265387
           (2,3)         -0.295      -0.149646
           (3,1)           0.03       0.107702
           (3,2)          -0.17      -0.304243
           (3,3)           0.05      0.0165912
          Q(1,1)            0.2       0.193553
          Q(2,1)           0.05      0.0408221
          Q(2,2)            0.3       0.252461
```

```
Q(3,1)          0.02     0.00690626
Q(3,2)           0.1      0.0922074
Q(3,3)          0.25       0.306271
```

The model coefficients between the pure AR models are closer than between the original VARMA model and the fitted AR model. Most model coefficients are within 20% or the original. Notice, too, that `estSpec` and `estSpecAR` are identical. This is because both are AR(2) models fitted to the same data series.

See Also

`vgxinfer` | `vgxpred` | `vgxvarx`

Related Examples

- "Fit a VAR Model" on page 7-33
- "Forecast a VAR Model" on page 7-50
- "VAR Model Case Study" on page 7-89

More About

- "Vector Autoregressive (VAR) Models" on page 7-3
- "VAR Model Estimation" on page 7-22

VAR Model Forecasting, Simulation, and Analysis

In this section...

VAR Model Forecasting

When you have models with parameters (known or estimated), you can examine the predictions of the models. For information on specifying models, see "Multivariate Time Series Model Creation" on page 7-14. For information on calibrating models, see "VAR Model Estimation" on page 7-22.

The main methods of forecasting are:

- Generating forecasts with error bounds using `vgxpred`
- Generating simulations with `vgxsim`
- Generating sample paths with `vgxproc`

These functions base their forecasts on a model specification and initial data. The functions differ in their innovations processes:

- `vgxpred` assumes zero innovations. Therefore, `vgxpred` yields a deterministic forecast.
- `vgxsim` assumes the innovations are jointly normal with covariance matrix Q. `vgxsim` yields pseudorandom (Monte Carlo) sample paths.
- `vgxproc` uses a separate input for the innovations process. `vgxproc` yields a sample path that is deterministically based on the innovations process.

`vgxpred` is faster and takes less memory than generating many sample paths using `vgxsim`. However, `vgxpred` is not as flexible as `vgxsim`. For example, suppose you transform some time series before making a model, and want to undo the transformation when examining forecasts. The error bounds given by transforms of `vgxpred` error bounds are not valid bounds. In contrast, the error bounds given by the statistics of transformed simulations are valid.

For examples, see "Forecast a VAR Model" on page 7-50, "Forecast a VAR Model Using Monte Carlo Simulation" on page 7-53, and "Simulate Responses of Estimated VARX Model" on page 7-80.

How vgxpred and vgxsim Work

`vgxpred` generates two quantities:

- A deterministic forecast time series based on 0 innovations
- Time series of forecast covariances based on the Q matrix

The simulations for models with VMA terms uses presample innovation terms. Presample innovation terms are values of ε_t for times before the forecast period that affect the MA terms. For definitions of the terms MA, Q, and ε_t, see "Types of VAR Models" on page 7-3. If you do not provide all requisite presample innovation terms, `vgxpred` assumes the value 0 for missing terms.

`vgxsim` generates random time series based on the model using normal random innovations distributed with Q covariances. The simulations of models with MA terms uses presample innovation terms. If you do not provide all requisite presample innovation terms, `vgxsim` assumes the value 0 for missing terms.

Data Scaling

If you scaled any time series before fitting a model, you can unscale the resulting time series to understand its predictions more easily.

- If you scaled a series with `log`, transform predictions of the corresponding model with `exp`.
- If you scaled a series with `diff(log)`, transform predictions of the corresponding model with `cumsum(exp)`. `cumsum` is the inverse of `diff`; it calculates cumulative sums. As in integration, you must choose an appropriate additive constant for the cumulative sum. For example, take the log of the final entry in the corresponding data series, and use it as the first term in the series before applying `cumsum`.

Calculating Impulse Responses

You can examine the effect of *impulse responses* to models with the `vgxproc` function. An impulse response is the deterministic response of a time series model to an innovations process that has the value of one standard deviation in one component at the initial time,

and zeros in all other components and times. `vgxproc` simulates the evolution of a time series model from a given innovations process. Therefore, `vgxproc` is appropriate for examining impulse responses.

The only difficulty in using `vgxproc` is determining exactly what is "the value of one standard deviation in one component at the initial time." This value can mean different things depending on your model.

- For a structural model, B_0 is usually a known diagonal matrix, and Q is an identity matrix. In this case, the impulse response to component i is the square root of $B(i,i)$.
- For a nonstructural model, there are several choices. The simplest choice, though not necessarily the most accurate, is to take component i as the square root of $Q(i,i)$. Other possibilities include taking the Cholesky decomposition of Q, or diagonalizing Q and taking the square root of the diagonal matrix.

For an example, see "Generate Impulse Responses for a VAR model" on page 7-42.

See Also

`vgxpred` | `vgxproc` | `vgxsim` | `vgxvarx`

Related Examples

- "Fit a VAR Model" on page 7-33
- "Forecast a VAR Model" on page 7-50
- "Forecast a VAR Model Using Monte Carlo Simulation" on page 7-53
- "Simulate Responses of Estimated VARX Model" on page 7-80
- "Generate Impulse Responses for a VAR model" on page 7-42
- "VAR Model Case Study" on page 7-89

More About

- "Vector Autoregressive (VAR) Models" on page 7-3
- "Multivariate Time Series Data Structures" on page 7-8
- "VAR Model Estimation" on page 7-22
- "Multivariate Time Series Models with Regression Terms" on page 7-57

Generate Impulse Responses for a VAR model

This example shows how to generate impulse responses of an interest rate shock on real GDP using `vgxproc`.

Load the `Data_USEconModel` data set. This example uses two time series: the logarithm of real GDP, and the real 3-month T-bill rate, both differenced to be approximately stationary. Suppose that a VAR(4) model is appropriate to describe the time series.

```
load Data_USEconModel
DEF = log(DataTable.CPIAUCSL);
GDP = log(DataTable.GDP);
rGDP = diff(GDP - DEF); % Real GDP is GDP - deflation
TB3 = 0.01*DataTable.TB3MS;
dDEF = 4*diff(DEF); % Scaling
rTB3 = TB3(2:end) - dDEF; % Real interest is deflated
Y = [rGDP,rTB3];
```

Define the forecast horizon.

```
FDates = datenum({'30-Jun-2009'; '30-Sep-2009'; '31-Dec-2009';
'31-Mar-2010'; '30-Jun-2010'; '30-Sep-2010'; '31-Dec-2010';
'31-Mar-2011'; '30-Jun-2011'; '30-Sep-2011'; '31-Dec-2011';
'31-Mar-2012'; '30-Jun-2012'; '30-Sep-2012'; '31-Dec-2012';
'31-Mar-2013'; '30-Jun-2013'; '30-Sep-2013'; '31-Dec-2013';
'31-Mar-2014'; '30-Jun-2014' });
FT = numel(FDates);
```

Fit a VAR(4) model specification:

```
Spec = vgxset('n',2,'nAR',4,'Constant',true);
impSpec = vgxvarx(Spec,Y(5:end,:),[],Y(1:4,:));
impSpec = vgxset(impSpec,'Series',...
  {'Transformed real GDP','Transformed real 3-mo T-bill rate'});
```

Generate the innovations processes both with and without an impulse (shock):

```
W0 = zeros(FT, 2); % Innovations without a shock
W1 = W0;
W1(1,2) = sqrt(impSpec.Q(2,2)); % Innovations with a shock
```

Generate the processes with and without the shock:

```
Yimpulse = vgxproc(impSpec,W1,[],Y); % Process with shock
```

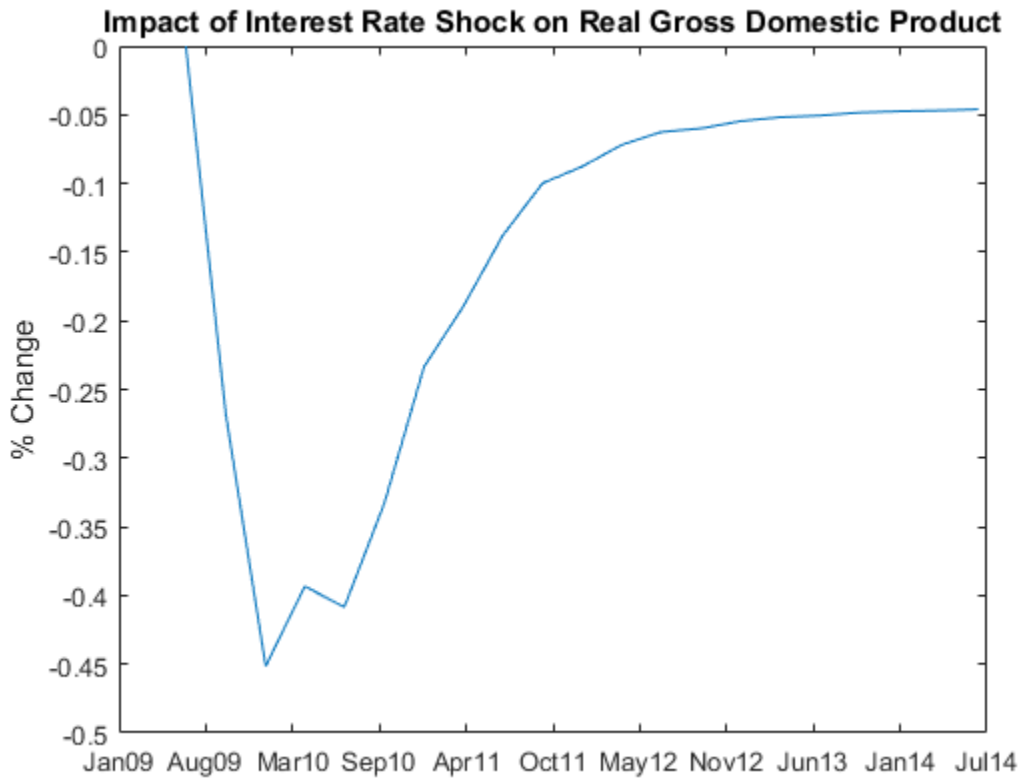
```
Ynoimp = vgxproc(impSpec,W0,[],Y); % Process with no shock
```

Undo the scaling for the GDP processes:

```
Yimp1 = exp(cumsum(Yimpulse(:,1))); % Undo scaling
Ynoimp1 = exp(cumsum(Ynoimp(:,1)));
```

Compute and plot the relative difference between the calculated GDPs:

```
RelDiff = (Yimp1 - Ynoimp1) ./ Yimp1;
plot(FDates,100*RelDiff);dateaxis('x',12)
title(...
'Impact of Interest Rate Shock on Real Gross Domestic Product')
ylabel('% Change')
```



The graph shows that an increased interest rate causes a dip in the real GDP for a short time. Afterwards the real GDP begins to climb again, reaching its former level in about 1 year.

See Also

`armairf` | `vgxpred` | `vgxproc` | `vgxsim` | `vgxvarx`

Related Examples

- “Fit a VAR Model” on page 7-33
- “Forecast a VAR Model” on page 7-50
- “VAR Model Case Study” on page 7-89

More About

- “Vector Autoregressive (VAR) Models” on page 7-3
- “VAR Model Forecasting, Simulation, and Analysis” on page 7-39

Compare Generalized and Orthogonalized Impulse Response Functions

This example shows the differences between orthogonal and generalized impulse response functions using the three-dimensional VAR(2) model in [74], p. 78. The variables in the model represent the quarterly rates of fixed investment, disposable income, and consumption expenditures of Germany. The estimated model is

$$y_t = \begin{bmatrix} -0.017 \\ 0.016 \\ 0.013 \end{bmatrix} + \begin{bmatrix} -0.320 & 0.146 & 0.961 \\ 0.044 & -0.153 & 0.289 \\ -0.002 & 0.225 & -0.264 \end{bmatrix} y_{t-1} + \begin{bmatrix} -0.161 & 0.115 & 0.934 \\ 0.050 & 0.019 & -0.010 \\ 0.034 & 0.355 & -0.022 \end{bmatrix} y_{t-2} + \varepsilon_t,$$

where $y_t = [y_{1t} \quad y_{2t} \quad y_{3t}]'$ and $\varepsilon_t = [\varepsilon_{1t} \quad \varepsilon_{2t} \quad \varepsilon_{3t}]'$. The estimated covariance matrix of the innovations is

$$\hat{\Sigma} = \begin{bmatrix} 21.30 & 0.72 & 1.23 \\ 0.72 & 1.37 & 0.61 \\ 1.23 & 0.61 & 0.89 \end{bmatrix} 10^{-4}.$$

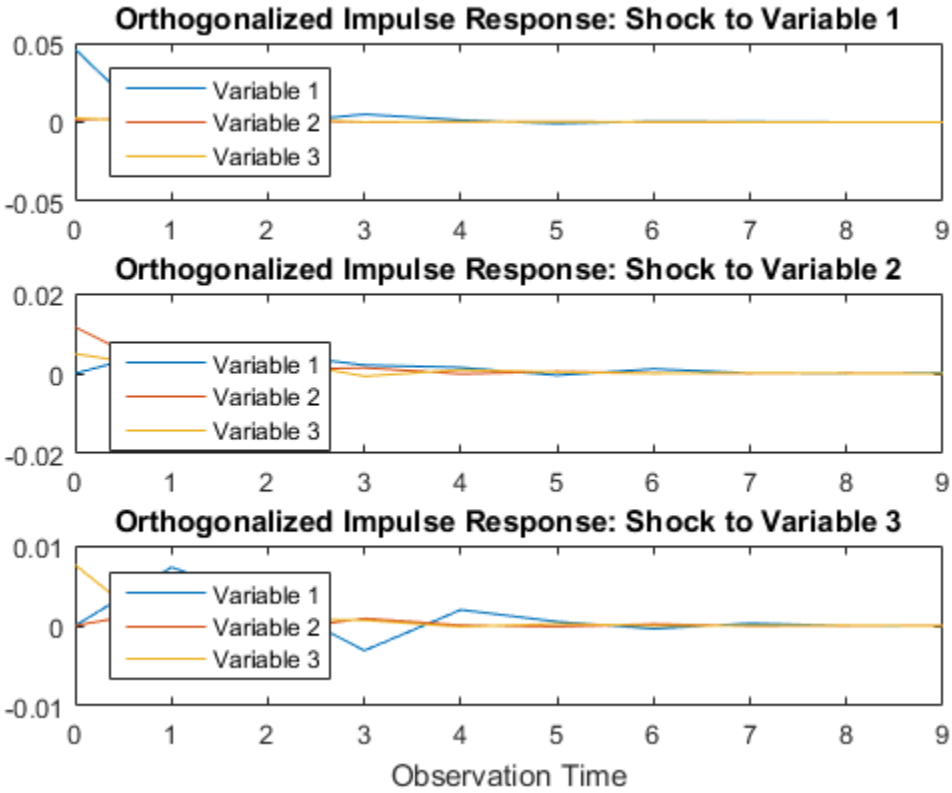
The VAR(2) model contains a constant, but because the impulse response function is the derivative of y_t with respect to ε_t, the constant does not affect the impulse response function.

Create a cell vector containing the autoregressive coefficient matrices, and a matrix for the innovations covariance matrix.

```
AR1 = [-0.320  0.146  0.961;
        0.044 -0.153  0.289;
       -0.002  0.225 -0.264];
AR2 = [-0.161 0.115  0.934;
        0.050 0.019 -0.010;
        0.034 0.355 -0.022];
ar0 = {AR1 AR2};

InnovCov = [21.30 0.72 1.23;
             0.72 1.37 0.61;
             1.23 0.61 0.89]*1e-4;
```

Plot and compute the orthogonalized impulse response function. Because no VMA coefficients exist, specify an empty array ([ ]) for the second input argument.

```
figure;
armairf(ar0,[],'InnovCov',InnovCov);
OrthoY = armairf(ar0,[],'InnovCov',InnovCov);
```



The impulse responses seem to die out after nine periods. `OrthoY` is a 10-by-3-by-3 matrix of impulse responses. The rows correspond to periods, columns correspond to a variable, and pages correspond to the variable receiving the shock.

Plot and compute the generalized impulse response function. Display both sets of impulse responses.

```
figure;
armairf(ar0,[],'InnovCov',InnovCov,'Method','generalized');
```

```
GenY = armairf(ar0,[],'InnovCov',InnovCov,'Method','generalized');

for j = 1:3
    fprintf('Shock to Response %d',j)
    table(OrthoY(:,:,j),GenY(:,:,j),'VariableNames',{'Orthogonal',...
        'Generalized'})
end
```

```
Shock to Response 1
ans =
```

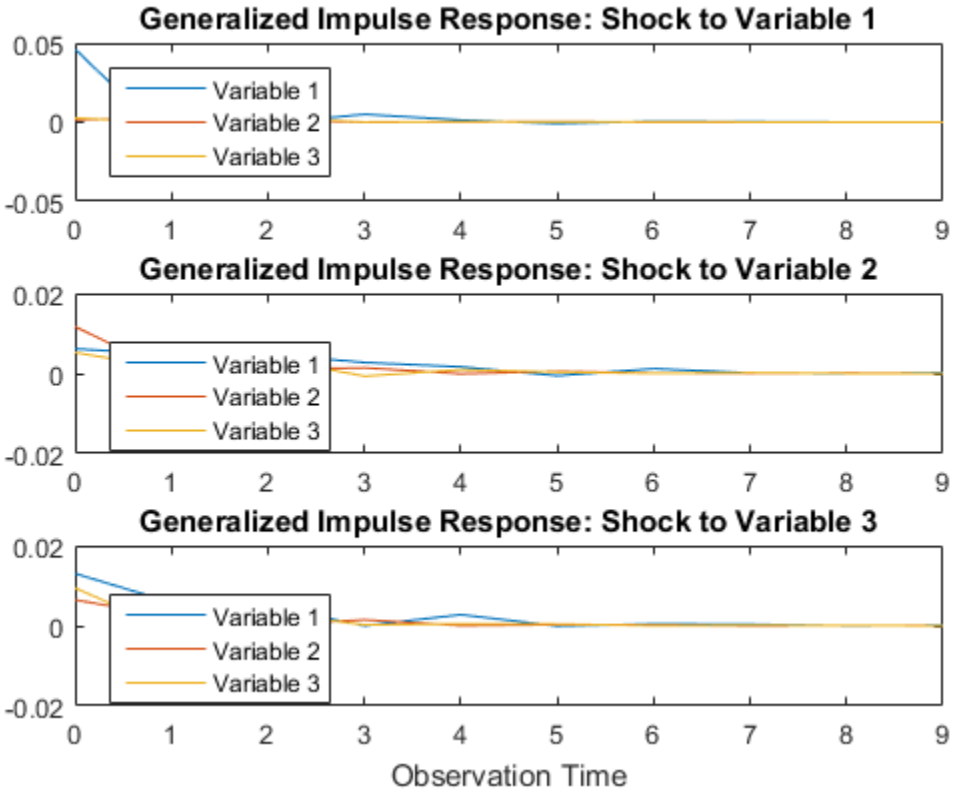
Orthogonal			Generalized		
0.046152	0.0015601	0.0026651	0.046152	0.0015601	0.0026
-0.01198	0.0025622	-0.00044488	-0.01198	0.0025622	-0.00044
-0.00098179	0.0012629	0.0027823	-0.00098179	0.0012629	0.0027
0.0049802	2.1799e-05	6.3661e-05	0.0049802	2.1799e-05	6.3661e
0.0013726	0.00018127	0.00033187	0.0013726	0.00018127	0.00033
-0.00083369	0.00037736	0.00012609	-0.00083369	0.00037736	0.00012
0.00055287	1.0779e-05	0.00015701	0.00055287	1.0779e-05	0.00015
0.00027093	3.2276e-05	6.2713e-05	0.00027093	3.2276e-05	6.2713e
3.7154e-05	5.1385e-05	9.3341e-06	3.7154e-05	5.1385e-05	9.3341e
2.325e-05	1.0003e-05	2.8313e-05	2.325e-05	1.0003e-05	2.8313e

```
Shock to Response 2
ans =
```

Orthogonal			Generalized		
0	0.0116	0.0049001	0.0061514	0.011705	0.0052
0.0064026	-0.00035872	0.0013164	0.0047488	-1.4011e-05	0.0012
0.0050746	0.00088845	0.0035692	0.0048985	0.0010489	0.0039
0.0020934	0.001419	-0.00069114	0.0027385	0.0014093	-0.00067
0.0014919	-8.9823e-05	0.00090697	0.0016616	-6.486e-05	0.00094
-0.00043831	0.00048004	0.00032749	-0.00054552	0.00052606	0.00034
0.0011216	6.5734e-05	2.1313e-05	0.0011853	6.6585e-05	4.205e
0.00010281	2.9385e-05	0.00015523	0.000138	3.3424e-05	0.0001
-3.2553e-05	0.00010201	2.6429e-05	-2.731e-05	0.00010795	2.7437e
0.00018252	-5.2551e-06	2.6551e-05	0.00018399	-3.875e-06	3.0088e

```
Shock to Response 3
ans =
```

Orthogonal			Generalized		
0	0	0.0076083	0.013038	0.006466	0.009
0.0073116	0.0021988	-0.0020086	0.0058379	0.0023108	-0.0010
0.0031572	-0.00067127	0.00084299	0.0049047	0.00027687	0.0033
-0.0030985	0.00091269	0.00069346	-4.6882e-06	0.0014793	0.00021
0.001993	6.1109e-05	-0.00012102	0.00277	5.3838e-05	0.00046
0.00050636	-0.00010115	0.00024511	-5.4815e-05	0.00027437	0.0004
-0.00036814	0.00021062	3.6381e-06	0.00044188	0.00020705	5.8359e
0.00028783	-2.6426e-05	2.3079e-05	0.00036206	3.0686e-06	0.00011
1.3105e-05	8.9361e-06	4.9558e-05	4.1567e-06	7.4706e-05	5.6331e
1.6913e-05	2.719e-05	-1.1202e-05	0.00011501	2.2025e-05	1.2756e



If `armairf` shocks the first variable, then the impulse responses of all variables are equivalent between methods. The second and third pages illustrate that the generalized and orthogonal impulse responses are generally different. However, if `InnovCov` is diagonal, then both methods produce the same impulse responses.

Another difference between the two methods is that generalized impulse responses are invariant to the order of the variables. However, orthogonal impulse responses differ with varying variable order.

See Also

`armairf` | `vgxproc`

Related Examples

- "Fit a VAR Model" on page 7-33
- "Generate VEC Model Impulse Responses" on page 7-138

More About

- "Vector Autoregressive (VAR) Models" on page 7-3
- "VAR Model Forecasting, Simulation, and Analysis" on page 7-39

Forecast a VAR Model

This example shows how to use `vgxpred` to forecast a VAR model.

`vgxpred` enables you to generate forecasts with error estimates. `vgxpred` requires:

- A fully-specified model (for example , `impSpec` in what follows)
- The number of periods for the forecast (for example, `FT` in what follows)

`vgxpred` optionally takes:

- An exogenous data series
- A presample time series (e.g., `Y(end-3:end,:)` in what follows)
- Extra paths

Load the `Data_USEconModel` data set. This example uses two time series: the logarithm of real GDP, and the real 3-month T-bill rate, both differenced to be approximately stationary. Suppose that a VAR(4) model is appropriate to describe the time series.

```
load Data_USEconModel
DEF = log(DataTable.CPIAUCSL);
GDP = log(DataTable.GDP);
rGDP = diff(GDP - DEF); % Real GDP is GDP - deflation
TB3 = 0.01*DataTable.TB3MS;
dDEF = 4*diff(DEF); % Scaling
rTB3 = TB3(2:end) - dDEF; % Real interest is deflated
Y = [rGDP,rTB3];
```

Fit a VAR(4) model specification:

```
Spec = vgxset('n',2,'nAR',4,'Constant',true);
impSpec = vgxvarx(Spec,Y(5:end,:),[],Y(1:4,:));
impSpec = vgxset(impSpec,'Series',...
  {'Transformed real GDP','Transformed real 3-mo T-bill rate'});
```

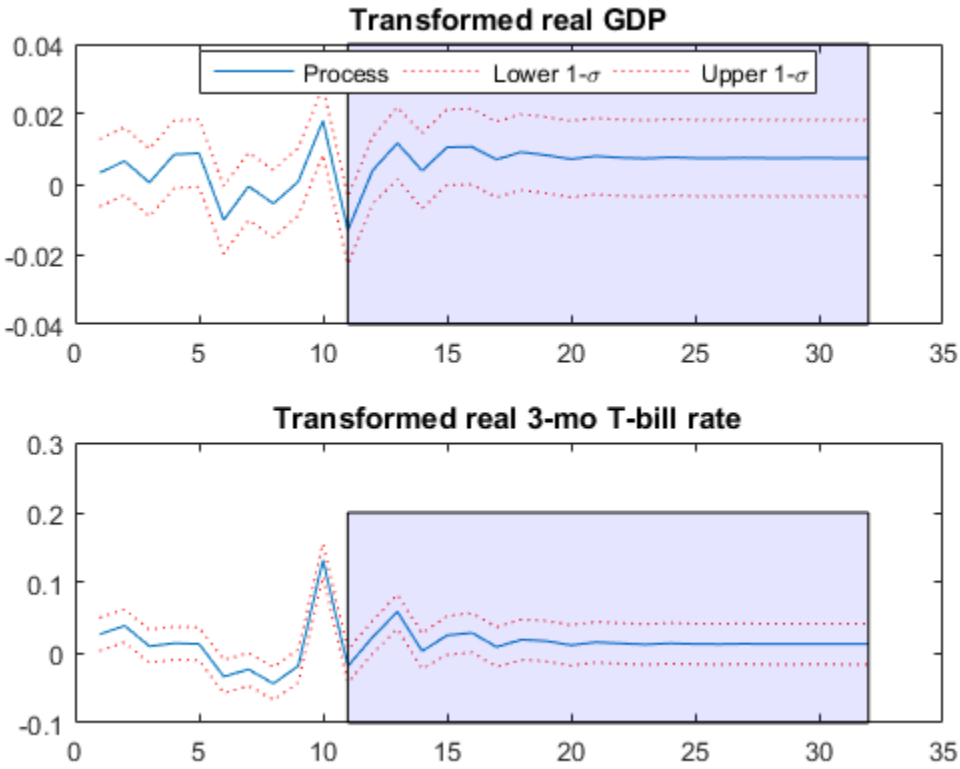
Predict the evolution of the time series:

```
FDates = datenum({'30-Jun-2009'; '30-Sep-2009'; '31-Dec-2009';
'31-Mar-2010'; '30-Jun-2010'; '30-Sep-2010'; '31-Dec-2010';
'31-Mar-2011'; '30-Jun-2011'; '30-Sep-2011'; '31-Dec-2011';
'31-Mar-2012'; '30-Jun-2012'; '30-Sep-2012'; '31-Dec-2012';
'31-Mar-2013'; '30-Jun-2013'; '30-Sep-2013'; '31-Dec-2013';
'31-Mar-2014'; '30-Jun-2014' });
FT = numel(FDates);
```

```
[Forecast,ForecastCov] = vgxpred(impSpec,FT,[],...
    Y(end-3:end,:));
```

View the forecast using `vgxplot`:

```
vgxplot(impSpec,Y(end-10:end,:),Forecast,ForecastCov);
```



See Also

`vgxpred` | `vgxproc` | `vgxvarx`

Related Examples

- "Fit a VAR Model" on page 7-33

- "Forecast a VAR Model Using Monte Carlo Simulation" on page 7-53
- "VAR Model Case Study" on page 7-89

More About

- "Vector Autoregressive (VAR) Models" on page 7-3
- "VAR Model Forecasting, Simulation, and Analysis" on page 7-39

Forecast a VAR Model Using Monte Carlo Simulation

This example shows how to use Monte Carlo simulation via `vgxsim` to forecast a VAR model.

`vgxsim` enables you to generate simulations of time series based on your model. If you have a trustworthy model structure, you can use these simulations as sample forecasts.

vgxsim requires:

- A model (`impSpec` in what follows)
- The number of periods for the forecast (`FT` in what follows)

vgxsim optionally takes:

- An exogenous data series
- A presample time series (`Y(end-3:end,:)` in what follows)
- Presample innovations
- The number of realizations to simulate (`2000` in what follows)

Load the `Data_USEconModel` data set. This example uses two time series: the logarithm of real GDP, and the real 3-month T-bill rate, both differenced to be approximately stationary. For illustration, a VAR(4) model describes the time series.

```
load Data_USEconModel
DEF = log(DataTable.CPIAUCSL);
GDP = log(DataTable.GDP);
rGDP = diff(GDP - DEF); % Real GDP is GDP - deflation
TB3 = 0.01*DataTable.TB3MS;
dDEF = 4*diff(DEF); % Scaling
rTB3 = TB3(2:end) - dDEF; % Real interest is deflated
Y = [rGDP,rTB3];
```

Fit a VAR(4) model specification:

```
Spec = vgxset('n',2,'nAR',4,'Constant',true);
impSpec = vgxvarx(Spec,Y(5:end,:),[],Y(1:4,:));
impSpec = vgxset(impSpec,'Series',...
  {'Transformed real GDP','Transformed real 3-mo T-bill rate'});
```

Define the forecast horizon.

```
FDates = datenum({'30-Jun-2009'; '30-Sep-2009'; '31-Dec-2009';
'31-Mar-2010'; '30-Jun-2010'; '30-Sep-2010'; '31-Dec-2010';
'31-Mar-2011'; '30-Jun-2011'; '30-Sep-2011'; '31-Dec-2011';
'31-Mar-2012'; '30-Jun-2012'; '30-Sep-2012'; '31-Dec-2012';
'31-Mar-2013'; '30-Jun-2013'; '30-Sep-2013'; '31-Dec-2013';
'31-Mar-2014'; '30-Jun-2014' });
FT = numel(FDates);
```

Simulate the model for 10 steps, replicated 2000 times:

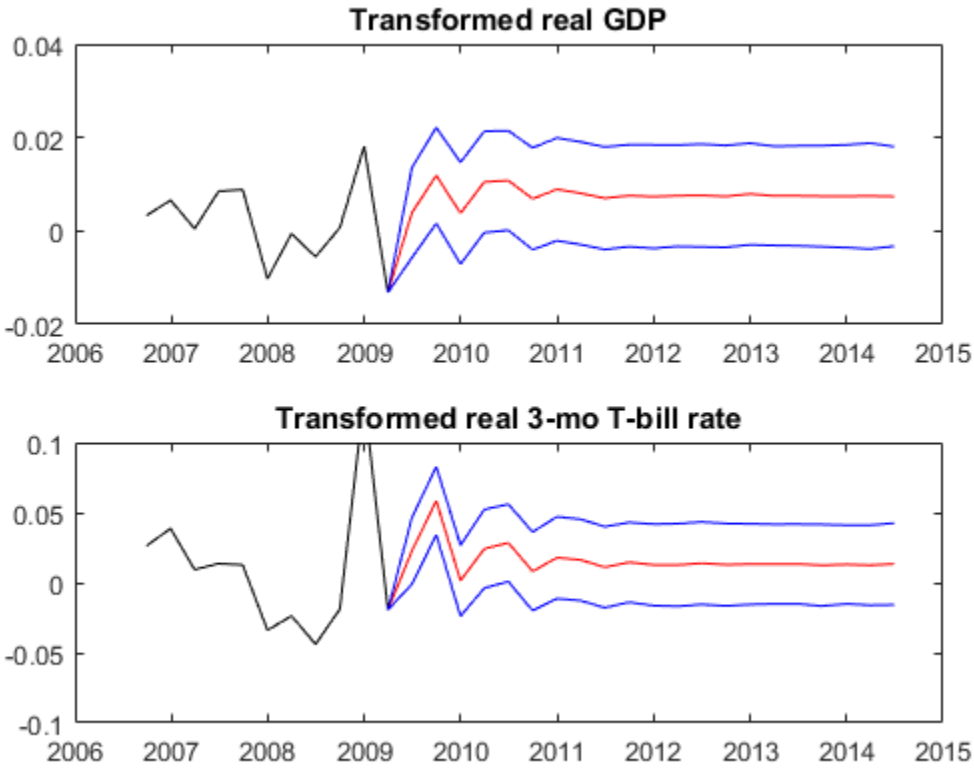
```
rng(1); %For reproducibility
Ysim = vgxsim(impSpec,FT,[],Y(end-3:end,:),[],2000);
```

Calculate the mean and standard deviation of the simulated series:

```
Ymean = mean(Ysim,3); % Calculate means
Ystd = std(Ysim,0,3); % Calculate std deviations
```

Plot the means +/- 1 standard deviation for the simulated series:

```
subplot(2,1,1)
plot(dates(end-10:end),Y(end-10:end,1),'k')
hold('on')
plot([dates(end);FDates],[Y(end,1);Ymean(:,1)],'r')
plot([dates(end);FDates],[Y(end,1);Ymean(:,1)]+[0;Ystd(:,1)],'b')
plot([dates(end);FDates],[Y(end,1);Ymean(:,1)]-[0;Ystd(:,1)],'b')
datetick('x')
title('Transformed real GDP')
subplot(2,1,2)
plot(dates(end-10:end),Y(end-10:end,2),'k')
hold('on')
axis([dates(end-10),FDates(end),-.1,.1]);
plot([dates(end);FDates],[Y(end,2);Ymean(:,2)],'r')
plot([dates(end);FDates],[Y(end,2);Ymean(:,2)]+[0;Ystd(:,2)],'b')
plot([dates(end);FDates],[Y(end,2);Ymean(:,2)]-[0;Ystd(:,2)],'b')
datetick('x')
title('Transformed real 3-mo T-bill rate')
```



See Also

`vgxpred` | `vgxproc` | `vgxsim` | `vgxvarx`

Related Examples

- “Fit a VAR Model” on page 7-33
- “Forecast a VAR Model” on page 7-50
- “Simulate Responses of Estimated VARX Model” on page 7-80
- “VAR Model Case Study” on page 7-89

More About

- "Vector Autoregressive (VAR) Models" on page 7-3
- "VAR Model Forecasting, Simulation, and Analysis" on page 7-39

Multivariate Time Series Models with Regression Terms

In this section...

Incorporate feedback from exogenous predictors, or study their linear associations to the response series by including a regression component in multivariate time series models. By order of increasing complexity, examples of multivariate, time series, regression models include:

- Modeling the effects of an intervention or to include shared intercepts among several responses. In these cases, the exogenous series are indicator variables.
- Modeling the contemporaneous linear associations between a subset of exogenous series to each response. Applications include CAPM analysis and studying the effects of prices of items on their demand. These applications are examples of seemingly unrelated regression (SUR). For more details, see "Implement Seemingly Unrelated Regression Analyses" on page 7-64 and "Estimate the Capital Asset Pricing Model Using SUR" on page 7-74.
- Modeling the linear associations between contemporaneous and lagged, exogenous series and the response as part of a multivariate, distributed lag model. Applications include determining how a change in monetary growth affects real gross domestic product (GDP) and gross national income (GNI).
- Any combination of SUR and the distributed lag model that includes the lagged effects of responses, also known as simultaneous equation models. VARMAX modeling is an example (see "Types of VAR Models" on page 7-3).

The general equation for a multivariate, time series, regression model is

$$\boldsymbol{y}_t = a + X_t \cdot b + \sum_{i=1}^{p} A_i \boldsymbol{y}_{t-i} + \sum_{j=1}^{q} B_j \varepsilon_{t-j} + \varepsilon_t,$$

where, in particular,

- X_t is an n-by-r design matrix.

  - Row j of X_t contains the observations of the regression variables that correspond to the period t observation of response series j.
  - Column k of X_t corresponds to the period t observations of regression variable k. (There are r regression variables composed from the exogenous series. For details, see "Design Matrix Structure for Including Exogenous Data" on page 7-58.)
  - X_t can contain lagged exogenous series.
- b is an r-by-1 vector of regression coefficients corresponding to the r regression variables. The column entries of X_t share a common regression coefficient for all t. That is, the regression component of the response series ($y_t = [y_{1t},y_{2t},...,y_{nt}]'$) for period t is

$$\begin{bmatrix} X(1,1)_t b_1 + \cdots + X(1,r)_t b_r \\ X(2,1)_t b_1 + \cdots + X(2,r)_t b_r \\ \vdots \\ X(n,1)_t b_1 + \cdots + X(n,r)_t b_r \end{bmatrix}.$$

- a is an n-by-1 vector of intercepts corresponding to the n response series.

Design Matrix Structure for Including Exogenous Data

Overview

For maximum flexibility, construct a design matrix that linearly associates the exogenous series to each response series. It helps to think of the design matrix as a vector of T smaller, block design matrices. The rows of block design matrix t correspond to observation t of the response series, and the columns correspond to the regression coefficients of the regression variables.

`vgxvarx` estimates the regression component of multivariate time series models using the Statistics and Machine Learning Toolbox function `mvregress`. Therefore, you must pass the design matrix as a T-by-1 cell vector, where cell t is the n-by-r numeric, block, design matrix at period t, n is the number of response series, and r is the number of *regression variables* in the design. That is, the structure of the entire design matrix is

$$\text{Regression variables } X_t$$
$$\left\{ \begin{bmatrix} X(1,1)_1 & \cdots & X(1,r)_1 \\ \vdots & \ddots & \vdots \\ X(n,1)_1 & \cdots & X(n,r)_1 \end{bmatrix} \begin{bmatrix} X(1,1)_2 & \cdots & X(1,r)_2 \\ \vdots & \ddots & \vdots \\ X(n,1)_2 & \cdots & X(n,r)_2 \end{bmatrix} \vdots \begin{bmatrix} X(1,1)_T & \cdots & X(1,r)_T \\ \vdots & \ddots & \vdots \\ X(n,1)_T & \cdots & X(n,r)_T \end{bmatrix} \right\}$$

At each time t, the n-by-r matrix X_t multiplies the r-by-1 vector b, yielding an n-by-1 vector of linear combinations. This setup implies suggests that:

- The number of regression variables might differ from the number of exogenous series. That is, you can associate different sets of exogenous series among response series.
- Each block design matrix in the cell vector must have the same dimensionality. That is, the multivariate time series framework does not accommodate time-varying models. The state-space framework does accommodate time-varying, multivariate time series models. For details, see ssm.

`vgxinfer`, `vgxpred`, `vgxproc`, and `vgxsim` accommodate multiple response paths. You can associate a common design matrix for all response paths by passing in a cell vector of design matrices. You can also associate a different design matrix to each response path by passing in a T-by-M cell matrix of design matrices, where M is the number of response paths and cell (t,m) is an n-by-r numeric, design matrix at period t (denoted $X_t^{(m)}$). That is, the structure of the entire design matrix for all paths is

$$\begin{array}{ccccc} & \text{Path 1} & & \cdots & \text{Path } M \\ \left\{ \right. & \begin{bmatrix} X(1,1)_1^{(1)} & \cdots & X(1,r)_1^{(1)} \\ \vdots & \ddots & \vdots \\ X(n,1)_1^{(1)} & \cdots & X(n,r)_1^{(1)} \end{bmatrix}, & \cdots & , \begin{bmatrix} X(1,1)_1^{(M)} & \cdots & X(1,r)_1^{(M)} \\ \vdots & \ddots & \vdots \\ X(n,1)_1^{(M)} & \cdots & X(n,r)_1^{(M)} \end{bmatrix} \\ & \begin{bmatrix} X(1,1)_2^{(1)} & \cdots & X(1,r)_2^{(1)} \\ \vdots & \ddots & \vdots \\ X(n,1)_2^{(1)} & \cdots & X(n,r)_2^{(1)} \end{bmatrix}, & \cdots & , \begin{bmatrix} X(1,1)_2^{(M)} & \cdots & X(1,r)_2^{(M)} \\ \vdots & \ddots & \vdots \\ X(n,1)_2^{(M)} & \cdots & X(n,r)_2^{(M)} \end{bmatrix} \\ & \vdots & \ddots & & \vdots \\ & \begin{bmatrix} X(1,1)_T^{(1)} & \cdots & X(1,r)_T^{(1)} \\ \vdots & \ddots & \vdots \\ X(n,1)_T^{(1)} & \cdots & X(n,r)_T^{(1)} \end{bmatrix}, & \cdots & , \begin{bmatrix} X(1,1)_T^{(M)} & \cdots & X(1,r)_T^{(M)} \\ \vdots & \ddots & \vdots \\ X(n,1)_T^{(M)} & \cdots & X(n,r)_T^{(M)} \end{bmatrix} \\ & & & & \left. \right\}. \end{array}$$

For more details on how to structure design matrices for `mvregress`, see "Set Up Multivariate Regression Problems".

Examples of Design Matrix Structures

- Intervention model — Suppose a tariff is imposed over some time period. You suspect that this tariff affects GNP and three other economic time series. To determine the effects of the tariff, use an intervention model, where the response series are the four econometric series, and the exogenous, regression variables are indicator variables representing the presence of the tariff in the system. Here are two ways of including the exogenous tariffs.

  - Responses share a regression coefficient — Each block design matrix (or cell) consists of either `ones(4,1)` or `zeros(4,1)`, where a `1` indicates that the tariff is in the system, and `0` otherwise. That is, at period t, a cell of the entire design matrix contains one of

$$\begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \text{ or } \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

- Responses do not share regression coefficients — Each block matrix (or cell) consists of either `eye(4)` or `zeros(4)`. That is, at period t, a cell of the entire design matrix contains one of

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{ or } \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}.$$

  In this case, the sole exogenous, indicator variable expands to four regression variables. The advantage of the larger (replicated) formulation is that it allows for `vgxvarx` to estimate the influence of the tariff on each response series separately.

  The resulting estimated regression coefficient vector $\hat{b}$ can have differing values for each component. The different values reflect the different direct influences of the tariff on each time series.

Once you have the entire design matrix (denoted `Design`), you must put each block design matrix that composes `Design` into the respective cells of a T-by-1 cell vector (or cell matrix for multiple paths). To do this, use `mat2cell`. Specify to break up `Design` into `T`, `4`-by-`size(Design,2)` block design matrices using

```
DesignCell = mat2cell(Design,4*ones(T,1),size(Design,2))
```

`DesignCell` is the properly structured regression variables that you can now pass into `vgxvarx` to estimate the model parameters.

- SUR that associates all exogenous series to each response series — If the columns of a `X` are exogenous series, then, to associate all exogenous series to each response,

  1. Create the entire design matrix by expanding `X` using its Kronecker product with the n-by-n identity matrix, e.g., if there are four responses, then the entire design matrix is

     ```
     Design = kron(X,eye(4));
     ```

  2. Put each block design matrix into the cells of a T-by-1 cell vector using `mat2cell`. Each block matrix has four rows and `size(Design,2)` columns.

- Linear trend — You can model linear trends in your data by including the exogenous matrix `eye(n)*t` in cell `t` of the entire design matrix.

Estimation of Models that Include Exogenous Data

Before you estimate a multivariate, time series, regression model using `vgxvarx`, specify the number of regression variables in the created model. (For details on specifying a multivariate time series model using `vgxset`, see "Multivariate Time Series Model Creation" on page 7-14). Recall from "Design Matrix Structure for Including Exogenous Data" on page 7-58 that the number of regression variables in the model is the number of columns in each design matrix denoted *r*. You can indicate the number of regression variables several ways:

- For a new model,
  - Specify the `nX` name-value pair argument as the number of regression variables when you create the model using `vgxset`.
  - Specify the `bsolve` name-value pair argument as the logical vector `true(r,1)`

  `vgxset` creates a multivariate time series model object, and fills in the appropriate properties. In what follows, `Mdl` denotes a created multivariate time aeries model in the Workspace.
- For a model in the Workspace, set either of the `nX` or `bsolve` properties to *r* or `true(r,1)`, respectively, using dot notation, e.g., `Mdl.nX = r`.

You can also exclude a subset of regression coefficient from being estimated. For example, to exclude the first regression coefficient, set `'bsolve', [false(1);true(r-1,1)]`. Be aware that if the model is new (i.e, `Mdl.b = []`), then the software sets any coefficient it doesn't estimate to `0`. To fix a coefficient to a value:

1. Enter

   ```
   Mdl.b = ones(r,1);
   ```

2. Specify values for the elements you want to hold fixed during estimation in the `b` property. For example, to specify that the first regression coefficient should be held at `2` during estimation, enter

   ```
   Mdl.b(1) = 2;
   ```

3. Enter

   ```
   Mdl.bsolve = [false(1);true(r-1,1)];
   ```

The software does not estimate regression intercepts (α) by default. To include a different regression intercept for each response series, specify `'Constant',true` when you create

the model using `vgxset`, or set the `Constant` property of a model in the Workspace to `true` using dot notation. Alternatively, you can specify `'asolve',true(n,1)` or set the `asolve` property to `true(n,1)`. To exclude a regression intercept from estimation, follow the same steps as for excluding a regression coefficient.

To estimate the regression coefficients, pass the model, response data, and the cell vector of design matrices (see "Design Matrix Structure for Including Exogenous Data" on page 7-58) to `vgxvarx`. For details on how `vgxvarx` works when it estimates regression coefficients, see "How vgxvarx Works" on page 7-25.

Be aware that the presence of exogenous series in a multivariate time series model might destabilized the fitted model.

See Also

`mvregress` | `vgxpred` | `vgxvarx`

Related Examples

- "Fit a VAR Model" on page 7-33
- "Implement Seemingly Unrelated Regression Analyses" on page 7-64
- "Estimate the Capital Asset Pricing Model Using SUR" on page 7-74
- "Simulate Responses of Estimated VARX Model" on page 7-80

More About

- "Vector Autoregressive (VAR) Models" on page 7-3

Implement Seemingly Unrelated Regression Analyses

This example shows how to prepare exogenous data for several seemingly unrelated regression (SUR) analyses. The response and exogenous series are random paths from a standard Gaussian distribution.

In seemingly unrelated regression (SUR), each response variable is a function of a subset of the exogenous series, but not of any endogenous variable. That is, for $j = 1, .., n$ and $t = 1, ..., T$, the model for response j at period t is

$$y_{jt} = a_j + b_{j1}x_{k_1t} + b_{j2}x_{k_2t} + ... + b_{jk_j}x_{k_jt} + \varepsilon_{jt}$$

The indices of the regression coefficients and exogenous predictors indicate that:

- You can associate each response with a different subset of exogenous predictors.
- The response series might not share intercepts or regression coefficients.

SUR accommodates intra-period innovation heteroscedasticity and correlation, but inter-period innovation independence and homoscedasticity, i.e.,

$$E\left(\varepsilon_{it}\varepsilon_{js}|X\right) = \begin{cases} 0; & t \neq s,\ i \neq j \\ \sigma_{ij}; & i \neq j,\ t = s \\ \sigma_i^2 > 0; & i = j,\ t = s \end{cases} .$$

Simulate Data from the True Model

Suppose that the true model is

$$\begin{aligned} y_{1t} &= 1 + 2x_{1t} - 1.5x_{2t} + 0.5x_{3t} + 0.75x_{4t} + \varepsilon_{1t} \\ y_{2t} &= -1 + 4x_{1t} + 2.5x_{2t} - 1.75x_{3t} - 0.05x_{4t} + \varepsilon_{2t} \\ y_{3t} &= 0.5 - 2x_{1t} + 0.5x_{2t} - 1.5x_{3t} + 0.7x_{4t} + \varepsilon_{3t} \end{aligned} ,$$

where ε_{jt}, $j = 1, ..., n$ are multivariate Gaussian random variables each having mean zero and jointly having covariance matrix

$$\Sigma = \begin{bmatrix} 1 & 0.5 & -0.05 \\ 0.5 & 1 & 0.25 \\ -0.05 & 0.25 & 1 \end{bmatrix}$$

Suppose that the paths represent different econometric measurements, e.g. stock returns.

Simulate four exogenous predictor paths from the standard Gaussian distribution.

```
rng(1);   % For reproducibility
n = 3;    % Number of response series
nExo = 4; % Number of exogenous series
T = 100;
X = randn(100,nExo);
```

The multivariate time series analysis functions of Econometrics Toolbox™ require you to input the exogenous data in a `T`-by-1 cell vector. Cell t of the cell vector is a design matrix indicating the linear relationship of the exogenous variables with each response series at period t. Specifically, each design matrix in the cell array:

- Has n rows, each corresponding to a response series.
- Has $nr = 12$ columns since, in this example, all exogenous variables are in the regression component of each response series.

To create the cell vector of design matrices for this case, first expand the exogenous predictor data by finding its Kronecker product with the n-by- n identity matrix.

```
ExpandX1 = kron(X,eye(n));
r1 = size(ExpandX1,2); % Number of regression variables
```

`ExpandX1` is an nT-by- nr numeric matrix formed by multiplying each element of `X` to the n-by- n identity matrix, and then putting the product in the corresponding position of a T-by-1 block matrix of n-by- nr matrices.

Create the cell vector of design matrices by putting each consecutive n-by- nr block matrices of `ExpandX1` into the cells of a T-by-1 cell vector. Verify that one of the cells contains the expected design matrix (e.g. the third cell)).

```
CellX1 = mat2cell(ExpandX1,n*ones(T,1));
CellX1{3}
X(3,:)
```

```
ans =

  Columns 1 through 7

   -0.7585         0         0    1.9302         0         0    1.8562
         0   -0.7585         0         0    1.9302         0         0
```

```
         0         0   -0.7585         0         0    1.9302         0

  Columns 8 through 12

         0         0    1.3411         0         0
    1.8562         0         0    1.3411         0
         0    1.8562         0         0    1.3411


ans =

   -0.7585    1.9302    1.8562    1.3411
```

In period 3, all observed predictors are associated with each response series.

Create a multivariate time series model object that characterizes the true model using vgxset.

```
aTrue = [1; -1; 0.5];
bTrue = [2; 4; -2; -1.5; 2.5; 0.5; 0.5; -1.75; -1.5; 0.75; -0.05; 0.7];
InnovCov = [1 0.5 -0.05; 0.5 1 0.25; -0.05 0.25 1];
TrueMdl = vgxset('n',n,'b',bTrue,'a',aTrue,'Q',InnovCov)
Y = vgxsim(TrueMdl,100,CellX1);
```

```
TrueMdl =

      Model: 3-D VARMAX(0,0,12) with Additive Constant
          n: 3
        nAR: 0
        nMA: 0
         nX: 12
          a: [1 -1 0.5] additive constants
          b: [12x1] regression coefficients
          Q: [3x3] covariance matrix
```

SUR Using All Predictors for Each Response Series

Create a multivariate time series model suitable for SUR using `vgxset`. You must specify the number of response series (`'n'`), the number of regression variables (`'nX'`), and whether to include different regression intercepts for each response series (`'Constant'`).

```
Mdl1 = vgxset('n',n,'nX',r1,'Constant',true)
```

```
Mdl1 =

      Model: 3-D VARMAX(0,0,12) with Additive Constant
          n: 3
        nAR: 0
        nMA: 0
         nX: 12
          a: []
          b: []
          Q: []
```

`Mdl1` is a multivariate time series model object. Unlike `TrueMdl`, none of the coefficients, intercepts, and intra-period covariance matrix have values. Therefore, `Mdl1` is suitable for estimation.

Estimate the regression coefficients using vgxvarx. Extract the residuals. Display the estimated model using vgxdisp

```
[EstMdl1,~,~,W] = vgxvarx(Mdl1,Y,CellX1);
vgxdisp(EstMdl1)
```

```
  Model  : 3-D VARMAX(0,0,12) with Additive Constant
  a Constant:
        0.978981
        -1.06438
        0.453232
  b Regression Parameter:
         1.76856
         3.85757
        -2.20089
        -1.55085
         2.44067
        0.464144
         0.69588
        -1.71386
         -1.6414
        0.670357
      -0.0564374
        0.565809
  Q Innovations Covariance:
         1.38503       0.667301      -0.159136
```

```
    0.667301    0.973123    0.216492
   -0.159136    0.216492    0.993384
```

`EstMdl` is a multivariate time series model containing the estimated parameters. `W` is a T-by- n matrix of residuals. By default, `vgxvarx` models a full, intra-period innovations covariance matrix.

Alternatively, and in this case, you can use the backslash operator on `X` and `Y`. However, you must include a column of ones in `X` for the intercepts.

```
coeff = [ones(T,1) X]\Y
```

```
coeff =

    0.9790   -1.0644    0.4532
    1.7686    3.8576   -2.2009
   -1.5508    2.4407    0.4641
    0.6959   -1.7139   -1.6414
    0.6704   -0.0564    0.5658
```

`coeff` is a `nExo + 1`-by- `n` matrix of estimated regression coefficients and intercepts. The estimated intercepts are in the first row, and the rest of the matrix contains the estimated regression coefficients

Compare all estimates to their true values.

```
fprintf('\n');
fprintf('                Intercepts      \n');
fprintf('     True     |   vgxvarx   |  backslash\n');
fprintf('--------------------------------------\n');
for j = 1:n
    fprintf('  %8.4f    |  %8.4f   | %8.4f\n',aTrue(j),EstMdl1.a(j),coeff(1,j));
end

cB = coeff';
cB = cB(:);
fprintf('\n');
fprintf('               Coefficients      \n');
fprintf('     True     |   vgxvarx   |  backslash\n');
fprintf('--------------------------------------\n');
for j = 1:r1
    fprintf('  %8.4f    |  %8.4f   | %8.4f\n',bTrue(j),...
```

```
        EstMdl1.b(j),cB(n + j));
end

fprintf('\n');
fprintf('                  Innovations Covariance\n');
fprintf('            True            |             vgxvarx\n');
fprintf('------------------------------------------------------------\n');
for j = 1:n
    fprintf('%8.4f %8.4f %8.4f  |   %8.4f %8.4f %8.4f\n',...
        InnovCov(j,:),EstMdl1.Q(j,:));
end
```

```
               Intercepts
     True     |   vgxvarx   |  backslash
--------------------------------------
    1.0000    |    0.9790   |   0.9790
   -1.0000    |   -1.0644   |  -1.0644
    0.5000    |    0.4532   |   0.4532

              Coefficients
     True     |   vgxvarx   |  backslash
--------------------------------------
    2.0000    |    1.7686   |   1.7686
    4.0000    |    3.8576   |   3.8576
   -2.0000    |   -2.2009   |  -2.2009
   -1.5000    |   -1.5508   |  -1.5508
    2.5000    |    2.4407   |   2.4407
    0.5000    |    0.4641   |   0.4641
    0.5000    |    0.6959   |   0.6959
   -1.7500    |   -1.7139   |  -1.7139
   -1.5000    |   -1.6414   |  -1.6414
    0.7500    |    0.6704   |   0.6704
   -0.0500    |   -0.0564   |  -0.0564
    0.7000    |    0.5658   |   0.5658

                  Innovations Covariance
            True            |             vgxvarx
------------------------------------------------------------
  1.0000    0.5000   -0.0500  |     1.3850    0.6673   -0.1591
  0.5000    1.0000    0.2500  |     0.6673    0.9731    0.2165
 -0.0500    0.2500    1.0000  |    -0.1591    0.2165    0.9934
```

The estimates from implementing `vgxvarx` and the backslash operator are the same, and are fairly close to their corresponding true values.

One way to check the relationship strength between the predictors and responses is to compute the coefficient of determination (i.e., the fraction of variation explained by the predictors), which is

$$R^2 = 1 - \frac{\sum_j^n \hat{\sigma}_{\varepsilon j}^2}{\sum_j^n \hat{\sigma}_{Yj}^2},$$

where $\hat{\sigma}_{\varepsilon j}^2$ is the estimated variance of residual series j, and $\hat{\sigma}_{Yj}^2$ is the estimated variance of response series j.

```
R2 = 1 - sum(diag(cov(W)))/sum(diag(cov(Y)))
```

```
R2 =

    0.9118
```

The SUR model and predictor data explain approximately 91% of the variation in the response data.

SUR Using a Unique Predictor for Each Response Series

For each period t, create block design matrices such that response series j is linearly associated to predictor series j, $j = 1, ..., 3$. Put the block design matrices in cells of a T-by-1 cell vector in chronological order.

```
CellX2 = cell(T,1);
for j = 1:T
    CellX2{j} = diag(X(j,1:n));
end
r2 = size(CellX2{1},2);
```

Create a multivariate time series model by using `vgxset` and specifying the number of response series, the number of regression variables, and whether to include different regression intercepts for each response series.

```
Mdl2 = vgxset('n',n,'nX',r2,'Constant',true);
```

Estimate the regression coefficients using `vgxvarx`. Display the estimated parameters. Compute the coefficient of determination.

```
[EstMdl2,~,~,W2] = vgxvarx(Mdl2,Y,CellX2);
```

```
vgxdisp(EstMdl2)
R2 = 1 - sum(diag(cov(W2)))/sum(diag(cov(Y)))
```

```
  Model  : 3-D VARMAX(0,0,3) with Additive Constant
  a Constant:
         1.07752
        -1.43445
        0.674376
  b Regression Parameter:
         1.01491
         3.83837
        -2.71834
  Q Innovations Covariance:
         4.96205         4.91571        -1.86546
         4.91571         20.8263        -11.0945
        -1.86546        -11.0945         7.75392

R2 =

    0.1177
```

The model and predictors explain approximately 12% of the variation in the response series. This should not be surprising since the model is not the same as the response-generating process.

SUR Using a Shared Intercept for All Response Series

Create block design matrices such that each response series j is linearly associated to all predictor series j. Prepend the resulting design matrix with a vector of ones representing the common intercept.

```
ExpandX3 = [ones(T*n,1) kron(X,eye(n))];
r3 = size(ExpandX3,2);
```

Put the block design matrices into the cells of a T-by-1 cell vector in chronological order.

```
CellX3 = mat2cell(ExpandX3,n*ones(T,1));
```

Create a multivariate time series model by using `vgxset` and specifying the number of response series and the number of regression variables. By default, `vgxset` excludes regression intercepts.

```
Mdl3 = vgxset('n',n,'nX',r3);
```

Estimate the regression coefficients using `vgxvarx`. Display the estimated parameters. Compute the coefficient of determination.

```
[EstMdl3,~,~,W3] = vgxvarx(Mdl3,Y,CellX3);
vgxdisp(EstMdl3)
a = EstMdl3.b(1)
R2 = 1 - sum(diag(cov(W3)))/sum(diag(cov(Y)))
```

```
  Model  : 3-D VARMAX(0,0,13) with No Additive Constant
  b Regression Parameter:
        0.388833
         1.73468
         3.94099
        -2.20458
        -1.56878
         2.48483
        0.462187
        0.802394
        -1.97614
        -1.62978
         0.63972
       0.0190058
        0.562466
  Q Innovations Covariance:
         1.72265      -0.164059      -0.122294
       -0.164059        3.02031        0.12577
       -0.122294        0.12577       0.997404

a =

    0.3888


R2 =

    0.9099
```

The shared, estimated regression intercept is `0.389`, and the other coefficients are similar to the first SUR implementation. The model and predictors explain

approximately 91% of the variation in the response series. This should not be surprising since the model almost the same as the response-generating process.

See Also

`mvregress` | `vgxpred` | `vgxvarx`

Related Examples

- "Fit a VAR Model" on page 7-33
- "Estimate the Capital Asset Pricing Model Using SUR" on page 7-74
- "Simulate Responses of Estimated VARX Model" on page 7-80
- "VAR Model Case Study" on page 7-89

More About

- "Vector Autoregressive (VAR) Models" on page 7-3
- "Multivariate Time Series Models with Regression Terms" on page 7-57

Estimate the Capital Asset Pricing Model Using SUR

This example shows how to implement the capital asset pricing model (CAPM) using the Econometric Toolbox™ multivariate time series framework.

The CAPM model characterizes comovements between asset and market prices. Under this framework, individual asset returns are linearly associated with the return of the whole market (for details, see [52], [74], and [60]). That is, given the return series of all stocks in a market (M_t) and the return of a riskless asset (C_t), the CAPM model for return series j (R_j) is

$$R_{jt} - C_t = a_j + b_j(M_t - C_t) + \varepsilon_{jt}$$

for all assets $j = 1, ..., n$ in the market.

$a = [a_1 \ ... \ a_n]'$ is an n-by-1 vector of *asset alphas* that should be zero, and it is of interest to investigate assets whose asset alphas are significantly away from zero. $b = [b_1 \ ... \ b_n]'$ is a n-by-1 vector of *asset betas* that specify the degree of comovement between the asset being modeled and the market. An interpretation of element j of b is

- If $b_j = 1$, then asset j moves in the same direction and with the same volatility as the market, i.e., is positively correlated with the market .
- If $b_j = -1$, then asset j moves in the opposite direction, but with the same volatility as the market, i.e., is negatively correlated with the market.
- If $b_j = 0$, then asset j is uncorrelated with the market.

In general:

- $\text{sign}(b_j)$ determines the direction that the asset is moving relative to the market as described in the previous bullets.
- $|b_j|$ is the factor that determines how much more or less volatile asset j is relative to the market. For example, if $|b_j| = 10$, then asset j is 10 times as volatile as the market.

Load and Process the Data

Load the CAPM data set included in the Financial Toolbox™.

```
load CAPMuniverse
varWithNaNs = Assets(any(isnan(Data),1))
dateRange = datestr([Dates(1) Dates(end)])
```

```
varWithNaNs =

    'AMZN'    'GOOG'


dateRange =

03-Jan-2000
07-Nov-2005
```

The variable `Data` is a 1471-by-14 numeric matrix containing the daily returns of a set of 12 stocks (columns 1 through 12), one rickless asset (column 13), and the return of the whole market (column 14). The returns were measured from 03Jan2000 through 07Nov2005. `AMZN` and `GOOG` had their IPO during sampling, and so they have missing values.

Assign variables for the response and predictor series.

```
Y = bsxfun(@minus,Data(:,1:12),Data(:,14));
X = Data(:,13) - Data(:,14);
[T,n] = size(Y)
```

```
T =

        1471


n =

    12
```

`Y` is a 1471-by-12 matrix of the returns adjusted by the riskless return. `X` is a 1471-by-1 vector of the market return adjusted by the riskless return.

Create block design matrices for each period, and put them into cells of a T-by-1 cell vector. You can specify that each response series has its own intercept when you create

the multivariate time series model object. Therefore, do not consider the intercept when you create the block design matrices.

```
Design = kron(X,eye(n));
CellX = mat2cell(Design,n*ones(T,1));
nX = size(Design,2);
```

Create the Multivariate Time Series Model

Create a multivariate time series model that characterizes the CAPM model. You must specify the number of response series, whether to give each response series equation an intercept, and the number of regression variables.

```
Mdl = vgxset('n',n,'Constant',true,'nX',nX);
```

`Mdl` is a multivariate time series model object that characterizes the desired CAPM model.

Estimate the Multivariate Time Series Model

Pass the CAPM model specification (`Mdl`), the response series (`Y`), and the cell vector of block design matrices (`CellX`) to `vgxvarx`. Request to return the estimated multivariate time series model and the estimated coefficient standard errors. `vgxvarx` maximizes the likelihood using the expectation-conditional-maximization (ECM) algorithm. ECM accommodates missing response values directly (i.e., without imputation), but at the cost of computation time.

```
[EstMdl,EstCoeffSEMdl] = vgxvarx(Mdl,Y,CellX);
```

`EstMdl` and `EstCoeffSEMdl` have the same structure as `Mdl`, but `EstMdl` contains the parameter estimates and `EstCoeffSEMdl` contains the estimated standard errors of the parameter estimates. `EstCoeffSEMdl`:

- Contains the biased maximum likelihood standard errors.
- Does not include the estimated standard errors of the intra-period covariances. To include the standard errors of the intra-period covariances, specify the name-value pair `'StdErrType','all'` in `vgxvarx`.

Analyze Coefficient Estimates

Display the regression estimates, their standard errors, and their t statistics. By default, the software estimates, stores, and displays standard errors from maximum likelihood. Specify to use the unbiased least squares standard errors.

```
dispMdl = vgxset(EstMdl,'Q',[]) % Suppress printing covariance estimates
vgxdisp(dispMdl,EstCoeffSEMdl,'DoFAdj',true)
```

```
dispMdl =

      Model: 12-D VARMAX(0,0,12) with Additive Constant
          n: 12
         nAR: 0
         nMA: 0
          nX: 12
           a: [12x1] additive constants
     asolve: [12x1 logical] additive constant indicators
           b: [12x1] regression coefficients
     bsolve: [12x1 logical] regression coefficient indicators
           Q: []
     Qsolve: [12x12 logical] covariance matrix indicators

  Model  : 12-D VARMAX(0,0,12) with Additive Constant
           Standard errors with DoF adjustment (least-squares)
       Parameter          Value     Std. Error    t-Statistic
  -------------- -------------- -------------- --------------
            a(1)     0.00116454    0.000869904         1.3387
            a(2)    0.000715822     0.00121752       0.587932
            a(3)   -0.000223753    0.000806185      -0.277546
            a(4)   -2.44513e-05    0.000689289     -0.0354732
            a(5)     0.00140469     0.00101676        1.38153
            a(6)     0.00412219    0.000910392        4.52793
            a(7)    0.000116143     0.00068952       0.168441
            a(8)   -1.37697e-05    0.000456934     -0.0301351
            a(9)    0.000110279    0.000710953       0.155114
           a(10)   -0.000244727    0.000521036      -0.469693
           a(11)     3.2336e-05    0.000861501      0.0375346
           a(12)    0.000128267     0.00103773       0.123603
            b(1)         1.22939      0.0741875        16.5714
            b(2)         1.36728       0.103833        13.1681
            b(3)          1.5653      0.0687534        22.7669
            b(4)         1.25942      0.0587843        21.4245
            b(5)         1.34406      0.0867116        15.5003
            b(6)        0.617321      0.0776404        7.95103
            b(7)         1.37454      0.0588039         23.375
            b(8)         1.08069      0.0389684        27.7326
            b(9)         1.60024      0.0606318        26.3928
           b(10)          1.1765      0.0444352        26.4767
           b(11)         1.50103      0.0734709        20.4303
```

```
    b(12)        1.65432      0.0885002        18.6928
```

To determine whether the parameters are significantly away from zero, suppose that a t statistic of 3 or more indicates significance.

Response series 6 has a significant asset alpha.

```
sigASymbol = Assets(6)
```

```
sigASymbol =

    'GOOG'
```

As a result, `GOOG` has exploitable economic properties.

All asset betas are greater than 3. This indicates that all assets are significantly correlated with the market.

However, `GOOG` has an asset beta of approximately `0.62`, whereas all other asset betas are greater than 1. This indicates that the magnitude of the volatility of `GOOG` is approximately 62% of the market volatility. The reason for this is that `GOOG` steadily and almost consistently appreciated in value while the market experienced volatile horizontal movements.

For more details and an alternative analysis, see "Capital Asset Pricing Model with Missing Data".

See Also

`mvregress` | `vgxpred` | `vgxvarx`

Related Examples

- "Fit a VAR Model" on page 7-33
- "Implement Seemingly Unrelated Regression Analyses" on page 7-64
- "Simulate Responses of Estimated VARX Model" on page 7-80
- "VAR Model Case Study" on page 7-89

More About

- "Vector Autoregressive (VAR) Models" on page 7-3

- "Multivariate Time Series Models with Regression Terms" on page 7-57

Simulate Responses of Estimated VARX Model

This example shows how to estimate a multivariate time series model that contains lagged endogenous and exogenous variables, and how to simulate responses. The response series are the quarterly:

- Changes in real gross domestic product (rGDP) rates (y_{1t})
- Real money supply (rM1SL) rates (y_{2t})
- Short-term interest rates (i.e., three-month treasury bill yield, y_{3t})

from March, 1959 through March, 2009 . The exogenous series is the quarterly changes in the unemployment rates (x_t).

Suppose that a model for the responses is this VARX(4,3) model

$$\begin{bmatrix} \iota_1 \\ \iota_2 \\ \iota_3 \end{bmatrix} + \begin{bmatrix} x_t & 0 & 0 \\ 0 & x_t & 0 \\ 0 & 0 & x_t \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} + \begin{bmatrix} \phi_{11}^{(1)} & \phi_{12}^{(1)} & \phi_{13}^{(1)} \\ \phi_{21}^{(1)} & \phi_{22}^{(1)} & \phi_{23}^{(1)} \\ \phi_{31}^{(1)} & \phi_{32}^{(1)} & \phi_{33}^{(1)} \end{bmatrix} \begin{bmatrix} y_{1,t-1} \\ y_{2,t-1} \\ y_{3,t-1} \end{bmatrix} + \ldots + \begin{bmatrix} \phi_{11}^{(4)} & \phi_{12}^{(4)} & \phi_{13}^{(4)} \\ \phi_{21}^{(4)} & \phi_{22}^{(4)} & \phi_{23}^{(4)} \\ \phi_{31}^{(4)} & \phi_{32}^{(4)} & \phi_{33}^{(4)} \end{bmatrix} \begin{bmatrix} y_{1,t-} \\ y_{2,t-} \\ y_{3,t-} \end{bmatrix}$$

Preprocess the Data

Load the U.S. macroeconomic data set. Flag the series and their periods that contain missing values (indicated by `NaN` values).

```
load Data_USEconModel
varNaN = any(ismissing(DataTable),1); % Variables containing NaN values
seriesWithNaNs = series(varNaN)
```

```
seriesWithNaNs =

  Columns 1 through 3

    '(FEDFUNDS) Effec...'      '(GS10) Ten-year ...'      '(M1SL) M1 money ...'

  Columns 4 through 5

    '(M2SL) M2 money ...'      '(UNRATE) Unemplo...'
```

In this data set, the variables that contain missing values entered the sample later than the other variables. There are no missing values after sampling started for a particular variable.

`vgxvarx` accommodates missing values for responses, but not for regression variables. Flag all periods corresponding to a missing regression variable value.

```
idx = ~isnan(DataTable.UNRATE);
```

For the rest of the example, consider only those values that of the series indicated by a `true` in `idx`.

Compute rGDP and rM1SL, and the growth rates of rGDP, rM1SL, short-term interest rates, and the unemployment rate. `Description` contains a description of the data and the variable names. Reserve the last three years of data to investigate the out-of-sample performance of the estimated model.

```
rGDP = DataTable.GDP(idx)./(DataTable.GDPDEF(idx)/100);
rM1SL = DataTable.M1SL(idx)./(DataTable.GDPDEF(idx)/100);

dLRGDP = diff(log(rGDP));              % rGDP growth rate
dLRM1SL = diff(log(rM1SL));            % rM1SL growth rate
d3MTB = diff(DataTable.TB3MS(idx));    % Change in short-term interest rate (3MTB)
dUNRATE = diff(DataTable.UNRATE(idx)); % Change in unemployment rate

T = numel(d3MTB);    % Total sample size
oosT = 12;           % Out-of-sample size
estT = T - oosT;     % Estimation sample size
estIdx = 1:estT;     % Estimation sample indices
oosIdx = (T - 11):T; % Out-of-sample indices
dates = dates((end - T + 1):end);

EstY = [dLRGDP(estIdx) dLRM1SL(estIdx) d3MTB(estIdx)]; % In-sample responses
estX = dUNRATE(estIdx);                                % In-sample exogenous data
n = size(EstY,2);

OOSY = [dLRGDP(oosIdx) dLRM1SL(oosIdx) d3MTB(oosIdx)]; % Out-of-sample responses
oosX = dUNRATE(oosIdx);                                % Out-of-sample exogenous data
```

Create the Design Matrices

Create an `estT`-by-1 cell vector of block design matrices that associate the predictor series with each response such that the responses do not share a coefficient.

```
EstExpandX = kron(estX,eye(n));
EstCellX = mat2cell(EstExpandX,n*ones(estT,1));
nX = size(EstExpandX,2);
```

Specify the VARX Model

Specify a multivariate time series model object that characterizes the VARX(4,3) model using `vgxset`.

```
Mdl = vgxset('n',n,'nAR',4,'nX',nX,'Constant',true);
```

Estimate the VAR(4) Model

Estimate the parameters of the VARX(4,3) model using `vgxvarx`. Display the parameter estimates.

```
EstMdl = vgxvarx(Mdl,EstY,EstCellX);
vgxdisp(EstMdl)
```

```
  Model  : 3-D VARMAX(4,0,3) with Additive Constant
           Conditional mean is AR-stable and is MA-invertible
  a Constant:
      0.00811792
     0.000709263
       0.0465824
  b Regression Parameter:
      -0.0162116
     -0.00163933
       -1.50115
  AR(1) Autoregression Matrix:
      -0.0375772     -0.0133236      0.00108218
     -0.00519697       0.177963     -0.00501432
       -0.873992       -6.89049       -0.165888
  AR(2) Autoregression Matrix:
       0.0753033      0.0775643       -0.001049
      0.00282857        0.29064     -0.00159574
         4.00724       0.465046       -0.221024
  AR(3) Autoregression Matrix:
      -0.0927688     -0.0240239    -0.000549057
       0.0627837      0.0686179     -0.00212185
        -7.52241         10.247        0.227121
  AR(4) Autoregression Matrix:
       0.0646951     -0.0792765    -0.000176166
       0.0276958     0.00922231    -0.000183861
```

```
        1.38523       -11.8774       0.0518154
  Q Innovations Covariance:
     3.57524e-05    7.05807e-06   -4.23542e-06
     7.05807e-06    9.67992e-05    -0.00188786
    -4.23542e-06    -0.00188786       0.777151
```

`EstMdl` is a multivariate time series model object containing the estimated parameters.

Simulate Out-Of-Sample Response Paths Using the Same Exogenous Data per Path

Simulate 1000, 3 year response series paths from the estimated model assuming that the exogenous unemployment rate is a fixed series. Since the model contains 4 lags per endogenous variable, specify the last 4 observations in the estimation sample as presample data.

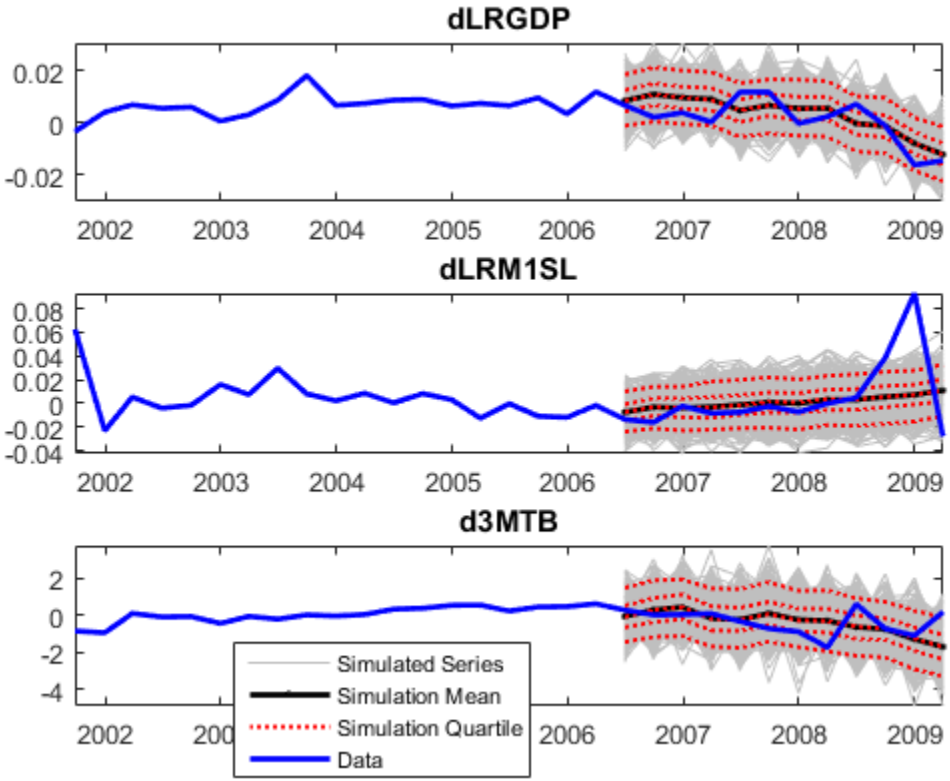
```
OOSExpandX = kron(oosX,eye(n));
OOSCellX = mat2cell(OOSExpandX,n*ones(oosT,1));
numPaths = 1000;
Y0 = EstY((end-3):end,:);
rng(1); % For reproducibility
YSim = vgxsim(EstMdl,oosT,OOSCellX,Y0,[],numPaths);
```

`YSim` is a 12-by-3-by-1000 numeric array of simulated responses. The rows of `YSim` correspond to out-of-sample periods, the columns correspond to the response series, and the leaves correspond to paths.

Plot the response data and the simulated responses. Identify the 5%, 25%, 75% and 95% percentiles, and the mean and median of the simulated series at each out-of-sample period.

```
YSimBar = mean(YSim,3);
YSimQrtl = quantile(YSim,[0.05 0.25 0.5 0.75 0.95],3);
RepDates = repmat(dates(oosIdx),1,1000);
respNames = {'dLRGDP' 'dLRM1SL' 'd3MTB'};

figure;
for j = 1:n;
    subplot(3,1,j);
    h1 = plot(dates(oosIdx),squeeze(YSim(:,j,:)),'Color',0.75*ones(3,1));
    hold on;
    h2 = plot(dates(oosIdx),YSimBar(:,j),'.-k','LineWidth',2);
    h3 = plot(dates(oosIdx),squeeze(YSimQrtl(:,j,:)),':r','LineWidth',1.5);
    h4 = plot(dates((end - 30):end),[EstY((end - 18):end,j);OOSY(:,j)],...
        'b','LineWidth',2);
```

```
    title(sprintf('%s',respNames{j}));
    datetick;
    axis tight;
    hold off;
end
legend([h1(1) h2(1) h3(1) h4],{'Simulated Series','Simulation Mean',...
    'Simulation Quartile','Data'},'Location',[0.4 0.1 0.01 0.01],...
    'FontSize',8);
```



Simulate Out-Of-Sample Response Paths Using Random Exogenous Data

Suppose that the change in the unemployment rate is an AR(4) model, and fit the model to the estimation sample data.

```
MdlUNRATE = arima('ARLags',1:4);
EstMdlUNRATE = estimate(MdlUNRATE,estX,'Display','off');
```

`EstMdlUNRATE` is an arima class model object containing the parameter estimates.

Simulate 1000, 3 year paths from the estimated AR(4) model for the change in unemployment rate. Since the model contains 4 lags, specify the last 4 observations in the estimation sample as presample data.

```
XSim = simulate(EstMdlUNRATE,oosT,'Y0',estX(end-3:end),...
    'NumPaths',numPaths);
```

`XSim` is a 12-by-1000 numeric matrix of simulated exogenous paths. The rows correspond to periods and the columns correspond to paths.

Create a cell matrix of block design matrices to organize the exogenous data, where each column corresponds to a path.

```
ExpandXSim = kron(XSim,eye(n));
size(ExpandXSim)
CellXSim = mat2cell(ExpandXSim,n*ones(oosT,1),n*ones(1,numPaths));
size(CellXSim)
CellXSim{1,1}
```

```
ans =

          36        3000


ans =

          12        1000


ans =

    0.7901         0         0
         0    0.7901         0
         0         0    0.7901
```

`ExpandXSim` is a 36-by-3000 numeric matrix, and `CellXSim` is a 12-by-1000 cell matrix of mutually exclusive, neighboring, 3-by-3 block matrices in `ExpandXSim`.

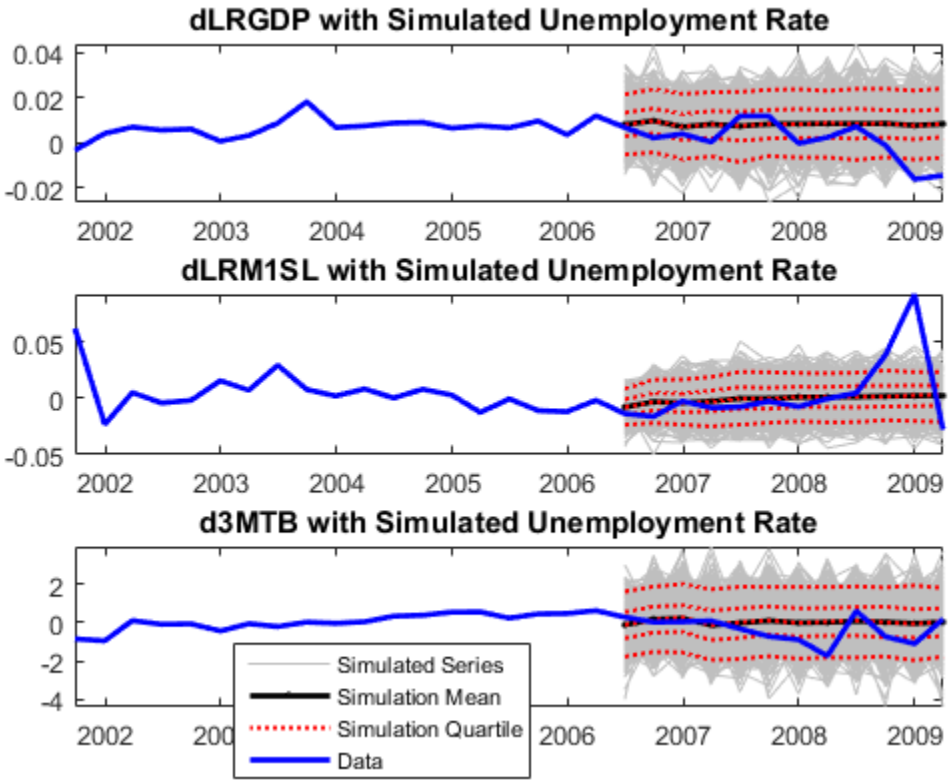
Simulate 3 years of 1000 future response series paths from the estimated model using the simulated exogenous data. Since the model contains 4 lags per endogenous variable, specify the last 4 observations in the estimation sample as presample data.

```
YSimRX = vgxsim(EstMdl,oosT,CellXSim,Y0,[],numPaths);
```

`YSimRX` is a 12-by-3-by-1000 numeric array of simulated responses.

Plot the response data and the simulated responses. Identify the 5%, 25%, 75% and 95% percentiles, and the mean and median of the simulated series at each out-of-sample period.

```
YSimBarRX = mean(YSimRX,3);
YSimQrtlRX = quantile(YSimRX,[0.05 0.25 0.5 0.75 0.95],3);

figure;
for j = 1:n;
    subplot(3,1,j);
    h1 = plot(dates(oosIdx),squeeze(YSimRX(:,j,:)),'Color',0.75*ones(3,1));
    hold on;
    h2 = plot(dates(oosIdx),YSimBarRX(:,j),'.-k','LineWidth',2);
    h3 = plot(dates(oosIdx),squeeze(YSimQrtlRX(:,j,:)),':r','LineWidth',1.5);
    h4 = plot(dates((end - 30):end),[EstY((end - 18):end,j);OOSY(:,j)],...
        'b','LineWidth',2);
    title(sprintf('%s with Simulated Unemployment Rate',respNames{j}));
    datetick;
    axis tight;
    hold off;
end
legend([h1(1) h2(1) h3(1) h4],{'Simulated Series','Simulation Mean',...
    'Simulation Quartile','Data'},'Location',[0.4 0.1 0.01 0.01],...
    'FontSize',8)
```



See Also

`mvregress` | `vgxpred` | `vgxvarx`

Related Examples

- “Fit a VAR Model” on page 7-33
- “Implement Seemingly Unrelated Regression Analyses” on page 7-64
- “Estimate the Capital Asset Pricing Model Using SUR” on page 7-74
- “VAR Model Case Study” on page 7-89

More About

- "Vector Autoregressive (VAR) Models" on page 7-3
- "Multivariate Time Series Models with Regression Terms" on page 7-57

VAR Model Case Study

This example shows how to analyze a VAR model.

Overview of Case Study

This section contains an example of the workflow described in "Building VAR Models". The example uses three time series: GDP, M1 money supply, and the 3-month T-bill rate. The example shows:

- Loading and transforming the data for stationarity
- Partitioning the transformed data into presample, estimation, and forecast intervals to support a backtesting experiment
- Making several models
- Fitting the models to the data
- Deciding which of the models is best
- Making forecasts based on the best model

Load and Transform Data

The file `Data_USEconModel` ships with Econometrics Toolbox™ software. The file contains time series from the Federal Reserve Bank of St. Louis Economics Data (FRED) database in a tabular array. This example uses three of the time series:

- GDP (`GDP`)
- M1 money supply (`M1SL`)
- 3-month T-bill rate (`TB3MS`)

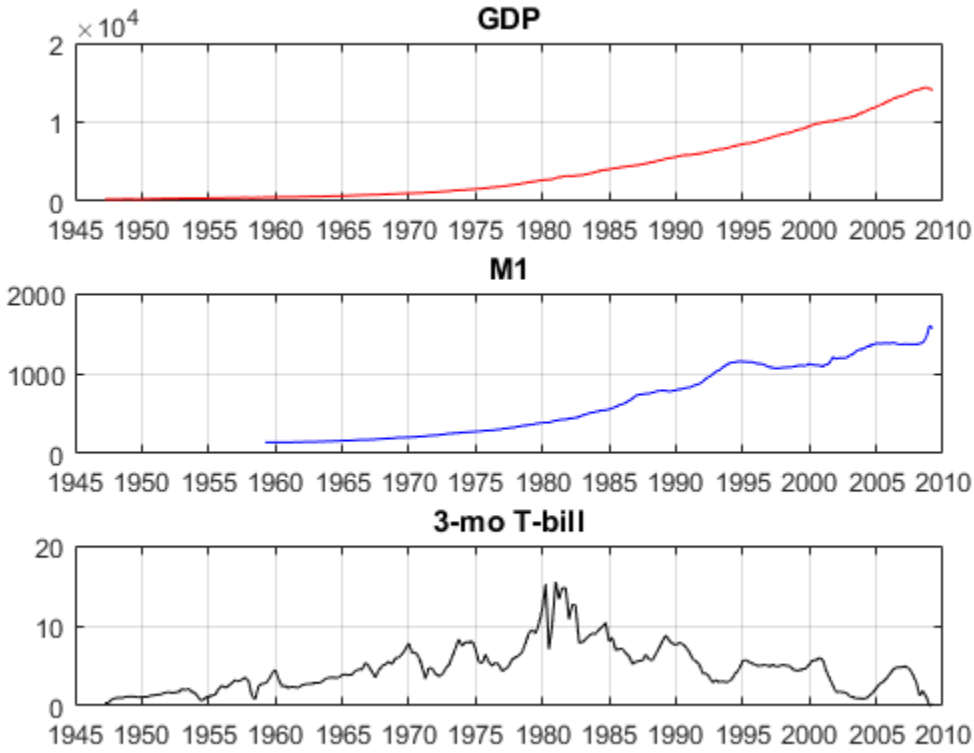
Load the data into a time series matrix `Y`.

```
load Data_USEconModel
gdp = DataTable.GDP;
m1 = DataTable.M1SL;
tb3 = DataTable.TB3MS;
Y = [gdp,m1,tb3];
```

Plot the data to look for trends.

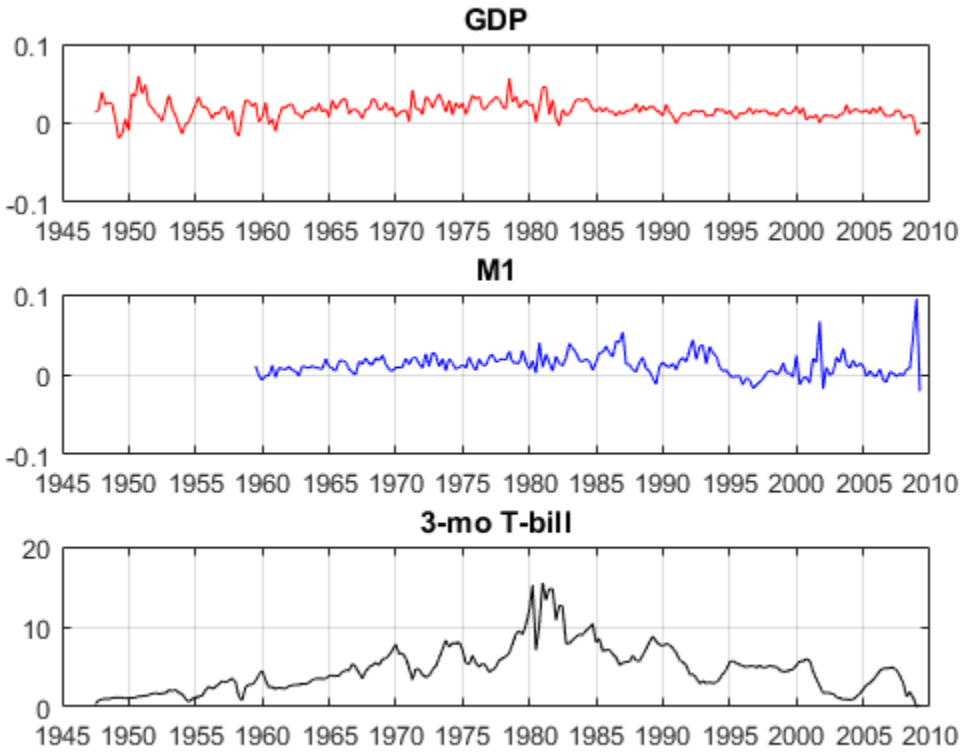
```
figure
subplot(3,1,1)
plot(dates,Y(:,1),'r');
```

```
title('GDP')
datetick('x')
grid on
subplot(3,1,2);
plot(dates,Y(:,2),'b');
title('M1')
datetick('x')
grid on
subplot(3,1,3);
plot(dates, Y(:,3), 'k')
title('3-mo T-bill')
datetick('x')
grid on
hold off
```



The GDP and M1 data appear to grow, while the T-bill returns show no long-term growth. To counter the trends in GDP and M1, take a difference of the logarithms of the data. Taking a difference shortens the time series. Therefore, truncate the T-bill series and date series `X` so that the `Y` data matrix has the same number of rows for each column.

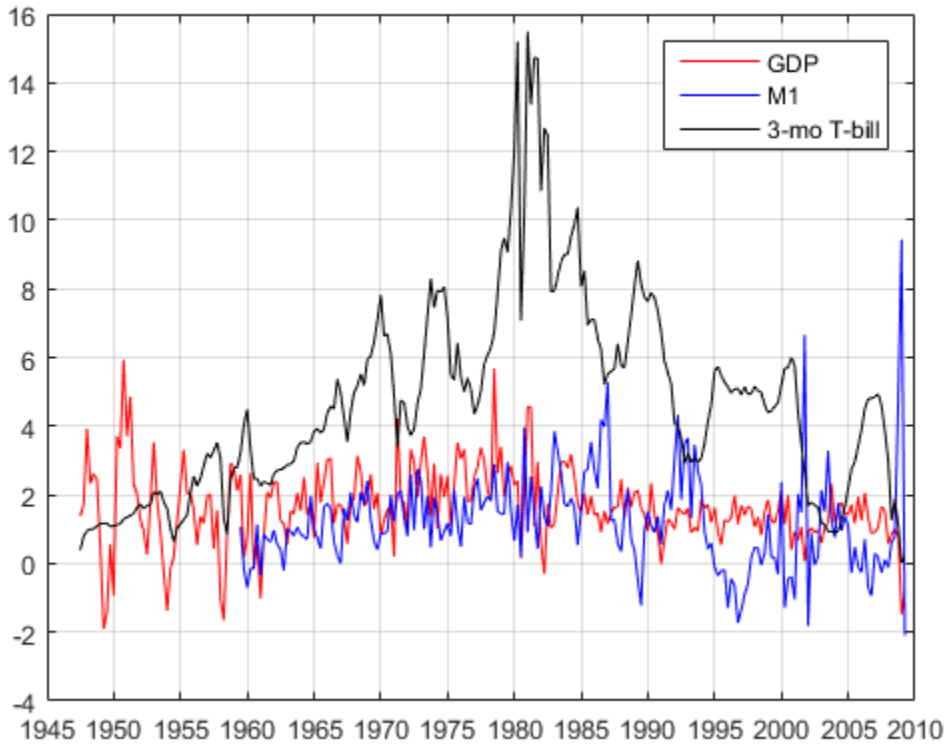
```
Y = [diff(log(Y(:,1:2))), Y(2:end,3)]; % Transformed data
X = dates(2:end);

figure
subplot(3,1,1)
plot(X,Y(:,1),'r');
title('GDP')
datetick('x')
grid on
subplot(3,1,2);
plot(X,Y(:,2),'b');
title('M1')
datetick('x')
grid on
subplot(3,1,3);
plot(X, Y(:,3),'k'),
title('3-mo T-bill')
datetick('x')
grid on
```



The scale of the first two columns is about 100 times smaller than the third. Multiply the first two columns by 100 so that the time series are all roughly on the same scale. This scaling makes it easy to plot all the series on the same plot. More importantly, this type of scaling makes optimizations more numerically stable (for example, maximizing loglikelihoods).

```
Y(:,1:2) = 100*Y(:,1:2);
figure
plot(X,Y(:,1),'r');
hold on
plot(X,Y(:,2),'b');
datetick('x')
grid on
plot(X,Y(:,3),'k');
```

```
legend('GDP','M1','3-mo T-bill');
hold off
```



Select and Fit the Models

You can select many different models for the data. This example uses four models.

- VAR(2) with diagonal autoregressive and covariance matrices
- VAR(2) with full autoregressive and covariance matrices
- VAR(4) with diagonal autoregressive and covariance matrices
- VAR(4) with full autoregressive and covariance matrices

Make all the series the same length, and transform them to be stationary and on a similar scale.

```
dGDP = 100*diff(log(gdp(49:end)));
dM1 = 100*diff(log(m1(49:end)));
dT3 = diff(tb3(49:end));
Y = [dGDP dM1 dT3];
```

Create the four models.

```
dt = logical(eye(3));
VAR2diag = vgxset('ARsolve',repmat({dt},2,1),...
    'asolve',true(3,1),'Series',{'GDP','M1','3-mo T-bill'});
VAR2full = vgxset(VAR2diag,'ARsolve',[]);
VAR4diag = vgxset(VAR2diag,'nAR',4,'ARsolve',repmat({dt},4,1));
VAR4full = vgxset(VAR2full,'nAR',4);
```

The matrix `dt` is a diagonal logical matrix. `dt` specifies that both the autoregressive matrices for `VAR2diag` and `VAR4diag` are diagonal. In contrast, the specifications for `VAR2full` and `VAR4full` have empty matrices instead of `dt`. Therefore, `vgxvarx` fits the defaults, which are full matrices for autoregressive and correlation matrices.

To assess the quality of the models, divide the response data `Y` into three periods: presample, estimation, and forecast. Fit the models to the estimation data, using the presample period to provide lagged data. Compare the predictions of the fitted models to the forecast data. The estimation period is in sample, and the forecast period is out of sample (also known as backtesting).

For the two VAR(4) models, the presample period is the first four rows of `Y`. Use the same presample period for the VAR(2) models so that all the models are fit to the same data. This is necessary for model fit comparisons. For both models, the forecast period is the final 10% of the rows of `Y`. The estimation period for the models goes from row 5 to the 90% row. Define these data periods.

```
YPre = Y(1:4,:);
T = ceil(.9*size(Y,1));
YEst = Y(5:T,:);
YF = Y((T+1):end,:);
TF = size(YF,1);
```

Now that the models and time series exist, you can easily fit the models to the data.

```
[EstSpec1,EstStdErrors1,logL1,W1] = ...
```

```
    vgxvarx(VAR2diag,YEst,[],YPre,'CovarType','Diagonal');
[EstSpec2,EstStdErrors2,logL2,W2] = ...
    vgxvarx(VAR2full,YEst,[],YPre);
[EstSpec3,EstStdErrors3,logL3,W3] = ...
    vgxvarx(VAR4diag,YEst,[],YPre,'CovarType','Diagonal');
[EstSpec4,EstStdErrors4,logL4,W4] = ...
    vgxvarx(VAR4full,YEst,[],YPre);
```

- The `EstSpec` structures are the fitted models.
- The `EstStdErrors` structures contain the standard errors of the fitted models.
- The `logL` values are the loglikelihoods of the fitted models, which you use to help select the best model.
- The `W` vectors are the estimated innovations (residuals) processes, which are the same size as `YEst`.
- The specification for `EstSpec1` and `EstSpec3` includes diagonal covariance matrices.

Check Model Adequacy

You can check whether the estimated models are stable and invertible using the `vgxqual` function. (There are no MA terms in these models, so the models are necessarily invertible.) The test shows that all the estimated models are stable.

```
[isStable1,isInvertible1] = vgxqual(EstSpec1);
[isStable2,isInvertible2] = vgxqual(EstSpec2);
[isStable3,isInvertible3] = vgxqual(EstSpec3);
[isStable4,isInvertible4] = vgxqual(EstSpec4);
[isStable1,isStable2,isStable3,isStable4]
```

```
ans =

     1     1     1     1
```

You can also look at the estimated specification structures. Each contains a line stating whether the model is stable.

```
EstSpec4
```

```
EstSpec4 =

      Model: 3-D VAR(4) with Additive Constant
```

```
    Series: {'GDP' 'M1' '3-mo T-bill'}
         n: 3
       nAR: 4
       nMA: 0
        nX: 0
         a: [0.524224 0.106746 -0.671714] additive constants
    asolve: [1 1 1 logical] additive constant indicators
        AR: {4x1 cell} stable autoregressive process
   ARsolve: {4x1 cell of logicals} autoregressive lag indicators
         Q: [3x3] covariance matrix
    Qsolve: [3x3 logical] covariance matrix indicators
```

`AR: {4x1 cell} stable autoregressive process` appears in the output indicating that the autoregressive process is stable.

You can compare the restricted (diagonal) AR models to their unrestricted (full) counterparts using `lratiotest`. The test rejects or fails to reject the hypothesis that the restricted models are adequate, with a default 5% tolerance. This is an in-sample test.

Apply the likelihood ratio tests by counting the parameters in the models using `vgxcount`, and then use `lratiotest` to perform the tests.

```
[n1,n1p] = vgxcount(EstSpec1);
[n2,n2p] = vgxcount(EstSpec2);
[n3,n3p] = vgxcount(EstSpec3);
[n4,n4p] = vgxcount(EstSpec4);
reject1 = lratiotest(logL2,logL1,n2p - n1p)
reject3 = lratiotest(logL4,logL3,n4p - n3p)
reject4 = lratiotest(logL4,logL2,n4p - n2p)
```

```
reject1 =

     1


reject3 =

     1


reject4 =
```

```
     0
```

The 1 results indicate that the likelihood ratio test rejected both the restricted models, AR(1) and AR(3), in favor of the corresponding unrestricted models. Therefore, based on this test, the unrestricted AR(2) and AR(4) models are preferable. However, the test does not reject the unrestricted AR(2) model in favor of the unrestricted AR(4) model. (This test regards the AR(2) model as an AR(4) model with restrictions that the autoregression matrices AR(3) and AR(4) are 0.) Therefore, it seems that the unrestricted AR(2) model is the best model.

To find the best model in a set, minimize the Akaike information criterion (AIC). Use in-sample data to compute the AIC. Calculate the criterion for the four models.

```
AIC = aicbic([logL1 logL2 logL3 logL4],[n1p n2p n3p n4p])
```

```
AIC =

   1.0e+03 *

    1.5129    1.4462    1.5122    1.4628
```

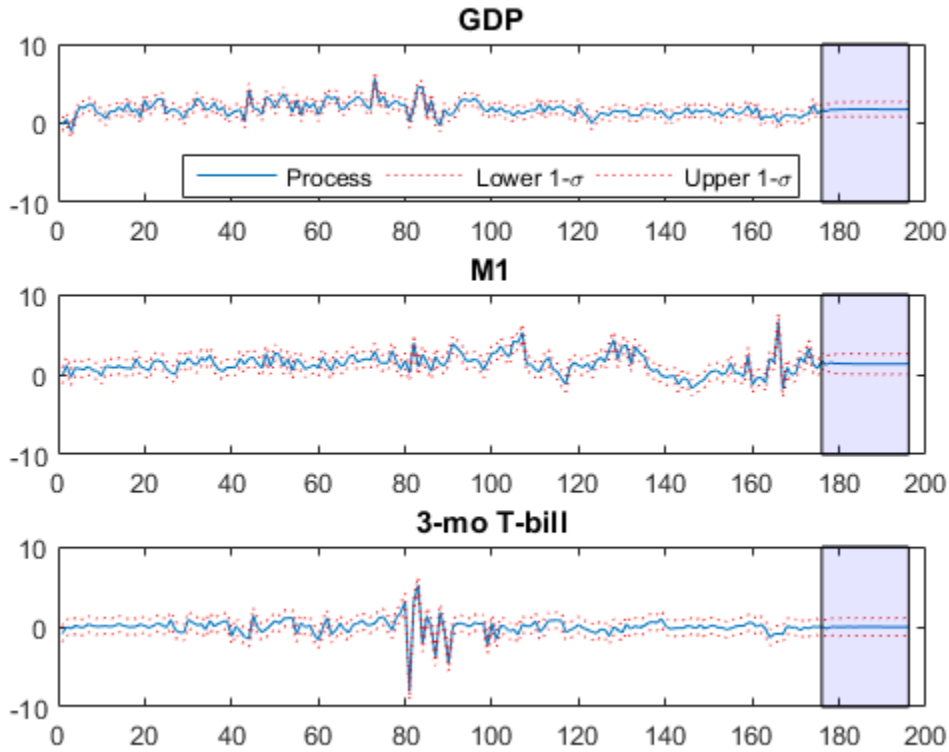
The best model according to this criterion is the unrestricted AR(2) model. Notice, too, that the unrestricted AR(4) model has lower Akaike information than either of the restricted models. Based on this criterion, the unrestricted AR(2) model is best, with the unrestricted AR(4) model coming next in preference.

To compare the predictions of the four models against the forecast data YF, use vgxpred. This function returns both a prediction of the mean time series, and an error covariance matrix that gives confidence intervals about the means. This is an out-of-sample calculation.

```
[FY1,FYCov1] = vgxpred(EstSpec1,TF,[],YEst);
[FY2,FYCov2] = vgxpred(EstSpec2,TF,[],YEst);
[FY3,FYCov3] = vgxpred(EstSpec3,TF,[],YEst);
[FY4,FYCov4] = vgxpred(EstSpec4,TF,[],YEst);
```

This plot shows the predictions in the shaded region to the right.

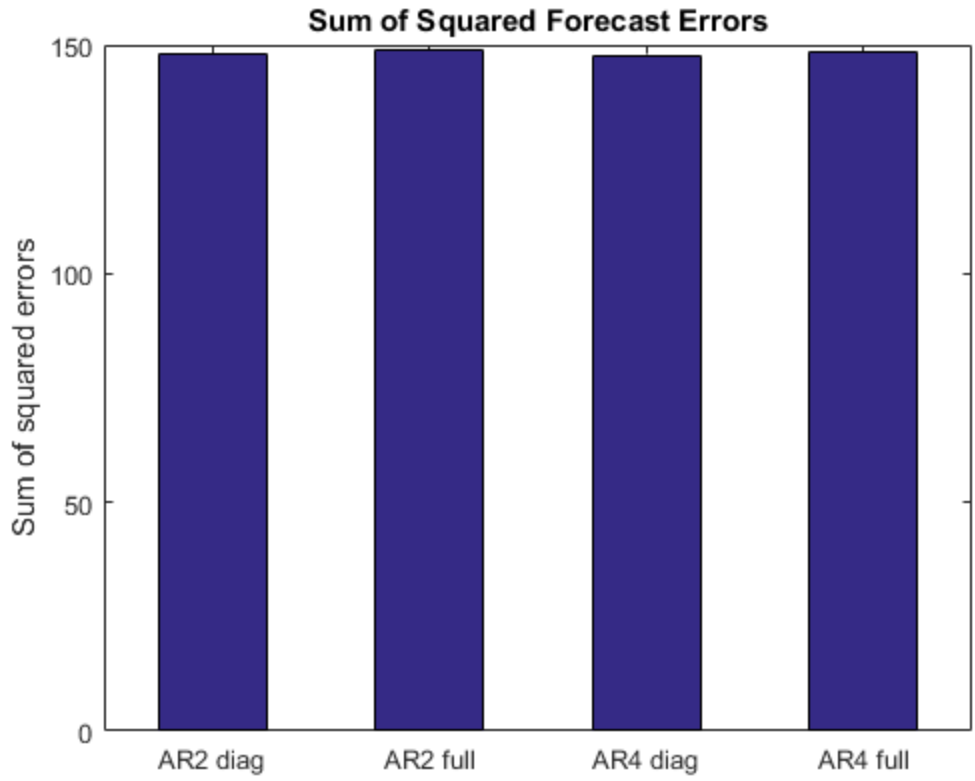
```
figure
vgxplot(EstSpec2,YEst,FY2,FYCov2)
```



It is now straightforward to calculate the sum-of-squares error between the predictions and the data, YF.

```
Error1 = YF - FY1;
Error2 = YF - FY2;
Error3 = YF - FY3;
Error4 = YF - FY4;

SSerror1 = Error1(:)' * Error1(:);
SSerror2 = Error2(:)' * Error2(:);
SSerror3 = Error3(:)' * Error3(:);
SSerror4 = Error4(:)' * Error4(:);
figure
bar([SSerror1 SSerror2 SSerror3 SSerror4],.5)
```

```
ylabel('Sum of squared errors')
set(gca,'XTickLabel',...
    {'AR2 diag' 'AR2 full' 'AR4 diag' 'AR4 full'})
title('Sum of Squared Forecast Errors')
```



The predictive performances of the four models are similar.

The full AR(2) model seems to be the best and most parsimonious fit. Its model parameters are as follows.

```
vgxdisp(EstSpec2)
```

```
  Model  : 3-D VAR(2) with Additive Constant
           Conditional mean is AR-stable and is MA-invertible
  Series : GDP
```

```
Series : M1
Series : 3-mo T-bill
a Constant:
      0.687401
        0.3856
     -0.915879
AR(1) Autoregression Matrix:
      0.272374     -0.0162214      0.0928186
     0.0563884       0.240527      -0.389905
      0.280759     -0.0712716       -0.32747
AR(2) Autoregression Matrix:
      0.242554       0.140464      -0.177957
    0.00130726       0.380042     -0.0484981
      0.260414       0.024308       -0.43541
Q Innovations Covariance:
      0.632182       0.105925       0.216806
      0.105925       0.991607      -0.155881
      0.216806      -0.155881         1.00082
```

Forecast Observations

You can make predictions or forecasts using the fitted model (`EstSpec4`) either by:

- Running `vgxpred` based on the last few rows of `YF`
- Simulating several time series with `vgxsim`

In both cases, transform the forecasts so they are directly comparable to the original time series.

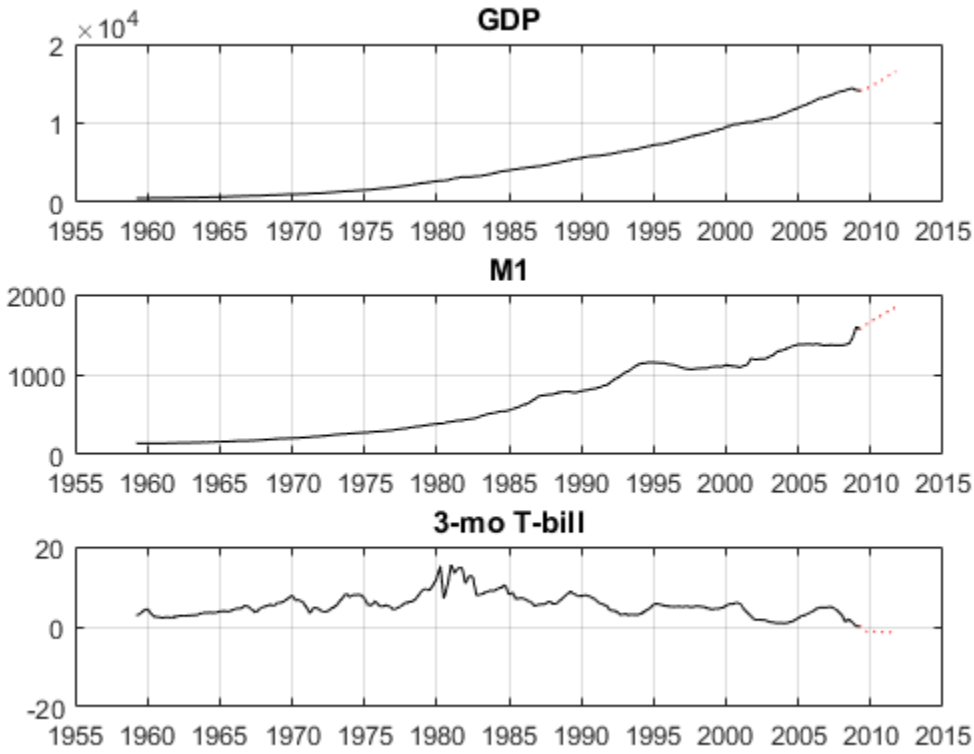
Generate 10 predictions from the fitted model beginning at the latest times using `vgxpred`.

```
[YPred,YCov] = vgxpred(EstSpec2,10,[],YF);
```

Transform the predictions by undoing the scaling and differencing applied to the original data. Make sure to insert the last observation at the beginning of the time series before using `cumsum` to undo the differencing. And, since differencing occurred after taking logarithms, insert the logarithm before using `cumsum`.

```
YFirst = [gdp,m1,tb3];
YFirst = YFirst(49:end,:);           % Remove NaNs
dates = dates(49:end);
EndPt = YFirst(end,:);
EndPt(1:2) = log(EndPt(1:2));
YPred(:,1:2) = YPred(:,1:2)/100;     % Rescale percentage
```

```
YPred = [EndPt; YPred];              % Prepare for cumsum
YPred(:,1:3) = cumsum(YPred(:,1:3));
YPred(:,1:2) = exp(YPred(:,1:2));
lastime = dates(end);
timess = lastime:91:lastime+910;     % Insert forecast horizon

figure
subplot(3,1,1)
plot(timess,YPred(:,1),':r')
hold on
plot(dates,YFirst(:,1),'k')
datetick('x')
grid on
title('GDP')
subplot(3,1,2);
plot(timess,YPred(:,2),':r')
hold on
plot(dates,YFirst(:,2),'k')
datetick('x')
grid on
title('M1')
subplot(3,1,3);
plot(timess,YPred(:,3),':r')
hold on
plot(dates,YFirst(:,3),'k')
datetick('x')
grid on
title('3-mo T-bill')
hold off
```

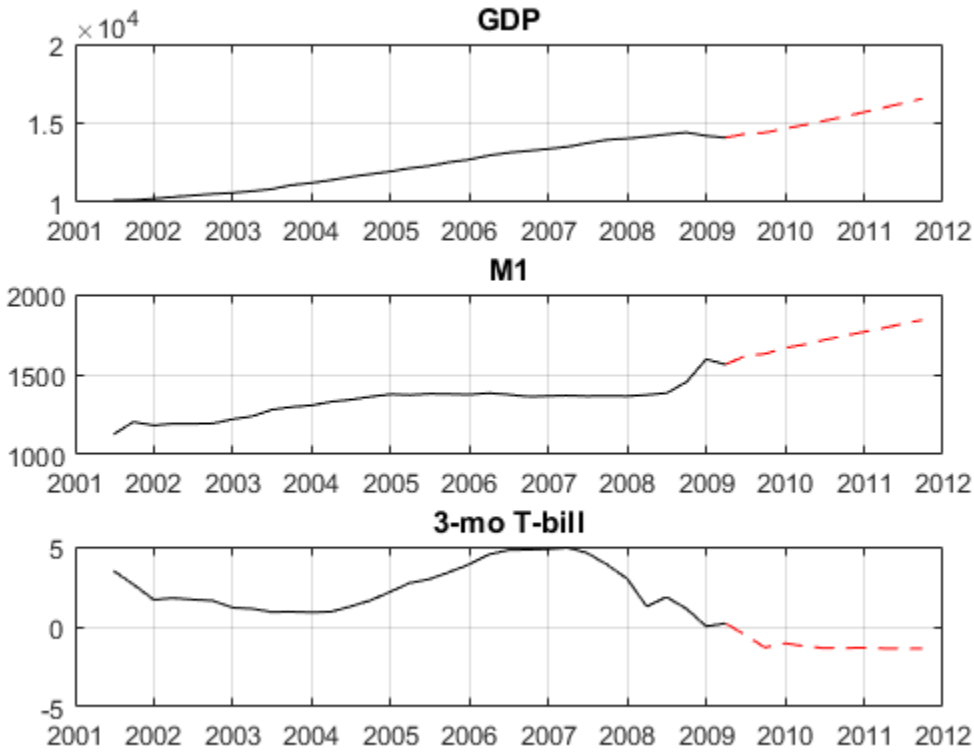


The plots show the extrapolations in dotted red, and the original data series in solid black.

Look at the last few years in this plot to get a sense of how the predictions relate to the latest data points.

```
YLast = YFirst(170:end,:);
timeslast = dates(170:end);

figure
subplot(3,1,1)
plot(timess,YPred(:,1),'--r')
hold on
plot(timeslast,YLast(:,1),'k')
```

```
datetick('x')
grid on
title('GDP')
subplot(3,1,2);
plot(timess,YPred(:,2),'--r')
hold on
plot(timeslast,YLast(:,2),'k')
datetick('x')
grid on
title('M1')
subplot(3,1,3);
plot(timess,YPred(:,3),'--r')
hold on
plot(timeslast,YLast(:,3),'k')
datetick('x')
grid on
title('3-mo T-bill')
hold off
```



The forecast shows increasing GDP, little growth in M1, and a slight decline in the interest rate. However, the forecast has no error bars.

Alternatively, you can generate 10 predictions from the fitted model beginning at the latest times using `vgxsim`. This method simulates 2000 time series times, and then generates the means and standard deviations for each period. The means of the deviates for each period are the predictions for that period.

Simulate a time series from the fitted model beginning at the latest times.

```
rng(1); % For reproducibility
YSim = vgxsim(EstSpec2,10,[],YF,[],2000);
```

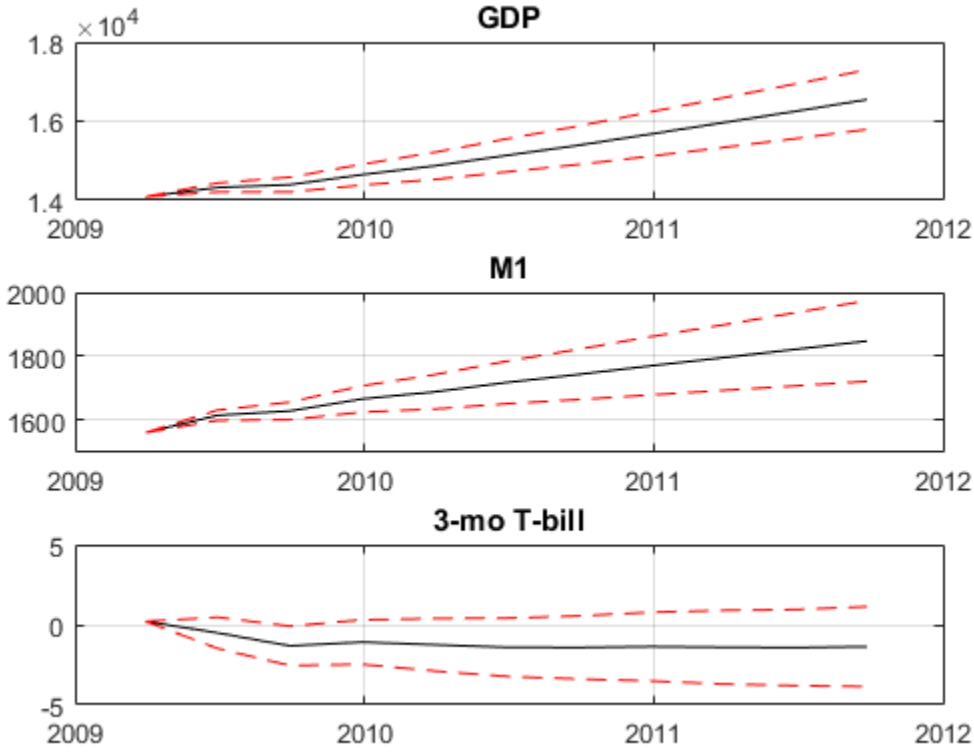
Transform the predictions by undoing the scaling and differencing applied to the original data. Make sure to insert the last observation at the beginning of the time series before using `cumsum` to undo the differencing. And, since differencing occurred after taking logarithms, insert the logarithm before using `cumsum`.

```
YFirst = [gdp,m1,tb3];
EndPt = YFirst(end,:);
EndPt(1:2) = log(EndPt(1:2));
YSim(:,1:2,:) = YSim(:,1:2,:)/100;
YSim = [repmat(EndPt,[1,1,2000]);YSim];
YSim(:,1:3,:) = cumsum(YSim(:,1:3,:));
YSim(:,1:2,:) = exp(YSim(:,1:2,:));
```

Compute the mean and standard deviation of each series, and plot the results. The plot has the mean in black, with a +/- 1 standard deviation in red.

```
YMean = mean(YSim,3);
YSTD = std(YSim,0,3);

figure
subplot(3,1,1)
plot(timess,YMean(:,1),'k')
datetick('x')
grid on
hold on
plot(timess,YMean(:,1)+YSTD(:,1),'--r')
plot(timess,YMean(:,1)-YSTD(:,1),'--r')
title('GDP')
subplot(3,1,2);
plot(timess,YMean(:,2),'k')
hold on
datetick('x')
grid on
plot(timess,YMean(:,2)+YSTD(:,2),'--r')
plot(timess,YMean(:,2)-YSTD(:,2),'--r')
title('M1')
subplot(3,1,3);
plot(timess,YMean(:,3),'k')
hold on
datetick('x')
grid on
plot(timess,YMean(:,3)+YSTD(:,3),'--r')
plot(timess,YMean(:,3)-YSTD(:,3),'--r')
title('3-mo T-bill')
```

```
hold off
```



The plots show increasing growth in GDP, moderate to little growth in M1, and uncertainty about the direction of T-bill rates.

See Also

`aicbic` | `lratiotest` | `vgxpred` | `vgxset` | `vgxsim` | `vgxvarx`

Related Examples

- “Fit a VAR Model” on page 7-33
- “Forecast a VAR Model” on page 7-50

- "Forecast a VAR Model Using Monte Carlo Simulation" on page 7-53

More About

- "Vector Autoregressive (VAR) Models" on page 7-3
- "Multivariate Time Series Model Creation" on page 7-14
- "VAR Model Estimation" on page 7-22
- "VAR Model Forecasting, Simulation, and Analysis" on page 7-39
- "Multivariate Time Series Models with Regression Terms" on page 7-57

Cointegration and Error Correction Analysis

In this section...

Integration and Cointegration

A univariate time series y_t is *integrated* if it can be brought to stationarity through differencing. The number of differences required to achieve stationarity is called the *order of integration*. Time series of order d are denoted $I(d)$. Stationary series are denoted $I(0)$.

An n-dimensional time series y_t is *cointegrated* if some linear combination $\beta_1 y_{1t} + \ldots + \beta_n y_{nt}$ of the component variables is stationary. The combination is called a *cointegrating relation*, and the coefficients $\beta = (\beta_1, \ldots, \beta_n)'$ form a *cointegrating vector*. Cointegration is usually associated with systems of $I(1)$ variables, since any $I(0)$ variables are trivially cointegrated with other variables using a vector with coefficient 1 on the $I(0)$ component and coefficient 0 on the other components. The idea of cointegration can be generalized to systems of higher-order variables if a linear combination reduces their common order of integration.

Cointegration is distinguished from traditional economic equilibrium, in which a balance of forces produces stable long-term levels in the variables. Cointegrated variables are generally unstable in their levels, but exhibit mean-reverting "spreads" (generalized by the cointegrating relation) that force the variables to move around common stochastic trends. Cointegration is also distinguished from the short-term synchronies of positive covariance, which only measures the tendency to move together at each time step. Modification of the VAR model to include cointegrated variables balances the short-term dynamics of the system with long-term tendencies.

Cointegration and Error Correction

The tendency of cointegrated variables to revert to common stochastic trends is expressed in terms of *error-correction*. If y_t is an n-dimensional time series and β is a cointegrating

vector, then the combination $\beta' y_{t-1}$ measures the "error" in the data (the deviation from the stationary mean) at time $t-1$. The rate at which series "correct" from disequilibrium is represented by a vector α of *adjustment speeds*, which are incorporated into the VAR model at time t through a multiplicative *error-correction term* $\alpha\beta' y_{t-1}$.

In general, there may be multiple cointegrating relations among the variables in y_t, in which case the vectors α and β become matrices A and B, with each column of B representing a specific relation. The error-correction term becomes $AB' y_{t-1} = C y_{t-1}$. Adding the error-correction term to a VAR model in differences produces the *vector error-correction* (*VEC*) *model*:

$$\Delta y_t = C y_{t-1} + \sum_{i=1}^{q} B_i \Delta y_{t-i} + \varepsilon_t.$$

If the variables in y_t are all $I(1)$, the terms involving differences are stationary, leaving only the error-correction term to introduce long-term stochastic trends. The rank of the *impact matrix* C determines the long-term dynamics. If C has full rank, the system y_t is stationary in levels. If C has rank 0, the error-correction term disappears, and the system is stationary in differences. These two extremes correspond to standard choices in univariate modeling. In the multivariate case, however, there are intermediate choices, corresponding to *reduced ranks* between 0 and n. If C is restricted to reduced rank r, then C factors into (nonunique) n-by-r matrices A and B with $C = AB'$, and there are r independent cointegrating relations among the variables in y_t.

By collecting differences, a VEC(q) model can be converted to a VAR(p) model in levels, with $p = q+1$:

$$y_t = A_1 y_{t-1} + \ldots + A_p y_{t-p} + \varepsilon_t.$$

Conversion between VEC(q) and VAR(p) representations of an n-dimensional system are carried out by the functions `vec2var` and `var2vec` using the formulas:

$$\left.\begin{aligned} A_1 &= C + I_n + B_1 \\ A_i &= B_i - B_{i-1},\ i = 2,\ldots,q \\ A_p &= -B_q \end{aligned}\right\} \text{VEC}(q) \text{ to VAR}(p = q+1) \text{ (using } \texttt{vec2var}\text{)}$$

$$\left.\begin{aligned} C &= \sum_{i=1}^{p} A_i - I_n \\ B_i &= -\sum_{j=i+1}^{p} A_j \end{aligned}\right\} \text{VAR}(p) \text{ to VEC}(q = p-1) \text{ (using } \texttt{var2vec}\text{)}$$

Because of the equivalence of the two representations, a VEC model with a reduced-rank error-correction coefficient is often called a *cointegrated VAR model*. In particular, cointegrated VAR models can be simulated and forecast using standard VAR techniques.

The Role of Deterministic Terms

The cointegrated VAR model is often augmented with exogenous terms Dx:

$$\Delta y_t = AB'y_{t-1} + \sum_{i=1}^{q} B_i \Delta y_{t-i} + Dx + \varepsilon_t.$$

Variables in x may include seasonal or interventional dummies, or deterministic terms representing trends in the data. Since the model is expressed in differences Δy_t, constant terms in x represent linear trends in the levels of y_t and linear terms represent quadratic trends. In contrast, constant and linear terms in the cointegrating relations have the usual interpretation as intercepts and linear trends, although restricted to the stationary variable formed by the cointegrating relation. Johansen [61] considers five cases for $AB'y_{t-1} + Dx$ which cover the majority of observed behaviors in macroeconomic systems:

Case	Form of $AB'y_{t-1} + Dx$	Model Interpretation
H2	$AB'y_{t-1}$	There are no intercepts or trends in the cointegrating relations and there are no trends in the data. This model is only appropriate if all series have zero mean.
H1*	$A(By_{t-1} + c_0)$	There are intercepts in the cointegrating relations and there are no trends in the data. This model is appropriate for nontrending data with nonzero mean.

H1	$A(B'y_{t-1}+c_0) + c_1$	There are intercepts in the cointegrating relations and there are linear trends in the data. This is a model of *deterministic cointegration*, where the cointegrating relations eliminate both stochastic and deterministic trends in the data.
H*	$A(B'y_{t-1} + c_0 + d_0t) + c_1$	There are intercepts and linear trends in the cointegrating relations and there are linear trends in the data. This is a model of *stochastic cointegration*, where the cointegrating relations eliminate stochastic but not deterministic trends in the data.
H	$A(B'y_{t-1} + c_0 + d_0t) + c_1 + d_1t$	There are intercepts and linear trends in the cointegrating relations and there are quadratic trends in the data. Unless quadratic trends are actually present in the data, this model may produce good in-sample fits but poor out-of-sample forecasts.

In Econometrics Toolbox, deterministic terms outside of the cointegrating relations, c_1 and d_1, are identified by projecting constant and linear regression coefficients, respectively, onto the orthogonal complement of A.

Cointegration Modeling

Integration and cointegration both present opportunities for transforming variables to stationarity. Integrated variables, identified by unit root and stationarity tests, can be differenced to stationarity. Cointegrated variables, identified by cointegration tests, can be combined to form new, stationary variables. In practice, it must be determined if such transformations lead to more reliable models, with variables that retain an economic interpretation.

Generalizing from the univariate case can be misleading. In the standard Box-Jenkins [15] approach to univariate ARMA modeling, stationarity is an essential assumption. Without it, the underlying distribution theory and estimation techniques become invalid. In the corresponding multivariate case, where the VAR model is unrestricted and there is no cointegration, choices are less straightforward. If the goal of a VAR analysis is to determine relationships among the original variables, differencing loses information. In this context, Sims, Stock, and Watson [97] advise against differencing, even in the

presence of unit roots. If, however, the goal is to simulate an underlying data-generating process, integrated levels data can cause a number of problems. Model specification tests lose power due to an increase in the number of estimated parameters. Other tests, such as those for Granger causality, no longer have standard distributions, and become invalid. Finally, forecasts over long time horizons suffer from inconsistent estimates, due to impulse responses that do not decay. Enders [35] discusses modeling strategies.

In the presence of cointegration, simple differencing is a model misspecification, since long-term information appears in the levels. Fortunately, the cointegrated VAR model provides intermediate options, between differences and levels, by mixing them together with the cointegrating relations. Since all terms of the cointegrated VAR model are stationary, problems with unit roots are eliminated.

Cointegration modeling is often suggested, independently, by economic theory. Examples of variables that are commonly described with a cointegrated VAR model include:

- Money stock, interest rates, income, and prices (common models of money demand)
- Investment, income, and consumption (common models of productivity)
- Consumption and long-term income expectation (Permanent Income Hypothesis)
- Exchange rates and prices in foreign and domestic markets (Purchasing Power Parity)
- Spot and forward currency exchange rates and interest rates (Covered Interest Rate Parity)
- Interest rates of different maturities (Term Structure Expectations Hypothesis)
- Interest rates and inflation (Fisher Equation)

Since these theories describe long-term equilibria among the variables, accurate estimation of cointegrated models may require large amounts of low-frequency (annual, quarterly, monthly) macroeconomic data. As a result, these models must consider the possibility of structural changes in the underlying data-generating process during the sample period.

Financial data, by contrast, is often available at high frequencies (hours, minutes, microseconds). The mean-reverting spreads of cointegrated financial series can be modeled and examined for arbitrage opportunities. For example, the Law of One Price suggests cointegration among the following groups of variables:

- Prices of assets with identical cash flows
- Prices of assets and dividends
- Spot, future, and forward prices

- Bid and ask prices

See Also

`egcitest` | `jcitest` | `jcontest`

Related Examples

- "Test for Cointegration Using the Engle-Granger Test" on page 7-121
- "Determine Cointegration Rank of VEC Model" on page 7-114
- "Estimate VEC Model Parameters Using egcitest" on page 7-126
- "Simulate and Forecast a VEC Model" on page 7-129
- "Test for Cointegration Using the Johansen Test" on page 7-144
- "Estimate VEC Model Parameters Using jcitest" on page 7-147
- "Compare Approaches to Cointegration Analysis" on page 7-150
- "Test Cointegrating Vectors" on page 7-155
- "Test Adjustment Speeds" on page 7-158

More About

- "Vector Autoregressive (VAR) Models" on page 7-3
- "Identifying Single Cointegrating Relations" on page 7-116
- "Identifying Multiple Cointegrating Relations" on page 7-143
- "Testing Cointegrating Vectors and Adjustment Speeds" on page 7-154

Determine Cointegration Rank of VEC Model

This example shows how to convert an n-dimensional VAR model to a VEC model, and then compute and interpret the cointegration rank of the resulting VEC model.

The rank of the error-correction coefficient matrix, C, determines the cointegration rank. If rank(C) is:

- Zero, then the converted VEC(p) model is a stationary VAR(p - 1) model in terms of Δy_t, without any cointegration relations.
- n, then the VAR(p) model is stable in terms of y_t.
- The integer r such that $0 < r < n$, then there are r cointegrating relations. That is, there are r linear combinations that comprise stationary series. You can factor the error-correction term into the two n-by- r matrices $C = \alpha\beta'$. α contains the adjustment speeds, and β the cointegration matrix. This factorization is not unique.

For more details, see "Cointegration and Error Correction" and [74], Chapter 6.3.

Consider the following VAR(2) model.

$$y_t = \begin{bmatrix} 1 & 0.26 & 0 \\ -0.1 & 1 & 0.35 \\ 0.12 & -0.05 & 1.15 \end{bmatrix} y_{t-1} + \begin{bmatrix} -0.2 & -0.1 & -0.1 \\ 0.6 & -0.4 & -0.1 \\ -0.02 & -0.03 & -0.1 \end{bmatrix} y_{t-2} + \varepsilon_t.$$

Create the variables `A1` and `A2` for the autoregressive coefficients. Pack the matrices into a cell vector.

```
A1 = [1 0.26 0; -0.1 1 0.35; 0.12 -0.5 1.15];
A2 = [-0.2 -0.1 -0.1; 0.6 -0.4 -0.1; -0.02 -0.03 -0.1];
Var = {A1 A2};
```

Compute the autoregressive and error-correction coefficient matrices of the equivalent VEC model.

```
[Vec,C] = var2vec(Var);
```

Because the degree of the VAR model is 2, the resulting VEC model has degree $q = 2 - 1$. Hence, `Vec` is a one-dimensional cell array containing the autoregressive coefficient matrix.

Determine the cointegration rank by computing the rank of the error-correction coefficient matrix `C`.

```
r = rank(C)
```

```
r =

     2
```

The cointegrating rank is `2`. This result suggests that there are two independent linear combinations of the three variables that are stationary.

See Also

`var2vec` | `vec2var`

Related Examples

- "Test Cointegrating Vectors" on page 7-155
- "Test Adjustment Speeds" on page 7-158

More About

- "Cointegration and Error Correction Analysis" on page 7-108
- "Identifying Single Cointegrating Relations" on page 7-116

Identifying Single Cointegrating Relations

In this section...

The Engle-Granger Test for Cointegration

Modern approaches to cointegration testing originated with Engle and Granger [37]. Their method is simple to describe: regress the first component y_{1t} of y_t on the remaining components of y_t and test the residuals for a unit root. The null hypothesis is that the series in y_t are *not* cointegrated, so if the residual test fails to find evidence against the null of a unit root, the Engle-Granger test fails to find evidence that the estimated regression relation is cointegrating. Note that you can write the regression equation as $y_{1t} - b_1 y_{2t} - \ldots - b_d y_{dt} - c_0 = \beta' y_t - c_0 = \varepsilon_t$, where $\beta = [1 \quad -b']'$ is the cointegrating vector and c_0 is the intercept. A complication of the Engle-Granger approach is that the residual series is estimated rather than observed, so the standard asymptotic distributions of conventional unit root statistics do not apply. Augmented Dickey-Fuller tests (`adftest`) and Phillips-Perron tests (`pptest`) can not be used directly. For accurate testing, distributions of the test statistics must be computed specifically for the Engle-Granger test.

The Engle-Granger test is implemented in Econometrics Toolbox by the function `egcitest`. For an example, see "Test for Cointegration Using the Engle-Granger Test" on page 7-121.

Limitations of the Engle-Granger Test

The Engle-Granger method has several limitations. First of all, it identifies only a single cointegrating relation, among what might be many such relations. This requires one of the variables, y_{1t}, to be identified as "first" among the variables in y_t. This choice, which is usually arbitrary, affects both test results and model estimation. To see this, permute the three interest rates in the Canadian data and estimate the cointegrating relation for each choice of a "first" variable.

```
load Data_Canada
```

```
Y = Data(:,3:end); % Interest rate data
PO = perms([1 2 3]);
[~,idx] = unique(PO(:,1));
    % Rows of PO with unique regressand y1
P = PO(idx,:); % Unique regressions
numPerms = size(P,1);

% Preallocate:
TO = size(Y,1);
H = zeros(1,numPerms);
PVal = zeros(1,numPerms);
CIR = zeros(TO,numPerms);

% Run all tests:
for i = 1:numPerms

    YPerm = Y(:,P(i,:));
    [h,pValue,~,~,reg] = egcitest(YPerm,'test','t2');
    H(i) = h;
    PVal(i) = pValue;
    cOi = reg.coeff(1);
    bi = reg.coeff(2:3);
    betai = [1;-bi]
    CIR(:,i) = YPerm*betai-cOi;

end

% Display the test results:
H,PVal
```

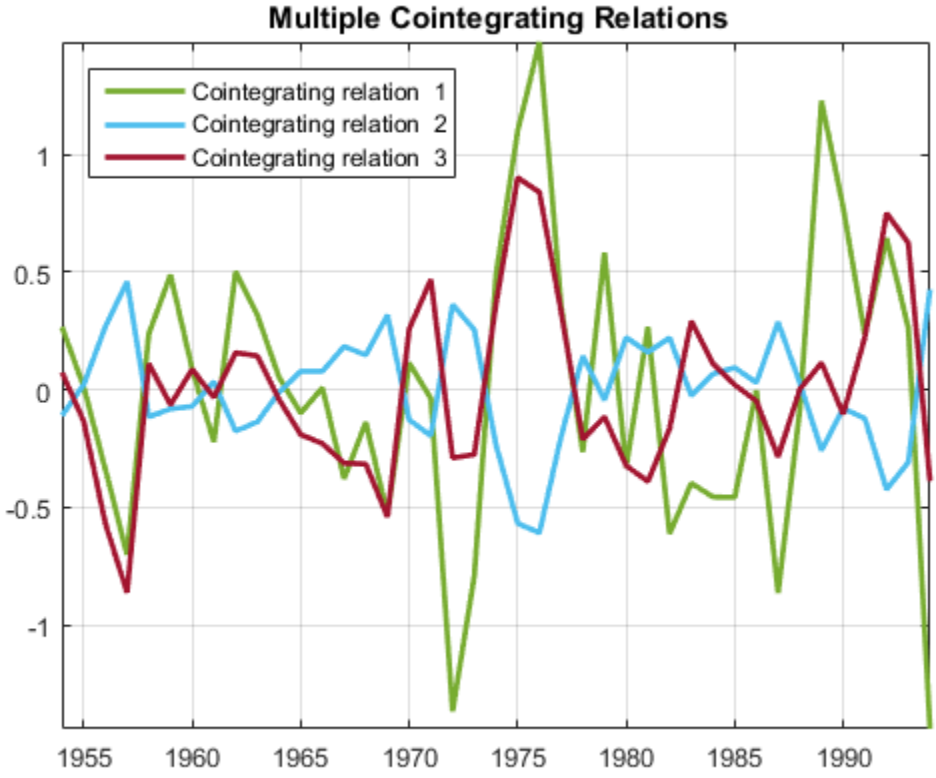
```
betai =

    1.0000
   -2.2209
    1.0718


betai =

    1.0000
   -0.6029
   -0.3472
```

```
betai =

    1.0000
   -1.4394
    0.4001


H =

     1     1     0


PVal =

    0.0202    0.0290    0.0625
```

For this data, two regressands identify cointegration while the third regressand fails to do so. Asymptotic theory indicates that the test results will be identical in large samples, but the finite-sample properties of the test make it cumbersome to draw reliable inferences.

A plot of the identified cointegrating relations shows the previous estimate (Cointegrating relation 1), plus two others. There is no guarantee, in the context of Engle-Granger estimation, that the relations are independent: Plot the cointegrating relations:

```
h = gca;
COrd = h.ColorOrder;
h.NextPlot = 'ReplaceChildren';
h.ColorOrder = circshift(COrd,3);
plot(dates,CIR,'LineWidth',2)
title('{\bf Multiple Cointegrating Relations}')
legend(strcat({'Cointegrating relation  '}, ...
     num2str((1:numPerms)')),'location','NW');
axis tight
grid on
```



Another limitation of the Engle-Granger method is that it is a two-step procedure, with one regression to estimate the residual series, and another regression to test for a unit root. Errors in the first estimation are necessarily carried into the second estimation. The estimated, rather than observed, residual series requires entirely new tables of critical values for standard unit root tests.

Finally, the Engle-Granger method estimates cointegrating relations independently of the VEC model in which they play a role. As a result, model estimation also becomes a two-step procedure. In particular, deterministic terms in the VEC model must be estimated conditionally, based on a predetermined estimate of the cointegrating

vector. For an example of VEC model parameter estimation, see “Estimate VEC Model Parameters Using egcitest”.

See Also

`egcitest`

Related Examples

- “Test for Cointegration Using the Engle-Granger Test” on page 7-121
- “Estimate VEC Model Parameters Using egcitest” on page 7-126
- “Simulate and Forecast a VEC Model” on page 7-129

More About

- “Vector Autoregressive (VAR) Models” on page 7-3
- “Cointegration and Error Correction Analysis” on page 7-108

Test for Cointegration Using the Engle-Granger Test

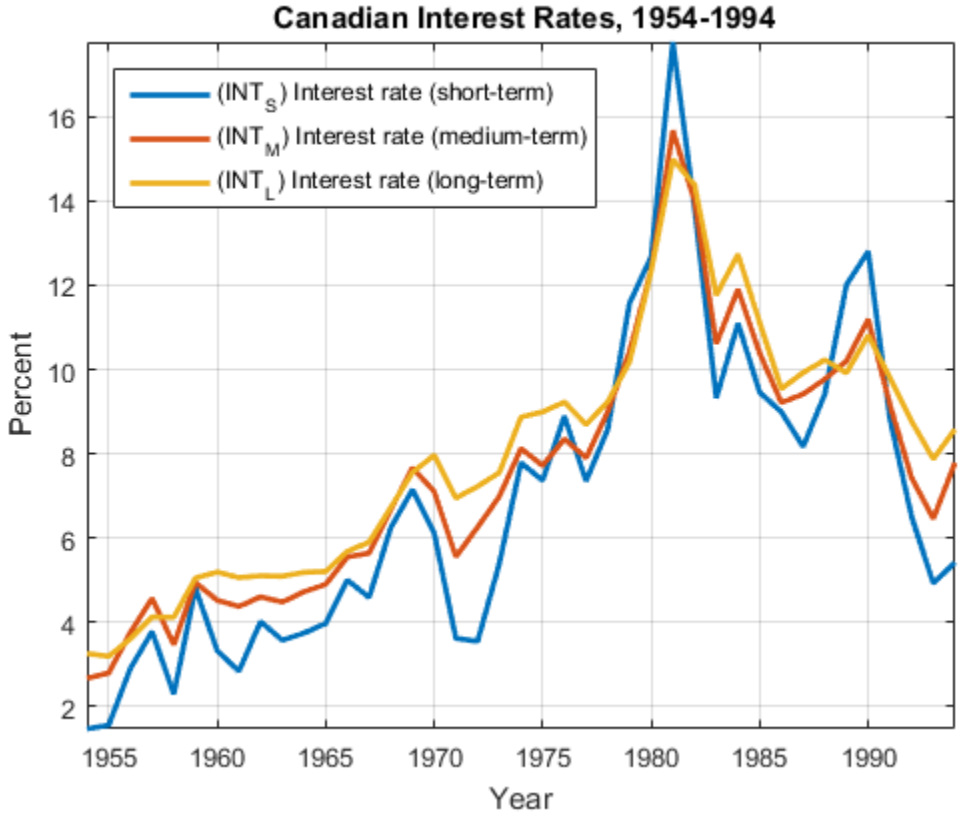
This example shows how to test the null hypothesis that there are no cointegrating relationships among the response series composing a multivariate model.

Load `Data_Canada` into the MATLAB® Workspace. The data set contains the term structure of Candian interest rates [75]. Extract the short-term, medium-term, and long-term interest rate series.

```
load Data_Canada
Y = Data(:,3:end); % Multivariate response series
```

Plot the response series.

```
figure
plot(dates,Y,'LineWidth',2)
xlabel 'Year';
ylabel 'Percent';
names = series(3:end);
legend(names,'location','NW')
title '{\bf Canadian Interest Rates, 1954-1994}';
axis tight
grid on
```



The plot shows evidence of cointegration among the three series, which move together with a mean-reverting spread.

To test for cointegration, compute both the τ (`t1`) and z (`t2`) Dickey-Fuller statistics. `egcitest` compares the test statistics to tabulated values of the Engle-Granger critical values.

```
[h,pValue,stat,cValue] = egcitest(Y,'test',{'t1','t2'})
```

```
h =

     0     1
```

```
pValue =

    0.0526    0.0202


stat =

   -3.9321  -25.4538


cValue =

   -3.9563  -22.1153
```

The τ test fails to reject the null of no cointegration, but just barely, with a p-value only slightly above the default 5% significance level, and a statistic only slightly above the left-tail critical value. The z test does reject the null of no cointegration.

The test regresses `Y(:,1)` on `Y(:,2:end)` and (by default) an intercept `c0`. The residual series is

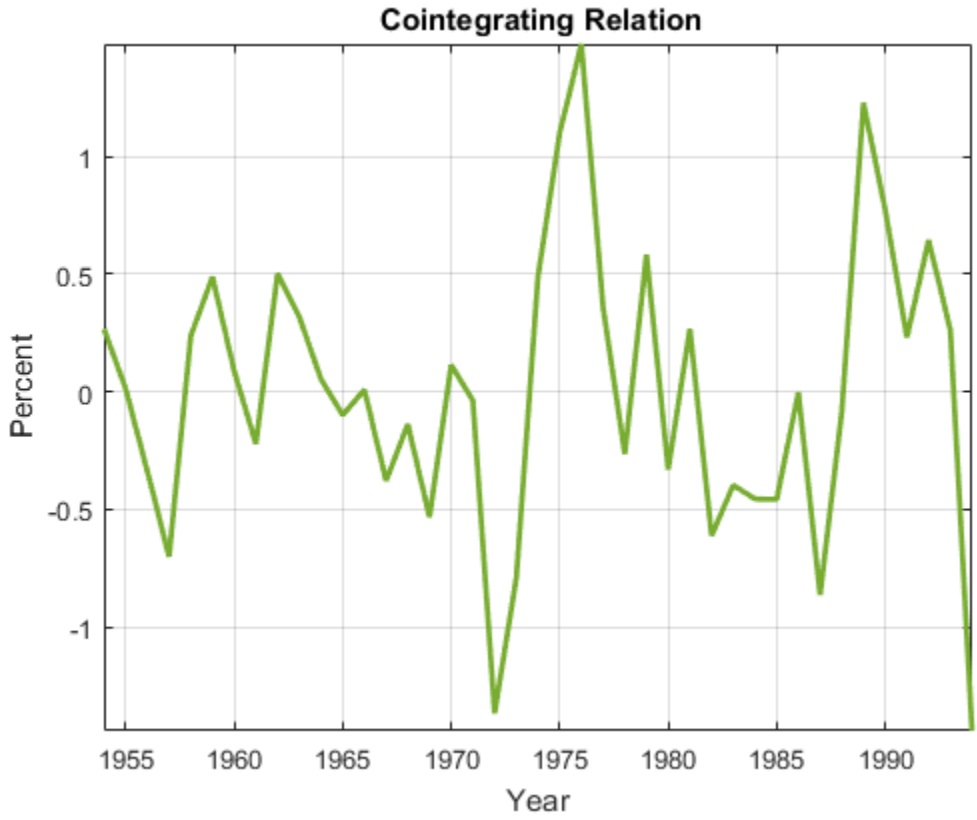
`[Y(:,1) Y(:,2:end)]*beta - c0 = Y(:,1) - Y(:,2:end)*b - c0`.

The fifth output argument of `egcitest` contains, , among other regression statistics, the regression coefficients `c0` and `b`.

Examine the regression coefficients to examine the hypothesized cointegrating vector `beta = [1; -b]`.

```
[~,~,~,~,reg] = egcitest(Y,'test','t2');

c0 = reg.coeff(1);
b = reg.coeff(2:3);
beta = [1;-b];
h = gca;
COrd = h.ColorOrder;
h.NextPlot = 'ReplaceChildren';
h.ColorOrder = circshift(COrd,3);
plot(dates,Y*beta-c0,'LineWidth',2);
title '{\bf Cointegrating Relation}';
axis tight;
legend off;
```

```
grid on;
```



The combination appears relatively stationary, as the test confirms.

See Also

`egcitest`

Related Examples

- "Estimate VEC Model Parameters Using egcitest" on page 7-126
- "Simulate and Forecast a VEC Model" on page 7-129

More About

- "Vector Autoregressive (VAR) Models" on page 7-3
- "Cointegration and Error Correction Analysis" on page 7-108
- "Identifying Single Cointegrating Relations" on page 7-116

Estimate VEC Model Parameters Using egcitest

This example shows how to estimate the parameters of a vector error-correctin (VEC) model. Before estimating VEC model parameters, you must determine whether there are any cointegrating relations (see "Test for Cointegration Using the Engle-Granger Test"). You can estimate the remaining VEC model coefficients using ordinary least squares (OLS).

Following from "Test for Cointegration Using the Engle-Granger Test", load the `Data_Canada` data set. Run the Engle-Granger cointegration test on the small-term, medium-term, and long-term interest rate series.

```
load Data_Canada
Y = Data(:,3:end); % Interest rate data
[~,~,~,~,reg] = egcitest(Y,'test','t2');
c0 = reg.coeff(1);
b = reg.coeff(2:3);
beta = [1;-b];
```

Suppose that a model selection procedure indicates the adequacy of $q = 2$ lags in a VEC(q) model. Subsequently, the model is

$$\Delta y_t = \alpha(\beta' y_{t-1} + c_0) + \sum_{i=1}^{2} B_i \Delta y_{t-i} + c_1 + \varepsilon_t.$$

Because you estimated `c0` and β = `[1; -b]` previously, you can conditionally estimate α, `B1`, `B2`, and `c1` by:

1. Forming the required lagged differences
2. Regress the first difference of the series onto the q lagged differences and the estimated cointegration term.

Form the lagged difference series.

```
q = 2;
[numObs,numDims] = size(Y);
tBase = (q+2):numObs;                    % Commensurate time base, all lags
T = length(tBase);                       % Effective sample size
YLags = lagmatrix(Y,0:(q+1));            % Y(t-k) on observed time base
LY = YLags(tBase,(numDims+1):2*numDims); % Y(t-1) on commensurate time base
```

Form multidimensional differences so that the k^{th} `numDims`-wide block of columns in `DelatYLags` contains `(1-L)Y(t-k+1)`.

```
DeltaYLags = zeros(T,(q+1)*numDims);
for k = 1:(q+1)
    DeltaYLags(:,((k-1)*numDims+1):k*numDims) = ...
               YLags(tBase,((k-1)*numDims+1):k*numDims) ...
             - YLags(tBase,(k*numDims+1):(k+1)*numDims);
end

DY = DeltaYLags(:,1:numDims);        % (1-L)Y(t)
DLY = DeltaYLags(:,(numDims+1):end); % [(1-L)Y(t-1),...,(1-L)Y(t-q)]
```

Regress the the first difference of the series onto the q lagged differences and the estimated cointegration term. Include an intercept in the regression.

```
X = [(LY*beta-c0),DLY,ones(T,1)];
P = (X\DY)'; % [alpha,B1,...,Bq,c1]
alpha = P(:,1);
B1 = P(:,2:4);
B2 = P(:,5:7);
c1 = P(:,end);
```

Display the VEC model coefficients.

```
alpha,b,c0,B1,B2,c1
```

```
alpha =

   -0.6336
    0.0595
    0.0269


b =

    2.2209
   -1.0718


c0 =

   -1.2393
```

```
B1 =

    0.1649   -0.1465   -0.0416
   -0.0024    0.3816   -0.3716
    0.0815    0.1790   -0.1528


B2 =

   -0.3205    0.9506   -0.9514
   -0.1996    0.5169   -0.5211
   -0.1751    0.6061   -0.5419


c1 =

    0.1516
    0.1508
    0.1503
```

See Also

`egcitest` | `mvnrnd`

Related Examples

- "Test for Cointegration Using the Engle-Granger Test" on page 7-121
- "Simulate and Forecast a VEC Model" on page 7-129

More About

- "Vector Autoregressive (VAR) Models" on page 7-3
- "Cointegration and Error Correction Analysis" on page 7-108
- "Identifying Single Cointegrating Relations" on page 7-116

Simulate and Forecast a VEC Model

This example shows how to generate forecasts for a VEC(q) model several ways:

- Monte Carlo forecast paths using the VEC(q) model directly
- Minimum mean square error (MMSE) forecasts using the VAR model representation of the VEC(q) model.
- Monte Carlo forecast paths using the VAR(p) model representation of the VEC(q) model.

This example follows from "Estimate VEC Model Parameters Using egcitest".

Load Data and Preprocess

Load the `Data_Canada` data set. Extract the interest rate data.

```
load Data_Canada
Y = Data(:,3:end);
```

Estimate VEC(q) Model

Assuming that the interest rate data follows a VEC(2) model, fit the model to the data.

```
[~,~,~,~,reg] = egcitest(Y,'test','t2');
c0 = reg.coeff(1);
b = reg.coeff(2:3);
beta = [1; -b];
q = 2;
[numObs,numDims] = size(Y);
tBase = (q+2):numObs;                % Commensurate time base, all lags
T = length(tBase);                   % Effective sample size
DeltaYLags = zeros(T,(q+1)*numDims);
YLags = lagmatrix(Y,0:(q+1));        % Y(t-k) on observed time base
LY = YLags(tBase,(numDims+1):2*numDims);
for k = 1:(q+1)
    DeltaYLags(:,((k-1)*numDims+1):k*numDims) = ...
              YLags(tBase,((k-1)*numDims+1):k*numDims) ...
            - YLags(tBase,(k*numDims+1):(k+1)*numDims);
end
DY = DeltaYLags(:,1:numDims);        % (1-L)Y(t)
DLY = DeltaYLags(:,(numDims+1):end); % [(1-L)Y(t-1),...,(1-L)Y(t-q)]
X = [(LY*beta-c0),DLY,ones(T,1)];
P = (X\DY)';                         % [alpha,B1,...,Bq,c1]
```

```
alpha = P(:,1);
C = alpha*beta';                        % Error-correction coefficient matrix
B1 = P(:,2:4);                          % VEC(2) model coefficient
B2 = P(:,5:7);                          % VEC(2) model coefficient
c1 = P(:,end);
b = (alpha*c0 + c1)';                   % VEC(2) model constant offset
res = DY-X*P';
EstCov = cov(res);
```

Monte Carlo Forecasts Using VEC Model

Specify a 10-period forecast horizon. Set `numPaths` to generate 1000 paths. Because q, the degree of the VEC model, is 2, reserve three presample observations for y_t.

```
forecastHorizon = 10;
numPaths        = 1000;
psSize          = 3;

PSY = repmat(Y(end-(psSize-1):end,:),1,1,numPaths); % Presample
YSimVEC = zeros(forecastHorizon,numDims,numPaths);  % Preallocate forecasts
YSimVEC = [PSY; YSimVEC];
```

Generate Monte Carlo forecasts by adding the simulated innovations to the estimated VEC(2) model.

```
rng('default'); % For reproducibility
for p = 1:numPaths
    for t = (1:forecastHorizon) + psSize;
        eps = mvnrnd([0 0 0],EstCov);
        YSimVEC(t,:,p) = YSimVEC(t-1,:,p) + (C*YSimVEC(t-1,:,p)')'...
                    + (YSimVEC(t-1,:,p) - YSimVEC(t - 2,:,p))*B1'...
                    + (YSimVEC(t-2,:,p) - YSimVEC(t - 3,:,p))*B2'...
                    + b + eps;
    end
end
```

`YSimVEC` is a 13-by-3-by-1000 numeric array. Its rows correspond to presample and forecast periods, columns correspond to the time series, and the pages correspond to a draw.

Compute the mean of the forecasts for each period and time series over all paths. Construct 95% percentile forecast intervals for each period and time series.

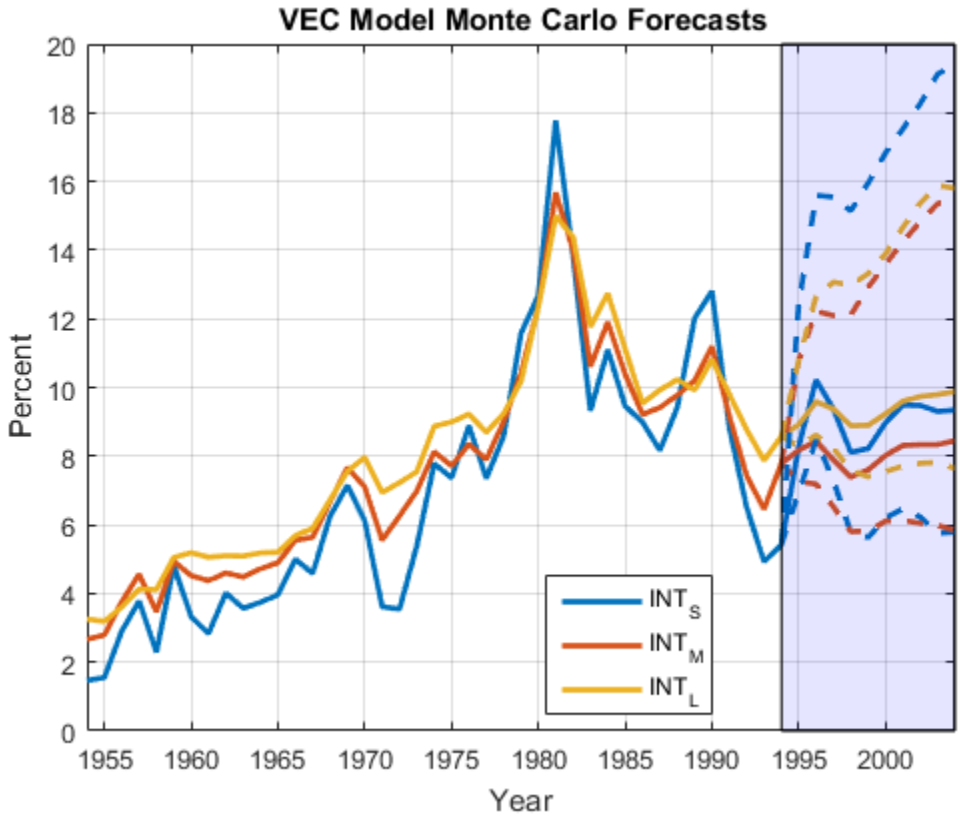
```
FMCVEC   = mean(YSimVEC((psSize + 1):end,:,:),3);
```

```
CIMCVEC = quantile(YSimVEC((psSize + 1):end,:,:),[0.25,0.975],3);
```

`FMCVEC` is a 10-by-3 numeric matrix containing the Monte Carlo forecasts for each period (row) and time series (column). `CIMCVEC` is a 10-by-3-by-2 numeric array containing the 2.5% and 97.5% percentiles (pages) of the draws for each period (row) and time series (column).

Plot the effective-sample observations, the mean forecasts, and the 95% percentile confidence intervals.

```
fDates = dates(end) + (0:forecastHorizon)';
figure;
h1 = plot([dates; fDates(2:end)],[Y; FMCVEC],'LineWidth',2);
h2 = gca;
hold on
h3 = plot(repmat(fDates,1,3),[Y(end,:,:); CIMCVEC(:,:,1)],'--',...
    'LineWidth',2);
h3(1).Color = h1(1).Color;
h3(2).Color = h1(2).Color;
h3(3).Color = h1(3).Color;
h4 = plot(repmat(fDates,1,3),[Y(end,:,:); CIMCVEC(:,:,2)],'--',...
    'LineWidth',2);
h4(1).Color = h1(1).Color;
h4(2).Color = h1(2).Color;
h4(3).Color = h1(3).Color;
patch([fDates(1) fDates(1) fDates(end) fDates(end)],...
    [h2.YLim(1) h2.YLim(2) h2.YLim(2) h2.YLim(1)],'b','FaceAlpha',0.1)
xlabel('Year')
ylabel('Percent')
title('{\bf VEC Model Monte Carlo Forecasts}')
axis tight
grid on
legend(h1,DataTable.Properties.VariableNames(3:end),'Location','Best');
```



The plot suggests that `INT_S` has lower forecast accuracy that the other two series because its confidence bounds are the widest.

MMSE Forecasts Using VAR(p) Representation

Compute the autoregressive coefficient matrices of the equivalent VAR(3) model to the VEC(2) model (i.e., the coefficients of y_{t-1}, y_{t-2}, and y_{t-3}).

```
A = vec2var({B1 B2},C);
```

`A` is a 1-by-3 row cell vector. `A{j}` is the autoregressive coefficient matrix for lag term `j`. The constant offset (`b`) of the VEC(2) model and the constant offset of the equivalent VAR(3) model are equal.

Create a VAR(3) model object.

```
VAR3 = vgxset('AR',A,'a',b,'Q',EstCov);
```

Forecast over a 10 period horizon. Compute 95% individual, Wald-type confidence intervals for each series.

```
[MMSEF,CovF] = vgxpred(VAR3,forecastHorizon,[],Y);
var = cellfun(@diag,CovF,'UniformOutput',false);
CIF = zeros(forecastHorizon,numDims,2); % Preallocation
for j = 1:forecastHorizon
    stdev = sqrt(var{j});
    CIF(j,:,1) = MMSEF(j,:) - 1.96*stdev';
    CIF(j,:,2) = MMSEF(j,:) + 1.96*stdev';
end
```

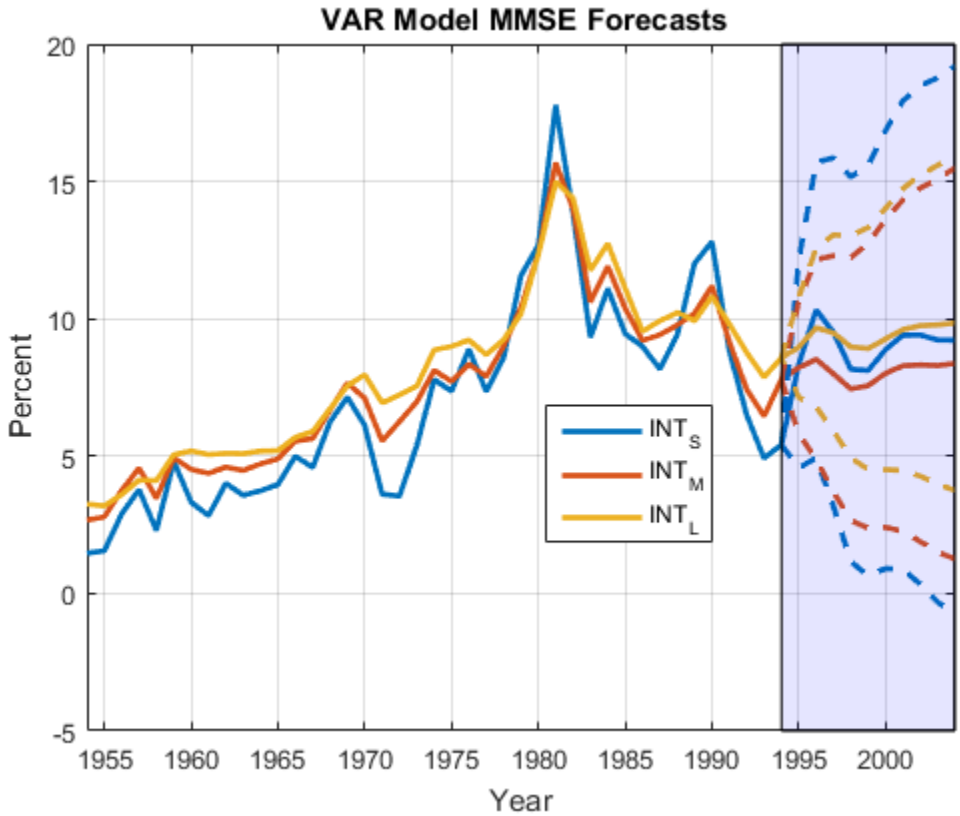
The confidence intervals do not account for the correlation between the forecasted series at a particular time.

`MMSEF` is a 10-by-3 numeric matrix of the MMSE forecasts for the VAR(3) model. Rows correspond to forecast periods and columns to time series. `CovF` is a 10-by-1 cell vector of forecast covariance matrices, in which each row corresponds to a forecast period.

Plot the effective-sample observations, the MMSE forecasts, and the 95% Wald confidence intervals.

```
figure;
h1 = plot([dates; fDates(2:end)],[Y; MMSEF],'LineWidth',2);
h2 = gca;
hold on
h3 = plot(repmat(fDates,1,3),[Y(end,:,:); CIF(:,:,1)],'--',...
    'LineWidth',2);
h3(1).Color = h1(1).Color;
h3(2).Color = h1(2).Color;
h3(3).Color = h1(3).Color;
h4 = plot(repmat(fDates,1,3),[Y(end,:,:); CIF(:,:,2)],'--',...
    'LineWidth',2);
h4(1).Color = h1(1).Color;
h4(2).Color = h1(2).Color;
h4(3).Color = h1(3).Color;
patch([fDates(1) fDates(1) fDates(end) fDates(end)],...
    [h2.YLim(1) h2.YLim(2) h2.YLim(2) h2.YLim(1)],'b','FaceAlpha',0.1)
xlabel('Year')
ylabel('Percent')
```

```
title('{\bf VAR Model MMSE Forecasts}')
axis tight
grid on
legend(h1,DataTable.Properties.VariableNames(3:end),'Location','Best');
```



The MMSE forecasts are very close to the VEC(2) model Monte Carlo forecast means. However, the confidence bounds are wider.

Monte Carlo Forecasts Using VAR(p) Representation

Simulate 1000 paths of the VAR(3) model into the forecast horizon. Use the observations as presample data.

```
YSimVAR = vgxsim(VAR3,forecastHorizon,[],Y,[],numPaths);
```

`YSimVAR` is a 10-by-3-by-1000 numeric array similar to `YSimVEC`.

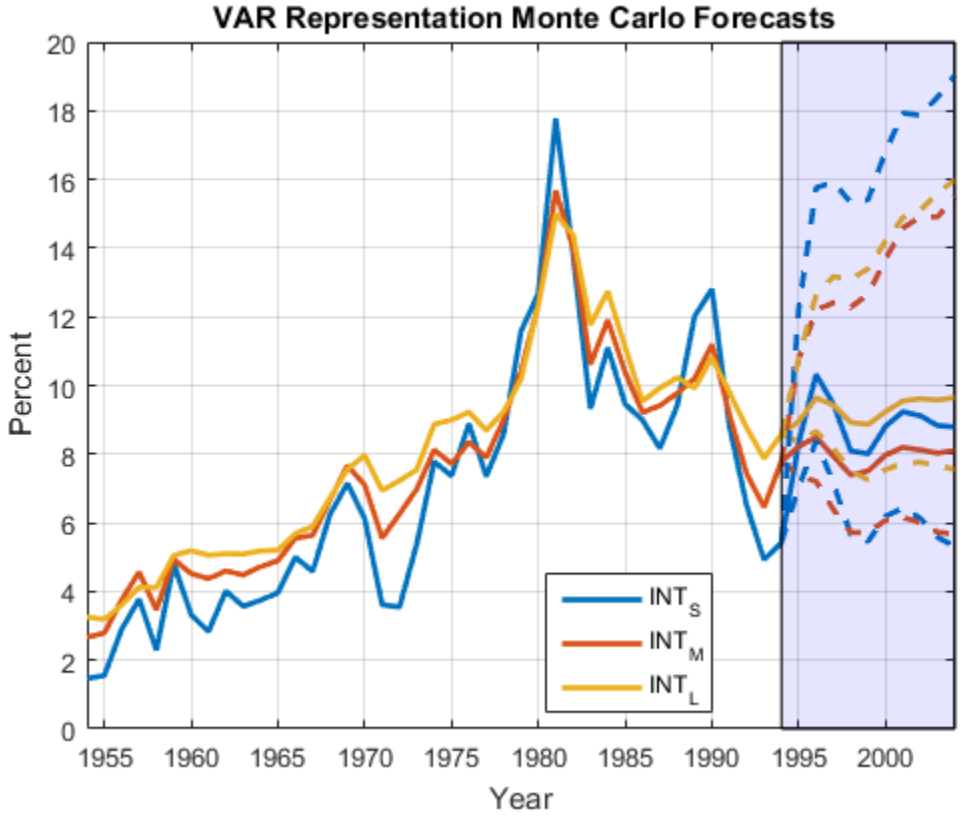
Compute the mean of the forecasts for each period and time series over all paths. Construct 95% percentile forecast intervals for each period and time series.

```
FMCVAR   = mean(YSimVAR,3);
CIMCVAR  = quantile(YSimVAR,[0.25,0.975],3);
```

`FMCVAR` is a 10-by-3 numeric matrix containing the Monte Carlo forecasts for each period and time series. `CIMCVAR` is a 10-by-3-by-2 numeric array containing the 2.5% and 97.5% percentiles of the draws for each period and time series.

Plot the effective-sample observations, the mean forecasts, and the 95% percentile confidence intervals.

```
figure;
h1 = plot([dates; fDates(2:end)],[Y; FMCVAR],'LineWidth',2);
h2 = gca;
hold on
h3 = plot(repmat(fDates,1,3),[Y(end,:,:); CIMCVAR(:,:,1)],'--',...
    'LineWidth',2);
h3(1).Color = h1(1).Color;
h3(2).Color = h1(2).Color;
h3(3).Color = h1(3).Color;
h4 = plot(repmat(fDates,1,3),[Y(end,:,:); CIMCVAR(:,:,2)],'--',...
    'LineWidth',2);
h4(1).Color = h1(1).Color;
h4(2).Color = h1(2).Color;
h4(3).Color = h1(3).Color;
patch([fDates(1) fDates(1) fDates(end) fDates(end)],...
    [h2.YLim(1) h2.YLim(2) h2.YLim(2) h2.YLim(1)],'b','FaceAlpha',0.1)
xlabel('Year')
ylabel('Percent')
title('{\bf VAR Representation Monte Carlo Forecasts}')
axis tight
grid on
legend(h1,DataTable.Properties.VariableNames(3:end),'Location','Best');
```



All sets of mean forecasts are very close. Both the VAR(3) and VEC(2) sets of Monte Carlo forecasts are almost equivalent.

See Also

`egcitest` | `mvnrnd` | `vec2var` | `vgxpred` | `vgxset` | `vgxsim`

Related Examples

- "Test for Cointegration Using the Engle-Granger Test" on page 7-121
- "Estimate VEC Model Parameters Using egcitest" on page 7-126

More About

- "Vector Autoregressive (VAR) Models" on page 7-3
- "Cointegration and Error Correction Analysis" on page 7-108
- "Identifying Single Cointegrating Relations" on page 7-116

Generate VEC Model Impulse Responses

This example shows how to generate impulse responses for this VEC(3) model ([74], Ch. 6.7):

$$\begin{aligned}\Delta y_t &= \begin{bmatrix} 0.24 & -0.08 \\ 0 & -0.31 \end{bmatrix}\Delta y_{t-1} + \begin{bmatrix} 0 & -0.13 \\ 0 & -0.37 \end{bmatrix}\Delta y_{t-2} + \begin{bmatrix} 0.20 & -0.06 \\ 0 & -0.34 \end{bmatrix}\Delta y_{t-3} \\ &+ \begin{bmatrix} -0.07 \\ 0.17 \end{bmatrix}\begin{bmatrix} 1 & -4 \end{bmatrix} y_{t-1} + \varepsilon_t\end{aligned}$$

y_t is a 2 dimensional time series. $\Delta y_t = y_t - y_{t-1}$. ε_t is a 2 dimensional series of mean zero, Gaussian innovations with covariance matrix

$$\Sigma = 10^{-5}\begin{bmatrix} 2.61 & -0.15 \\ -0.15 & 2.31 \end{bmatrix}.$$

Specify the VEC(3) model autoregressive coefficient matrices B_1, B_2, and B_3, the error-correction coefficient matrix C, and the innovations covariance matrix Σ.

```
B1    = [0.24 -0.08;
         0.00 -0.31];
B2    = [0.00 -0.13;
         0.00 -0.37];
B3    = [0.20 -0.06;
         0.00 -0.34];
C     = [-0.07; 0.17]*[1 -4];
Sigma = [ 2.61 -0.15;
         -0.15  2.31]*1e-5;
```

Compute the autoregressive coefficient matrices that compose the VAR(4) model that is equivalent to the VEC(3) model.

```
B = {B1; B2; B3};
A = vec2var(B,C);
```

`A` is a 4-by-1 cell vector containing the 2-by-2, VAR(4) model autoregressive coefficient matrices. Cell `A{j}` contains the coefficient matrix for lag `j` in difference-equation notation. The VAR(4) is in terms of y_t rather than Δy_t.

Compute the forecast error impulse responses (FEIR) for the VAR(4) representation. That is, accept the default identity matrix for the innovations covariance. Specify to return the impulse responses for the first 20 periods.

```
numObs = 20;
IR = cell(2,1); % Preallocation
IR{1} = armairf(A,[],'NumObs',numObs);
```

To compute impulse responses, `armairf` filters an innovation standard deviation shock from one series to itself and all other series. In this case, the magnitude of the shock is 1 for each series.

Compute orthogonalized impulse responses by supplying the innovations covariance matrix. Specify to return the impulse responses for the first 20 periods.

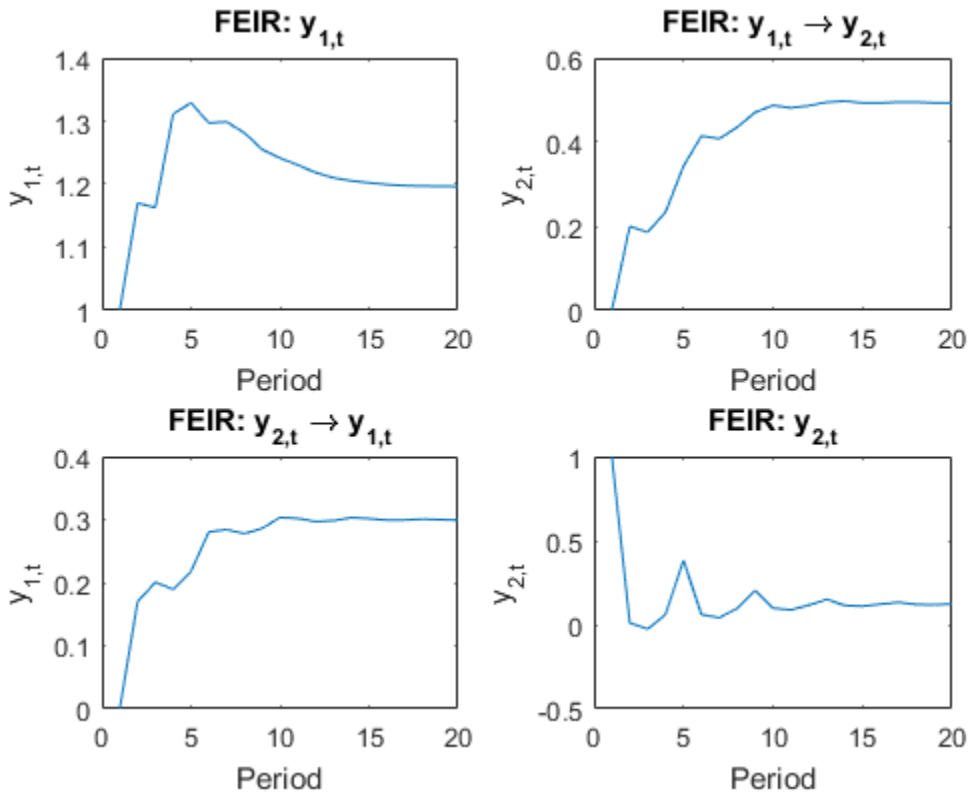
```
IR{2} = armairf(A,[],'InnovCov',Sigma,'NumObs',numObs);
```

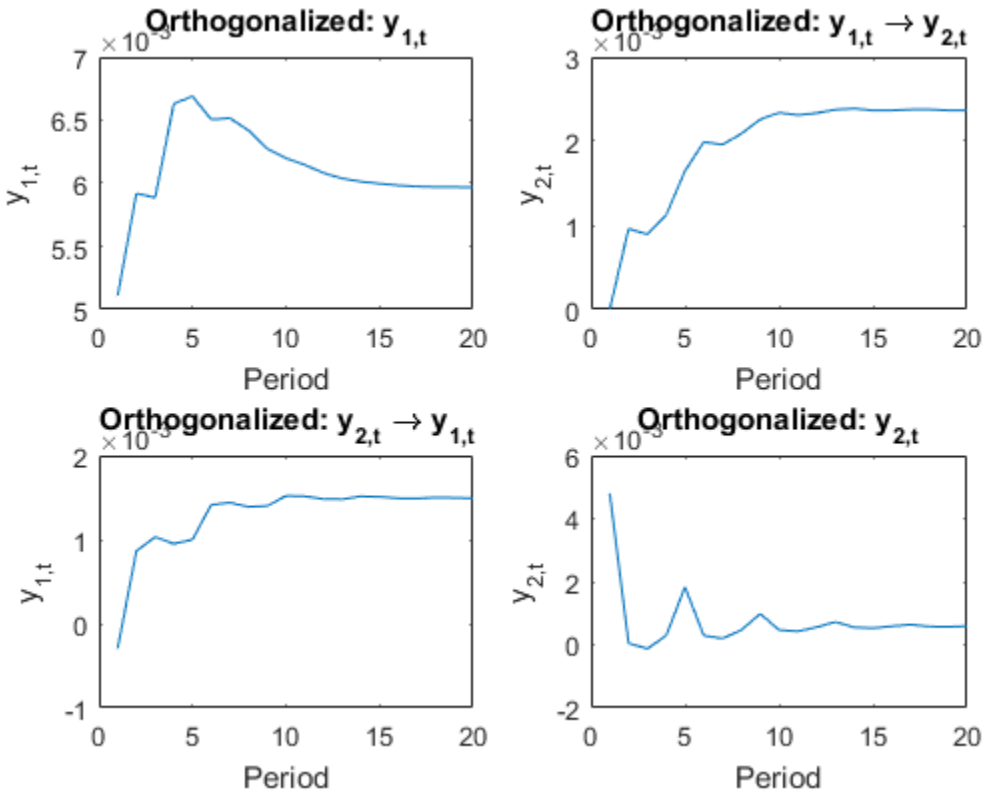
For orthogonalized impulse responses, the innovations covariance governs the magnitude of the filtered shock.

Plot the FEIR and the orthogonalized impulse responses for all series.

```
type = {'FEIR','Orthogonalized'};
for j = 1:2;
    figure;
    imp = IR{j};
    subplot(2,2,1);
    plot(imp(:,1,1))
    title(sprintf('%s: y_{1,t}',type{j}));
    ylabel('y_{1,t}');
    xlabel('Period');
    subplot(2,2,2);
    plot(imp(:,1,2))
    title(sprintf('%s: y_{1,t} \\rightarrow y_{2,t}',type{j}));
    ylabel('y_{2,t}');
    xlabel('Period');
    subplot(2,2,3);
    plot(imp(:,2,1))
    title(sprintf('%s: y_{2,t} \\rightarrow y_{1,t}',type{j}));
    ylabel('y_{1,t}');
    xlabel('Period');
    subplot(2,2,4);
    plot(imp(:,2,2))
```

```
    title(sprintf('%s: y_{2,t}',type{j}));
    ylabel('y_{2,t}');
    xlabel('Period');
end
```





Because the innovations covariance is almost diagonal, the FEIR and orthogonalized impulse responses have similar dynamic behaviors ([74], Ch. 6.7). However, the scale of the plots are markedly different.

See Also

`armairf` | `vec2var`

Related Examples

- "Generate Impulse Responses for a VAR model" on page 7-42
- "Simulate and Forecast a VEC Model" on page 7-129

More About

- "Vector Autoregressive (VAR) Models" on page 7-3
- "Cointegration and Error Correction Analysis" on page 7-108

Identifying Multiple Cointegrating Relations

The Johansen test for cointegration addresses many of the limitations of the Engle-Granger method. It avoids two-step estimators and provides comprehensive testing in the presence of multiple cointegrating relations. Its maximum likelihood approach incorporates the testing procedure into the process of model estimation, avoiding conditional estimates. Moreover, the test provides a framework for testing restrictions on the cointegrating relations B and the adjustment speeds A in the VEC model.

At the core of the Johansen method is the relationship between the rank of the impact matrix $C = AB'$ and the size of its eigenvalues. The eigenvalues depend on the form of the VEC model, and in particular on the composition of its deterministic terms (see "The Role of Deterministic Terms" on page 7-110). The method infers the cointegration rank by testing the number of eigenvalues that are statistically different from 0, then conducts model estimation under the rank constraints. Although the method appears to be very different from the Engle-Granger method, it is essentially a multivariate generalization of the augmented Dickey-Fuller test for unit roots. See, e.g., [35].

The Johansen test is implemented in Econometrics Toolbox by the function `jcitest`. For an example, see "Test for Cointegration Using the Johansen Test" on page 7-144.

See Also

`jcitest`

Related Examples

- "Test for Cointegration Using the Engle-Granger Test" on page 7-121
- "Estimate VEC Model Parameters Using jcitest" on page 7-147
- "Compare Approaches to Cointegration Analysis" on page 7-150

More About

- "Vector Autoregressive (VAR) Models" on page 7-3
- "Cointegration and Error Correction Analysis" on page 7-108
- "Identifying Multiple Cointegrating Relations" on page 7-143

Test for Cointegration Using the Johansen Test

This example shows how to assess whether a multivariate time series has multiple cointegrating relations using the Johansen test.

Load `Data_Canada` into the MATLAB® Workspace. The data set contains the term structure of Candian interest rates [75]. Extract the short-term, medium-term, and long-term interest rate series.

```
load Data_Canada
Y = Data(:,3:end); % Interest rate data
```

To illustrate the input and output structure of `jcitest` when conducting multiple tests, test for the cointegration rank using the default H1 model and two different lag structures.

```
[h,pValue,stat,cValue] = jcitest(Y,'model','H1','lags',1:2);
```

```
************************
Results Summary (Test 1)

Data: Y
Effective sample size: 39
Model: H1
Lags: 1
Statistic: trace
Significance level: 0.05


r  h  stat      cValue   pValue   eigVal
----------------------------------------
0  1  35.3442   29.7976  0.0104   0.3979
1  1  15.5568   15.4948  0.0490   0.2757
2  0  2.9796    3.8415   0.0843   0.0736

************************
Results Summary (Test 2)

Data: Y
Effective sample size: 38
Model: H1
Lags: 2
Statistic: trace
```

```
Significance level: 0.05

r  h  stat      cValue   pValue   eigVal
---------------------------------------
0  0  25.8188   29.7976  0.1346   0.2839
1  0  13.1267   15.4948  0.1109   0.2377
2  0  2.8108    3.8415   0.0937   0.0713
```

The default "trace" test assesses null hypotheses $H(r)$ of cointegration rank less than or equal to r against the alternative $H(n)$, where n is the dimension of the data. The summaries show that the first test rejects a cointegration rank of 0 (no cointegration) and just barely rejects a cointegration rank of 1, but fails to reject a cointegration rank of 2. The inference is that the data exhibit 1 or 2 cointegrating relationships. With an additional lag in the model, the second test fails to reject any of the cointegration ranks, providing little by way of inference. It is important to determine a reasonable lag length for the VEC model (as well as the general form of the model) before testing for cointegration.

Because the Johansen method, by its nature, tests multiple rank specifications for each specification of the remaining model parameters, `jcitest` returns the results in the form of tabular arrays, and indexes by null rank and test number.

Display the test results, `h`.

```
h
```

```
h =

           r0       r1       r2
          _____    _____    _____

    t1    true     true     false
    t2    false    false    false
```

Column headers indicate tests `r0`, `r1`, and `r2`, respectively, of $H(0)$, $H(1)$, and $H(2)$ against $H(3)$. Row headers `t1` and `t2` indicate the two separate tests (two separate lag structures) specified by the input parameters.

Access the result for the second test at null rank $r = 0$ useing tabular array indexing.

```
h20 = h.r0(2)

h20 =

     0
```

See Also

`jcitest`

Related Examples

- "Estimate VEC Model Parameters Using jcitest" on page 7-147
- "Compare Approaches to Cointegration Analysis" on page 7-150

More About

- "Vector Autoregressive (VAR) Models" on page 7-3
- "Cointegration and Error Correction Analysis" on page 7-108
- "Identifying Multiple Cointegrating Relations" on page 7-143

Estimate VEC Model Parameters Using jcitest

In addition to testing for multiple cointegrating relations, `jcitest` produces maximum likelihood estimates of VEC model coefficients under the rank restrictions on B. Estimation information is returned in an optional fifth output argument, and can be displayed by setting an optional input parameter. For example, the following estimates a VEC(2) model of the data, and displays the results under each of the rank restrictions $r = 0$, $r = 1$, and $r = 2$:

```
load Data_Canada
Y = Data(:,3:end); % Interest rate data
[~,~,~,~,mles] = jcitest(Y,'model','H1','lags',2,...
    'display','params');


**************************
Parameter Estimates (Test 1)

r = 0
------
B1 =
   -0.1848    0.5704   -0.3273
    0.0305    0.3143   -0.3448
    0.0964    0.1485   -0.1406

B2 =
   -0.6046    1.6615   -1.3922
   -0.1729    0.4501   -0.4796
   -0.1631    0.5759   -0.5231

c1 =
    0.1420
    0.1517
    0.1508


r = 1
------
A =
   -0.6259
   -0.2261
   -0.0222

B =
```

```
    0.7081
    1.6282
   -2.4581

B1 =
    0.0579    1.0824   -0.8718
    0.1182    0.4993   -0.5415
    0.1050    0.1667   -0.1600

B2 =
   -0.5462    2.2436   -1.7723
   -0.1518    0.6605   -0.6169
   -0.1610    0.5966   -0.5366

c0 =
    2.2351

c1 =
   -0.0366
    0.0872
    0.1444


r = 2
------
A =
   -0.6259    0.1379
   -0.2261   -0.0480
   -0.0222    0.0137

B =
    0.7081   -2.4407
    1.6282    6.2883
   -2.4581   -3.5321

B1 =
    0.2438    0.6395   -0.6729
    0.0535    0.6533   -0.6107
    0.1234    0.1228   -0.1403

B2 =
   -0.3857    1.7970   -1.4915
   -0.2076    0.8158   -0.7146
   -0.1451    0.5524   -0.5089
```

```
c0 =
    2.0901
   -3.0289

c1 =
   -0.0104
    0.0137
    0.1528
```

`mles` is a tabular array of structure arrays, with each structure containing information for a particular test under a particular rank restriction. Since both tabular arrays and structure arrays use similar indexing, you can access the tabular array and then the structure using dot notation. For example, to access the rank 2 matrix of cointegrating relations:

```
B = mles.r2.paramVals.B
```

```
B =

    0.7081   -2.4407
    1.6282    6.2883
   -2.4581   -3.5321
```

See Also

`jcitest`

Related Examples

- "Test for Cointegration Using the Johansen Test" on page 7-144
- "Compare Approaches to Cointegration Analysis" on page 7-150

More About

- "Vector Autoregressive (VAR) Models" on page 7-3
- "Cointegration and Error Correction Analysis" on page 7-108
- "Identifying Single Cointegrating Relations" on page 7-116

Compare Approaches to Cointegration Analysis

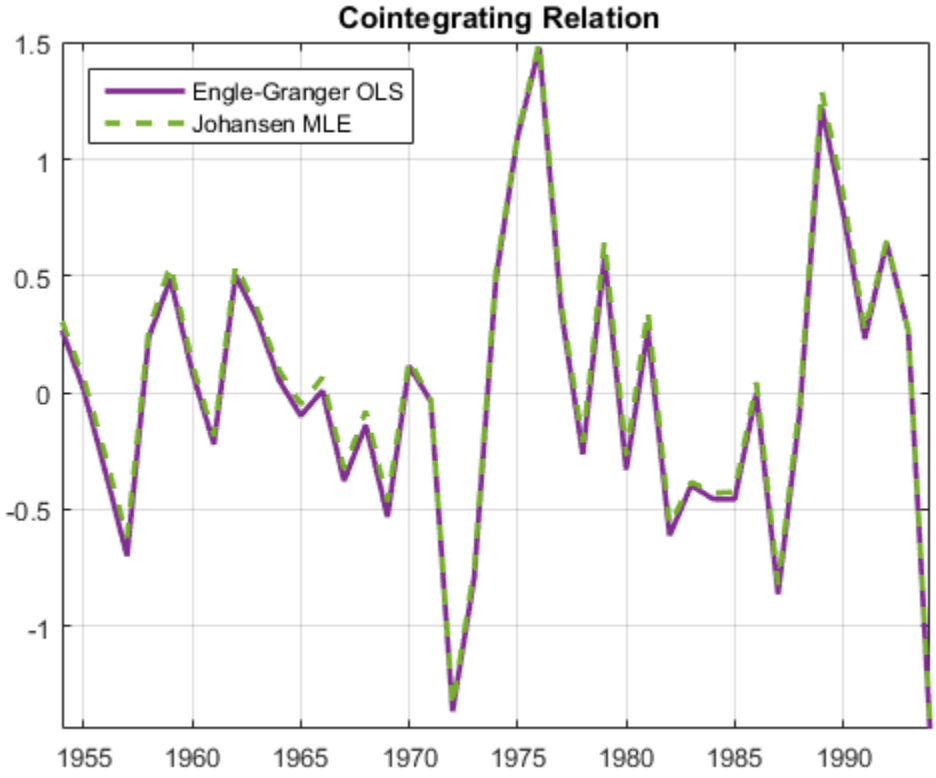
Comparing inferences and estimates from the Johansen and Engle-Granger approaches can be challenging, for a variety of reasons. First of all, the two methods are essentially different, and may disagree on inferences from the same data. The Engle-Granger two-step method for estimating the VEC model, first estimating the cointegrating relation and then estimating the remaining model coefficients, differs from Johansen's maximum likelihood approach. Secondly, the cointegrating relations estimated by the Engle-Granger approach may not correspond to the cointegrating relations estimated by the Johansen approach, especially in the presence of multiple cointegrating relations. It is important, in this context, to remember that cointegrating relations are not uniquely defined, but depend on the decomposition $C = AB'$ of the impact matrix.

Nevertheless, the two approaches should provide generally comparable results, if both begin with the same data and seek out the same underlying relationships. Properly normalized, cointegrating relations discovered by either method should reflect the mechanics of the data-generating process, and VEC models built from the relations should have comparable forecasting abilities.

As the following shows in the case of the Canadian interest rate data, Johansen's H1* model, which is the closest to the default settings of `egcitest`, discovers the same cointegrating relation as the Engle-Granger test, assuming a cointegration rank of 2:

```
load Data_Canada
Y = Data(:,3:end); % Interest rate data
[~,~,~,~,reg] = egcitest(Y,'test','t2');
c0 = reg.coeff(1);
b = reg.coeff(2:3);
beta = [1; -b];

[~,~,~,~,mles] = jcitest(Y,'model','H1*');
BJ2 = mles.r2.paramVals.B;
c0J2 = mles.r2.paramVals.c0;

% Normalize the 2nd cointegrating relation with respect to
%  the 1st variable, to make it comparable to Engle-Granger:
BJ2n = BJ2(:,2)/BJ2(1,2);
c0J2n = c0J2(2)/BJ2(1,2);

% Plot the normalized Johansen cointegrating relation together
%  with the original Engle-Granger cointegrating relation:
```

```
h = gca;
COrd = h.ColorOrder;

plot(dates,Y*beta-c0,'LineWidth',2,'Color',COrd(4,:))
hold on
plot(dates,Y*BJ2n+c0J2n,'--','LineWidth',2,'Color',COrd(5,:))
legend('Engle-Granger OLS','Johansen MLE','Location','NW')
title('{\bf Cointegrating Relation}')
axis tight
grid on
hold off


************************
Results Summary (Test 1)

Data: Y
Effective sample size: 40
Model: H1*
Lags: 0
Statistic: trace
Significance level: 0.05


r  h  stat      cValue   pValue   eigVal
-----------------------------------------
0  1  38.8360   35.1929  0.0194   0.4159
1  0  17.3256   20.2619  0.1211   0.2881
2  0  3.7325    9.1644   0.5229   0.0891
```



See Also

`jcitest`

Related Examples

- "Estimate VEC Model Parameters Using jcitest" on page 7-147
- "Test for Cointegration Using the Johansen Test" on page 7-144

More About

- "Vector Autoregressive (VAR) Models" on page 7-3
- "Cointegration and Error Correction Analysis" on page 7-108

- “Identifying Single Cointegrating Relations” on page 7-116

Testing Cointegrating Vectors and Adjustment Speeds

A separate Econometrics Toolbox function, `jcontest`, uses the Johansen framework to test linear constraints on cointegrating relations B and adjustment speeds A, and estimates VEC model parameters under the additional constraints. Constraint testing allows you to assess the validity of relationships suggested by economic theory.

Constraints imposed by `jcontest` take one of two forms. Constraints of the form $R'A = 0$ or $R'B = 0$ specify particular combinations of the variables to be held fixed during testing and estimation. These constraints are equivalent to parameterizations $A = H\varphi$ or $B = H\varphi$, where H is the orthogonal complement of R (in MATLAB, `null(R')`) and φ is a vector of free parameters. The second constraint type specifies particular vectors in the column space of A or B. The number of constraints that `jcontest` can impose is restricted by the rank of the matrix being tested, which can be inferred by first running `jcitest`.

See Also

`jcitest` | `jcontest`

Related Examples

- "Test Cointegrating Vectors" on page 7-155
- "Test Adjustment Speeds" on page 7-158

More About

- "Vector Autoregressive (VAR) Models" on page 7-3
- "Cointegration and Error Correction Analysis" on page 7-108
- "Identifying Multiple Cointegrating Relations" on page 7-143

Test Cointegrating Vectors

Tests on B answer questions about the space of cointegrating relations. The column vectors in B, estimated by `jcitest`, do not uniquely define the cointegrating relations. Rather, they estimate a space of cointegrating relations, given by the span of the vectors. Tests on B allow you to determine if other potentially interesting relations lie in that space. When constructing constraints, interpret the rows and columns of the n-by- r matrix B as follows:

- Row i of B contains the coefficients of variable y_{it} in each of the r cointegrating relations.
- Column j of B contains the coefficients of each of the n variables in cointegrating relation j.

One application of `jcontest` is to pretest variables for their order of integration. At the start of any cointegration analysis, trending variables are typically tested for the presence of a unit root. These pretests can be carried out with combinations of standard unit root and stationarity tests such as `adftest`, `pptest`, `kpsstest`, or `lmctest`. Alternatively, `jcontest` lets you carry out stationarity testing within the Johansen framework. To do so, specify a cointegrating vector that is 1 at the variable of interest and 0 elsewhere, and then test to see if that vector is in the space of cointegrating relations. The following tests all of the variables in `Y` a single call:

```
load Data_Canada
Y = Data(:,3:end); % Interest rate data
[h0,pValue0] = jcontest(Y,1,'BVec',{[1 0 0]',[0 1 0]',[0 0 1]'})
```

```
h0 =

     1     1     1


pValue0 =

   1.0e-03 *

    0.3368    0.1758    0.1310
```

The second input argument specifies a cointegration rank of 1, and the third and fourth input arguments are a parameter/value pair specifying tests of specific vectors in the

space of cointegrating relations. The results strongly reject the null of stationarity for each of the variables, returning very small *p*-values.

Another common test of the space of cointegrating vectors is to see if certain combinations of variables suggested by economic theory are stationary. For example, it may be of interest to see if interest rates are cointegrated with various measures of inflation (and, via the Fisher equation, if real interest rates are stationary). In addition to the interest rates already examined, `Data_Canada.mat` contains two measures of inflation, based on the CPI and the GDP deflator, respectively. To demonstrate the test procedure (without any presumption of having identified an adequate model), we first run `jcitest` to determine the rank of *B*, then test the stationarity of a simple spread between the CPI inflation rate and the short-term interest rate:

```
y1 = Data(:,1); % CPI-based inflation rate
YI = [y1,Y];

% Test if inflation is cointegrated with interest rates:
[h,pValue] = jcitest(YI);
% Test if y1 - y2 is stationary:
[hB,pValueB] = jcontest(YI,1,'BCon',[1 -1 0 0]')
```

```
***********************
Results Summary (Test 1)

Data: YI
Effective sample size: 40
Model: H1
Lags: 0
Statistic: trace
Significance level: 0.05


r  h  stat      cVvalue   pValue   eigVal
----------------------------------------
0  1  58.0038   47.8564  0.0045   0.5532
1  0  25.7783   29.7976  0.1359   0.3218
2  0  10.2434   15.4948  0.2932   0.1375
3  1  4.3263    3.8415   0.0376   0.1025

hB =

     1
```

```
pValueB =

    0.0242
```

The first test provides evidence of cointegration, and fails to reject a cointegration rank $r = 1$. The second test, assuming $r = 1$, rejects the hypothesized cointegrating relation. Of course, reliable economic inferences would need to include proper model selection, with corresponding settings for the `'model'` and other default parameters.

See Also

`jcitest` | `jcontest`

Related Examples

- "Test Adjustment Speeds" on page 7-158

More About

- "Testing Cointegrating Vectors and Adjustment Speeds" on page 7-154
- "Cointegration and Error Correction Analysis" on page 7-108

Test Adjustment Speeds

Tests on A answer questions about common driving forces in the system. When constructing constraints, interpret the rows and columns of the n-by- r matrix A as follows:

- Row i of A contains the adjustment speeds of variable y_{it} to disequilibrium in each of the r cointegrating relations.
- Column j of A contains the adjustment speeds of each of the n variables to disequilibrium in cointegrating relation j.

For example, an all-zero row in A indicates a variable that is weakly exogenous with respect to the coefficients in B. Such a variable may affect other variables, but does not adjust to disequilibrium in the cointegrating relations. Similarly, a standard unit vector column in A indicates a variable that is exclusively adjusting to disequilibrium in a particular cointegrating relation.

To demonstrate, we test for weak exogeneity of the inflation rate with respect to interest rates:

```
load Data_Canada
Y = Data(:,3:end); % Interest rate data
y1 = Data(:,1); % CPI-based inflation rate
YI = [y1,Y];

[hA,pValueA] = jcontest(YI,1,'ACon',[1 0 0 0]')
```

```
hA =

     0


pValueA =

    0.3206
```

The test fails to reject the null hypothesis. Again, the test is conducted with default settings. Proper economic inference would require a more careful analysis of model and rank specifications.

Constrained parameter estimates are accessed via a fifth output argument from `jcontest`. For example, the constrained, rank 1 estimate of A is obtained by referencing the fifth output with dot (`.`) indexing:

```
[~,~,~,~,mles] = jcontest(YI,1,'ACon',[1 0 0 0]');
Acon = mles.paramVals.A
```

```
Acon =

         0
    0.1423
    0.0865
    0.2862
```

The first row of A is 0, as specified by the constraint.

See Also

`jcitest` | `jcontest`

Related Examples

- "Test Cointegrating Vectors" on page 7-155

More About

- "Testing Cointegrating Vectors and Adjustment Speeds" on page 7-154
- "Cointegration and Error Correction Analysis" on page 7-108

State-Space Models

What Are State-Space Models?

In this section...

Definitions

State-Space Model

A *state-space model* is a discrete-time, stochastic model that contains two sets of equations:

- One describing how a latent process transitions in time (the *state equation*)
- Another describing how an observer measures the latent process at each period (the *observation equation*)

Symbolically, you can write a linear, multivariate, time-varying, Gaussian state-space model using the following system of equations

$$\begin{aligned} x_t &= A_t x_{t-1} + B_t u_t \\ y_t - Z_t \beta &= C_t x_t + D_t \varepsilon_t, \end{aligned}$$

for $t = 1,\ldots,T$.

- $x_t = \left[x_{t1},\ldots,x_{tm_t} \right]'$ is an m_t-dimensional state vector describing the dynamics of some, possibly unobservable, phenomenon at period t. The initial state distribution (x_0) is Gaussian with mean μ_0 and covariance matrix Σ_0.

- $y_t = \left[y_{t1},...,y_{tn_t} \right]'$ is an n_t-dimensional observation vector describing how the states are measured by observers at period t.
- A_t is the m_t-by-m_{t-1} state-transition matrix describing how the states at time t transition to the states at period $t-1$.
- B_t is the m_t-by-k_t state-disturbance-loading matrix describing how the states at period t combine with the innovations at period t.
- C_t is the n_t-by-m_t measurement-sensitivity matrix describing how the observations at period t relate to the states at period t.
- D_t is the n_t-by-h_t observation-innovation matrix describing how the observations at period t combine with the observation errors at period t.
- The matrices A_t, B_t, C_t, and D_t are referred to as *coefficient matrices*, and might contain unknown parameters.
- $u_t = \left[u_{t1},...,u_{tk_t} \right]'$ is a k_t-dimensional, Gaussian, white-noise, unit-variance vector of state disturbances at period t.
- $\varepsilon_t = \left[\varepsilon_{t1},...,\varepsilon_{th_t} \right]'$ is an h_t-dimensional, Gaussian, white-noise, unit-variance vector of observation innovations at period t.
- ε_t and u_t are uncorrelated.
- For time-invariant state-space models,
  - $Z_t = \left[z_{t1} \quad z_{t2} \quad \cdots \quad z_{td} \right]$ is row t of a T-by-d matrix of predictors Z. Each column of Z corresponds to a predictor, and each successive row to a successive period. If the observations are multivariate, then all predictors deflate each observation.
  - β is a d-by-n matrix of regression coefficients for Z_t.

To write a time-invariant state-space model, drop the t subscripts of all coefficient matrices and dimensions.

Diffuse State-Space Model

A diffuse state-space model is a state-space model that can contain at least one state with an infinite initial variance, called a *diffuse state*. In addition to having an infinite initial variance, all diffuse states are uncorrelated with all other states in the model. There are several motivations for using diffuse state-space models:

- The study of very early starting points of some nonstationary systems, such as random walk process, leads to initial distribution variances that approach infinity.
- An infinite variance specification for an initial state distribution indicates complete ignorance, or no prior knowledge, of the diffuse states. The advantage of this specification is that the analysis of these states is more objective. That is, the observations, rather than additional distribution assumptions, aid in understanding the diffuse states. The disadvantage is that posterior distributions of the states might be improper, and the likelihood function is unbounded. However, with enough data and an identifiable, Gaussian state-space model, the filtered and smoothed states, and a likelihood based on them, can be computed using the diffuse Kalman filter.
- Represent a static, initial state as unknown parameter by attributing to it an infinite variance.

Time-Invariant State-Space Models

In a *time-invariant* state-space model:

- The coefficient matrices are equivalent for all periods.
- The number of states, state disturbances, observations, and observation innovations are the same for all periods.

For example, for all t, the following system of equations

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix} = \begin{bmatrix} \phi_1 & 0 \\ 0 & \phi_2 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \end{bmatrix} + \begin{bmatrix} 0.5 & 0 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} u_{1,t} \\ u_{2,t} \end{bmatrix}$$

$$y_t = \begin{bmatrix} \phi_3 & 1 \end{bmatrix} \begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix} + 0.2\varepsilon_t$$

represents a time-invariant state-space model.

Time-Varying State-Space Model

In a *time-varying* state-space model:

- The coefficient matrices might change from period to period.
- The number of states, state disturbances, observations, and observation innovations might change from period to period. For example, this might happen if there is a regime shift or one of the states or observations cannot be measured during the sampling time frame. Also, you can model seasonality using time-varying models.

To illustrate a regime shift, suppose, for $t = 1,..,10$

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix} = \begin{bmatrix} \phi_1 & 0 \\ 0 & \phi_2 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \end{bmatrix} + \begin{bmatrix} 0.5 & 0 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} u_{1,t} \\ u_{2,t} \end{bmatrix},$$
$$y_t = \begin{bmatrix} \phi_3 & 1 \end{bmatrix} \begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix} + 0.2\varepsilon_t$$

for $t = 11$

$$x_{1,t} = \begin{bmatrix} \phi_4 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \end{bmatrix} + 0.5u_{1,t},$$
$$y_t = \phi_5 x_{1,t} + 0.2\varepsilon_t$$

and for $t = 12,..,T$

$$\begin{aligned} x_{1,t} &= \phi_4 + 0.5u_{1,t} \\ y_t &= \phi_5 x_{1,t} + 0.2\varepsilon_t \end{aligned}.$$

There are three sets of state transition matrices, whereas there are only two sets of the other coefficient matrices.

State-Space Model Creation

To create a standard or diffuse state-space model, use ssm or dssm, respectively. For time-invariant models, explicitly specify the parametric form of your state-space model by supplying the coefficient matrices. For time-variant, complex models, or models that require constraints, supply a parameter-to-matrix mapping function. The software can infer the type of state (stationary, the constant one, or nonstationary), but it is best practice to supply the state type using, for example, the `StateType` name-value pair argument.

To filter and smooth the states of a specified `ssm` or `dssm` model, the software uses the standard Kalman filter or the diffuse Kalman filter. To implement either, the software requires the parameters of the initial state distribution (x_0).

- For stationary states (`StateType` is `0`), the initial means, variances, and covariances are finite, and the software infers them. However, you can specify other values using the properties `Mean0` and `Cov0`, and dot notation.
- For states that are the constant one for all periods (`StateType` is `1`), the initial state means are 1 and covariances are 0.
- For nonstationary or diffuse states (`StateType` is `2`):
  - For standard state-space model, the initial state means are 0 and initial state variance is `1e7` by default. To specify an initial state covariance of `Inf`, create a `dssm` model object instead.
  - For diffuse state-space models, the initial state means are 0 and initial state variance is `Inf`.

See Also

dssm | esitmate | esitmate | filter | filter | forecast | forecast | smooth | smooth | ssm

Related Examples

- "Explicitly Create State-Space Model Containing Known Parameter Values" on page 8-17
- "Create State Space Model with Unknown Parameters" on page 8-20
- "Create State-Space Model Containing ARMA State" on page 8-24
- "Implicitly Create State-Space Model Containing Regression Component" on page 8-28
- "Implicitly Create Time-Varying State-Space Model" on page 8-32

More About

- "What Is the Kalman Filter?" on page 8-8

What Is the Kalman Filter?

In this section...

Standard Kalman Filter

In the state-space model framework, the Kalman filter estimates the values of a latent, linear, stochastic, dynamic process based on possibly mismeasured observations. Given distribution assumptions on the uncertainty, the Kalman filter also estimates model parameters via maximum likelihood.

Starting with initial values for states ($x_{0|0}$), the initial state variance-covariance matrix ($P_{0|0}$), and initial values for all unknown parameters (θ_0), the simple Kalman filter:

1. Estimates, for $t = 1,...,T$:
   a. The 1-step-ahead vector of state forecasts vector for period t ($\hat{x}_{t|t-1}$) and its variance-covariance matrix ($P_{t|t-1}$)
   b. The 1-step-ahead vector of observation forecasts for period t ($\hat{y}_{t|t-1}$) and its estimated variance-covariance matrix ($V_{t|t-1}$)
   c. The filtered states for period t ($\hat{x}_{t|t}$) and its estimated variance-covariance matrix ($P_{t|t}$)

2 Feeds the forecasted and filtered estimates into the data likelihood function

$$\ln p(y_T,...,y_1) = \sum_{t=1}^{T} \ln \phi(y_t; y_{t|t-1}, V_{t|t-1}),$$

where $\phi(y_t; y_{t|t-1}, V_{t|t-1})$ is the multivariate normal probability density function with mean $\hat{y}_{t|t-1}$ and variance $V_{t|t-1}$.

3 Feeds this procedure into an optimizer to maximize the likelihood with respect to the model parameters.

State Forecasts

s-step-ahead, state forecasts are estimates of the states at period t using all information (for example, the observed responses) up to period $t - s$.

The m_t-by-1 vector of 1-step-ahead, state forecasts at period t is $x_{t|t-1} = E(x_t \mid y_{t-1},...,y_1)$. The estimated vector of state forecasts is

$$\hat{x}_{t|t-1} = A_t \hat{x}_{t-1|t-1},$$

where $\hat{x}_{t-1|t-1}$ is the m_{t-1}-by-1 filtered state vector at period $t - 1$.

At period t, the 1-step-ahead, state forecasts have the variance-covariance matrix

$$P_{t|t-1} = A_t P_{t-1|t-1} A_t' + B_t B_t',$$

where $P_{t-1|t-1}$ is the estimated variance-covariance matrix of the filtered states at period $t - 1$, given all information up to period $t - 1$.

The corresponding 1-step-ahead forecasted observation is $\hat{y}_{t|t-1} = C_t \hat{x}_{t|t-1}$, and its variance-covariance matrix is $V_{t|t-1} = Var(y_t \mid y_{t-1},...,y_1) = C_t P_{t|t-1} C_t' + D_t D_t'$.

In general, the s-step-ahead, forecasted state vector is $x_{t|t-s} = E(x_t \mid y_{t-s},\ldots,y_1)$. The s-step-ahead, vector of state forecasts is

$$\hat{x}_{t+s|t} = \left(\prod_{j=t+1}^{t+s} A_j\right) x_{t|t}$$

and the s-step-ahead, forecasted observation vector is

$$\hat{y}_{t+s|t} = C_{t+s}\hat{x}_{t+s|t}.$$

Filtered States

State forecasts at period t, updated using all information (for example the observed responses) up to period t.

The m_t-by-1 vector of filtered states at period t is $x_{t|t} = E(x_t \mid y_t,\ldots,y_1)$. The estimated vector of filtered states is

$$\hat{x}_{t|t} = \hat{x}_{t|t-1} + K_t\hat{\varepsilon}_t,$$

where:

- $\hat{x}_{t|t-1}$ is the vector of state forecasts at period t using the observed responses from periods 1 through $t - 1$.
- K_t is the m_t-by-h_t raw Kalman gain matrix for period t.
- $\hat{\varepsilon}_t = y_t - C_t\hat{x}_{t|t-1}$ is the h_t-by-1 vector of estimated observation innovations at period t.

In other words, the filtered states at period t are the forecasted states at period t plus an adjustment based on the trustworthiness of the observation. Trustworthy observations have very little corresponding observation innovation variance (for example, the maximum eigenvalue of D_tD_t' is relatively small). Consequently, for a given estimated observation innovation, the term $K_t\hat{\varepsilon}_t$ has a higher impact on the values of the filtered states than untrustworthy observations.

At period t, the filtered states have variance-covariance matrix

$$P_{t|t} = P_{t|t-1} - K_t C_t P_{t|t-1}^{'},$$

where $P_{t|t-1}$ is the estimated variance-covariance matrix of the state forecasts at period t, given all information up to period $t - 1$.

Smoothed States

Smoothed states are estimated states at period t, which are updated using all available information (for example, all of the observed responses).

The m_t-by-1 vector of smoothed states at period t is $x_{t|T} = E(x_t \mid y_T, ..., y_1)$. The estimated vector of smoothed states is

$$\hat{x}_{t|T} = \hat{x}_{t|t-1} + P_{t|t-1} r_t,$$

where:

- $\hat{x}_{t|t-1}$ are the state forecasts at period t using the observed responses from periods 1 to $t - 1$.
- $P_{t|t-1}$ is the estimated variance-covariance matrix of the state forecasts, given all information up to period $t - 1$.
- $r_t = \sum_{s=t}^{T} \left\{ \left[\prod_{j=t}^{s-1} (A_t - K_t C_t) \right] C_s^{'} V_{s|s-1}^{-1} \nu_s \right\}$, where,
  - K_t is the m_t-by-h_t raw Kalman gain matrix for period t.
  - $V_{t|t-1} = C_t P_{t|t-1} C_t^{'} + D_t D_t^{'}$, which is the estimated variance-covariance matrix of the forecasted observations.
  - $\nu_t = y_t - \hat{y}_{t|t-1}$, which is the difference between the observation and its forecast at period t.

Smoothed State Disturbances

Smoothed state disturbances are estimated, state disturbances at period t, which are updated using all available information (for example, all of the observed responses).

The k_t-by-1 vector of smoothed, state disturbances at period t is $u_{t|T} = E(u_t \mid y_T,...,y_1)$. The estimated vector of smoothed, state disturbances is

$$\hat{u}_{t|T} = B_t' r_t,$$

where r_t is the variable in the formula to estimate the smoothed states.

At period t, the smoothed state disturbances have variance-covariance matrix

$$U_{t|T} = I - B_t' N_t B_t,$$

where N_t is the variable in the formula to estimate the variance-covariance matrix of the smoothed states.

The software computes smoothed estimates using backward recursion of the Kalman filter.

Forecasted Observations

s-step-ahead, forecasted observations are estimates of the observations at period t using all information (for example, the observed responses) up to period $t - s$.

The n_t-by-1 vector of 1-step-ahead, forecasted observations at period t is $y_{t|t-1} = E(y_t \mid y_{t-1},...,y_1)$. The estimated vector of forecasted observations is

$$\hat{y}_{t|t-1} = C_t \hat{x}_{t|t-1},$$

where $\hat{x}_{t|t-1}$ is the m_t-by-1 estimated vector of state forecasts at period t.

At period t, the 1-step-ahead, forecasted observations have variance-covariance matrix

$$V_{t|t-1} = Var\left(y_t \mid y_{t-1},\ldots,y_1\right) = C_t P_{t|t-1} C_t' + D_t D_t'.$$

where $P_{t|t-1}$ is the estimated variance-covariance matrix of the state forecasts at period t, given all information up to period $t - 1$.

In general, the s-step-ahead, vector of state forecasts is $x_{t|t-s} = E\left(x_t \mid y_{t-s},\ldots,y_1\right)$. The s-step-ahead, forecasted observation vector is

$$\hat{y}_{t+s|t} = C_{t+s}\hat{x}_{t+s|t}.$$

Smoothed Observation Innovations

Smoothed observation innovations are estimated, observation innovations at period t, which are updated using all available information (for example, all of the observed responses).

The h_t-by-1 vector of smoothed, observation innovations at period t is $\varepsilon_{t|T} = E\left(\varepsilon_t \mid y_T,\ldots,y_1\right)$. The estimated vector of smoothed, observation innovations is

$$\hat{\varepsilon}_t = D_t' V_{t|t-1}^{-1} v_t - D_t' K_t' r_{t+1},$$

where:

- r_t and v_t are the variables in the formula to estimate the smoothed states.
- K_t is the m_t-by-h_t raw Kalman gain matrix for period t.
- $V_{t|t-1} = C_t P_{t|t-1} C_t' + D_t D_t'$, which is the estimated variance-covariance matrix of the forecasted observations.

At period t, the smoothed observation innovations have variance-covariance matrix

$$E_{t|T} = I - D_t'\left(V_{t|t-1}^{-1} - K_t' N_{t+1} K_t\right) D_t.$$

The software computes smoothed estimates using backward recursion of the Kalman filter.

Kalman Gain

- The *raw Kalman gain* is a matrix that indicates how much to weigh the observations during recursions of the Kalman filter.

  The raw Kalman gain is an m_t-by-h_t matrix computed using

  $$K_t = P_{t|t-1}C_t'\left(C_tP_{t|t-1}C_t' + D_tD_t'\right)^{-1},$$

  where $P_{t|t-1}$ is the estimated variance-covariance matrix of the state forecasts, given all information up to period $t - 1$.

  The value of the raw Kalman gain determines how much weight to put on the observations. For a given estimated observation innovation, if the maximum eigenvalue of D_tD_t' is relatively small, then the raw Kalman gain imparts a relatively large weight on the observations. If the maximum eigenvalue of D_tD_t' is relatively large, then the raw Kalman gain imparts a relatively small weight on the observations. Consequently, the filtered states at period t are close to the corresponding state forecasts.
- Consider obtaining the 1-step-ahead state forecasts for period $t + 1$ using all information up to period t. The *adjusted Kalman gain* ($K_{adj,t}$) is the amount of weight put on the estimated observation innovation for period t ($\hat{\varepsilon}_t$) as compared to the 2-step-ahead state forecast ($\hat{x}_{t+1|t-1}$).

  That is,

  $$\hat{x}_{t+1|t} = A_t\hat{x}_{t|t} = A_t\hat{x}_{t|t-1} + A_tK_t\hat{\varepsilon}_t = \hat{x}_{t+1|t-1} + K_{adj,t}\hat{\varepsilon}_t.$$

Backward Recursion of the Kalman Filter

Backward recursion of the Kalman filter estimates smoothed states, state disturbances, and observation innovations.

The software estimates the smoothed values by:

1 Setting $r_{T+1} = 0$, and N_{T+1} to an m_T-by-m_T matrix of 0s

2 For $t = T,...,1$, it recursively computes:

   a r_t (see "Smoothed States" on page 8-11)

   b $\hat{x}_{t|T}$, which is the matrix of smoothed states

   c N_t (see "Smoothed States" on page 8-11)

   d $P_{t|T}$, which is the estimated variance-covariance matrix of the smoothed states

   e $\hat{u}_{t|T}$, which is the matrix of smoothed state disturbances

   f $U_{t|T}$, which is the estimated variance-covariance matrix of the smoothed state disturbances

   g $\hat{\varepsilon}_{t|T}$, which is the matrix of smoothed observation innovations

   h $E_{t|T}$, which is the estimated variance-covariance matrix of the smoothed observation innovations

Diffuse Kalman Filter

Consider a state-space model written so that the m diffuse states (x_d) are segregated from the n stationary states (x_s). That is, the moments of the initial distributions are

$$\mu_0 = \begin{bmatrix} \mu_{d0} \\ \mu_{s0} \end{bmatrix} \text{ and } \Sigma_0 = \begin{bmatrix} \Sigma_{d0} & 0 \\ 0 & \Sigma_{s0} \end{bmatrix}.$$

- μ_{d0} is an m-vector of zeros
- μ_{s0} is an n-vector of real numbers
- $\Sigma_{d0} = \kappa I_m$, where I_m is the m-by-m identity matrix and κ is a positive real number.
- Σ_{s0} is an n-by-n positive definite matrix.
- The diffuse states are uncorrelated with each other and the stationary states.

One way to analyze such a model is by setting κ to a relatively large, positive real number, and then implement the standard Kalman filter (see ssm). This treatment is

an approximation to an analysis that treats the diffuse states as if their initial state covariance approaches infinity.

The *diffuse Kalman filter* or *exact-inital Kalman filter* [33] treats the diffuse states by taking κ to ∞. The diffuse Kalman filter filters in two stages: the first stage initializes the model so that it can subsequently be filtered using the standard Kalman filter, which is the second stage. The initialization stage mirrors the standard Kalman filter. It sets all initial filtered states to zero, and then augments that vector of initial filtered states with the identity matrix, which composes an $(m + n)$-by-$(m + n + 1)$ matrix. After a sufficient number of periods, the precision matrices become nonsingular. That is, the diffuse Kalman filter uses enough periods at the beginning of the series to initialize the model. You can consider this period as presample data.

The second stage commences when the precision matrices are nonsingular. Specifically, the initialization stage returns a vector of filtered states and their precision matrix. Then, the standard Kalman filter uses those estimates and the remaining data to filter, smooth, and estimate parameters. For more details, see dssm and [33], Sec. 5.2.

References

[1] Durbin J., and S. J. Koopman. *Time Series Analysis by State Space Methods*. 2nd ed. Oxford: Oxford University Press, 2012.

See Also

dssm | esitmate | esitmate | filter | filter | forecast | forecast | smooth | smooth | ssm

More About

- "What Are State-Space Models?" on page 8-3

Explicitly Create State-Space Model Containing Known Parameter Values

This example shows how to create a time-invariant, state-space model containing known parameter values using `ssm`.

Define a state-space model containing two independent, AR(1) states with Gaussian disturbances that have standard deviations 0.1 and 0.3, respectively. Specify that the observation is the deterministic sum of the two states. Symbolically, the equation is

$$\begin{bmatrix} x_{t,1} \\ x_{t,2} \end{bmatrix} = \begin{bmatrix} 0.5 & 0 \\ 0 & -0.2 \end{bmatrix} \begin{bmatrix} x_{t-1,1} \\ x_{t-1,2} \end{bmatrix} + \begin{bmatrix} 0.1 & 0 \\ 0 & 0.3 \end{bmatrix} \begin{bmatrix} u_{t,1} \\ u_{t,2} \end{bmatrix}$$

$$y_t = \begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} x_{t,1} \\ x_{t,2} \end{bmatrix}.$$

Specify the state-transition coefficient matrix.

```
A = [0.5 0; 0 -0.2];
```

Specify the state-disturbance-loading coefficient matrix.

```
B = [0.1 0; 0 0.3];
```

Specify the measurement-sensitivity coefficient matrix.

```
C = [1 1];
```

Define the state-space model using `ssm`.

```
Mdl = ssm(A,B,C)
```

```
Mdl =


State-space model type: <a href="matlab: doc ssm">ssm</a>

State vector length: 2
Observation vector length: 1
State disturbance vector length: 2
```

```
Observation innovation vector length: 0
Sample size supported by model: Unlimited

State variables: x1, x2,...
State disturbances: u1, u2,...
Observation series: y1, y2,...
Observation innovations: e1, e2,...

State equations:
x1(t) = (0.50)x1(t-1) + (0.10)u1(t)
x2(t) = -(0.20)x2(t-1) + (0.30)u2(t)

Observation equation:
y1(t) = x1(t) + x2(t)

Initial state distribution:

Initial state means
 x1  x2
  0   0

Initial state covariance matrix
     x1    x2
 x1  0.01   0
 x2   0    0.09

State types
     x1          x2
 Stationary  Stationary
```

`Mdl` is an `ssm` model containing unknown parameters. A detailed summary of `Mdl` prints to the Command Window. By defualt, the software sets the initial state means and covariance matrix using the stationary distributions.

It is good practice to verify that the state and observations equations are correct. If the equations are not correct, then it might help to expand the state-space equation by hand.

Simulate states or observations from `Mdl` using `simulate`, or forecast states or observations using `forecast`.

See Also

disp | estimate | ssm

Related Examples

- "Explicitly Create State-Space Model Containing Unknown Parameters" on page 8-20
- "Create State-Space Model Containing ARMA State" on page 8-24
- "Implicitly Create Time-Invariant State-Space Model" on page 8-22
- "Implicitly Create Time-Varying State-Space Model" on page 8-32
- "Estimate Time-Invariant State-Space Model" on page 8-41
- "Create State-Space Model with Random State Coefficient" on page 8-38

More About

- "What Are State-Space Models?" on page 8-3

Create State Space Model with Unknown Parameters

In this section...

Explicitly Create State-Space Model Containing Unknown Parameters

This example shows how to create a time-invariant, state-space model containing unknown parameter values using `ssm`.

Define a state-space model containing two dependent MA(1) states, and an additive-error observation model. Symbolically, the equation is

$$\begin{bmatrix} x_{t,1} \\ x_{t,2} \\ x_{t,3} \\ x_{t,4} \end{bmatrix} = \begin{bmatrix} 0 & \theta_1 & \lambda_1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \theta_3 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_{t-1,1} \\ x_{t-1,2} \\ x_{t-1,3} \\ x_{t-1,4} \end{bmatrix} + \begin{bmatrix} \sigma_1 & 0 \\ 1 & 0 \\ 0 & \sigma_2 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} u_{t,1} \\ u_{t,2} \end{bmatrix}$$

$$y_t = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_{t,1} \\ x_{t,2} \\ x_{t,3} \\ x_{t,4} \end{bmatrix} + \begin{bmatrix} \sigma_3 & 0 \\ 0 & \sigma_4 \end{bmatrix} \begin{bmatrix} \varepsilon_{t,1} \\ \varepsilon_{t,2} \end{bmatrix}.$$

Note that the states $x_{t,1}$ and $x_{t,3}$ are the two dependent MA(1) processes. The states $x_{t,2}$ and $x_{t,4}$ help construct the lag-one, MA effects. For example, $x_{t,2}$ picks up the first disturbance ($u_{t,1}$), and $x_{t,1}$ picks up $x_{t-1,2} = u_{t-1,1}$. In all, $x_{t,1} = \lambda_1 x_{t-1,3} + u_{t,1} + \theta_1 u_{t-1,1}$, which is an MA(1) with $x_{t-1,3}$ as an input.

Specify the state-transition coefficient matrix. Use `NaN` values to indicate unknown parameters.

```
A = [0 NaN NaN 0; 0 0 0 0; 0 0 0 NaN; 0 0 0 0];
```

Specify the state-disturbance-loading coefficient matrix.

```
B = [NaN 0; 1 0; 0 NaN; 0 1];
```

Specify the measurement-sensitivity coefficient matrix.

```
C = [1 0 0 0; 0 0 1 0];
```

Specify the observation-innovation coefficient matrix.

```
D = [NaN 0; 0 NaN];
```

Use `ssm` to define the state-space model.

```
Mdl = ssm(A,B,C,D)
```

```
Mdl =


State-space model type: <a href="matlab: doc ssm">ssm</a>

State vector length: 4
Observation vector length: 2
State disturbance vector length: 2
Observation innovation vector length: 2
Sample size supported by model: Unlimited
Unknown parameters for estimation: 7

State variables: x1, x2,...
State disturbances: u1, u2,...
Observation series: y1, y2,...
Observation innovations: e1, e2,...
Unknown parameters: c1, c2,...

State equations:
x1(t) = (c1)x2(t-1) + (c2)x3(t-1) + (c4)u1(t)
x2(t) = u1(t)
x3(t) = (c3)x4(t-1) + (c5)u2(t)
x4(t) = u2(t)

Observation equations:
y1(t) = x1(t) + (c6)e1(t)
y2(t) = x3(t) + (c7)e2(t)

Initial state distribution:

Initial state means are not specified.
Initial state covariance matrix is not specified.
```

```
State types are not specified.
```

`Mdl` is an `ssm` model containing unknown parameters. A detailed summary of `Mdl` prints to the Command Window. It is good practice to verify that the state and observations equations are correct.

Pass `Mdl` and data to `estimate` to estimate the unknown parameters.

Implicitly Create Time-Invariant State-Space Model

This example shows how to create a time-invariant state-space model by passing a parameter-mapping function describing the model to `ssm` (that is, *implicitly* create a state-space model). The state model is AR(1) model. The states are observed with bias, but without random error. Set the initial state mean and variance, and specify that the state is stationary.

Write a function that specifies how the parameters in `params` map to the state-space model matrices, the initial state values, and the type of state.

```
% Copyright 2015 The MathWorks, Inc.

function [A,B,C,D,Mean0,Cov0,StateType] = timeInvariantParamMap(params)
% Time-invariant state-space model parameter mapping function example. This
% function maps the vector params to the state-space matrices (A, B, C, and
% D), the initial state value and the initial state variance (Mean0 and
% Cov0), and the type of state (StateType). The state model is AR(1)
% without observation error.
    varu1 = exp(params(2)); % Positive variance constraint
    A = params(1);
    B = sqrt(varu1);
    C = params(3);
    D = [];
    Mean0 = 0.5;
    Cov0 = 100;
    StateType = 0;
end
```

Save this code as a file named `timeInvariantParamMap` to a folder on your MATLAB® path.

Create the state-space model by passing the function `timeInvariantParamMap` as a function handle to `ssm`.

```
Mdl = ssm(@timeInvariantParamMap);
```

The software implicitly defines the state-space model. Usually, you cannot verify state-space models that you implicitly define.

`Mdl` is an `ssm` model object containing unknown parameters. You can estimate the unknown parameters by passing `Mdl` and response data to `estimate`.

See Also

disp | estimate | ssm

Related Examples

- "Explicitly Create State-Space Model Containing Known Parameter Values" on page 8-17
- "Create State-Space Model Containing ARMA State" on page 8-24
- "Implicitly Create Time-Varying State-Space Model" on page 8-32
- "Create State-Space Model with Random State Coefficient" on page 8-38

More About

- "What Are State-Space Models?" on page 8-3

Create State-Space Model Containing ARMA State

This example shows how to create an stationary ARMA model subject to measurement error using `ssm`.

To explicitly create a state-space model, it is helpful to write the state and observation equations in matrix form. In this example, the state of interest is the ARMA(2,1) process

$$x_t = c + \phi_1 x_{t-1} + \phi_2 x_{t-2} + u_t + \theta_1 u_{t-1},$$

where u_t is Gaussian with mean 0 and known standard deviation 0.5.

The variables x_t, x_{t-1}, and u_t are in the state-space model framework. Therefore, the terms c, $\phi_2 x_{t-2}$, and $\theta_1 u_{t-1}$ require "dummy states" to be included in the model.

Subsequently, the state equation is

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \\ x_{3,t} \\ x_{4,t} \end{bmatrix} = \begin{bmatrix} \phi_1 & c & \phi_2 & \theta_1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \\ x_{3,t-1} \\ x_{4,t-1} \end{bmatrix} + \begin{bmatrix} 0.5 \\ 0 \\ 0 \\ 1 \end{bmatrix} u_{1,t}$$

Note that:

- c corresponds to a state ($x_{2,t}$) that is always `1`.
- $x_{3,t} = x_{1,t-1}$, and $x_{1,t}$ has the term $\phi_2 x_{3,t-1} = \phi_2 x_{1,t-2}$.
- $x_{1,t}$ has the term $0.5 u_{1,t}$. `ssm` puts state disturbances as Gaussian random variables with mean 0 and variance 1. Therefore, the factor `0.5` is the standard deviation of the state disturbance.
- $x_{4,t} = u_{1,t}$, and $x_{1,t}$ has the term $\theta_1 x_{4,t} = \theta_1 u_{1,t-1}$.

The observation equation is unbiased for the ARMA(2,1) state process. The observation innovations are Gaussian with mean 0 and known standard deviation 0.1. Symbolically, the observation equation is

$$y_t = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t} \\ x_{2,t} \\ x_{3,t} \\ x_{4,t} \end{bmatrix} + 0.1\varepsilon_t.$$

You can include a measurement-sensitivity factor (a bias) by replacing `1` in the row vector by a scalar or unknown parameter.

Define the state-transition coefficient matrix. Use `NaN` values to indicate unknown parameters.

```
A = [NaN NaN NaN NaN; 0 1 0 0; 1 0 0 0; 0 0 0 0];
```

Define the state-disturbance-loading coefficient matrix.

```
B = [0.5; 0; 0; 1];
```

Define the measurement-sensitivity coefficient matrix.

```
C = [1 0 0 0];
```

Define the observation-innovation coefficient matrix.

```
D = 0.1;
```

Use `ssm` to create the state-space model. Set the initial-state mean (`Mean0`) to a vector of zeros and covariance matrix (`Cov0`) to the identity matrix, except set the mean and variance of the constant state to `1` and `0`, respectively. Specify the type of initial state distributions (`StateType`) by noting that:

- $x_{1,t}$ is a stationary, ARMA(2,1) process.
- $x_{2,t}$ is the constant 1 for all periods.
- $x_{3,t}$ is the lagged ARMA process, so it is stationary.
- $x_{4,t}$ is a white-noise process, so it is stationary.

```
Mean0 = [0; 1; 0; 0];
Cov0 = eye(4);
Cov0(2,2) = 0;
StateType = [0; 1; 0; 0];
Mdl = ssm(A,B,C,D,'Mean0',Mean0,'Cov0',Cov0,'StateType',StateType);
```

`Mdl` is an `ssm` model. You can use dot notation to access its properties. For example, print `A` by entering `Mdl.A`.

Use `disp` to verify the state-space model.

```
disp(Mdl)
```

```
State-space model type: <a href="matlab: doc ssm">ssm</a>
```

```
State vector length: 4
Observation vector length: 1
State disturbance vector length: 1
Observation innovation vector length: 1
Sample size supported by model: Unlimited
Unknown parameters for estimation: 4

State variables: x1, x2,...
State disturbances: u1, u2,...
Observation series: y1, y2,...
Observation innovations: e1, e2,...
Unknown parameters: c1, c2,...

State equations:
x1(t) = (c1)x1(t-1) + (c2)x2(t-1) + (c3)x3(t-1) + (c4)x4(t-1) + (0.50)u1(t)
x2(t) = x2(t-1)
x3(t) = x1(t-1)
x4(t) = u1(t)

Observation equation:
y1(t) = x1(t) + (0.10)e1(t)

Initial state distribution:

Initial state means
 x1  x2  x3  x4
  0   1   0   0

Initial state covariance matrix
     x1  x2  x3  x4
 x1   1   0   0   0
 x2   0   0   0   0
 x3   0   0   1   0
 x4   0   0   0   1

State types
     x1         x2         x3          x4
 Stationary  Constant  Stationary  Stationary
```

If you have a set of responses, you can pass them and `Mdl` to `estimate` to estimate the parameters.

See Also

disp | estimate | ssm

Related Examples

- "Explicitly Create State-Space Model Containing Known Parameter Values" on page 8-17
- "Explicitly Create State-Space Model Containing Unknown Parameters" on page 8-20
- "Implicitly Create Time-Invariant State-Space Model" on page 8-22
- "Implicitly Create Time-Varying State-Space Model" on page 8-32
- "Estimate Time-Invariant State-Space Model" on page 8-41
- "Create State-Space Model with Random State Coefficient" on page 8-38

More About

- "What Are State-Space Models?" on page 8-3

Implicitly Create State-Space Model Containing Regression Component

This example shows how to implicitly create a state-space model that contains a regression component in the observation equation. The state model is an ARMA(1,1).

Write a function that specifies how the parameters in `params` map to the state-space model matrices, the initial state values, and the type of state. Specify the regression component by deflating the observations within the function. Symbolically, the model is:

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix} = \begin{bmatrix} \phi_1 & \theta_1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \end{bmatrix} + \begin{bmatrix} \sigma_1 \\ 1 \end{bmatrix} u_{1t}$$
$$y_t - \beta z_t = a x_{1,t} + \sigma_2 \varepsilon_t.$$

```
% Copyright 2015 The MathWorks, Inc.

function [A,B,C,D,Mean0,Cov0,StateType,DeflateY] = regressionParamMap(params,y,z)
% State-space model with a regression component parameter mapping function
% example. This function maps the vector params to the state-space matrices
% (A, B, C, and D), the initial state value and the initial state variance
% (Mean0 and Cov0), and the type of state (StateType). The state model is
% an ARMA(1,1).
    varu1 = exp(params(3)); % Positive variance constraint
    vare1 = exp(params(4));
    A = [params(1) params(2); 0 0];
    B = [sqrt(varu1); 1];
    C = [1 0];
    D = sqrt(vare1);
    Mean0 = [0.5 0.5];
    Cov0 = eye(2);
    StateType = [0 0];
    DeflateY = y - params(5)*z;
end
```

Save this code as a file named `regressionParamMap` on your MATLAB® path.

Create the state-space model by passing the function `regressionParamMap` as a function handle to `ssm`.

```
Mdl = ssm(@(params)regressionParamMap(params,y,z));
```

`ssm` implicitly creates the state-space model. Usually, you cannot verify implicitly defined state-space models.

Before creating the model, ensure that the data `y` and `z` exist in your workspace.

See Also

disp | estimate | ssm

Related Examples

- "Implicitly Create Time-Invariant State-Space Model" on page 8-22
- "Implicitly Create Time-Varying State-Space Model" on page 8-32
- "Estimate State-Space Model Containing Regression Component" on page 8-55
- "Create State-Space Model with Random State Coefficient" on page 8-38

More About

- "What Are State-Space Models?" on page 8-3

Implicitly Create Diffuse State-Space Model Containing Regression Component

This example shows how to implicitly create a diffuse state-space model that contains a regression component in the observation equation. The state model contains an ARMA(1,1) state and random walk.

Write a function that specifies how the parameters in `params` map to the state-space model matrices, to the initial state values, and to the type of state. Specify the regression component by deflating the observations within the function. Symbolically, the model is:

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \\ x_{3,t} \end{bmatrix} = \begin{bmatrix} \phi_1 & \theta_1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \\ x_{3,t-1} \end{bmatrix} + \begin{bmatrix} \sigma_1 & 0 \\ 1 & 0 \\ 0 & \sigma_3 \end{bmatrix} \begin{bmatrix} u_{1,t} \\ u_{3,t} \end{bmatrix}$$

$$y_t - \beta z_t = a_1 x_{1,t} + a_2 x_{3,t} + \sigma_2 \varepsilon_t.$$

```
% Copyright 2015 The MathWorks, Inc.

function [A,B,C,D,Mean0,Cov0,StateType,DeflateY] = diffuseRegressionParamMap(params,y,z
% Diffuse state-space model with a regression component parameter mapping
% function example. This function maps the vector params to the state-space
% matrices (A, B, C, and D) and indicates the type of states (StateType).
% The state model contains an ARMA(1,1) model and a random walk.
    varu1 = exp(params(3)); % Positive variance constraint
    vare1 = exp(params(5));
    A = [params(1) params(2); 0 0];
    B = [sqrt(varu1) 0; 1 0];
    C = [varu1 0];
    D = sqrt(vare1);
    Mean0 = []; % Let software infer Mean0
    Cov0 = [];  % Let software infer Cov0
    StateType = [0 0 2];
    DeflateY = y - params(6)*z;
end
```

Save this code as a file named `diffuseRegressionParamMap.m` to a folder on your MATLAB® path.

Create the diffuse state-space model by passing `diffuseRegressionParamMap` as a function handle to `dssm`.

```
Mdl = dssm(@(params)diffuseRegressionParamMap(params,y,z));
```

`dssm` implicitly creates the diffuse state-space model. Usually, you cannot verify implicitly defined state-space models.

Before creating the model, ensure that the variables `y` and `z` exist in your workspace.

See Also

dssm

More About

- "What Are State-Space Models?" on page 8-3

Implicitly Create Time-Varying State-Space Model

This example shows how to create a time-varying, state-space model by passing a parameter-mapping function describing the model to `ssm` (i.e., *implicitly* create a state-space model).

Suppose that from periods 1 through 10, the state model are stationary AR(2) and MA(1) models, respectively, and the observation model is the sum of the two states. From periods 11 through 20, the state model only includes the first AR(2) model.

Symbolically, the models are:

$$\begin{bmatrix} x_{1t} \\ x_{2t} \\ x_{3t} \\ x_{4t} \end{bmatrix} = \begin{bmatrix} \phi_1 & \phi_2 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \theta \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \\ x_{3,t-1} \\ x_{4,t-1} \end{bmatrix} + \begin{bmatrix} \sigma_1 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} u_{1t} \\ u_{2t} \end{bmatrix} \text{ for } t = 1, \ldots, 10,$$

$$y_t = a_1 (x_{1t} + x_{3t}) + \sigma_2 \varepsilon_t.$$

$$\begin{bmatrix} x_{1t} \\ x_{2t} \end{bmatrix} = \begin{bmatrix} \phi_1 & \phi_2 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \\ x_{3,t-1} \\ x_{4,t-1} \end{bmatrix} + \begin{bmatrix} \sigma_1 \\ 0 \end{bmatrix} u_{1t} \text{ for } t = 11,$$

$$y_t = a_2 x_{1t} + \sigma_3 \varepsilon_t.$$

$$\begin{bmatrix} x_{1t} \\ x_{2t} \end{bmatrix} = \begin{bmatrix} \phi_1 & \phi_2 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \end{bmatrix} + \begin{bmatrix} \sigma_1 \\ 0 \end{bmatrix} u_{1t} \text{ for } t = 12, \ldots, 20.$$

$$y_t = a_2 x_{1t} + \sigma_3 \varepsilon_t.$$

Write a function that specifies how the parameters in `params` map to the state-space model matrices, the initial state values, and the type of state.

```
% Copyright 2015 The MathWorks, Inc.

function [A,B,C,D,Mean0,Cov0,StateType] = timeVariantParamMap(params)
% Time-variant state-space model parameter mapping function example. This
% function maps the vector params to the state-space matrices (A, B, C, and
```

```
% D), the initial state value and the initial state variance (Mean0 and
% Cov0), and the type of state (StateType). From periods 1 through 10, the
% state model is an AR(2) and an MA(1) model, and the observation model is
% the sum of the two states. From periods 11 through 20, the state model is
% just the AR(2) model.
    varu11 = exp(params(3));  % Positive variance constraints
    vare11 = exp(params(6));
    vare12 = exp(params(8));
    A1 = {[params(1) params(2) 0 0; 1 0 0 0; 0 0 0 params(4); 0 0 0 0]};
    B1 = {[sqrt(varu11) 0; 0 0; 0 1; 0 1]};
    C1 = {params(5)*[1 0 1 0]};
    D1 = {sqrt(vare11)};
    Mean0 = [0.5 0.5 0 0];
    Cov0 = eye(4);
    StateType = [0 0 0 0];
    A2 = {[params(1) params(2) 0 0; 1 0 0 0]};
    B2 = {[sqrt(varu11); 0]};
    A3 = {[params(1) params(2); 1 0]};
    B3 = {[sqrt(varu11); 0]};
    C3 = {params(7)*[1 0]};
    D3 = {sqrt(vare12)};
    A = [repmat(A1,10,1);A2;repmat(A3,9,1)];
    B = [repmat(B1,10,1);B2;repmat(B3,9,1)];
    C = [repmat(C1,10,1);repmat(C3,10,1)];
    D = [repmat(D1,10,1);repmat(D3,10,1)];
end
```

Save this code as a file named `timeVariantParamMap.m` on your MATLAB® path.

Create the state-space model by passing the function `timeVariantParamMap` as a function handle to `ssm`.

```
Mdl = ssm(@timeVariantParamMap);
```

`ssm` implicitly creates the state-space model. Usually, you cannot verify implicitly created state-space models.

`Mdl` is an `ssm` model object containing unknown parameters. You can estimate the unknown parameters by passing `Mdl` and response data to `estimate`.

See Also

disp | estimate | ssm

Related Examples

- “Implicitly Create Time-Invariant State-Space Model” on page 8-22
- “Estimate Time-Varying State-Space Model” on page 8-45
- “Create State-Space Model with Random State Coefficient” on page 8-38

More About

- “What Are State-Space Models?” on page 8-3

Implicitly Create Time-Varying Diffuse State-Space Model

This example shows how to create a diffuse state-space model in which one of the state variables drops out of the model after a certain period.

Suppose that a latent process comprises an AR(2) and an MA(1) model. There are 50 periods, and the MA(1) process drops out of the model for the final 25 periods. Consequently, the state equation for the first 25 periods is

$$x_{1,t} = \phi_1 x_{1,t-1} - \phi_2 x_{1,t-2} + u_{1,t}$$
$$x_{2,t} = u_{2,t} + \theta_1 u_{2,t-1},$$

and for the last 25 periods, it is

$$x_{1,t} = \phi_1 x_{1,t-1} - \phi_2 x_{1,t-2} + u_{1,t},$$

where $u_{1,t}$ and $u_{2,t}$ are Gaussian with mean 0 and standard deviation 1.

The latent processes are measured using

$$y_t = a\left(x_{1,t} + x_{2,t}\right) + \varepsilon_t,$$

for the first 25 periods, and

$$y_t = b x_{1,t} + \varepsilon_t$$

for the last 25 periods, where ε_t is Gaussian with mean 0 and standard deviation 1.

Together, the latent process and observation equations make up a state-space model. If the coefficients are unknown parameters, the state-space model is

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \\ x_{3,t} \\ x_{4,t} \end{bmatrix} = \begin{bmatrix} \phi_1 & \phi_2 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \theta_1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \\ x_{3,t-1} \\ x_{4,t-1} \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} u_{1,t} \\ u_{2,t} \end{bmatrix}$$
$$y_t = a(x_{1,t} + x_{3,t}) + \varepsilon_t$$

for the first 25 periods,

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix} = \begin{bmatrix} \phi_1 & \phi_2 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \\ x_{3,t-1} \\ x_{4,t-1} \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} u_{1,t}$$

$$y_t = b x_{1,t} + \varepsilon_t$$

for period 26, and

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix} = \begin{bmatrix} \phi_1 & \phi_2 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} u_{1,t}$$

$$y_t = b x_{1,t} + \varepsilon_t$$

for the last 24 periods.

Write a function that specifies how the parameters in `params` map to the state-space model matrices, the initial state values, and the type of state.

```
% Copyright 2015 The MathWorks, Inc.

function [A,B,C,D,Mean0,Cov0,StateType] = diffuseAR2MAParamMap(params,T)
%diffuseAR2MAParamMap Time-variant diffuse state-space model parameter
%mapping function
%
% This function maps the vector params to the state-space matrices (A, B,
% C, and D) and the type of state (StateType). From periods 1 to T/2, the
% state model is an AR(2) and an MA(1) model, and the observation model is
% the sum of the two states. From periods T/2 + 1 to T, the state model is
% just the AR(2) model.  The AR(2) model is diffuse.
    A1 = {[params(1) params(2) 0 0; 1 0 0 0; 0 0 0 params(3); 0 0 0 0]};
    B1 = {[1 0; 0 0; 0 1; 0 1]};
    C1 = {params(4)*[1 0 1 0]};
    Mean0 = [];
    Cov0 = [];
    StateType = [2 2 0 0];
    A2 = {[params(1) params(2) 0 0; 1 0 0 0]};
    B2 = {[1; 0]};
    A3 = {[params(1) params(2); 1 0]};
    B3 = {[1; 0]};
    C3 = {params(5)*[1 0]};
    A = [repmat(A1,T/2,1);A2;repmat(A3,(T-2)/2,1)];
```

```
    B = [repmat(B1,T/2,1);B2;repmat(B3,(T-2)/2,1)];
    C = [repmat(C1,T/2,1);repmat(C3,T/2,1)];
    D = 1;
end
```

Save this code as a file named `diffuseAR2MAParamMap` on your MATLAB® path.

Create the diffuse state-space model by passing the function `diffuseAR2MAParamMap` as a function handle to `dssm`. This example uses 50 observations.

```
T = 50;
Mdl = dssm(@(params)diffuseAR2MAParamMap(params,T));
```

`dssm` implicitly creates the diffuse state-space model. Usually, you cannot verify diffuse state-space models that are implicitly created.

`dssm` contains unknown parameters. You can simulate data and then estimate parameters using `estimate`.

See Also

dssm | esitmate | filter

Related Examples

- “Estimate Time-Varying Diffuse State-Space Model” on page 8-50
- “Filter Time-Varying Diffuse State-Space Model” on page 8-68

More About

- “What Are State-Space Models?” on page 8-3
- “What Is the Kalman Filter?” on page 8-8

Create State-Space Model with Random State Coefficient

This example shows how to create a time-varying, state-space model containing a random, state coefficient.

Write a function that specifies how the parameters in `params` map to the state-space model matrices, the initial state values, and the type of state. Symbolically, the model is

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & \phi \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 0 & \sigma \end{bmatrix} \begin{bmatrix} u_{1,t} \\ u_{2,t} \end{bmatrix}$$

$$y_t = \begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix} + \varepsilon_t .$$

ϕ is a random coefficient.

```
% Copyright 2015 The MathWorks, Inc.

function [A,B,C,D] = randomCoeffParamMap(c)
% State-space model parameter-to-matrix mapping function with a random
% coefficient example.  There are two states: one is a random walk with
% disturbance variance 1, and the other is a first-order Markov model with
% a random coefficient and an unknown variance.  The observation equation
% is the sum of the two states, and the innovation has variance 1.
A = diag([1,c(1)*rand]);
B = [1 0; 0 c(2)];
C = [1,1];
D = 1;
end
```

Create the state-space model by passing `randomCoeffParamMap` as a function handle to `ssm`.

```
rng('default'); % For reproducibility
Mdl = ssm(@randomCoeffParamMap);
```

`ssm` implicitly creates the `ssm` model `Mdl`.

Display `Mdl` using `disp`. Specify initial parameters values.

```
disp(Mdl,[3; 5])
```

```
State-space model type: <a href="matlab: doc ssm">ssm</a>

State vector length: 2
Observation vector length: 1
State disturbance vector length: 2
Observation innovation vector length: 1
Sample size supported by model: Unlimited

State variables: x1, x2,...
State disturbances: u1, u2,...
Observation series: y1, y2,...
Observation innovations: e1, e2,...

State equations:
x1(t) = x1(t-1) + u1(t)
x2(t) = (0.38)x2(t-1) + (5)u2(t)

Observation equation:
y1(t) = x1(t) + x2(t) + e1(t)

Initial state distribution:

Initial state means
 x1  x2
  0   0

Initial state covariance matrix
     x1     x2
 x1  1e+07  0
 x2  0      1e+07

State types
    x1       x2
 Diffuse  Diffuse
```

`disp` sets the parameters to their initial values, or functions of their initial values. In this case, the first parameter is the initial values times a random number.

See Also

disp | ssm

Related Examples

- “Implicitly Create Time-Invariant State-Space Model” on page 8-22
- “Implicitly Create Time-Varying State-Space Model” on page 8-32
- “Estimate State-Space Model Containing Regression Component” on page 8-55

More About

- “What Are State-Space Models?” on page 8-3

Estimate Time-Invariant State-Space Model

This example shows how to generate data from a known model, specify a state-space model containing unknown parameters corresponding to the data generating process, and then fits the state-space model to the data.

Suppose that a latent process is this AR(1) process

$$x_t = 0.5x_{t-1} + u_t,$$

where u_t is Gaussian with mean 0 and standard deviation 1.

Generate a random series of 100 observations from x_t, assuming that the series starts at 1.5.

```
T = 100;
ARMdl = arima('AR',0.5,'Constant',0,'Variance',1);
x0 = 1.5;
rng(1); % For reproducibility
x = simulate(ARMdl,T,'Y0',x0);
```

Suppose further that the latent process is subject to additive measurement error as indicated in the equation

$$y_t = x_t + \varepsilon_t,$$

where ε_t is Gaussian with mean 0 and standard deviation 0.1.

Use the random latent state process (`x`) and the observation equation to generate observations.

```
y = x + 0.1*randn(T,1);
```

Together, the latent process and observation equations compose a state-space model. Supposing that the coefficients and variances are unknown parameters, the state-space model is

$$\begin{aligned} x_t &= \phi x_{t-1} + \sigma_1 u_t \\ y_t &= x_t + \sigma_2 \varepsilon_t. \end{aligned}$$

Specify the state-transition coefficient matrix. Use `NaN` values for unknown parameters.

```
A = NaN;
```

Specify the state-disturbance-loading coefficient matrix.

```
B = NaN;
```

Specify the measurement-sensitivity coefficient matrix.

```
C = 1;
```

Specify the observation-innovation coefficient matrix

```
D = NaN;
```

Specify the state-space model using the coefficient matrices. Also, specify the initial state mean, variance, and distribution (which is stationary).

```
Mean0 = 0;
Cov0 = 10;
StateType = 0;
Mdl = ssm(A,B,C,D,'Mean0',Mean0,'Cov0',Cov0,'StateType',StateType);
```

`Mdl` is an `ssm` model. Verify that the model is correctly specified using the display in the Command Window.

Pass the observations to estimate to estimate the parameter. Set a starting value for the parameter to `params0`. σ_1 and σ_2 must be positive, so set the lower bound constraints using the `'lb'` name-value pair argument. Specify that the lower bound of ϕ is `-Inf`.

```
params0 = [0.9; 0.5; 0.1];
EstMdl = estimate(Mdl,y,params0,'lb',[-Inf; 0; 0])
```

```
Method: Maximum likelihood (fmincon)
Sample size: 100
Logarithmic  likelihood:     -140.532
Akaike   info criterion:      287.064
Bayesian info criterion:      294.879
      |     Coeff      Std Err   t Stat    Prob
-------------------------------------------------
 c(1) | 0.45425       0.19870   2.28612  0.02225
 c(2) | 0.89013       0.30359   2.93205  0.00337
 c(3) | 0.38750       0.57857   0.66976  0.50302
      |
      |  Final State   Std Dev   t Stat    Prob
 x(1) | 1.52989       0.35621   4.29496  0.00002

EstMdl =
```

```
State-space model type: <a href="matlab: doc ssm">ssm</a>

State vector length: 1
Observation vector length: 1
State disturbance vector length: 1
Observation innovation vector length: 1
Sample size supported by model: Unlimited

State variables: x1, x2,...
State disturbances: u1, u2,...
Observation series: y1, y2,...
Observation innovations: e1, e2,...

State equation:
x1(t) = (0.45)x1(t-1) + (0.89)u1(t)

Observation equation:
y1(t) = x1(t) + (0.39)e1(t)

Initial state distribution:

Initial state means
 x1
  0

Initial state covariance matrix
     x1
 x1  10

State types
     x1
 Stationary
```

`EstMdl` is an `ssm` model. The results of the estimation appear in the Command Window, contain the fitted state-space equations, and contain a table of parameter estimates, their standard errors, t statistics, and p-values.

You can use or display, for example the fitted state-transition matrix using dot notation.

```
EstMdl.A
```

```
ans =

    0.4543
```

Pass `EstMdl` to `forecast` to forecast observations, or to `simulate` to conduct a Monte Carlo study.

See Also

estimate | forecast | simulate | ssm

Related Examples

- "Create State-Space Model Containing ARMA State" on page 8-24
- "Implicitly Create Time-Invariant State-Space Model" on page 8-22
- "Estimate Time-Varying State-Space Model" on page 8-45
- "Estimate State-Space Model Containing Regression Component" on page 8-55

Estimate Time-Varying State-Space Model

This example shows how to:

1. Generate data from a known model.
2. Create a time-varying, state-space model containing unknown parameters corresponding to the data generating process.
3. Fit the state-space model to the data.

Suppose that an AR(2) and an MA(1) model comprise a latent process. There are 50 periods, and the MA(1) process drops out of the model for the final 25 periods. Subsequently, the state equation for the first 25 periods is

$$x_{1,t} = 0.7x_{1,t-1} - 0.2x_{1,t-2} + u_{1,t}$$
$$x_{2,t} = u_{2,t} + 0.6u_{2,t-1},$$

For the last 25 periods, the state equation is

$$x_{1,t} = 0.7x_{1,t-1} - 0.2x_{1,t-2} + u_{1,t},$$

where $u_{1,t}$ and $u_{2,t}$ are Gaussian with mean 0 and standard deviation 1.

Generate a random series of 50 observations from $x_{1,t}$ and $x_{2,t}$, assuming that the series starts at 1.5 and 1, respectively.

```
T = 50;
ARMdl = arima('AR',{0.7,-0.2},'Constant',0,'Variance',1);
MAMdl = arima('MA',0.6,'Constant',0,'Variance',1);
x0 = [1.5 1; 1.5 1];
rng(1);
x = [simulate(ARMdl,T,'Y0',x0(:,1)),...
    [simulate(MAMdl,T/2,'Y0',x0(:,2));nan(T/2,1)]];
```

The last 25 values for the simulated MA(1) data are missing.

Suppose further that the latent processes are measured using

$$y_t = 2\left(x_{1,t} + x_{2,t}\right) + \varepsilon_t,$$

for the first 25 periods, and

$$y_t = 2x_{1,t} + \varepsilon_t$$

for the last 25 periods. ε_t is Gaussian with mean 0 and standard deviation 1.

Generate observations using the random latent state process (`x`) and the observation equation.

```
y = 2*nansum(x')'+randn(T,1);
```

Together, the latent process and observation equations compose a state-space model. Supposing that the coefficients are unknown parameters, the state-space model is

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \\ x_{3,t} \\ x_{4,t} \end{bmatrix} = \begin{bmatrix} \phi_1 & \phi_2 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \theta_1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \\ x_{3,t-1} \\ x_{4,t-1} \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} u_{1,t} \\ u_{2,t} \end{bmatrix} \text{ for } t = 1, \ldots, 25,$$

$$y_t = a(x_{1,t} + x_{3,t}) + \varepsilon_t$$

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix} = \begin{bmatrix} \phi_1 & \phi_2 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \\ x_{3,t-1} \\ x_{4,t-1} \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} u_{1,t} \text{ for } t = 26,$$

$$y_t = bx_{1,t} + \varepsilon_t$$

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix} = \begin{bmatrix} \phi_1 & \phi_2 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} u_{1,t} \text{ for } t = 27, \ldots, 50$$

$$y_t = bx_{1,t} + \varepsilon_t$$

Write a function that specifies how the parameters in `params` map to the state-space model matrices, the initial state values, and the type of state.

```
% Copyright 2015 The MathWorks, Inc.

function [A,B,C,D,Mean0,Cov0,StateType] = AR2MAParamMap(params,T)
%AR2MAParamMap Time-variant state-space model parameter mapping function
%
% This function maps the vector params to the state-space matrices (A, B,
% C, and D), the initial state value and the initial state variance (Mean0
% and Cov0), and the type of state (StateType). From periods 1 to T/2, the
% state model is an AR(2) and an MA(1) model, and the observation model is
% the sum of the two states. From periods T/2 + 1 to T, the state model is
% just the AR(2) model.
    A1 = {[params(1) params(2) 0 0; 1 0 0 0; 0 0 0 params(3); 0 0 0 0]};
    B1 = {[1 0; 0 0; 0 1; 0 1]};
```

```
    C1 = {params(4)*[1 0 1 0]};
    Mean0 = ones(4,1);
    Cov0 = 10*eye(4);
    StateType = [0 0 0 0];
    A2 = {[params(1) params(2) 0 0; 1 0 0 0]};
    B2 = {[1; 0]};
    A3 = {[params(1) params(2); 1 0]};
    B3 = {[1; 0]};
    C3 = {params(5)*[1 0]};
    A = [repmat(A1,T/2,1);A2;repmat(A3,(T-2)/2,1)];
    B = [repmat(B1,T/2,1);B2;repmat(B3,(T-2)/2,1)];
    C = [repmat(C1,T/2,1);repmat(C3,T/2,1)];
    D = 1;
end
```

Save this code in a file named `AR2MAParamMap` and put it in your MATLAB® path.

Create the state-space model by passing the function `AR2MAParamMap` as a function handle to `ssm`.

```
Mdl = ssm(@(params)AR2MAParamMap(params,T));
```

`ssm` implicitly defines the state-space model. Usually, you cannot verify implicitly defined state-space models.

Pass the observed responses (`y`) to `estimate` to estimate the parameters. Specify positive initial values for the unknown parameters.

```
params0 = 0.1*ones(5,1);
EstMdl = estimate(Mdl,y,params0)
```

```
Method: Maximum likelihood (fminunc)
Sample size: 50
Logarithmic  likelihood:     -114.957
Akaike   info criterion:      239.913
Bayesian info criterion:      249.473
      |     Coeff       Std Err   t Stat     Prob
-------------------------------------------------
 c(1) |  0.47870       0.26634    1.79733  0.07229
 c(2) |  0.00809       0.27179    0.02975  0.97626
 c(3) |  0.55735       0.80958    0.68844  0.49118
 c(4) |  1.62679       0.41622    3.90848  0.00009
 c(5) |  1.90021       0.49563    3.83391  0.00013
```

```
      |
      |   Final State   Std Dev    t Stat    Prob
 x(1) | -0.81229       0.46815   -1.73511  0.08272
 x(2) | -0.31449       0.45918   -0.68490  0.49341

EstMdl =


State-space model type: <a href="matlab: doc ssm">ssm</a>

State vector length: Time-varying
Observation vector length: 1
State disturbance vector length: Time-varying
Observation innovation vector length: 1
Sample size supported by model: 50

State variables: x1, x2,...
State disturbances: u1, u2,...
Observation series: y1, y2,...
Observation innovations: e1, e2,...

State equations of period 1, 2, 3,..., 25:
x1(t) = (0.48)x1(t-1) + (8.09e-03)x2(t-1) + u1(t)
x2(t) = x1(t-1)
x3(t) = (0.56)x4(t-1) + u2(t)
x4(t) = u2(t)

State equations of period 26:
x1(t) = (0.48)x1(t-1) + (8.09e-03)x2(t-1) + u1(t)
x2(t) = x1(t-1)

State equations of period 27, 28, 29,..., 50:
x1(t) = (0.48)x1(t-1) + (8.09e-03)x2(t-1) + u1(t)
x2(t) = x1(t-1)


Observation equation of period 1, 2, 3,..., 25:
y1(t) = (1.63)x1(t) + (1.63)x3(t) + e1(t)

Observation equation of period 26, 27, 28,..., 50:
y1(t) = (1.90)x1(t) + e1(t)


Initial state distribution:
```

```
Initial state means
 x1  x2  x3  x4
  1   1   1   1

Initial state covariance matrix
     x1  x2  x3  x4
 x1  10  0   0   0
 x2  0   10  0   0
 x3  0   0   10  0
 x4  0   0   0   10

State types
     x1          x2          x3          x4
 Stationary  Stationary  Stationary  Stationary
```

The estimated parameters are within 1 standard error of their true values, but the standard errors are quite high. Likelihood surfaces of state-space models might contain local maxima. Therefore, it is good practice to try several initial parameter values, or consider using `refine`.

See Also

estimate | forecast | refine | simulate | ssm

Related Examples

- "Create State-Space Model Containing ARMA State" on page 8-24
- "Implicitly Create Time-Invariant State-Space Model" on page 8-22
- "Estimate Time-Invariant State-Space Model" on page 8-41
- "Estimate State-Space Model Containing Regression Component" on page 8-55

More About

- "What Are State-Space Models?" on page 8-3
- "What Is the Kalman Filter?" on page 8-8

Estimate Time-Varying Diffuse State-Space Model

This example shows how to:

1. Generate data from a known model.
2. Create a time-varying, diffuse state-space model containing unknown parameters corresponding to the data generating process. The diffuse specification indicates complete ignorance of the true state values.
3. Fit the diffuse state-space model to the data.

Suppose that an AR(2) and an MA(1) model comprise a latent process. There are 50 periods, and the MA(1) process drops out of the model for the final 25 periods. Consequently, the state equation for the first 25 periods is

$$x_{1,t} = 0.7x_{1,t-1} - 0.2x_{1,t-2} + u_{1,t}$$
$$x_{2,t} = u_{2,t} + 0.6u_{2,t-1}.$$

For the last 25 periods, the state equation is

$$x_{1,t} = 0.7x_{1,t-1} - 0.2x_{1,t-2} + u_{1,t},$$

where $u_{1,t}$ and $u_{2,t}$ are Gaussian with mean 0 and standard deviation 1.

Generate a random series of 50 observations from $x_{1,t}$ and $x_{2,t}$, assuming that the series starts at 1.5 and 1, respectively.

```
T = 50;
ARMdl = arima('AR',{0.7,-0.2},'Constant',0,'Variance',1);
MAMdl = arima('MA',0.6,'Constant',0,'Variance',1);
x0 = [1.5 1; 1.5 1];
rng(1);
x = [simulate(ARMdl,T,'Y0',x0(:,1)),...
    [simulate(MAMdl,T/2,'Y0',x0(:,2));nan(T/2,1)]];
```

The last 25 values for the simulated MA(1) data are missing.

The latent processes are measured using

$$y_t = 2\left(x_{1,t} + x_{2,t}\right) + \varepsilon_t$$

for the first 25 periods, and

$$y_t = 2x_{1,t} + \varepsilon_t$$

for the last 25 periods. ε_t is Gaussian with mean 0 and standard deviation 1.

Generate observations using the random, latent state process (`x`) and the observation equation.

```
y = 2*nansum(x')' + randn(T,1);
```

Together, the latent process and observation equations make up a state-space model. If the coefficients are unknown parameters, the state-space model is

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \\ x_{3,t} \\ x_{4,t} \end{bmatrix} = \begin{bmatrix} \phi_1 & \phi_2 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \theta_1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \\ x_{3,t-1} \\ x_{4,t-1} \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} u_{1,t} \\ u_{2,t} \end{bmatrix} \text{ for } t = 1, \ldots, 25,$$

$$y_t = a(x_{1,t} + x_{3,t}) + \varepsilon_t$$

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix} = \begin{bmatrix} \phi_1 & \phi_2 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \\ x_{3,t-1} \\ x_{4,t-1} \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} u_{1,t} \text{ for } t = 26,$$

$$y_t = bx_{1,t} + \varepsilon_t$$

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix} = \begin{bmatrix} \phi_1 & \phi_2 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} u_{1,t} \text{ for } t = 27, \ldots, 50.$$

$$y_t = bx_{1,t} + \varepsilon_t$$

Write a function that specifies how the parameters in `params` map to the state-space model matrices, the initial state values, and the type of state.

```
% Copyright 2015 The MathWorks, Inc.

function [A,B,C,D,Mean0,Cov0,StateType] = diffuseAR2MAParamMap(params,T)
%diffuseAR2MAParamMap Time-variant diffuse state-space model parameter
%mapping function
%
% This function maps the vector params to the state-space matrices (A, B,
% C, and D) and the type of state (StateType). From periods 1 to T/2, the
% state model is an AR(2) and an MA(1) model, and the observation model is
% the sum of the two states. From periods T/2 + 1 to T, the state model is
% just the AR(2) model.  The AR(2) model is diffuse.
```

```
    A1 = {[params(1) params(2) 0 0; 1 0 0 0; 0 0 0 params(3); 0 0 0 0]};
    B1 = {[1 0; 0 0; 0 1; 0 1]};
    C1 = {params(4)*[1 0 1 0]};
    Mean0 = [];
    Cov0 = [];
    StateType = [2 2 0 0];
    A2 = {[params(1) params(2) 0 0; 1 0 0 0]};
    B2 = {[1; 0]};
    A3 = {[params(1) params(2); 1 0]};
    B3 = {[1; 0]};
    C3 = {params(5)*[1 0]};
    A = [repmat(A1,T/2,1);A2;repmat(A3,(T-2)/2,1)];
    B = [repmat(B1,T/2,1);B2;repmat(B3,(T-2)/2,1)];
    C = [repmat(C1,T/2,1);repmat(C3,T/2,1)];
    D = 1;
end
```

Save this code in a file named `diffuseAR2MAParamMap` on your MATLAB® path.

Create the state-space model by passing the function `diffuseAR2MAParamMap` as a function handle to `dssm`.

```
Mdl = dssm(@(params)diffuseAR2MAParamMap(params,T));
```

`dssm` implicitly defines the diffuse state-space model. Usually, you cannot verify diffuse state-space models that are implicitly created.

To estimate the parameters, pass the observed responses (`y`) to `estimate` . Specify positive initial values for the unknown parameters.

```
params0 = 0.1*ones(5,1);
EstMdl = estimate(Mdl,y,params0)
```

```
Method: Maximum likelihood (fminunc)
Effective Sample size:             48
Logarithmic  likelihood:     -110.313
Akaike   info criterion:      230.626
Bayesian info criterion:      240.186
      |     Coeff       Std Err   t Stat     Prob
-------------------------------------------------
 c(1) |  0.44041       0.27687    1.59069  0.11168
 c(2) |  0.03949       0.29585    0.13349  0.89380
 c(3) |  0.78364       1.49223    0.52515  0.59948
```

```
 c(4) |  1.64260       0.66737    2.46133  0.01384
 c(5) |  1.90409       0.49374    3.85648  0.00012
      |
      |   Final State   Std Dev    t Stat    Prob
 x(1) | -0.81932       0.46706   -1.75420  0.07940
 x(2) | -0.29909       0.45939   -0.65107  0.51500

EstMdl =


State-space model type: <a href="matlab: doc dssm">dssm</a>

State vector length: Time-varying
Observation vector length: 1
State disturbance vector length: Time-varying
Observation innovation vector length: 1
Sample size supported by model: 50

State variables: x1, x2,...
State disturbances: u1, u2,...
Observation series: y1, y2,...
Observation innovations: e1, e2,...

State equations of period 1, 2, 3,..., 25:
x1(t) = (0.44)x1(t-1) + (0.04)x2(t-1) + u1(t)
x2(t) = x1(t-1)
x3(t) = (0.78)x4(t-1) + u2(t)
x4(t) = u2(t)

State equations of period 26:
x1(t) = (0.44)x1(t-1) + (0.04)x2(t-1) + u1(t)
x2(t) = x1(t-1)

State equations of period 27, 28, 29,..., 50:
x1(t) = (0.44)x1(t-1) + (0.04)x2(t-1) + u1(t)
x2(t) = x1(t-1)


Observation equation of period 1, 2, 3,..., 25:
y1(t) = (1.64)x1(t) + (1.64)x3(t) + e1(t)

Observation equation of period 26, 27, 28,..., 50:
y1(t) = (1.90)x1(t) + e1(t)
```

```
Initial state distribution:

Initial state means
 x1  x2  x3  x4
  0   0   0   0

Initial state covariance matrix
     x1   x2   x3    x4
 x1  Inf  0     0     0
 x2  0    Inf   0     0
 x3  0    0    1.61   1
 x4  0    0     1     1

State types
    x1       x2        x3          x4
 Diffuse  Diffuse  Stationary  Stationary
```

The estimated parameters are within one standard error of their true values, but the standard errors are quite high. Likelihood surfaces of state-space models might contain local maxima. Therefore, try several initial parameter values, or consider using `refine`.

See Also

dssm | estimate | refine

Related Examples

- "Implicitly Create Time-Varying Diffuse State-Space Model" on page 8-35
- "Implicitly Create Diffuse State-Space Model Containing Regression Component" on page 8-30
- "Filter Time-Varying Diffuse State-Space Model" on page 8-68

More About

- "What Are State-Space Models?" on page 8-3
- "What Is the Kalman Filter?" on page 8-8

Estimate State-Space Model Containing Regression Component

This example shows how to fit a state-space model that has an observation-equation regression component.

Suppose that the linear relationship between the change in the unemployment rate and the nominal gross national product (nGNP) growth rate is of interest. Suppose further that the first difference of the unemployment rate is an ARMA(1,1) series. Symbolically, and in state-space form, the model is

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix} = \begin{bmatrix} \phi & \theta \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix} u_{1,t}$$
$$y_t - \beta Z_t = x_{1,t} + \sigma \varepsilon_t,$$

where:

- $x_{1,t}$ is the change in the unemployment rate at time t.
- $x_{2,t}$ is a dummy state for the MA(1) effect.
- $y_{1,t}$ is the observed change in the unemployment rate being deflated by the growth rate of nGNP (Z_t).
- $u_{1,t}$ is the Gaussian series of state disturbances having mean 0 and standard deviation 1.
- ε_t is the Gaussian series of observation innovations having mean 0 and standard deviation σ.

Load the Nelson-Plosser data set, which contains the unemployment rate and nGNP series, among other things.

```
load Data_NelsonPlosser
```

Preprocess the data by taking the natural logarithm of the nGNP series, and the first difference of each. Also, remove the starting `NaN` values from each series.

```
isNaN = any(ismissing(DataTable),2);       % Flag periods containing NaNs
gnpn = DataTable.GNPN(~isNaN);
u = DataTable.UR(~isNaN);
T = size(gnpn,1);                          % Sample size
Z = [ones(T-1,1) diff(log(gnpn))];
```

```
y = diff(u);
```

This example proceeds using series without `NaN` values. However, using the Kalman filter framework, the software can accommodate series containing missing values.

Specify the coefficient matrices. Use `NaN` values to indicate unknown parameters.

```
A = [NaN NaN; 0 0];
B = [1; 1];
C = [1 0];
D = NaN;
```

Specify the state-space model using `ssm`. Since $x_{1,t}$ is an ARMA(1,1) process, and $x_{2,t}$ is white noise, specify that they are stationary processes.

```
StateType = [0; 0];
Mdl = ssm(A,B,C,D,'StateType',StateType);
```

Estimate the model parameters. Specify the regression component and its initial value for optimization using the `'Predictors'` and `'Beta0'` name-value pair arguments, respectively. Restrict the estimate of σ to all positive, real numbers, but allow all other parameters to be unbounded.

```
params0 = [0.3 0.2 0.1]; % Chosen arbitrarily
EstMdl = estimate(Mdl,y,params0,'Predictors',Z,'Beta0',[0.1 0.1],...
    'lb',[-Inf,-Inf,0,-Inf,-Inf]);
```

```
Method: Maximum likelihood (fmincon)
Sample size: 61
Logarithmic  likelihood:     -99.7245
Akaike   info criterion:      209.449
Bayesian info criterion:      220.003
           |      Coeff       Std Err    t Stat     Prob
-------------------------------------------------------------
 c(1)      |  -0.34098       0.29608    -1.15164  0.24948
 c(2)      |   1.05003       0.41377     2.53771  0.01116
 c(3)      |   0.48592       0.36790     1.32079  0.18657
 y <- z(1) |   1.36121       0.22338     6.09358   0
 y <- z(2) | -24.46711       1.60018   -15.29024   0
           |
           |    Final State   Std Dev     t Stat    Prob
 x(1)      |   1.01264       0.44690     2.26592  0.02346
 x(2)      |   0.77718       0.58917     1.31912  0.18713
```

A table of estimates and statistics output to the Command Window. `EstMdl` is an `ssm` model, and you can access its properties using dot notation.

See Also

estimate | forecast | simulate | ssm

Related Examples

- "Create State-Space Model Containing ARMA State" on page 8-24
- "Implicitly Create Time-Invariant State-Space Model" on page 8-22
- "Estimate Time-Invariant State-Space Model" on page 8-41
- "Estimate Time-Varying State-Space Model" on page 8-45

More About

- "What Are State-Space Models?" on page 8-3
- "What Is the Kalman Filter?" on page 8-8

Filter States of State-Space Model

This example shows how to filter states of a known, time-invariant, state-space model.

Suppose that a latent process is an AR(1). Subsequently, the state equation is

$$x_t = 0.5x_{t-1} + u_t,$$

where u_t is Gaussian with mean 0 and standard deviation 1.

Generate a random series of 100 observations from x_t, assuming that the series starts at 1.5.

```
T = 100;
ARMdl = arima('AR',0.5,'Constant',0,'Variance',1);
x0 = 1.5;
rng(1); % For reproducibility
x = simulate(ARMdl,T,'Y0',x0);
```

Suppose further that the latent process is subject to additive measurement error. Subsequently, the observation equation is

$$y_t = x_t + \varepsilon_t,$$

where ε_t is Gaussian with mean 0 and standard deviation 0.75. Together, the latent process and observation equations compose a state-space model.

Use the random latent state process (`x`) and the observation equation to generate observations.

```
y = x + 0.75*randn(T,1);
```

Specify the four coefficient matrices.

```
A = 0.5;
B = 1;
C = 1;
D = 0.75;
```

Specify the state-space model using the coefficient matrices.

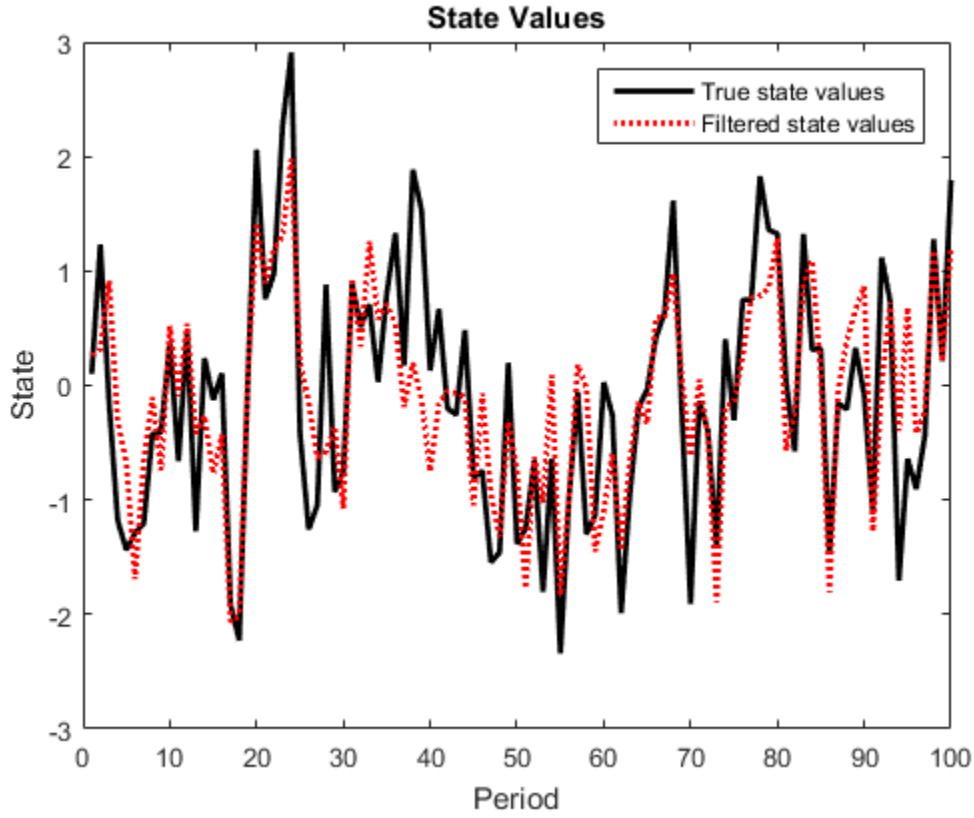
```
Mdl = ssm(A,B,C,D)
```

```
Mdl =


State-space model type: <a href="matlab: doc ssm">ssm</a>

State vector length: 1
Observation vector length: 1
State disturbance vector length: 1
Observation innovation vector length: 1
Sample size supported by model: Unlimited

State variables: x1, x2,...
State disturbances: u1, u2,...
Observation series: y1, y2,...
Observation innovations: e1, e2,...

State equation:
x1(t) = (0.50)x1(t-1) + u1(t)

Observation equation:
y1(t) = x1(t) + (0.75)e1(t)

Initial state distribution:

Initial state means
 x1
  0

Initial state covariance matrix
     x1
 x1  1.33

State types
     x1
 Stationary
```

`Mdl` is an `ssm` model. Verify that the model is correctly specified using the display in the Command Window. The software infers that the state process is stationary. Subsequently, the software sets the initial state mean and covariance to the mean and variance of the stationary distribution of an AR(1) model.

Estimate the filtered states for periods 1 through 100. Plot the true state values and the estimated, filtered states.

```
filteredX = filter(Mdl,y);

figure
plot(1:T,x,'-k',1:T,filteredX,':r','LineWidth',2)
title({'State Values'})
xlabel('Period')
ylabel('State')
legend({'True state values','Filtered state values'})
```



The true values and filter estimates are approximately the same.

See Also

estimate | filter | smooth | ssm

Related Examples

- "Create State-Space Model Containing ARMA State" on page 8-24
- "Smooth States of State-Space Model" on page 8-80

Filter Time-Varying State-Space Model

This example shows how to generate data from a known model, fit a state-space model to the data, and then filter the states.

Suppose that a latent process comprises an AR(2) and an MA(1) model. There are 50 periods, and the MA(1) process drops out of the model for the final 25 periods. Subsequently, the state equation for the first 25 periods is

$$x_{1,t} = 0.7x_{1,t-1} - 0.2x_{1,t-2} + u_{1,t}$$
$$x_{2,t} = u_{2,t} + 0.6u_{2,t-1},$$

and for the last 25 periods, it is

$$x_{1,t} = 0.7x_{1,t-1} - 0.2x_{1,t-2} + u_{1,t},$$

where $u_{1,t}$ and $u_{2,t}$ are Gaussian with mean 0 and standard deviation 1.

Assuming that the series starts at 1.5 and 1, respectively, generate a random series of 50 observations from $x_{1,t}$ and $x_{2,t}$.

```
T = 50;
ARMdl = arima('AR',{0.7,-0.2},'Constant',0,'Variance',1);
MAMdl = arima('MA',0.6,'Constant',0,'Variance',1);
x0 = [1.5 1; 1.5 1];
rng(1);
x = [simulate(ARMdl,T,'Y0',x0(:,1)),...
    [simulate(MAMdl,T/2,'Y0',x0(:,2));nan(T/2,1)]];
```

The last 25 values for the simulated MA(1) data are `NaN` values.

Suppose further that the latent processes are measured using

$$y_t = 2\left(x_{1,t} + x_{2,t}\right) + \varepsilon_t,$$

for the first 25 periods, and

$$y_t = 2x_{1,t} + \varepsilon_t$$

for the last 25 periods, where ε_t is Gaussian with mean 0 and standard deviation 1.

Use the random latent state process (`x`) and the observation equation to generate observations.

```
y = 2*nansum(x')'+randn(T,1);
```

Together, the latent process and observation equations compose a state-space model. Supposing that the coefficients are unknown parameters, the state-space model is

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \\ x_{3,t} \\ x_{4,t} \end{bmatrix} = \begin{bmatrix} \phi_1 & \phi_2 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \theta_1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \\ x_{3,t-1} \\ x_{4,t-1} \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} u_{1,t} \\ u_{2,t} \end{bmatrix}$$
$$y_t = a(x_{1,t} + x_{3,t}) + \varepsilon_t$$

for the first 25 periods,

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix} = \begin{bmatrix} \phi_1 & \phi_2 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \\ x_{3,t-1} \\ x_{4,t-1} \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} u_{1,t}$$
$$y_t = bx_{1,t} + \varepsilon_t$$

for period 26, and

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix} = \begin{bmatrix} \phi_1 & \phi_2 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} u_{1,t}$$
$$y_t = bx_{1,t} + \varepsilon_t$$

for the last 24 periods.

Write a function that specifies how the parameters in `params` map to the state-space model matrices, the initial state values, and the type of state.

```
% Copyright 2015 The MathWorks, Inc.

function [A,B,C,D,Mean0,Cov0,StateType] = AR2MAParamMap(params,T)
%AR2MAParamMap Time-variant state-space model parameter mapping function
%
```

```
% This function maps the vector params to the state-space matrices (A, B,
% C, and D), the initial state value and the initial state variance (Mean0
% and Cov0), and the type of state (StateType). From periods 1 to T/2, the
% state model is an AR(2) and an MA(1) model, and the observation model is
% the sum of the two states. From periods T/2 + 1 to T, the state model is
% just the AR(2) model.
    A1 = {[params(1) params(2) 0 0; 1 0 0 0; 0 0 0 params(3); 0 0 0 0]};
    B1 = {[1 0; 0 0; 0 1; 0 1]};
    C1 = {params(4)*[1 0 1 0]};
    Mean0 = ones(4,1);
    Cov0 = 10*eye(4);
    StateType = [0 0 0 0];
    A2 = {[params(1) params(2) 0 0; 1 0 0 0]};
    B2 = {[1; 0]};
    A3 = {[params(1) params(2); 1 0]};
    B3 = {[1; 0]};
    C3 = {params(5)*[1 0]};
    A = [repmat(A1,T/2,1);A2;repmat(A3,(T-2)/2,1)];
    B = [repmat(B1,T/2,1);B2;repmat(B3,(T-2)/2,1)];
    C = [repmat(C1,T/2,1);repmat(C3,T/2,1)];
    D = 1;
end
```

Save this code as a file named `AR2MAParamMap` on your MATLAB® path.

Create the state-space model by passing the function `AR2MAParamMap` as a function handle to `ssm`.

```
Mdl = ssm(@(params)AR2MAParamMap(params,T));
```

`ssm` implicitly creates the state-space model. Usually, you cannot verify an implicitly defined state-space model.

Pass the observed responses (`y`) to `estimate` to estimate the parameters. Specify an arbitrary set of positive initial values for the unknown parameters.

```
params0 = 0.1*ones(5,1);
EstMdl = estimate(Mdl,y,params0);
```

```
Method: Maximum likelihood (fminunc)
Sample size: 50
Logarithmic  likelihood:     -114.957
Akaike   info criterion:      239.913
```

```
Bayesian info criterion:      249.473
      |     Coeff       Std Err   t Stat     Prob
-------------------------------------------------
 c(1) |  0.47870       0.26634    1.79733  0.07229
 c(2) |  0.00809       0.27179    0.02975  0.97626
 c(3) |  0.55735       0.80958    0.68844  0.49118
 c(4) |  1.62679       0.41622    3.90848  0.00009
 c(5) |  1.90021       0.49563    3.83391  0.00013
      |
      |   Final State   Std Dev    t Stat    Prob
 x(1) | -0.81229       0.46815   -1.73511  0.08272
 x(2) | -0.31449       0.45918   -0.68490  0.49341
```

`EstMdl` is an `ssm` model containing the estimated coefficients. Likelihood surfaces of state-space models might contain local maxima. Therefore, it is good practice to try several initial parameter values, or consider using `refine`.

Filter the states and obtain state forecasts by passing `EstMdl` and the observed responses to `filter`.

```
[~,~,Output]= filter(EstMdl,y);
```

`Output` is a `T`-by-1 structure array containing the filtered states and state forecasts, among other things.

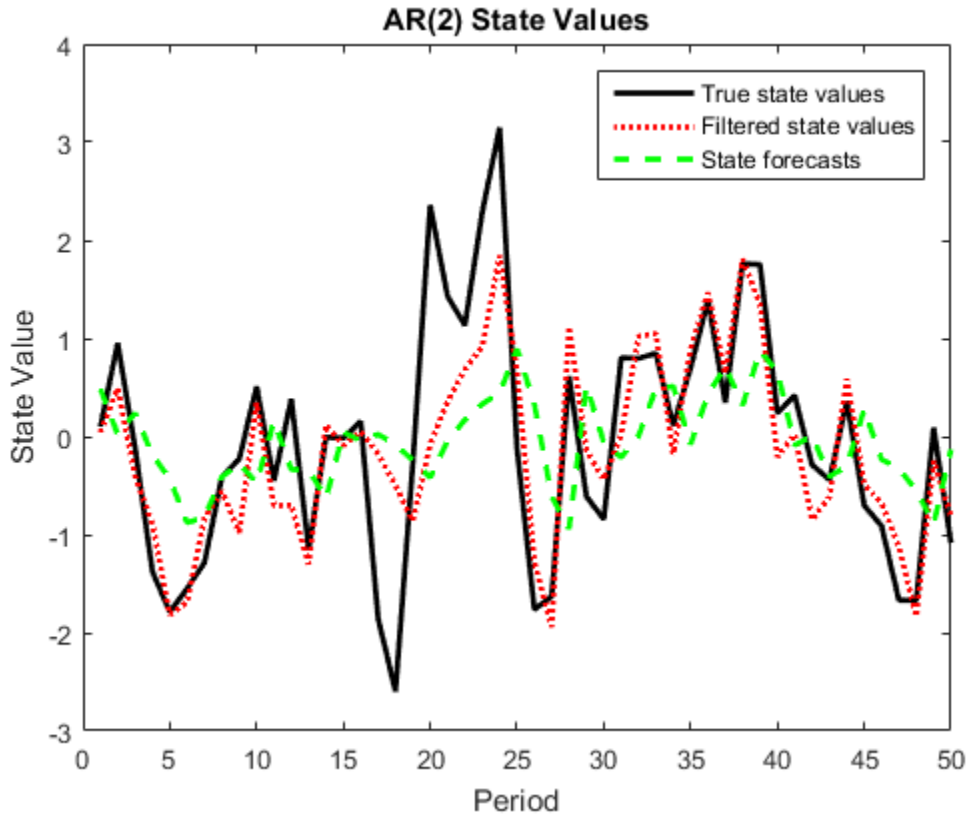
Extract the filtered and forecasted states from the cell arrays. Recall that the two, different states are in positions 1 and 3. The states in positions 2 and 4 help specify the processes of interest.

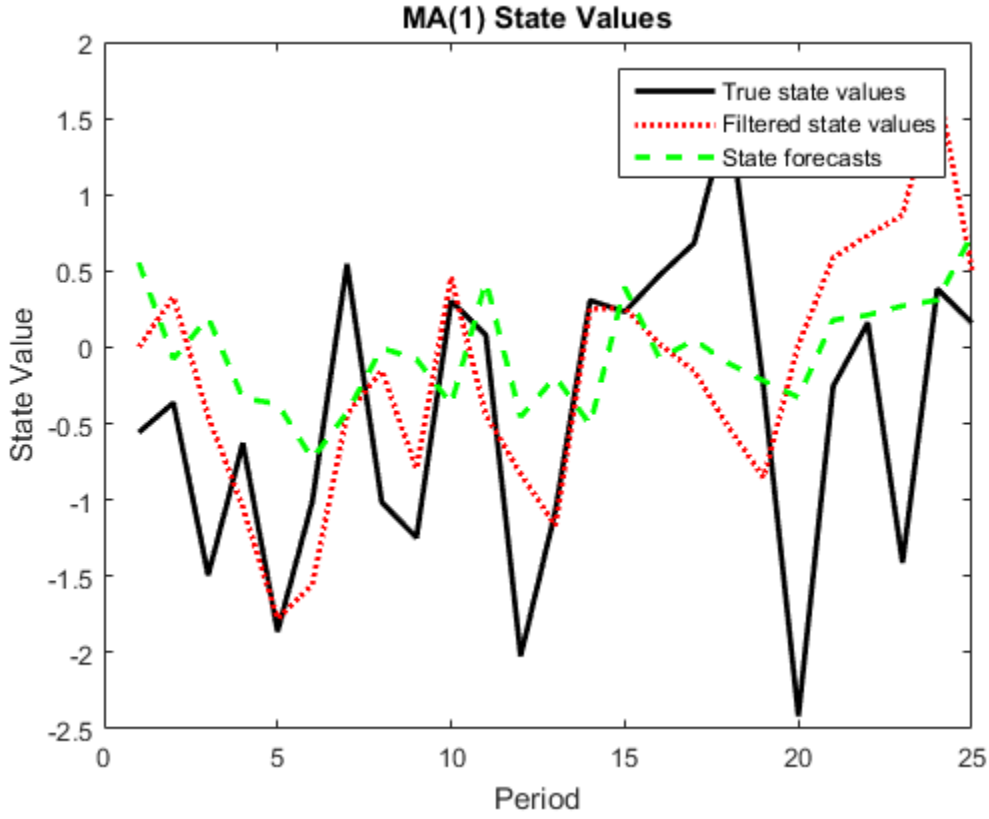
```
stateIndx = [1 3]; % State indices of interest
FilteredStates = NaN(T,numel(stateIndx));
ForecastedStates = NaN(T,numel(stateIndx));

for t = 1:T
    maxInd = size(Output(t).FilteredStates,1);
    mask = stateIndx <= maxInd;
    FilteredStates(t,mask) = Output(t).FilteredStates(stateIndx(mask),1);
  ForecastedStates(t,mask) = Output(t).ForecastedStates(stateIndx(mask),1);
end
```

Plot the true state values, the filtered states, and the state forecasts together for each model.

```
figure
```

```
plot(1:T,x(:,1),'-k',1:T,FilteredStates(:,1),':r',...
    1:T,ForecastedStates(:,1),'--g','LineWidth',2);
title('AR(2) State Values')
xlabel('Period')
ylabel('State Value')
legend({'True state values','Filtered state values','State forecasts'});

figure
plot(1:T,x(:,2),'-k',1:T,FilteredStates(:,2),':r',...
    1:T,ForecastedStates(:,2),'--g','LineWidth',2);
title('MA(1) State Values')
xlabel('Period')
ylabel('State Value')
legend({'True state values','Filtered state values','State forecasts'});
```





See Also

estimate | filter | forecast | refine | smooth | ssm

Related Examples

- "Implicitly Create Time-Varying State-Space Model" on page 8-32
- "Estimate Time-Varying State-Space Model" on page 8-45
- "Smooth Time-Varying State-Space Model" on page 8-84

Filter Time-Varying Diffuse State-Space Model

This example shows how to generate data from a known model, fit a diffuse state-space model to the data, and then filter the states.

Suppose that a latent process comprises an AR(2) and an MA(1) model. There are 50 periods, and the MA(1) process drops out of the model for the final 25 periods. Consequently, the state equation for the first 25 periods is

$$x_{1,t} = 0.7x_{1,t-1} - 0.2x_{1,t-2} + u_{1,t}$$
$$x_{2,t} = u_{2,t} + 0.6u_{2,t-1},$$

and for the last 25 periods, it is

$$x_{1,t} = 0.7x_{1,t-1} - 0.2x_{1,t-2} + u_{1,t},$$

where $u_{1,t}$ and $u_{2,t}$ are Gaussian with mean 0 and standard deviation 1.

Assuming that the series starts at 1.5 and 1, respectively, generate a random series of 50 observations from $x_{1,t}$ and $x_{2,t}$.

```
T = 50;
ARMdl = arima('AR',{0.7,-0.2},'Constant',0,'Variance',1);
MAMdl = arima('MA',0.6,'Constant',0,'Variance',1);
x0 = [1.5 1; 1.5 1];
rng(1);
x = [simulate(ARMdl,T,'Y0',x0(:,1)),...
    [simulate(MAMdl,T/2,'Y0',x0(:,2));nan(T/2,1)]];
```

The last 25 values for the simulated MA(1) data are `NaN` values.

The latent processes are measured using

$$y_t = 2\left(x_{1,t} + x_{2,t}\right) + \varepsilon_t$$

for the first 25 periods, and

$$y_t = 2x_{1,t} + \varepsilon_t$$

for the last 25 periods, where ε_t is Gaussian with mean 0 and standard deviation 1.

Use the random latent state process (`x`) and the observation equation to generate observations.

```
y = 2*nansum(x')'+randn(T,1);
```

Together, the latent process and observation equations make up a state-space model. The coefficients are unknown parameters, the state-space model is

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \\ x_{3,t} \\ x_{4,t} \end{bmatrix} = \begin{bmatrix} \phi_1 & \phi_2 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \theta_1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \\ x_{3,t-1} \\ x_{4,t-1} \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} u_{1,t} \\ u_{2,t} \end{bmatrix}$$
$$y_t = a(x_{1,t} + x_{3,t}) + \varepsilon_t$$

for the first 25 periods,

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix} = \begin{bmatrix} \phi_1 & \phi_2 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \\ x_{3,t-1} \\ x_{4,t-1} \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} u_{1,t}$$
$$y_t = bx_{1,t} + \varepsilon_t$$

for period 26, and

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix} = \begin{bmatrix} \phi_1 & \phi_2 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} u_{1,t}$$
$$y_t = bx_{1,t} + \varepsilon_t$$

for the last 24 periods.

Write a function that specifies how the parameters in `params` map to the state-space model matrices, the initial state values, and the type of state.

```
% Copyright 2015 The MathWorks, Inc.

function [A,B,C,D,Mean0,Cov0,StateType] = diffuseAR2MAParamMap(params,T)
%diffuseAR2MAParamMap Time-variant diffuse state-space model parameter
%mapping function
%
```

```
% This function maps the vector params to the state-space matrices (A, B,
% C, and D) and the type of state (StateType). From periods 1 to T/2, the
% state model is an AR(2) and an MA(1) model, and the observation model is
% the sum of the two states. From periods T/2 + 1 to T, the state model is
% just the AR(2) model.  The AR(2) model is diffuse.
    A1 = {[params(1) params(2) 0 0; 1 0 0 0; 0 0 0 params(3); 0 0 0 0]};
    B1 = {[1 0; 0 0; 0 1; 0 1]};
    C1 = {params(4)*[1 0 1 0]};
    Mean0 = [];
    Cov0 = [];
    StateType = [2 2 0 0];
    A2 = {[params(1) params(2) 0 0; 1 0 0 0]};
    B2 = {[1; 0]};
    A3 = {[params(1) params(2); 1 0]};
    B3 = {[1; 0]};
    C3 = {params(5)*[1 0]};
    A = [repmat(A1,T/2,1);A2;repmat(A3,(T-2)/2,1)];
    B = [repmat(B1,T/2,1);B2;repmat(B3,(T-2)/2,1)];
    C = [repmat(C1,T/2,1);repmat(C3,T/2,1)];
    D = 1;
end
```

Save this code as a file named `diffuseAR2MAParamMap` on your MATLAB® path.

Create the diffuse state-space model by passing the function `diffuseAR2MAParamMap` as a function handle to `dssm`.

```
Mdl = dssm(@(params)diffuseAR2MAParamMap(params,T));
```

`dssm` implicitly creates the diffuse state-space model. Usually, you cannot verify diffuse state-space models that are implicitly created.

To estimate the parameters, pass the observed responses (`y`) to `estimate`. Specify an arbitrary set of positive initial values for the unknown parameters.

```
params0 = 0.1*ones(5,1);
EstMdl = estimate(Mdl,y,params0);
```

```
Method: Maximum likelihood (fminunc)
Effective Sample size:             48
Logarithmic  likelihood:     -110.313
Akaike   info criterion:      230.626
Bayesian info criterion:      240.186
```

```
      |     Coeff       Std Err   t Stat     Prob
-------------------------------------------------
 c(1) |  0.44041       0.27687    1.59069  0.11168
 c(2) |  0.03949       0.29585    0.13349  0.89380
 c(3) |  0.78364       1.49223    0.52515  0.59948
 c(4) |  1.64260       0.66737    2.46133  0.01384
 c(5) |  1.90409       0.49374    3.85648  0.00012
      |
      |   Final State   Std Dev    t Stat    Prob
 x(1) | -0.81932       0.46706   -1.75420  0.07940
 x(2) | -0.29909       0.45939   -0.65107  0.51500
```

`EstMdl` is a `dssm` model containing the estimated coefficients. Likelihood surfaces of state-space models might contain local maxima. Therefore, try several initial parameter values, or consider using `refine`.

Filter the states and obtain state forecasts by passing `EstMdl` and the observed responses to `filter`.

```
[~,~,Output]= filter(EstMdl,y);
```

`Output` is a `T`-by-1 structure array that contains the filtered states and state forecasts.

Convert `Output` to a table.

```
OutputTbl = struct2table(Output);
OutputTbl(1:10,1:5) % Display first ten rows of first five variables
```

```
ans =

    LogLikelihood    FilteredStates    FilteredStatesCov    ForecastedStates    Forecas
    _____________    ______________    _________________    ________________    _______

    []               []                []                   []                  []
    []               []                []                   []                  []
    [-2.3218]        [4x1 double]      [4x4 double]         [4x1 double]        [4x4 do
    [-2.4464]        [4x1 double]      [4x4 double]         [4x1 double]        [4x4 do
    [-3.8758]        [4x1 double]      [4x4 double]         [4x1 double]        [4x4 do
    [-2.5212]        [4x1 double]      [4x4 double]         [4x1 double]        [4x4 do
    [-1.9016]        [4x1 double]      [4x4 double]         [4x1 double]        [4x4 do
    [-1.9284]        [4x1 double]      [4x4 double]         [4x1 double]        [4x4 do
    [-2.4110]        [4x1 double]      [4x4 double]         [4x1 double]        [4x4 do
    [-2.6502]        [4x1 double]      [4x4 double]         [4x1 double]        [4x4 do
```

The first two rows of the table contain empty cells or zeros, which correspond to the observations required to initialize the diffuse Kalman filter. That is, `SwitchTime` is 2.

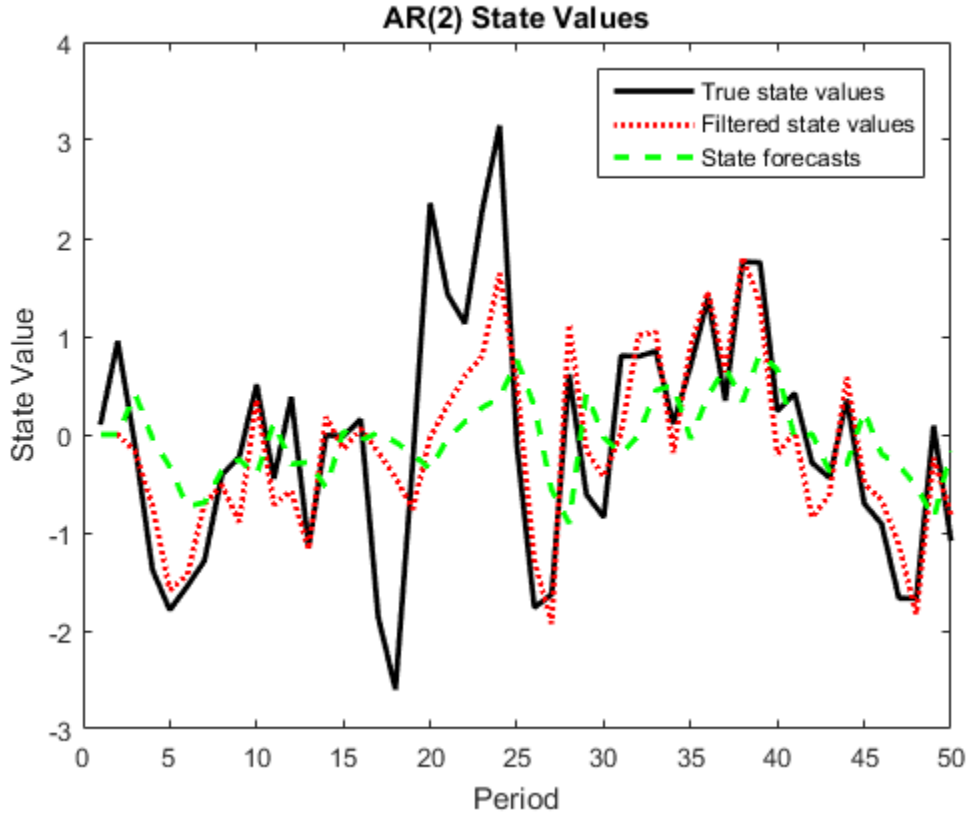
```
SwitchTime = 2;
```

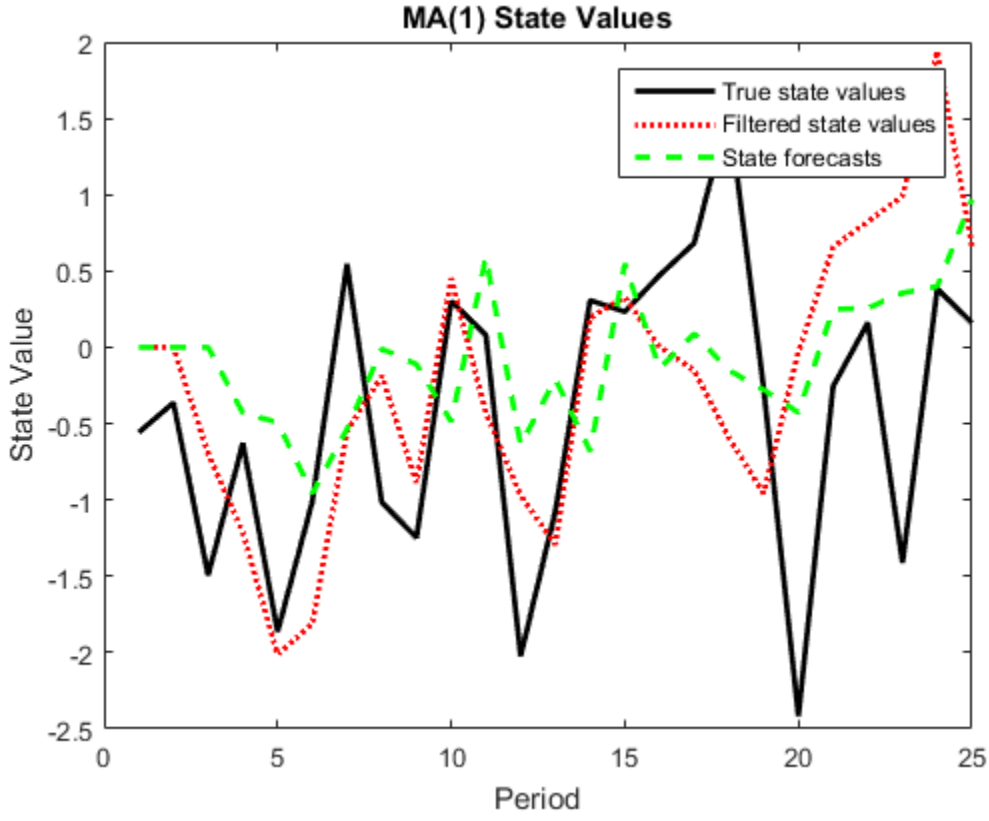
Extract the filtered and forecasted states from the table. Recall that the two different states are in positions 1 and 3. The states in positions 2 and 4 help to specify the processes of interest.

```
stateIdx = [1 3];                          % State indices of interest

FilteredStates = NaN(T,numel(stateIdx));
ForecastedStates = NaN(T,numel(stateIdx));

for t = (SwitchTime + 1):T
    maxInd = size(Output(t).FilteredStates,1);
    mask = stateIdx <= maxInd;
    FilteredStates(t,mask) = Output(t).FilteredStates(stateIdx(mask),1);
  ForecastedStates(t,mask) = Output(t).ForecastedStates(stateIdx(mask),1);
end

FilteredStates(1:SwitchTime,:) = 0;
ForecastedStates(1:SwitchTime,:) = 0;
```

Plot the true state values, the filtered states, and the state forecasts together for each model.

```
figure
plot(1:T,x(:,1),'-k',1:T,FilteredStates(:,1),':r',...
    1:T,ForecastedStates(:,1),'--g','LineWidth',2);
title('AR(2) State Values')
xlabel('Period')
ylabel('State Value')
legend({'True state values','Filtered state values','State forecasts'});

figure
plot(1:T,x(:,2),'-k',1:T,FilteredStates(:,2),':r',...
    1:T,ForecastedStates(:,2),'--g','LineWidth',2);
title('MA(1) State Values')
xlabel('Period')
ylabel('State Value')
legend({'True state values','Filtered state values','State forecasts'});
```


AR(2) State Values
True state values
Filtered state values
State forecasts
State Value
Period
4
3
2
1
0
-1
-2
-3
0
5
10
15
20
25
30
35
40
45
50



See Also

dssm | esitmate | smooth

Related Examples

- "Implicitly Create Time-Varying Diffuse State-Space Model" on page 8-35
- "Implicitly Create Diffuse State-Space Model Containing Regression Component" on page 8-30
- "Estimate Time-Varying Diffuse State-Space Model" on page 8-50
- "Smooth Time-Varying Diffuse State-Space Model" on page 8-91

More About

- "What Are State-Space Models?" on page 8-3
- "What Is the Kalman Filter?" on page 8-8

Filter States of State-Space Model Containing Regression Component

This example shows how to filter states of a time-invariant, state-space model that contains a regression component.

Suppose that the linear relationship between the change in the unemployment rate and the nominal gross national product (nGNP) growth rate is of interest. Suppose further that the first difference of the unemployment rate is an ARMA(1,1) series. Symbolically, and in state-space form, the model is

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix} = \begin{bmatrix} \phi & \theta \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix} u_{1,t}$$
$$y_t - \beta Z_t = x_{1,t} + \sigma \varepsilon_t,$$

where:

- $x_{1,t}$ is the change in the unemployment rate at time t.
- $x_{2,t}$ is a dummy state for the MA(1) effect.
- $y_{1,t}$ is the observed change in the unemployment rate being deflated by the growth rate of nGNP (Z_t).
- $u_{1,t}$ is the Gaussian series of state disturbances having mean 0 and standard deviation 1.
- ε_t is the Gaussian series of observation innovations having mean 0 and standard deviation σ.

Load the Nelson-Plosser data set, which contains the unemployment rate and nGNP series, among other things.

```
load Data_NelsonPlosser
```

Preprocess the data by taking the natural logarithm of the nGNP series, and the first difference of each series. Also, remove the starting `NaN` values from each series.

```
isNaN = any(ismissing(DataTable),2);       % Flag periods containing NaNs
gnpn = DataTable.GNPN(~isNaN);
u = DataTable.UR(~isNaN);
T = size(gnpn,1);                          % Sample size
Z = [ones(T-1,1) diff(log(gnpn))];
y = diff(u);
```

Though this example removes missing values, the software can accommodate series containing missing values in the Kalman filter framework.

Specify the coefficient matrices.

```
A = [NaN NaN; 0 0];
B = [1; 1];
C = [1 0];
D = NaN;
```

Specify the state-space model using `ssm`.

```
Mdl = ssm(A,B,C,D);
```

Estimate the model parameters. Specify the regression component and its initial value for optimization using the `'Predictors'` and `'Beta0'` name-value pair arguments, respectively. Restrict the estimate of σ to all positive, real numbers.

```
params0 = [0.3 0.2 0.2];
[EstMdl,estParams] = estimate(Mdl,y,params0,'Predictors',Z,...
    'Beta0',[0.1 0.2],'lb',[-Inf,-Inf,0,-Inf,-Inf]);
```

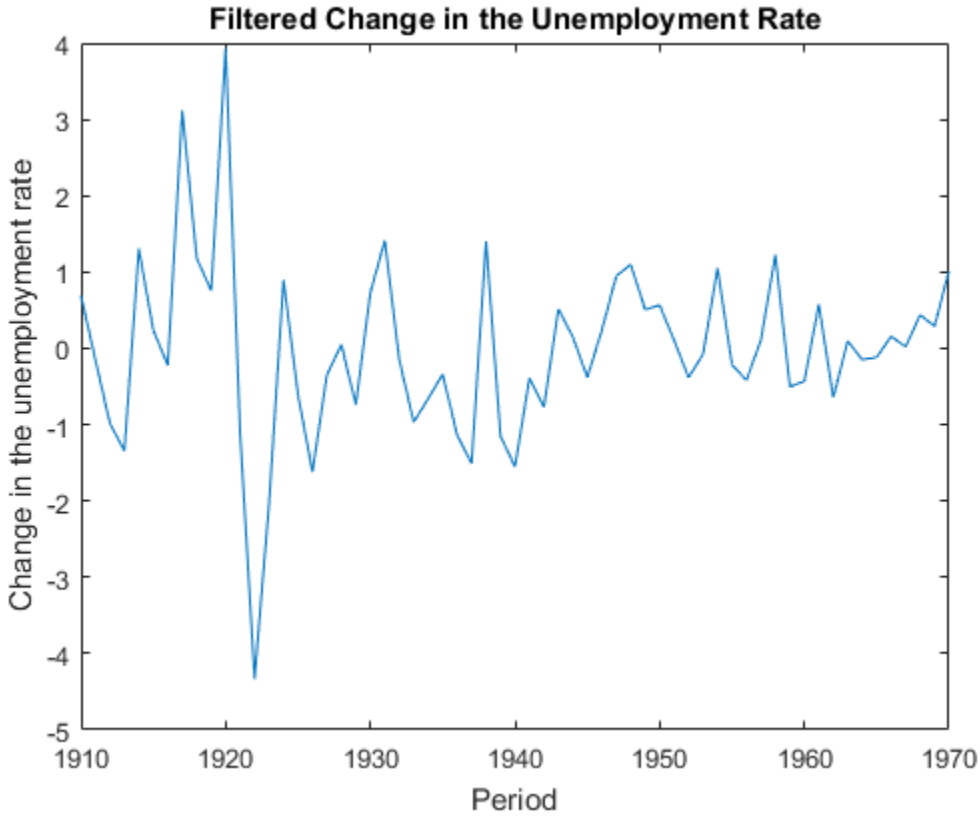
```
Method: Maximum likelihood (fmincon)
Sample size: 61
Logarithmic  likelihood:     -99.7245
Akaike   info criterion:      209.449
Bayesian info criterion:      220.003
           |      Coeff       Std Err    t Stat     Prob
-------------------------------------------------------------
 c(1)      |  -0.34098       0.29608    -1.15164  0.24948
 c(2)      |   1.05003       0.41377     2.53771  0.01116
 c(3)      |   0.48592       0.36790     1.32080  0.18657
 y <- z(1) |   1.36121       0.22338     6.09358   0
 y <- z(2) | -24.46711       1.60018   -15.29024   0
           |
           |    Final State   Std Dev     t Stat    Prob
 x(1)      |   1.01264       0.44690     2.26592  0.02346
 x(2)      |   0.77718       0.58917     1.31912  0.18713
```

`EstMdl` is an `ssm` model, and you can access its properties using dot notation.

Filter the estimated state-space model. `EstMdl` does not store the data or the regression coefficients, so you must pass in them in using the name-value pair arguments `'Predictors'` and `'Beta'`, respectively. Plot the estimated, filtered states. Recall that

the first state is the change in the unemployment rate, and the second state helps build the first.

```
filteredX = filter(EstMdl,y,'Predictors',Z,'Beta',estParams(end-1:end));

figure
plot(dates(end-(T-1)+1:end),filteredX(:,1));
xlabel('Period')
ylabel('Change in the unemployment rate')
title('Filtered Change in the Unemployment Rate')
```



See Also

estimate | filter | smooth | ssm

Related Examples

- "Create State-Space Model Containing ARMA State" on page 8-24
- "Estimate State-Space Model Containing Regression Component" on page 8-55
- "Smooth States of State-Space Model Containing Regression Component" on page 8-99

Smooth States of State-Space Model

This example shows how to smooth the states of a known, time-invariant, state-space model.

Suppose that a latent process is an AR(1) model. Subsequently, the state equation is

$$x_t = 0.5x_{t-1} + u_t,$$

where u_t is Gaussian with mean 0 and standard deviation 0.5.

Generate a random series of 100 observations from x_t, assuming that the series starts at 1.5.

```
T = 100;
ARMdl = arima('AR',0.5,'Constant',0,'Variance',0.5^2);
x0 = 1.5;
rng(1); % For reproducibility
x = simulate(ARMdl,T,'Y0',x0);
```

Suppose further that the latent process is subject to additive measurement error. Subsequently, the observation equation is

$$y_t = x_t + \varepsilon_t,$$

where ε_t is Gaussian with mean 0 and standard deviation 0.05. Together, the latent process and observation equations compose a state-space model.

Use the random latent state process (`x`) and the observation equation to generate observations.

```
y = x + 0.05*randn(T,1);
```

Specify the four coefficient matrices.

```
A = 0.5;
B = 1;
C = 1;
D = 0.75;
```

Specify the state-space model using the coefficient matrices.

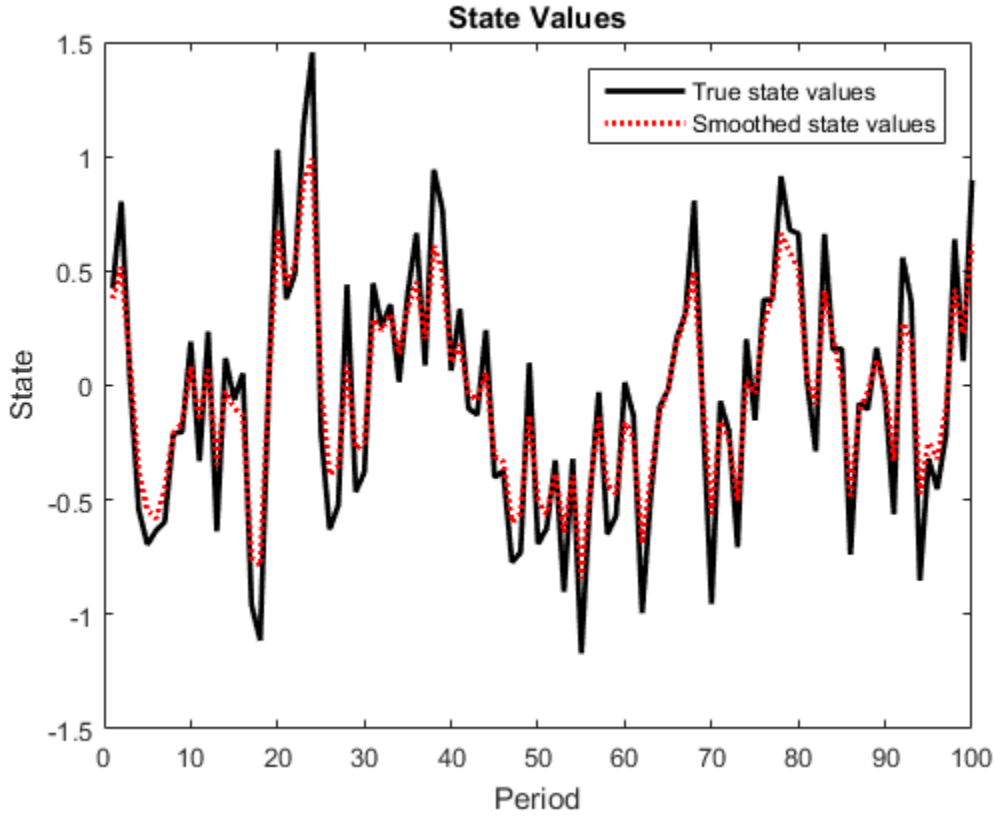
```
Mdl = ssm(A,B,C,D)
```

```
Mdl =


State-space model type: <a href="matlab: doc ssm">ssm</a>

State vector length: 1
Observation vector length: 1
State disturbance vector length: 1
Observation innovation vector length: 1
Sample size supported by model: Unlimited

State variables: x1, x2,...
State disturbances: u1, u2,...
Observation series: y1, y2,...
Observation innovations: e1, e2,...

State equation:
x1(t) = (0.50)x1(t-1) + u1(t)

Observation equation:
y1(t) = x1(t) + (0.75)e1(t)

Initial state distribution:

Initial state means
 x1
  0

Initial state covariance matrix
     x1
 x1  1.33

State types
     x1
 Stationary
```

`Mdl` is an `ssm` model. Verify that the model is correctly specified using the display in the Command Window. The software infers that the state process is stationary. Subsequently, the software sets the initial state mean and covariance to the mean and variance of the stationary distribution of an AR(1) model.

Smooth the states for periods 1 through 100. Plot the true state values and the smoothed states.

```
SmoothedX = smooth(Mdl,y);

figure
plot(1:T,x,'-k',1:T,SmoothedX,':r','LineWidth',2)
title({'State Values'})
xlabel('Period')
ylabel('State')
legend({'True state values','Smoothed state values'})
```



See Also

estimate | filter | smooth | ssm

Related Examples

- "Create State-Space Model Containing ARMA State" on page 8-24
- "Filter States of State-Space Model" on page 8-58

Smooth Time-Varying State-Space Model

This example shows how to generate data from a known model, fit a state-space model to the data, and then smooth the states.

Suppose that a latent process comprises an AR(2) and an MA(1) model. There are 50 periods, and the MA(1) process drops out of the model for the final 25 periods. Subsequently, the state equation for the first 25 periods is

$$\begin{aligned} x_{1,t} &= 0.7x_{1,t-1} - 0.2x_{1,t-2} + u_{1,t} \\ x_{2,t} &= u_{2,t} + 0.6u_{2,t-1}, \end{aligned}$$

and for the last 25 periods, it is

$$x_{1,t} = 0.7x_{1,t-1} - 0.2x_{1,t-2} + u_{1,t},$$

where $u_{1,t}$ and $u_{2,t}$ are Gaussian with mean 0 and standard deviation 1.

Assuming that the series starts at 1.5 and 1, respectively, generate a random series of 50 observations from $x_{1,t}$ and $x_{2,t}$.

```
T = 50;
ARMdl = arima('AR',{0.7,-0.2},'Constant',0,'Variance',1);
MAMdl = arima('MA',0.6,'Constant',0,'Variance',1);
x0 = [1.5 1; 1.5 1];
rng(1);
x = [simulate(ARMdl,T,'Y0',x0(:,1)),...
    [simulate(MAMdl,T/2,'Y0',x0(:,2));nan(T/2,1)]];
```

The last 25 values for the simulated MA(1) data are `NaN` values.

Suppose further that the latent processes are measured using

$$y_t = 2\left(x_{1,t} + x_{2,t}\right) + \varepsilon_t,$$

for the first 25 periods, and

$$y_t = 2x_{1,t} + \varepsilon_t$$

for the last 25 periods, where ε_t is Gaussian with mean 0 and standard deviation 1.

Use the random latent state process (`x`) and the observation equation to generate observations.

```
y = 2*nansum(x')'+randn(T,1);
```

Together, the latent process and observation equations compose a state-space model. Supposing that the coefficients are unknown parameters, the state-space model is

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \\ x_{3,t} \\ x_{4,t} \end{bmatrix} = \begin{bmatrix} \phi_1 & \phi_2 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \theta_1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \\ x_{3,t-1} \\ x_{4,t-1} \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} u_{1,t} \\ u_{2,t} \end{bmatrix}$$
$$y_t = a(x_{1,t} + x_{3,t}) + \varepsilon_t$$

for the first 25 periods,

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix} = \begin{bmatrix} \phi_1 & \phi_2 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \\ x_{3,t-1} \\ x_{4,t-1} \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} u_{1,t}$$
$$y_t = bx_{1,t} + \varepsilon_t$$

for period 26, and

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix} = \begin{bmatrix} \phi_1 & \phi_2 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} u_{1,t}$$
$$y_t = bx_{1,t} + \varepsilon_t$$

for the last 24 periods.

Write a function that specifies how the parameters in `params` map to the state-space model matrices, the initial state values, and the type of state.

```
% Copyright 2015 The MathWorks, Inc.

function [A,B,C,D,Mean0,Cov0,StateType] = AR2MAParamMap(params,T)
%AR2MAParamMap Time-variant state-space model parameter mapping function
%
% This function maps the vector params to the state-space matrices (A, B,
% C, and D), the initial state value and the initial state variance (Mean0
% and Cov0), and the type of state (StateType). From periods 1 to T/2, the
% state model is an AR(2) and an MA(1) model, and the observation model is
% the sum of the two states. From periods T/2 + 1 to T, the state model is
% just the AR(2) model.
    A1 = {[params(1) params(2) 0 0; 1 0 0 0; 0 0 0 params(3); 0 0 0 0]};
    B1 = {[1 0; 0 0; 0 1; 0 1]};
    C1 = {params(4)*[1 0 1 0]};
    Mean0 = ones(4,1);
    Cov0 = 10*eye(4);
    StateType = [0 0 0 0];
    A2 = {[params(1) params(2) 0 0; 1 0 0 0]};
    B2 = {[1; 0]};
    A3 = {[params(1) params(2); 1 0]};
    B3 = {[1; 0]};
    C3 = {params(5)*[1 0]};
    A = [repmat(A1,T/2,1);A2;repmat(A3,(T-2)/2,1)];
    B = [repmat(B1,T/2,1);B2;repmat(B3,(T-2)/2,1)];
    C = [repmat(C1,T/2,1);repmat(C3,T/2,1)];
    D = 1;
end
```

Save this code as a file named `AR2MAParamMap` on your MATLAB® path.

Create the state-space model by passing the function `AR2MAParamMap` as a function handle to `ssm`.

```
Mdl = ssm(@(params)AR2MAParamMap(params,T));
```

`ssm` implicitly creates the state-space model. Usually, you cannot verify an implicitly defined state-space model.

Pass the observed responses (`y`) to `estimate` to estimate the parameters. Specify an arbitrary set of positive initial values for the unknown parameters.

```
params0 = 0.1*ones(5,1);
EstMdl = estimate(Mdl,y,params0);
```

```
Method: Maximum likelihood (fminunc)
Sample size: 50
Logarithmic  likelihood:     -114.957
Akaike   info criterion:      239.913
Bayesian info criterion:      249.473
      |     Coeff       Std Err   t Stat     Prob
-------------------------------------------------------
 c(1) |  0.47870       0.26634    1.79733  0.07229
 c(2) |  0.00809       0.27179    0.02975  0.97626
 c(3) |  0.55735       0.80958    0.68844  0.49118
 c(4) |  1.62679       0.41622    3.90848  0.00009
 c(5) |  1.90021       0.49563    3.83391  0.00013
      |
      |   Final State   Std Dev    t Stat    Prob
 x(1) | -0.81229       0.46815   -1.73511  0.08272
 x(2) | -0.31449       0.45918   -0.68490  0.49341
```

`EstMdl` is an `ssm` model containing the estimated coefficients. Likelihood surfaces of state-space models might contain local maxima. Therefore, it is good practice to try several initial parameter values, or consider using `refine`.

Smooth the states and estimate the variance-covariance matrices of the smoothed states by passing `EstMdl` and the observed responses to `smooth`.

```
[~,~,Output]= smooth(EstMdl,y);
```

`Output` is a `T`-by-1 structure array containing the smoothed states and their variance-covariance matrices, among other things.

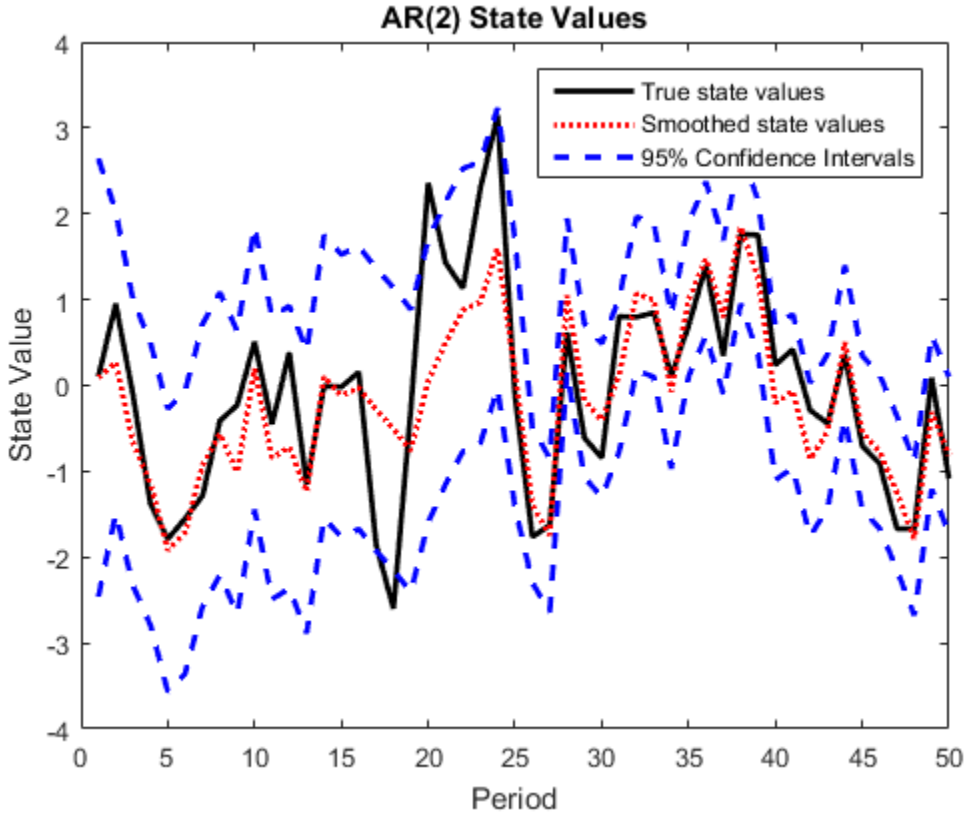
Extract the smoothed states and their variances from the cell arrays. Recall that the two, different states are in positions 1 and 3. The states in positions 2 and 4 help specify the processes of interest.

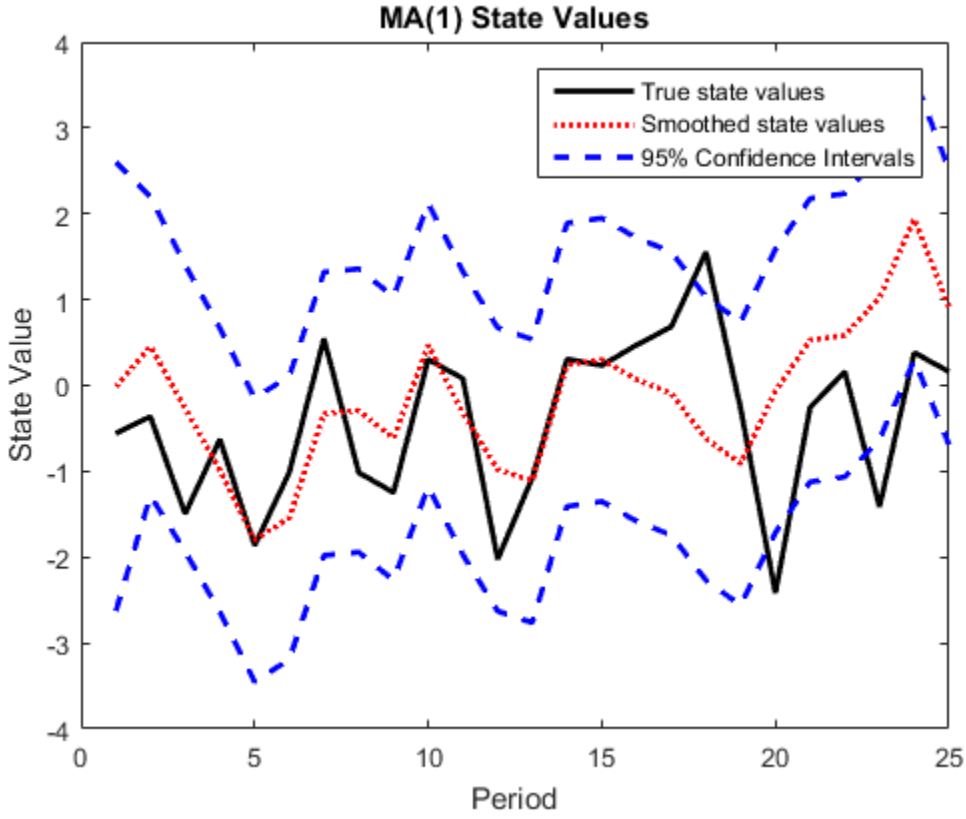
```
stateIndx = [1 3]; % State Indices of interest
SmoothedStates = NaN(T,numel(stateIndx));
SmoothedStatesCov = NaN(T,numel(stateIndx));

for t = 1:T
    maxInd1 = size(Output(t).SmoothedStates,1);
  maxInd2 = size(Output(t).SmoothedStatesCov,1);
    mask1 = stateIndx <= maxInd1;
  mask2 = stateIndx <= maxInd2;
    SmoothedStates(t,mask1) = ...
```

```
        Output(t).SmoothedStates(stateIndx(mask1),1);
  SmoothedStatesCov(t,mask2) = ...
        diag(Output(t).SmoothedStatesCov(stateIndx(mask2),...
        stateIndx(mask2)));
end
```

Plot the true state values, the smoothed state values, and their individual 95% Wald-type confidence intervals for each model.

```
AR2SSCIlb = SmoothedStates(:,1) - 1.95*sqrt(SmoothedStatesCov(:,1));
AR2SSCIub = SmoothedStates(:,1) + 1.95*sqrt(SmoothedStatesCov(:,1));
AR2SSIntervals = [AR2SSCIlb AR2SSCIub];

MA1SSCIlb = SmoothedStates(:,2) - 1.95*sqrt(SmoothedStatesCov(:,2));
MA1SSCIub = SmoothedStates(:,2) + 1.95*sqrt(SmoothedStatesCov(:,2));
MA1SSIntervals = [MA1SSCIlb MA1SSCIub];

figure
plot(1:T,x(:,1),'-k',1:T,SmoothedStates(:,1),':r',...
  1:T,AR2SSIntervals,'--b','LineWidth',2);
title('AR(2) State Values')
xlabel('Period')
ylabel('State Value')
legend({'True state values','Smoothed state values',...
    '95% Confidence Intervals'});

figure
plot(1:T,x(:,2),'-k',1:T,SmoothedStates(:,2),':r',...
  1:T,MA1SSIntervals,'--b','LineWidth',2);
title('MA(1) State Values')
xlabel('Period')
ylabel('State Value')
legend({'True state values','Smoothed state values',...
    '95% Confidence Intervals'});
```


AR(2) State Values
True state values
Smoothed state values
95% Confidence Intervals
State Value
Period
4
3
2
1
0
-1
-2
-3
-4
0
5
10
15
20
25
30
35
40
45
50



See Also

estimate | filter | refine | smooth | ssm

Related Examples

- "Implicitly Create Time-Varying State-Space Model" on page 8-32
- "Estimate Time-Varying State-Space Model" on page 8-45
- "Filter Time-Varying State-Space Model" on page 8-62

Smooth Time-Varying Diffuse State-Space Model

This example shows how to generate data from a known model, fit a diffuse state-space model to the data, and then smooth the states.

Suppose that a latent process comprises an AR(2) and an MA(1) model. There are 50 periods, and the MA(1) process drops out of the model for the final 25 periods. Consequently, the state equation for the first 25 periods is

$$x_{1,t} = 0.7x_{1,t-1} - 0.2x_{1,t-2} + u_{1,t}$$
$$x_{2,t} = u_{2,t} + 0.6u_{2,t-1},$$

and for the last 25 periods, it is

$$x_{1,t} = 0.7x_{1,t-1} - 0.2x_{1,t-2} + u_{1,t},$$

where $u_{1,t}$ and $u_{2,t}$ are Gaussian with mean 0 and standard deviation 1.

Assuming that the series starts at 1.5 and 1, respectively, generate a random series of 50 observations from $x_{1,t}$ and $x_{2,t}$.

```
T = 50;
ARMdl = arima('AR',{0.7,-0.2},'Constant',0,'Variance',1);
MAMdl = arima('MA',0.6,'Constant',0,'Variance',1);
x0 = [1.5 1; 1.5 1];
rng(1);
x = [simulate(ARMdl,T,'Y0',x0(:,1)),...
    [simulate(MAMdl,T/2,'Y0',x0(:,2));nan(T/2,1)]];
```

The last 25 values for the simulated MA(1) data are `NaN` values.

The latent processes are measured using

$$y_t = 2\left(x_{1,t} + x_{2,t}\right) + \varepsilon_t,$$

for the first 25 periods, and

$$y_t = 2x_{1,t} + \varepsilon_t$$

for the last 25 periods, where ε_t is Gaussian with mean 0 and standard deviation 1.

Use the random latent state process (`x`) and the observation equation to generate observations.

```
y = 2*nansum(x')'+randn(T,1);
```

Together, the latent process and observation equations make up a state-space model. If the coefficients are unknown parameters, the state-space model is

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \\ x_{3,t} \\ x_{4,t} \end{bmatrix} = \begin{bmatrix} \phi_1 & \phi_2 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \theta_1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \\ x_{3,t-1} \\ x_{4,t-1} \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} u_{1,t} \\ u_{2,t} \end{bmatrix}$$
$$y_t = a(x_{1,t} + x_{3,t}) + \varepsilon_t$$

for the first 25 periods,

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix} = \begin{bmatrix} \phi_1 & \phi_2 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \\ x_{3,t-1} \\ x_{4,t-1} \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} u_{1,t}$$
$$y_t = bx_{1,t} + \varepsilon_t$$

for period 26, and

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix} = \begin{bmatrix} \phi_1 & \phi_2 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} u_{1,t}$$
$$y_t = bx_{1,t} + \varepsilon_t$$

for the last 24 periods.

Write a function that specifies how the parameters in `params` map to the state-space model matrices, the initial state values, and the type of state.

```
% Copyright 2015 The MathWorks, Inc.

function [A,B,C,D,Mean0,Cov0,StateType] = diffuseAR2MAParamMap(params,T)
%diffuseAR2MAParamMap Time-variant diffuse state-space model parameter
%mapping function
%
```

```
% This function maps the vector params to the state-space matrices (A, B,
% C, and D) and the type of state (StateType). From periods 1 to T/2, the
% state model is an AR(2) and an MA(1) model, and the observation model is
% the sum of the two states. From periods T/2 + 1 to T, the state model is
% just the AR(2) model.  The AR(2) model is diffuse.
    A1 = {[params(1) params(2) 0 0; 1 0 0 0; 0 0 0 params(3); 0 0 0 0]};
    B1 = {[1 0; 0 0; 0 1; 0 1]};
    C1 = {params(4)*[1 0 1 0]};
    Mean0 = [];
    Cov0 = [];
    StateType = [2 2 0 0];
    A2 = {[params(1) params(2) 0 0; 1 0 0 0]};
    B2 = {[1; 0]};
    A3 = {[params(1) params(2); 1 0]};
    B3 = {[1; 0]};
    C3 = {params(5)*[1 0]};
    A = [repmat(A1,T/2,1);A2;repmat(A3,(T-2)/2,1)];
    B = [repmat(B1,T/2,1);B2;repmat(B3,(T-2)/2,1)];
    C = [repmat(C1,T/2,1);repmat(C3,T/2,1)];
    D = 1;
end
```

Save this code as a file named `diffuseAR2MAParamMap` on your MATLAB® path.

Create the diffuse state-space model by passing the function `diffuseAR2MAParamMap` as a function handle to `dssm`.

```
Mdl = dssm(@(params)diffuseAR2MAParamMap(params,T));
```

`dssm` implicitly creates the diffuse state-space model. Usually, you cannot verify diffuse state-space models that are implicitly created.

To estimate the parameters, pass the observed responses (`y`) to `estimate`. Specify an arbitrary set of positive initial values for the unknown parameters.

```
params0 = 0.1*ones(5,1);
EstMdl = estimate(Mdl,y,params0);
```

```
Method: Maximum likelihood (fminunc)
Effective Sample size:             48
Logarithmic  likelihood:     -110.313
Akaike   info criterion:      230.626
Bayesian info criterion:      240.186
```

```
      |     Coeff       Std Err   t Stat     Prob
-------------------------------------------------
 c(1) |  0.44041       0.27687    1.59069  0.11168
 c(2) |  0.03949       0.29585    0.13349  0.89380
 c(3) |  0.78364       1.49223    0.52515  0.59948
 c(4) |  1.64260       0.66737    2.46133  0.01384
 c(5) |  1.90409       0.49374    3.85648  0.00012
      |
      |   Final State   Std Dev    t Stat    Prob
 x(1) | -0.81932       0.46706   -1.75420  0.07940
 x(2) | -0.29909       0.45939   -0.65107  0.51500
```

`EstMdl` is a `dssm` model containing the estimated coefficients. Likelihood surfaces of state-space models might contain local maxima. Therefore, try several initial parameter values, or consider using `refine`.

Smooth the states and obtain the smoothed state covariance matrix per period by passing `EstMdl` and the observed responses to `smooth`.

```
[~,~,Output]= smooth(EstMdl,y);
```

`Output` is a `T`-by-1 structure array that contains the smoothed states.

Convert `Output` to a table.

```
OutputTbl = struct2table(Output);
OutputTbl(1:10,1:4) % Display first ten rows of first four variables
```

```
ans =

    LogLikelihood    SmoothedStates    SmoothedStatesCov    SmoothedStateDisturb
    _____________    ______________    _________________    ____________________

    []               []                []                   []
    []               []                []                   []
    [-2.3218]        [4x1 double]      [4x4 double]         [2x1 double]
    [-2.4464]        [4x1 double]      [4x4 double]         [2x1 double]
    [-3.8758]        [4x1 double]      [4x4 double]         [2x1 double]
    [-2.5212]        [4x1 double]      [4x4 double]         [2x1 double]
    [-1.9016]        [4x1 double]      [4x4 double]         [2x1 double]
    [-1.9284]        [4x1 double]      [4x4 double]         [2x1 double]
    [-2.4110]        [4x1 double]      [4x4 double]         [2x1 double]
    [-2.6502]        [4x1 double]      [4x4 double]         [2x1 double]
```

The first two rows of the table contain empty cells or zeros, which correspond to the observations required to initialize the diffuse Kalman filter. That is, `SwitchTime` is 2.

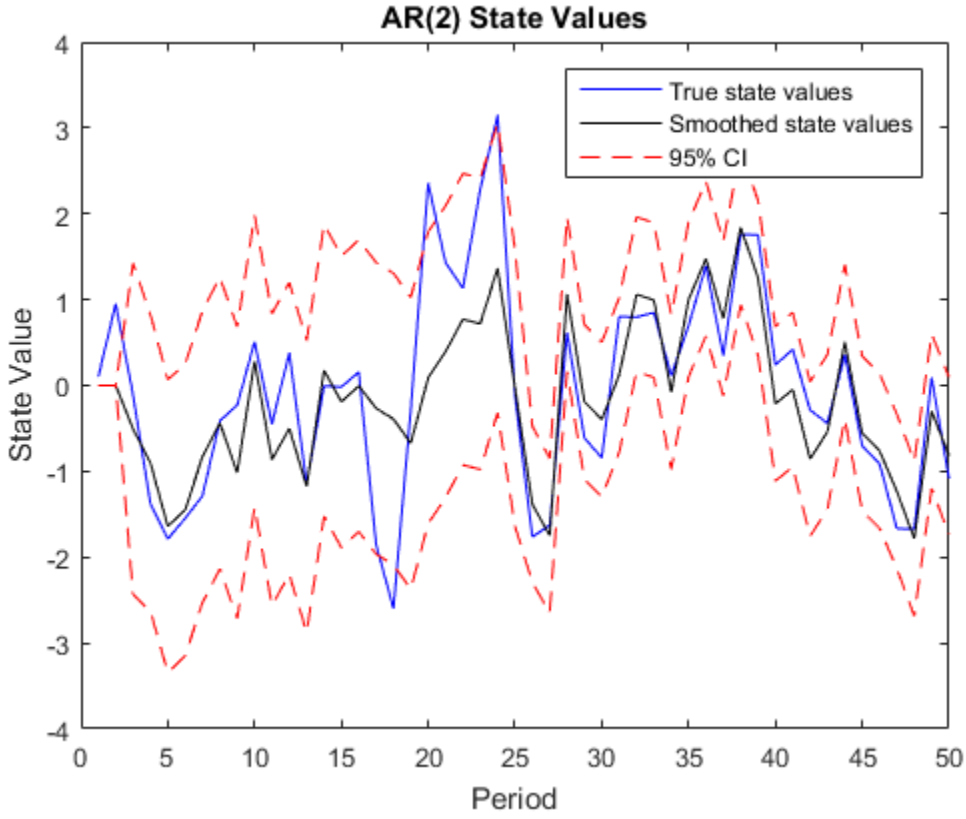
```
SwitchTime = 2;
```

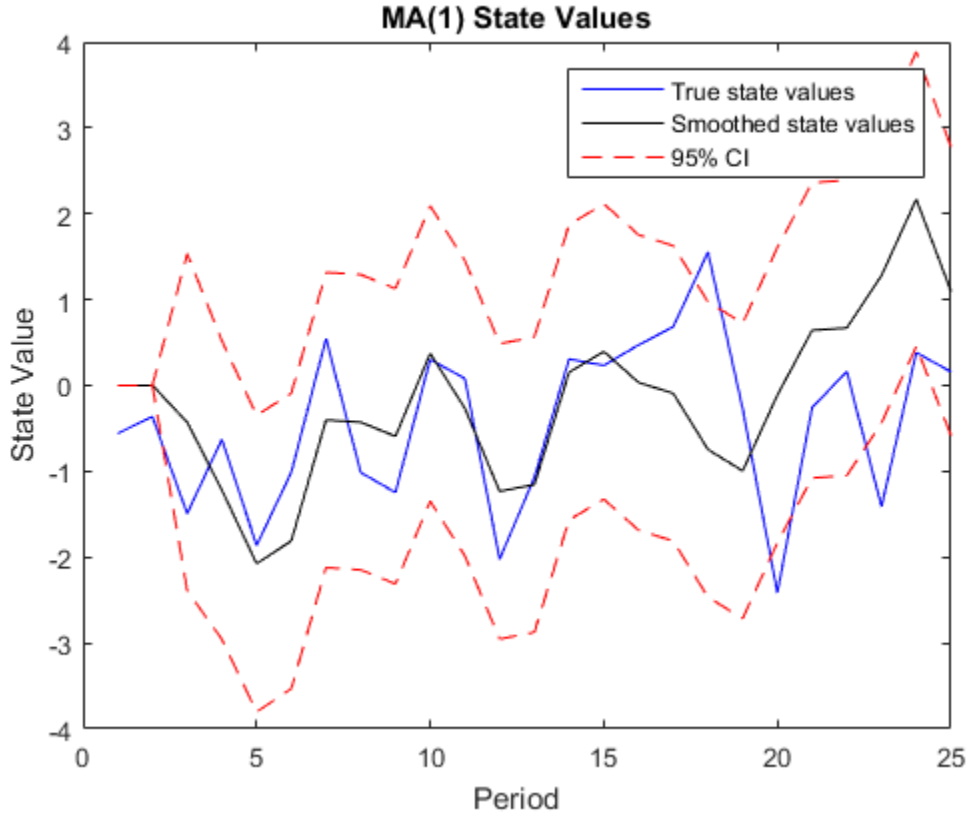
Extract the smoothed states from `Output`, and compute their 95% individual, Wald-type confidence intervals. Recall that the two different states are in positions 1 and 3. The states in positions 2 and 4 help to specify the processes of interest.

```
stateIdx = [1 3]; % State indices of interest

SmoothedStates = nan(T,numel(stateIdx));
CI = nan(T,2,numel(stateIdx));

for t = (SwitchTime + 1):T
    maxInd = size(Output(t).SmoothedStates,1);
    mask = stateIdx <= maxInd;
    SmoothedStates(t,mask) = Output(t).SmoothedStates(stateIdx(mask),1);
 CovX = Output(t).SmoothedStatesCov(stateIdx(mask),stateIdx(mask));
    CI(t,:,1) = SmoothedStates(t,1) + 1.96*sqrt(CovX(1,1))*[-1 1];
    if (max(stateIdx(mask)) > 1)
        CI(t,:,2) = SmoothedStates(t,2) + 1.96*sqrt(CovX(2,2))*[-1 1];
    end
end

SmoothedStates(1:SwitchTime,:) = 0;
CI(1:SwitchTime,:,:) = 0;
```

Plot the true state values, the smoothed states, and the 95% smoothed-state confidence intervals for each model.

```
figure
plot(1:T,x(:,1),'b',1:T,SmoothedStates(:,1),'k',1:T,CI(:,:,1),'--r');
title('AR(2) State Values')
xlabel('Period')
ylabel('State Value')
legend({'True state values','Smoothed state values','95% CI'});

figure
plot(1:T,x(:,2),'b',1:T,SmoothedStates(:,2),'k',1:T,CI(:,:,2),'--r');
title('MA(1) State Values')
xlabel('Period')
ylabel('State Value')
legend({'True state values','Smoothed state values','95% CI'});
```


AR(2) State Values
True state values
Smoothed state values
95% CI
State Value
Period
4
3
2
1
0
-1
-2
-3
-4
0
5
10
15
20
25
30
35
40
45
50



See Also

dssm | esitmate | smooth

Related Examples

- "Implicitly Create Time-Varying Diffuse State-Space Model" on page 8-35
- "Implicitly Create Diffuse State-Space Model Containing Regression Component" on page 8-30
- "Estimate Time-Varying Diffuse State-Space Model" on page 8-50
- "Filter Time-Varying Diffuse State-Space Model" on page 8-68

More About

- "What Are State-Space Models?" on page 8-3
- "What Is the Kalman Filter?" on page 8-8

Smooth States of State-Space Model Containing Regression Component

This example shows how to smooth states of a time-invariant, state-space model that contains a regression component.

Suppose that the linear relationship between the change in the unemployment rate and the nominal gross national product (nGNP) growth rate is of interest. Suppose further that the first difference of the unemployment rate is an ARMA(1,1) series. Symbolically, and in state-space form, the model is

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix} = \begin{bmatrix} \phi & \theta \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix} u_{1,t}$$
$$y_t - \beta Z_t = x_{1,t} + \sigma \varepsilon_t,$$

where:

- $x_{1,t}$ is the change in the unemployment rate at time t.
- $x_{2,t}$ is a dummy state for the MA(1) effect.
- $y_{1,t}$ is the observed change in the unemployment rate being deflated by the growth rate of nGNP (Z_t).
- $u_{1,t}$ is the Gaussian series of state disturbances having mean 0 and standard deviation 1.
- ε_t is the Gaussian series of observation innovations having mean 0 and standard deviation σ.

Load the Nelson-Plosser data set, which contains the unemployment rate and nGNP series, among other things.

```
load Data_NelsonPlosser
```

Preprocess the data by taking the natural logarithm of the nGNP series, and the first difference of each series. Also, remove the starting `NaN` values from each series.

```
isNaN = any(ismissing(DataTable),2);       % Flag periods containing NaNs
gnpn = DataTable.GNPN(~isNaN);
u = DataTable.UR(~isNaN);
T = size(gnpn,1);                          % Sample size
Z = [ones(T-1,1) diff(log(gnpn))];
y = diff(u);
```

Though this example removes missing values, the software can accommodate series containing missing values in the Kalman filter framework.

Specify the coefficient matrices.

```
A = [NaN NaN; 0 0];
B = [1; 1];
C = [1 0];
D = NaN;
```

Specify the state-space model using `ssm`.

```
Mdl = ssm(A,B,C,D);
```

Estimate the model parameters. Specify the regression component and its initial value for optimization using the `'Predictors'` and `'Beta0'` name-value pair arguments, respectively. Restrict the estimate of σ to all positive, real numbers.

```
params0 = [0.2 0.2 0.1];
[EstMdl,estParams] = estimate(Mdl,y,params0,'Predictors',Z,...
    'Beta0',[0.2 0.1],'lb',[-Inf,-Inf,0,-Inf,-Inf]);
```

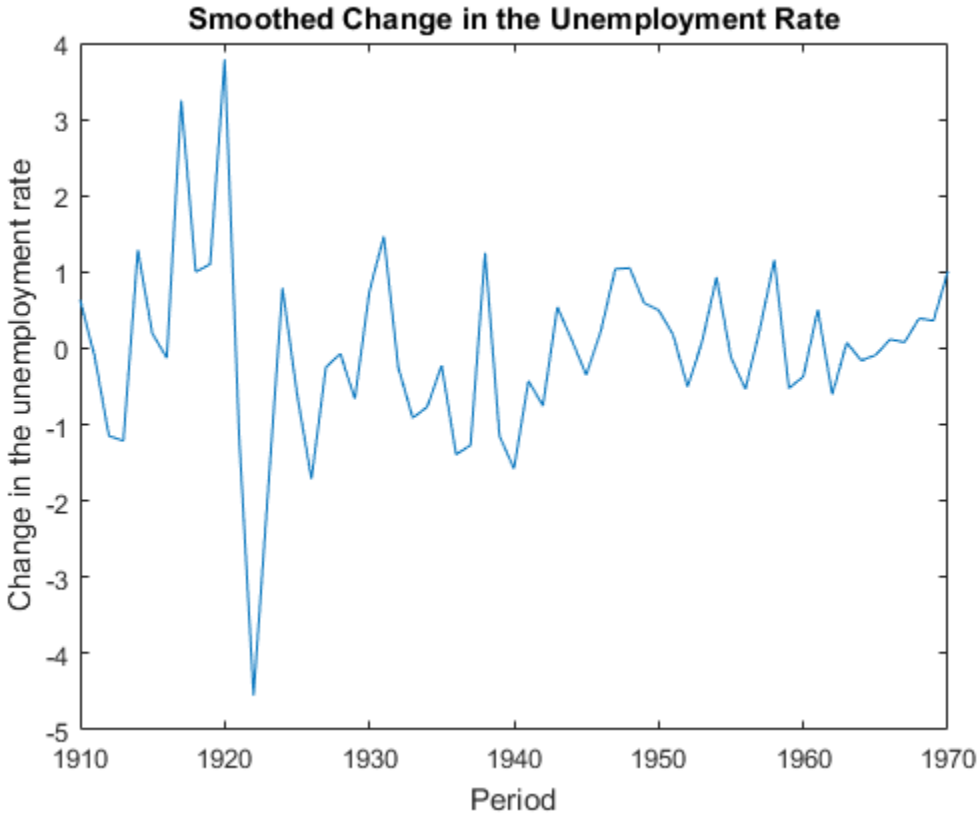
```
Method: Maximum likelihood (fmincon)
Sample size: 61
Logarithmic  likelihood:     -99.7245
Akaike   info criterion:      209.449
Bayesian info criterion:      220.003
           |      Coeff       Std Err    t Stat     Prob
-------------------------------------------------------------
 c(1)      |  -0.34098       0.29608    -1.15164  0.24948
 c(2)      |   1.05003       0.41377     2.53771  0.01116
 c(3)      |   0.48592       0.36790     1.32079  0.18657
 y <- z(1) |   1.36121       0.22338     6.09358   0
 y <- z(2) | -24.46711       1.60018   -15.29024   0
           |
           |    Final State   Std Dev     t Stat    Prob
 x(1)      |   1.01264       0.44690     2.26592  0.02346
 x(2)      |   0.77718       0.58917     1.31912  0.18713
```

`EstMdl` is an `ssm` model, and you can access its properties using dot notation.

Smooth the states. `EstMdl` does not store the data or the regression coefficients, so you must pass in them in using the name-value pair arguments `'Predictors'` and `'Beta'`,

respectively. Plot the smoothed states. Recall that the first state is the change in the unemployment rate, and the second state helps build the first.

```
SmoothedX = smooth(EstMdl,y,'Predictors',Z,'Beta',estParams(end-1:end));

figure
plot(dates(end-(T-1)+1:end),SmoothedX(:,1));
xlabel('Period')
ylabel('Change in the unemployment rate')
title('Smoothed Change in the Unemployment Rate')
```



See Also

estimate | filter | smooth | ssm

Related Examples

- "Create State-Space Model Containing ARMA State" on page 8-24
- "Estimate State-Space Model Containing Regression Component" on page 8-55
- "Filter States of State-Space Model Containing Regression Component" on page 8-76

Simulate States and Observations of Time-Invariant State-Space Model

This example shows how to simulate states and observations of a known, time-invariant state-space model.

Suppose that a latent process is an AR(1) model. Subsequently, the state equation is

$$x_t = 0.5x_{t-1} + u_t,$$

where u_t is Gaussian with mean 0 and standard deviation 1.

Generate a random series of 100 observations from x_t, assuming that the series starts at 1.5.

```
T = 100;
ARMdl = arima('AR',0.5,'Constant',0,'Variance',1);
x0 = 1.5;
rng(1); % For reproducibility
x = simulate(ARMdl,T,'Y0',x0);
```

Suppose further that the latent process is subject to additive measurement error. Subsequently, the observation equation is

$$y_t = x_t + \varepsilon_t,$$

where ε_t is Gaussian with mean 0 and standard deviation 0.75. Together, the latent process and observation equations compose a state-space model.

Use the random latent state process (`x`) and the observation equation to generate observations.

```
y = x + 0.75*randn(T,1);
```

Specify the four coefficient matrices.

```
A = 0.5;
B = 1;
C = 1;
D = 0.75;
```

Specify the state-space model using the coefficient matrices.

```
Mdl = ssm(A,B,C,D)
```

```
Mdl =


State-space model type: <a href="matlab: doc ssm">ssm</a>

State vector length: 1
Observation vector length: 1
State disturbance vector length: 1
Observation innovation vector length: 1
Sample size supported by model: Unlimited

State variables: x1, x2,...
State disturbances: u1, u2,...
Observation series: y1, y2,...
Observation innovations: e1, e2,...

State equation:
x1(t) = (0.50)x1(t-1) + u1(t)

Observation equation:
y1(t) = x1(t) + (0.75)e1(t)

Initial state distribution:

Initial state means
 x1
  0

Initial state covariance matrix
     x1
 x1  1.33

State types
     x1
 Stationary
```

`Mdl` is an `ssm` model. Verify that the model is correctly specified using the display in the Command Window. The software infers that the state process is stationary. Subsequently, the software sets the initial state mean and covariance to the mean and variance of the stationary distribution of an AR(1) model.

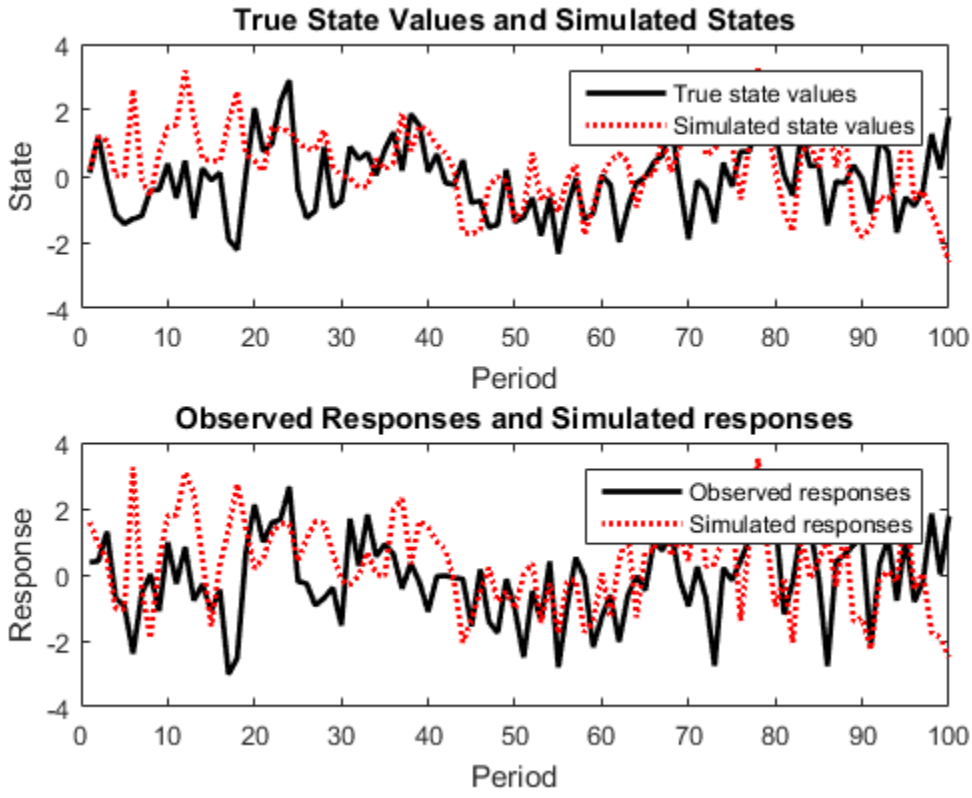
Simulate one path each of states and observations. Specify that the paths span 100 periods.

```
[simY,simX] = simulate(Mdl,100);
```

`simY` is a 100-by-1 vector of simulated responses. `simX` is a 100-by-1 vector of simulated states.

Plot the true state values with the simulated states. Also, plot the observed responses with the simulated responses.

```
figure
subplot(2,1,1)
plot(1:T,x,'-k',1:T,simX,':r','LineWidth',2)
title({'True State Values and Simulated States'})
xlabel('Period')
ylabel('State')
legend({'True state values','Simulated state values'})
subplot(2,1,2)
plot(1:T,y,'-k',1:T,simY,':r','LineWidth',2)
title({'Observed Responses and Simulated responses'})
xlabel('Period')
ylabel('Response')
legend({'Observed responses','Simulated responses'})
```



By default, `simulate` simulates one path for each state and observation in the state-space model. To conduct a Monte Carlo study, specify to simulate a large number of paths.

See Also

simulate | ssm

Related Examples

- "Explicitly Create State-Space Model Containing Known Parameter Values" on page 8-17
- "Simulate Time-Varying State-Space Model" on page 8-107

Simulate Time-Varying State-Space Model

This example shows how to generate data from a known model, fit a state-space model to the data, and then simulate series from the fitted model.

Suppose that a set of latent processes comprises an AR(2) and an MA(1) model. There are 50 periods, and the MA(1) process drops out of the model for the final 25 periods. Subsequently, the state equation for the first 25 periods is

$$x_{1,t} = 0.7x_{1,t-1} - 0.2x_{1,t-2} + u_{1,t}$$
$$x_{2,t} = u_{2,t} + 0.6u_{2,t-1},$$

and for the last 25 periods, it is

$$x_{1,t} = 0.7x_{1,t-1} - 0.2x_{1,t-2} + u_{1,t},$$

where $u_{1,t}$ and $u_{2,t}$ are Gaussian with mean 0 and standard deviation 1.

Assuming that the series starts at 1.5 and 1, respectively, generate a random series of 50 observations from $x_{1,t}$ and $x_{2,t}$.

```
T = 50;
ARMdl = arima('AR',{0.7,-0.2},'Constant',0,'Variance',1);
MAMdl = arima('MA',0.6,'Constant',0,'Variance',1);
x0 = [1.5 1; 1.5 1];
rng(1);
x = [simulate(ARMdl,T,'Y0',x0(:,1)),...
    [simulate(MAMdl,T/2,'Y0',x0(:,2));nan(T/2,1)]];
```

The last 25 values for the simulated MA(1) data are `NaN` values.

Suppose further that the latent processes are measured using

$$y_t = 2\left(x_{1,t} + x_{2,t}\right) + \varepsilon_t,$$

for the first 25 periods, and

$$y_t = 2x_{1,t} + \varepsilon_t$$

for the last 25 periods, where ε_t is Gaussian with mean 0 and standard deviation 1.

Use the random latent state process (`x`) and the observation equation to generate observations.

```
y = 2*nansum(x')'+randn(T,1);
```

Together, the latent process and observation equations compose a state-space model. Supposing that the coefficients are unknown parameters, the state-space model is

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \\ x_{3,t} \\ x_{4,t} \end{bmatrix} = \begin{bmatrix} \phi_1 & \phi_2 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \theta_1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \\ x_{3,t-1} \\ x_{4,t-1} \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} u_{1,t} \\ u_{2,t} \end{bmatrix}$$
$$y_t = a(x_{1,t} + x_{3,t}) + \varepsilon_t$$

for the first 25 periods,

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix} = \begin{bmatrix} \phi_1 & \phi_2 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \\ x_{3,t-1} \\ x_{4,t-1} \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} u_{1,t}$$
$$y_t = bx_{1,t} + \varepsilon_t$$

for period 26, and

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix} = \begin{bmatrix} \phi_1 & \phi_2 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} u_{1,t}$$
$$y_t = bx_{1,t} + \varepsilon_t$$

for the last 24 periods.

Write a function that specifies how the parameters in `params` map to the state-space model matrices, the initial state values, and the type of state.

```
% Copyright 2015 The MathWorks, Inc.

function [A,B,C,D,Mean0,Cov0,StateType] = AR2MAParamMap(params,T)
%AR2MAParamMap Time-variant state-space model parameter mapping function
%
% This function maps the vector params to the state-space matrices (A, B,
% C, and D), the initial state value and the initial state variance (Mean0
% and Cov0), and the type of state (StateType). From periods 1 to T/2, the
```

```
% state model is an AR(2) and an MA(1) model, and the observation model is
% the sum of the two states. From periods T/2 + 1 to T, the state model is
% just the AR(2) model.
    A1 = {[params(1) params(2) 0 0; 1 0 0 0; 0 0 0 params(3); 0 0 0 0]};
    B1 = {[1 0; 0 0; 0 1; 0 1]};
    C1 = {params(4)*[1 0 1 0]};
    Mean0 = ones(4,1);
    Cov0 = 10*eye(4);
    StateType = [0 0 0 0];
    A2 = {[params(1) params(2) 0 0; 1 0 0 0]};
    B2 = {[1; 0]};
    A3 = {[params(1) params(2); 1 0]};
    B3 = {[1; 0]};
    C3 = {params(5)*[1 0]};
    A = [repmat(A1,T/2,1);A2;repmat(A3,(T-2)/2,1)];
    B = [repmat(B1,T/2,1);B2;repmat(B3,(T-2)/2,1)];
    C = [repmat(C1,T/2,1);repmat(C3,T/2,1)];
    D = 1;
end
```

Save this code as a file named `AR2MAParamMap` on your MATLAB® path.

Create the state-space model by passing the function `AR2MAParamMap` as a function handle to `ssm`.

```
Mdl = ssm(@(params)AR2MAParamMap(params,T));
```

`ssm` implicitly creates the state-space model. Usually, you cannot verify an implicitly defined state-space model.

Pass the observed responses (`y`) to `estimate` to estimate the parameters. Specify an arbitrary set of positive initial values for the unknown parameters.

```
params0 = 0.1*ones(5,1);
EstMdl = estimate(Mdl,y,params0);
```

```
Method: Maximum likelihood (fminunc)
Sample size: 50
Logarithmic  likelihood:     -114.957
Akaike   info criterion:      239.913
Bayesian info criterion:      249.473
      |     Coeff       Std Err   t Stat     Prob
-------------------------------------------------
```

```
 c(1) |  0.47870       0.26634    1.79733  0.07229
 c(2) |  0.00809       0.27179    0.02975  0.97626
 c(3) |  0.55735       0.80958    0.68844  0.49118
 c(4) |  1.62679       0.41622    3.90848  0.00009
 c(5) |  1.90021       0.49563    3.83391  0.00013
      |
      |   Final State   Std Dev    t Stat    Prob
 x(1) | -0.81229       0.46815   -1.73511  0.08272
 x(2) | -0.31449       0.45918   -0.68490  0.49341
```

`EstMdl` is an `ssm` model containing the estimated coefficients. Likelihood surfaces of state-space models might contain local maxima. Therefore, it is good practice to try several initial parameter values, or consider using `refine`.

Simulate one path of responses, states, state disturbances, and observation innovations from `Mdl`. Specify that each path has `T` periods of simulated variants.

```
[Y,X,U,E]= simulate(EstMdl,T);
```

- `Y` is a `T`-by-1 vector of simulated observations.
- `X` is a `T`-by-1 cell vector of simulated states. Cells 1 through 25 contain 4-by-1 vectors, and cells 26 through 50 contain 2-by-1 vectors.
- `U` is a `T`-by-1 cell vector of simulated state disturbances. Cells 1 through 25 contain 4-by-1 vectors, and cells 26 through 50 contain 2-by-1 vectors.
- `E` is a `T`-by-1 vector of simulated observation innovations.

Access a cell of the simulated states using cell indexing, for example access cell 5 using `X{5}`.

```
simStatesPeriod5 = X{5}
```

```
simStatesPeriod5 =

   -0.8660
   -2.2826
   -0.7071
    0.2177
```

See Also

estimate | refine | simulate | ssm

Related Examples

- "Implicitly Create Time-Invariant State-Space Model" on page 8-22
- "Estimate Time-Varying State-Space Model" on page 8-45
- "Simulate States and Observations of Time-Invariant State-Space Model" on page 8-103

Simulate States of Time-Varying State-Space Model Using Simulation Smoother

This example generates data from a known model, fits a state-space model to the data, and then simulates series from the fitted model using the simulation smoother.

Suppose that a latent process comprises an AR(2) and an MA(1) model. There are 50 periods, and the MA(1) process drops out of the model for the final 25 periods. Subsequently, the state equation for the first 25 periods is

$$x_{1,t} = 0.7x_{1,t-1} - 0.2x_{1,t-2} + u_{1,t}$$
$$x_{2,t} = u_{2,t} + 0.6u_{2,t-1},$$

and for the last 25 periods, it is

$$x_{1,t} = 0.7x_{1,t-1} - 0.2x_{1,t-2} + u_{1,t},$$

where $u_{1,t}$ and $u_{2,t}$ are Gaussian with mean 0 and standard deviation 1.

Assuming that the series starts at 1.5 and 1, respectively, generate a random series of 50 observations from $x_{1,t}$ and $x_{2,t}$.

```
T = 50;
ARMdl = arima('AR',{0.7,-0.2},'Constant',0,'Variance',1);
MAMdl = arima('MA',0.6,'Constant',0,'Variance',1);
x0 = [1.5 1; 1.5 1];
rng(1);
x = [simulate(ARMdl,T,'Y0',x0(:,1)),...
    [simulate(MAMdl,T/2,'Y0',x0(:,2));nan(T/2,1)]];
```

The last 25 values for the simulated MA(1) data are `NaN` values.

Suppose further that the latent processes are measured using

$$y_t = 2\left(x_{1,t} + x_{2,t}\right) + \varepsilon_t,$$

for the first 25 periods, and

$$y_t = 2x_{1,t} + \varepsilon_t$$

for the last 25 periods, where ε_t is Gaussian with mean 0 and standard deviation 1.

Use the random latent state process (`x`) and the observation equation to generate observations.

```
y = 2*nansum(x')'+randn(T,1);
```

Together, the latent process and observation equations compose a state-space model. Supposing that the coefficients are unknown parameters, the state-space model is

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \\ x_{3,t} \\ x_{4,t} \end{bmatrix} = \begin{bmatrix} \phi_1 & \phi_2 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \theta_1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \\ x_{3,t-1} \\ x_{4,t-1} \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} u_{1,t} \\ u_{2,t} \end{bmatrix}$$

$$y_t = a(x_{1,t} + x_{3,t}) + \varepsilon_t$$

for the first 25 periods,

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix} = \begin{bmatrix} \phi_1 & \phi_2 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \\ x_{3,t-1} \\ x_{4,t-1} \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} u_{1,t}$$

$$y_t = bx_{1,t} + \varepsilon_t$$

for period 26, and

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix} = \begin{bmatrix} \phi_1 & \phi_2 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} u_{1,t}$$

$$y_t = bx_{1,t} + \varepsilon_t$$

for the last 24 periods.

Write a function that specifies how the parameters in `params` map to the state-space model matrices, the initial state values, and the type of state.

```
% Copyright 2015 The MathWorks, Inc.

function [A,B,C,D,Mean0,Cov0,StateType] = AR2MAParamMap(params,T)
%AR2MAParamMap Time-variant state-space model parameter mapping function
%
% This function maps the vector params to the state-space matrices (A, B,
```

```
% C, and D), the initial state value and the initial state variance (Mean0
% and Cov0), and the type of state (StateType). From periods 1 to T/2, the
% state model is an AR(2) and an MA(1) model, and the observation model is
% the sum of the two states. From periods T/2 + 1 to T, the state model is
% just the AR(2) model.
    A1 = {[params(1) params(2) 0 0; 1 0 0 0; 0 0 0 params(3); 0 0 0 0]};
    B1 = {[1 0; 0 0; 0 1; 0 1]};
    C1 = {params(4)*[1 0 1 0]};
    Mean0 = ones(4,1);
    Cov0 = 10*eye(4);
    StateType = [0 0 0 0];
    A2 = {[params(1) params(2) 0 0; 1 0 0 0]};
    B2 = {[1; 0]};
    A3 = {[params(1) params(2); 1 0]};
    B3 = {[1; 0]};
    C3 = {params(5)*[1 0]};
    A = [repmat(A1,T/2,1);A2;repmat(A3,(T-2)/2,1)];
    B = [repmat(B1,T/2,1);B2;repmat(B3,(T-2)/2,1)];
    C = [repmat(C1,T/2,1);repmat(C3,T/2,1)];
    D = 1;
end
```

Save this code as a file named `AR2MAParamMap` on your MATLAB® path.

Create the state-space model by passing the function `AR2MAParamMap` as a function handle to `ssm`.

```
Mdl = ssm(@(params)AR2MAParamMap(params,T));
```

`ssm` implicitly creates the state-space model. Usually, you cannot verify an implicitly defined state-space model.

Simulate one path of states from Mdl using the simulation smoother. Specify that the parameter-to-matrix mapping function has seven output arguments. Also, specify the unknown values of the parameters.

```
simParams = [0.48 0.0081 0.56 1.63 1.9];
X = simsmooth(Mdl,y,'NumOut',7,'Params',simParams);
```

`X` is a `T`-by-1 cell vector of simulated states. Cells 1 through 25 contain 4-by-1 vectors, and cells 26 through 50 contain 2-by-1 vectors.

Access a cell using cell indexing, for example, access cell 5 using `X{5}`.

```
simStatesPeriod5 = X{5}
```

```
simStatesPeriod5 =

   -1.7591
   -1.5404
   -1.5171
   -1.1417
```

See Also

estimate | refine | simsmooth | simulate | ssm

Related Examples

- "Simulate States and Observations of Time-Invariant State-Space Model" on page 8-103
- "Compare Simulation Smoother to Smoothed States" on page 8-162
- "Estimate Random Parameter of State-Space Model" on page 8-116

More About

- "What Are State-Space Models?" on page 8-3
- "What Is the Kalman Filter?" on page 8-8

Estimate Random Parameter of State-Space Model

This example shows how to estimate a random, autoregressive coefficient of a state in a state-space model. That is, this example takes a Bayesian view of state-space model parameter estimation by using the "zig-zag" estimation method.

Suppose that two states ($x_{1,t}$ and $x_{2,t}$) represent the net exports of two countries at the end of the year.

- $x_{1,t}$ is a unit root process with a disturbance variance of σ_1^2.
- $x_{2,t}$ is an AR(1) process with an unknown, random coefficient and a disturbance variance of σ_2^2.
- An observation (y_t) is the exact sum of the two net exports. That is, the net exports of the individual states are unknown.

Symbolically, the true state-space model is

$$\begin{aligned} x_{1,t} &= x_{1,t-1} + \sigma_1 u_{1,t} \\ x_{2,t} &= \phi x_{2,t-1} + \sigma_2 u_{2,t} \\ y_t &= x_{1,t} + x_{2,t} \end{aligned}$$

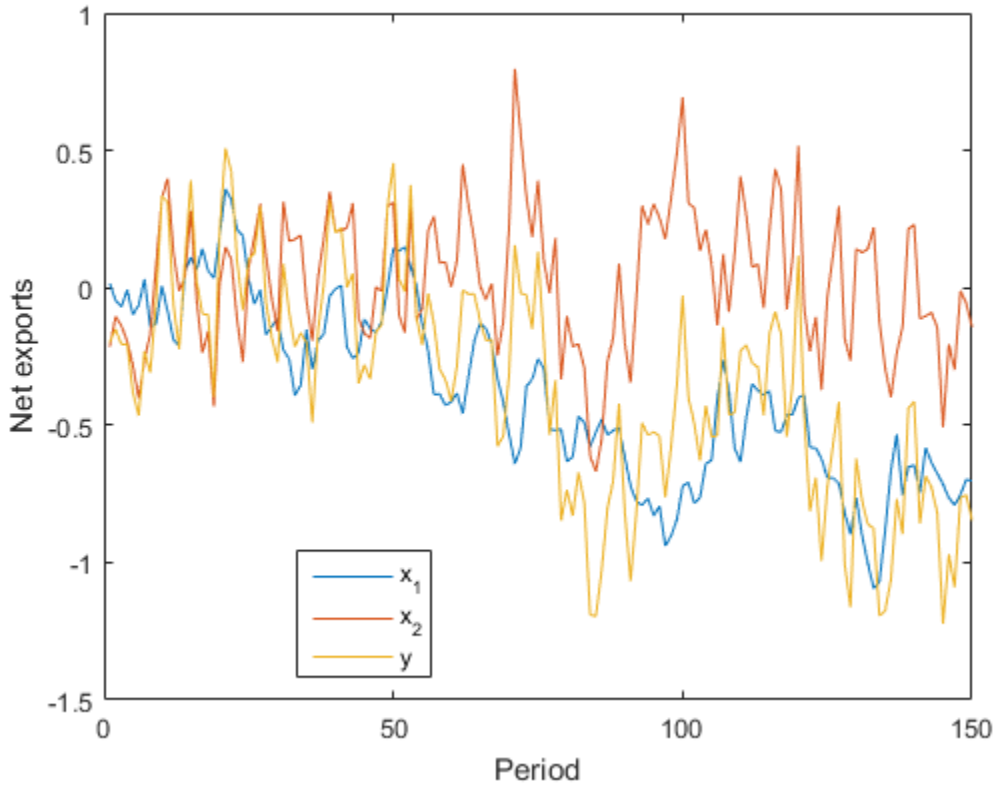
Simulate Data

Simulate 100 years of net exports from:

- A unit root process with a mean zero, Gaussian noise series that has variance 0.1^2.
- An AR(1) process with an autoregressive coefficient of 0.6 and a mean zero, Gaussian noise series that has variance 0.2^2.
- $x_{1,0} = x_{2,0} = 0$.
- Create an observation series by summing the two net exports per year.

```
rng(100); % For reproducibility
T = 150;
sigma1 = 0.1;
sigma2 = 0.2;
phi = 0.6;

u1 = randn(T,1)*sigma1;
x1 = cumsum(u1);
```

```
Mdl2 = arima('AR',phi,'Variance',sigma2^2,'Constant',0);
x2 = simulate(Mdl2,T,'YO',0);
y = x1 + x2;

figure;
plot([x1 x2 y])
legend('x_1','x_2','y','Location','Best');
ylabel('Net exports');
xlabel('Period');
```



The Zig-Zag Estimation Method

Treat ϕ as if it is unknown and random, and use the zig-zag method to recover its distribution. To implement the zig-zag method:

1. Choose an initial value for ϕ in the interval (-1,1), and denote it ϕ_z.

2. Create the true state-space model, that is, an `ssm` model object that represents the data-generating process.

3. Use the simulation smoother (`simsmooth`) to draw a random path from the distribution of the second smoothed states. Symbolically, $x_{2,z,t} \sim P(x_{2,t}|y_t, x_{1,t}, \phi = \phi_z)$.

4. Create another state-space model that has this form

$$\phi_{z,t} = \phi_{z,t-1}$$
$$x_{2,z,t} = x_{2,z,t-1}\phi_{z,t} + 0.8u_{2,t}$$

In words, $\phi_{z,t}$ is a static state and $x_{2,z,t}$ is an "observed" series with time varying coefficient $C_t = x_{2,z,t-1}$.

5. Use the simulation smoother to draw a random path from the distribution of the smoothed $\phi_{z,t}$ series. Symbolically, $\phi_{z,t} \sim P(\phi|x_{2,z,t})$, where $x_{2,z,t}$ encompasses the structure of the true state-space model and the observations. $\phi_{z,t}$ is static, so you can just reserve one value (ϕ_z).

6. Repeat steps 2 - 5 many times and store ϕ_z each iteration.

7. Perform diagnostic checks on the simulation series. That is, construct:

- Trace plots to determine the burn in period and whether the Markov chain is mixing well.
- Autocorrelation plots to determine how many draws need removing to obtain a well-mixed Markov chain.

8. The remaining series represents draws from the posterior distribution of ϕ. You can compute descriptive statistics, or plot a histogram to determine the qualities of the distribution.

Estimate Random Coefficient Using Zig-Zag Method

Specify initial values, preallocate, and create the true state-space model.

```
phi0 = -0.3;              % Initial value of phi
Z = 1000;                 % Number of times to iterate the zig-zag method
phiz = [phi0; nan(Z,1)]; % Preallocate

A = [1 0; 0 NaN];
B = [sigma1; sigma2];
C = [1 1];
Mdl = ssm(A,B,C,'StateType',[2; 0]);
```

Mdl is an `ssm` model object. The `NaN` acts as a placeholder for ϕ.

Iterate steps 2 - 5 of the zig-zag method.

```
for j = 2:(Z + 1);
    % Draw a random path from smoothed x_2 series.
    xz = simsmooth(Mdl,y,'Params',phiz(j-1));
    % The second column of xz is a draw from the posterior distribution of x_2.

    % Create the intermediate state-space model.
    Az = 1;
    Bz = 0;
    Cz = num2cell(xz((1:(end - 1)),2));
    Dz = sigma2;
    Mdlz = ssm(Az,Bz,Cz,Dz,'StateType',2);

    % Draw a random path from the smoothed phiz series.
    phizvec = simsmooth(Mdlz,xz(2:end,2));
    phiz(j) = phizvec(1);
    % phiz(j) is a draw from the posterior distribution of phi
end
```

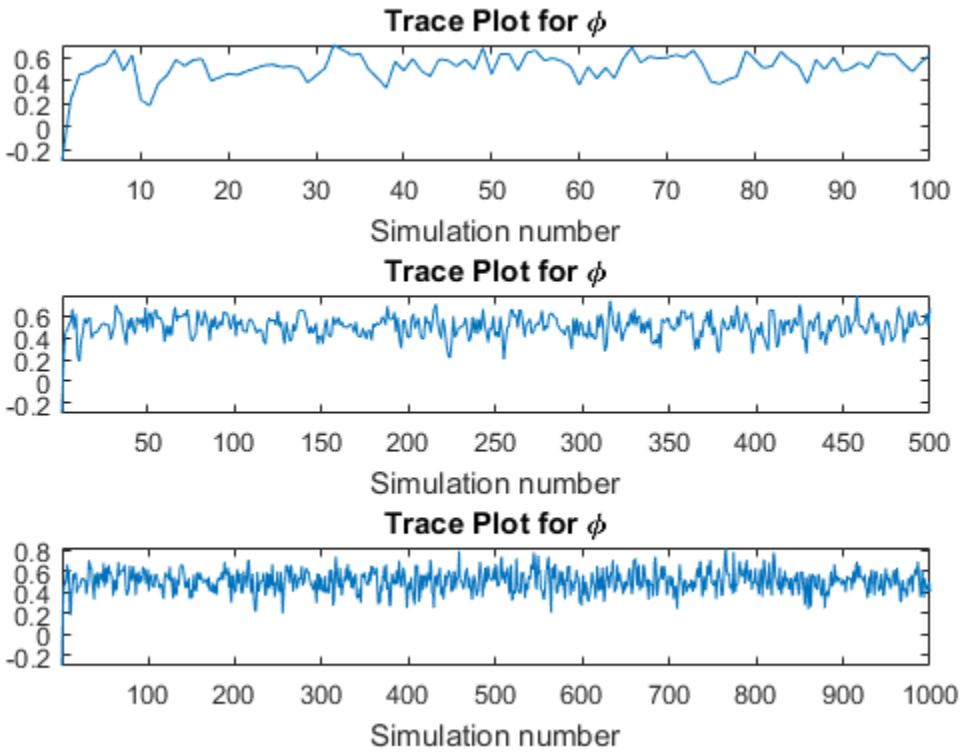
`phiz` is a Markov chain. Before analyzing the posterior distribution of ϕ, you should assess whether to impose a burn-in period, or the severity of the autocorrelation in the chain.

Determine Quality of Simulation

Draw a trace plot for the first 100, 500, and all of the random draws.

```
vec = [100 500 Z];
figure;
for j = 1:3;
    subplot(3,1,j)
```

```
    plot(phiz(1:vec(j)));
    title('Trace Plot for \phi');
    xlabel('Simulation number');
    axis tight;
end
```

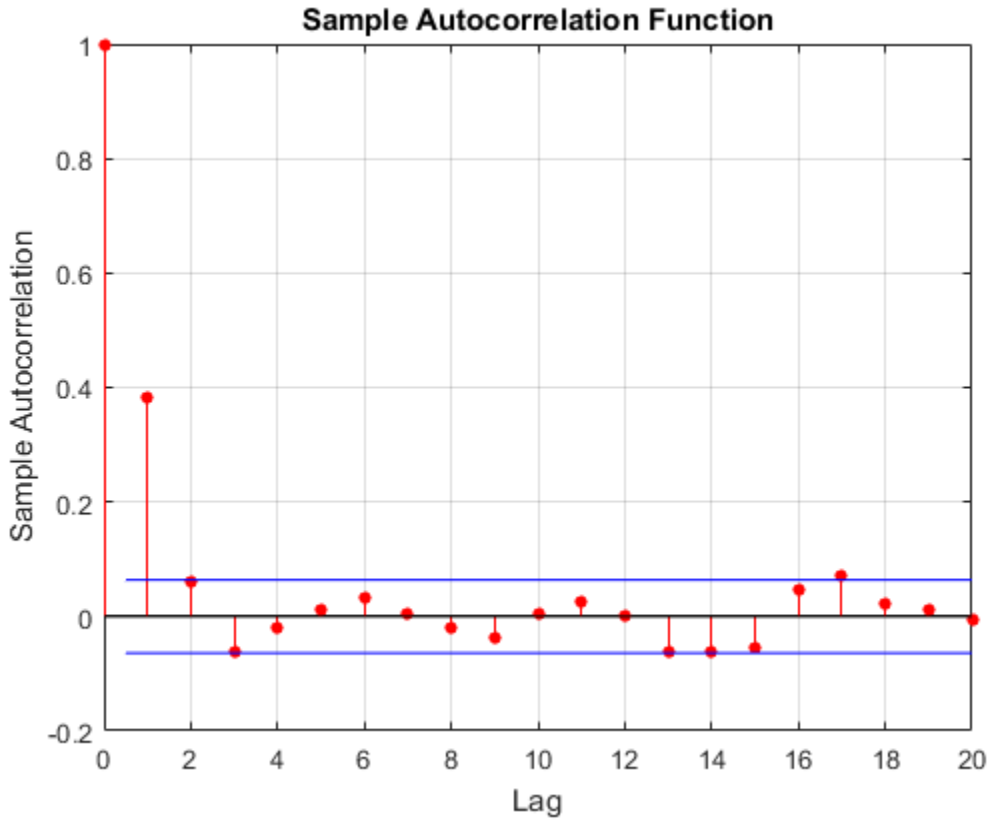


According to the first plot, transient effects die down after about 20 draws. Therefore, a short burn-in period should suffice. The plot of the entire simulation shows that the series settles around a center.

Plot the autocorrelation function of the series after removing the first 20 draws.

```
burnOut = 21:Z;
```

```
figure;
autocorr(phiz(burnOut));
```



The autocorrelation function dies out rather quickly. It doesn't seem like autocorrelation in the chain is an issue.

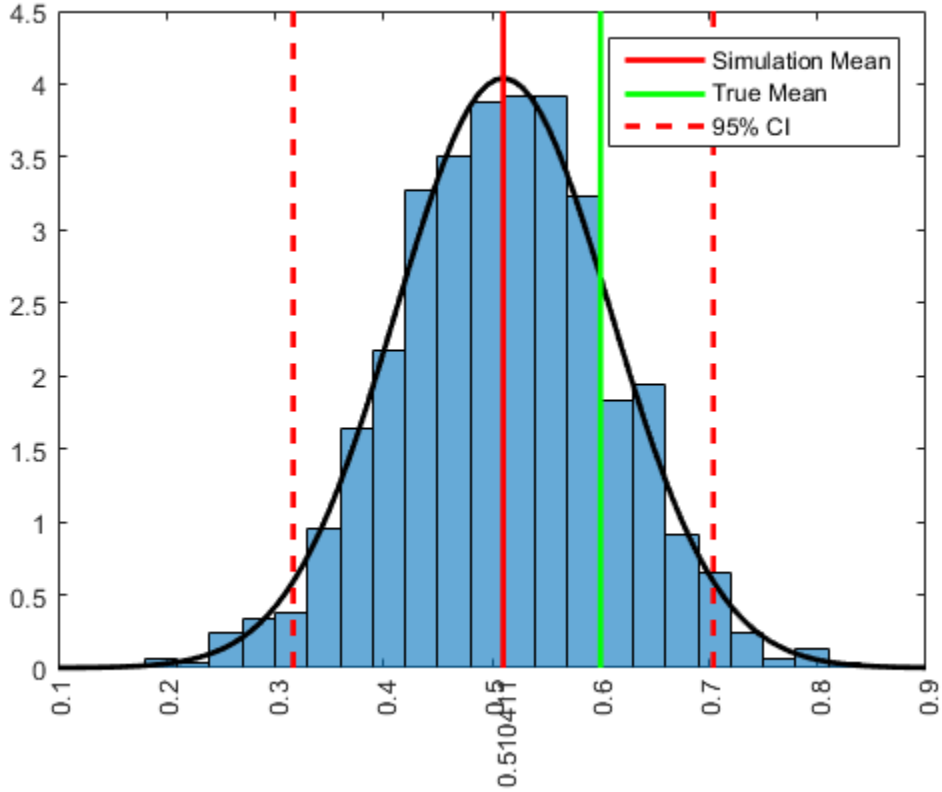
Determine qualities of the posterior distribution of ϕ by computing simulation statistics and by plotting a histogram of the reduced set of random draws.

```
xbar = mean(phiz(burnOut))
xstd = std(phiz(burnOut))
ci = norminv([0.025,0.975],xbar,xstd); % 95% confidence interval
```

```
figure;
histogram(phiz(burnOut),'Normalization','pdf');
h = gca;
hold on;
simX = linspace(h.XLim(1),h.XLim(2),100);
simPDF = normpdf(simX,xbar,xstd);
plot(simX,simPDF,'k','LineWidth',2);
h1 = plot([xbar xbar],h.YLim,'r','LineWidth',2);
h2 = plot([0.6 0.6],h.YLim,'g','LineWidth',2);
h3 = plot([ci(1) ci(1)],h.YLim,'--r',...
    [ci(2) ci(2)],h.YLim,'--r','LineWidth',2);
legend([h1 h2 h3(1)],{'Simulation Mean','True Mean','95% CI'});
h.XTick = sort([h.XTick xbar]);
h.XTickLabel{h.XTick == xbar} = xbar;
h.XTickLabelRotation = 90;
```

```
xbar =

    0.5104
```

```
xstd =

    0.0988
```



The posterior distribution of ϕ is roughly normal with mean and standard deviation approximately 0.51 and 0.1, respectively. The true mean of ϕ is 0.6, and it is less than one standard deviation to the right of the simulation mean.

Compute the maximum likelihood estimate of ϕ. That is, treat ϕ as a fixed, but unknown parameter, and then estimate `Mdl` using the Kalman filter and maximum likelihood.

```
[~,estParams] = estimate(Mdl,y,phi0)
```

```
Method: Maximum likelihood (fminunc)
Sample size: 150
Logarithmic  likelihood:     -10.1434
```

```
Akaike   info criterion:      22.2868
Bayesian info criterion:      25.2974
      |     Coeff       Std Err     t Stat       Prob
-------------------------------------------------------
 c(1) |  0.53590       0.19183    2.79360      0.00521
      |
      |   Final State   Std Dev    t Stat        Prob
 x(1) | -0.85059       0.00000   -6.45811e+08   0
 x(2) |  0.00454        0          Inf          0

estParams =

    0.5359
```

The MLE of ϕ is 0.54. Both estimates are within one standard deviation or standard error from the true value of ϕ.

See Also

estimate | refine | simsmooth | simulate | ssm

Related Examples

- "Simulate States and Observations of Time-Invariant State-Space Model" on page 8-103
- "Compare Simulation Smoother to Smoothed States" on page 8-162

More About

- "What Are State-Space Models?" on page 8-3
- "What Is the Kalman Filter?" on page 8-8

Forecast State-Space Model Using Monte-Carlo Methods

This example shows how to forecast a state-space model using Monte-Carlo methods, and to compare the Monte-Carlo forecasts to the theoretical forecasts.

Suppose that the relationship between the change in the unemployment rate ($x_{1,t}$) and the nominal gross national product (nGNP) growth rate ($x_{3,t}$) can be expressed in the following, state-space model form.

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \\ x_{3,t} \\ x_{4,t} \end{bmatrix} = \begin{bmatrix} \phi_1 & \theta_1 & \gamma_1 & 0 \\ 0 & 0 & 0 & 0 \\ \gamma_2 & 0 & \phi_2 & \theta_2 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \\ x_{3,t-1} \\ x_{4,t-1} \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} u_{1,t} \\ u_{2,t} \end{bmatrix}$$

$$\begin{bmatrix} y_{1,t} \\ y_{2,t} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t} \\ x_{2,t} \\ x_{3,t} \\ x_{4,t} \end{bmatrix} + \begin{bmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{bmatrix} \begin{bmatrix} \varepsilon_{1,t} \\ \varepsilon_{2,t} \end{bmatrix},$$

where:

- $x_{1,t}$ is the change in the unemployment rate at time *t*.
- $x_{2,t}$ is a dummy state for the MA(1) effect on $x_{1,t}$.
- $x_{3,t}$ is the nGNP growth rate at time *t*.
- $x_{4,t}$ is a dummy state for the MA(1) effect on $x_{3,t}$.
- $y_{1,t}$ is the observed change in the unemployment rate.
- $y_{2,t}$ is the observed nGNP growth rate.
- $u_{1,t}$ and $u_{2,t}$ are Gaussian series of state disturbances having mean 0 and standard deviation 1.
- $\varepsilon_{1,t}$ is the Gaussian series of observation innovations having mean 0 and standard deviation σ_1.
- $\varepsilon_{2,t}$ is the Gaussian series of observation innovations having mean 0 and standard deviation σ_2.

Load the Nelson-Plosser data set, which contains the unemployment rate and nGNP series, among other things.

```
load Data_NelsonPlosser
```

Preprocess the data by taking the natural logarithm of the nGNP series, and the first difference of each. Also, remove the starting `NaN` values from each series.

```
isNaN = any(ismissing(DataTable),2);          % Flag periods containing NaNs
gnpn = DataTable.GNPN(~isNaN);
u = DataTable.UR(~isNaN);
T = size(gnpn,1);                            % Sample size
y = zeros(T-1,2);                            % Preallocate
y(:,1) = diff(u);
y(:,2) = diff(log(gnpn));
```

This example proceeds using series without `NaN` values. However, using the Kalman filter framework, the software can accommodate series containing missing values.

To determine how well the model forecasts observations, remove the last 10 observations for comparison.

```
numPeriods = 10;                    % Forecast horizon
isY = y(1:end-numPeriods,:);        % In-sample observations
oosY = y(end-numPeriods+1:end,:);   % Out-of-sample observations
```

Specify the coefficient matrices.

```
A = [NaN NaN NaN 0; 0 0 0 0; NaN 0 NaN NaN; 0 0 0 0];
B = [1 0; 1 0; 0 1; 0 1];
C = [1 0 0 0; 0 0 1 0];
D = [NaN 0; 0 NaN];
```

Specify the state-space model using `ssm`. Verify that the model specification is consistent with the state-space model.

```
Mdl = ssm(A,B,C,D)
```

```
Mdl =


State-space model type: <a href="matlab: doc ssm">ssm</a>

State vector length: 4
Observation vector length: 2
State disturbance vector length: 2
```

```
Observation innovation vector length: 2
Sample size supported by model: Unlimited
Unknown parameters for estimation: 8

State variables: x1, x2,...
State disturbances: u1, u2,...
Observation series: y1, y2,...
Observation innovations: e1, e2,...
Unknown parameters: c1, c2,...

State equations:
x1(t) = (c1)x1(t-1) + (c3)x2(t-1) + (c4)x3(t-1) + u1(t)
x2(t) = u1(t)
x3(t) = (c2)x1(t-1) + (c5)x3(t-1) + (c6)x4(t-1) + u2(t)
x4(t) = u2(t)

Observation equations:
y1(t) = x1(t) + (c7)e1(t)
y2(t) = x3(t) + (c8)e2(t)

Initial state distribution:

Initial state means are not specified.
Initial state covariance matrix is not specified.
State types are not specified.
```

Estimate the model parameters, and use a random set of initial parameter values for optimization. Restrict the estimate of σ_1 and σ_2 to all positive, real numbers using the `'lb'` name-value pair argument. For numerical stability, specify the Hessian when the software computes the parameter covariance matrix, using the `'CovMethod'` name-value pair argument.

```
rng(1);
params0 = rand(8,1);
[EstMdl,estParams] = estimate(Mdl,isY,params0,...
    'lb',[-Inf -Inf -Inf -Inf -Inf -Inf 0 0],'CovMethod','hessian');
```

```
Method: Maximum likelihood (fmincon)
Sample size: 51
Logarithmic  likelihood:      -170.92
Akaike   info criterion:       357.84
Bayesian info criterion:      373.295
```

```
      |     Coeff       Std Err    t Stat     Prob
-------------------------------------------------------
 c(1) |  0.06750       0.16548     0.40791  0.68334
 c(2) | -0.01372       0.05887    -0.23302  0.81575
 c(3) |  2.71201       0.27039    10.03006   0
 c(4) |  0.83816       2.84586     0.29452  0.76836
 c(5) |  0.06273       2.83471     0.02213  0.98234
 c(6) |  0.05197       2.56873     0.02023  0.98386
 c(7) |  0.00272       2.40764     0.00113  0.99910
 c(8) |  0.00016       0.13942     0.00113  0.99910
      |
      |   Final State   Std Dev     t Stat    Prob
 x(1) | -0.00000       0.00272    -0.00033  0.99973
 x(2) |  0.12237       0.92954     0.13164  0.89527
 x(3) |  0.04049       0.00016   256.67560   0
 x(4) |  0.01183       0.00016    72.49713   0
```

`EstMdl` is an `ssm` model, and you can access its properties using dot notation.

Filter the estimated, state-space model, and extract the filtered states and their variances from the final period.

```
[~,~,Output] = filter(EstMdl,isY);
```

Modify the estimated, state-space model so that the initial state means and covariances are the filtered states and their covariances of the final period. This sets up simulation over the forecast horizon.

```
EstMdl1 = EstMdl;
EstMdl1.Mean0 = Output(end).FilteredStates;
EstMdl1.Cov0 = Output(end).FilteredStatesCov;
```

Simulate `5e5` paths of observations from the fitted, state-space model `EstMdl`. Specify to simulate observations for each period.

```
numPaths = 5e5;
SimY = simulate(EstMdl1,10,'NumPaths',numPaths);
```

`SimY` is a `10`-by- `2`-by- `numPaths` array containing the simulated observations. The rows of `SimY` correspond to periods, the columns correspond to an observation in the model, and the pages correspond to paths.

Estimate the forecasted observations and their 95% confidence intervals in the forecast horizon.

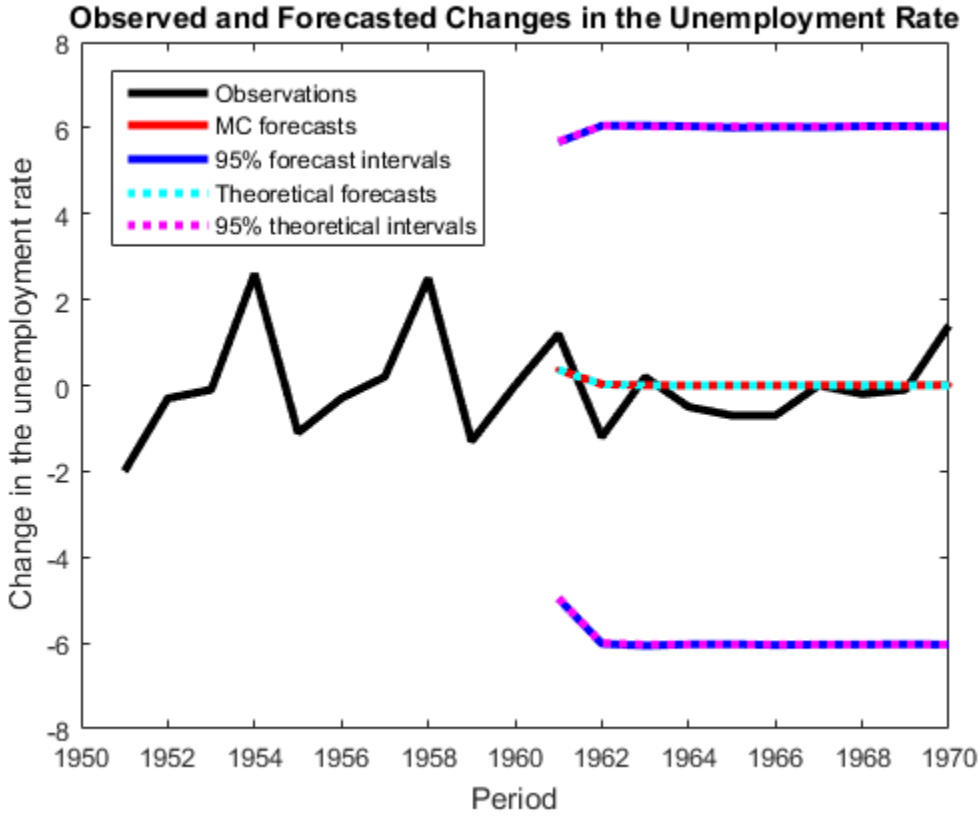
```
MCFY = mean(SimY,3);
CIFY = quantile(SimY,[0.025 0.975],3);
```

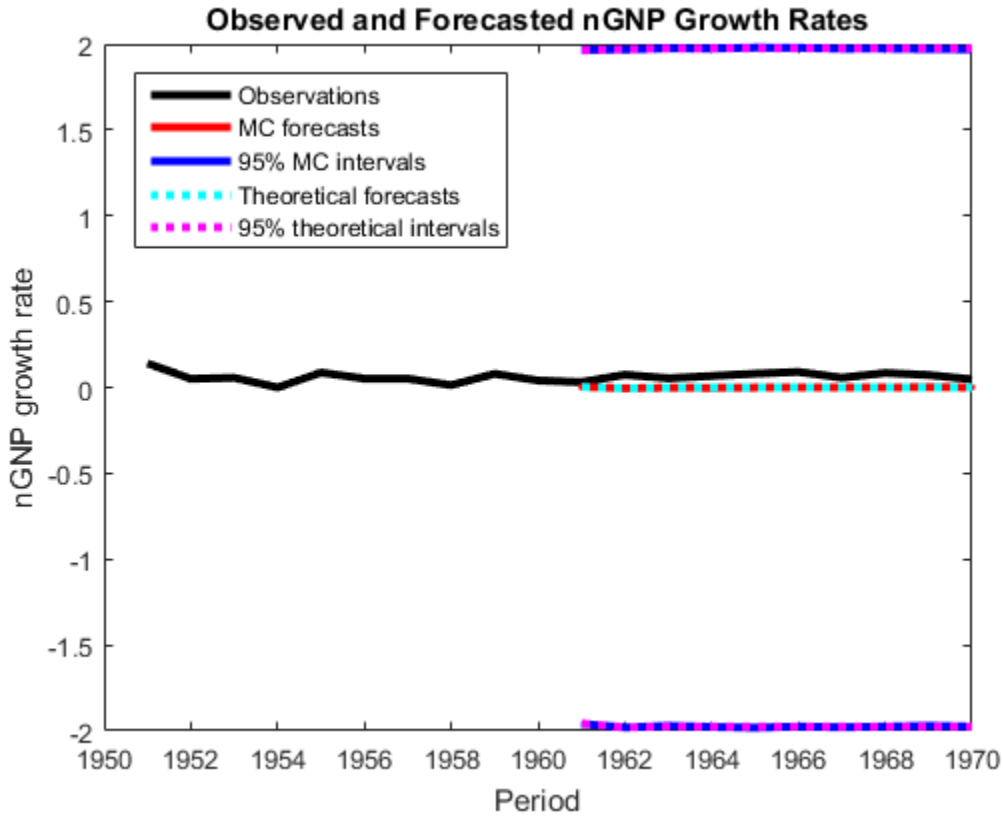
Estimate the theoretical forecast bands.

```
[Y,YMSE] = forecast(EstMdl,10,isY);
Lb = Y - sqrt(YMSE)*1.96;
Ub = Y + sqrt(YMSE)*1.96;
```

Plot the forecasted observations with their true values and the forecast intervals.

```
figure
h = plot(dates(end-numPeriods-9:end),[isY(end-9:end,1);oosY(:,1)],'-k',...
    dates(end-numPeriods+1:end),MCFY(end-numPeriods+1:end,1),'.-r',...
    dates(end-numPeriods+1:end),CIFY(end-numPeriods+1:end,1,1),'-b',...
    dates(end-numPeriods+1:end),CIFY(end-numPeriods+1:end,1,2),'-b',...
    dates(end-numPeriods+1:end),Y(:,1),':c',...
    dates(end-numPeriods+1:end),Lb(:,1),':m',...
    dates(end-numPeriods+1:end),Ub(:,1),':m',...
    'LineWidth',3);
xlabel('Period')
ylabel('Change in the unemployment rate')
legend(h([1,2,4:6]),{'Observations','MC forecasts',...
    '95% forecast intervals','Theoretical forecasts',...
    '95% theoretical intervals'},'Location','Best')
title('Observed and Forecasted Changes in the Unemployment Rate')

figure
h = plot(dates(end-numPeriods-9:end),[isY(end-9:end,2);oosY(:,2)],'-k',...
    dates(end-numPeriods+1:end),MCFY(end-numPeriods+1:end,2),'.-r',...
    dates(end-numPeriods+1:end),CIFY(end-numPeriods+1:end,2,1),'-b',...
    dates(end-numPeriods+1:end),CIFY(end-numPeriods+1:end,2,2),'-b',...
    dates(end-numPeriods+1:end),Y(:,2),':c',...
    dates(end-numPeriods+1:end),Lb(:,2),':m',...
    dates(end-numPeriods+1:end),Ub(:,2),':m',...
    'LineWidth',3);
xlabel('Period')
ylabel('nGNP growth rate')
legend(h([1,2,4:6]),{'Observations','MC forecasts',...
    '95% MC intervals','Theoretical forecasts','95% theoretical intervals'},...
    'Location','Best')
title('Observed and Forecasted nGNP Growth Rates')
```


Observed and Forecasted Changes in the Unemployment Rate
Observations
MC forecasts
95% forecast intervals
Theoretical forecasts
95% theoretical intervals
Change in the unemployment rate
Period
8
6
4
2
0
-2
-4
-6
-8
1950
1952
1954
1956
1958
1960
1962
1964
1966
1968
1970



See Also

estimate | forecast | refine | simulate | ssm

Related Examples

- "Create State-Space Model Containing ARMA State" on page 8-24
- "Estimate Time-Invariant State-Space Model" on page 8-41
- "Simulate States and Observations of Time-Invariant State-Space Model" on page 8-103

More About

- "What Are State-Space Models?" on page 8-3
- "What Is the Kalman Filter?" on page 8-8

Forecast State-Space Model Observations

This example shows how to forecast observations of a known, time-invariant, state-space model.

Suppose that a latent process is an AR(1). Subsequently, the state equation is

$$x_t = 0.5x_{t-1} + u_t,$$

where u_t is Gaussian with mean 0 and standard deviation 1.

Generate a random series of 100 observations from x_t, assuming that the series starts at 1.5.

```
T = 100;
ARMdl = arima('AR',0.5,'Constant',0,'Variance',1);
x0 = 1.5;
rng(1); % For reproducibility
x = simulate(ARMdl,T,'Y0',x0);
```

Suppose further that the latent process is subject to additive measurement error. Subsequently, the observation equation is

$$y_t = x_t + \varepsilon_t,$$

where ε_t is Gaussian with mean 0 and standard deviation 0.75. Together, the latent process and observation equations compose a state-space model.

Use the random latent state process (`x`) and the observation equation to generate observations.

```
y = x + 0.75*randn(T,1);
```

Specify the four coefficient matrices.

```
A = 0.5;
B = 1;
C = 1;
D = 0.75;
```

Specify the state-space model using the coefficient matrices.

```
Mdl = ssm(A,B,C,D)
```

```
Mdl =
```

```
State-space model type: <a href="matlab: doc ssm">ssm</a>

State vector length: 1
Observation vector length: 1
State disturbance vector length: 1
Observation innovation vector length: 1
Sample size supported by model: Unlimited

State variables: x1, x2,...
State disturbances: u1, u2,...
Observation series: y1, y2,...
Observation innovations: e1, e2,...

State equation:
x1(t) = (0.50)x1(t-1) + u1(t)

Observation equation:
y1(t) = x1(t) + (0.75)e1(t)

Initial state distribution:

Initial state means
 x1
  0

Initial state covariance matrix
     x1
 x1  1.33

State types
     x1
 Stationary
```

`Mdl` is an `ssm` model. Verify that the model is correctly specified using the display in the Command Window. The software infers that the state process is stationary. Subsequently, the software sets the initial state mean and covariance to the mean and variance of the stationary distribution of an AR(1) model.

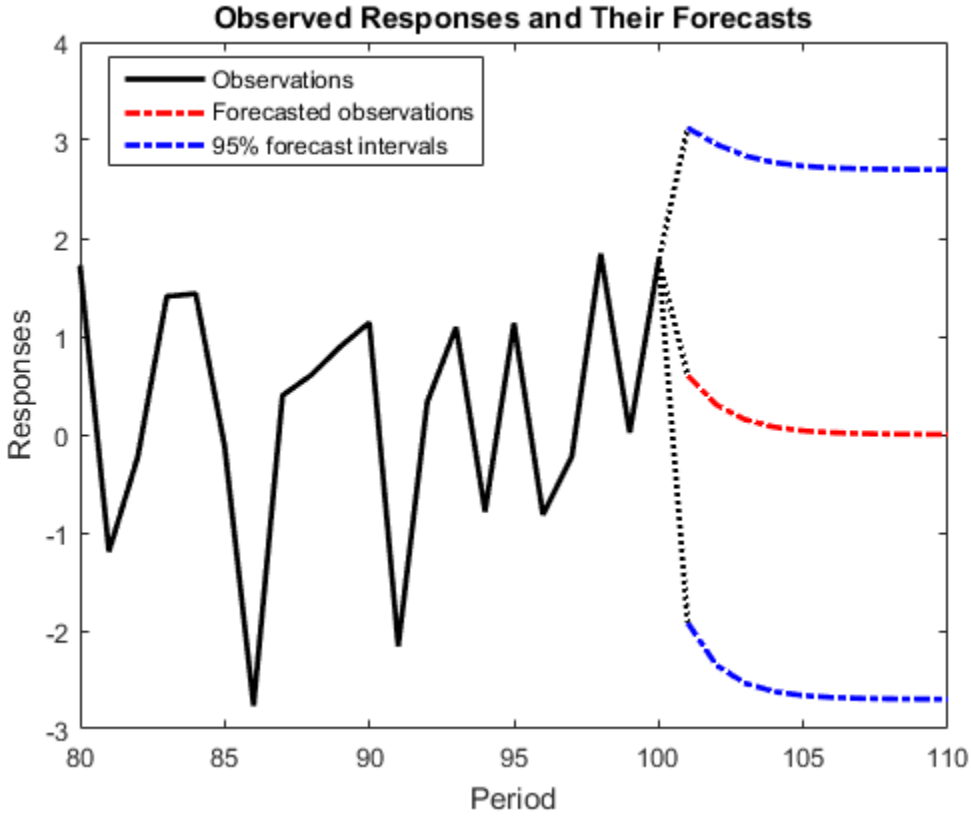
Forecast the observations 10 periods into the future, and estimate their variances.

```
numPeriods = 10;
```

```
[ForecastedY,YMSE] = forecast(Mdl,numPeriods,y);
```

Plot the forecasts with the in-sample responses, and 95% Wald-type forecast intervals.

```
ForecastIntervals(:,1) = ForecastedY - 1.96*sqrt(YMSE);
ForecastIntervals(:,2) = ForecastedY + 1.96*sqrt(YMSE);

figure
plot(T-20:T,y(T-20:T),'-k',T+1:T+numPeriods,ForecastedY,'-.r',...
    T+1:T+numPeriods,ForecastIntervals,'-.b',...
    T:T+1,[y(end)*ones(3,1),[ForecastedY(1);ForecastIntervals(1,:)']],':k',...
    'LineWidth',2)
hold on
title({'Observed Responses and Their Forecasts'})
xlabel('Period')
ylabel('Responses')
legend({'Observations','Forecasted observations','95% forecast intervals'},...
    'Location','Best')
hold off
```



See Also

forecast | ssm

Related Examples

- "Explicitly Create State-Space Model Containing Known Parameter Values" on page 8-17
- "Forecast Time-Varying State-Space Model" on page 8-143
- "Forecast Observations of State-Space Model Containing Regression Component" on page 8-138
- "Forecast State-Space Model Using Monte-Carlo Methods" on page 8-125

- "Forecast State-Space Model Containing Regime Change in the Forecast Horizon" on page 8-149

More About

- "What Are State-Space Models?" on page 8-3
- "What Is the Kalman Filter?" on page 8-8

Forecast Observations of State-Space Model Containing Regression Component

This example shows how to estimate a regression model containing a regression component, and then forecast observations from the fitted model.

Suppose that the linear relationship between the change in the unemployment rate and the nominal gross national product (nGNP) growth rate is of interest. Suppose further that the first difference of the unemployment rate is an ARMA(1,1) series. Symbolically, and in state-space form, the model is

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix} = \begin{bmatrix} \phi & \theta \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix} u_{1,t}$$
$$y_t - \beta Z_t = x_{1,t} + \sigma \varepsilon_t,$$

where:

- $x_{1,t}$ is the change in the unemployment rate at time t.
- $x_{2,t}$ is a dummy state for the MA(1) effect.
- $y_{1,t}$ is the observed change in the unemployment rate being deflated by the growth rate of nGNP (Z_t).
- $u_{1,t}$ is the Gaussian series of state disturbances having mean 0 and standard deviation 1.
- ε_t is the Gaussian series of observation innovations having mean 0 and standard deviation σ.

Load the Nelson-Plosser data set, which contains the unemployment rate and nGNP series, among other things.

```
load Data_NelsonPlosser
```

Preprocess the data by taking the natural logarithm of the nGNP series, and the first difference of each series. Also, remove the starting `NaN` values from each series.

```
isNaN = any(ismissing(DataTable),2);       % Flag periods containing NaNs
gnpn = DataTable.GNPN(~isNaN);
u = DataTable.UR(~isNaN);
```

```
T = size(gnpn,1);                           % Sample size
Z = [ones(T-1,1) diff(log(gnpn))];
y = diff(u);
```

Though this example removes missing values, the software can accommodate series containing missing values in the Kalman filter framework.

To determine how well the model forecasts observations, remove the last 10 observations for comparison.

```
numPeriods = 10;                    % Forecast horizon
isY = y(1:end-numPeriods);          % In-sample observations
oosY = y(end-numPeriods+1:end);     % Out-of-sample observations
ISZ = Z(1:end-numPeriods,:);        % In-sample predictors
OOSZ = Z(end-numPeriods+1:end,:);   % Out-of-sample predictors
```

Specify the coefficient matrices.

```
A = [NaN NaN; 0 0];
B = [1; 1];
C = [1 0];
D = NaN;
```

Specify the state-space model using `ssm`.

```
Mdl = ssm(A,B,C,D);
```

Estimate the model parameters. Specify the regression component and its initial value for optimization using the `'Predictors'` and `'Beta0'` name-value pair arguments, respectively. Restrict the estimate of σ to all positive, real numbers. For numerical stability, specify the Hessian when the software computes the parameter covariance matrix, using the `'CovMethod'` name-value pair argument.

```
params0 = [0.3 0.2 0.1]; % Chosen arbitrarily
[EstMdl,estParams] = estimate(Mdl,isY,params0,'Predictors',ISZ,...
    'Beta0',[0.1 0.2],'lb',[-Inf,-Inf,0,-Inf,-Inf],'CovMethod','hessian');
```

```
Method: Maximum likelihood (fmincon)
Sample size: 51
Logarithmic  likelihood:     -87.2409
Akaike   info criterion:      184.482
Bayesian info criterion:      194.141
```

```
           |      Coeff       Std Err    t Stat     Prob
-----------------------------------------------------------
 c(1)      |   -0.31780       0.19429    -1.63572  0.10190
 c(2)      |    1.21242       0.48882     2.48031  0.01313
 c(3)      |    0.45583       0.63930     0.71301  0.47584
 y <- z(1) |    1.32407       0.26313     5.03201   0
 y <- z(2) |  -24.48733       1.90115   -12.88024   0
           |
           |    Final State   Std Dev     t Stat    Prob
 x(1)      |   -0.38117       0.42842    -0.88971  0.37363
 x(2)      |    0.23402       0.66222     0.35339  0.72380
```

`EstMdl` is an `ssm` model, and you can access its properties using dot notation.

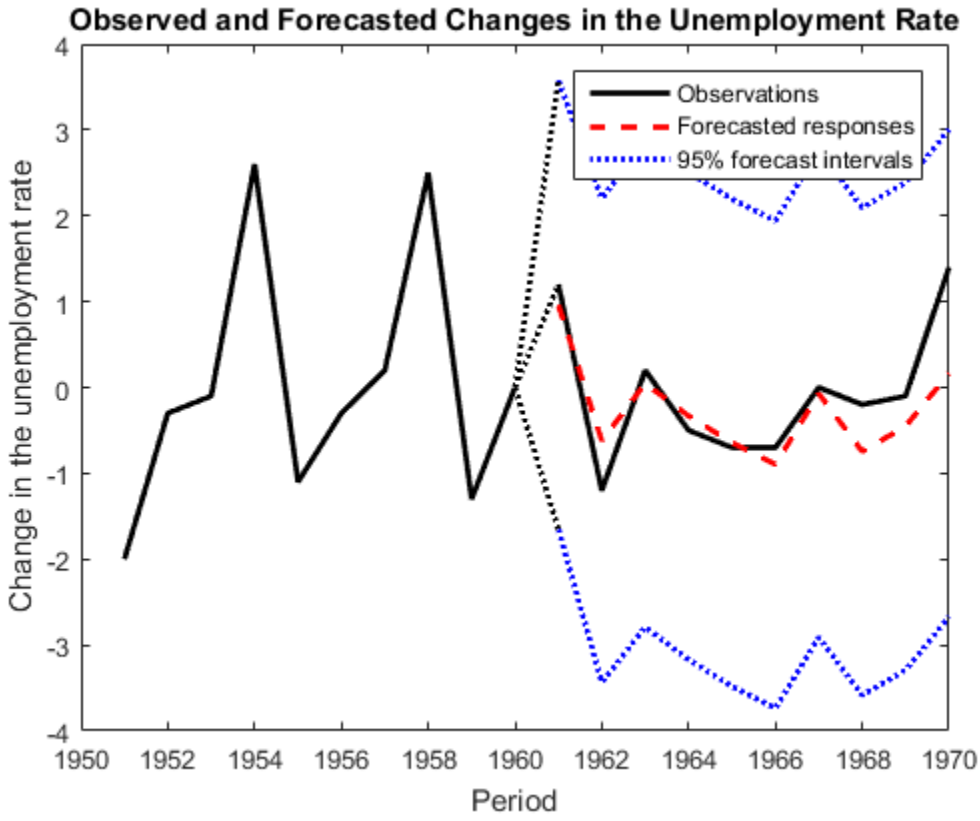
Forecast observations over the forecast horizon. `EstMdl` does not store the data set, so you must pass it in appropriate name-value pair arguments.

```
[fY,yMSE] = forecast(EstMdl,numPeriods,isY,'PredictorsO',ISZ,...
    'PredictorsF',OOSZ,'Beta',estParams(end-1:end));
```

`fY` is a 10-by-1 vector containing the forecasted observations, and `yMSE` is a 10-by-1 vector containing the variances of the forecasted observations.

Obtain 95% Wald-type forecast intervals. Plot the forecasted observations with their true values and the forecast intervals.

```
ForecastIntervals(:,1) = fY - 1.96*sqrt(yMSE);
ForecastIntervals(:,2) = fY + 1.96*sqrt(yMSE);

figure
h = plot(dates(end-numPeriods-9:end-numPeriods),isY(end-9:end),'-k',...
    dates(end-numPeriods+1:end),oosY,'-k',...
    dates(end-numPeriods+1:end),fY,'--r',...
    dates(end-numPeriods+1:end),ForecastIntervals,':b',...
    dates(end-numPeriods:end-numPeriods+1),...
    [isY(end)*ones(3,1),[oosY(1);ForecastIntervals(1,:)']],':k',...
    'LineWidth',2);
xlabel('Period')
ylabel('Change in the unemployment rate')
legend(h([1,3,4]),{'Observations','Forecasted responses',...
    '95% forecast intervals'})
title('Observed and Forecasted Changes in the Unemployment Rate')
```



This model seems to forecast the changes in the unemployment rate well.

See Also

estimate | forecast | refine | ssm

Related Examples

- "Create State-Space Model Containing ARMA State" on page 8-24
- "Estimate State-Space Model Containing Regression Component" on page 8-55
- "Forecast State-Space Model Observations" on page 8-133
- "Forecast State-Space Model Using Monte-Carlo Methods" on page 8-125

- “Forecast State-Space Model Containing Regime Change in the Forecast Horizon” on page 8-149

More About

- “What Are State-Space Models?” on page 8-3
- “What Is the Kalman Filter?” on page 8-8

Forecast Time-Varying State-Space Model

This example shows how to generate data from a known model, fit a state-space model to the data, and then forecast states and observations states from the fitted model.

Suppose that a latent process comprises an AR(2) and an MA(1) model. There are 50 periods, and the MA(1) process drops out of the model for the final 25 periods. Subsequently, the state equation for the first 25 periods is

$$x_{1,t} = 0.7x_{1,t-1} - 0.2x_{1,t-2} + u_{1,t}$$
$$x_{2,t} = u_{2,t} + 0.6u_{2,t-1},$$

and for the last 25 periods, it is

$$x_{1,t} = 0.7x_{1,t-1} - 0.2x_{1,t-2} + u_{1,t},$$

where $u_{1,t}$ and $u_{2,t}$ are Gaussian with mean 0 and standard deviation 1.

Assuming that the series starts at 1.5 and 1, respectively, generate a random series of 50 observations from $x_{1,t}$ and $x_{2,t}$.

```
T = 50;
ARMdl = arima('AR',{0.7,-0.2},'Constant',0,'Variance',1);
MAMdl = arima('MA',0.6,'Constant',0,'Variance',1);
x0 = [1.5 1; 1.5 1];
rng(1);
x = [simulate(ARMdl,T,'Y0',x0(:,1)),...
    [simulate(MAMdl,T/2,'Y0',x0(:,2));nan(T/2,1)]];
```

The last 25 values for the simulated MA(1) data are `NaN` values.

Suppose further that the latent processes are measured using

$$y_t = 2\left(x_{1,t} + x_{2,t}\right) + \varepsilon_t,$$

for the first 25 periods, and

$$y_t = 2x_{1,t} + \varepsilon_t$$

for the last 25 periods, where ε_t is Gaussian with mean 0 and standard deviation 1.

Use the random latent state process (`x`) and the observation equation to generate observations.

```
y = 2*nansum(x')'+randn(T,1);
```

Together, the latent process and observation equations compose a state-space model. Supposing that the coefficients are unknown parameters, the state-space model is

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \\ x_{3,t} \\ x_{4,t} \end{bmatrix} = \begin{bmatrix} \phi_1 & \phi_2 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \theta_1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \\ x_{3,t-1} \\ x_{4,t-1} \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} u_{1,t} \\ u_{2,t} \end{bmatrix}$$
$$y_t = a(x_{1,t} + x_{3,t}) + \varepsilon_t$$

for the first 25 periods,

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix} = \begin{bmatrix} \phi_1 & \phi_2 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \\ x_{3,t-1} \\ x_{4,t-1} \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} u_{1,t}$$
$$y_t = bx_{1,t} + \varepsilon_t$$

for period 26, and

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix} = \begin{bmatrix} \phi_1 & \phi_2 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} u_{1,t}$$
$$y_t = bx_{1,t} + \varepsilon_t$$

for the last 24 periods.

Write a function that specifies how the parameters in `params` map to the state-space model matrices, the initial state values, and the type of state.

```
% Copyright 2015 The MathWorks, Inc.

function [A,B,C,D,Mean0,Cov0,StateType] = AR2MAParamMap(params,T)
%AR2MAParamMap Time-variant state-space model parameter mapping function
%
% This function maps the vector params to the state-space matrices (A, B,
% C, and D), the initial state value and the initial state variance (Mean0
% and Cov0), and the type of state (StateType). From periods 1 to T/2, the
% state model is an AR(2) and an MA(1) model, and the observation model is
% the sum of the two states. From periods T/2 + 1 to T, the state model is
% just the AR(2) model.
    A1 = {[params(1) params(2) 0 0; 1 0 0 0; 0 0 0 params(3); 0 0 0 0]};
    B1 = {[1 0; 0 0; 0 1; 0 1]};
    C1 = {params(4)*[1 0 1 0]};
    Mean0 = ones(4,1);
    Cov0 = 10*eye(4);
    StateType = [0 0 0 0];
    A2 = {[params(1) params(2) 0 0; 1 0 0 0]};
    B2 = {[1; 0]};
    A3 = {[params(1) params(2); 1 0]};
    B3 = {[1; 0]};
    C3 = {params(5)*[1 0]};
    A = [repmat(A1,T/2,1);A2;repmat(A3,(T-2)/2,1)];
    B = [repmat(B1,T/2,1);B2;repmat(B3,(T-2)/2,1)];
    C = [repmat(C1,T/2,1);repmat(C3,T/2,1)];
    D = 1;
end
```

Save this code as a file named `AR2MAParamMap` on your MATLAB® path.

Create the state-space model by passing the function `AR2MAParamMap` as a function handle to `ssm`.

```
Mdl = ssm(@(params)AR2MAParamMap(params,T));
```

`ssm` implicitly creates the state-space model. Usually, you cannot verify an implicitly defined state-space model.

Pass the observed responses (`y`) to `estimate` to estimate the parameters. Specify an arbitrary set of positive initial values for the unknown parameters.

```
params0 = 0.1*ones(5,1);
EstMdl = estimate(Mdl,y,params0);
```

```
Method: Maximum likelihood (fminunc)
Sample size: 50
Logarithmic  likelihood:     -114.957
Akaike   info criterion:      239.913
Bayesian info criterion:      249.473
      |     Coeff       Std Err   t Stat     Prob
-------------------------------------------------
 c(1) |  0.47870       0.26634    1.79733  0.07229
 c(2) |  0.00809       0.27179    0.02975  0.97626
 c(3) |  0.55735       0.80958    0.68844  0.49118
 c(4) |  1.62679       0.41622    3.90848  0.00009
 c(5) |  1.90021       0.49563    3.83391  0.00013
      |
      |   Final State   Std Dev    t Stat    Prob
 x(1) | -0.81229       0.46815   -1.73511  0.08272
 x(2) | -0.31449       0.45918   -0.68490  0.49341
```

`EstMdl` is an `ssm` model containing the estimated coefficients. Likelihood surfaces of state-space models might contain local maxima. Therefore, it is good practice to try several initial parameter values, or consider using `refine`.

Forecast observations and states five periods into the future. Also, obtain measures of variability for the forecasts.

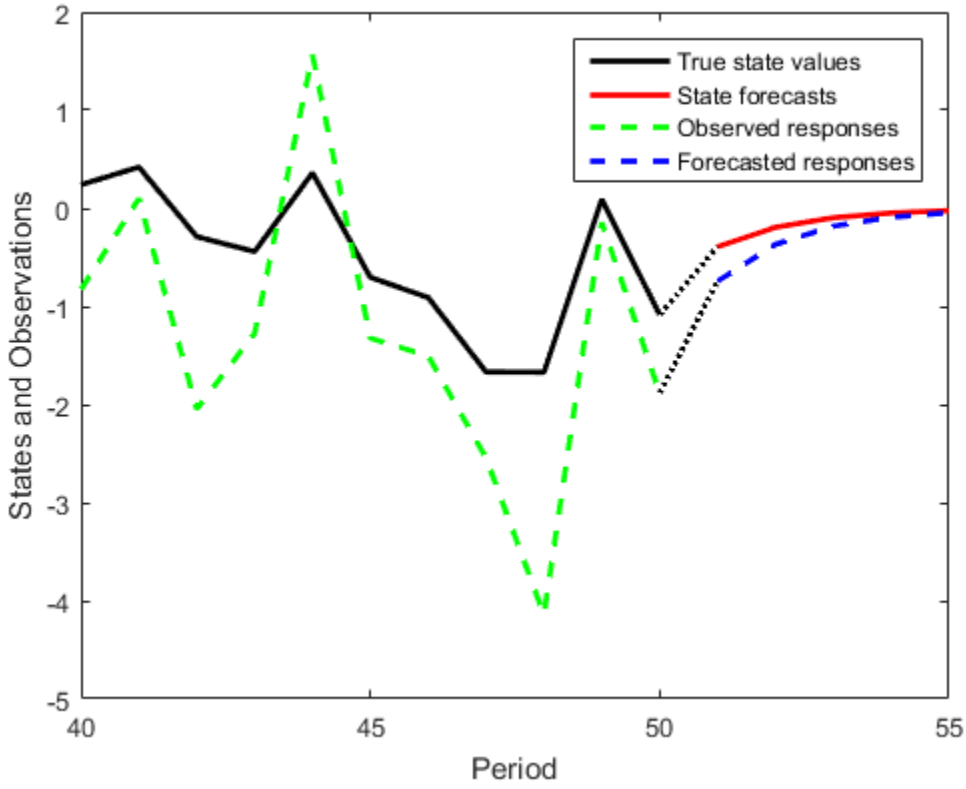
```
numPeriods = 5;
[fY,yMSE,FX,XMSE]= forecast(EstMdl,numPeriods,y);
```

`forecast` uses `EstMdl.A{end}`, ..., `EstMdl.D{end}` to forecast the state-space model. `fY` and `yMSE` are `numPeriods`-by-1 numeric vectors of forecasted observations and variances of the forecasted observations, respectively. `FX` and `XMSE` are `numPeriods`-by-2 matrices of state forecasts and variances of the state forecasts. The columns indicate the state, and the rows indicate the period. For all output arguments, the last row corresponds to the latest forecast.

Plot the observations, true states, forecasted observations, and state forecasts.

```
figure;
plot(T-10:T,x(T-10:T,1),'-k',T+1:T+numPeriods,FX(:,1),'-r',...
    T-10:T,y(T-10:T),'--g',T+1:T+numPeriods,fY,'--b',...
    T:T+1,[y(T),fY(1);x(T,1),FX(1,1)]',':k','LineWidth',2);
xlabel('Period')
ylabel('States and Observations')
legend({'True state values','State forecasts',...
```

```
'Observed responses','Forecasted responses'});
```



See Also

estimate | forecast | refine | ssm

Related Examples

- “Implicitly Create Time-Varying State-Space Model” on page 8-32
- “Estimate Time-Varying State-Space Model” on page 8-45
- “Forecast State-Space Model Using Monte-Carlo Methods” on page 8-125
- “Forecast State-Space Model Containing Regime Change in the Forecast Horizon” on page 8-149

More About

- "What Are State-Space Models?" on page 8-3
- "What Is the Kalman Filter?" on page 8-8

Forecast State-Space Model Containing Regime Change in the Forecast Horizon

This example shows how to forecast a time-varying, state-space model, in which there is a regime change in the forecast horizon.

Suppose that you observed a multivariate process for 75 periods, and you want to forecast the process 25 periods into the future. Also, suppose that you can specify the process as a state-space model. For periods 1 through 50, the state-space model has one state: a stationary AR(2) model with a constant term. At period 51, the state-space model includes a random walk. The states are observed unbiasedly, but with additive measurement error. Symbolically, the model is

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \\ x_{3,t} \\ x_{4,t} \end{bmatrix} = \begin{bmatrix} 0.5 & -0.2 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \\ x_{3,t-1} \\ x_{4,t-1} \end{bmatrix} + \begin{bmatrix} 0.1 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0.5 \end{bmatrix} \begin{bmatrix} u_{1,t} \\ u_{2,t} \end{bmatrix}.$$

$$y_t = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{1,t} \\ x_{2,t} \\ x_{3,t} \\ x_{4,t} \end{bmatrix} + \begin{bmatrix} 0.3 & 0 \\ 0 & 0.2 \end{bmatrix} \begin{bmatrix} \varepsilon_{1,t} \\ \varepsilon_{2,t} \end{bmatrix}$$

For periods 1 through 50, the random walk process is not in the model.

Specify the in-sample, coefficient matrices.

```
A1 = {[0.5 0.2 1; 1 0 0; 0 0 1]};                % A for periods 1 - 50
A2 = {[0.5 0.2 1; 1 0 0; 0 0 1; 0 0 0]};         % A for period 51
A3 = {[0.5 0.2 1 0; 1 0 0 0; 0 0 1 0; 0 0 0 1]}; % A for periods 51 - 75
A = [repmat(A1,50,1); A2; repmat(A3,24,1)];

B1 = {[0.1; 0; 0]};              % B for periods 1 - 50
B3 = {[0.1 0; 0 0; 0 0; 0 0.5]}; % B for periods 51 - 75
B = [repmat(B1,50,1); repmat(B3,25,1)];

C1 = {[1 0 0]};            % C for periods 1 - 50
C3 = {[1 0 0 0; 0 0 0 1]}; % C for periods 51 - 75
C = [repmat(C1,50,1); repmat(C3,25,1)];
```

```
D1 = {0.3};            % D for periods 1 - 50
D3 = {[0.3 0; 0 0.2]}; % D for periods 51 - 75
D = [repmat(D1,50,1); repmat(D3,25,1)];
```

Specify the state space model, and the initial state means and covariance matrix. It is best practice to specify the types of each state using the `'StateType'` name-value pair argument. Only specify the initial state parameters for the three states that start the state-space model.

```
Mean0 = [1/(1-0.5-0.2); 1/(1-0.5-0.2); 1];
Cov0 = [0.02 0.01 0; 0.01 0.02 0; 0 0 0];
StateType = [0; 0; 1];
Mdl = ssm(A,B,C,D,'Mean0',Mean0,'Cov0',Cov0,'StateType',StateType)
```

```
Mdl =


State-space model type: <a href="matlab: doc ssm">ssm</a>

State vector length: Time-varying
Observation vector length: Time-varying
State disturbance vector length: Time-varying
Observation innovation vector length: Time-varying
Sample size supported by model: 75

State variables: x1, x2,...
State disturbances: u1, u2,...
Observation series: y1, y2,...
Observation innovations: e1, e2,...

State equations of period 1, 2, 3,..., 50:
x1(t) = (0.50)x1(t-1) + (0.20)x2(t-1) + x3(t-1) + (0.10)u1(t)
x2(t) = x1(t-1)
x3(t) = x3(t-1)

State equations of period 51:
x1(t) = (0.50)x1(t-1) + (0.20)x2(t-1) + x3(t-1) + (0.10)u1(t)
x2(t) = x1(t-1)
x3(t) = x3(t-1)
x4(t) = (0.50)u2(t)

State equations of period 52, 53, 54,..., 75:
x1(t) = (0.50)x1(t-1) + (0.20)x2(t-1) + x3(t-1) + (0.10)u1(t)
x2(t) = x1(t-1)
```

```
x3(t) = x3(t-1)
x4(t) = x4(t-1) + (0.50)u2(t)


Observation equation of period 1, 2, 3,..., 50:
y1(t) = x1(t) + (0.30)e1(t)

Observation equations of period 51, 52, 53,..., 75:
y1(t) = x1(t) + (0.30)e1(t)
y2(t) = x4(t) + (0.20)e2(t)


Initial state distribution:

Initial state means
  x1    x2   x3
 3.33  3.33   1

Initial state covariance matrix
     x1    x2    x3
 x1  0.02  0.01   0
 x2  0.01  0.02   0
 x3   0     0     0

State types
     x1          x2         x3
 Stationary  Stationary  Constant
```

`Mdl` is a time-varying, `ssm` model without unknown parameters. The software sets initial state means and covariane values based on the type of state.

Simulate 75 observations from `Mdl`.

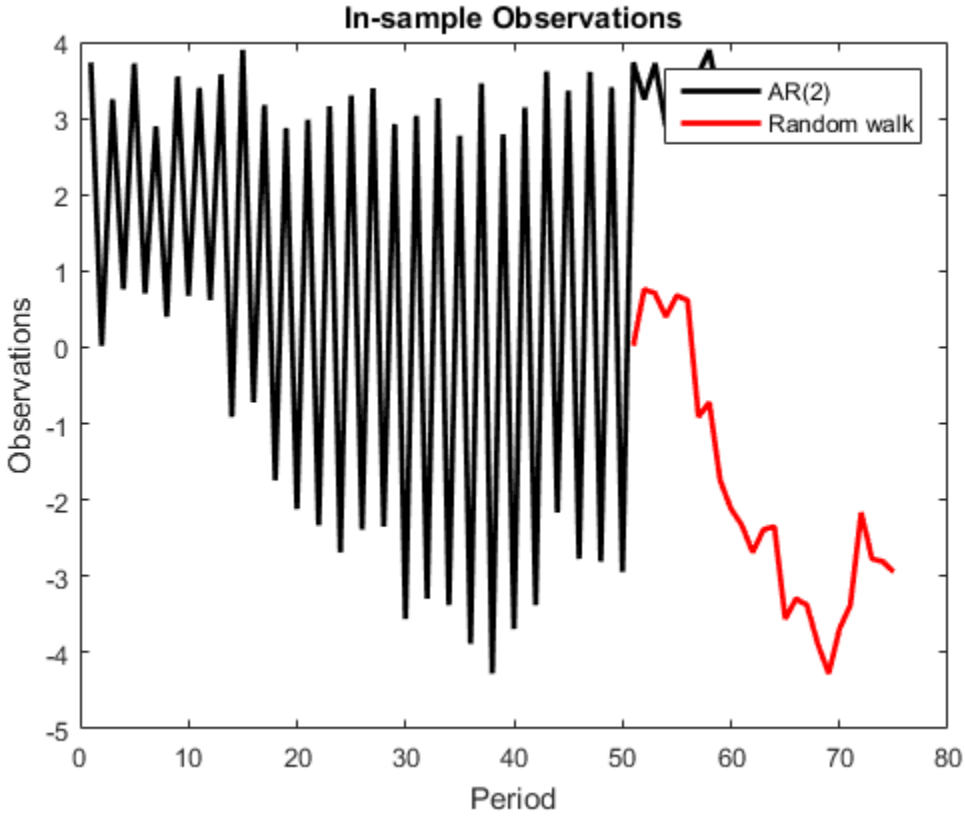
```
rng(1); % For reproducibility
Y = simulate(Mdl,75);
```

`y` is a 75-by-1 cell vector. Cells 1 through 50 contain scalars, and cells 51 through 75 contain 2-by-1 numeric vectors. Cell *j* corresponds to the observations of period *j*, specified by the observation model.

Plot the simulated responses.

```
y1 = cell2mat(Y(51:75));              % Observations for periods 1 - 50
```

```
d1 = cell2mat(Y(51:75));
y2 = [d1(((1:25)*2)-1) d1((1:25)*2)]; % Observations for periods 51 - 75

figure
plot(1:75,[y1;y2(:,1)],'-k',1:75,[nan(50,1);y2(:,2)],'-r','LineWidth',2')
title('In-sample Observations')
ylabel('Observations')
xlabel('Period')
legend({'AR(2)','Random walk'})
```



Suppose that the random walk process drops out of the state space in the 20th period of the forecast horizon.

Specify the coefficient matrices for the forecast period.

```
A4 = {[0.5 0.2 1 0; 1 0 0 0; 0 0 1 0; 0 0 0 1]};  % A for periods 76 - 95
A5 = {[0.5 0.2 1 0; 1 0 0 0; 0 0 1 0]};           % A for period 96
A6 = {[0.5 0.2 1; 1 0 0; 0 0 1]};                 % A for periods 97 - 100
fhA = [repmat(A4,20,1); A5; repmat(A6,4,1)];

B4 = {[0.1 0; 0 0; 0 0; 0 0.5]}; % B for periods 76 - 95
B6 = {[0.1; 0; 0]};              % B for periods 96 - 100
fhB = [repmat(B4,20,1); repmat(B6,5,1)];

C4 = {[1 0 0 0; 0 0 0 1]}; % C for periods 76 - 95
C6 = {[1 0 0]};            % C for periods 96 - 100
fhC = [repmat(C4,20,1); repmat(C6,5,1)];

D4 = {[0.3 0; 0 0.2]}; % D for periods 76 - 95
D6 = {0.3};            % D for periods 96 - 100
fhD = [repmat(D4,20,1); repmat(D6,5,1)];
```

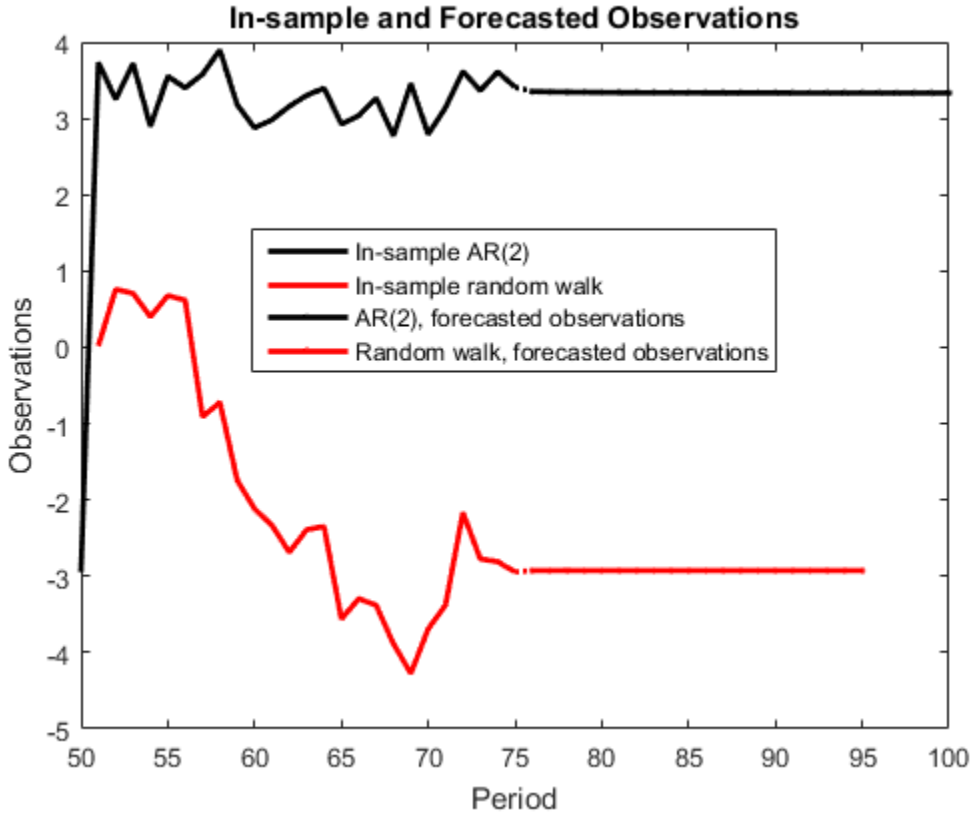
Forecast observations over the forecast horizon.

```
FY = forecast(Mdl,25,Y,'A',fhA,'B',fhB,'C',fhC,'D',fhD);
```

FY is a 25-by-1 cell vector. Cells 1 through 20 contain 2-by-1 numeric vectors, and cells 51 through 75 contain scalars. Cell *j* corresponds to the observations of period *j*, specified by the forecast-horizon, observation model.

Plot the forecasts with the in-sample observations.

```
d2 = cell2mat(FY(1:20));
FY1 = [d2(((1:20)*2)-1) d2((1:20)*2)]; % Forecasts for periods 76 - 95
FY2 = cell2mat(FY(21:25));             % Forecasts for periods 96 - 100

figure
plot(1:75,[y1;y2(:,1)],'-k',1:75,[nan(50,1);y2(:,2)],'-r',...
    76:100,[FY1(:,1); FY2],'.-k',76:100,[FY1(:,2); nan(5,1)],'.-r',...
    75:76,[y2(end,1) FY1(1,1)],':k',75:76,[y2(end,2) FY1(1,2)],':r',...
    'LineWidth',2')
title('In-sample and Forecasted Observations')
ylabel('Observations')
xlabel('Period')
xlim([50,100])
legend({'In-sample AR(2)','In-sample random walk',...
    'AR(2), forecasted observations',...
    'Random walk, forecasted observations'},'Location','Best')%% Title
% This example shows
```



See Also

estimate | forecast | refine | ssm

Related Examples

- "Implicitly Create Time-Varying State-Space Model" on page 8-32
- "Estimate Time-Varying State-Space Model" on page 8-45
- "Forecast State-Space Model Observations" on page 8-133
- "Forecast Time-Varying State-Space Model" on page 8-143
- "Forecast Observations of State-Space Model Containing Regression Component" on page 8-138

- "Forecast State-Space Model Using Monte-Carlo Methods" on page 8-125

More About

- "What Are State-Space Models?" on page 8-3
- "What Is the Kalman Filter?" on page 8-8

Forecast Time-Varying Diffuse State-Space Model

This example shows how to generate data from a known model, fit a diffuse state-space model to the data, and then forecast states and observations states from the fitted model.

Suppose that a latent process comprises an AR(2) and an MA(1) model. There are 50 periods, and the MA(1) process drops out of the model for the final 25 periods. Consequently, the state equation for the first 25 periods is

$$x_{1,t} = 0.7x_{1,t-1} - 0.2x_{1,t-2} + u_{1,t}$$
$$x_{2,t} = u_{2,t} + 0.6u_{2,t-1},$$

and for the last 25 periods, it is

$$x_{1,t} = 0.7x_{1,t-1} - 0.2x_{1,t-2} + u_{1,t},$$

where $u_{1,t}$ and $u_{2,t}$ are Gaussian with mean 0 and standard deviation 1.

Assuming that the series starts at 1.5 and 1, respectively, generate a random series of 50 observations from $x_{1,t}$ and $x_{2,t}$.

```
T = 50;
ARMdl = arima('AR',{0.7,-0.2},'Constant',0,'Variance',1);
MAMdl = arima('MA',0.6,'Constant',0,'Variance',1);
x0 = [1.5 1; 1.5 1];
rng(1);
x = [simulate(ARMdl,T,'Y0',x0(:,1)),...
    [simulate(MAMdl,T/2,'Y0',x0(:,2));nan(T/2,1)]];
```

The last 25 values for the simulated MA(1) data are `NaN` values.

The latent processes are measured using

$$y_t = 2\left(x_{1,t} + x_{2,t}\right) + \varepsilon_t$$

for the first 25 periods, and

$$y_t = 2x_{1,t} + \varepsilon_t$$

for the last 25 periods, where ε_t is Gaussian with mean 0 and standard deviation 1.

Use the random latent state process (`x`) and the observation equation to generate observations.

```
y = 2*nansum(x')'+randn(T,1);
```

Together, the latent process and observation equations make up a state-space model. If the coefficients are unknown parameters, the state-space model is

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \\ x_{3,t} \\ x_{4,t} \end{bmatrix} = \begin{bmatrix} \phi_1 & \phi_2 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \theta_1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \\ x_{3,t-1} \\ x_{4,t-1} \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} u_{1,t} \\ u_{2,t} \end{bmatrix}$$
$$y_t = a(x_{1,t} + x_{3,t}) + \varepsilon_t$$

for the first 25 periods,

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix} = \begin{bmatrix} \phi_1 & \phi_2 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \\ x_{3,t-1} \\ x_{4,t-1} \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} u_{1,t}$$
$$y_t = bx_{1,t} + \varepsilon_t$$

for period 26, and

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix} = \begin{bmatrix} \phi_1 & \phi_2 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} u_{1,t}$$
$$y_t = bx_{1,t} + \varepsilon_t$$

for the last 24 periods.

Write a function that specifies how the parameters in `params` map to the state-space model matrices, the initial state values, and the type of state.

```
% Copyright 2015 The MathWorks, Inc.

function [A,B,C,D,Mean0,Cov0,StateType] = diffuseAR2MAParamMap(params,T)
%diffuseAR2MAParamMap Time-variant diffuse state-space model parameter
%mapping function
%
% This function maps the vector params to the state-space matrices (A, B,
% C, and D) and the type of state (StateType). From periods 1 to T/2, the
% state model is an AR(2) and an MA(1) model, and the observation model is
% the sum of the two states. From periods T/2 + 1 to T, the state model is
% just the AR(2) model.  The AR(2) model is diffuse.
    A1 = {[params(1) params(2) 0 0; 1 0 0 0; 0 0 0 params(3); 0 0 0 0]};
    B1 = {[1 0; 0 0; 0 1; 0 1]};
    C1 = {params(4)*[1 0 1 0]};
    Mean0 = [];
    Cov0 = [];
    StateType = [2 2 0 0];
    A2 = {[params(1) params(2) 0 0; 1 0 0 0]};
    B2 = {[1; 0]};
    A3 = {[params(1) params(2); 1 0]};
    B3 = {[1; 0]};
    C3 = {params(5)*[1 0]};
    A = [repmat(A1,T/2,1);A2;repmat(A3,(T-2)/2,1)];
    B = [repmat(B1,T/2,1);B2;repmat(B3,(T-2)/2,1)];
    C = [repmat(C1,T/2,1);repmat(C3,T/2,1)];
    D = 1;
end
```

Save this code as a file named `diffuseAR2MAParamMap` on your MATLAB® path.

Create the diffuse state-space model by passing the function `diffuseAR2MAParamMap` as a function handle to `dssm`.

```
Mdl = dssm(@(params)diffuseAR2MAParamMap(params,T));
```

`dssm` implicitly creates the diffuse state-space model. Usually, you cannot verify diffuse state-space models that are implicitly created.

To estimate the parameters, pass the observed responses (`y`) to `estimate`. Specify an arbitrary set of positive initial values for the unknown parameters.

```
params0 = 0.1*ones(5,1);
EstMdl = estimate(Mdl,y,params0);
```

```
Method: Maximum likelihood (fminunc)
Effective Sample size:             48
Logarithmic  likelihood:     -110.313
Akaike   info criterion:      230.626
Bayesian info criterion:      240.186
      |     Coeff       Std Err   t Stat     Prob
-------------------------------------------------------
 c(1) |  0.44041       0.27687    1.59069  0.11168
 c(2) |  0.03949       0.29585    0.13349  0.89380
 c(3) |  0.78364       1.49223    0.52515  0.59948
 c(4) |  1.64260       0.66737    2.46133  0.01384
 c(5) |  1.90409       0.49374    3.85648  0.00012
      |
      |   Final State   Std Dev    t Stat    Prob
 x(1) | -0.81932       0.46706   -1.75420  0.07940
 x(2) | -0.29909       0.45939   -0.65107  0.51500
```

`EstMdl` is a `dssm` model containing the estimated coefficients. Likelihood surfaces of state-space models might contain local maxima. Therefore, try several initial parameter values, or consider using `refine`.

Forecast observations and states five periods into the future. Also, obtain measures of variability for the forecasts.

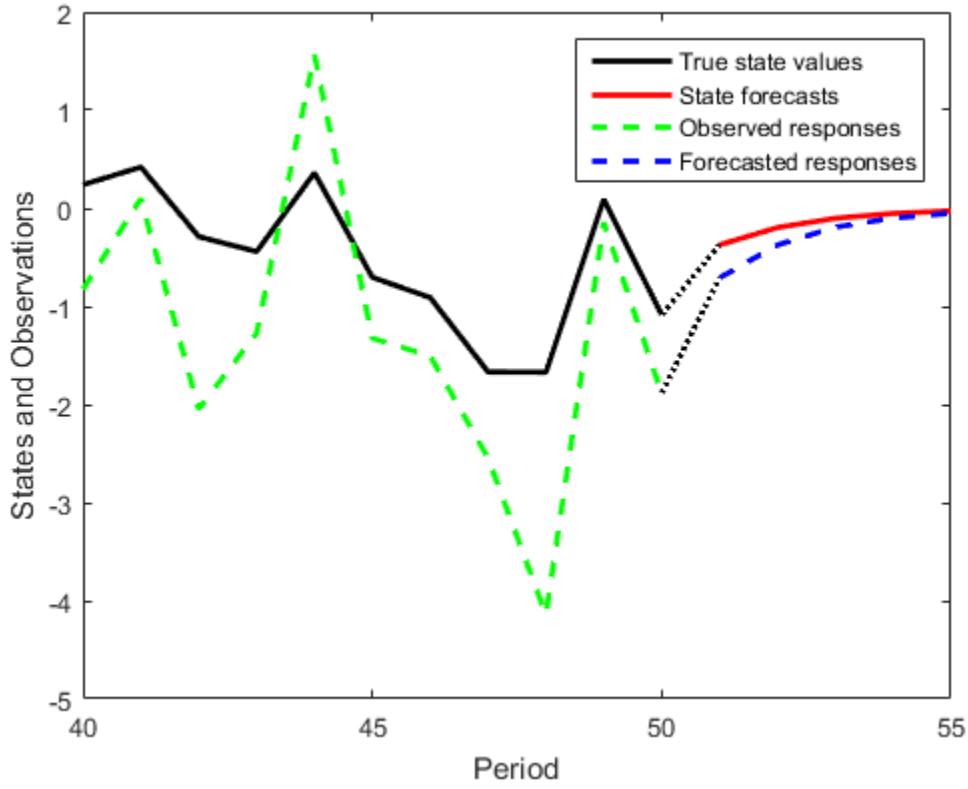
```
numPeriods = 5;
[fY,yMSE,FX,XMSE] = forecast(EstMdl,numPeriods,y);
```

`forecast` uses `EstMdl.A{end}`, ..., `EstMdl.D{end}` to forecast the diffuse state-space model. `fY` and `yMSE` are `numPeriods`-by-1 numeric vectors of forecasted observations and variances of the forecasted observations, respectively. `FX` and `XMSE` are `numPeriods`-by-2 matrices of state forecasts and variances of the state forecasts. The columns indicate the state, and the rows indicate the period. For all output arguments, the last row corresponds to the latest forecast.

Plot the observations, true states, forecasted observations, and state forecasts.

```
figure;
plot(T-10:T,x(T-10:T,1),'-k',T+1:T+numPeriods,FX(:,1),'-r',...
    T-10:T,y(T-10:T),'--g',T+1:T+numPeriods,fY,'--b',...
    T:T+1,[y(T),fY(1);x(T,1),FX(1,1)]',':k','LineWidth',2);
xlabel('Period')
ylabel('States and Observations')
legend({'True state values','State forecasts',...
```

```
'Observed responses','Forecasted responses'});
```



See Also

dssm | esitmate | forecast

Related Examples

- “Implicitly Create Time-Varying Diffuse State-Space Model” on page 8-35
- “Implicitly Create Diffuse State-Space Model Containing Regression Component” on page 8-30
- “Estimate Time-Varying Diffuse State-Space Model” on page 8-50

More About

- "What Are State-Space Models?" on page 8-3
- "What Is the Kalman Filter?" on page 8-8

Compare Simulation Smoother to Smoothed States

This example shows how the results of the state-space model simulation smoother (`simsmooth`) compare to the smoothed states (`smooth`).

Suppose that the relationship between the change in the unemployment rate ($x_{1,t}$) and the nominal gross national product (nGNP) growth rate ($x_{3,t}$) can be expressed in the following, state-space model form.

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \\ x_{3,t} \\ x_{4,t} \end{bmatrix} = \begin{bmatrix} \phi_1 & \theta_1 & \gamma_1 & 0 \\ 0 & 0 & 0 & 0 \\ \gamma_2 & 0 & \phi_2 & \theta_2 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \\ x_{3,t-1} \\ x_{4,t-1} \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} u_{1,t} \\ u_{2,t} \end{bmatrix}$$

$$\begin{bmatrix} y_{1,t} \\ y_{2,t} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t} \\ x_{2,t} \\ x_{3,t} \\ x_{4,t} \end{bmatrix} + \begin{bmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{bmatrix} \begin{bmatrix} \varepsilon_{1,t} \\ \varepsilon_{2,t} \end{bmatrix},$$

where:

- $x_{1,t}$ is the change in the unemployment rate at time t.
- $x_{2,t}$ is a dummy state for the MA(1) effect on $x_{1,t}$.
- $x_{3,t}$ is the nGNP growth rate at time t.
- $x_{4,t}$ is a dummy state for the MA(1) effect on $x_{3,t}$.
- $y_{1,t}$ is the observed change in the unemployment rate.
- $y_{2,t}$ is the observed nGNP growth rate.
- $u_{1,t}$ and $u_{2,t}$ are Gaussian series of state disturbances having mean 0 and standard deviation 1.
- $\varepsilon_{1,t}$ is the Gaussian series of observation innovations having mean 0 and standard deviation σ_1.
- $\varepsilon_{2,t}$ is the Gaussian series of observation innovations having mean 0 and standard deviation σ_2.

Load the Nelson-Plosser data set, which contains the unemployment rate and nGNP series, among other things.

```
load Data_NelsonPlosser
```

Preprocess the data by taking the natural logarithm of the nGNP series, and the first difference of each. Also, remove the starting `NaN` values from each series.

```
isNaN = any(ismissing(DataTable),2);       % Flag periods containing NaNs
gnpn = DataTable.GNPN(~isNaN);
u = DataTable.UR(~isNaN);
T = size(gnpn,1);                          % Sample size
y = zeros(T-1,2);                          % Preallocate
y(:,1) = diff(u);
y(:,2) = diff(log(gnpn));
```

This example proceeds using series without `NaN` values. However, using the Kalman filter framework, the software can accommodate series containing missing values.

Specify the coefficient matrices.

```
A = [NaN NaN NaN 0; 0 0 0 0; NaN 0 NaN NaN; 0 0 0 0];
B = [1 0;1 0 ; 0 1; 0 1];
C = [1 0 0 0; 0 0 1 0];
D = [NaN 0; 0 NaN];
```

Specify the state-space model using `ssm`. Verify that the model specification is consistent with the state-space model.

```
Mdl = ssm(A,B,C,D)
```

```
Mdl =


State-space model type: <a href="matlab: doc ssm">ssm</a>

State vector length: 4
Observation vector length: 2
State disturbance vector length: 2
Observation innovation vector length: 2
Sample size supported by model: Unlimited
Unknown parameters for estimation: 8

State variables: x1, x2,...
State disturbances: u1, u2,...
Observation series: y1, y2,...
Observation innovations: e1, e2,...
```

```
Unknown parameters: c1, c2,...

State equations:
x1(t) = (c1)x1(t-1) + (c3)x2(t-1) + (c4)x3(t-1) + u1(t)
x2(t) = u1(t)
x3(t) = (c2)x1(t-1) + (c5)x3(t-1) + (c6)x4(t-1) + u2(t)
x4(t) = u2(t)

Observation equations:
y1(t) = x1(t) + (c7)e1(t)
y2(t) = x3(t) + (c8)e2(t)

Initial state distribution:

Initial state means are not specified.
Initial state covariance matrix is not specified.
State types are not specified.
```

Estimate the model parameters, and use a random set of initial parameter values for optimization. Restrict the estimate of σ_1 and σ_2 to all positive, real numbers using the `'lb'` name-value pair argument. For numerical stability, specify the Hessian when the software computes the parameter covariance matrix, using the `'CovMethod'` name-value pair argument.

```
rng(1);
params0 = rand(8,1);
[EstMdl,estParams] = estimate(Mdl,y,params0,...
    'lb',[-Inf -Inf -Inf -Inf -Inf -Inf 0 0],'CovMethod','hessian');
```

```
Method: Maximum likelihood (fmincon)
Sample size: 61
Logarithmic  likelihood:     -199.397
Akaike   info criterion:      414.793
Bayesian info criterion:       431.68
      |     Coeff       Std Err    t Stat     Prob
-------------------------------------------------------
 c(1) |  0.03387       0.15213     0.22262  0.82383
 c(2) | -0.01258       0.05749    -0.21876  0.82684
 c(3) |  2.49856       0.22759    10.97828   0
 c(4) |  0.77437       2.58647     0.29939  0.76464
 c(5) |  0.13993       2.64354     0.05293  0.95779
 c(6) |  0.00368       2.45466     0.00150  0.99880
 c(7) |  0.00238       2.11321     0.00113  0.99910
```

```
 c(8) |  0.00014       0.12685     0.00113  0.99910
      |
      |   Final State   Std Dev     t Stat    Prob
 x(1) |  1.40000       0.00238   587.37950   0
 x(2) |  0.21778       0.91641     0.23765  0.81216
 x(3) |  0.04730       0.00014   329.53907   0
 x(4) |  0.03568       0.00015   240.96251   0
```

`EstMdl` is an `ssm` model, and you can access its properties using dot notation.

Simulate `1e4` paths of observations from the fitted, state-space model `EstMdl` using the simulation smoother. Specify to simulate observations for each period.

```
numPaths = 1e4;
SimX = simsmooth(EstMdl,y,'NumPaths',numPaths);
```

`SimX` is a `T - 1`-by-`4`-by-`numPaths` matrix containing the simulated states. The rows of `SimX` correspond to periods, the columns correspond to a state in the model, and the pages correspond to paths.

Estimate the smoothed state means, standard deviations, and 95% confidence intervals.

```
SmoothBar = mean(SimX,3);
SmoothSTD = std(SimX,0,3);
SmoothCIL = SmoothBar - 1.96*SmoothSTD;
SmoothCIU = SmoothBar + 1.96*SmoothSTD;
```

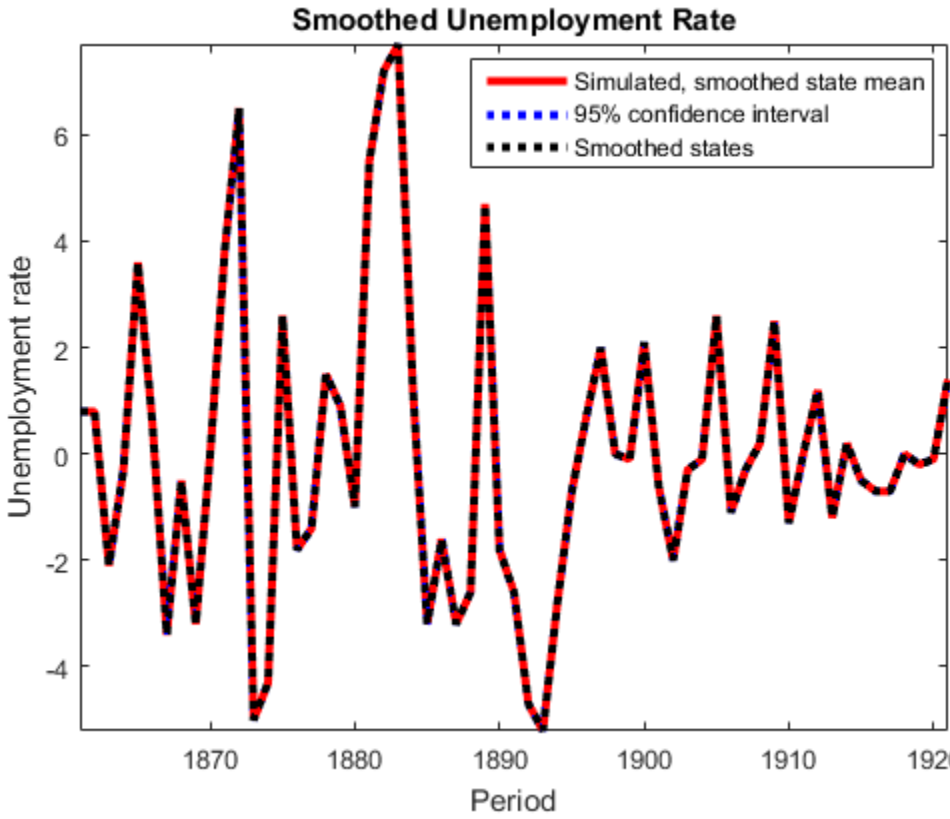
Estimate smooth states using `smooth`.

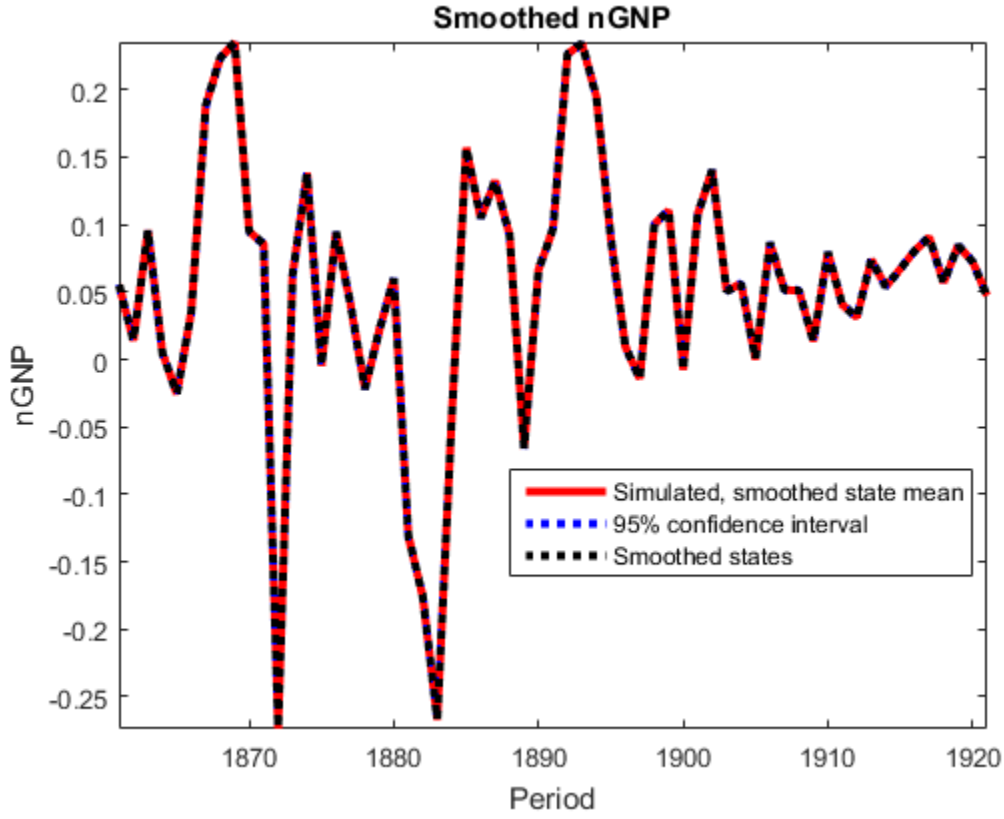
```
SmoothX = smooth(EstMdl,y);
```

Plot the smoothed states, and the means of the simulated states and their 95% confidence intervals.

```
figure
h = plot(dates(2:T),SmoothBar(:,1),'-r',...
    dates(2:T),SmoothCIL(:,1),':b',...
    dates(2:T),SmoothCIU(:,1),':b',...
    dates(2:T),SmoothX(:,1),':k',...
    'LineWidth',3);
xlabel 'Period';
ylabel 'Unemployment rate';
legend(h([1,2,4]),{'Simulated, smoothed state mean','95% confidence interval',...
    'Smoothed states'},'Location','Best');
title 'Smoothed Unemployment Rate';
```

```
axis tight

figure
h = plot(dates(2:T),SmoothBar(:,3),'-r',...
    dates(2:T),SmoothCIL(:,3),':b',...
    dates(2:T),SmoothCIU(:,3),':b',...
    dates(2:T),SmoothX(:,3),':k',...
    'LineWidth',3);
xlabel 'Period';
ylabel 'nGNP';
legend(h([1,2,4]),{'Simulated, smoothed state mean','95% confidence interval',...
    'Smoothed states'},'Location','Best');
title 'Smoothed nGNP';
axis tight
```





The simulated state means are practically identical to the smoothed states.

See Also

simsmooth | simulate | smooth | ssm

Rolling-Window Analysis of Time-Series Models

Rolling-window analysis of a time-series model assesses:

- The stability of the model over time. A common time-series model assumption is that the coefficients are constant with respect to time. Checking for instability amounts to examining whether the coefficients are time-invariant.
- The forecast accuracy of the model.

In this section...

Rolling-Window Analysis for Parameter Stability

Suppose that you have data for all periods in the sample. To check the stability of a time-series model using a rolling window:

1. Choose a rolling window size, m, i.e., the number of consecutive observation per rolling window. The size of the rolling window will depend on the sample size, T, and periodicity of the data. In general, you can use a short rolling window size for data collected in short intervals, and a larger size for data collected in longer intervals. Longer rolling window sizes tend to yield smoother rolling window estimates than shorter sizes.
2. Suppose that the number of increments between successive rolling windows is 1 period, then partition the entire data set into $N = T - m + 1$ subsamples. The first rolling window contains observations for period 1 through m, the second rolling window contains observations for period 2 through $m + 1$, and so on.

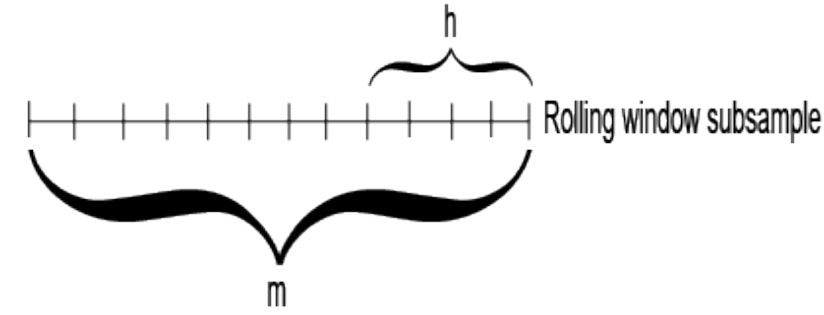
   There are variations on the partitions, e.g., rather than roll one observation ahead, you can roll four observations for quarterly data.
3. Estimate the model using each rolling window subsamples.
4. Plot each estimate and point-wise confidence intervals (i.e., $\hat{\theta} \pm 2\left[\widehat{SE}\left(\hat{\theta}\right)\right]$) over the rolling window index to see how the estimate changes with time. You should expect a little fluctuation for each, but large fluctuations or trends indicate that the parameter might be time varying.

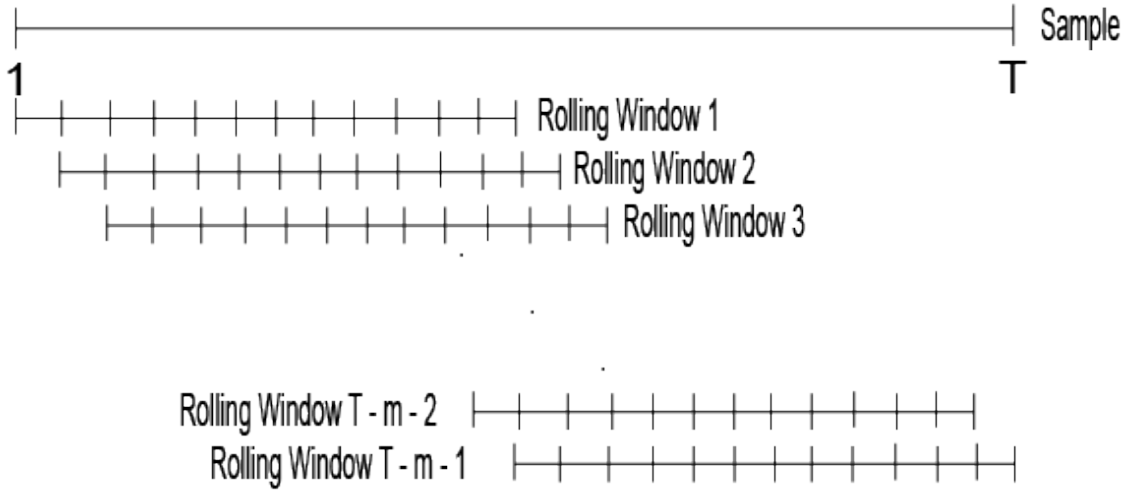
For more details on assessing the stability of a model using rolling window analysis, see [1].

Rolling Window Analysis for Predictive Performance

Suppose that you have data for all periods in the sample. You can *backtest* to check the predictive performance of several time-series models using a rolling window. These steps outline how to backtest.

1. Choose a rolling window size, m, i.e., the number of consecutive observation per rolling window. The size of the rolling window depends on the sample size, T, and periodicity of the data. In general, you can use a short rolling window size for data collected in short intervals, and a larger size for data collected in longer intervals. Longer rolling window sizes tend to yield smoother rolling window estimates than shorter sizes.
2. Choose a forecast horizon, h. The forecast horizon depends on the application and periodicity of the data. The following illustrates how the rolling window partitions the data set.
3. If the number of increments between successive rolling windows is 1 period, then partition the entire data set into $N = T - m + 1$ subsamples. The first rolling window contains observations for period 1 through m, the second rolling window contains observations for period 2 through $m + 1$, and so on. The figure illustrates the partitions.





There are variations on the partitions, e.g., rather than roll one observation ahead, you can roll four observations for quarterly data.

4 For each rolling window subsample:

   a Estimate each model.

   b Estimate h-step-ahead forecasts.

c Compute the forecast errors for each forecast, that is $e_{nj} = y_{m-h+n+j} - \hat{y}_{nj}$, where:

- e_{nj} is the forecast error of rolling window n for the j-step-ahead forecast.
- y is the response.
- $\hat{y}_{nj}$ is the j-step-ahead forecast of rolling window subsample n.

5 Compute the root forecast mean squared errors (RMSEs) using the forecast errors for each step-ahead forecast type. In other words,

$$RMSE_j = \sqrt{\frac{\sum_{n=1}^{N} e_{nj}^2}{n}} \text{ for } j = 1,\ldots,h.$$

6 Compare the RMSEs among the models. The model with the lowest set of RMSEs has the best predictive performance.

For more details on backtesting, see [1].

References

[1] Zivot, E., and J. Wang. *Modeling Financial Time Series with S_PLUS®*. 2nd ed. NY: Springer Science+Business Media, Inc., 2006.

Related Examples

- "Assess Model Stability Using Rolling Window Analysis" on page 8-172
- "Choose State-Space Model Specification Using Backtesting" on page 8-181

Assess State-Space Model Stability Using Rolling Window Analysis

In this section...

Assess Model Stability Using Rolling Window Analysis

This example shows how to use a rolling window to check whether the parameters of a time-series model are time invariant. This example analyzes two time series:

- Time-series 1: simulated data from a known, time-invariant model
- Time-series 2: simulated data from a known, time-varying model

Completely specify this AR(1) model for Time-series 1:

$$y_t = 0.6y_{t-1} + \varepsilon_t$$

where ε_t is Gaussian with mean 0 and variance 1. Completely specify this time-varying model for Time-series 2:

$$\begin{aligned} y_t &= 0.2y_{t-1} + \varepsilon_t; t = 1, ..., 100 \\ y_t &= 0.75y_{t-1} + \varepsilon_t; t = 101, ..., 150 \\ y_t &= -0.5y_{t-1} + \varepsilon_t; t = 151, ..., 200, \end{aligned}$$

where ε_t is Gaussian with mean 0 and variance 1.

```
Mdl1 = arima('AR',0.6,'Constant',0,'Variance',1);

Mdl2 = cell(3,1); % Preallocate
ARMdl2 = [0.2 0.75 -0.5];
for j = 1:3;
    Mdl2{j} = arima('AR',ARMdl2(j),'Constant',0,'Variance',1);
end
```

`Mdl1` is an `arima` model objects. You can access its properties using dot notation. `Mdl2` is a cell array of `arima` model objects. You can you cell indexing and dot notation to access properties of the models within `Mdl2`. For example, to access the AR parameter value of the third model in `Mdl3`, enter `Mdl2{3}.AR`.

Simulate `T` = 200 periods of data from `Mdl1` and `Mdl2`. Use a presample response of 0 for both series.

```
rng(1); % For reproducibility
T = 200;
y1 = simulate(Mdl1,T,'Y0',0);
timeMdl2 = [100 50 50]; % Number of observations per model in Mdl2
y2 = 0;
for k = 1:numel(Mdl2);
    y2 = [y2; simulate(Mdl2{k},timeMdl2(k),'Y0',y2(end))];
end

Y = [y1 y2(2:end)];
```

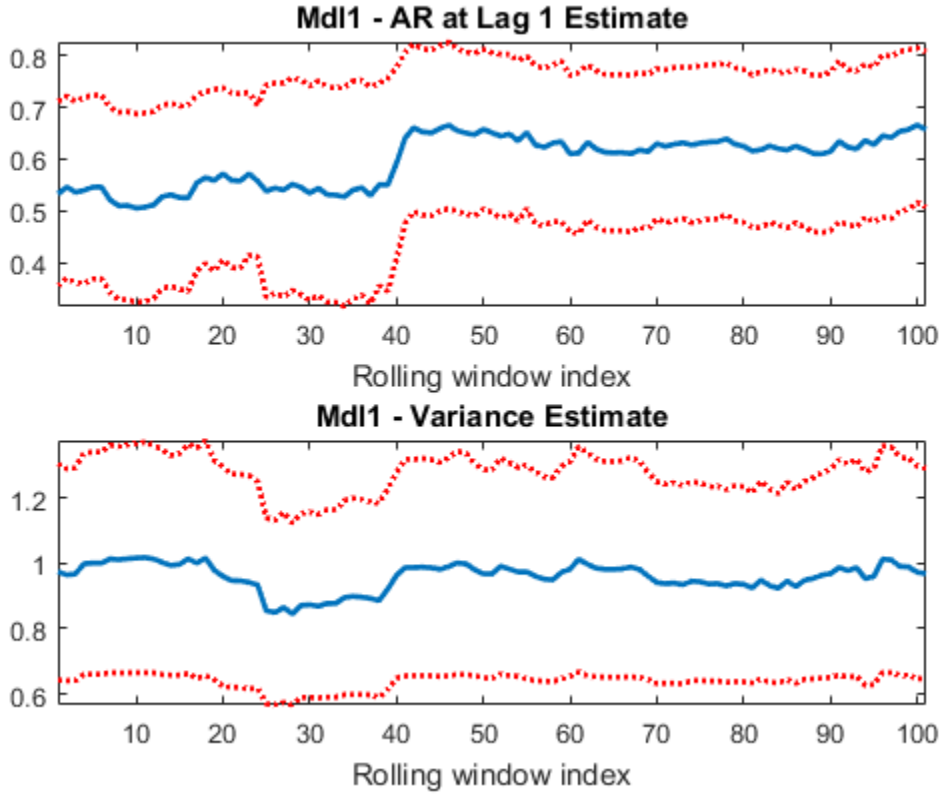
Specify empty AR(1) models for the estimation of `Mdl1`, `Mdl2`, and `Mdl3`. Estimate all three models using the respective data sets and a rolling window size of 40 periods. Also, use a rolling window increment of one period. Store the autoregressive parameters and estimated innovations variance.

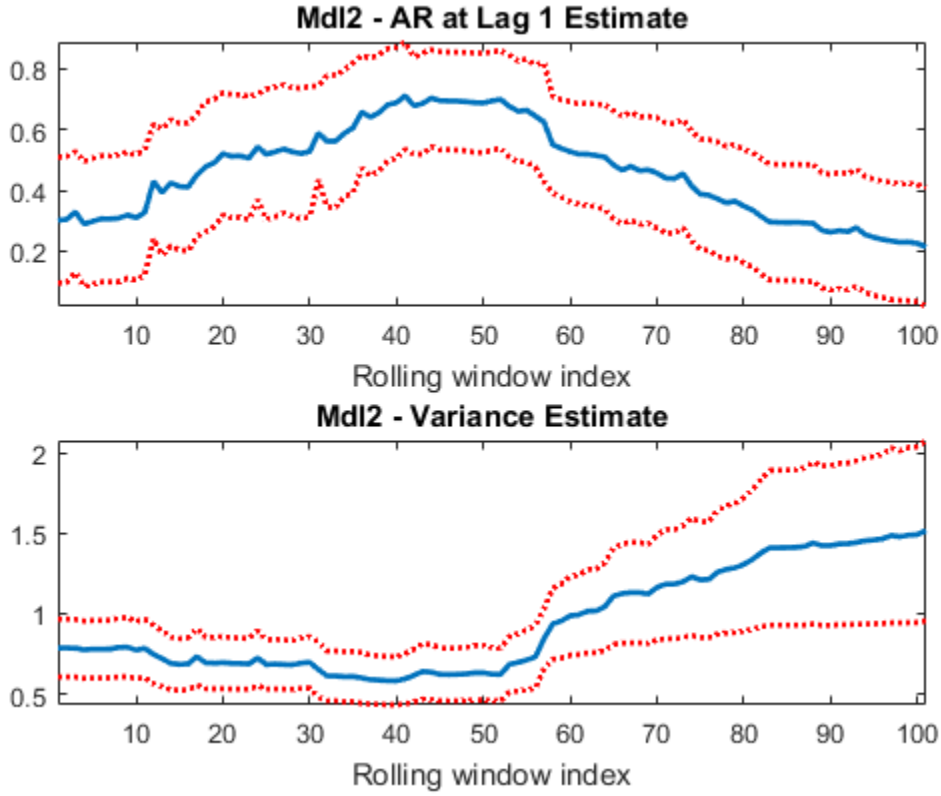
```
ToEstMdl = arima('ARLags',1,'Constant',0);

m = 100;       % Rolling window size
N = T - m + 1; % Number of rolling windows

EstParams = cell(2,1);     % Preallocate for estimates
EstParamsMat = zeros(N,2);
EstParamsSE = cell(2,1);
EstParamsSEMat = zeros(N,2);

for j = 1:2;
    for k = 1:N;
        idxRW =  k:(m + k - 1); % In-sample indices
        [EstMdl,EstParamCov] = estimate(ToEstMdl,Y(idxRW,j),'Display','off');
        EstParamsMat(k,:) = [EstMdl.AR{1} EstMdl.Variance];
        EstParamsSEMat(k,:) = sqrt([EstParamCov(2,2) EstParamCov(3,3)]);
    end
    EstParams{j} = EstParamsMat;
    EstParamsSE{j} = EstParamsSEMat;
end
```

Plot the estimates and their point-wise confidence intervals over the rolling window index.

```
titleMdls = {'Mdl1','Mdl2'};

for j = 1:2;
    figure;
    subplot(2,1,1);
```

```
    Estimates = EstParams{j};
    SEs = EstParamsSE{j};
    plot(Estimates(:,1),'LineWidth',2);
    hold on;
    plot(Estimates(:,1) + 2*SEs(:,1),'r:','LineWidth',2);
    plot(Estimates(:,1) - 2*SEs(:,1),'r:','LineWidth',2);
    title(sprintf('%s - AR at Lag 1 Estimate',titleMdls{j}));
    xlabel 'Rolling window index';
    axis tight;
    hold off;
    subplot(2,1,2);
    plot(Estimates(:,2),'LineWidth',2);
    hold on;
    plot(Estimates (:,2) + 2*SEs(:,2),'r:','LineWidth',2);
    plot(Estimates(:,2) - 2*SEs(:,2),'r:','LineWidth',2);
    title(sprintf('%s - Variance Estimate',titleMdls{j}));
    xlabel 'Rolling window index';
    axis tight;
    hold off;
end
```


Mdl1 - AR at Lag 1 Estimate
0.8
0.7
0.6
0.5
0.4
10
20
30
40
50
60
70
80
90
100
Rolling window index
Mdl1 - Variance Estimate
1.2
1
0.8
0.6
10
20
30
40
50
60
70
80
90
100
Rolling window index



For `Mdl1`, the AR estimate does not vary much from 0.6, and the estimates are not significantly different from one another (pair-wise). Similar results occur for the variance of `Mdl1`. The AR estimate of `Mdl2` grows, and then falls, which indicates time dependence. Also, based on the confidence intervals, there is evidence that some estimates differ from others. Though the variance did not change during simulation, there seems to be heteroscedasticity possibly induced by the instability of the model.

Assess Stability of Implicitly Created State-Space Model

This example shows how to specify and estimate a state space model when conducting a rolling window analysis for stability. A rolling window analysis for an explicitly defined

state-space model is straightforward, so this example focuses on implicitly defined state-space models.

Consider this state-space model:

$$\begin{aligned} x_t &= \phi x_{t-1} + \varepsilon_t \\ y_t - \beta z_t &= x_t + u_t \end{aligned},$$

where ε_t and u_t are Gaussian process with mean 0 and variance 1. Create the function `rwParamMap.m`, which specifies how the parameters in `params` map to the state-space model matrices, the initial state values, and the type of state, and save it in your working folder.

```
function [A,B,C,D,Mean0,Cov0,StateType,deflateY] = rwParamMap(params,y,Z)
%rwParamMap Parameter-to-matrix mapping function for rolling window example
%using ssm and specifying an AR(1) state model
%   The state space model specified by rwParamMap contains a stationary
%   AR(1) state, the observation model includes a regression component, and
%   the variances of the innovation and disturbances are 1. The response y
%   is deflated by the regression component specified by the predictor
%   variables x.
A = params(1);
B = 1;
C = 1;
D = 1;
Mean0 = [];
Cov0 = [];
StateType = 0;
deflateY = y - params(2)*Z;
end
```

The software does not support the simulation of implicit models containing a regression component. Therefore, to simulate data from this model, you must specify all model components up to the regression component. You can do this explicitly since this example uses a simple state-space model. Otherwise, you can create another function and define another state-space model implicitly (e.g., for time-varying state-space models).

```
Mdl2Sim = ssm(NaN,1,1,1,'StateType',1);
```

`Mdl2Sim` is an implicitly defined `ssm` object.

Simulate a 200-period path of random standard Gaussian data. Then, simulate responses from `Mdl2Sim`, and inflate the responses with the regression component. For this example, use $\phi = 0.6$ and $\beta = 2$.

```
rng(1); % For reproducibility
T = 200;
Z = randn(T,1);
phi = 0.6;
beta = 2;
deflateY = simulate(Mdl2Sim,T,'Params',phi);
y = deflateY + Z*beta;
```

`y` is the inflated, simulated response path, and `Z` is the simulated predictor series.

If you define a state-space model implicitly and the response and predictor data (i.e., `y` and `Z`) exist in the MATLAB Workspace, then the software creates a link from the parameter-to-matrix mapping function those series. If the data do not exist in the MATLAB Workspace, then the software creates the model, but you must provide the data using the appropriate name-value pair arguments when you, e.g., estimate the model.

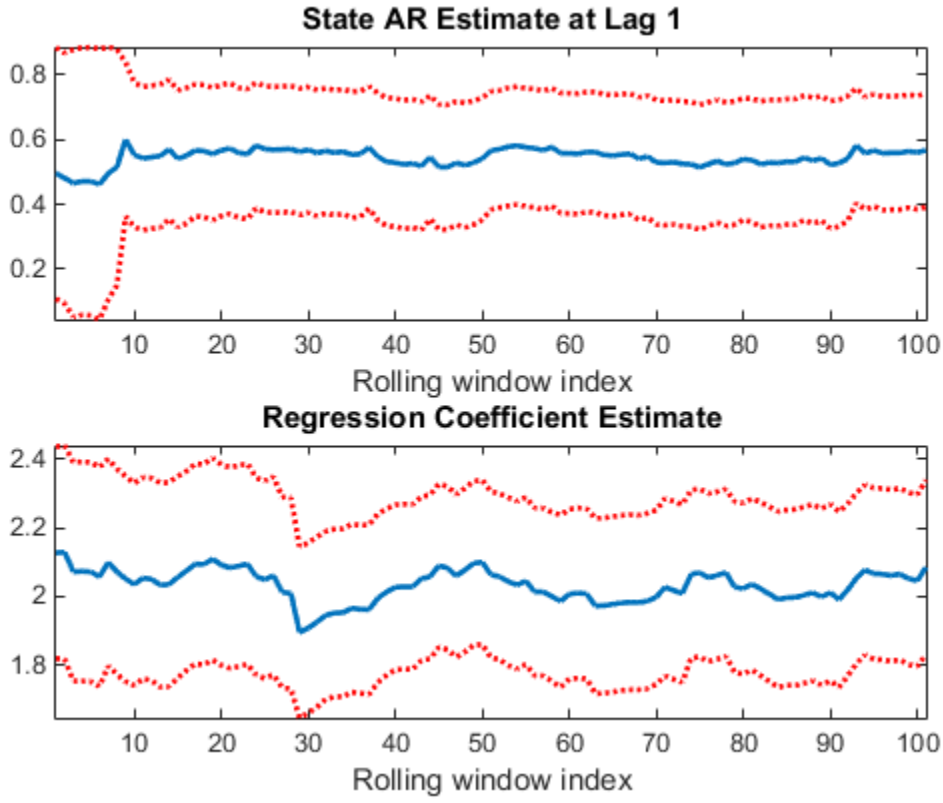
Therefore, to conduct a rolling window analysis when the state-space model is implicitly defined and there is a regression component, you must specify the state-space model indicating the indices of the data to be analyzed for each window. Conduct a rolling window analysis of the simulated data. Let the rolling window length be 100 periods for this example.

```
m = 100;
N = T - m + 1;        % Number of rolling windows

EstParams = nan(N,2); % Preallocation
EstParamSE = nan(N,2);

for j = 1:N;
    idxRW = j:(m + j - 1);
    Mdl = ssm(@(c)rwParamMap(c,y(idxRW),Z(idxRW)));
    [~,EstParams(j,:),EstParamCov] = estimate(Mdl,y(idxRW),[0.5 1]',...
        'Display','off');
    EstParamSE(j,:) = sqrt(diag(EstParamCov));
end
```

Plot the estimates and point-wise confidence intervals for the AR parameter and regression coefficient.

```
figure;
```

```
subplot(2,1,1);
plot(EstParams(:,1),'LineWidth',2);
hold on;
plot(EstParams(:,1) + 2*EstParamSE(:,1),':r','LineWidth',2);
plot(EstParams(:,1) - 2*EstParamSE(:,1),':r','LineWidth',2);
title 'State AR Estimate at Lag 1';
xlabel 'Rolling window index';
axis tight;
hold off;
subplot(2,1,2);
plot(EstParams(:,2),'LineWidth',2);
hold on;
plot(EstParams(:,2) + 2*EstParamSE(:,2),':r','LineWidth',2);
plot(EstParams(:,2) - 2*EstParamSE(:,2),':r','LineWidth',2);
title 'Regression Coefficient Estimate';
xlabel 'Rolling window index';
axis tight;
hold off;
```



The plots indicate that the model is stable since the AR estimate does not deviate much from its mean, nor does the regression coefficient estimate.

Choose State-Space Model Specification Using Backtesting

This example shows how to choose the state-space model specification with the best predictive performance using a rolling window. A rolling window analysis for an explicitly defined state-space model is straightforward, so this example focuses on implicitly defined state-space models.

Suppose that the linear relationship between the change in the observed unemployment rate and the nominal gross national product (nGNP) growth rate is of interest. Suppose further that you want to choose between an AR(1) or an AR(2) model for the first difference of the unemployment rate (i.e., the state). That is,

$$\begin{aligned} &\text{Model 1:} \quad x_t = \phi_{11}x_{t-1} + \phi_{12}x_{t-1} + \varepsilon_t \\ &\qquad\qquad y_t - \beta_1 z_t = x_t + u_t \\ &\text{Model 2:} \quad x_t = \phi_{21}x_{t-1} + \varepsilon_t \\ &\qquad\qquad y_t - \beta_2 z_t = x_t + u_t \end{aligned},$$

where:

- ε_t and u_t are Gaussian process with mean 0 and variance 1.
- x_t is the true unemployment rate at time t.
- y_t is the observed unemployment.
- z_t is the nGNP rate.

Create the functions `rwParamMap.m` and `rwAR2ParamMap.m` in separate functions, which specify how the parameters in `params` map to the state-space model matrices, the initial state values, and the type of state, and save them in your working folder.

```
function [A,B,C,D,Mean0,Cov0,StateType,deflateY] = rwParamMap(params,y,Z)
%rwParamMap Parameter-to-matrix mapping function for rolling window example
%using ssm and specifying an AR(1) state model
%   The state space model specified by rwParamMap contains a stationary
%   AR(1) state, the observation model includes a regression component, and
%   the variances of the innovation and disturbances are 1. The response y
%   is deflated by the regression component specified by the predictor
%   variables x.
A = params(1);
B = 1;
C = 1;
```

```
D = 1;
Mean0 = [];
Cov0 = [];
StateType = 0;
deflateY = y - params(2)*Z;
end


function [A,B,C,D,Mean0,Cov0,StateType,deflateY] = rwAR2ParamMap(params,y,Z)
%rwParamMap Parameter-to-matrix mapping function for rolling window example
%using ssm and specifying an AR(2) state model
%   The state space model specified by rwParamMap contains a stationary
%   AR(2) state, the observation model includes a regression component, and
%   the variances of the innovation and disturbances are 1. The response y
%   is deflated by the regression component specified by the predictor
%   variables x.
A = [params(1) params(2); 1 0];
B = [1; 0];
C = [1 0];
D = 1;
Mean0 = [];
Cov0 = [];
StateType = [0 0];
deflateY = y - params(3)*Z;
end
```

Load the Nelson-Plosser data set, which contains the unemployment rate and nGNP series, among other things.

```
load Data_NelsonPlosser
```

Preprocess the data by taking the natural logarithm of the nGNP series, and the first difference of each. Also, remove the starting `NaN` values from each series.

```
isNaN = any(ismissing(DataTable),2); % Flag periods containing NaNs
gnpn = DataTable.GNPN(~isNaN);
ur = DataTable.UR(~isNaN);           % Sample size
Z = diff(log(gnpn));
y = diff(ur);
T = size(y,1);
```

If you define a state-space model implicitly and the response and predictor data (i.e., `y` and `Z`) exist in the MATLAB Workspace, then the software creates a link from the

parameter-to-matrix mapping function those series. If the data do not exist in the MATLAB Workspace, then the software creates the model, but you must provide the data using the appropriate name-value pair arguments when you, e.g., estimate the model.

Therefore, to conduct a rolling window analysis when the state-space model is implicitly defined and there is a regression component, you must specify the state-space model indicating the indices of the data to be analyzed for each window. Conduct a rolling window analysis of the simulated data. Let the rolling window length (`m`) be 40 periods and the forecast horizon (`h`) be 10 periods. For this example, assume that the time-series are stable (i.e., all parameters are time-invariant).

```
m = 40;
N = T - m + 1;      % Number of rolling windows
h = 10;

fError1 = nan(N,h); % Preallocation
fError2 = nan(N,h);

for j = 1:N;
    idxRW = j:(m + j - h - 1);
  idxFH = (m + j - h):(m + j - 1);
    Mdl1 = ssm(@(c)rwParamMap(c,y(idxRW),Z(idxRW)));
    Mdl2 = ssm(@(c)rwARMAParamMap(c,y(idxRW),Z(idxRW)));
    [EstMdl1,estParams1] = estimate(Mdl1,y(idxRW),[0.5 -20]',...
        'Display','off');
    [EstMdl2,estParams2] = estimate(Mdl2,y(idxRW),[0.5 0.1 -20]',...
       'Display','off');
    fY1 = forecast(EstMdl1,h,y(idxRW),'PredictorsO',Z(idxRW),...
        'PredictorsF',Z(idxFH),'Beta',estParams1(end));
    fY2 = forecast(EstMdl2,h,y(idxRW),'PredictorsO',Z(idxRW),...
        'PredictorsF',Z(idxFH),'Beta',estParams2(end));
  fError1(j,:) = y(idxFH) - fY1;
  fError2(j,:) = y(idxFH) - fY2;
end
```

Compute the RMSE for each step-ahead forecast, and compare them for each model.

```
fRMSE1 = sqrt(mean(fError1.^2));
fRMSE2 = sqrt(mean(fError2.^2));
fRMSE1 < fRMSE2

ans =

     1     1     1     0     0     0     0     0     0     0
```

Overall, the predictive performance of the AR(2) model is better than the AR(1) model.

Alternatively, you can compare the predictive performance of the models using the Diebold-Mariano test. For more details, see [1].

9

Functions — Alphabetical List

adftest

Augmented Dickey-Fuller test

Syntax

```
h = adftest(Y)
h = adftest(Y,Name,Value)
[h,pValue] = adftest( ___ )
[h,pValue,stat,cValue,reg] = adftest( ___ )
```

Description

`h = adftest(Y)` returns a logical value with the rejection decision from conducting an augmented Dickey-Fuller test for a unit root in a univariate time series, `Y`.

`h = adftest(Y,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

- If any `Name,Value` argument is a vector, then all `Name,Value` arguments specified must be vectors of equal length or length one. `adftest(Y,Name,Value)` treats each element of a vector input as a separate test, and returns a vector of rejection decisions.
- If any `Name,Value` argument is a row vector, then `adftest(Y,Name,Value)` returns a row vector.

`[h,pValue] = adftest( ___ )` returns the rejection decision and p-value for the hypothesis test, using any of the input arguments in the previous syntaxes.

`[h,pValue,stat,cValue,reg] = adftest( ___ )` additionally returns the test statistic, critical value, and a structure of regression statistics for the hypothesis test.

Examples

Conduct a Dickey-Fuller Test Without Augmentation

Test a time series for a unit root using the default autoregression model without augmented difference terms.

Load Canadian inflation rate data.

```
load Data_Canada
Y = DataTable.INF_C;
```

Test the time series for a unit root.

```
h = adftest(Y)
```

```
h =

     0
```

The result `h = 0` indicates that this test fails to reject the null hypothesis of a unit root against the autoregressive alternative.

Conduct an Augmented Dickey-Fuller Test Against a Trend-Stationary Alternative

Test a time series for a unit root against a trend-stationary alternative augmented with lagged difference terms.

Load a time series of GDP data, and calculate its log.

```
load Data_GDP;
Y = log(Data);
```

Test for a unit root against a trend-stationary alternative, augmenting the model with 0, 1, and 2 lagged difference terms.

```
h = adftest(Y,'model','TS','lags',0:2)
```

```
h =

     0     0     0
```

`adftest` treats the three lag choices as three separate tests, and returns a vector with rejection decisions for each test. The values `h = 0` indicate that all three tests fail to reject the null hypothesis of a unit root against the trend-stationary alternative.

Choose the Number of Lagged Difference Terms to Include in the Augmented Model

Test a time series for a unit root against trend-stationary alternatives augmented with different numbers of lagged difference terms. Look at the regression statistics

corresponding to each of the alternative models to choose how many lagged difference terms to include in the augmented model.

Load a time series of GDP data, and calculate its log.

```
load Data_GDP;
Y = log(Data);
```

Test for a unit root using three different choices for the number of lagged difference terms. Return the regression statistics for each alternative model.

```
[h,~,~,~,reg] = adftest(Y,'model','TS','lags',0:2);
```

`adftest` treats each of the three lag choices as separate tests, and returns results for each test. `reg` is an array of three data structures, corresponding to each alternative model.

Display the names of the coefficients included in each of the three alternatives.

```
reg.names
```

```
ans =

    'c'
    'd'
    'a'


ans =

    'c'
    'd'
    'a'
    'b1'


ans =

    'c'
    'd'
    'a'
    'b1'
    'b2'
```

The output shows which terms are included in the three alternative models. The first model has no added difference terms, the second model has one difference term (`b1`), and the third model has two difference terms (`b1` and `b2`).

Display the t-statistics and corresponding p-values for each coefficient in the three alternative models.

```
[reg(1).tStats.t reg(1).tStats.pVal]
[reg(2).tStats.t reg(2).tStats.pVal]
[reg(3).tStats.t reg(3).tStats.pVal]
```

```
ans =

    2.0533    0.0412
    1.8842    0.0608
   61.4717    0.0000


ans =

    2.9026    0.0041
    2.7681    0.0061
   64.1396    0.0000
    5.6514    0.0000


ans =

    3.2568    0.0013
    3.1249    0.0020
   62.7825    0.0000
    4.7586    0.0000
    1.7615    0.0795
```

The returned t-statistics and p-values correspond to the coefficients in `reg.names`. These results indicate that the coefficient on the first difference term is significantly different from zero in both the second and third models, but the coefficient on the second term in the third model is not. This suggests augmenting the model with one lagged difference term is adequate.

Compare the BIC for each of the three alternatives.

```
reg.BIC
```

```
ans =

  -1.4774e+03


ans =

  -1.4966e+03


ans =

  -1.4878e+03
```

Based on the BIC values, choose the model augmented with one lagged difference term because it has the best (that is, the smallest) BIC value.

- "Unit Root Tests" on page 3-44

Input Arguments

Y — Univariate time series

column vector

Univariate time series, specified as a column vector. The last element is the most recent observation. `adftest` ignores missing observations, indicated by `NaN`s.

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'alpha',0.1,'lags',0:2` specifies three tests with 0, 1, and 2 lagged difference terms conducted at the 0.1 significance level

'alpha' — Significance levels

`0.05` (default) | scalar | vector

Significance levels for the hypothesis tests, specified as the comma-separated pair consisting of `'alpha'` and a scalar or vector. Use a vector to conduct multiple tests. All values of `alpha` must be between `0.001` and `0.999`.

Example: `'alpha',0.01`

Data Types: `double`

`'lags'` — Number of lagged difference terms

`0` (default) | nonnegative integer | vector of nonnegative integers

Number of lagged difference terms to include in the model, specified as the comma-separated pair consisting of `'lags'` and a nonnegative integer or vector of nonnegative integers. Use a vector to conduct multiple tests.

Example: `'lags',[0,1,2]`

Data Types: `double`

`'model'` — Model variant

`'AR'` (default) | `'ARD'` | `'TS'`

Model variant, specified as the comma-separated pair consisting of `'model'` and a string or cell array of strings. Use a cell array of strings to conduct multiple tests with different model variants. The possible values are:

`'AR'` Autoregressive model variant, which specifies a test of the null model

$$y_t = y_{t-1} + \beta_1 \Delta y_{t-1} + \beta_2 \Delta y_{t-2} + \ldots + \beta_p \Delta y_{t-p} + \varepsilon_t$$

against the alternative model

$$y_t = \phi y_{t-1} + \beta_1 \Delta y_{t-1} + \beta_2 \Delta y_{t-2} + \ldots + \beta_p \Delta y_{t-p} + \varepsilon_t,$$

with AR(1) coefficient, $\phi < 1$.

`'ARD'` Autoregressive model with drift variant, which specifies a test of the null model

$$y_t = y_{t-1} + \beta_1 \Delta y_{t-1} + \beta_2 \Delta y_{t-2} + \ldots + \beta_p \Delta y_{t-p} + \varepsilon_t$$

against the alternative model

$$y_t = c + \phi y_{t-1} + \beta_1 \Delta y_{t-1} + \beta_2 \Delta y_{t-2} + \ldots + \beta_p \Delta y_{t-p} + \varepsilon_t,$$

with drift coefficient, *c*, and AR(1) coefficient, $\phi < 1$.

`'TS'` Trend-stationary model variant, which specifies a test of the null model

$$y_t = c + y_{t-1} + \beta_1 \Delta y_{t-1} + \beta_2 \Delta y_{t-2} + \ldots + \beta_p \Delta y_{t-p} + \varepsilon_t$$

against the alternative model

$$y_t = c + \delta t + \phi y_{t-1} + \beta_1 \Delta y_{t-1} + \beta_2 \Delta y_{t-2} + \ldots + \beta_p \Delta y_{t-p} + \varepsilon_t,$$

with drift coefficient, *c*, deterministic trend coefficient, δ, and AR(1) coefficient, $\phi < 1$.

Example: `'model',{'AR','ARD'}`

`'test'` — Test statistic

`'t1'` (default) | `'t2'` | `'F'`

Test statistic, specified as the comma-separated pair consisting of `'test'` and a string or cell array of strings with these possible values:

`'t1'` Standard *t* statistic,

$$t_1 = (\phi - 1) / se,$$

computed using the OLS estimate of the AR(1) coefficient, $\hat{\phi}$, and its standard error (*se*), in the alternative model.

The test assesses the significance of the restriction, $\phi - 1 = 0$.

`'t2'` Lag-adjusted, unstudentized *t* statistic,

$$t_2 = N(\phi - 1) / (1 - \beta_1 - \ldots - \beta_p),$$

computed using the OLS estimates of the AR(1) coefficient and stationary coefficients in the alternative model. N is the effective sample size, adjusted for lags and missing values.

The test assesses the significance of the restriction, $\phi - 1 = 0$.

`'F'` F statistic for assessing the significance of a joint restriction on the alternative model.

- For model variant `'ARD'`, the restrictions are $\phi - 1 = 0$ and $c = 0$.
- For model variant `'TS'`, the restrictions are $\phi - 1 = 0$ and $\delta = 0$.

An F statistic is invalid for model variant `'AR'`.

Use a cell array of strings to conduct multiple tests using different test statistics.

Example: `'test','t2'`

Output Arguments

h — Test rejection decisions

logical | vector of logicals

Test rejection decisions, returned as a logical value or vector of logical values with length equal to the number of tests conducted.

- `h = 1` indicates rejection of the unit-root null in favor of the alternative model.
- `h = 0` indicates failure to reject the unit-root null.

pValue — Test statistic p-values

scalar | vector

Test statistic p-values, returned as a scalar or vector with length equal to the number of tests conducted.

- If the test statistic is `'t1'` or `'t2'`, then the p-values are left-tail probabilities.
- If the test statistic is `'F'`, then the p-values are right-tail probabilities.

stat — Test statistics

scalar | vector

Test statistics, returned as a scalar or vector with length equal to the number of tests conducted. `adftest` computes test statistics using ordinary least squares (OLS) estimates of the coefficients in the alternative model.

`cValue` — Critical values

scalar | vector

Critical values, returned as a scalar or vector with length equal to the number of tests conducted.

- If the test statistic is `'t1'` or `'t2'`, then the critical values are for left-tail probabilities.
- If the test statistic is `'F'`, then the critical values are for right-tail probabilities.

`reg` — Regression statistics

data structure | data structure array

Regression statistics for ordinary least squares (OLS) estimation of coefficients in the alternative model, returned as a data structure or data structure array with length equal to the number of tests conducted.

Each data structure has the following fields.

Field	Description
`num`	Length of input series with `NaN`s removed
`size`	Effective sample size, adjusted for lags
`names`	Regression coefficient names
`coeff`	Estimated coefficient values
`se`	Estimated coefficient standard errors
`Cov`	Estimated coefficient covariance matrix
`tStats`	*t* statistics of coefficients and p-values
`FStat`	*F* statistic and p-value
`yMu`	Mean of the lag-adjusted input series
`ySigma`	Standard deviation of the lag-adjusted input series
`yHat`	Fitted values of the lag-adjusted input series
`res`	Regression residuals

Field	Description
DWStat	Durbin-Watson statistic
SSR	Regression sum of squares
SSE	Error sum of squares
SST	Total sum of squares
MSE	Mean square error
RMSE	Standard error of the regression
RSq	R^2 statistic
aRSq	Adjusted R^2 statistic
LL	Loglikelihood of data under Gaussian innovations
AIC	Akaike information criterion
BIC	Bayesian (Schwarz) information criterion
HQC	Hannan-Quinn information criterion

More About

Augmented Dickey-Fuller Test for a Unit Root

The *Augmented Dickey-Fuller test for a unit root* assesses the null hypothesis of a unit root using the model

$$y_t = c + \delta t + \phi y_{t-1} + \beta_1 \Delta y_{t-1} + \ldots + \beta_p \Delta y_{t-p} + \varepsilon_t,$$

where

- Δ is the differencing operator, such that $\Delta y_t = y_t - y_{t-1}$.
- The number of lagged difference terms, p, is user specified.
- ε_t is a mean zero innovation process.

The null hypothesis of a unit root is

$$H_0 : \phi = 1.$$

Under the alternative hypothesis, $\phi < 1$.

Variants of the model allow for different growth characteristics. The model with $\delta = 0$ has no trend component, and the model with $c = 0$ and $\delta = 0$ has no drift or trend.

A test that fails to reject the null hypothesis, fails to reject the possibility of a unit root.

Algorithms

- `adftest` performs ordinary least squares (OLS) regression to estimate the coefficients in the alternative model.
- Dickey-Fuller statistics follow nonstandard distributions under the null hypothesis (even asymptotically). Critical values for a range of sample sizes and significance levels have been tabulated using Monte Carlo simulations of the null model with Gaussian innovations, with five million replications per sample size.
- For small samples, the tabulated critical values are only valid for Gaussian innovations. For large samples, the tabulated values are still valid for non-Gaussian innovations.
- `adftest` interpolates critical values and p-values from the tables. The tables for test types `'t1'` and `'t2'` are identical to those for `pptest`.

- "Unit Root Nonstationarity" on page 3-34

See Also

`i10test` | `kpsstest` | `lmctest` | `pptest` | `vratiotest`

Introduced in R2009b

aicbic

Akaike or Bayesian information criteria

Syntax

```
aic = aicbic(logL,numParam)
[aic,bic] = aicbic(logL,numParam,numObs)
```

Description

`aic = aicbic(logL,numParam)` returns Akaike information criteria (AIC) corresponding to optimized loglikelihood function values (`logL`), as returned by `estimate`, and the model parameters, `numParam`.

`[aic,bic] = aicbic(logL,numParam,numObs)` additionally returns Bayesian information criteria (BIC) corresponding to `logL`, `numParam`, and the sample sizes associated with each `logL` value.

Examples

Compare AIC Statistics

Calculate and interpret the AIC for four models.

The loglikelihood function values (`logL`) and the number of model parameters (`numParam`) from four multivariate time series analyses are:

```
logL1 = -681.4724;
logL2 = -632.3158;
logL3 = -663.4615;
logL4 = -605.9439;

numParam1 = 12;
numParam2 = 27;
numParam3 = 18;
```

```
numParam4 = 45;
```

Calculate the AIC.

```
aic = aicbic([logL1,logL2,logL3,logL4], ...
    [numParam1,numParam2,numParam3,numParam4])
```

```
aic =

   1.0e+03 *

    1.3869    1.3186    1.3629    1.3019
```

The model with the lowest AIC has the best fit. Therefore, the fourth model fits best.

Information Criteria Statistics for Simulated Data

Compare information criteria statistics for several model fits.

Specify the model

$$y_t = -4 + 0.2y_{t-1} + 0.5y_{t-2} + \varepsilon_t,$$

where ε_t is Gaussian with mean 0 and variance 2. Simulate data from this model.

```
rng(1);  % For random data reproducibility
T = 100; % Sample size
DGP = arima('Constant',-4,'AR',[0.2, 0.5], ...
    'Variance',2);
y = simulate(DGP,T);
```

Define three competing models to fit to the data.

```
EstMdl1 = arima('ARLags',1);
EstMdl2 = arima('ARLags',1:2);
EstMdl3 = arima('ARLags',1:3);
```

Fit the models to the data.

```
logL = zeros(3,1); % Preallocate loglikelihood vector
[~,~,logL(1)] = estimate(EstMdl1,y,'print',false);
[~,~,logL(2)] = estimate(EstMdl2,y,'print',false);
[~,~,logL(3)] = estimate(EstMdl3,y,'print',false);
```

Compute the AIC and BIC for each model.

```
[aic,bic] = aicbic(logL, [3; 4; 5], T*ones(3,1))
```

```
aic =

  381.7732
  358.2422
  358.8479


bic =

  389.5887
  368.6629
  371.8737
```

The model containing two autoregressive lag parameters fits best since it yields the lowest information criteria. The structure of the best fitting model matches the model structure that simulated the data.

- "Time Series Regression V: Predictor Selection"
- "Example: Using Akaike Information Criterion to Calculate the Minimal Requisite Lag" on page 7-20
- "Choose ARMA Lags Using BIC" on page 5-135
- "Compare Conditional Variance Models Using Information Criteria" on page 6-87
- "VAR Model Case Study" on page 7-89

Input Arguments

`logL` — Optimized loglikelihood values

scalar | vector

Optimized loglikelihood objective function values associated with various model fits, specified as a scalar or vector.

Obtain an optimized loglikelihood value using estimate, infer, `vgxvarx`, or an Optimization Toolbox function such as `fmincon` or `fminunc`.

Data Types: `double` | `single`

`numParam` — Number of estimated parameters

scalar | vector

Number of estimated parameters associated with each corresponding fitted model in `logL`, specified as a positive integer, or a vector of positive integers having the same length as `logL`.

If `numParam` is a scalar, then `aicbic` applies it to all `logL` values.

For univariate time series models, use `length(info.X)` to obtain `numParam` from a fitted model returned by `estimate`.

For multivariate time series models, obtain `numParam` using `vgxcount` from a `vgxset` or `vgxvarx` model specification.

Data Types: `double` | `single`

`numObs` — Sample sizes

scalar | vector

Sample sizes of the observed series associated with each corresponding fitted model in `logL`, specified as a positive integer, or a vector of positive integers having the same length as `logL`.

`aicbic` requires `numObs` to compute the BIC.

If `numObs` is a scalar, then `aicbic` applies it to all `logL` values.

Data Types: `double` | `single`

Output Arguments

`aic` — AIC statistics

scalar | vector

AIC statistics associated with each corresponding fitted model in `logL`, returned as a vector with the same length as `logL`.

`bic` — BIC statistics

scalar | vector

BIC statistics associated with each corresponding fitted model in `logL`, returned as a vector with the same length as `logL`.

More About

Akaike Information Criterion

A model fit statistic considers goodness-of-fit and parsimony. Select models that minimize AIC.

When comparing multiple model fits, additional model parameters often yield larger, optimized loglikelihood values. Unlike the optimized loglikelihood value, AIC penalizes for more complex models, i.e., models with additional parameters.

The formula for AIC, which provides insight into its relationship to the optimized loglikelihood and its penalty for complexity, is:

$$\texttt{aic} = -2(\texttt{logL}) + 2(\texttt{numParam}).$$

Bayesian Information Criterion

A model fit statistic considers goodness-of-fit and parsimony. Select models that minimize BIC.

Like AIC, BIC uses the optimal loglikelihood function value and penalizes for more complex models, i.e., models with additional parameters. The penalty of BIC is a function of the sample size, and so is typically more severe than that of AIC.

The formula for BIC is:

$$\texttt{bic} = -2(\texttt{logL}) + \texttt{numParam} * \log(\texttt{numObs}).$$

- “Information Criteria” on page 3-63

References

[1] Box, G. E. P., G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis: Forecasting and Control*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

See Also

estimate | fmincon | fminunc | infer | lmtest | vgxcount | vgxset | vgxvarx | waldtest

Introduced before R2006a

archtest

Engle test for residual heteroscedasticity

Syntax

```
h = archtest(res)
h = archtest(res,Name,Value)
[h,pValue] = archtest( ___ )
[h,pValue,stat,cValue] = archtest( ___ )
```

Description

`h = archtest(res)` returns a logical value with the rejection decision from conducting the Engle's ARCH test for residual heteroscedasticity in the univariate residual series `res`.

`h = archtest(res,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

- If any `Name,Value` pair argument is a vector, then all `Name,Value` pair arguments that you specify must be vectors of equal length or scalars. `archtest(res,Name,Value)` treats each element of a vector input as a separate test, and returns a vector of rejection decisions.
- If any `Name,Value` pair argument is a row vector, then `archtest(res,Name,Value)` returns row vectors.

`[h,pValue] = archtest( ___ )` returns the rejection decision and *p*-value for the hypothesis test, using any of the input arguments in the previous syntaxes.

`[h,pValue,stat,cValue] = archtest( ___ )` additionally returns the test statistic (`stat`) and critical value (`cValue`) for the hypothesis test.

Examples

Test a Time Series for ARCH Effects

Load the Deutschmark/British pound foreign-exchange rate data set.

```
load Data_MarkPound
```

Convert the prices to returns.

```
returns = price2ret(Data);
```

Compute the deviations of the return series.

```
residuals = returns - mean(returns);
```

Test the return series for ARCH effects using the residuals.

```
h = archtest(residuals)
```

```
h =

     1
```

The result `h = 1` indicates that you should reject null hypothesis of no conditional heteroscedasticity and conclude that there are significant ARCH effects in the return series.

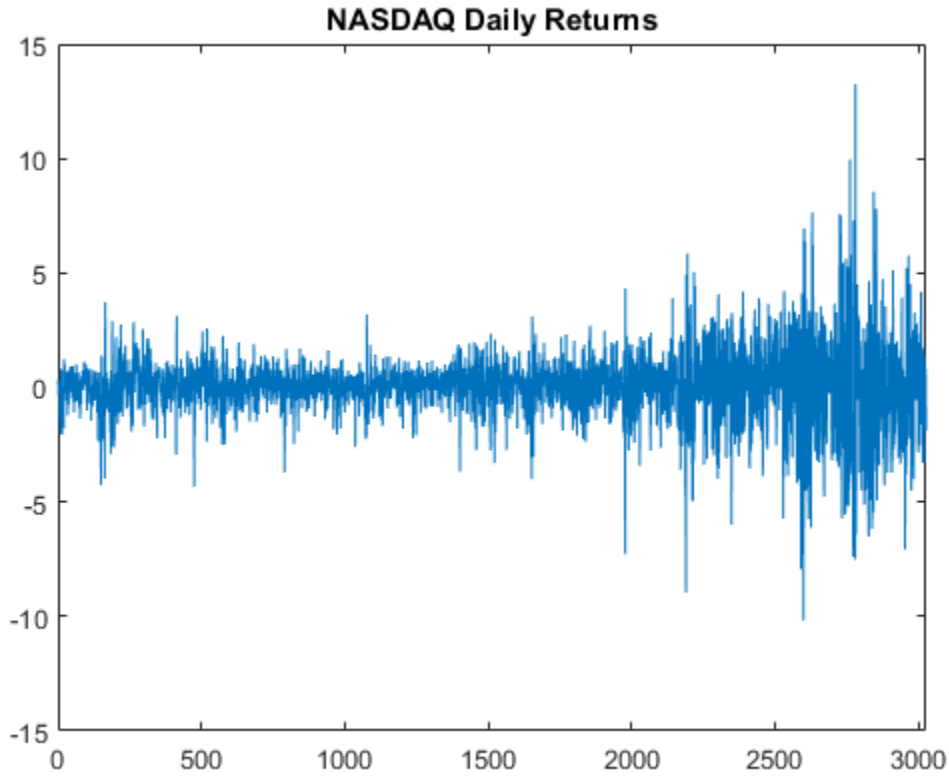
Specify the Lag Structure in an ARCH Test

To draw valid inferences from Engle's ARCH test, you should determine a suitable number of lags for the model. Do this by fitting the model over a range of plausible lags, and comparing the fitted models. Choose the number of lags that yields the best fitting model for the ARCH test.

Load and Process the Data

Load the NASDAQ data included in the toolbox. Convert the daily close composite index series to a percentage return series.

```
load Data_EquityIdx;
price = DataTable.NASDAQ;
ret = 100*price2ret(price);
T = length(ret);

figure
plot(ret)
xlim([0,T])
title('NASDAQ Daily Returns')
```



The last quarter of the return series seems to have higher variance than the first three quarters. This volatile behavior indicates conditional heteroscedasticity. Also, the series seems to fluctuate at a constant level.

The returns are of relatively high frequency. Therefore, the daily changes can be small. For numerical stability, it is good practice to scale such data.

Determine a Suitable Number of Lags

Fit the model over a grid of lags. Choose the number of lags that corresponds to the best fitting model.

```
numLags = 4;
logL = zeros(numLags,1); % Preallocate fit statistics
```

```
for k = 1:numLags
    Mdl = garch(0,k); % Specify garch model
    [~,~,logL(k)] = estimate(Mdl,ret,'Display','off'); % Obtain loglikelihood
end

fitStats = aicbic(logL,1:numLags); % Get AIC
lags = find(min(fitStats)) % Obtain suitable number of lags
```

```
lags =

     1
```

`lags = 1` indicates that it is reasonable to conduct the ARCH test using one lag.

Conduct the ARCH Test

Calculate the residuals, and use them to conduct the ARCH test at a 1% significance level.

```
r = ret - mean(ret); % Returns fluctuate at constant level
[h,pValue,stat,cValue] = archtest(ret,'Lags',lags,'Alpha',0.01)
```

```
h =

     1


pValue =

     0


stat =

  204.2625


cValue =

    6.6349
```

`h = 1` indicates that the software rejects the null hypothesis of no ARCH effects. `pValue = 0` indicates that the evidence is strong for the rejection of the null.

- "Time Series Regression VI: Residual Diagnostics"
- "Detect ARCH Effects" on page 3-28

Input Arguments

`res` — Residual series

vector

Residual series for which the software computes the test statistic, specified as a vector. The last element corresponds to the most recent observation.

Typically, you fit a model to an observed time series, and `res` is the (standardized) residuals from the fitted model.

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'lags',1:4,'alpha',0.1` specifies four tests with 1, 2, 3, and 4 lagged terms conducted at the 0.1 significance level.

`'lags'` — Number of lagged terms

1 (default) | positive integer | vector of positive integers

Number of lagged terms to include in the test statistic calculation, specified as the comma-separated pair consisting of `'lags'` and a positive integer or vector of positive integers.

Use a vector to conduct multiple tests.

Each element of `lags` must be less than `length(res) – 1`.

Example: `'lags',1:4`

`'alpha'` — Significance levels

0.05 (default) | scalar | vector

Significance levels for the hypothesis tests, specified as the comma-separated pair consisting of `'alpha'` and a scalar or vector.

Use a vector to conduct multiple tests.

Each element of `alpha` must be greater than 0 and less than 1.

Example: `'alpha',0.01`

Output Arguments

`h` — Test rejection decisions

logical | vector of logicals

Test rejection decisions, returned as a logical value or vector of logical values with length equal to the number of tests that the software conducts.

- `h = 1` indicates rejection of the no ARCH effects null hypothesis in favor of the alternative.
- `h = 0` indicates failure to reject the no ARCH effects null hypothesis.

`pValue` — Test statistic *p*-values

scalar | vector

Test statistic *p*-values, returned as a scalar or vector with length equal to the number of tests that the software conducts.

`stat` — Test statistics

scalar | vector

Test statistics, returned as a scalar or vector with length equal to the number of tests that the software conducts.

`cValue` — Critical values

scalar | vector

Critical values, returned as a scalar or vector with length equal to the number of tests that the software conducts.

More About

Engle's ARCH Test

Engle's ARCH test assesses the null hypothesis that a series of residuals (r_t) exhibits no conditional heteroscedasticity (ARCH effects), against the alternative that an ARCH(L) model describes the series.

The ARCH(L) model has the following form:

$$r_t^2 = a_0 + a_1 r_{t-1}^2 + \ldots + a_L r_{t-L}^2 + e_t,$$

where there is at least one $a_j \neq 0$, $j = 0,..,L$.

The test statistic is the Lagrange multiplier statistic TR^2, where:

- T is the sample size.
- R^2 is the coefficient of determination from fitting the ARCH(L) model for a number of lags (L) via regression.

Under the null hypothesis, the asymptotic distribution of the test statistic is chi-square with L degrees of freedom.

Tips

- You must determine a suitable number of lags to draw valid inferences from Engle's ARCH test. One method is to:

  1. Fit a sequence of arima, `garch`, `egarch`, or `gjr` models using `estimate`. Restrict each model by specifying progressively smaller ARCH lags (i.e., ARCH effects corresponding to increasingly smaller lag polynomial terms).
  2. Obtain loglikelihoods from the estimated models.
  3. Use `lratiotest` to evaluate the significance of each restriction. Alternatively, determine information criteria using `aicbic` and combine them with measures of fit.

- Residuals in an ARCH process are dependent, but not correlated. Thus, `archtest` tests for heteroscedasticity without autocorrelation. To test for autocorrelation, use `lbqtest`.

- GARCH(P,Q) processes are locally equivalent to ARCH($P + Q$) processes. If `archtest(res,'lags',lags)` shows evidence of conditional heteroscedasticity in residuals from a mean model, then it might be better to model a GARCH(P,Q) model with $P + Q =$ `lags`.

- "Engle's ARCH Test" on page 3-25

References

[1] Box, G. E. P., G.M. Jenkins, and G.C. Reinsel. Time Series Analysis: Forecasting and Control. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

[2] Engle, R. "Autoregressive Conditional Heteroscedasticity with Estimates of the Variance of United Kingdom Inflation." Econometrica. Vol. 96, 1988, pp. 893–920.

See Also

`aicbic` | `autocorr` | estimate | `estimate` | `lbqtest` | `lratiotest`

Introduced before R2006a

arima class

Create ARIMA or ARIMAX time series model

Description

`arima` creates model objects for stationary or unit root nonstationary linear time series model. This includes moving average (MA), autoregressive (AR), mixed autoregressive and moving average (ARMA), integrated (ARIMA), multiplicative seasonal, and linear time series models that include a regression component (ARIMAX).

Specify models with known coefficients, estimate coefficients with data using `estimate`, or simulate models with `simulate`. By default, the variance of the innovations is a positive scalar, but you can specify any supported conditional variance model, such as a GARCH model.

Construction

`Mdl = arima` creates an ARIMA model of degrees zero.

`Mdl = arima(p,D,q)` creates a nonseasonal linear time series model using autoregressive degree `p`, differencing degree `D`, and moving average degree `q`.

`Mdl = arima(Name,Value)` creates a linear time series model using additional options specified by one or more `Name,Value` pair arguments. `Name` is the property name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Input Arguments

Note: You can only use these arguments for nonseasonal models. For seasonal models, use the name-value syntax.

p

Positive integer indicating the degree of the nonseasonal autoregressive polynomial.

D

Nonnegative integer indicating the degree of nonseasonal integration in the linear time series.

q

Positive integer indicating the degree of the nonseasonal moving average polynomial.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

'AR'

Cell vector of nonseasonal autoregressive coefficients corresponding to a stable polynomial. When specified without `ARLags`, `AR` is a cell vector of coefficients at lags 1,2,... to the degree of the nonseasonal autoregressive polynomial. When specified with `ARLags`, `AR` is an equivalent-length cell vector of coefficients associated with the lags in `ARLags`.

Default: Cell vector of `NaN`s.

'ARLags'

Vector of positive integer lags associated with the `AR` coefficients.

Default: Vector of integers 1,2,... to the degree of the nonseasonal autoregressive polynomial.

'Beta'

Real vector of coefficients corresponding to the regression component in an ARIMAX conditional mean model.

Default: `[ ]` (no regression coefficients corresponding to a regression component)

`'Constant'`

Scalar constant in the linear time series.

Default: `NaN`

`'D'`

Nonnegative integer indicating the degree of the nonseasonal differencing lag operator polynomial (the degree of nonseasonal integration) in the linear time series.

Default: `0` (no nonseasonal integration)

`'Distribution'`

Conditional probability distribution of the innovation process. `Distribution` is a string you specify as `'Gaussian'` or `'t'`. Alternatively, specify it as a data structure with the field `Name` to store the distribution `'Gaussian'` or `'t'`. If the distribution is `'t'`, then the structure also needs the field `DoF` to store the degrees of freedom.

Default: `'Gaussian'`

`'MA'`

Cell vector of nonseasonal moving average coefficients corresponding to an invertible polynomial. When specified without `MALags`, `MA` is a cell vector of coefficients at lags 1,2,... to the degree of the nonseasonal moving average polynomial. When specified with `MALags`, `MA` is an equivalent-length cell vector of coefficients associated with the lags in `MALags`.

Default: Cell vector of `NaN`s.

`'MALags'`

Vector of positive integer lags associated with the `MA` coefficients.

Default: Vector of integers 1,2,... to the degree of the nonseasonal moving average polynomial.

`'SAR'`

Cell vector of seasonal autoregressive coefficients corresponding to a stable polynomial. When specified without `SARLags`, `SAR` is a cell vector of coefficients at lags 1,2,... to the

degree of the seasonal autoregressive polynomial. When specified with SARLags, SAR is an equivalent-length cell vector of coefficients associated with the lags in SARLags.

Default: Cell vector of NaNs.

'SARLags'

Vector of positive integer lags associated with the SAR coefficients.

Default: Vector of integers 1,2,... to the degree of the seasonal autoregressive polynomial.

'SMA'

Cell vector of seasonal moving average coefficients corresponding to an invertible polynomial. When specified without SMALags, SMA is a cell vector of coefficients at lags 1,2,... to the degree of the seasonal moving average polynomial. When specified with SMALags, SMA is an equivalent-length cell vector of coefficients associated with the lags in SMALags.

Default: Cell vector of NaNs.

'SMALags'

Vector of positive integer lags associated with the SMA coefficients.

Default: Vector of integers 1,2,... to the degree of the seasonal moving average polynomial.

'Seasonality'

Nonnegative integer indicating the degree of the seasonal differencing lag operator polynomial in the linear time series model.

Default: 0 (no seasonal integration)

'Variance'

Positive scalar variance of the model innovations, or a supported conditional variance model object (e.g., a garch model object).

Default: NaN

Notes

- Each AR, SAR, MA, and SMA coefficient is associated with an underlying lag operator polynomial and is subject to a near-zero tolerance exclusion test. That is, the software

  compares each coefficient to the default lag operator zero tolerance, `1e-12`. If the magnitude of a coefficient is greater than `1e-12`, then the software includes it in the model. Otherwise, the software considers the coefficient sufficiently close to 0, and excludes it from the model. For additional details, see LagOp.
- Specify the lags associated with the seasonal polynomials SAR and SMA in the periodicity of the observed data, and not as multiples of the `Seasonality` parameter. This convention does not conform to standard Box and Jenkins [1] notation, but it is a more flexible approach for incorporating multiplicative seasonality.

Properties

AR

Cell vector of nonseasonal autoregressive coefficients corresponding to a stable polynomial. Associated lags are 1,2,... to the degree of the nonseasonal autoregressive polynomial, or as specified in `ARLags`.

Beta

Real vector of regression coefficients corresponding to a regression component.

Constant

Scalar constant in the linear time series model.

D

Nonnegative integer indicating the degree of nonseasonal integration in the linear time series.

Distribution

Data structure for the conditional probability distribution of the innovation process. The field `Name` stores the distribution name `'Gaussian'` or `'t'`. If the distribution is `'t'`, then the structure also has the field `DoF` to store the degrees of freedom.

MA

Cell vector of nonseasonal moving average coefficients corresponding to an invertible polynomial. Associated lags are 1,2,... to the degree of the nonseasonal moving average polynomial, or as specified in `MALags`.

P

Degree of the compound autoregressive polynomial. `P` is the total number of lagged observations necessary to initialize the autoregressive component of the model.

`P` includes the effects of nonseasonal and seasonal integration captured by the properties `D` and `Seasonality`, respectively, and the nonseasonal and seasonal autoregressive polynomials `AR` and `SAR`, respectively.

The property `P` does not necessarily conform to standard Box and Jenkins notation. It only conforms if the model has no integration nor seasonal autoregressive component.

Q

Degree of the compound moving average polynomial. `Q` is the total number of lagged innovations necessary to initialize the moving average component of the model. `Q` includes the effects of nonseasonal and seasonal moving average polynomials `MA` and `SMA`, respectively.

The property `Q` does not necessarily conform to standard Box and Jenkins notation. It only conforms if the model has no seasonal moving average component.

SAR

Cell vector of seasonal autoregressive coefficients corresponding to a stable polynomial. Associated lags are 1,2,... to the degree of the seasonal autoregressive polynomial, or as specified in `SARLags`.

SMA

Cell vector of seasonal moving average coefficients corresponding to an invertible polynomial. Associated lags are 1,2,... to the degree of the seasonal moving average polynomial, or as specified in `SMALags`.

Seasonality

Nonnegative integer indicating the seasonal differencing polynomial degree in the linear time series model.

`Variance`

Positive scalar variance of the model innovations, or a supported conditional variance model (e.g., a `garch` model).

Methods

estimate	Estimate ARIMA or ARIMAX model parameters
filter	Filter disturbances using ARIMA or ARIMAX model
forecast	Forecast ARIMA or ARIMAX process
impulse	Impulse response function
infer	Infer ARIMA or ARIMAX model residuals or conditional variances
print	Display parameter estimation results for ARIMA or ARIMAX models
simulate	Monte Carlo simulation of ARIMA or ARIMAX models

Definitions

Lag Operator

The lag operator L is defined as $L^i y_t = y_{t-i}$. You can create lag operator polynomials using them to condense the notation and solve linear difference equations. The lag operator polynomials in the linear time series model definitions are:

- $\phi(L) = 1 - \phi L - \phi^2 L^2 - \ldots - \phi^p L^p$, which is the degree p autoregressive polynomial.
- $\theta(L) = 1 + \theta L + \theta^2 L^2 + \ldots + \theta^q L^q$, which is the degree q moving average polynomial.
- $\Phi(L) = 1 - \Phi_{p_1} L^{p_1} - \Phi_{p_2} L^{p_2} - \ldots - \Phi_{p_s} L^{p_s}$, which is the degree p_s seasonal autoregressive polynomial.

- $\Theta(L) = 1 + \Theta_{q_1} L^{q_1} + \Theta_{q_2} L^{q_2} + \ldots + \Theta_{q_s} L^{q_s}$, which is the degree q_s seasonal moving average polynomial.

Note: The degrees of the lag operators in the seasonal polynomials $\Phi(L)$ and $\Theta(L)$ do not conform to those defined by Box and Jenkins [1]. In other words, Econometrics Toolbox does not treat $p_1 = s$, $p_2 = 2s$,...,$p_s = c_p s$ nor $q_1 = s$, $q_2 = 2s$,...,$q_s = c_q s$ where c_p and c_q are positive integers. The software is flexible as it lets you specify the lag operator degrees. See "Multiplicative ARIMA Model Specifications" on page 5-48.

Linear Time Series Model

A linear time series model for response process y_t and innovations ε_t is a "What Is a Stochastic Process?" on page 1-20 that has the form

$$y_t = c + \phi_1 y_{t-1} + \ldots + \phi_p y_{t-p} + \varepsilon_t + \theta_1 \varepsilon_{t-1} + \ldots + \theta q \varepsilon_{t-q}.$$

In lag operator notation, this model is

$$\phi(L) y_t = c + \theta(L) \varepsilon_t.$$

The general times series model, which includes differencing, multiplicative seasonality, and seasonal differencing, is

$$\phi(L)(1-L)^D \Phi(L)(1-L^s)^{D_s} y_t = c + \theta(L)\Theta(L)\varepsilon_t.$$

- The coefficients of the nonseasonal and seasonal autoregressive polynomials $\phi(L)$ and $\Phi(L)$ correspond to `AR` and `SAR`, respectively. The degrees of these polynomials are p and p_s. Similarly, the coefficients of polynomials $\theta(L)$ and $\Theta(L)$ correspond to `MA` and `SMA`. The degrees of these polynomials are q and q_s, respectively.
- The polynomials $(1-L)^D$ and $(1-L^s)^{D_s}$ have a degree of nonseasonal and seasonal integration D and D_s, respectively. Note that s corresponds to model property `Seasonality`. D_s is 1 if `Seasonality` is nonzero, and it is 0 otherwise. That is, the software applies first-order seasonal differencing if `Seasonality` ≥ 1.
- The model property `P` is equal to $p + D + p_s + D_s$.

- The model property Q is equal to $q + q_s$.
- You can extend this model by including a matrix of predictor data. For details, see "ARIMA Model Including Exogenous Covariates" on page 5-58.

Stationarity Requirements

The ARMA(p,q) model,

$$\phi(L)y_t = c + \theta(L)\varepsilon_t,$$

where ε_t has mean 0, variance σ^2, and $Cov(\varepsilon_t, \varepsilon_s) = 0$ for $t \neq s$, is stationary if its expected value, variance, and covariance between elements of the series are independent of time. For example, the MA(q) model, with $c = 0$, is stationary for any $q < \infty$ because

- $E(y_t) = \theta(L)0 = 0,$
- $Var(y_t) = \sigma^2 \sum_{i=1}^{q} \theta_i^2,$ and
- $$Cov(y_t, y_{t-s}) = \begin{cases} \sigma^2(\theta_s + \theta_1\theta_{s-1} + \theta_2\theta_{s-2} + \ldots + \theta_q\theta_{s-q}) \text{ if } s \geq q \\ 0 \text{ otherwise.} \end{cases}$$

are free of t for all time points [1].

Unit Root

The time series $\{y_t; t = 1,\ldots,T\}$ is a unit root process if its expected value, variance, or covariance grows with time. Subsequently, the time series is not stationary.

References

[1] Box, G. E. P., G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis: Forecasting and Control*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

[2] Enders, W. *Applied Econometric Time Series*. Hoboken, NJ: John Wiley & Sons, Inc., 1995.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Specify a Nonseasonal ARIMA Model

Specify an ARIMA(2,1,2) model,

$$(1 - \phi_1 L - \phi_2 L^2)(1 - L)y_t = (1 + \theta_1 L + \theta_2 L^2)\varepsilon_t$$

```
Mdl = arima(2,1,2)
```

```
Mdl =

    ARIMA(2,1,2) Model:
    --------------------
    Distribution: Name = 'Gaussian'
               P: 3
               D: 1
               Q: 2
        Constant: NaN
              AR: {NaN NaN} at Lags [1 2]
             SAR: {}
              MA: {NaN NaN} at Lags [1 2]
             SMA: {}
        Variance: NaN
```

The model is nonseasonal, so you can use shorthand syntax. The result is a model with two nonseasonal AR coefficients ($p = 2$), two nonseasonal MA coefficients ($q = 2$), and one degree of differencing ($D = 1$). The property `P` is equal to $p + D = 3$. `NaN` values indicate estimable parameters.

Modify an ARIMA Model Object

Create, and then modify an `arima` model.

Specify an AR(3) model with known coefficients,

$$y_t = 0.05 + 0.6y_{t-1} + 0.2y_{t-2} - 0.1y_{t-3} + \varepsilon_t,$$

where ε_t has a Gaussian distribution with mean 0 and variance 0.01.

```
Mdl = arima('Constant',0.05,'AR',{0.6,0.2,-0.1},...
 'Variance',0.01)
```

```
Mdl =

    ARIMA(3,0,0) Model:
    --------------------
    Distribution: Name = 'Gaussian'
               P: 3
               D: 0
               Q: 0
        Constant: 0.05
              AR: {0.6 0.2 -0.1} at Lags [1 2 3]
             SAR: {}
              MA: {}
             SMA: {}
        Variance: 0.01
```

Modify the model object to make all the model parameters unknown (set them to `NaN`).

```
Mdl.Constant = NaN;
Mdl.AR{1:3} = NaN;
Mdl.Variance = NaN
```

```
Mdl =

    ARIMA(3,0,0) Model:
    --------------------
    Distribution: Name = 'Gaussian'
               P: 3
               D: 0
               Q: 0
        Constant: NaN
              AR: {NaN NaN NaN} at Lags [1 2 3]
             SAR: {}
              MA: {}
             SMA: {}
```

```
        Variance: NaN
```

Make the innovation distribution a t distribution with 10 degrees of freedom.

```
tdist = struct('Name','t','DoF',10);
Mdl.Distribution = tdist
```

```
Mdl =

    ARIMA(3,0,0) Model:
    --------------------
    Distribution: Name = 't', DoF = 10
               P: 3
               D: 0
               Q: 0
        Constant: NaN
              AR: {NaN NaN NaN} at Lags [1 2 3]
             SAR: {}
              MA: {}
             SMA: {}
        Variance: NaN
```

Specify an Additive Seasonal ARIMA Model

Specify an MA model with no constant, and moving average terms at lags 1, 2, and 12,

$$y_t = \varepsilon_t + \theta_1\varepsilon_{t-1} + \theta_2\varepsilon_{t-2} + \theta_{12}\varepsilon_{t-12}.$$

```
Mdl = arima('Constant',0,'MALags',[1,2,12])
```

```
Mdl =

    ARIMA(0,0,12) Model:
    --------------------
    Distribution: Name = 'Gaussian'
               P: 0
               D: 0
               Q: 12
        Constant: 0
              AR: {}
             SAR: {}
              MA: {NaN NaN NaN} at Lags [1 2 12]
             SMA: {}
```

```
        Variance: NaN
```

Specify a Multiplicative Seasonal Model

Specify a multiplicative seasonal ARIMA model with seasonal and nonseasonal integration,

$$(1-L)(1-L^{12})y_t = (1+\theta_1)(1+\theta_{12}L^{12})\varepsilon_t.$$

```
Mdl = arima('Constant',0,'D',1,'Seasonality',12,...
 'MALags',1,'SMALags',12)
```

```
Mdl =

    ARIMA(0,1,1) Model Seasonally Integrated with Seasonal MA(12):
    --------------------------------------------------------------
    Distribution: Name = 'Gaussian'
               P: 13
               D: 1
               Q: 13
        Constant: 0
              AR: {}
             SAR: {}
              MA: {NaN} at Lags [1]
             SMA: {NaN} at Lags [12]
     Seasonality: 12
        Variance: NaN
```

An ARIMA model is assumed to be multiplicative any time `SMALags` or `SARLags` are specified.

Specify an ARIMAX Model

Specify an ARIMAX model with one or more regression coefficients corresponding to predictor data.

Specify the ARIMAX(1,1,1) model,

$$(1-0.2L)(1-L)^1 y_t = 0.5x_t + (1+0.3L)\varepsilon_t$$

using `arima`.

```
Mdl = arima('AR',0.2,'D',1,'MA',0.3,'Beta',0.5)
```

```
Mdl =

    ARIMAX(1,1,1) Model:
    --------------------
    Distribution: Name = 'Gaussian'
               P: 2
               D: 1
               Q: 1
        Constant: NaN
              AR: {0.2} at Lags [1]
             SAR: {}
              MA: {0.3} at Lags [1]
             SMA: {}
            Beta: [0.5]
        Variance: NaN
```

In the output, the property `P` is sum of the AR lags and degree of nonseasonal integration `p + D = 2`.

Modify this ARIMAX(1,1,1) model by adding two more regression coefficients,

```
Mdl.Beta=[0.5,4,-0.6]
```

```
Mdl =

    ARIMAX(1,1,1) Model:
    --------------------
    Distribution: Name = 'Gaussian'
               P: 2
               D: 1
               Q: 1
        Constant: NaN
              AR: {0.2} at Lags [1]
             SAR: {}
              MA: {0.3} at Lags [1]
             SMA: {}
            Beta: [0.5 4 -0.6]
        Variance: NaN
```

Specify a Composite Conditional Variance Model

Specify an ARIMA(1,0,1) conditional mean model with a GARCH(1,1) conditional variance model.

Specify the conditional mean model.

```
Mdl = arima(1,0,1);
```

Specify the conditional variance model.

```
Mdl.Variance = garch(1,1)
```

```
Mdl =

    ARIMA(1,0,1) Model:
    --------------------
    Distribution: Name = 'Gaussian'
               P: 1
               D: 0
               Q: 1
        Constant: NaN
              AR: {NaN} at Lags [1]
             SAR: {}
              MA: {NaN} at Lags [1]
             SMA: {}
        Variance: [GARCH(1,1) Model]
```

- "Modify Properties of Conditional Mean Model Objects" on page 5-65
- "Specify Conditional Mean Model Innovation Distribution" on page 5-72
- "AR Model Specifications" on page 5-21
- "MA Model Specifications" on page 5-29
- "ARMA Model Specifications" on page 5-37
- "ARIMA Model Specifications" on page 5-43
- "ARIMAX Model Specifications" on page 5-61
- "Multiplicative ARIMA Model Specifications" on page 5-48
- "Multiplicative ARIMA Model Specifications" on page 5-48
- "Specify Multiplicative ARIMA Model" on page 5-52
- "Specify Conditional Mean and Variance Models" on page 5-79

See Also

estimate | filter | forecast | impulse | infer | print | simulate

More About

- “Specify Conditional Mean Models Using arima” on page 5-6
- “Stochastic Process Characteristics” on page 1-20
- “Conditional Mean Models” on page 5-3
- “Autoregressive Model” on page 5-18
- “Moving Average Model” on page 5-27
- “Autoregressive Moving Average Model” on page 5-34
- “ARIMA Model” on page 5-41
- “ARIMA Model Including Exogenous Covariates” on page 5-58
- “Multiplicative ARIMA Model” on page 5-46
- Using garch Objects
- Using egarch Objects
- Using gjr Objects

arima

Class: regARIMA

Convert regression model with ARIMA errors to ARIMAX model

Syntax

```
ARIMAX = arima(Mdl)
[ARIMAX,XNew] = arima(Mdl,Name,Value)
```

Description

`ARIMAX = arima(Mdl)` converts the univariate regression model with ARIMA time series errors `Mdl` to a model of type arima including a regression component (ARIMAX).

`[ARIMAX,XNew] = arima(Mdl,Name,Value)` returns an updated regression matrix of predictor data using additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

`Mdl`

Regression model with ARIMA time series errors, as created by regARIMA or estimate.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'X'`

Predictor data for the regression component of `Mdl`, specified as the comma-separated pair consisting of `'X'` and a matrix.

The last row of `X` contains the latest observation of each series.

Each column of `X` is a separate time series.

Output Arguments

`ARIMAX`

ARIMAX model equivalent to the regression model with ARIMA errors `Mdl`, returned as a model of type `arima`.

`XNew`

Updated predictor data matrix for the regression component of `ARIMAX`, returned as a matrix.

`XNew` has the same number of rows as `X`. The last row of `XNew` contains the latest observation of each series.

Each column of `XNew` is a separate time series. The number of columns of `XNew` is one plus the number of nonzero autoregressive coefficients in the difference equation of `Mdl`.

Examples

Convert a Regression Model with ARMA Errors to an ARIMAX Model

Convert a regression model with ARMA(4,1) errors to an ARIMAX model using the `arima` converter.

Specify the regression model with ARMA(4,1) errors:

$$\begin{aligned} y_t &= 1 + 0.5X_t + u_t \\ u_t &= 0.8u_{t-1} - 0.4u_{t-4} + \varepsilon_t + 0.3\varepsilon_{t-1}, \end{aligned}$$

where ε_t is Gaussian with mean 0 and variance 1.

```
Mdl = regARIMA('AR',{0.8, -0.4},'MA',0.3,...
    'ARLags',[1 4],'Intercept',1,'Beta',0.5,...
    'Variance',1)
```

```
Mdl =

    Regression with ARIMA(4,0,1) Error Model:
    ----------------------------------------
    Distribution: Name = 'Gaussian'
       Intercept: 1
            Beta: [0.5]
               P: 4
               D: 0
               Q: 1
              AR: {0.8 -0.4} at Lags [1 4]
             SAR: {}
              MA: {0.3} at Lags [1]
             SMA: {}
        Variance: 1
```

You can verify that the lags of the autoregressive terms are 1 and 4 in the AR row.

Generate random predictor data.

```
rng(1); % For reproducibility
T = 20;
X = randn(T,1);
```

Convert `Mdl` to an ARIMAX model.

```
[ARIMAX,XNew] = arima(Mdl,'X',X);
ARIMAX
```

```
ARIMAX =

    ARIMAX(4,0,1) Model:
    --------------------
    Distribution: Name = 'Gaussian'
               P: 4
               D: 0
               Q: 1
        Constant: 0.6
              AR: {0.8 -0.4} at Lags [1 4]
             SAR: {}
              MA: {0.3} at Lags [1]
             SMA: {}
            Beta: [1 -0.8 0.4]
```

```
    Variance: 1
```

The new `arima` model, `ARIMAX`, is

$$y_t = 0.6 + Z_t\Gamma + 0.8y_{t-1} - 0.4y_{t-4} + \varepsilon_t + 0.3\varepsilon_{t-1},$$

where

$$Z_t\Gamma = \begin{bmatrix} 0.5x_1 & NaN & NaN \\ 0.5x_2 & 0.5x_1 & NaN \\ 0.5x_3 & 0.5x_2 & NaN \\ 0.5x_4 & 0.5x_3 & NaN \\ 0.5x_5 & 0.5x_4 & 0.5x_1 \\ \vdots & \vdots & \vdots \\ 0.5_T & 0.5x_{T-1} & 0.5x_{T-4} \end{bmatrix} \begin{bmatrix} 1 \\ -0.8 \\ 0.4 \end{bmatrix}$$

and x_j is row j of `X`. Since the product of the autoregressive and integration polynomials is $\phi(L) = (1 - 0.8L + 0.4L^4)$, `ARIMAX.Beta` is simply `[1 -0.8 0.4]`. Note that the software carries over the autoregressive and moving average coefficients from `Mdl` to `ARIMAX`. Also, `Mdl.Intercept` = 1 and `ARIMAX.Constant` = (1 - 0.8 + 0.4)(1) = 0.6, i.e., the `regARIMA` model intercept and `arima` model constant are generally unequal.

Convert a Regression Model with ARIMA Errors to an ARIMAX Model

Convert a regression model with seasonal ARIMA errors to an ARIMAX model using the `arima` converter.

Specify the regression model with $\text{ARIMA}(2,1,1) \times (1,1,0)_2$ errors:

$$y_t = X_t \begin{bmatrix} -2 \\ 1 \end{bmatrix} + u_t$$
$$(1 - 0.3L + 0.15L^2)(1 - L)(1 - 0.2L^2)(1 - L^2)u_t = (1 + 0.1L)\varepsilon_t,$$

where ε_t is Gaussian with mean 0 and variance 1.

```
Mdl = regARIMA('AR',{0.3, -0.15},'MA',0.1,...
    'ARLags',[1 2],'SAR',0.2,'SARLags',2,...
    'Intercept',0,'Beta',[-2; 1],'Variance',1,'D',1,...
```

```
    'Seasonality',2)

Mdl =

    Regression with ARIMA(2,1,1) Error Model Seasonally Integrated with Seasonal AR(2)
    ------------------------------------------------------------------------------------
    Distribution: Name = 'Gaussian'
       Intercept: 0
            Beta: [-2 1]
               P: 7
               D: 1
               Q: 1
              AR: {0.3 -0.15} at Lags [1 2]
             SAR: {0.2} at Lags [2]
              MA: {0.1} at Lags [1]
             SMA: {}
     Seasonality: 2
        Variance: 1
```

Generate predictor data.

```
rng(1); % For reproducibility
T = 20;
X = randn(T,2);
```

Convert `Mdl` to an ARIMAX model.

```
[ARIMAX,XNew] = arima(Mdl,'X',X);
ARIMAX

ARIMAX =

    ARIMAX(2,1,1) Model Seasonally Integrated with Seasonal AR(2):
    --------------------------------------------------------------
    Distribution: Name = 'Gaussian'
               P: 7
               D: 1
               Q: 1
        Constant: 0
              AR: {0.3 -0.15} at Lags [1 2]
             SAR: {0.2} at Lags [2]
              MA: {0.1} at Lags [1]
             SMA: {}
```

```
        Beta: [1 -1.3 -0.75 1.41 -0.34 -0.08 0.09 -0.03]
Seasonality: 2
   Variance: 1
```

`Mdl.Beta` has length 2, but `ARIMAX.Beta` has length 8. This is because the product of the autoregressive and integration polynomials, $\phi(L)(1-L)\Phi(L)(1-L^s)$, is

$$1 - 1.3L - 0.75L^2 + 1.41L^3 - 0.34L^4 - 0.08L^5 + 0.09L^6 - 0.03L^7.$$

You can see that when you add seasonality, seasonal lag terms, and integration to a model, the size of `XNew` can grow quite large. A conversion such as this might not be ideal for analyses involving small sample sizes.

Algorithms

Let X denote the matrix of concatenated predictor data vectors (or design matrix) and β denote the regression component for the regression model with ARIMA errors, `Mdl`.

- If you specify `X`, then `arima` returns `XNew` in a certain format. Suppose that the nonzero autoregressive lag term degrees of `Mdl` are $0 < a_1 < a_2 < ... < P$, which is the largest lag term degree. The software obtains these lag term degrees by expanding and reducing the product of the seasonal and nonseasonal autoregressive lag polynomials, and the seasonal and nonseasonal integration lag polynomials

  $$\phi(L)(1-L)^D\Phi(L)(1-L^s).$$

  - The first column of `XNew` is $X\beta$.
  - The second column of `XNew` is a sequence of a_1 `NaN`s, and then the product $X_{a_1}\beta$, where $X_{a_1}\beta = L^{a_1}X\beta$.
  - The jth column of `XNew` is a sequence of a_j `NaN`s, and then the product $X_{a_j}\beta$, where $X_{a_j}\beta = L^{a_j}X\beta$.
  - The last column of `XNew` is a sequence of a_p `NaN`s, and then the product $X_p\beta$, where $X_p\beta = L^pX\beta$.

Suppose that `Mdl` is a regression model with ARIMA(3,1,0) errors, and $\phi_1 = 0.2$ and $\phi_3 = 0.05$. Then the product of the autoregressive and integration lag polynomials is

$$(1 - 0.2L - 0.05L^3)(1 - L) = 1 - 1.2L + 0.02L^2 - 0.05L^3 + 0.05L^4.$$

This implies that `ARIMAX.Beta` is `[1 -1.2 0.02 -0.05 0.05]` and `XNew` is

$$\begin{bmatrix} x_1\beta & \texttt{NaN} & \texttt{NaN} & \texttt{NaN} & \texttt{NaN} \\ x_2\beta & x_1\beta & \texttt{NaN} & \texttt{NaN} & \texttt{NaN} \\ x_3\beta & x_2\beta & x_1\beta & \texttt{NaN} & \texttt{NaN} \\ x_4\beta & x_3\beta & x_2\beta & x_1\beta & \texttt{NaN} \\ x_5\beta & x_4\beta & x_3\beta & x_2\beta & x_1\beta \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ x_T\beta & x_{T-1}\beta & x_{T-2}\beta & x_{T-3}\beta & x_{T-4}\beta \end{bmatrix},$$

where x_j is the jth row of X.

- If you do not specify `X`, then `arima` returns `XNew` as an empty matrix without rows and one plus the number of nonzero autoregressive coefficients in the difference equation of `Mdl` columns.

See Also

arima | estimate | regARIMA

More About

- "Time Series Regression Models" on page 4-3
- "Compare Alternative ARIMA Model Representations" on page 4-136

arma2ar

Convert ARMA model to AR model

Syntax

```
ar = arma2ar(ar0,ma0)
ar = arma2ar(ar0,ma0,numLags)
```

Description

`ar = arma2ar(ar0,ma0)` returns the coefficients of the truncated, infinite-order AR model approximation to an ARMA model having AR and MA coefficients specified by `ar0` and `ma0`, respectively.

`arma2ar`:

- Accepts:
  - Vectors or cell vectors of matrices in difference-equation notation.
  - LagOp lag operator polynomials corresponding to the AR and MA polynomials in lag operator notation.
- Accommodates time series models that are univariate or multivariate (i.e., `numVars` variables compose the model), stationary or integrated, structural or in reduced form, and invertible.
- Assumes that the model constant c is 0.

`ar = arma2ar(ar0,ma0,numLags)` returns the first nonzero `numLags` lag-term coefficients of the infinite-order AR model approximation of an ARMA model having AR coefficients `ar0` and MA coefficients `ma0`.

Examples

Convert an ARMA model to an AR Model

Find the lag coefficients of the truncated, AR approximation of this univariate, stationary, and invertible ARMA model

$$y_t = 0.2y_{t-1} - 0.1y_{t-2} + \varepsilon_t + 0.5\varepsilon_{t-1}.$$

The ARMA model is in difference-equation notation because the left side contains only y_t and its coefficient 1. Create a vector containing the AR lag term coefficients in order starting from t - 1.

```
ar0 = [0.2 -0.1];
```

Alternatively, you can create a cell vector of the scalar coefficients.

Create a vector containing the MA lag term coefficient.

```
ma0 = 0.5;
```

Convert the ARMA model to an AR model by obtaining the coefficients of the truncated approximation of the infinite-lag polynomial.

```
ar = arma2ar(ar0,ma0)
```

```
ar =

    0.7000   -0.4500    0.2250   -0.1125    0.0562   -0.0281    0.0141
```

`ar` is a numeric vector because `ar0` and `ma0` are numeric vectors.

The approximate AR model truncated at 7 lags is

$$\begin{aligned} y_t &= 0.7y_{t-1} - 0.45y_{t-2} + 0.225y_{t-3} - 0.1125y_{t-4} + 0.0562y_{t-5} + \ldots \\ &\quad -0.0281y_{t-6} + 0.0141y_{t-7} + \varepsilon_t \end{aligned}$$

Convert an MA(3) Model to an AR(5) Model

Find the first five lag coefficients of the AR approximation of this univariate and invertible MA(3) model

$$y_t = \varepsilon_t - 0.2\varepsilon_{t-1} + 0.5\varepsilon_{t-3}.$$

The MA model is in difference-equation notation because the left side contains only y_t and its coefficient of 1. Create a cell vector containing the MA lag term coefficient in

order starting from t - 1. Because the second lag term of the MA model is missing, specify a `0` for its coefficient.

```
ma0 = {-0.2 0 0.5};
```

Convert the MA model to an AR model with at most five lag coefficents of the truncated approximation of the infinite-lag polynomial. Because there is no AR contribution, specify an empty cell (`{}`) for the AR coefficients.

```
numLags = 5;
ar0 = {};
ar = arma2ar(ar0,ma0,numLags)
```

```
ar =

    [-0.2000]    [-0.0400]    [0.4920]    [0.1984]    [0.0597]
```

`ar` is a cell vector of scalars because at least one of `ar0` and `ma0` is a cell vector.

The approximate AR(5) model is

$$y_t = -0.2y_{t-1} - 0.04y_{t-2} + 0.492y_{t-3} + 0.1984y_{t-4} + 0.0597y_{t-5} + \varepsilon_t$$

Convert a Structural VARMA model to a Structural VAR model

Find the coefficients of the truncated, structural VAR equivalent of the structural, stationary, and invertible VARMA model

$$\left\{\begin{bmatrix} 1 & 0.2 & -0.1 \\ 0.03 & 1 & -0.15 \\ 0.9 & -0.25 & 1 \end{bmatrix} - \begin{bmatrix} -0.5 & 0.2 & 0.1 \\ 0.3 & 0.1 & -0.1 \\ -0.4 & 0.2 & 0.05 \end{bmatrix} L^4 - \begin{bmatrix} -0.05 & 0.02 & 0.01 \\ 0.1 & 0.01 & 0.001 \\ -0.04 & 0.02 & 0.005 \end{bmatrix} L^8 \right\} y_t =$$
$$\left\{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} + \begin{bmatrix} -0.02 & 0.03 & 0.3 \\ 0.003 & 0.001 & 0.01 \\ 0.3 & 0.01 & 0.01 \end{bmatrix} L^4 \right\} \varepsilon_t$$

where $y_t = [y_{1t} \quad y_{2t} \quad y_{3t}]'$ and $\varepsilon_t = [\varepsilon_{1t} \quad \varepsilon_{2t} \quad \varepsilon_{3t}]'$.

The VARMA model is in lag operator notation because the response and innovation vectors are on opposite sides of the equation.

Create a cell vector containing the VAR matrix coefficients. Because this model is a structural model, start with the coefficient of y_t and enter the rest in order by lag. Because the equation is in lag operator notation, include the sign in front of each matrix. Construct a vector that indicates the degree of the lag term for the corresponding coefficients.

```
var0 = {[1 0.2 -0.1; 0.03 1 -0.15; 0.9 -0.25 1],...
    -[-0.5 0.2 0.1; 0.3 0.1 -0.1; -0.4 0.2 0.05],...
    -[-0.05 0.02 0.01; 0.1 0.01 0.001; -0.04 0.02 0.005]};
var0Lags = [0 4 8];
```

Create a cell vector containing the VMA matrix coefficients. Because this model is a structural model, start with the coefficient of ε_t and enter the rest in order by lag. Construct a vector that indicates the degree of the lag term for the corresponding coefficients.

```
vma0 = {eye(3),...
    [-0.02 0.03 0.3; 0.003 0.001 0.01; 0.3 0.01 0.01]};
vma0Lags = [0 4];
```

`arma2ar` requires `LagOp` lag operator polynomials for input arguments that comprise structural VAR or VMA models. Construct separate `LagOp` polynomials that describe the VAR and VMA components of the VARMA model.

```
VARLag = LagOp(var0,'Lags',var0Lags);
VMALag = LagOp(vma0,'Lags',vma0Lags);
```

`VARLags` and `VMALags` are `LagOp` lag operator polynomials that describe the VAR and VMA components of the VARMA model.

Convert the VARMA model to a VAR model by obtaining the coefficients of the truncated approximation of the infinite-lag polynomial.

```
VAR = arma2ar(VARLag,VMALag)
```

```
VAR =

    3-D Lag Operator Polynomial:
    ----------------------------
        Coefficients: [Lag-Indexed Cell Array with 4 Non-Zero Coefficients]
                Lags: [0 4 8 12]
              Degree: 12
           Dimension: 3
```

VAR is a LagOP lag operator polynomial. All coefficients except those corresponding to lags 0, 4, 8, and 12 are 3-by-3 matrices of zeros.

Convert the coefficients to difference-equation notation by reflecting the VAR lag operator polynomial around lag zero.

```
VARDiffEqn = reflect(VAR);
```

Display the nonzero coefficients of the resulting VAR models.

```
lag2Idx = VAR.Lags + 1; % Lags start at 0.  Add 1 to convert to indices.

varCoeff = toCellArray(VAR);
varDiffEqnCoeff = toCellArray(VARDiffEqn);

fprintf    ('          Lag Operator     |   Difference Equation\n')
for j = 1:numel(lag2Idx)
    fprintf('_____________________________Lag %d_____________________________\n',...
        lag2Idx(j) - 1)
    fprintf('%8.3f %8.3f %8.3f | %8.3f %8.3f %8.3f\n',...
        [varCoeff{lag2Idx(j)} varDiffEqnCoeff{lag2Idx(j)}]')
    fprintf('_____________________________________________________________\n')
end
```

```
          Lag Operator     |   Difference Equation
_____________________________Lag 0_____________________________
   1.000    0.200   -0.100 |    1.000    0.200   -0.100
   0.030    1.000   -0.150 |    0.030    1.000   -0.150
   0.900   -0.250    1.000 |    0.900   -0.250    1.000

_____________________________Lag 4_____________________________
   0.249   -0.151   -0.397 |   -0.249    0.151    0.397
  -0.312   -0.099    0.090 |    0.312    0.099   -0.090
   0.091   -0.268   -0.029 |   -0.091    0.268    0.029

_____________________________Lag 8_____________________________
   0.037    0.060   -0.012 |   -0.037   -0.060    0.012
  -0.101   -0.007    0.000 |    0.101    0.007   -0.000
  -0.033    0.029    0.114 |    0.033   -0.029   -0.114

_____________________________Lag 12____________________________
   0.014   -0.007   -0.034 |   -0.014    0.007    0.034
   0.000   -0.000   -0.001 |   -0.000    0.000    0.001
  -0.010   -0.018    0.002 |    0.010    0.018   -0.002

_____________________________________________________________
```

The coefficients of lags 4, 8, and 12 are opposites between `VAR` and `VARDiffEqn`.

Convert ARMA Model That Includes a Constant to an AR Model

Find the lag coefficients and constant of the truncated AR approximation of this univariate, stationary, and invertible ARMA model.

$$y_t = 1.5 + 0.2y_{t-1} - 0.1y_{t-2} + \varepsilon_t + 0.5\varepsilon_{t-1}.$$

The ARMA model is in difference-equation notation because the left side contains only y_t and its coefficient of 1. Create separate vectors for the AR and MA lag term coefficients in order starting from *t* - 1.

```
ar0 = [0.2 -0.1];
ma0 = 0.5;
```

Convert the ARMA model to an AR model by obtaining the first five coefficients of the truncated approximation of the infinite-lag polynimial.

```
numLags = 5;
ar = arma2ar(ar0,ma0,numLags)
```

```
ar =

    0.7000   -0.4500    0.2250   -0.1125    0.0562
```

To compute the constant of the AR model, consider the ARMA model in lag operator notation.

$$\left(1 - 0.2L + 0.1L^2\right) y_t = 1.5 + (1 + 0.5L)\varepsilon_t$$

or

$$\Phi(L)y_t = 1.5 + \Theta(L)\varepsilon_t$$

Part of the conversion involves premultiplying both sides of the equation by the inverse of the MA lag operator polynomial, as in this equation.

$$\Theta^{-1}(L)\Phi(L)y_t = \Theta^{-1}(L)1.5 + \varepsilon_t$$

To compute the inverse of MA lag operator polynomial, use the lag operator left-division object function mldivide (LagOp).

```
Theta = LagOp([1 0.5]);
ThetaInv = mldivide(Theta,1,'RelTol',1e-5);
```

`ThetaInv` is a `LagOp` lag operator polynomial.

The application of lag operator polynomials to constants results in the product of the constant with the sum of the coefficients. Apply `ThetaInv` to the ARMA model constant to obtain the AR model constant.

```
arConstant = 1.5*sum(cell2mat(toCellArray(ThetaInv)))
```

```
arConstant =

    1.0000
```

The approximate AR model is

$$y_t = 1 + 0.7y_{t-1} - 0.45y_{t-2} + 0.225y_{t-3} - 0.1125y_{t-4} + 0.0562y_{t-5} + \varepsilon_t.$$

Input Arguments

`ar0` — Autoregressive coefficients

numeric vector | cell vector of square, numeric matrices | `LagOp` lag operator polynomial object

Autoregressive coefficients of the ARMA(p,q) model, specified as a numeric vector, cell vector of square, numeric matrices, or a LagOp lag operator polynomial object. If `ar0` is a vector (numeric or cell), then the coefficient of y_t is the identity. To specify a structural AR polynomial (i.e., the coefficient of y_t is not the identity), use `LagOp` lag operator polynomials.

- For univariate time series models, `ar0` is a numeric vector, cell vector of scalars, or a one-dimensional `LagOp` lag operator polynomial. For vectors, `ar0` has length p and the elements correspond to lagged responses composing the AR polynomial in difference-equation notation. That is, `ar0(j)` or `ar0{j}` is the coefficient of y_{t-j}.
- For `numVars`-dimensional time series models, `ar0` is a cell vector of `numVars`-by-`numVars` numeric matrices or an `numVars`-dimensional `LagOp` lag operator polynomial. For cell vectors:
  - `ar0` has length p.

  - `ar0` and `ma0` must contain `numVars`-by-`numVars` matrices.
  - The elements of `ar0` correspond to the lagged responses composing the AR polynomial in difference equation notation. That is, `ar0{j}` is the coefficient matrix of y_{t-j}.
  - Row k of an AR coefficient matrix contains the AR coefficients in the equation of the variable y_k. Subsequently, column k must correspond to variable y_k, and the column and row order of all autoregressive and moving average coefficients must be consistent.
- For `LagOp` lag operator polynomials:
  - The first element of the `Coefficients` property corresponds to the coefficient of y_t (to accommodate structural models). All other elements correspond to the coefficients of the subsequent lags in the `Lags` property.
  - To construct a univariate model in reduced form, specify `1` for the first coefficient. For `numVars`-dimensional multivariate models, specify `eye(numVars)` for the first coefficient.
  - When you work from a model in difference-equation notation, negate the AR coefficients of the lagged responses to construct the lag-operator polynomial equivalent. For example, consider $y_t = 0.5y_{t-1} - 0.8y_{t-2} + \varepsilon_t - 0.6\varepsilon_{t-1} + 0.08\varepsilon_{t-2}$. The model is in difference-equation form. To convert to an AR model, enter the following into the command window.

    ```
    ar = arma2ar([0.5 -0.8], [-0.6 0.08]);
    ```

    The ARMA model written in "Lag Operator Notation" on page 9-60 is $\left(1 - 0.5L + 0.8L^2\right)y_t = \left(1 - 0.6L + 0.08L^2\right)\varepsilon_t$. The AR coefficients of the lagged responses are negated compared to the corresponding coefficients in difference-equation format. In this form, to obtain the same result, enter the following into the command window.

    ```
    ar0 = LagOp({1 -0.5 0.8});
    ma0 = LagOp({1 -0.6 0.08});
    ar = arma2ar(ar0, ma0);
    ```

It is a best practice for `ar0` to constitute a stationary or unit-root stationary (integrated) time series model.

If the ARMA model is strictly an MA model, then specify `[]` or `{}` for `ar0`.

`ma0` — Moving average coefficients

numeric vector | cell vector of square, numeric matrices | `LagOp` lag operator polynomial object

Moving average coefficients of the ARMA(p,q) model, specified as a numeric vector, cell vector of square, numeric matrices, or a LagOp lag operator polynomial object. If `ma0` is a vector (numeric or cell), then the coefficient of ε_t is the identity. To specify a structural MA polynomial (i.e., the coefficient of ε_t is not the identity), use `LagOp` lag operator polynomials.

- For univariate time series models, `ma0` is a numeric vector, cell vector of scalars, or a one-dimensional `LagOp` lag operator polynomial. For vectors, `ma0` has length q and the elements correspond to lagged innovations composing the AR polynomial in difference-equation notation. That is, `ma0(j)` or `ma0{j}` is the coefficient of ε_{t-j}.
- For `numVars`-dimensional time series models, `ma0` is a cell vector of numeric `numVars`-by-`numVars` numeric matrices or an `numVars`-dimensional `LagOp` lag operator polynomial. For cell vectors:
  - `ma0` has length q.
  - `ar0` and `ma0` must both contain `numVars`-by-`numVars` matrices.
  - The elements of `ma0` correspond to the lagged responses composing the AR polynomial in difference equation notation. That is, `ma0{j}` is the coefficient matrix of y_{t-j}.
- For `LagOp` lag operator polynomials:
  - The first element of the `Coefficients` property corresponds to the coefficient of ε_t (to accommodate structural models). All other elements correspond to the coefficients of the subsequent lags in the `Lags` property.
  - To construct a univariate model in reduced form, specify `1` for the first coefficient. For `numVars`-dimensional multivariate models, specify `eye(numVars)` for the first coefficient.

It is a best practice for `ma0` to constitute an invertible time series model.

`numLags` — Maximum number of lag-term coefficients to return

positive integer

Maximum number of lag-term coefficients to return, specified as a positive integer.

If you specify `'numLags'`, then `arma2ar` truncates the output polynomial at a maximum of `numLags` lag terms, and then returns the remaining coefficients. As a result, the output vector has `numLags` elements or is at most a degree `numLags` LagOp lag operator polynomial.

By default, `arma2ar` determines the number of lag coefficients to return by the stopping criteria of mldivide.

Data Types: `double`

Output Arguments

`ar` — Coefficients of the truncated AR model

numeric vector | cell vector of square, numeric matrices | `LagOp` lag operator polynomial object

Coefficients of the truncated AR model approximation of the ARMA model, returned as a numeric vector, cell vector of square, numeric matrices, or a LagOp lag operator polynomial object. `ar` has `numLags` elements, or is at most a degree `numLags` `LagOp` lag operator polynomial.

The data types and orientations of `ar0` and `ma0` determine the data type and orientation of `ar`. If `ar0` or `ma0` are of the same data type or have the same orientation, then `ar` shares the common data type or orientation. If at least one of `ar0` or `ma0` is a `LagOp` lag operator polynomial, then `ar` is a `LagOp` lag operator polynomial. Otherwise, if at least one of `ar0` or `ma0` is a cell vector, then `ar` is a cell vector. If `ar0` and `ma0` are cell or numeric vectors and at least one is a row vector, then `ar` is a row vector.

If `ar` is a cell or numeric vector, then the order of the elements of `ar` corresponds to the order of the coefficients of the lagged responses in difference-equation notation starting with the coefficient of y_{t-1}. The resulting AR model is in reduced form.

If `ar` is a `LagOp` lag operator polynomial, then the order of the coefficients of `ar` corresponds to the order of the coefficients of the lagged responses in lag operator notation starting with the coefficient of y_t. If $\Phi_0 \neq I_{\text{numVars}}$, then the resulting AR model is structural. To view the coefficients in difference-equation notation, pass `ar` to reflect.

More About

Difference-Equation Notation

A linear time series model written in *difference-equation notation* positions the present value of the response and its structural coefficient on the left side of the equation. The right side of the equation contains the sum of the lagged responses, present innovation, and lagged innovations with corresponding coefficients.

That is, a linear time series written in difference-equation notation is

$$\Phi_0 y_t = c + \Phi_1 y_{t-1} + \ldots + \Phi_p y_{t-p} + \Theta_0 \varepsilon_t + \Theta_1 \varepsilon_{t-1} + \ldots + \Theta_q \varepsilon_{t-q},$$

where

- y_t is an `numVars`-dimensional vector representing the responses of `numVars` variables at time t, for all t and for `numVars` ≥ 1.
- ε_t is an `numVars`-dimensional vector representing the innovations at time t.
- Φ_j is the `numVars`-by-`numVars` matrix of AR coefficients of the response y_{t-j}, for $j = 0,\ldots,p$.
- Θ_k is the `numVars`-by-`numVars` matrix of MA coefficients of the innovation ε_{t-k}., $k = 0,\ldots,q$.
- c is the n-dimensional model constant.
- For models in reduced form, $\Phi_0 = \Theta_0 = I_{\text{numVars}}$, which is the `numVars`-dimensional identity matrix.

Lag Operator Notation

A time series model written in *lag-operator notation* positions a p-degree lag operator polynomial on the present response on the left side of the equation. The right side of the equation contains the model constant and a q-degree lag operator polynomial on the present innovation.

That is, a linear time series model written in lag-operator notation is

$$\Phi(L) y_t = c + \Theta(L) \varepsilon_t,$$

where

- y_t is an `numVars`-dimensional vector representing the responses of `numVars` variables at time t, for all t and for `numVars` ≥ 1.
- $\Phi(L) = \Phi_0 - \Phi_1 L - \Phi_2 L^2 - ... - \Phi_p L^p$, which is the autoregressive, lag operator polynomial.
- L is the back-shift operator, i.e., $L^j y_t = y_{t-j}$.
- Φ_j is the `numVars`-by-`numVars` matrix of AR coefficients of the response y_{t-j}, for $j = 0,...,p$.
- ε_t is an `numVars`-dimensional vector representing the innovations at time t.
- $\Theta(L) = \Theta_0 + \Theta_1 L + \Theta_2 L^2 + ... + \Theta_q L^q$, which is the moving average, lag operator polynomial.
- Θ_k is the `numVars`-by-`numVars` matrix of MA coefficients of the innovation ε_{t-k}., $k = 0,...,q$.
- c is the `numVars`-dimensional model constant.
- For models in reduced form, $\Phi_0 = \Theta_0 = I_{\text{numVars}}$, which is the `numVars`-dimensional identity matrix.

When comparing lag operator notation to difference equation notation, the signs of the lagged AR coefficients appear negated relative to the corresponding terms in difference equation notation. The signs of the moving average coefficients are the same and appear on the same side.

For more details on lag operator notation, see "Lag Operator Notation" on page 1-22.

Tips

- To accommodate structural ARMA models, specify the input arguments `ar0` and `ma0` as LagOp lag operator polynomials.
- To access the cell vector of the lag operator polynomial coefficients of the output argument `ar`, enter `toCellArray(ar)`.
- To convert the model coefficients of the output argument from lag operator notation to the model coefficients in difference-equation notation, enter

  ```
  arDEN = toCellArray(reflect(ar));
  ```

  `arDEN` is a cell vector containing at most `numLags` + 1 coefficients corresponding to the lag terms in `ar.Lags` of the AR model equivalent of the input ARMA model

in difference-equation notation. The first element is the coefficient of y_t, the second element is the coefficient of y_{t-1}, and so on.

Algorithms

- The software computes the infinite-lag polynomial of the resulting AR model according to this equation in lag operator notation:

$$\Theta^{-1}(L)\Phi(L)y_t = \varepsilon_t,$$

where $\Phi(L) = \sum_{j=0}^{p} \Phi_j L^j$ and $\Theta(L) = \sum_{k=0}^{q} \Theta_k L^k$.

- `arma2ar` approximates the AR model coefficients whether `ar0` and `ma0` compose a stable polynomial (a polynomial that is stationary or invertible). To check for stability, use isStable.

  `isStable` requires a `LagOp` lag operator polynomial as input. For example, if `ar0` is a vector, enter the following code to check `ar0` for stationarity.

```
ar0LagOp = LagOp([1 -ar0]);
isStable(ar0LagOp)
```

  A `0` indicates that the polynomial is not stable.

  You can similarly check whether the AR approximation to the ARMA model (`ar`) is stationary.

- "Lag Operator Notation" on page 1-22

References

[1] Box, G. E. P., G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis: Forecasting and Control* 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

[2] Hamilton, J. D. *Time Series Analysis*. Princeton, NJ: Princeton University Press, 1994.

[3] Lutkepohl, H. *New Introduction to Multiple Time Series Analysis*. Springer-Verlag, 2007.

See Also

`arma2ma` | isStable | LagOp | reflect | toCellArray | `var2vec` | `vec2var` | `vgxvarx`

Introduced in R2015a

arma2ma

Convert ARMA model to MA model

Syntax

```
ma = arma2ma(ar0,ma0)
ma = arma2ma(ar0,ma0,numLags)
```

Description

`ma = arma2ma(ar0,ma0)` returns the coefficients of the truncated, infinite-order MA model approximation to an ARMA model having AR and MA coefficients specified by `ar0` and `ma0`, respectively.

`arma2ma`:

- Accepts:
  - Vectors or cell vectors of matrices in difference-equation notation.
  - LagOp lag operator polynomials corresponding to the AR and MA polynomials in lag operator notation.
- Accommodates time series models that are univariate or multivariate (i.e., `numVars` variables compose the model), stationary or integrated, structural or in reduced form, and invertible.
- Assumes that the model constant c is 0.

`ma = arma2ma(ar0,ma0,numLags)` returns the first nonzero `numLags` lag-term coefficients of the infinite-order MA model approximation of an ARMA model having AR coefficients `ar0` and MA coefficients `ma0`.

Examples

Convert an ARMA model to an MA Model

Find the lag coefficients of the truncated, MA approximation of this univariate, stationary, and invertible ARMA model

$$y_t = 0.2y_{t-1} - 0.1y_{t-2} + \varepsilon_t + 0.5\varepsilon_{t-1}.$$

The ARMA model is in difference-equation notation because the left side contains only y_t and its coefficient 1. Create a vector containing the AR lag term coefficients in order starting from t - 1.

```
ar0 = [0.2 -0.1];
```

Alternatively, you can create a cell vector of the scalar coefficients.

Create a vector containing the MA lag term coefficient.

```
ma0 = 0.5;
```

Convert the ARMA model to an MA model by obtaining the coefficients of the truncated approximation of the infinite-lag polynomial.

```
ma = arma2ma(ar0,ma0)
```

```
ma =

    0.7000    0.0400   -0.0620   -0.0164
```

`ma` is a numeric vector because `ar0` and `ma0` are numeric vectors.

The approximate MA model truncated at 4 lags is

$$y_t = \varepsilon_t + 0.7\varepsilon_{t-1} + 0.04\varepsilon_{t-2} - 0.062\varepsilon_{t-3} - 0.0164\varepsilon_{t-4}.$$

Convert an AR(3) Model to an MA(5) Model

Find the first five lag coefficients of the MA approximation of this univariate and stationary AR(3) model

$$y_t = -0.2y_{t-1} + 0.5y_{t-3} + \varepsilon_t.$$

The AR model is in difference-equation notation because the left side contains only y_t and its coefficient of 1. Create a cell vector containing the AR lag term coefficient in order starting from t - 1. Because the second lag term of the MA model is missing, specify a `0` for its coefficient.

```
ar0 = {-0.2 0 0.5};
```

Convert the AR model to an MA model with at most five lag coefficents of the truncated approximation of the infinite-lag polynomial. Because there is no MA contribution, specify an empty cell ({}) for the MA coefficients.

```
numLags = 5;
ma0 = {};
ma = arma2ma(ar0,ma0,numLags)
```

```
ma =

    [-0.2000]    [0.0400]    [0.4920]    [-0.1984]    [0.0597]
```

`ma` is a cell vector of scalars because at least one of `ar0` and `ma0` is a cell vector.

The approximate MA(5) model is

$$y_t = \varepsilon_t - 0.2\varepsilon_{t-1} + 0.04\varepsilon_{t-2} + 0.492\varepsilon_{t-3} - 0.1984\varepsilon_{t-4} + 0.0597\varepsilon_{t-5}$$

Convert a Structural VARMA model to a Structural VMA model

Find the coefficients of the truncated, structural VMA equivalent of the structural, stationary, and invertible VARMA model

$$\left\{\begin{bmatrix} 1 & 0.2 & -0.1 \\ 0.03 & 1 & -0.15 \\ 0.9 & -0.25 & 1 \end{bmatrix} + \begin{bmatrix} 0.5 & -0.2 & -0.1 \\ -0.3 & -0.1 & 0.1 \\ 0.4 & -0.2 & -0.05 \end{bmatrix} L^4 + \begin{bmatrix} 0.05 & -0.02 & -0.01 \\ -0.1 & -0.01 & -0.001 \\ 0.04 & -0.02 & -0.005 \end{bmatrix} L^8\right\} y_t =$$
$$\left\{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} + \begin{bmatrix} -0.02 & 0.03 & 0.3 \\ 0.003 & 0.001 & 0.01 \\ 0.3 & 0.01 & 0.01 \end{bmatrix} L^4\right\} \varepsilon_t$$

where $y_t = [y_{1t} \quad y_{2t} \quad y_{3t}]'$ and $\varepsilon_t = [\varepsilon_{1t} \quad \varepsilon_{2t} \quad \varepsilon_{3t}]'$.

The VARMA model is in lag operator notation because the response and innovation vectors are on opposite sides of the equation.

Create a cell vector containing the VAR matrix coefficients. Because this model is a structural model, start with the coefficient of y_t and enter the rest in order by lag.

Construct a vector that indicates the degree of the lag term for the corresponding coefficients.

```
var0 = {[1 0.2 -0.1; 0.03 1 -0.15; 0.9 -0.25 1],...
    [0.5 -0.2 -0.1; -0.3 -0.1 0.1; 0.4 -0.2 -0.05],...
    [0.05 -0.02 -0.01; -0.1 -0.01 -0.001; 0.04 -0.02 -0.005]};
var0Lags = [0 4 8];
```

Create a cell vector containing the VMA matrix coefficients. Because this model is a structural model, start with the coefficient of ε_t and enter the rest in order by lag. Construct a vector that indicates the degree of the lag term for the corresponding coefficients.

```
vma0 = {eye(3),...
    [-0.02 0.03 0.3; 0.003 0.001 0.01; 0.3 0.01 0.01]};
vma0Lags = [0 4];
```

`arma2ma` requires `LagOp` lag operator polynomials for input arguments that comprise structural VAR or VMA models. Construct separate `LagOp` polynomials that describe the VAR and VMA components of the VARMA model.

```
VARLag = LagOp(var0,'Lags',var0Lags);
VMALag = LagOp(vma0,'Lags',vma0Lags);
```

`VARLags` and `VMALags` are `LagOp` lag operator polynomials that describe the VAR and VMA components of the VARMA model.

Convert the VARMA model to a VMA model by obtaining the coefficients of the truncated approximation of the infinite-lag polynomial. Specify to return at most 12 lagged terms.

```
numLags = 12;
VMA = arma2ma(VARLag,VMALag,numLags)
```

```
VMA =

    3-D Lag Operator Polynomial:
    ----------------------------
        Coefficients: [Lag-Indexed Cell Array with 4 Non-Zero Coefficients]
                Lags: [0 4 8 12]
              Degree: 12
           Dimension: 3
```

`VMA` is a `LagOP` lag operator polynomial. All coefficients except those corresponding to lags 0, 4, 8, and 12 are 3-by-3 matrices of zeros.

Display the nonzero coefficients of the resulting VMA model.

```
lag2Idx = VMA.Lags + 1; % Lags start at 0.  Add 1 to convert to indices.
vmaCoeff = toCellArray(VMA);

for j = 1:numel(lag2Idx)
    fprintf('__________Lag %d_________\n',lag2Idx(j) - 1)
    fprintf('%8.3f %8.3f %8.3f \n',vmaCoeff{lag2Idx(j)})
    fprintf    ('_____________________________\n')
end
```

```
__________Lag 0_________
   0.943   -0.162   -0.889
  -0.172    1.068    0.421
   0.069    0.144    0.974

_____________________________
__________Lag 4_________
  -0.650    0.460    0.546
   0.370    0.000   -0.019
   0.383   -0.111   -0.312

_____________________________
__________Lag 8_________
   0.431   -0.138   -0.089
  -0.170    0.122    0.065
  -0.260    0.165    0.089

_____________________________
__________Lag 12________
  -0.216    0.078    0.047
   0.099   -0.013   -0.011
   0.153   -0.042   -0.026

_____________________________
```

Find the Unconditional Mean of ARMA Models

Find the lag coefficients and constant of the truncated MA approximation of this univariate, stationary, and invertible ARMA model

$$y_t = 1.5 + 0.2y_{t-1} - 0.1y_{t-2} + \varepsilon_t + 0.5\varepsilon_{t-1}.$$

The ARMA model is in difference-equation notation because the left side contains only y_t and its coefficient of 1. Create separate vectors for the AR and MA lag term coefficients in order starting from t - 1.

```
ar0 = [0.2 -0.1];
```

```
ma0 = 0.5;
```

Convert the ARMA model to an MA model by obtaining the first five coefficients of the truncated approximation of the infinite-lag polynimial.

```
numLags = 5;
ar = arma2ma(ar0,ma0,numLags)
```

```
ar =

    0.7000    0.0400   -0.0620   -0.0164    0.0029
```

To compute the constant of the MA model, consider the ARMA model in lag operator notation.

$$\left(1-0.2L+0.1L^2\right)y_t = 1.5 + (1+0.5L)\varepsilon_t$$

or

$$\Phi(L)y_t = 1.5 + \Theta(L)\varepsilon_t$$

Part of the conversion involves premultiplying both sides of the equation by the inverse of the AR lag operator polynomial, as in this equation.

$$y_t = \Phi^{-1}(L)1.5 + \Phi^{-1}(L)\Theta(L)\varepsilon_t$$

To compute the inverse of AR lag operator polynomial, use the lag operator left-division object function mldivide (LagOp).

```
Phi = LagOp([1 -0.2 0.1]);
PhiInv = mldivide(Phi,1,'RelTol',1e-5);
```

`PhiInv` is a `LagOp` lag operator polynomial.

The application of lag operator polynomials to constants results in the product of the constant with the sum of the coefficients. Apply `PhiInv` to the ARMA model constant to obtain the MA model constant.

```
maConstant = 1.5*sum(cell2mat(toCellArray(PhiInv)))
```

```
maConstant =

    1.6667
```

The approximate MA model is

$$y_t = 1.667 + 0.7\varepsilon_{t-1} + 0.04\varepsilon_{t-2} - 0.062\varepsilon_{t-3} - 0.0164\varepsilon_{t-4} + 0.0029\varepsilon_{t-5} + \varepsilon_t.$$

Since the unconditional expected value of all innovations is 0, the unconditional expected value (or mean) of the response series is

$$E(y_t) = 1.667.$$

Input Arguments

ar0 — Autoregressive coefficients

numeric vector | cell vector of square, numeric matrices | LagOp lag operator polynomial object

Autoregressive coefficients of the ARMA(p,q) model, specified as a numeric vector, cell vector of square, numeric matrices, or a LagOp lag operator polynomial object. If `ar0` is a vector (numeric or cell), then the coefficient of y_t is the identity. To specify a structural AR polynomial (i.e., the coefficient of y_t is not the identity), use `LagOp` lag operator polynomials.

- For univariate time series models, `ar0` is a numeric vector, cell vector of scalars, or a one-dimensional `LagOp` lag operator polynomial. For vectors, `ar0` has length p and the elements correspond to lagged responses composing the AR polynomial in difference-equation notation. That is, `ar0(j)` or `ar0{j}` is the coefficient of y_{t-j}.
- For `numVars`-dimensional time series models, `ar0` is a cell vector of `numVars`-by-`numVars` numeric matrices or an `numVars`-dimensional `LagOp` lag operator polynomial. For cell vectors:
  - `ar0` has length p.
  - `ar0` and `ma0` must contain `numVars`-by-`numVars` matrices.
  - The elements of `ar0` correspond to the lagged responses composing the AR polynomial in difference equation notation. That is, `ar0{j}` is the coefficient matrix of y_{t-j}.

  - Row k of an AR coefficient matrix contains the AR coefficients in the equation of the variable y_k. Subsequently, column k must correspond to variable y_k, and the column and row order of all autoregressive and moving average coefficients must be consistent.
- For `LagOp` lag operator polynomials:
  - The first element of the `Coefficients` property corresponds to the coefficient of y_t (to accommodate structural models). All other elements correspond to the coefficients of the subsequent lags in the `Lags` property.
  - To construct a univariate model in reduced form, specify `1` for the first coefficient. For `numVars`-dimensional multivariate models, specify `eye(numVars)` for the first coefficient.
  - When you work from a model in difference-equation notation, negate the AR coefficient of the lagged terms to construct the lag-operator polynomial equivalent. For example, consider $y_t = 0.5y_{t-1} - 0.8y_{t-2} + \varepsilon_t - 0.6\varepsilon_{t-1} + 0.08\varepsilon_{t-2}$. The model is in difference-equation notation. To convert to an MA model, enter the following into the command window.

    ```
    ma = arma2ma([0.5 -0.8], [-0.6 0.08]);
    ```

    The ARMA model in lag operator notation is $\left(1 - 0.5L + 0.8L^2\right)y_t = \left(1 - 0.6L + 0.08L^2\right)\varepsilon_t$. The AR coefficients of the lagged responses are negated compared to the corresponding coefficients in difference-equation format. In this form, to obtain the same result, enter the following into the command window.

    ```
    ar0 = LagOp({1 -0.5 0.8});
    ma0 = LagOp({1 -0.6 0.08});
    ma = arma2ma(ar0, ma0);
    ```

It is a best practice for `ar0` to constitute a stationary or unit-root stationary (integrated) time series model.

`ma0` — Moving average coefficients

numeric vector | cell vector of square, numeric matrices | `LagOp` lag operator polynomial object

Moving average coefficients of the ARMA(p,q) model, specified as a numeric vector, cell vector of square, numeric matrices, or a LagOp lag operator polynomial object. If `ma0` is

a vector (numeric or cell), then the coefficient of ε_t is the identity. To specify a structural MA polynomial (i.e., the coefficient of ε_t is not the identity), use `LagOp` lag operator polynomials.

- For univariate time series models, `ma0` is a numeric vector, cell vector of scalars, or a one-dimensional `LagOp` lag operator polynomial. For vectors, `ma0` has length q and the elements correspond to lagged innovations composing the AR polynomial in difference-equation notation. That is, `ma0(j)` or `ma0{j}` is the coefficient of ε_{t-j}.
- For `numVars`-dimensional time series models, `ma0` is a cell vector of numeric `numVars`-by-`numVars` numeric matrices or an `numVars`-dimensional `LagOp` lag operator polynomial. For cell vectors:
  - `ma0` has length q.
  - `ar0` and `ma0` must both contain `numVars`-by-`numVars` matrices.
  - The elements of `ma0` correspond to the lagged responses composing the AR polynomial in difference equation notation. That is, `ma0{j}` is the coefficient matrix of y_{t-j}.
- For `LagOp` lag operator polynomials:
  - The first element of the `Coefficients` property corresponds to the coefficient of ε_t (to accommodate structural models). All other elements correspond to the coefficients of the subsequent lags in the `Lags` property.
  - To construct a univariate model in reduced form, specify `1` for the first coefficient. For `numVars`-dimensional multivariate models, specify `eye(numVars)` for the first coefficient.

If the ARMA model is strictly an AR model, then specify `[]` or `{}`.

It is a best practice for `ma0` to constitute an invertible time series model.

`numLags` — Maximum number of lag-term coefficients to return

positive integer

Maximum number of lag-term coefficients to return, specified as a positive integer.

If you specify `'numLags'`, then `arma2ma` truncates the output polynomial at a maximum of `numLags` lag terms, and then returns the remaining coefficients. As a result, the output vector has `numLags` elements or is at most a degree `numLags` LagOp lag operator polynomial.

By default, `arma2ma` determines the number of lag coefficients to return by the stopping criteria of mldivide.

Data Types: `double`

Output Arguments

`ma` — Lag-term coefficients of the truncated MA model

numeric vector | cell vector of square, numeric matrices | `LagOp` lag operator polynomial object

Lag-term coefficients of the truncated MA model approximation of the ARMA model, returned as a numeric vector, cell vector of square, numeric matrices, or a LagOp lag operator polynomial object. `ma` has `numLags` elements, or is at most a degree `numLags` `LagOp` lag operator polynomial.

The data types and orientations of `arO` and `maO` determine the data type and orientation of `ma`. If `arO` or `maO` are of the same data type or have the same orientation, then `ma` shares the common data type or orientation. If at least one of `arO` or `maO` is a `LagOp` lag operator polynomial, then `ma` is a `LagOp` lag operator polynomial. Otherwise, if at least one of `arO` or `maO` is a cell vector, then `ma` is a cell vector. If `arO` and `maO` are cell or numeric vectors and at least one is a row vector, then `ma` is a row vector.

If `ma` is a cell or numeric vector, then the order of the elements of `ma` corresponds to the order of the coefficients of the lagged innovations in difference-equation notation starting with the coefficient of ε_{t-1}. The resulting MA model is in reduced form.

If `ma` is a `LagOp` lag operator polynomial, then the order of the coefficients of `ma` corresponds to the order of the coefficients of the lagged innovations in lag operator notation starting with the coefficient of ε_t. If $\Theta_0 \neq I_{\text{numVars}}$, then the resulting MA model is structural.

More About

Difference-Equation Notation

A linear time series model written in *difference-equation notation* positions the present value of the response and its structural coefficient on the left side of the equation. The

right side of the equation contains the sum of the lagged responses, present innovation, and lagged innovations with corresponding coefficients.

That is, a linear time series written in difference-equation notation is

$$\Phi_0 y_t = c + \Phi_1 y_{t-1} + \ldots + \Phi_p y_{t-p} + \Theta_0 \varepsilon_t + \Theta_1 \varepsilon_{t-1} + \ldots + \Theta_q \varepsilon_{t-q},$$

where

- y_t is an `numVars`-dimensional vector representing the responses of `numVars` variables at time t, for all t and for `numVars` ≥ 1.
- ε_t is an `numVars`-dimensional vector representing the innovations at time t.
- Φ_j is the `numVars`-by-`numVars` matrix of AR coefficients of the response y_{t-j}, for $j = 0,\ldots,p$.
- Θ_k is the `numVars`-by-`numVars` matrix of MA coefficients of the innovation ε_{t-k}., $k = 0,\ldots,q$.
- c is the n-dimensional model constant.
- For models in reduced form, $\Phi_0 = \Theta_0 = I_{\text{numVars}}$, which is the `numVars`-dimensional identity matrix.

Lag Operator Notation

A time series model written in *lag-operator notation* positions a p-degree lag operator polynomial on the present response on the left side of the equation. The right side of the equation contains the model constant and a q-degree lag operator polynomial on the present innovation.

That is, a linear time series model written in lag-operator notation is

$$\Phi(L) y_t = c + \Theta(L) \varepsilon_t,$$

where

- y_t is an `numVars`-dimensional vector representing the responses of `numVars` variables at time t, for all t and for `numVars` ≥ 1.
- $\Phi(L) = \Phi_0 - \Phi_1 L - \Phi_2 L^2 - \ldots - \Phi_p L^p$, which is the autoregressive, lag operator polynomial.

- L is the back-shift operator, i.e., $L^j y_t = y_{t-j}$.
- Φ_j is the `numVars`-by-`numVars` matrix of AR coefficients of the response y_{t-j}, for $j = 0,...,p$.
- ε_t is an `numVars`-dimensional vector representing the innovations at time t.
- $\Theta(L) = \Theta_0 + \Theta_1 L + \Theta_2 L^2 + ... + \Theta_q L^q$, which is the moving average, lag operator polynomial.
- Θ_k is the `numVars`-by-`numVars` matrix of MA coefficients of the innovation ε_{t-k}, $k = 0,...,q$.
- c is the `numVars`-dimensional model constant.
- For models in reduced form, $\Phi_0 = \Theta_0 = I_{\text{numVars}}$, which is the `numVars`-dimensional identity matrix.

When comparing lag operator notation to difference equation notation, the signs of the lagged AR coefficients appear negated relative to the corresponding terms in difference equation notation. The signs of the moving average coefficients are the same and appear on the same side.

For more details on lag operator notation, see "Lag Operator Notation" on page 1-22.

Tips

- To accommodate structural ARMA models, specify the input arguments `ar0` and `ma0` as LagOp lag operator polynomials.
- To access the cell vector of the lag operator polynomial coefficients of the output argument `ma`, enter `toCellArray(ma)`.

Algorithms

- The software computes the infinite-lag polynomial of the resulting MA model according to this equation in lag operator notation:

$$y_t = \Phi^{-1}(L)\Theta(L)\varepsilon_t$$

where $\Phi(L) = \sum_{j=0}^{p} \Phi_j L^j$ and $\Theta(L) = \sum_{k=0}^{q} \Theta_k L^k$.

- `arma2ma` approximates the MA model coefficients whether `ar0` and `ma0` compose a stable polynomial (a polynomial that is stationary or invertible). To check for stability, use isStable.

  `isStable` requires a `LagOp` lag operator polynomial as input. For example, if `ar0` is a vector, enter the following code to check `ar0` for stationarity.

  ```
  ar0LagOp = LagOp([1 -ar0]);
  isStable(ar0LagOp)
  ```

  A `0` indicates that the polynomial is not stable.

  You can similarly check whether the MA approximation to the ARMA model (`ma`) is invertible.

- "Lag Operator Notation" on page 1-22

References

v

[1] Box, G. E. P., G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis: Forecasting and Control* 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

[2] Hamilton, J. D. *Time Series Analysis*. Princeton, NJ: Princeton University Press, 1994.

[3] Lutkepohl, H. *New Introduction to Multiple Time Series Analysis*. Springer-Verlag, 2007.

See Also

`arma2ar` | `armairf` | isStable | LagOp | toCellArray | `var2vec` | `vec2var` | `vgxvarx`

Introduced in R2015a

armairf

Generate ARMA model impulse responses

Syntax

```
armairf(ar0,ma0)
armairf(ar0,ma0,Name,Value)

Y = armairf(ar0,ma0)
Y = armairf(ar0,ma0,Name,Value)
```

Description

`armairf(ar0,ma0)` returns a tiered plot of the impulse response function, or dynamic response of the system, that results from applying a one standard deviation shock to each of the `numVars` time series variables composing an ARMA(p,q) model. The autoregressive and moving average coefficients of the ARMA(p,q) model are `ar0` and `ma0`, respectively.

`armairf`

- Accepts:
  - Vectors or cell vectors of matrices in difference-equation notation.
  - LagOp lag operator polynomials corresponding to the AR and MA polynomials in lag operator notation.
- Accommodates time series models that are univariate or multivariate, stationary or integrated, structural or in reduced form, and invertible or noninvertible.
- Assumes that the model constant c is 0.

`armairf(ar0,ma0,Name,Value)` returns a tiered plot of the impulse response function with additional options specified by one or more `Name,Value` pair arguments. For example, you can specify the number of periods to plot the impulse response function or the computation method to use.

`Y = armairf(ar0,ma0)` returns the impulse responses (`Y`) that result from applying a one standard deviation shock to each of the `numVars` time series variables in an

ARMA(p,q) model. The autoregressive coefficients `ar0` and the moving average coefficients `ma0` compose the ARMA model.

`Y = armairf(ar0,ma0,Name,Value)` returns the impulse responses with additional options specified by one or more `Name,Value` pair arguments.

Examples

Plot Orthogonalized Impulse Response Function of Univariate ARMA Model

Plot the entire impulse response function of the univariate ARMA(2,1) model

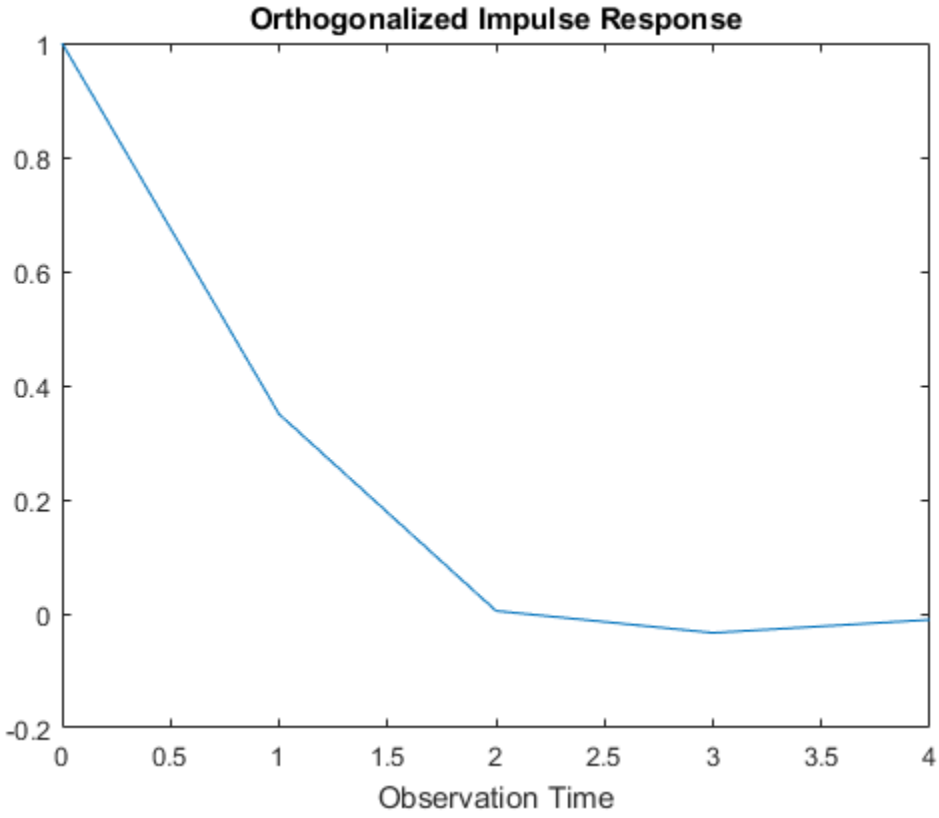
$$y_t = 0.3y_{t-1} - 0.1y_{t-2} + \varepsilon_t + 0.05\varepsilon_{t-1}$$

Create vectors for the autoregressive and moving average coefficients as you encounter them in the model expressed in difference-equation notation.

```
AR0 = [0.3 -0.1];
MA0 = 0.05;
```

Plot the orthogonalized impulse response function of y_t.

```
figure;
armairf(AR0,MA0);
```



Because y_t is univariate, there is one impulse response function in the plot. The impulse response dies after four periods.

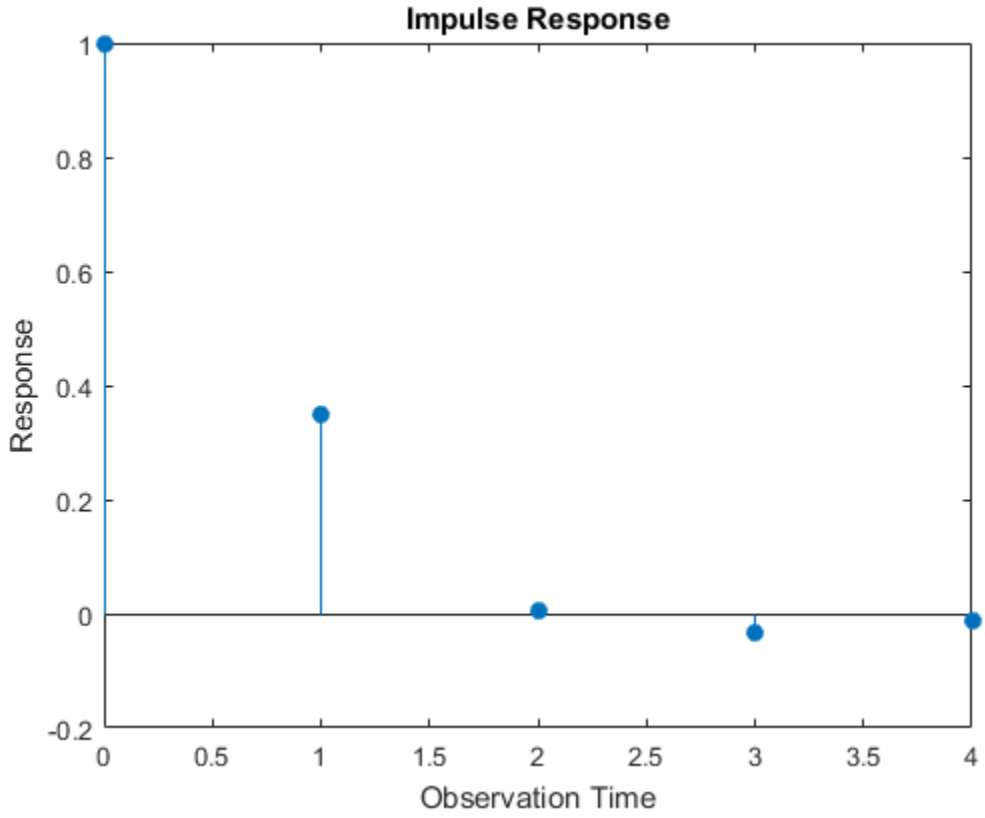
Alternatively, create an ARMA model that represents y_t. Specify that the variance of the innovations is 1, and that there is no model constant.

```
Mdl = arima('AR',AR0,'MA',MA0,'Variance',1,'Constant',0);
```

`Mdl` is an `arima` model object.

Plot the impulse response function using `Mdl`.

```
impulse(Mdl);
```



`impulse` uses a stem plot, whereas `armairf` uses a line plot. However, the impulse response functions between the two implementations are equivalent.

Plot Generalized Impulse Response Function of Univariate ARMA Model

Plot the entire impulse response function of the univariate ARMA(2,1) model

$$(1 - 0.3L + 0.1L^2)y_t = (1 + 0.05L)\varepsilon_t$$

Because the model is in lag operator form, create the polynomials using the coefficients as you encounter them in the model.

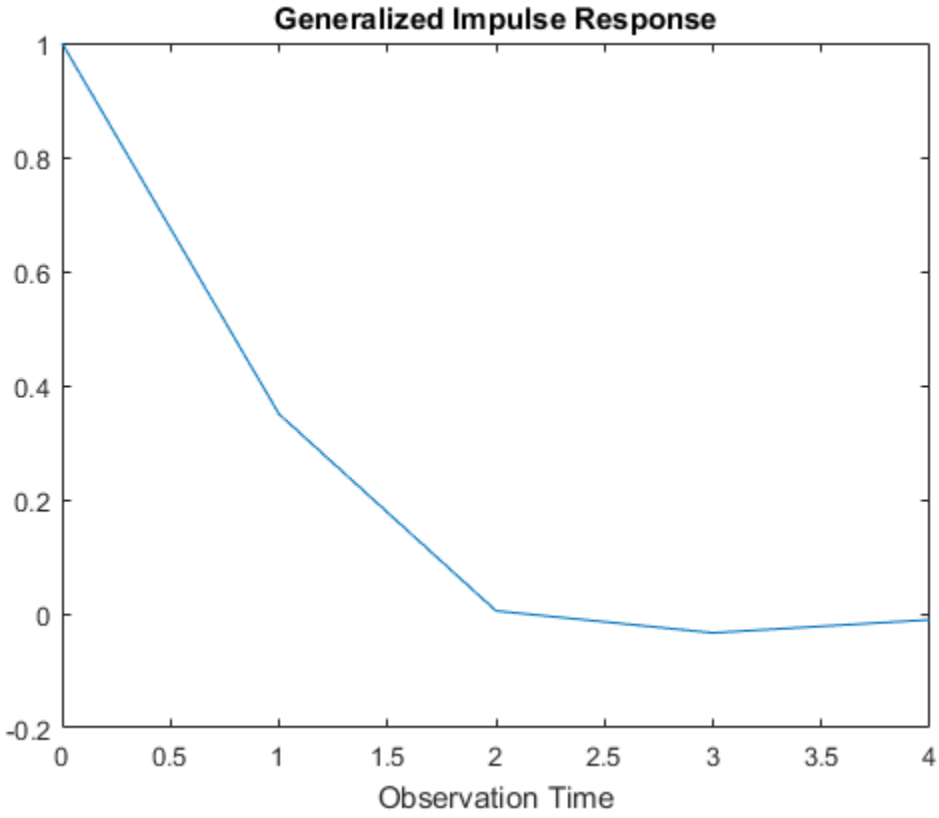
```
AROLag = LagOp([1 -0.3 0.1])
```

```
MAOLag = LagOp([1 0.05])


AROLag =

    1-D Lag Operator Polynomial:
    ----------------------------
        Coefficients: [1 -0.3 0.1]
                Lags: [0 1 2]
              Degree: 2
           Dimension: 1

MAOLag =

    1-D Lag Operator Polynomial:
    ----------------------------
        Coefficients: [1 0.05]
                Lags: [0 1]
              Degree: 1
           Dimension: 1
```

`AROLag` and `MAOLag` are `LagOp` lag operator polynomials representing the autoregressive and moving average lag operator polynomials, respectively.

Plot the generalized impulse response function by passing in the lag operator polynomials.

```
figure;
armairf(AROLag,MAOLag,'Method','generalized');
```



The impulse response function is equivalent to the impulse response function in "Plot Orthogonalized Impulse Response Function of Univariate ARMA Model".

Plot Generalized Impulse Response Function of VARMA Model

Plot the entire impulse response function of the structural VARMA(8,4) model

$$\left\{\begin{bmatrix} 1 & 0.2 & -0.1 \\ 0.03 & 1 & -0.15 \\ 0.9 & -0.25 & 1 \end{bmatrix} - \begin{bmatrix} -0.5 & 0.2 & 0.1 \\ 0.3 & 0.1 & -0.1 \\ -0.4 & 0.2 & 0.05 \end{bmatrix} L^4 - \begin{bmatrix} -0.05 & 0.02 & 0.01 \\ 0.1 & 0.01 & 0.001 \\ -0.04 & 0.02 & 0.005 \end{bmatrix} L^8\right\} y_t =$$
$$\left\{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} + \begin{bmatrix} -0.02 & 0.03 & 0.3 \\ 0.003 & 0.001 & 0.01 \\ 0.3 & 0.01 & 0.01 \end{bmatrix} L^4\right\} \varepsilon_t$$

where $y_t = [y_{1t} \quad y_{2t} \quad y_{3t}]'$ and $\varepsilon_t = [\varepsilon_{1t} \quad \varepsilon_{2t} \quad \varepsilon_{3t}]'$.

The VARMA model is in lag operator notation because the response and innovation vectors are on opposite sides of the equation.

Create a cell vector containing the VAR matrix coefficients. Because this model is a structural model in lag operator notation, start with the coefficient of y_t and enter the rest in order by lag. Construct a vector that indicates the degree of the lag term for the corresponding coefficients.

```
var0 = {[1 0.2 -0.1; 0.03 1 -0.15; 0.9 -0.25 1],...
    -[-0.5 0.2 0.1; 0.3 0.1 -0.1; -0.4 0.2 0.05],...
    -[-0.05 0.02 0.01; 0.1 0.01 0.001; -0.04 0.02 0.005]};
var0Lags = [0 4 8];
```

Create a cell vector containing the VMA matrix coefficients. Because this model is in lag operator notation, start with the coefficient of ε_t and enter the rest in order by lag. Construct a vector that indicates the degree of the lag term for the corresponding coefficients.

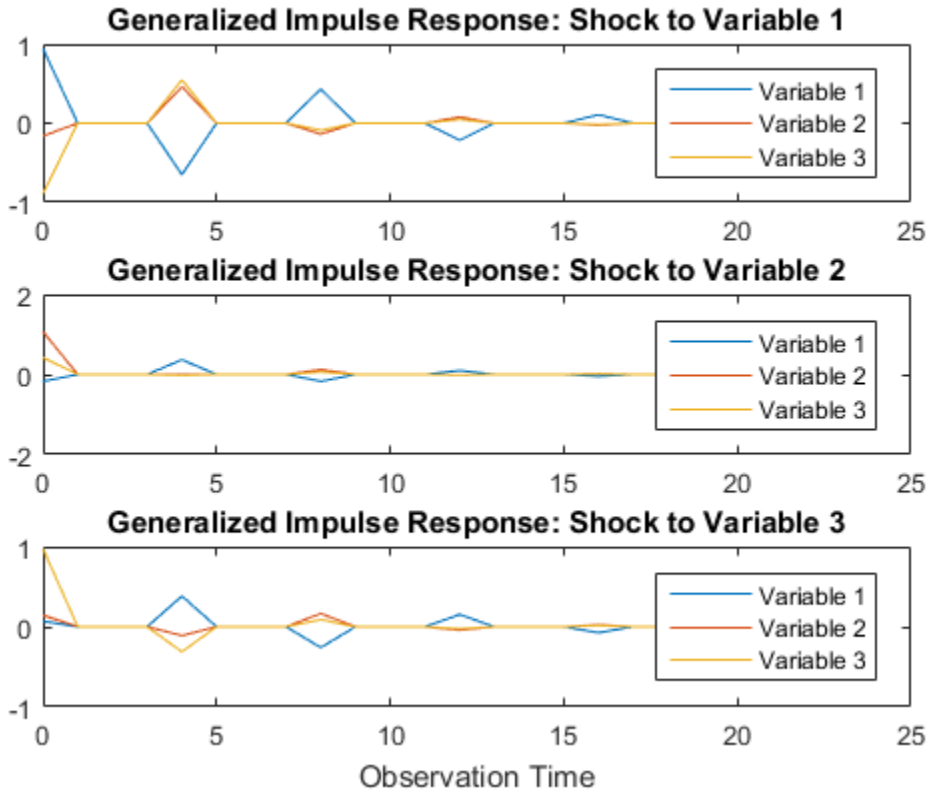
```
vma0 = {eye(3),...
    [-0.02 0.03 0.3; 0.003 0.001 0.01; 0.3 0.01 0.01]};
vma0Lags = [0 4];
```

Construct separate lag operator polynomials that describe the VAR and VMA components of the VARMA model.

```
VARLag = LagOp(var0,'Lags',var0Lags);
VMALag = LagOp(vma0,'Lags',vma0Lags);
```

Plot the impulse response function of the VARMA model.

```
figure;
armairf(VARLag,VMALag,'Method','generalized');
```



The figure contains three subplots. The top plot contains the impulse responses of all variables resulting from an innovation shock to $y_{1,t}$, the second plot from the top contains the impulse responses of all variables resulting from an innovation shock to $y_{2,t}$, and so on. Because the impulse responses die out after a finite number of periods, the VARMA model is stable.

Impulse Response Function of ARMA Model

Compute the entire, orthogonalized impulse response function of the univariate ARMA(2,1) model

$$y_t = 0.3y_{t-1} - 0.1y_{t-2} + \varepsilon_t + 0.05\varepsilon_{t-1}$$

Create vectors for the autoregressive and moving average coefficients as you encounter them in the model expressed in difference-equation notation.

```
AR0 = [0.3 -0.1];
MA0 = 0.05;
```

Plot the orthogonalized impulse response function of y_t.

```
y = armairf(AR0,MA0)
```

```
y =

    1.0000
    0.3500
    0.0050
   -0.0335
   -0.0105
```

`y` is a 5-by-1 vector of impulse responses. `y(1)` is the impulse response for time $t = 0$, `y(2)` is the impulse response for time $t = 1$, and so on. The impulse response function dies out after period $t = 4$.

Alternatively, create an ARMA model that represents y_t. Specify that the variance of the innovations is 1, and that there is no model constant.

```
Mdl = arima('AR',AR0,'MA',MA0,'Variance',1,'Constant',0);
```

`Mdl` is an `arima` model object.

Plot the impulse response function using `Mdl`.

```
y = impulse(Mdl)
```

```
y =

    1.0000
    0.3500
    0.0050
   -0.0335
   -0.0105
```

The impulse response functions between the two implementations are equivalent.

Impulse Response Function of VAR Model

Compute the generalized impulse response function of the 2 dimensional VAR(3) model

$$y_t = \begin{bmatrix} 1 & -0.2 \\ -0.1 & 0.3 \end{bmatrix} y_{t-1} - \begin{bmatrix} 0.75 & -0.1 \\ -0.05 & 0.15 \end{bmatrix} y_{t-2} + \begin{bmatrix} 0.55 & -0.02 \\ -0.01 & 0.03 \end{bmatrix} y_{t-3} + \varepsilon_t$$

$y_t = [y_{1,t} \quad y_{2,t}]'$, $\varepsilon_t = [\varepsilon_{1,t} \quad \varepsilon_{2,t}]'$, and, for all t, ε_t is Gaussian with mean zero and covariance matrix

$$\Sigma = \begin{bmatrix} 0.5 & -0.1 \\ -0.1 & 0.25 \end{bmatrix}$$

Create a cell vector of matrices for the autoregressive coefficients as you encounter them in the model expressed in difference-equation notation. Specify the innovation covariance matrix.

```
AR1 = [1 -0.2; -0.1 0.3];
AR2 = -[0.75 -0.1; -0.05 0.15];
AR3 = [0.55 -0.02; -0.01 0.03];
ar0 = {AR1 AR2 AR3};

InnovCov = [0.5 -0.1; -0.1 0.25];
```

Compute the entire, generalized impulse response function of y_t. Because there are no MA terms, specify an empty array ([ ]) for the second input argument.

```
Y = armairf(ar0,[],'Method','generalized','InnovCov',InnovCov);
size(Y)


ans =

    31     2     2
```

`Y` is a 31-by-2-2 array of impulse responses. Rows correspond to periods, columns correspond to variables, and pages correspond to the variable that `armairf` shocks. The `armairf` satisfies the stopping criterion after 31 periods. You can specify to stop sooner

using the 'NumObs' name-value pair argument. This is a good practice when there are many variables in the system.

Compute and display the generalized impulse responses for the first ten periods.

```
Y20 = armairf(ar0,[],'Method','generalized','InnovCov',InnovCov,...
    'NumObs',10)
```

```
Y20(:,:,1) =

    0.7071   -0.1414
    0.7354   -0.1131
    0.2135   -0.0509
    0.0526    0.0058
    0.2929    0.0040
    0.3717   -0.0300
    0.1872   -0.0325
    0.0730   -0.0082
    0.1360   -0.0001
    0.1841   -0.0116


Y20(:,:,2) =

   -0.2000    0.5000
   -0.3000    0.1700
   -0.1340   -0.0040
   -0.0112   -0.0113
   -0.0772   -0.0003
   -0.1435    0.0100
   -0.0936    0.0133
   -0.0301    0.0054
   -0.0388   -0.0003
   -0.0674    0.0028
```

The impulse responses appear to die out with increasing time. This suggests a stable system.

- “Generate Impulse Responses for a VAR model” on page 7-42
- “Compare Generalized and Orthogonalized Impulse Response Functions” on page 7-45
- “Generate VEC Model Impulse Responses” on page 7-138

Input Arguments

ar0 — Autoregressive coefficients

numeric vector | cell vector of square, numeric matrices | LagOp lag operator polynomial object

Autoregressive coefficients of the ARMA(p,q) model, specified as a numeric vector, cell vector of square, numeric matrices, or a LagOp lag operator polynomial object. If `ar0` is a vector (numeric or cell), then the coefficient of y_t is the identity. To specify a structural AR polynomial (i.e., the coefficient of y_t is not the identity), use `LagOp` lag operator polynomials.

- For univariate time series models, `ar0` is a numeric vector, cell vector of scalars, or a one-dimensional `LagOp` lag operator polynomial. For vectors, `ar0` has length p and the elements correspond to lagged responses composing the AR polynomial in difference-equation notation. That is, `ar0(j)` or `ar0{j}` is the coefficient of y_{t-j}.
- For `numVars`-dimensional time series models, `ar0` is a cell vector of `numVars`-by-`numVars` numeric matrices or an `numVars`-dimensional `LagOp` lag operator polynomial. For cell vectors:
  - `ar0` has length p.
  - `ar0` and `ma0` must contain `numVars`-by-`numVars` matrices.
  - The elements of `ar0` correspond to the lagged responses composing the AR polynomial in difference equation notation. That is, `ar0{j}` is the coefficient matrix of vector y_{t-j}.
  - Row k of an AR coefficient matrix contains the AR coefficients in the equation of the variable y_k. Subsequently, column k must correspond to variable y_k, and the column and row order of all autoregressive and moving average coefficients must be consistent.
- For `LagOp` lag operator polynomials:
  -
  - The first element of the `Coefficients` property corresponds to the coefficient of y_t (to accommodate structural models). All other elements correspond to the coefficients of the subsequent lags in the `Lags` property.
  - To construct a univariate model in reduced form, specify `1` for the first coefficient. For `numVars`-dimensional multivariate models, specify `eye(numVars)` for the first coefficient.

- `armairf` composes the model using lag operator notation. That is, when you work from a model in difference-equation notation, negate the AR coefficients of the lagged responses to construct the lag-operator polynomial equivalent. For example, consider $y_t = 0.5y_{t-1} - 0.8y_{t-2} + \varepsilon_t - 0.6\varepsilon_{t-1} + 0.08\varepsilon_{t-2}$. The model is in difference-equation form. To compute the impulse responses, enter the following into the command window.

  ```
  y = armairf([0.5 -0.8], [-0.6 0.08]);
  ```

  The ARMA model written in lag-operator notation is $\left(1 - 0.5L + 0.8L^2\right)y_t = \left(1 - 0.6L + 0.08L^2\right)\varepsilon_t$. The AR coefficients of the lagged responses are negated compared to the corresponding coefficients in difference-equation format. In this form, to obtain the same result, enter the following into the command window.

  ```
  ar0 = LagOp({1 -0.5 0.8});
  ma0 = LagOp({1 -0.6 0.08});
  y = armairf(ar0, ma0);
  ```

If the ARMA model is strictly an MA model, then specify an empty array or cell (`[]` or `{}`).

`ma0` — Moving average coefficients

numeric vector | cell vector of square, numeric matrices | `LagOp` lag operator polynomial object

Moving average coefficients of the ARMA(p,q) model, specified as a numeric vector, cell vector of square, numeric matrices, or a LagOp lag operator polynomial object. If `ma0` is a vector (numeric or cell), then the coefficient of ε_t is the identity. To specify a structural MA polynomial (i.e., the coefficient of ε_t is not the identity), use `LagOp` lag operator polynomials.

- For univariate time series models, `ma0` is a numeric vector, cell vector of scalars, or a one-dimensional `LagOp` lag operator polynomial. For vectors, `ma0` has length q and the elements correspond to lagged innovations composing the AR polynomial in difference-equation notation. That is, `ma0(j)` or `ma0{j}` is the coefficient of ε_{t-j}.
- For `numVars`-dimensional time series models, `ma0` is a cell vector of numeric `numVars`-by-`numVars` numeric matrices or an `numVars`-dimensional `LagOp` lag operator polynomial. For cell vectors:

  - `ma0` has length q.
  - `ar0` and `ma0` must both contain `numVars`-by-`numVars` matrices.
  - The elements of `ma0` correspond to the lagged responses composing the AR polynomial in difference equation notation. That is, `ma0{j}` is the coefficient matrix of y_{t-j}.
  - Row k of an MA coefficient matrix contains the MA coefficients in the equation of the variable y_k. Subsequently, column k must correspond to variable y_k, and the order of all autoregressive and moving average coefficients must be consistent.
- For `LagOp` lag operator polynomials:
  - The first element of the `Coefficients` property corresponds to the coefficient of ε_t (to accommodate structural models). All other elements correspond to the coefficients of the subsequent lags in the `Lags` property.
  - To construct a univariate model in reduced form, specify `1` for the first coefficient. For `numVars`-dimensional multivariate models, specify `eye(numVars)` for the first coefficient.

If the ARMA model is strictly an AR model, then specify an empty array or cell (`[]` or `{}`).

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Method','generalized','NumObs',10` specifies to compute generalized impulse responses for ten periods.

`'InnovCov'` — Covariance matrix

`eye(numVars)` (default) | numeric scalar | numeric matrix

Covariance matrix of the ARMA(p,q) model innovations ε_t, specified as the comma-separated pair consisting of `'InnovCov'` and a numeric scalar or an `numVars`-by-`numVars` numeric matrix. `InnovCov` must be a positive scalar or a positive definite matrix.

Example: `'InnovCov',0.2`

Data Types: `double`

`'NumObs'` — Number of periods in the impulse response function to return

positive integer

Number of periods in the impulse response function to return, specified as the comma-separated pair consisting of `'NumObs'` and a positive integer. `NumObs` specifies the number of rows in the output argument `Y`.

By default, `armairf` determines `NumObs` by the stopping criteria of mldivide.

Example: `'NumObs',10`

Data Types: `double`

`'Method'` — Impulse response function computation method

`'orthogonalized'` (default) | `'generalized'`

Impulse response function computation method, specified as the comma-separated pair consisting of `'Method'` and a string.

Value	Description
`'generalized'`	Compute impulse responses using one standard deviation innovation shocks.
`'orthogonalized'`	Compute impulse responses using orthogonalized, one standard deviation innovation shocks. `armairf` uses the Cholesky factorization of `InnovCov` for orthogonalization.

Example: `'Method','generalized'`

Data Types: `char`

Output Arguments

`Y` — Impulse responses

numeric column vector | numeric array

Impulse responses, returned as a numeric column vector or array.

If Y is a vector, then `Y(t)` is the impulse response at period t, where $t = 0,1,...,$`NumObs`.

Otherwise, `Y(t,j,k)` is the period-t impulse response of variable j shocked by a one standard impulse originating in variable k. $t = 0,1,...,$`NumObs`, $j = 1,2,...,$`numVars`, and $k = 1,2,...,$`numVars`. The variable order in Y corresponds to the variable order in `ar0` and `ma0`.

More About

Difference-Equation Notation

A linear time series model written in *difference-equation notation* positions the present value of the response and its structural coefficient on the left side of the equation. The right side of the equation contains the sum of the lagged responses, present innovation, and lagged innovations with corresponding coefficients.

That is, a linear time series written in difference-equation notation is

$$\Phi_0 y_t = c + \Phi_1 y_{t-1} + ... + \Phi_p y_{t-p} + \Theta_0 \varepsilon_t + \Theta_1 \varepsilon_{t-1} + ... + \Theta_q \varepsilon_{t-q},$$

where

- y_t is an `numVars`-dimensional vector representing the responses of `numVars` variables at time t, for all t and for `numVars` ≥ 1.
- ε_t is an `numVars`-dimensional vector representing the innovations at time t.
- Φ_j is the `numVars`-by-`numVars` matrix of AR coefficients of the response y_{t-j}, for $j = 0,...,p$.
- Θ_k is the `numVars`-by-`numVars` matrix of MA coefficients of the innovation ε_{t-k}., $k = 0,...,q$.
- c is the n-dimensional model constant.
- For models in reduced form, $\Phi_0 = \Theta_0 = I_{\text{numVars}}$, which is the `numVars`-dimensional identity matrix.

Impulse Response Function

An *impulse response function* of a time series model measures the changes in the future responses of all variables in the system when a variable is shocked by an impulse.

Suppose y_t is the ARMA(p,q) model containing `numVars` response variables

$$\Phi(L)y_t = \Theta(L)\varepsilon_t.$$

- $\Phi(L)$ is the lag operator polynomial of the autoregressive coefficients, i.e., $\Phi(L) = \Phi_0 - \Phi_1 L - \Phi_2 L^2 - \ldots - \Phi_p L^p$.
- $\Theta(L)$ is the lag operator polynomial of the moving average coefficients, i.e., $\Theta(L) = \Theta_0 + \Theta_1 L + \Theta_2 L^2 + \ldots + \Theta_q L^q$.
- ε_t is the vector of `numVars` innovations at time t. Assume that the innovations have zero mean and the constant, positive-definite covariance matrix Σ for all t.

The infinite-lag MA representation of y_t is

$$y_t = \Phi^{-1}(L)\Theta(L)\varepsilon_t = \Omega(L)\varepsilon_t.$$

Then, the general form of the impulse response function of y_t shocked by an impulse to variable j by one standard deviation of its innovation m periods into the future is

$$\psi_j(m) = C_m e_j.$$

- e_j is a selection vector of length `numVars` containing a one in element j and zeros elsewhere.
- For orthogonalized impulse responses, $C_m = \Omega_m P,$ where P is the lower triangular factor in the Cholesky factorization of Σ.
- For generalized impulse responses, $C_m = \sigma_j^{-1}\Omega_m\Sigma,$ where σ_j is the standard deviation of innovation j.

Lag Operator Notation

A time series model written in *lag-operator notation* positions a p-degree lag operator polynomial on the present response on the left side of the equation. The right side of the equation contains the model constant and a q-degree lag operator polynomial on the present innovation.

That is, a linear time series model written in lag-operator notation is

$$\Phi(L)y_t = c + \Theta(L)\varepsilon_t,$$

where

- y_t is an `numVars`-dimensional vector representing the responses of `numVars` variables at time t, for all t and for `numVars` ≥ 1.
- $\Phi(L) = \Phi_0 - \Phi_1 L - \Phi_2 L^2 - \ldots - \Phi_p L^p$, which is the autoregressive, lag operator polynomial.
- L is the back-shift operator, i.e., $L^j y_t = y_{t-j}$.
- Φ_j is the `numVars`-by-`numVars` matrix of AR coefficients of the response y_{t-j}, for $j = 0,\ldots,p$.
- ε_t is an `numVars`-dimensional vector representing the innovations at time t.
- $\Theta(L) = \Theta_0 + \Theta_1 L + \Theta_2 L^2 + \ldots + \Theta_q L^q$, which is the moving average, lag operator polynomial.
- Θ_k is the `numVars`-by-`numVars` matrix of MA coefficients of the innovation ε_{t-k}, $k = 0,\ldots,q$.
- c is the `numVars`-dimensional model constant.
- For models in reduced form, $\Phi_0 = \Theta_0 = I_{\text{numVars}}$, which is the `numVars`-dimensional identity matrix.

When comparing lag operator notation to difference equation notation, the signs of the lagged AR coefficients appear negated relative to the corresponding terms in difference equation notation. The signs of the moving average coefficients are the same and appear on the same side.

For more details on lag operator notation, see “Lag Operator Notation” on page 1-22.

Tips

- To compute *forecast error impulse responses*, use the default value of `InnovCov`, which is a `numVar`-by-`numVars` identity matrix. In this case, all available computation methods (see `Method`) result in equivalent impulse response functions.
- To accommodate structural ARMA(p,q) models, specify the input arguments `ar0` and `ma0` as LagOp lag operator polynomials.

Algorithms

- If `Method` is `'orthogonalized'`, then the resulting impulse response function depends on the order of the variables in the time series model. If `Method` is `'generalized'`, then the resulting impulse response function is invariant to the order of the variables. Therefore, the two methods generally produce different results.
- If `InnovCov` is a diagonal matrix, then the resulting generalized and orthogonal impulse response functions are identical. Otherwise, the resulting generalized and orthogonal impulse response functions are identical when the first variable shocks all variables only (i.e., `Y(:,:,1)`).

- "Impulse Response Function" on page 5-86

References

[1] Hamilton, J. D. *Time Series Analysis*. Princeton, NJ: Princeton University Press, 1994.

[2] Lutkepohl, H. *New Introduction to Multiple Time Series Analysis*. Springer-Verlag, 2007.

[3] Pesaran, H. H. and Y. Shin. "Generalized Impulse Response Analysis in Linear Multivariate Models." *Economic Letters*. Vol. 58, 1998, 17–29.

See Also

arima | impulse | LagOp | mldivide | `vec2var`

Introduced in R2015b

autocorr

Sample autocorrelation

Syntax

```
autocorr(y)
autocorr(y,numLags)
autocorr(y,numLags,numMA,numSTD)

acf = autocorr(y)
acf = autocorr(y,numLags)
acf = autocorr(y,numLags,numMA,numSTD)
[acf,lags,bounds] = autocorr( ___ )
```

Description

`autocorr(y)` plots the sample autocorrelation function (ACF) of the univariate, stochastic time series `y` with confidence bounds.

`autocorr(y,numLags)` plots the ACF, where `numLags` indicates the number of lags in the sample ACF.

`autocorr(y,numLags,numMA,numSTD)` plots the ACF, where `numMA` specifies the number of lags beyond which the theoretical ACF is effectively 0, and `numSTD` specifies the number of standard deviations of the sample ACF estimation error.

`acf = autocorr(y)` returns the sample ACF of the univariate, stochastic time series `y`.

`acf = autocorr(y,numLags)` returns the ACF, where `numLags` specifies the number of lags in the sample ACF.

`acf = autocorr(y,numLags,numMA,numSTD)` returns the ACF, where `numMA` specifies the number of lags beyond which the theoretical ACF is effectively 0, and `numSTD` specifies the number of standard deviations of the sample ACF estimation error.

`[acf,lags,bounds] = autocorr( ___ )` additionally returns the lags (`lags`) corresponding to the ACF and the approximate upper and lower confidence bounds (`bounds`), using any of the input arguments in the previous syntaxes.

Examples

Plot the Autocorrelation Function of a Time Series

Specify the MA(2) model:

$$y_t = \varepsilon_t - 0.5\varepsilon_{t-1} + 0.4\varepsilon_{t-2},$$

where ε_t is Gaussian with mean 0 and variance 1.

```
rng(1); % For reproducibility
Mdl = arima('MA',{-0.5 0.4},'Constant',0,'Variance',1)
```

```
Mdl =

    ARIMA(0,0,2) Model:
    --------------------
    Distribution: Name = 'Gaussian'
               P: 0
               D: 0
               Q: 2
        Constant: 0
              AR: {}
             SAR: {}
              MA: {-0.5 0.4} at Lags [1 2]
             SMA: {}
        Variance: 1
```

Simulate 1000 observations from `Mdl`.

```
y = simulate(Mdl,1000);
```

Compute the ACF.

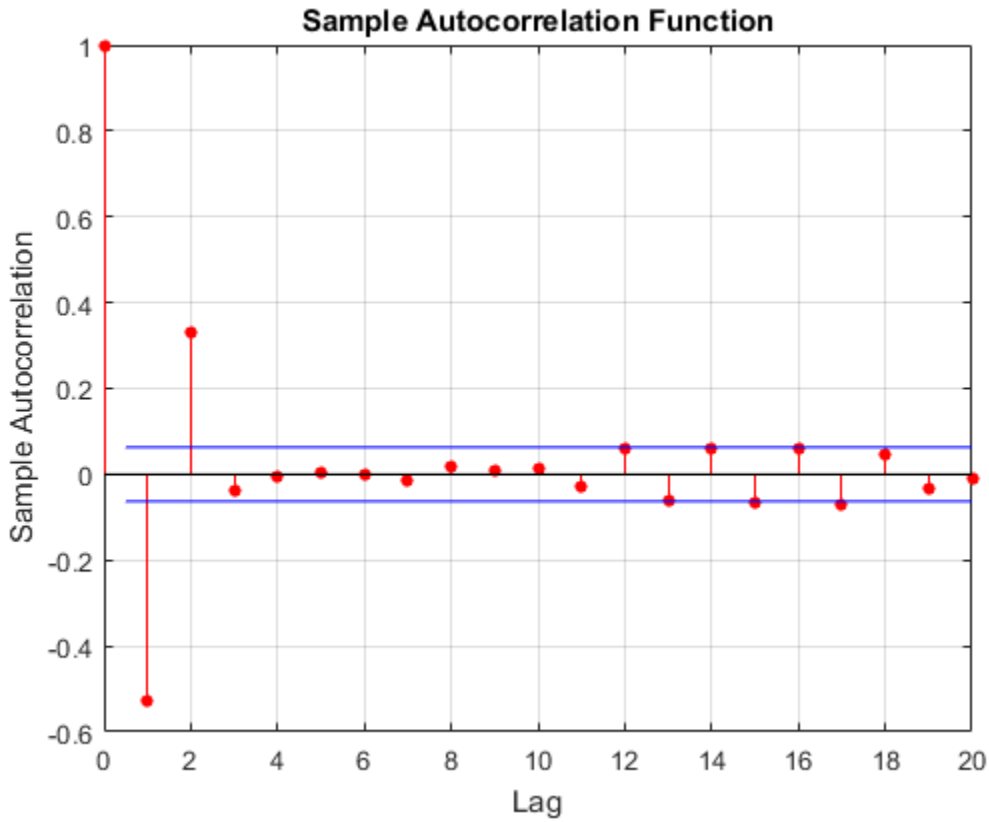
```
[ACF,lags,bounds] = autocorr(y,[],2);
bounds
```

```
bounds =

    0.0843
   -0.0843
```

[ ] tells the software to return the default number of lags (20). `numMA = 2` indicates that the ACF is effectively 0 after the second lag. `bounds` displays (-0.0843, 0.0843), which are the upper and lower confidence bounds.

Plot the ACF.

```
autocorr(y)
```



The ACF cuts off after the second lag. This behavior indicataes an MA(2) process.

Specify More Lags for the ACF Plot

Specify the mutliplicative seasonal ARMA $(2,0,1) \times (3,0,0)_{12}$ model:

$$(1 - 0.75L - 0.15L^2)(1 - 0.9L^{12} + 0.5L^{24} - 0.5L^{36})y_t = 2 + \varepsilon_t - 0.5\varepsilon_{t-1},$$

where ε_t is Gaussian with mean 0 and variance 1.

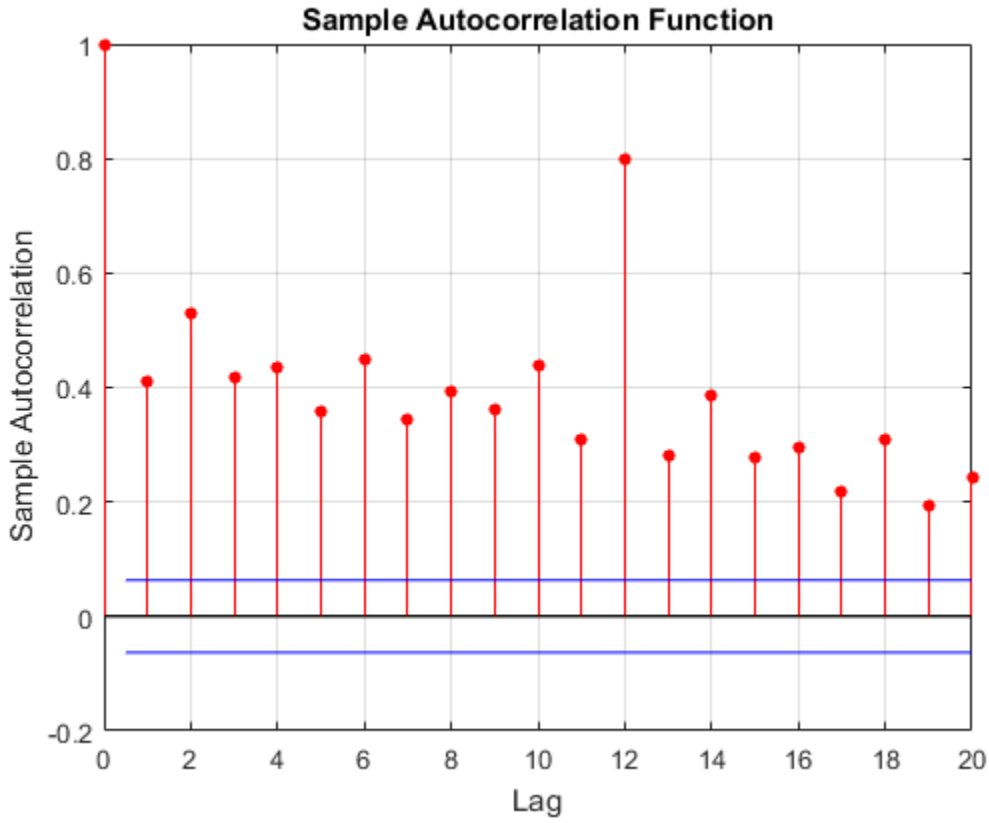
```
Mdl = arima('AR',{0.75,0.15},'SAR',{0.9,-0.5,0.5},...
    'SARLags',[12,24,36],'MA',-0.5,'Constant',2,...
    'Variance',1);
```

Simulate data from `Mdl`.

```
rng(1); % For reproducibility
y = simulate(Mdl,1000);
```

Plot the default autocorrelation function (ACF).

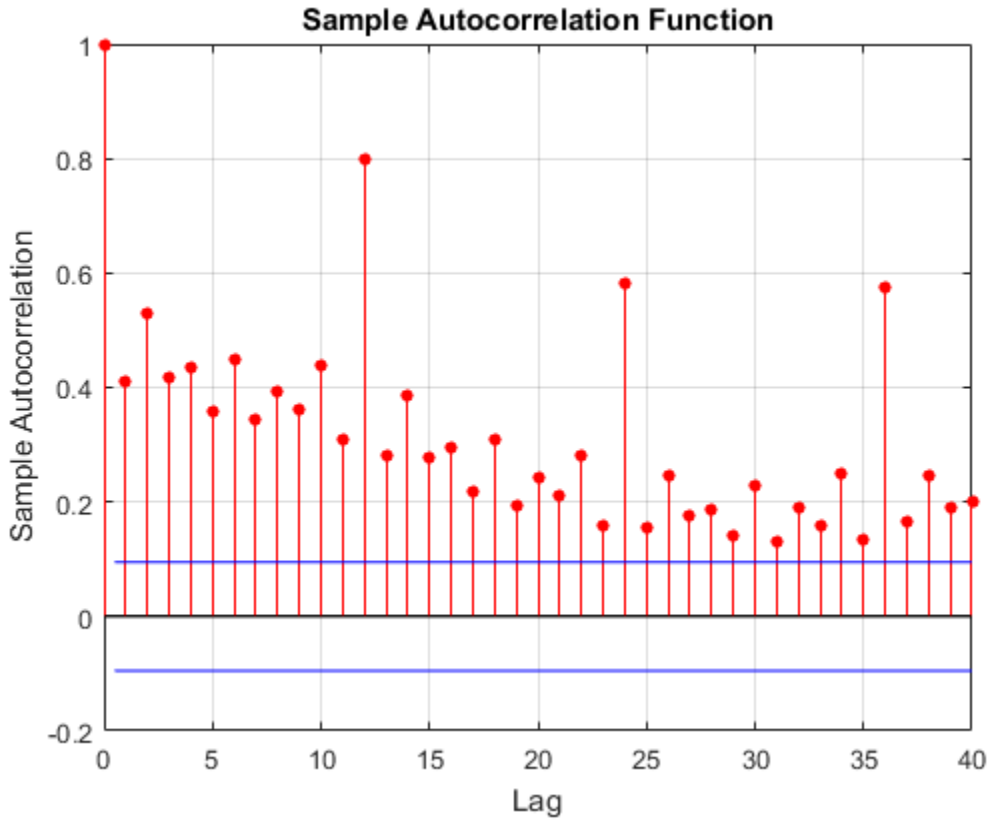
```
figure
autocorr(y)
```



The default correlogram does not display the dependence structure for higher lags.

Plot the ACF for 40 lags.

```
figure
autocorr(y,40,[],3)
```



The correlogram shows the larger correlations at lags 12, 24, and 36.

Compare the ACF for Normalized and Unnormalized Series

Although various estimates of the sample autocorrelation function exist, `autocorr` uses the form in Box, Jenkins, and Reinsel, 1994. In their estimate, they scale the correlation at each lag by the sample variance (`var(y,1)`) so that the autocorrelation at lag 0 is unity. However, certain applications require rescaling the normalized ACF by another factor.

Simulate 1000 observations from the standard Gaussian distribution.

```
rng(1); % For reproducibility
y = randn(1000, 1);
```

Compute the normalized and unnormalized sample ACF.

```
[normalizedACF, lags] = autocorr(y, 10);
unnormalizedACF = normalizedACF*var(y,1);
```

Compare the first 10 lags of the sample ACF with and without normalization.

```
[lags  normalizedACF  unnormalizedACF]
```

```
ans =

         0    1.0000    0.9960
    1.0000   -0.0180   -0.0180
    2.0000    0.0536    0.0534
    3.0000   -0.0206   -0.0205
    4.0000   -0.0300   -0.0299
    5.0000   -0.0086   -0.0086
    6.0000   -0.0108   -0.0107
    7.0000   -0.0116   -0.0116
    8.0000    0.0309    0.0307
    9.0000    0.0341    0.0340
   10.0000    0.0076    0.0075
```

- “Time Series Regression VI: Residual Diagnostics”
- “Detect Autocorrelation” on page 3-18
- “Box-Jenkins Model Selection” on page 3-4

Input Arguments

y — Observed univariate time series

vector

Observed univariate time series for which the software computes or plots the ACF, specified as a vector. The last element of `y` contains the most recent observation.

Data Types: `double`

numLags — Number of lags

`min(20,length(y)-1)` (default) | positive integer

Number of lags of the ACF that the software returns or plots, specified as a positive integer.

For example, `autocorr(y,10)` plots the ACF for lags 0 through 10.

Data Types: `double`

`numMA` — MA order

0 (default) | nonnegative integer

MA order that specifies the number of lags beyond which the theoretical ACF is effectively 0, specified as a nonnegative integer.

- `numMA` must be less than `numLags`.
- Specify `numMA` to assess whether the ACF is effectively zero beyond lag `numMA` [1]. The software uses Bartlett's approximation to estimate the large-lag standard error for lags that are greater than `numMA`.
- If `numMA = 0`, then the software assumes that `y` is a length T Gaussian white noise process. In this case, the standard error is approximately $\frac{1}{\sqrt{T}}$.

Example: `[~,~,bounds] = autocorr(y,[],5)`

Data Types: `double`

`numSTD` — Number of standard deviations

2 (default) | positive scalar

Number of standard deviations for the sample ACF estimation error assuming the theoretical ACF is 0 beyond lag `numMA`, specified as a positive scalar. For example, `autocorr(y,[],[],1.5)` plots the ACF with estimation error bounds 1.5 standard deviations away from 0.

If `numMA = 0` and `y` a length$T$ Gaussian process, then the confidence bounds are:

$$\pm\frac{\texttt{numSTD}}{\sqrt{T}}.$$

The default (`numSTD = 2`) corresponds to approximate 95% confidence bounds.

Data Types: `double`

Output Arguments

`acf` — Sample ACF

vector

Sample ACF of the univariate time series `y`, returned as a vector of length `numLags + 1`.

The elements of `acf` correspond to lags 0,1,2,...,`numLags`. The first element, which corresponds to lag 0, is unity (i.e., `acf(1) = 1`).

`lags` — Sample ACF lags

vector

Sample ACF lags, returned as a vector. Specifically, `lags = 0:numLags`.

`bounds` — Approximate confidence bounds

vector

Approximate confidence bounds of the ACF assuming `y` is an MA(`numMA`) process, returned as a two-element vector. `bounds` is approximate for `lags` > `numMA`.

More About

Autocorrelation Function

Measures the correlation between y_t and y_{t+k}, where $k = 0,...,K$ and y_t is a stochastic process.

According to [1], the formula for the autocorrelation for lag k is

$$r_k = \frac{c_k}{c_0},$$

where

- $$c_k = \frac{1}{T-1}\sum_{t=1}^{T-k}(y_t - \bar{y})(y_{t+k} - \bar{y}).$$

- c_0 is the sample variance of the time series.

The estimated standard error for the autocorrelation at lag k is

$$SE(r_k) = \sqrt{\frac{1}{T}\left(1 + 2\sum_{j=1}^{q} r_j^2\right)},$$

where q is the lag beyond which the theoretical ACF is effectively 0. If the series is completely random, then the standard error reduces to $1/\sqrt{T}$ [1].

Tips

To plot the ACF without confidence bounds, set `numSTD` to 0.

- "Box-Jenkins Methodology" on page 3-2
- "Autocorrelation and Partial Autocorrelation" on page 3-13

References

[1] Box, G. E. P., G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis: Forecasting and Control*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

[2] Hamilton, J. D. *Time Series Analysis*. Princeton, NJ: Princeton University Press, 1994.

See Also

`crosscorr` | `filter` | `parcorr`

Introduced before R2006a

chowtest

Chow test for structural change

Chow tests assess the stability of coefficients β in a multiple linear regression model of the form $y = X\beta + \varepsilon$. Data are split at specified break points. Coefficients are estimated in initial subsamples, then tested for compatibility with data in complementary subsamples.

Syntax

```
h = chowtest(X,y,bp)
h = chowtest(Tbl,bp)
h = chowtest(___,Name,Value)
[h,pValue,stat,cValue] = chowtest(___)
```

Description

`h = chowtest(X,y,bp)` returns test decisions (`h`) from conducting Chow tests on the multiple linear regression model `y` = `X`β + ε at the break points in `bp`.

`h = chowtest(Tbl,bp)` returns test decisions using the data in the tabular array `Tbl`. The first `numPreds` columns are the predictors (`X`) and the last column is the response (`y`).

`h = chowtest(___,Name,Value)` uses any of the input arguments in the previous syntaxes and additional options specified by one or more `Name,Value` pair arguments. For example, you can specify which type of Chow test to conduct or specify whether to include an intercept in the multiple regression model.

`[h,pValue,stat,cValue] = chowtest(___)` additionally returns p-values, test statistics, and critical values for the tests.

Examples

Test Consumption Model for Structural Change

Conduct Chow tests to assess whether there are structural changes in the equation for food demand around World War II.

Load the U.S. food consumption data set, which contains annual measurements from 1927 through 1962 with missing data due to the war.

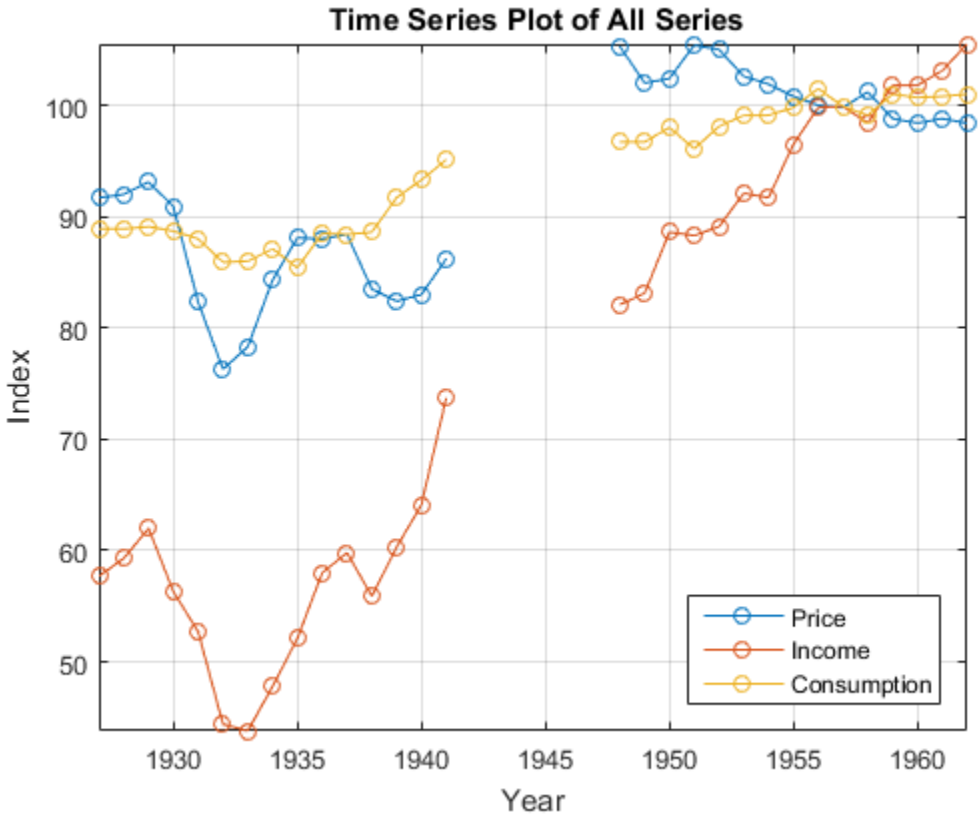
```
load Data_Consumption
```

For more details on the data, enter `Description` at the command prompt.

Suppose that you want to develop a model for consumption as determined by food prices and disposable income, and assess its stability through the economic shock through the war.

Plot the series.

```
P = Data(:,1); % Food price index
I = Data(:,2); % Disposable income index
Q = Data(:,3); % Food consumption index

figure;
plot(dates,[P I Q],'o-')
axis tight
grid on
xlabel('Year')
ylabel('Index')
title('{\bf Time Series Plot of All Series}')
legend({'Price','Income','Consumption'},'Location','SE')
```



Measurements are missing from 1942 through 1947, which correspond to World War II.

Assume that log consumption is a linear function of the logs of food price and income. That is,

$$\mathtt{LQ}_t = \beta_0 + \beta_1 \mathtt{LI}_t + \beta_2 \mathtt{LP} + \varepsilon_t.$$

ε_t is a Gaussian random variable with mean 0 and standard deviation σ^2.

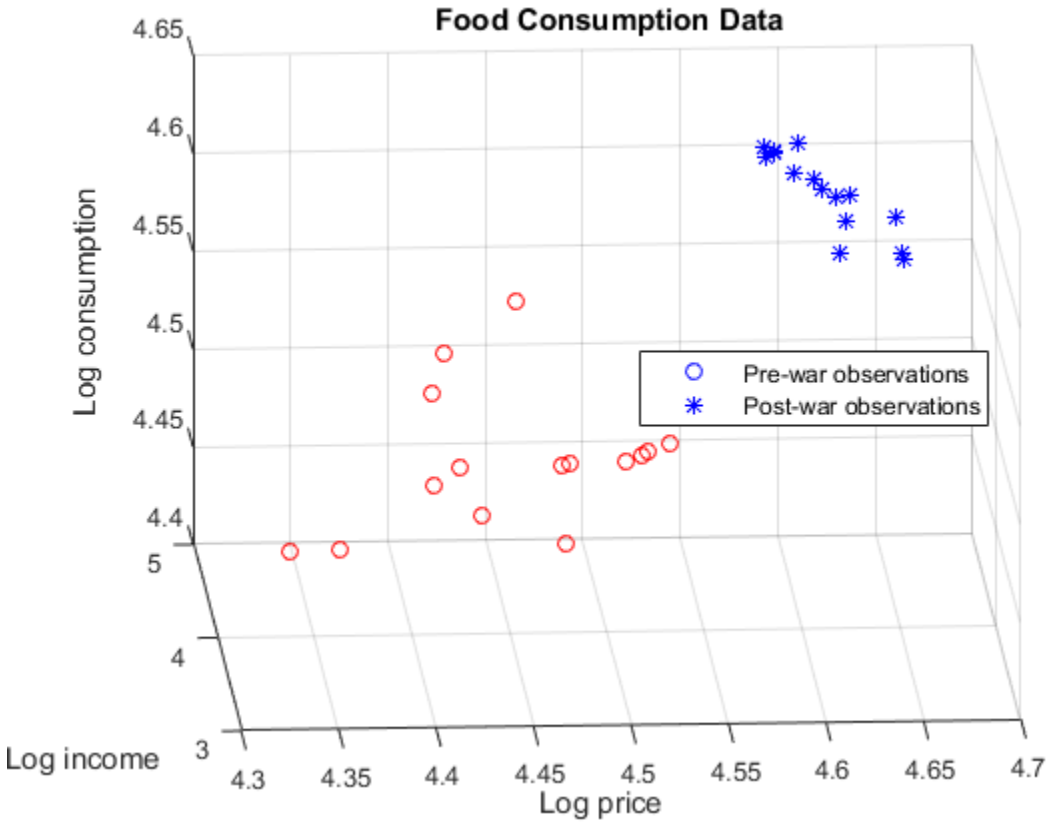
Apply the log transformation to each series.

```
LP = log(P);
```

```
LI = log(I);
LQ = log(Q);
```

Identify the indices before World War II. Plot log consumption with respect to the logs of food price and income.

```
preWarIdx = (dates <= 1941);

figure
scatter3(LP(preWarIdx),LI(preWarIdx),LQ(preWarIdx),[],'ro');
hold on
scatter3(LP(~preWarIdx),LI(~preWarIdx),LQ(~preWarIdx),[],'b*');
legend({'Pre-war observations','Post-war observations'},...
    'Location','Best')
xlabel('Log price')
ylabel('Log income')
zlabel('Log consumption')
title('{\bf Food Consumption Data}')
% Get a better view
h = gca;
h.CameraPosition = [4.3 -12.2 5.3];
```



Data relationships appear to be affected by the war.

Conduct two break point Chow tests at 5% level of significance. For the first test, set the break point at 1941. Set the break point of the other test at 1948.

```
bp = find(preWarIdx,1,'last');

h1941 = chowtest([LP LI],LQ,bp)
h1948 = chowtest([LP LI],LQ,bp + 1)
```

```
h1941 =

     1
```

```
h1948 =

     0
```

`h1941 = 1` indicates that there is significant evidence reject the null hypothesis that the coefficients are stable when the break points occur before the war. However, `h1948 = 0` indicates that there is not enough evidence to reject coefficient stability if the break point is after the war. This result suggests that the data at 1948 are influential.

Alternatively, you can supply a vector of break points to conduct three Chow tests.

```
h = chowtest([LP LI],LQ,[bp bp+1]);


RESULTS SUMMARY

***************
Test 1

Sample size: 30
Breakpoint: 15

Test type: breakpoint
Coefficients tested: All

Statistic: 5.5400
Critical value: 3.0088

P value: 0.0049
Significance level: 0.0500

Decision: Reject coefficient stability

***************
Test 2

Sample size: 30
Breakpoint: 16

Test type: breakpoint
Coefficients tested: All
```

```
Statistic: 1.2942
Critical value: 3.0088

P value: 0.2992
Significance level: 0.0500

Decision: Fail to reject coefficient stability
```

By default, `chowtest` displays a summary of the test results for each test when you conduct more than one test.

Test Model of Real U.S. GNP for Structural Change

Using the Chow test, assess the stability of an explanatory model of U.S. real gross national product (GNP) using the end of World War II as a break point.

Load the Nelson-Plosser data set.

```
load Data_NelsonPlosser
```

The time series in the data set contain annual, macroeconomic measurements from 1860 to 1970. For more details, a list of variables, and descriptions, enter `Description` in the command line.

Several series have missing data. Focus the sample to measurements from 1915 to 1970.

```
span = (1915 <= dates) & (dates <= 1970);
```

Assume that an appropriate multiple regression model to describe real GNP is

$$\mathrm{GNPR}_t = \beta_0 + \beta_1 \mathrm{IPI}_t + \beta_2 \mathrm{E}_t + \beta_3 \mathrm{WR}_t.$$

Collect the model variables into a tabular array. Position the predictors in the first three columns, and the response in the last column.

```
Mdl = DataTable(span,[4,5,10,1]);
```

Select the index corresponding to 1945, the end of World War II.

```
bp = find(strcmp(Mdl.Properties.RowNames,'1945'));
```

Using 1945 as a break point, conduct a break point test to assess whether all regression coefficients are stable.

```
h = chowtest(Mdl,bp)
```

```
h =

     1
```

`h = 1` indicates to reject the null hypothesis that the regression coefficients between the subsamples are equivalent.

In addition to returning a test decision, you can request that a test summary display in the Command Window.

```
h = chowtest(Mdl,bp,'Display','summary');
```

```
RESULTS SUMMARY

***************
Test 1

Sample size: 56
Breakpoint: 31

Test type: breakpoint
Coefficients tested: All

Statistic: 11.1036
Critical value: 2.5652

P value: 0.0000
Significance level: 0.0500

Decision: Reject coefficient stability
```

Assess Stability of Subsets of Regression Coefficients

Conduct a Chow test to assess the stability of a subset of regression coefficients. This example follows from “Test Consumption Model for Structural Change”.

Load the U.S. food consumption data set.

```
load Data_Consumption
P = Data(:,1);
I = Data(:,2);
Q = Data(:,3);
```

Apply the log transformation to each series.

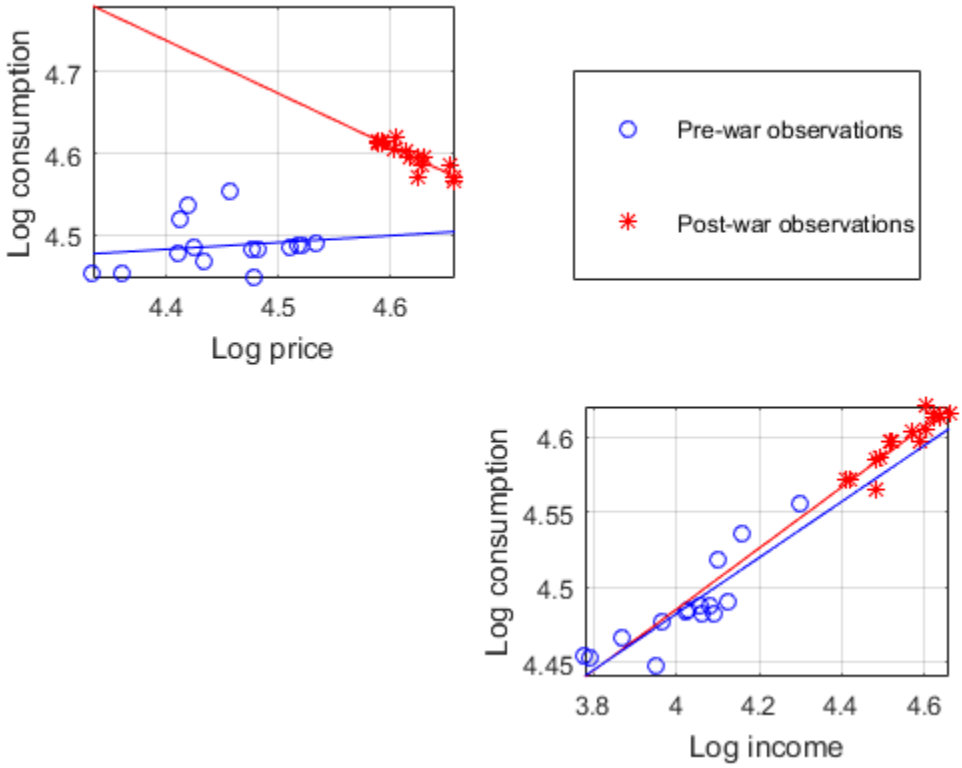
```
LP = log(P);
LI = log(I);
LQ = log(Q);
```

Identify the indices before World War II.

```
preWarIdx = (dates <= 1941);
```

Consider two regression models: one is log consumption onto log food price, and the other is log consumption onto log income. Plot scatter plots and regression lines for both models.

```
figure;
subplot(2,2,1)
plot(LP(preWarIdx),LQ(preWarIdx),'bo',LP(~preWarIdx),LQ(~preWarIdx),'r*');
axis tight
grid on
lsline;
xlabel('Log price')
ylabel('Log consumption')
legend('Pre-war observations','Post-war observations',...
    'Location',[0.6,0.6,0.25,0.25])
subplot(2,2,4)
plot(LI(preWarIdx),LQ(preWarIdx),'bo',LI(~preWarIdx),LQ(~preWarIdx),'r*');
axis tight
grid on
lsline
xlabel('Log income')
ylabel('Log consumption')
```



There is a clear break in food price elasticity between subsamples before and after the war. However, there doesn't appear to be such a break in income elasticity.

Conduct two Chow tests to determine whether there is statistical evidence to reject model continuity for both regression models. Because there are more observations in the complementary subsample than coefficients, conduct a break point test. Consider the elasticities in the test only. That is, specify `0` or `false` for the intercept (first coefficient), and `1` or `true` for elasticity (second coefficient).

```
bp = find(preWarIdx,1,'last'); % Index for 1941
chowtest(LP,LQ,bp,'Coeffs',[0 1],'Display','summary');
chowtest(LI,LQ,bp,'Coeffs',[0 1],'Display','summary');
```

```
RESULTS SUMMARY

***************
Test 1

Sample size: 30
Breakpoint: 15

Test type: breakpoint
Coefficients tested: 0  1

Statistic: 7.3947
Critical value: 4.2252

P value: 0.0115
Significance level: 0.0500

Decision: Reject coefficient stability


RESULTS SUMMARY

***************
Test 1

Sample size: 30
Breakpoint: 15

Test type: breakpoint
Coefficients tested: 0  1

Statistic: 0.1289
Critical value: 4.2252

P value: 0.7225
Significance level: 0.0500

Decision: Fail to reject coefficient stability
```

The first summary suggests to reject the null hypothesis that price elasticities are equivalent across subsamples at 5% level of significance. The second summary suggests to not reject the null hypothesis that income elasticities are equivalent across subsamples.

Consider a regression model of log consumption onto the logs of price and income. Conduct two break point tests: one that compares price elasticity across subsamples only, and another that compares income elasticity only.

```
chowtest([LP,LI],LQ,bp,'Coeffs',[0 1 0; 0 0 1]);
```

```
RESULTS SUMMARY

***************
Test 1

Sample size: 30
Breakpoint: 15

Test type: breakpoint
Coefficients tested: 0  1  0

Statistic: 0.0001
Critical value: 4.2597

P value: 0.9920
Significance level: 0.0500

Decision: Fail to reject coefficient stability

***************
Test 2

Sample size: 30
Breakpoint: 15

Test type: breakpoint
Coefficients tested: 0  0  1

Statistic: 2.8151
Critical value: 4.2597

P value: 0.1064
Significance level: 0.0500

Decision: Fail to reject coefficient stability
```

For both tests, there is not enough evidence to reject model stability at 5% level.

Model Structural Change

Simulate data for a linear model including a structural break in the intercept and one of the predictor coefficients. Then, choose specific coefficients to test for equality across a break point using the Chow test. Adjust parameters to assess the sensitivity of the Chow test.

Specify four predictors, 50 observations, and a break point at period 44 for the simulated linear model.

```
numPreds = 4;
numObs = 50;
bp = 44;
rng(1); % For reproducibility
```

Form the predictor data by specifying means for the predictors, and then adding random, standard Gaussian noise to each of the means.

```
mu = [0 1 2 3];
X = repmat(mu,numObs,1) + randn(numObs,numPreds);
```

Add a column of ones to the predictor data.

```
X = [ones(numObs,1) X];
```

Specify the true values of the regression coefficients, and that the intercept and the coefficient of the second predictor jump by 10%.

```
beta1 = [1 2 3 4 5]';                                % Initial subsample coefficients
beta2 = beta1 + [beta1(1)*0.1 0 beta1(3)*0.1 0 0 ]'; % Complementary subsample coeffici
X1 = X(1:bp,:);                                      % Initial subsample predictors
X2 = X(bp+1:end,:);                                  % Complementary subsample predicto
```

Specify a 2-by-5 logical matrix that indicates to first test the intercept and second regression coefficient, and then test all other coefficients.

```
test1 = [true false true false false];
Coeffs = [test1; ~test1]
```

```
Coeffs =

     1     0     1     0     0
```

```
     0     1     0     1     1
```

The null hypothesis for the first test (`Coeffs(1,:)`) is equality of the intercepts and the coefficients of the second predictor across subsamples. The null hypothesis for second test (`Coeffs(2,:)`) is equality of the first, third, and fourth predictors across subsamples.

Simulate data for the linear model

$$y = \begin{bmatrix} X1 & 0 \\ 0 & X2 \end{bmatrix} \begin{bmatrix} \text{beta1} \\ \text{beta2} \end{bmatrix} + \text{innov}.$$

Create `innov` as a vector of random Gaussian variates with mean zero and standard deviation 0.2.

```
sigma = 0.2;
innov = sigma*randn(numObs,1);
y = [X1 zeros(bp,size(X2,2)); zeros(numObs - bp,size(X1,2)) X2]*[beta1; beta2]...
    + innov;
```

Conduct the two break point tests indicated in `Coeffs`. Because there is an intercept in the predictor matrix `X` already, specify to suppress its inclusion in the linear model that `chowtest` fits.

```
chowtest(X,y,bp,'Intercept',false,'Coeffs',Coeffs,'Display','summary');
```

```
RESULTS SUMMARY

***************
Test 1

Sample size: 50
Breakpoint: 44

Test type: breakpoint
Coefficients tested: 1  0  1  0  0

Statistic: 5.7102
Critical value: 3.2317

P value: 0.0066
Significance level: 0.0500
```

```
Decision: Reject coefficient stability

***************
Test 2

Sample size: 50
Breakpoint: 44

Test type: breakpoint
Coefficients tested: 0  1  0  1  1

Statistic: 0.2497
Critical value: 2.8387

P value: 0.8611
Significance level: 0.0500

Decision: Fail to reject coefficient stability
```

At the default significance level, the Chow test correctly rejects the null hypothesis that there are no structural breaks at period `bp` for the intercept and the second coefficient, and correctly failed to reject the null hypothesis for the other coefficients.

Compare the break point test results to the results of the forecast test.

```
chowtest(X,y,bp,'Intercept',false,'Coeffs',Coeffs,'Test','forecast',...
    'Display','summary');
```

```
RESULTS SUMMARY

***************
Test 1

Sample size: 50
Breakpoint: 44

Test type: forecast
Coefficients tested: 1  0  1  0  0

Statistic: 3.7637
Critical value: 2.8451

P value: 0.0182
```

```
Significance level: 0.0500

Decision: Reject coefficient stability

***************
Test 2

Sample size: 50
Breakpoint: 44

Test type: forecast
Coefficients tested: 0  1  0  1  1

Statistic: 0.2135
Critical value: 2.6123

P value: 0.9293
Significance level: 0.0500

Decision: Fail to reject coefficient stability
```

In this case, the inferences from the tests are equivalent to those for the break point test.

- “Check Model Assumptions for Chow Test” on page 3-112
- “Power of the Chow Test” on page 3-123

Input Arguments

X — Predictor data

numeric matrix

Predictor data for the multiple linear regression model, specified as a `numObs`-by-`numPreds` numeric matrix.

`numObs` is the number of observations and `numPreds` is the number of predictor variables.

Data Types: `double`

y — Response data

numeric vector

Response data for the multiple linear regression model, specified as a `numObs`-by-1 numeric vector.

Data Types: `double`

`Tbl` — Combined predictor and response data
tabular array

Combined predictor and response data for the multiple linear regression model, specified as a `numObs`-by-`numPreds` + 1 tabular array.

The first `numPreds` columns of `Tbl` are the predictor data, and the last column is the response data.

Data Types: `table`

`bp` — Break points
positive integer | vector of positive integers

Break points for the tests, specified as a positive integer or a vector of positive integers.

Each break point is an index of a specific observation (row) in the data. The element `bp(j)` specifies to split the data into the initial and complementary samples indexed by `1:bp(j)` and `(bp(j) + 1):numObs`, respectively.

Data Types: `double`

Notes

- `NaN`s in the data indicate missing values. `chowtest` removes missing values using list-wise deletion. Removal of rows in the data reduces the effective sample size and changes the time base of the series.

- If `bp` is a scalar, then the number of tests, `numTests`, is the common dimension of name-value pair argument values. In this case, `chowtest` uses the same `bp` in each test. Otherwise, the length of `bp` determines `numTests`, and `chowtest` runs separate tests for each value in `bp`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Intercept',false,'Test','forecast'` specifies to exclude an intercept term from the regression model and to conduct a forecast test.

`'Intercept'` — Indicate whether to include intercept

`true` (default) | `false` | logical vector

Indicate whether to include an intercept when `chowtest` fits the regression model, specified as the comma-separated pair consisting of `'Intercept'` and `true`, `false`, or a logical vector of length `numTests`.

Value	Description
`true`	`chowtest` includes an intercept when fitting the regression model. `numCoeffs` = `numPreds` + 1.
`false`	`chowtest` does not include an intercept when fitting the regression model. `numCoeffs` = `numPreds`.

Example: `'Intercept',false(3,1)`

Data Types: `logical`

`'Test'` — Type of Chow test to conduct

`'breakpoint'` (default) | `'forecast'` | cell vector of strings

Type of Chow test to conduct, specified as the comma-separated pair consisting of `'Test'` and a string or cell vector of strings of length `numTests`.

Value	Description
`'breakpoint'` (default)	• `chowtest` directly assesses coefficient equality constraints using an F statistic. • Both subsamples must have more than `numCoeffs` observations.
`'forecast'`	• `chowtest` assess forecast performance using a modified F statistic. • The initial subsample must have more than `numCoeffs` observations.

For details on the value of `numCoeffs`, see the `'Intercept'` and `'Coeffs'` name-value pair arguments.

Example: `'Test',{'breakpoint' 'forecast'}`

Data Types: `char` | `cell`

`'Coeffs'` — Flags indicating which elements of β to test for equality

logical vector | logical array

Flags indicating which elements of β to test for equality, specified as the comma-separated pair consisting of a logical vector or array. Vector values must be of length `numCoeffs`. Array values must be of size `numTests`-by-`numCoeffs`.

If `'Intercept'` contains mixed logical values:

- `numCoeffs` is `numPreds` + 1
- `chowtest` ignores values in the first column of `'Coeffs'` for models without an intercept.

For example, suppose the regression model has three predictors (`numPreds` is 3) in a linear model, and you want to conduct two Chow tests (`numTests` is 2). Each test includes all regression parameters in the linear model. Also, you want `chowtest` to *fit* an intercept in the linear model for the first test only. Therefore, `Intercept` must be the logical array `[1 0]`. Because there is at least one model for which `chowtest` fits an intercept, `Coeffs` must be a 2-by-4 logical array (`numTests` is 2 and `numCoeffs` is `numPreds` + 1). The elements of `Coeffs(:,1)` correspond to whether to *test* the intercept irrespective of its presence in the model. Therefore, one way to specify `Coeffs` is `true(2,4)`. For the second test, `chowtest` does not fit an intercept, and so ignores the value `true` in `Coeffs(2,1)`. Because `chowtest` ignores `Coeffs(2,1)`, `Coeffs = [true(1,4); false true(1,3)]` yields the same result.

The default is `true(numTests,numCoeffs)`, which tests all of β for all tests.

Example: `'Coeffs',[false true; true true]`

`'Alpha'` — Nominal significance levels

`0.05` (default) | numeric scalar | numeric vector

Nominal significance levels for the tests, specified as the comma-separated pair consisting of `'Alpha'` and a numeric scalar or vector of length `numTests`. All elements of `Alpha` must be in the interval (0,1).

Example: `'Alpha',[0.5 0.1]`

Data Types: `double`

`'Display'` — Flag indicating whether to display test results in command window

`'off' | 'summary'`

Flag indicating whether to display test results in the command window, specified as the comma-separated pair consisting of `'Display'` and `'off'` or `'summary'`.

Value	Description	Default Value When
`'off'`	No display	`numTests` = 1
`'summary'`	For each test, display test information and results	`numTests` > 1

Example: `'Display','off'`

Data Types: `char`

Notes

- `chowtest` expands scalar and single-string input argument values, other than `'Display'`, to the size of `numTests`. Vector values and `'Coeffs'` arrays must share a common dimension, equal to `numTests`.

- If any of `bp`, `Intercept`, `Test`, or `Alpha` are row vectors, then all output arguments are row vectors.

Output Arguments

`h` — Test decisions

logical scalar | logical vector

Test decisions, returned as a logical scalar or logical vector of length `numTests`.

The null hypothesis (H_0) of the Chow test is that the coefficients (β) selected by `Coeffs` are identical across subsamples.

- `1` indicates rejection of H_0.

- `0` indicates failure to reject H_0.

`pValue` — *p*-values
numeric scalar | numeric vector

p-values, returned as a numeric scalar or vector of length `numTests`.

`stat` — Test statistics
numeric scalar | numeric vector

Test statistics, returned as a numeric scalar or vector of length `numTests`. For details, see "Chow Tests" on page 9-126.

`cValue` — Critical values for the tests
numeric scalar | numeric vector

Critical values for the tests, returned as a numeric scalar or vector of length `numTests`. `Alpha` determines the critical values.

More About

Chow Tests

Chow tests assess the stability of the coefficients (β) in a multiple linear regression model of the form $y = X\beta + \varepsilon$. Chow (1960) introduces two variations: the break point and forecast tests [1].

The break point test is a standard F test from the analysis of covariance. The forecast test makes use of the standard theory of prediction intervals. Chow's contribution is to place both tests general linear hypothesis framework, and develop appropriate test statistics for testing subsets of coefficients (see `Coeffs`). For test-statistic formulae, see [1].

Tips

- Chow tests assume continuity of the innovations variance across structural changes. Heteroscedasticity can distort the size and power of the test. You should verify the innovations-variance-continuity assumption holds before using the test results for inference.
- If both subsamples contain more than `numCoeffs` observations, then you can conduct a forecast test instead of a break point test. However, the forecast test might have

lower power relative to the break point test [1]. Nevertheless, Wilson (1978) suggests conducting the forecast test in the presence of unknown specification errors .

- You can apply the forecast test to cases where both subsamples have size greater than `numCoeffs`, where you would typically apply a breakpoint test. In such cases, the forecast test might have significantly reduced power relative to a break point test [1]. Nevertheless, Wilson (1978) suggests use of the forecast test in the presence of unknown specification errors [4].
- The forecast test is based on the unbiased predictions, with zero mean error, which result from stable coefficients. However, zero mean forecast error does not, in general, guarantee coefficient stability. Thus, forecast tests are most effective in checking for structural breaks, rather than model continuity [3].
- To obtain diagnostic statistics for each subsample, such as regression coefficient estimates, their standard errors, error sums of squares, etc., pass the appropriate data to `fitlm`. For details on working with LinearModel model objects, see "Multiple Linear Regression".

References

[1] Chow, G. C. "Tests of Equality Between Sets of Coefficients in Two Linear Regressions." *Econometrica*. Vol. 28, 1960, pp. 591–605.

[2] Fisher, F. M. "Tests of Equality Between Sets of Coefficients in Two Linear Regressions: An Expository Note." *Econometrica*. Vol. 38, 1970, pp. 361–66.

[3] Rea, J. D. "Indeterminacy of the Chow Test When the Number of Observations is Insufficient." *Econometrica*. Vol. 46, 1978, p. 229.

[4] Wilson, A. L. "When is the Chow Test UMP?" *The American Statistician*. Vol. 32, 1978, pp. 66–68.

See Also

`fitlm` | LinearModel

Introduced in R2015b

collintest

Belsley collinearity diagnostics

Syntax

```
collintest(X)
collintest(X,Name,Value)

sValue = collintest( ___ )
[sValue,condIdx,VarDecomp] = collintest( ___ )
```

Description

`collintest(X)` displays Belsley collinearity diagnostics for assessing the strength and sources of collinearity among variables in the matrix or tabular array `X`.

`collintest(X,Name,Value)` uses additional options specified by one or more `Name,Value` pairs.

`sValue = collintest( ___ )` returns the singular values in decreasing order, using any of the previous input arguments.

`[sValue,condIdx,VarDecomp] = collintest( ___ )` additionally returns the condition indices and variance decomposition proportions.

Examples

Display Belsley Collinearity Diagnostics

Display collinearity diagnostics for multiple time series.

Load data of Canadian inflation and interest rates.

```
load Data_Canada
```

Display the Belsley collinearity diagnostics, using all default options.

```
collintest(DataTable)
```

```
Variance Decomposition

 sValue  condIdx   INF_C   INF_G   INT_S   INT_M   INT_L
-------------------------------------------------------
 2.1748    1      0.0012  0.0018  0.0003  0.0000  0.0001
 0.4789   4.5413  0.0261  0.0806  0.0035  0.0006  0.0012
 0.1602  13.5795  0.3386  0.3802  0.0811  0.0011  0.0137
 0.1211  17.9617  0.6138  0.5276  0.1918  0.0004  0.0193
 0.0248  87.8245  0.0202  0.0099  0.7233  0.9979  0.9658
```

Only the last row in the display has a condition index larger than the default tolerance, 30. In this row, the last three variables (in the last three columns) have variance-decomposition proportions exceeding the default tolerance, 0.5. This suggests that the variables `INT_S`, `INT_M`, and `INT_L` exhibit multicollinearity.

Plot Belsley Collinearity Diagnostics

Plot collinearity diagnostics for multiple time series.

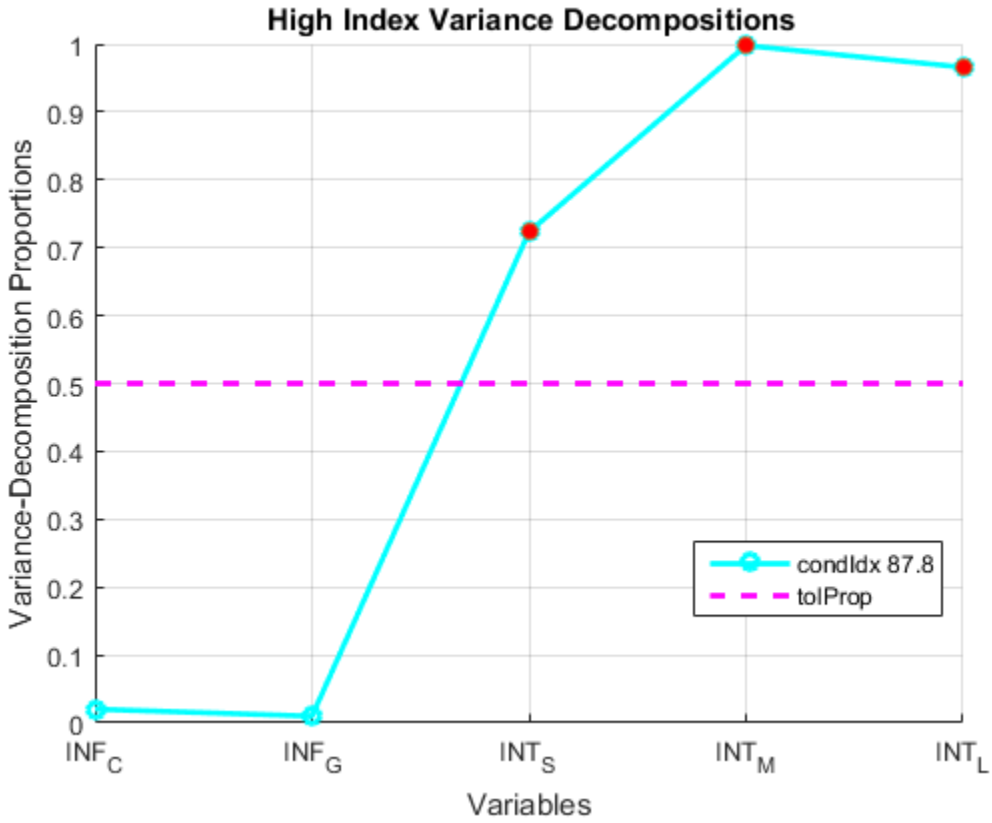
Load data of Canadian inflation and interest rates.

```
load Data_Canada
```

Plot the Belsley collinearity diagnostics using the `plot` option.

```
collintest(DataTable,'plot','on')
```

```
Variance Decomposition

 sValue  condIdx   INF_C   INF_G   INT_S   INT_M   INT_L
-------------------------------------------------------
 2.1748    1      0.0012  0.0018  0.0003  0.0000  0.0001
 0.4789   4.5413  0.0261  0.0806  0.0035  0.0006  0.0012
 0.1602  13.5795  0.3386  0.3802  0.0811  0.0011  0.0137
 0.1211  17.9617  0.6138  0.5276  0.1918  0.0004  0.0193
 0.0248  87.8245  0.0202  0.0099  0.7233  0.9979  0.9658
```



The plot corresponds to the values in the last row of variance-decomposition proportions, which is the only one with a condition index larger than the default tolerance, 30. The last three variables in this row have variance-decomposition proportions exceeding the default tolerance, 0.5, indicated by red markers in the plot.

Return Belsley Collinearity Diagnostics

Compute collinearity diagnostics for multiple time series and return the singular values, condition indices, and variance-decomposition proportions.

Load data of Canadian inflation and interest rates.

```
load Data_Canada
```

Compute the Belsley collinearity diagnostics. Turn off the results display using the `display` option.

```
[sv,conIdx,varDecomp] = collintest(DataTable,'display',...
    'off');
```

There is no display of the results.

Display the contents of `varDecomp`.

```
varDecomp
```

```
varDecomp =

    0.0012    0.0018    0.0003    0.0000    0.0001
    0.0261    0.0806    0.0035    0.0006    0.0012
    0.3386    0.3802    0.0811    0.0011    0.0137
    0.6138    0.5276    0.1918    0.0004    0.0193
    0.0202    0.0099    0.7233    0.9979    0.9658
```

The output argument `varDecomp` is a matrix of the variance-decomposition proportions. `sv` is a vector of singular values in descending order, and `condIdx` is a vector of condition indices in ascending order.

Input Arguments

`X` — Input regression variables

numeric matrix | tabular array

Input regression variables, specified as a `numObs`-by-`numVars` numeric matrix or tabular array. Each column of `X` corresponds to a variable, and each row corresponds to an observation. For models with an intercept, `X` should contain a column of ones.

`collintest` scales the columns of `X` to unit length before processing. Data in `X` should not be centered.

If `X` is a tabular array, then the variables must be numeric.

Data Types: `double` | `table`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'plot','on','tolIdx',35` displays a results plot with a tolerance index of 35

`'varNames'` — Variable names

cell vector of strings

Variable names used in displays and plots of the results, specified as the comma-separated pair consisting of `'varNames'` and a cell vector of strings. `varNames` must have length `numVars`, and each cell corresponds to a variable name. If an intercept term is present, then `varNames` must include the intercept term (e.g., include the name `'Const'`). The software truncates all variable names to the first five characters.

- If `X` is a matrix, then the default value of `varNames` is the cell vector of strings `{'var1','var2',...}`.
- If `X` is a tabular array, then the default value of `varNames` is the property `X.Properties.VariableNames`.

Example: `'varNames',{'Const','AGE','BBD'}`

Data Types: `cell`

`'display'` — Display results indicator

`'on'` (default) | `'off'`

Display results indicator for whether or not to display results in the Command Window, specified as the comma-separated pair consisting of `'display'` and one of `'on'` or `'off'`. If you specify the value `'on'`, then all outputs are displayed in tabular form.

Example: `'display','off'`

`'plot'` — Plot results indicator

`'off'` (default) | `'on'`

Plot results indicator for whether or not to plot results, specified as the comma-separated pair consisting of `'plot'` and one of `'on'` or `'off'`.

- If you specify the value `'on'`, then the plot shows the critical rows of the output `VarDecomp`, that is, those rows with condition indices above the input tolerance `tolIdx`.
- If a group of at least two variables in a critical row have variance-decomposition proportions above the input tolerance `tolProp`, then the group is identified with red markers.

Example: `'plot','on'`

`'tolIdx'` — Condition index tolerance

`30` (default) | scalar value of at least 1

Condition index tolerance, specified as the comma-separated pair consisting of `'tolIdx'` and a scalar value of at least one. `collintest` uses the tolerance to decide which indices are large enough to infer a near dependency in the data. The `tolIdx` value is only used when `plot` has the value `'on'`.

Example: `'tolIdx',25`

`'tolProp'` — Variance-decomposition proportion tolerance

`0.5` (default) | scalar between 0 and 1

Variance-decomposition proportion tolerance, specified as the comma-separated pair consisting of `'tolProp'` and a scalar value between zero and one. `collintest` uses the tolerance to decide which variables are involved in any near dependency. The `tolProp` value is only used when `plot` has the value `'on'`.

Example: `'tolProp',0.4`

Output Arguments

`sValue` — Singular values

vector in descending order

Singular values of scaled `X`, returned as a vector. The elements of `sValue` are in descending order.

`condIdx` — Condition indices

vector in ascending order

Condition indices, returned as a vector with elements in ascending order. All condition indices have value between one and the condition number of scaled `X`. Large indices

identify near dependencies among the variables in `X`. The size of the indices is a measure of how near dependencies are to collinearity.

`VarDecomp` — Variance-decomposition proportions
matrix

Variance-decomposition proportions, returned as a `numVars`-by-`numVars` matrix. Large proportions, combined with a large condition index, identify groups of variables involved in near dependencies. The size of the proportions is a measure of how badly the regression is degraded by the dependency.

More About

Belsley Collinearity Diagnostics

Belsley collinearity diagnostics assess the strength and sources of collinearity among variables in a multiple linear regression model.

To assess collinearity, the software computes singular values of the scaled variable matrix, *X*, and then converts them to condition indices. The conditional indices identify the number and strength of any near dependencies between variables in the variable matrix. The software decomposes the variance of the ordinary least squares (OLS) estimates of the regression coefficients in terms of the singular values to identify variables involved in each near dependency, and the extent to which the dependencies degrade the regression.

Condition Indices

The *condition indices* for a scaled matrix *X* identify the number and strength of any near dependencies in *X*.

For scaled matrix *X* with *p* columns and singular values $S_{(1)} \geq S_{(2)} \geq \ldots \geq S_{(p)}$, the condition indices for the columns of *X* are $S_{(1)}/S_{(j)},\ j = 1,\ldots,p$.

All condition indices are bounded between one and the condition number.

Condition Number

The *condition number* of a scaled matrix *X* is an overall diagnostic for detecting collinearity.

For scaled matrix X with p columns and singular values $S_{(1)} \geq S_{(2)} \geq \ldots \geq S_{(p)}$, the condition number is $S_{(1)}/S_{(p)}$.

The condition number achieves its lower bound of one when the columns of scaled X are orthonormal. The condition number rises as variates exhibit greater dependency.

A limitation of the condition number as a diagnostic is that it fails to provide specifics on the strength and sources of any near dependencies.

Multiple Linear Regression Model

A *multiple linear regression model* is a model of the form $Y = X\beta + \varepsilon$. X is a design matrix of regression variables, and β is a vector of regression coefficients.

Singular Values

The *singular values* of a scaled matrix X are the diagonal elements of the matrix S in the singular-value decomposition USV'.

In descending order, the singular values of the scaled matrix X with p columns are $S_{(1)} \geq S_{(2)} \geq \ldots \geq S_{(p)}$.

Variance-Decomposition Proportions

Variance-decomposition proportions identify groups of variates involved in near dependencies, and the extent to which the dependencies degrade the regression.

From the singular value decomposition USV' of scaled design matrix X (with p columns), let:

- V be the matrix of orthonormal eigenvectors of XX
- $S_{(1)} \geq S_{(2)} \geq \ldots \geq S_{(p)}$ be the ordered diagonal elements of the matrix S

The variance of the OLS estimate of the ith multiple linear regression coefficient, β_i, is proportional to the sum

$$V(i,1)^2/S_{(1)}^2 + V(i,2)^2/S_{(2)}^2 + \ldots + V(i,p)^2/S_{(p)}^2,$$

where $V(i,j)$ denotes the (*i*,*j*)th element of *V*.

The (*i*,*j*)th variance-decomposition proportion is the proportion of the *j*th term in the sum relative to the entire sum, *j* = 1,...,*p*.

The terms $S_{(j)}^2$ are the eigenvalues of scaled $X'X$. Thus, large variance-decomposition proportions correspond to small eigenvalues of $X'X$, a common diagnostic for collinearity. The singular-value decomposition provides a more direct, numerically stable view of the eigensystem of scaled $X'X$.

Tips

- For purposes of collinearity diagnostics, Belsley [1] shows that column scaling of the design matrix, `X`, is always desirable. However, he also shows that centering the data in `X` is undesirable. For models with an intercept, if you center the data in `X`, then the role of the constant term in any near dependency is hidden, and yields misleading diagnostics.
- Tolerances for identifying large condition indices and variance-decomposition proportions are comparable to critical values in standard hypothesis tests. Experience determines the most useful tolerance, but experiments suggest the `collintest` defaults are good starting points [1].

References

[1] Belsley, D. A., E. Kuh, and R. E. Welsh. *Regression Diagnostics*. New York, NY: John Wiley & Sons, Inc., 1980.

[2] Judge, G. G., W. E. Griffiths, R. C. Hill, H. Lütkepohl, and T. C. Lee. *The Theory and Practice of Econometrics*. New York, NY: John Wiley & Sons, Inc., 1985.

See Also

`corrplot`

Introduced in R2012a

Conditional Variance Model Properties

Specify conditional variance model functional form and parameter values

Conditional variance model properties specify the functional form and parameter values of `garch`, `egarch`, and `gjr` model objects.

Modify the functional form of the model or set values for parameters by changing property values. Use dot notation to refer to a particular model object and property. For example, one way to include a conditional mean offset to a GARCH(1,1) model for estimation is to enter:

```
Mdl = garch(1,1);
Mdl.Offset = NaN;
```

For more information about conditional variance model objects, see:

- Using garch Objects
- Using egarch Objects
- Using gjr Objects

For GARCH, EGARCH, and GJR Models

`P` — Largest lag among past conditional variance terms

nonnegative integer

This property is read only.

Largest lag among the past conditional variance terms, specified as a nonnegative integer. That is, `P` is the degree of the GARCH polynomial.

For GARCH and GJR models, `P` also specifies the minimum number of presample conditional variances the software requires to initiate the model. For EGARCH models, `P` specifies the largest lag among all past log conditional variances.

If you use name-value pair arguments to create a conditional variance model, then the software implements one of these alternatives:

- If you specify `'GARCHLags'`, then `P` is the largest lag therein.
- If you specify `'GARCH'`, then `P` is the number of elements therein.

- Otherwise, `P` is `0`.

Data Types: `double`

`Q` — Largest lag among past innovation terms

nonnegative integer

This property is read only.

Largest lag among the past innovation terms, specified as a nonnegative integer. That is:

- For GARCH models, `Q` is the degree of the ARCH polynomial.
- For EGARCH and GJR models, `Q` is the maximum degree between the ARCH and leverage polynomials.

`Q` also specifies the minimum number of presample innovations the software needs to initiate the model.

If you use name-value pair arguments to create a conditional variance model, then the software implements one of these alternatives:

- If you specify `'ARCHLags'`, then `Q` is the largest lag therein.
- If you specify `'ARCH'`, then `Q` is the number of elements therein.
- Otherwise, `Q` is `0`.
- For EGARCH and GJR models, `Q` is the maximum between the largest ARCH lag (as determined by the previous items) and the largest leverage lag.

Note: For EGARCH and GJR models, the software defines the property `Q` as the largest lag associated with nonzero `ARCH` and `Leverage` coefficients, or `max(ARCHLags,LeverageLags)`. Typically, the number and corresponding lags of nonzero `ARCH` and `Leverage` coefficients are equivalent, but this is not a requirement. In other words, the lengths of `ARCH` and `Leverage` might differ.

Data Types: `double`

`Constant` — Conditional variance model constant

`NaN` (default) | scalar

Conditional variance model constant, specified as a scalar.

For GARCH and GJR models, `Constant` is positive.

Data Types: `double`

`GARCH` — Coefficients corresponding to nonzero past conditional variance terms

cell vector of `NaN`s (default) | cell vector of scalars

Coefficients corresponding to nonzero past conditional variance terms, specified as a cell vector of scalars. For GARCH and GJR models, `GARCH` contains positive coefficients.

If you did not specify `'GARCHLags'` to create the model object, then the coefficients in `GARCH` correspond to lags 1 through `P`.

If you specified `'GARCHLags'` to create the model object, then the elements of `GARCH` correspond to the elements of `GARCHLags`. That is, the conditional variance that is `GARCHLag(j)` periods in the past has coefficient `GARCH{j}`. The length of `GARCH` and `GARCHLags` are equivalent.

The software subjects `GARCH` coefficients to a near-zero tolerance exclusion test. That is, the software:

1. Creates lag operator polynomials for each of the `GARCH` component
2. Compares each coefficient to the default lag operator zero tolerance, `1e-12`
3. Includes a coefficient in the model if its magnitude is greater than `1e-12`, and excludes the coefficient otherwise (i.e., the software considers excluded coefficients sufficiently close to zero).

For details, see LagOp.

Data Types: `cell`

`ARCH` — Coefficients corresponding to nonzero past innovation terms

cell vector of `NaN`s (default) | cell vector of scalars

Coefficients corresponding to the nonzero past innovation terms, specified as a cell vector of scalars. For GARCH and GJR models, `ARCH` contains the positive coefficients associated with the squared innovation terms. For EGARCH models, `ARCH` contains the coefficients associated with the magnitude of the nonzero past standardized innovations.

If you did not specify `'ARCHLags'` to create the model object, then the coefficients in `ARCH` correspond to lags 1 through the length of `ARCH`.

If you specified 'ARCHLags' to create the model object, then the elements of ARCH correspond to the elements of ARCHLags. That is, the innovation that is ARCHLags(j) periods in the past has coefficient ARCH{j}. The length of ARCH and ARCHLags are equivalent.

The software subjects ARCH coefficients to a near-zero tolerance exclusion test. That is, the software:

1. Creates lag operator polynomials for each of the ARCH component
2. Compares each coefficient to the default lag operator zero tolerance, 1e-12
3. Includes a coefficient in the model if its magnitude is greater than 1e-12, and excludes the coefficient otherwise (i.e., the software considers excluded coefficients sufficiently close to zero).

For details, see LagOp.

Data Types: cell

UnconditionalVariance — Model unconditional variance

positive scalar

This property is read only.

The model unconditional variance σ_ε^2, specified as a positive scalar.

- For GARCH models,

$$\sigma_\varepsilon^2 = \frac{\kappa}{(1 - \sum_{i=1}^{P}\gamma_i - \sum_{j=1}^{Q}\alpha_j)}.$$

- For EGARCH models,

$$\sigma_\varepsilon^2 = \exp\left\{\frac{\kappa}{(1 - \sum_{i=1}^{P}\gamma_i)}\right\}.$$

  This estimator is biased for the true unconditional variance.

- For GJR models,

$$\sigma_\varepsilon^2 = \frac{\kappa}{\left(1 - \sum_{i=1}^{P}\gamma_i - \sum_{j=1}^{Q}\alpha_j - \frac{1}{2}\sum_{j=1}^{Q}\xi_j\right)}.$$

In the formulas, κ is the conditional variance model constant.

Data Types: `double`

`Offset` — Innovation mean model offset

`0` (default) | scalar

Innovation mean model offset, or additive constant, specified as a scalar.

Data Types: `double`

`Distribution` — Conditional probability distribution of underlying disturbance process

data structure

Conditional probability distribution of the underlying disturbance process, specified as a data structure.

The `Name` field stores the name of the distribution, which contains the string `'Gaussian'` for the Gaussian distribution or `'t'` for the *t* distribution.

If `Name` is `'t'`, then `Distribution` also contains the `DoF` field, which is the *t*-distribution degrees of freedom.

By default, `Distribution` is `struct('Name','Gaussian')`. When you create the model object, if you specify that the innovations process has a *t* distribution using the `'Distribution'` name-value pair argument, then the `DoF` field is `NaN` by default.

Example: `Mdl.Distribution = struct('t',5);`

Data Types: `struct`

For EGARCH and GJR Models

`Leverage` — Coefficients corresponding to nonzero past leverage terms

cell vector of `NaN`s (default) | cell vector of scalars

Coefficients corresponding to the nonzero past leverage terms, specified as a cell vector of scalars.

If you do not specify 'LeverageLags' to create the model object, then the coefficients in Leverage correspond to lags 1 through the length of Leverage.

If you specify 'LeverageLags' to create the model object, then the elements of Leverage correspond to the elements of LeverageLags. That is, the innovation that is LeverageLags(j) periods in the past has coefficient Leverage{j}. The length of Leverage and LeverageLags are equivalent.

The software subjects Leverage coefficients to a near-zero tolerance exclusion test. That is, the software:

1. Creates lag operator polynomials for each of the Leverage component
2. Compares each coefficient to the default lag operator zero tolerance, 1e-12
3. Includes a coefficient in the model if its magnitude is greater than 1e-12, and excludes the coefficient otherwise (i.e., the software considers excluded coefficients sufficiently close to zero).

For details, see LagOp.

Data Types: cell

See Also

egarch | garch | gjr

More About

- "Conditional Variance Models" on page 6-2
- "GARCH Model" on page 6-3
- "EGARCH Model" on page 6-4
- "GJR Model" on page 6-6

Introduced in R2012a

corrplot

Plot variable correlations

Syntax

```
corrplot(X)
corrplot(X,Name,Value)

R = corrplot( ___ )
[R,PValue] = corrplot( ___ )
```

Description

`corrplot(X)` creates a matrix of plots showing correlations among pairs of variables in `X`. Histograms of the variables appear along the matrix diagonal; scatter plots of variable pairs appear off diagonal. The slopes of the least-squares reference lines in the scatter plots are equal to the displayed correlation coefficients.

`corrplot(X,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

`R = corrplot( ___ )` returns the correlation matrix of `X` displayed in the plots. You can use any of the previous input arguments.

`[R,PValue] = corrplot( ___ )` additionally returns the *p*-values corresponding to the elements of `R`, used to test the null hypothesis of no correlation against the alternative of a nonzero correlation.

Examples

Plot Pearson's Correlation Coefficients

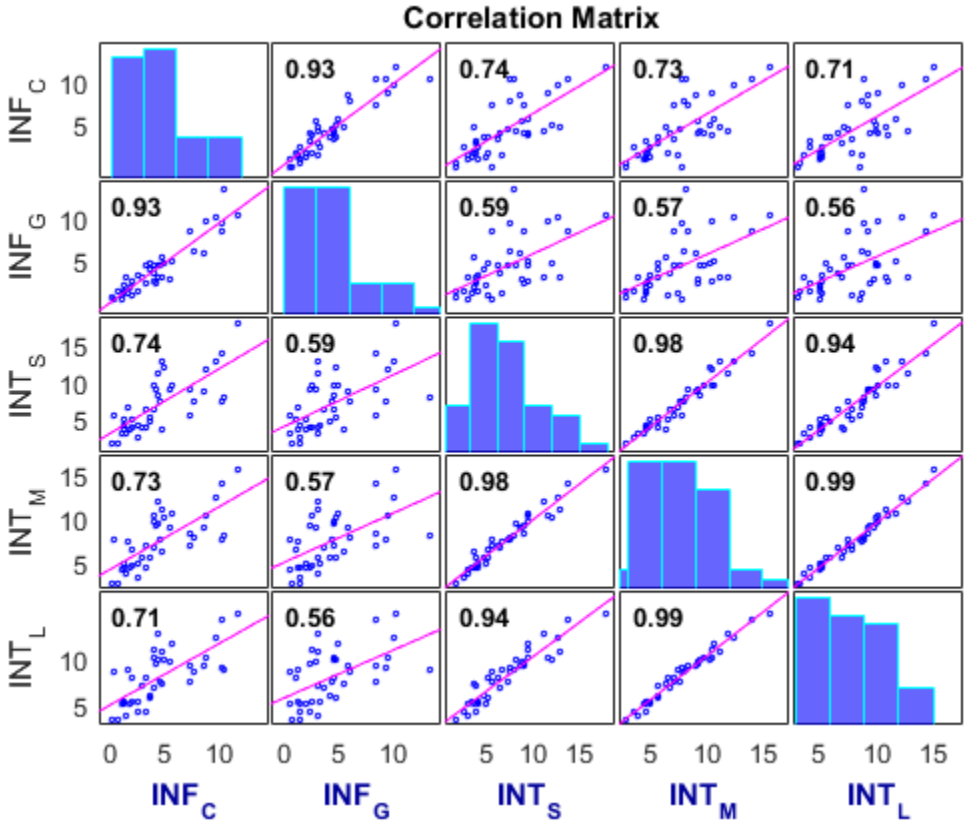
Plot correlations between multiple time series.

Load data on Canadian inflation and interest rates.

```
load Data_Canada
```

Plot the Pearson's linear correlation coefficients between all pairs of variables.

```
corrplot(DataTable)
```



The correlation plot shows that the short-term, medium-term, and long-term interest rates are highly correlated.

To examine the timestamp of a datum, enter `gname(dates)` into the Command Window, and the software presents an interactive cross hair over the plot. To expose the timestamp of a datum, click it using the cross hair.

Plot and Test Kendall's Rank Correlation Coefficients

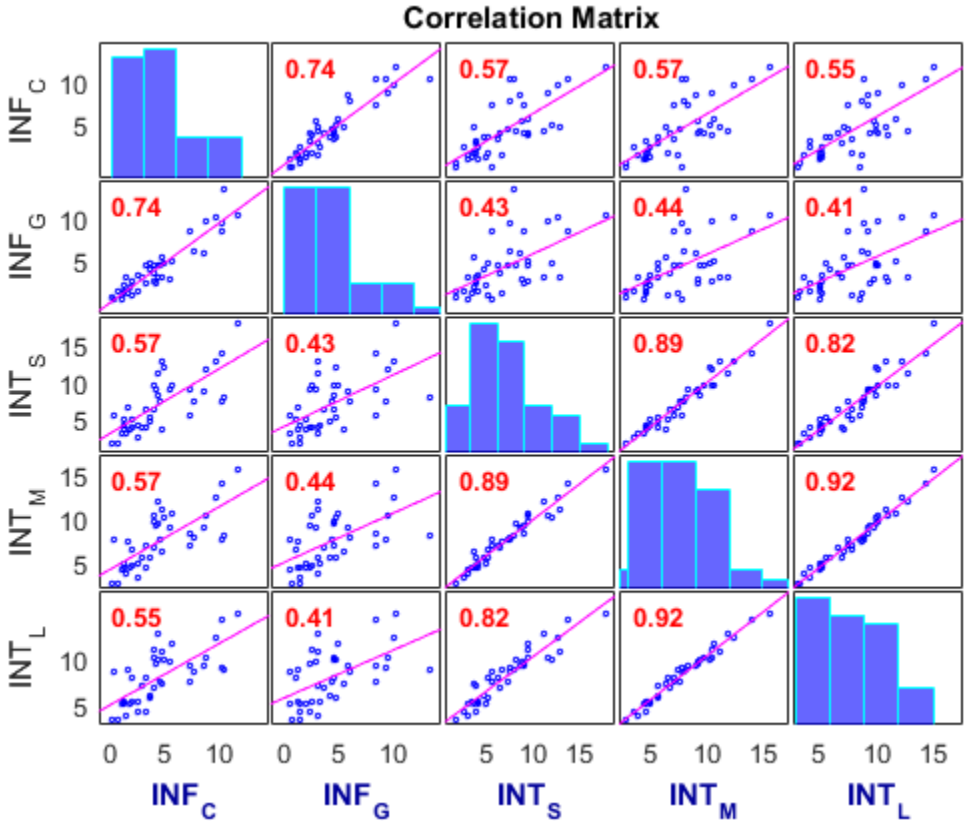
Plot Kendall's rank correlations between multiple time series. Conduct a hypothesis test to determine which correlations are significantly different from zero.

Load data on Canadian inflation and interest rates.

```
load Data_Canada
```

Plot the Kendall's rank correlation coefficients between all pairs of variables. Specify a hypothesis test to determine which correlations are significantly different from zero.

```
corrplot(DataTable,'type','Kendall','testR','on')
```



The correlation coefficients highlighted in red indicate which pairs of variables have correlations significantly different from zero. For these time series, all pairs of variables have correlations significantly different from zero.

Conduct Right-Tailed Correlation Tests

Test for correlations greater than zero between multiple time series.

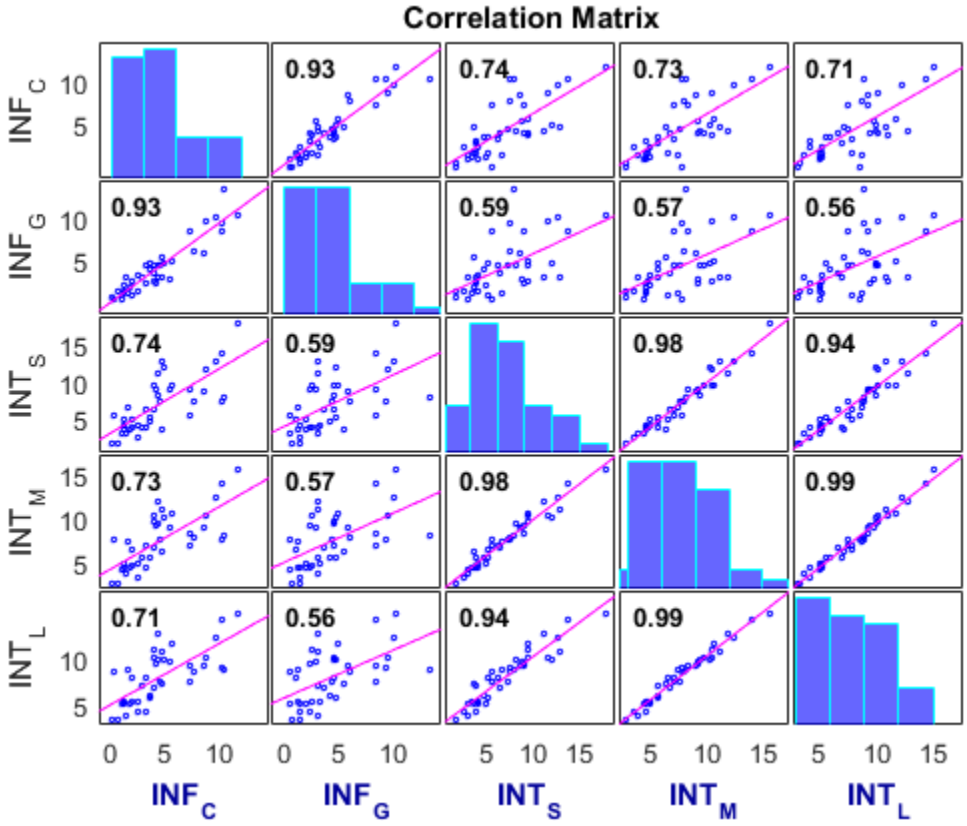
Load data on Canadian inflation and interest rates.

```
load Data_Canada
```

Return the pairwise Pearson's correlations and corresponding p-values for testing the null hypothesis of no correlation against the right-tailed alternative that the correlations are greater than zero.

```
[R,PValue] = corrplot(DataTable,'tail','right');
PValue
```

```
PValue =

    1.0000    0.0000    0.0000    0.0000    0.0000
    0.0000    1.0000    0.0000    0.0000    0.0001
    0.0000    0.0000    1.0000    0.0000    0.0000
    0.0000    0.0000    0.0000    1.0000    0.0000
    0.0000    0.0001    0.0000    0.0000    1.0000
```



The output `PValue` has pairwise p-values all less than the default 0.05 significance level, indicating that all pairs of variables have correlation significantly greater than zero.

Input Arguments

`X` — Data series

numeric matrix | tabular array

Data series that `corrplot` uses to plot correlations, specified as a `numObs`-by-`numVars` numeric matrix or tabular array. `X` consists of `numObs` observations made on `numVars` variables, and plots the correlations between the `numVars` variables.

If X is a tabular array, then the variables must be numeric.

Data Types: double | table

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: 'tails','right','alpha',0.1 specifies right-tailed tests at the 0.1 significance level

'type' — Correlation coefficient

'Pearson' (default) | 'Kendall' | 'Spearman'

Correlation coefficient to compute, specified as the comma-separated pair consisting of 'type' and one of the following:

'Pearson'	Pearson's linear correlation coefficient
'Kendall'	Kendall's rank correlation coefficient (τ)
'Spearman'	Spearman's rank correlation coefficient (ρ)

Example: 'type','Kendall'

'rows' — Option for handling rows with NaN values

'pairwise' (default) | 'all' | 'complete'

Option for handling rows with NaN values, specified as the comma-separated pair consisting of 'rows' and one of the following:

'all'	Use all rows, regardless of NaNs.
'complete'	Use only rows with no NaNs.
'pairwise'	Use rows with no NaNs in column i or j to compute $R(i,j)$.

Example: 'rows','complete'

'tail' — Alternative hypothesis

'both' (default) | 'right' | 'left'

Alternative hypothesis (H_a) used to compute the p-values, specified as the comma-separated pair consisting of 'tail' and one of the following:

'both'	H_a: Correlation is not zero.
'right'	H_a: Correlation is greater than zero.
'left'	H_a: Correlation is less than zero.

Example: 'tail','left'

'varNames' — Variable names

cell array of strings

Variable names to be used in the plots, specified as the comma-separated pair consisting of 'varNames' and a cell array of strings with numVars names. All variable names are truncated to the first five characters.

- If X is a matrix, then the default variable names are {'var1','var2',...}.
- If X is a tabular array, then the default variable names are X.Properties.VariableNames.

Example: 'varNames',{'CPF','AGE','BBD'}

'testR' — Significance tests indicator

'off' (default) | 'on'

Significance tests indicator for whether or not to test for significant correlations, specified as the comma-separated pair consisting of 'testR' and one of 'off' or 'on'. If you specify the value 'on', significant correlations are highlighted in red in the correlation matrix plot.

Example: 'testR','on'

'alpha' — Significance level

0.05 (default) | scalar between 0 and 1

Significance level for tests of correlation, specified as a scalar between 0 and 1.

Example: 'alpha',0.01

Output Arguments

`R` — Correlations

matrix

Correlations between pairs of variables in `X` that are displayed in the plots, returned as a `numVars`-by-`numVars` matrix.

`PValue` — *p*-values

matrix

p-values corresponding to significance tests on the elements of `R`, returned as a `numVars`-by-`numVars` matrix. The p-values are used to test the hypothesis of no correlation against the alternative of nonzero correlation.

More About

Tips

- The option `'rows','pairwise'`, which is the default, can return a correlation matrix that is not positive definite. The `'complete'` option always returns a positive-definite matrix, but in general the estimates are based on fewer observations.
- Use `gname` to identify points in the plots.

Algorithms

The software computes:

- *p*-values for Pearson's correlation by transforming the correlation to create a *t*-statistic with `numObs` – 2 degrees of freedom. The transformation is exact when `X` is normal.
- *p*-values for Kendall's and Spearman's rank correlations using either the exact permutation distributions (for small sample sizes) or large-sample approximations.
- *p*-values for two-tailed tests by doubling the more significant of the two one-tailed *p*-values.

See Also

`collintest` | `corr` | `gname`

Introduced in R2012a

crosscorr

Sample cross-correlation

Syntax

```
crosscorr(y1,y2)
crosscorr(y1,y2,numLags)
crosscorr(y1,y2,numLags,numSTD)

xcf = crosscorr(y1,y2)
xcf = crosscorr(y1,y2,numLags)
xcf = crosscorr(y1,y2,numLags,numSTD)
[xcf,lags,bounds] = crosscorr( ___ )
```

Description

`crosscorr(y1,y2)` plots the sample cross correlation (XCF) between the two univariate, stochastic time series `y1` and `y2` with confidence bounds.

`crosscorr(y1,y2,numLags)` plots the XCF, where `numLags` indicates the number of lags in the sample XCF.

`crosscorr(y1,y2,numLags,numSTD)` plots the XCF, where `numSTD` specifies the number of standard deviations of the sample XCF estimation error.

`xcf = crosscorr(y1,y2)` returns the sample cross-correlation function (XCF) between the two univariate, stochastic time series `y1` and `y2`.

`xcf = crosscorr(y1,y2,numLags)` returns the XCF, where `numLags` specifies the number of lags in the sample XCF.

`xcf = crosscorr(y1,y2,numLags,numSTD)` returns the XCF, where `numSTD` specifies the number of standard deviations of the sample ACF estimation error.

`[xcf,lags,bounds] = crosscorr( ___ )` additionally returns the lags (`lags`) corresponding to the ACF and the approximate upper and lower confidence bounds (`bounds`), using any of the input arguments in the previous syntaxes.

Examples

Plot the Cross Covariance of Two Time Series

Generate 100 random deviates from a Gaussian distribution with mean 0 and variance 1.

```
rng(1); % For reproducibility
x = randn(100,1);
```

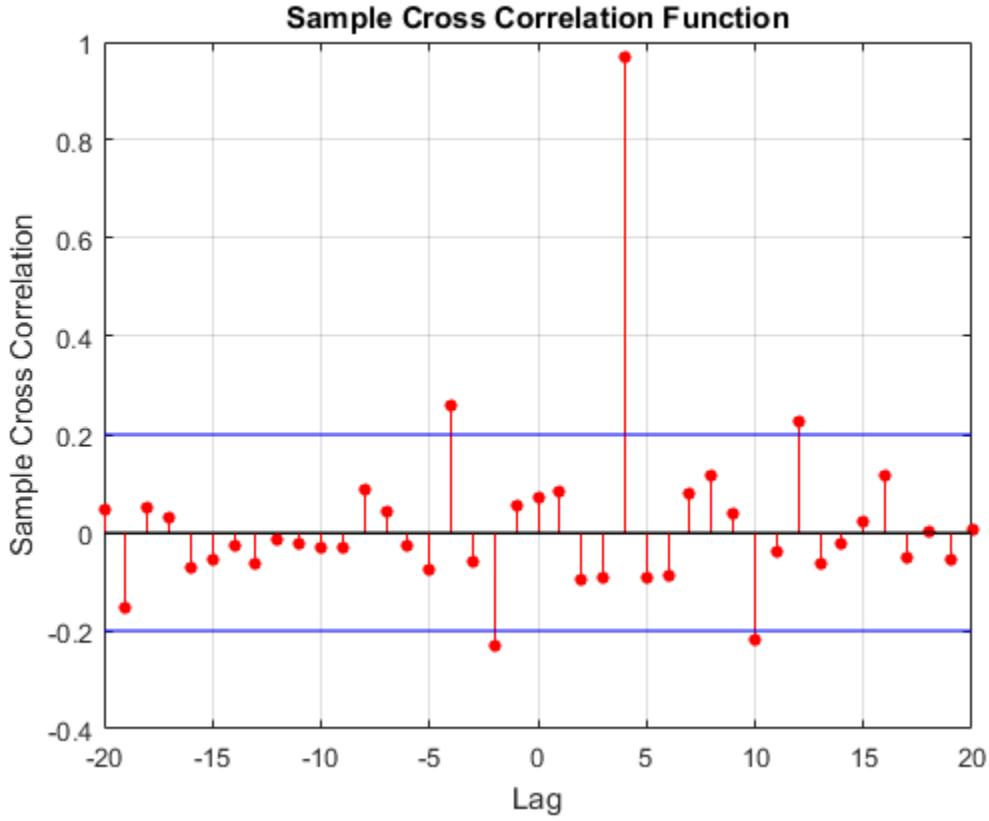
Create a 4-period delayed version of `x`.

```
y = lagmatrix(x,4);
```

Compute and plot the XCF.

```
y(isnan(y)) = 0; % crosscorr does not accept NaNs
[XCF,lags,bounds] = crosscorr(x,y);
bounds

crosscorr(x,y)
```

```
bounds =

    0.2000
   -0.2000
```



`bounds` displays the upper and lower confidence bounds, which are the horizontal lines in the XCF plot. As you should expect, `XCF` peaks at lag 4.

Specify More Lags for the Cross-Correlation Plot

Specify the AR(1) model for the first series:

$$y_{1t} = 2 + 0.3y_{1t-1} + \varepsilon_t,$$

where ε_t is Gaussian with mean 0 and variance 1.

```
MdlY1 = arima('AR',0.3,'Constant',2,'Variance',1);
```

Simulate data from `Mdl`.

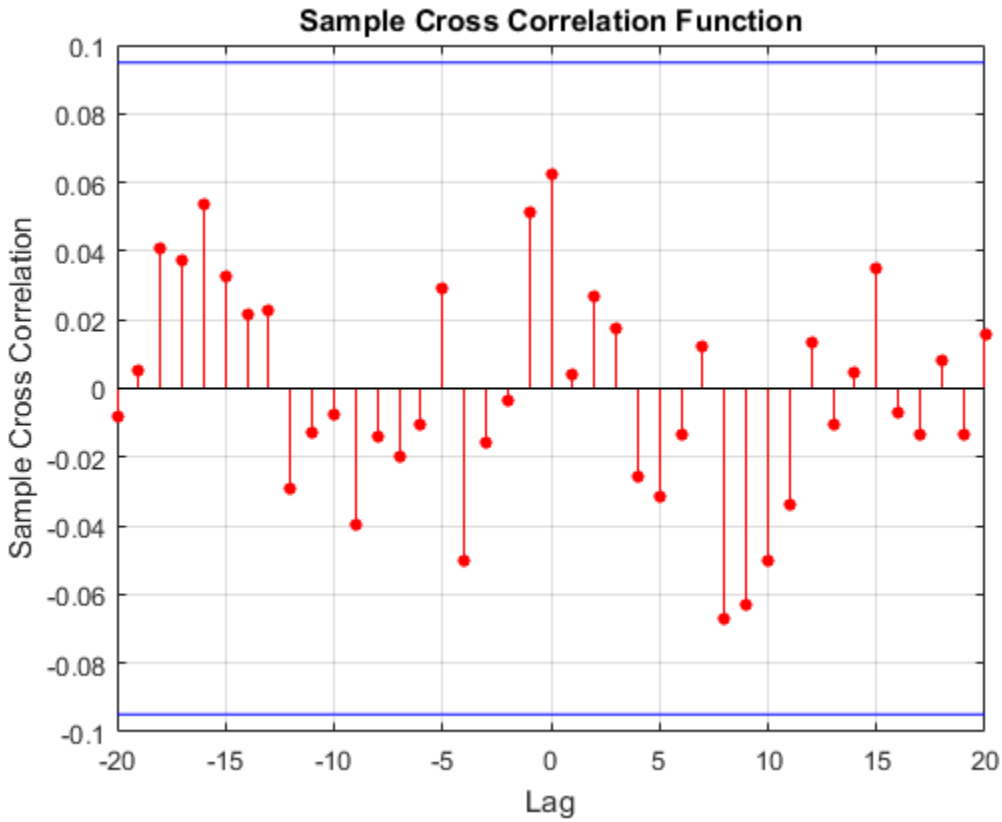
```
rng(1);
T = 1000;
y1 = simulate(MdlY1,T);
```

Simulate data for the second series by inducing correlation at lag 36.

```
y2 = [randn(36,1);y1(1:end-36)+randn(T-36,1)*0.1];
```

Plot the XCF using the default settings.

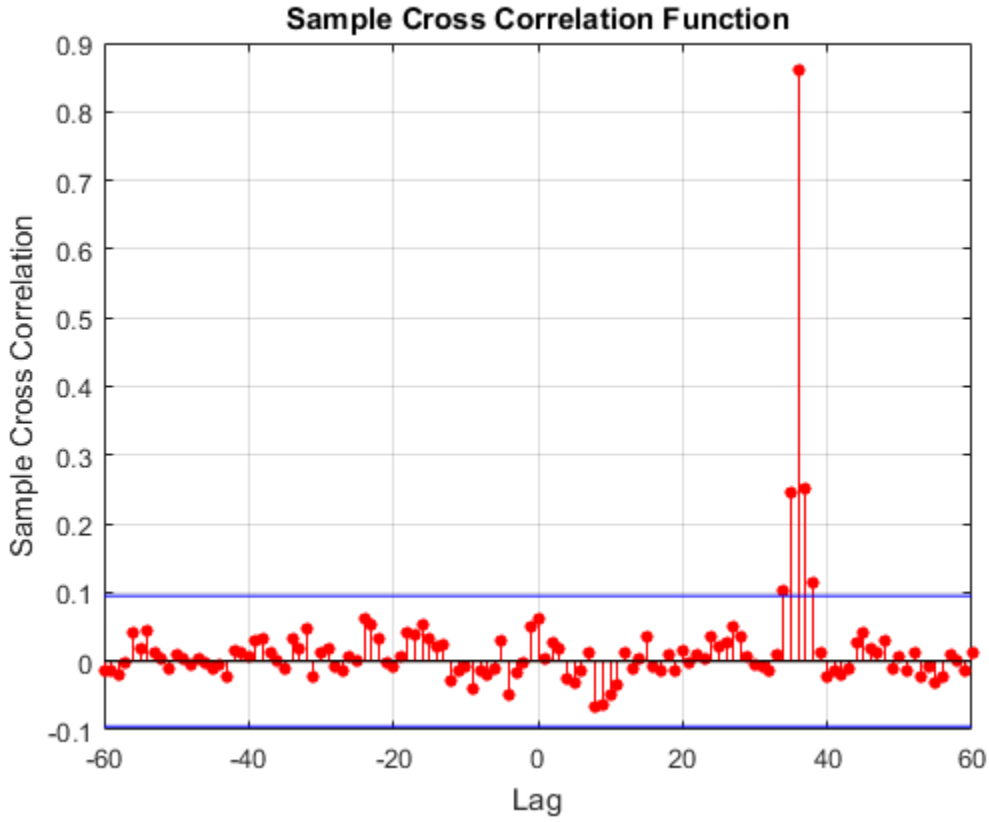
```
figure
crosscorr(y1,y2,[],3)
```



The plot does not indicate significant cross-correlation between the two series.

Plot the XCF for 60 lags on either side of lag 0.

```
figure
crosscorr(y1,y2,60,3)
```



The plot shows significant correlation at lag 36, as expected.

Input Arguments

y1 — First observed univariate time series

vector

First observed univariate time series for which the software computes or plots the XCF, specified as a vector. The last element of `y1` contains the most recent observation.

Data Types: `double`

`y2` — Second observed univariate time series

vector

Second observed univariate time series for which the software computes or plots the XCF, specified as a vector. The last element of `y2` contains the most recent observation.

Data Types: `double`

`numLags` — Number of lags

`min(20,min(length(y1),length(y2))-1)` (default) | positive integer

Number of lags of the XCF that the software returns or plots, specified as a positive integer. `crosscorr` returns the XCF at lags 0, ±1, ±2,... ±`numLags`.

For example, `crosscorr(y1,y2,10)` plots the XCF for lags 0, ±1, ±2,...,±10.

`numSTD` — Number of standard deviations

2 (default) | positive scalar

Number of standard deviations for the sample XCF estimation error assuming `y1` and `y2` are uncorrelated. For example, `crosscorr(y1,y2,[],1.5)` plots the XCF with estimation error bounds 1.5 standard deviations away from 0.

The default (`numSTD = 2`) corresponds to approximate 95% confidence bounds.

Output Arguments

`xcf` — Sample XCF

vector

Sample XCF between the univariate time series `y1` and `y2`, returned as a vector of length `2*numLags + 1`.

The elements of `xcf` correspond to lags 0, ±1, ±2,... ±`numLags`, with the center element containing the XCF for lag 0.

The software returns `xcf` in the same orientation as `y1`.

`lags` — Sample XCF lags

vector

Sample XCF lags, returned as a vector. Specifically, `lags = -numLags:numLags`.

`bounds` — Approximate confidence bounds

vector

Approximate confidence bounds of the XCF assuming `y1` and `y2` are uncorrelated, returned as a two-element vector.

More About

Sample Cross Correlation

The sample cross covariance function is an estimate of the covariance between two time series, y_{1t} and y_{2t}, at lags $k = 0, \pm1, \pm2,....$

For data pairs (y_{11},y_{21}), (y_{12},y_{22}),...,(y_{1T},y_{2T}), an estimate of the lag k cross-covariance is

$$c_{y_1y_2}(k) = \begin{cases} \frac{1}{T}\sum_{t=1}^{T-k}(y_{1t}-\bar{y}_1)(y_{2,t+k}-\bar{y}_2); k = 0,1,2,\ldots \\ \frac{1}{T}\sum_{t=1}^{T+k}(y_{2t}-\bar{y}_2)(y_{1,t-k}-\bar{y}_1); k = 0,-1,-2,\ldots \end{cases},$$

where $\bar{y}_1$ and $\bar{y}_2$ are the sample means of the series.

The sample standard deviations of the series are:

- $s_{y_1} = \sqrt{c_{y_1y_1}(0)}$, where $c_{y_1y_1}(0) = Var(y_1)$.
- $s_{y_2} = \sqrt{c_{y_2y_2}(0)}$, where $c_{y_2y_2}(0) = Var(y_2)$.

An estimate of the cross-correlation is

$$r_{y_1y_2}(k) = \frac{c_{y_1y_2}(k)}{s_{y_1}s_{y_2}}; k = 0,\pm1,\pm2,\ldots.$$

References

[1] Box, G. E. P., G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis: Forecasting and Control*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

See Also

`crosscorr` | `filter` | `parcorr`

Introduced before R2006a

disp

Class: dssm

Display summary information for diffuse state-space model

Syntax

```
disp(Mdl)
disp(Mdl,params)
disp( ___ ,Name,Value)
```

Description

`disp(Mdl)` displays summary information for the diffuse state-space model (dssm model object) `Mdl`. The display includes the state and observation equations as a system of scalar equations to facilitate model verification. The display also includes the coefficient dimensionalities, notation, and initial state distribution types.

The software displays unknown parameter values using `c1` for the first unknown parameter, `c2` for the second unknown parameter, and so on.

For time-varying models with more than 20 different sets of equations, the software displays the first and last 10 groups in terms of time (the last group is the latest).

`disp(Mdl,params)` displays the `dssm` model `Mdl` and applies initial values to the model parameters (`params`).

`disp( ___ ,Name,Value)` displays `Mdl` using additional options specified by one or more `Name,Value` pair arguments. For example, you can specify the number of digits to display after the decimal point for model coefficients, or the number of terms per row for state and observation equations. You can use any of the input arguments in the previous syntaxes.

Tips

- The software always displays explicitly specified state-space models (that is, models you create without using a parameter-to-matrix mapping function). Try explicitly specifying state-space models first so that you can verify them using `disp`.
- A parameter-to-matrix function that you specify to create `Mdl` is a black box to the software. Therefore, the software might not display complex, implicitly defined state-space models.

Input Arguments

`Mdl` — Diffuse state-space model

`dssm` model object

Diffuse state-space model, specified as a `dssm` model object returned by dssm or estimate.

`params` — Initial values for unknown parameters

`[ ]` (default) | numeric vector

Initial values for unknown parameters, specified as a numeric vector.

The elements of `params` correspond to the unknown parameters in the state-space model matrices `A`, `B`, `C`, and `D`, and, optionally, the initial state mean `Mean0` and covariance matrix `Cov0`.

- If you created `Mdl` explicitly (that is, by specifying the matrices without a parameter-to-matrix mapping function), then the software maps the elements of `params` to `NaN`s in the state-space model matrices and initial state values. The software searches for `NaN`s column-wise, following the order `A`, `B`, `C`, `D`, `Mean0`, `Cov0`.
- If you created `Mdl` implicitly (that is, by specifying the matrices with a parameter-to-matrix mapping function), then you must set initial parameter values for the state-space model matrices, initial state values, and state types within the parameter-to-matrices mapping function.

To set the type of initial state distribution, see dssm.

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'MaxStateEq'` — Maximum number of equations to display

`100` (default) | positive integer

Maximum number of equations to display, specified as the comma-separated pair consisting of `'MaxStateEq'` and a positive integer. If the maximum number of states among all periods is no larger than `MaxStateEq`, then the software displays the model equation by equation.

Example: `'MaxStateEq',10`

Data Types: `double`

`'NumDigits'` — Number of digits to display after decimal point

`2` (default) | nonnegative integer

Number of digits to display after the decimal point for known or estimated model coefficients, specified as the comma-separated pair consisting of `'NumDigits'` and a nonnegative integer.

Example: `'NumDigits',0`

Data Types: `double`

`'Period'` — Period to display state and observation equations

positive integer

Period to display state and observation equations for time-varying state-space models, specified as the comma-separated pair consisting of `'Period'` and a positive integer.

By default, the software displays state and observation equations for all periods.

If `Period` exceeds the maximum number of observations that the model supports, then the software displays state and observation equations for all periods. If the model has more than 20 different sets of equations, then the software displays the first and last 10 groups in terms of time (the last group is the latest).

Example: `'Period',120`

Data Types: `double`

`'PredictorsPerRow'` — Number of equation terms to display per row
5 (default) | positive integer

Number of equation terms to display per row, specified as the comma-separated pair consisting of `'PredictorsPerRow'` and a positive integer.

Example: `'PredictorsPerRow',3`

Data Types: `double`

Algorithms

- If you implicitly create `Mdl`, and if the software cannot infer locations for unknown parameters from the parameter-to-matrix function, then the software evaluates these parameters using their initial values and displays them as numeric values. This evaluation can occur when the parameter-to-matrix function has a random, unknown coefficient, which is a convenient form for a Monte Carlo study.
- The software displays the initial state distributions as numeric values. This type of display occurs because, in many cases, the initial distribution depends on the values of the state equation matrices `A` and `B`. These values are often a complicated function of unknown parameters. In such situations, the software does not display the initial distribution symbolically. Additionally, if `Mean0` and `Cov0` contain unknown parameters, then the software evaluates and displays numeric values for the unknown parameters.

Examples

Verify Explicitly Created Diffuse State-Space Model

An important step in state-space model analysis is to ensure that the software interpretes the state and observation equation matrices as you intend. Use `disp` to help you verify the diffuse state-space model.

Define a diffuse state-space model, where the state equation is an AR(2) model, and the observation equation is the difference between the current and previous state plus the observation error. Symbolically, the state-space model is

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \\ x_{3,t} \end{bmatrix} = \begin{bmatrix} 0.6 & 0.2 & 0.5 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \\ x_{3,t-1} \end{bmatrix} + \begin{bmatrix} 0.3 \\ 0 \\ 0 \end{bmatrix} u_{1,t}$$

$$y_t = \begin{bmatrix} 1 & -1 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t} \\ x_{2,t} \\ x_{3,t} \end{bmatrix} + 0.1\varepsilon_t.$$

Assume the initial state distribution is diffuse.

There are three states: $x_{1,t}$ is the AR(2) process, $x_{2,t}$ represents $x_{1,t-1}$, and $x_{3,t}$ is the AR(2) model constant.

Define the state-transition matrix.

```
A = [0.6 0.2 0.5; 1 0 0; 0 0 1];
```

Define the state-disturbance-loading matrix.

```
B = [0.3; 0; 0];
```

Define the measurement-sensitivity matrix.

```
C = [1 -1 0];
```

Define the observation-innovation matrix.

```
D = 0.1;
```

Specify the state-space model using `dssm`. Identify the type of initial state distributions (`StateType`) by noting the following:

- $x_{1,t}$ is an AR(2) process with diffuse initial distribution.
- $x_{2,t}$ is the same AR(2) process as $x_{1,t}$.
- $x_{3,t}$ is the constant 1 for all periods.

```
StateType = [2 2 1];
Mdl = dssm(A,B,C,D,'StateType',StateType);
```

`Mdl` is a `dssm` model.

Verify the diffuse state-space model using `disp`.

```
disp(Mdl)
```

```
State-space model type: <a href="matlab: doc dssm">dssm</a>

State vector length: 3
Observation vector length: 1
State disturbance vector length: 1
Observation innovation vector length: 1
Sample size supported by model: Unlimited

State variables: x1, x2,...
State disturbances: u1, u2,...
Observation series: y1, y2,...
Observation innovations: e1, e2,...

State equations:
x1(t) = (0.60)x1(t-1) + (0.20)x2(t-1) + (0.50)x3(t-1) + (0.30)u1(t)
x2(t) = x1(t-1)
x3(t) = x3(t-1)

Observation equation:
y1(t) = x1(t) - x2(t) + (0.10)e1(t)

Initial state distribution:

Initial state means
 x1  x2  x3
  0   0   1

Initial state covariance matrix
     x1   x2   x3
 x1  Inf  0     0
 x2  0    Inf   0
 x3  0    0     0

State types
    x1       x2       x3
 Diffuse  Diffuse  Constant
```

`Cov0` has infinite variance for the AR(2) states.

Display Diffuse State-Space Model with Initial Values

Create a diffuse state-space model containing two independent, autoregressive states, and where the observations are the deterministic sum of the two states. Symbolically, the system of equations is

$$\begin{bmatrix} x_{t,1} \\ x_{t,2} \end{bmatrix} = \begin{bmatrix} \phi_1 & 0 \\ 0 & \phi_2 \end{bmatrix} \begin{bmatrix} x_{t-1,1} \\ x_{t-1,2} \end{bmatrix} + \begin{bmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{bmatrix} \begin{bmatrix} u_{t,1} \\ u_{t,2} \end{bmatrix}$$

$$y_t = \begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} x_{t,1} \\ x_{t,2} \end{bmatrix}.$$

Specify the state-transition matrix.

```
A = [NaN 0; 0 NaN];
```

Specify the state-disturbance-loading matrix.

```
B = [NaN 0; 0 NaN];
```

Specify the measurement-sensitivity matrix.

```
C = [1 1];
```

Create the diffuse state-space model by using `dssm`. Specify that the first state is stationary and the second is diffuse.

```
StateType = [0; 2];
Mdl = dssm(A,B,C,'StateType',StateType);
```

`Mdl` is a `dssm` model object.

Display the state-space model. Specify initial values for the unknown parameters and the initial state means and covariance matrix as follows:

- $\phi_{1,0} = \phi_{2,0} = 0.1$.
- $\sigma_{1,0} = \sigma_{2,0} = 0.2$.

```
params = [0.1; 0.1; 0.2; 0.2];
disp(Mdl,params)
```

```
State-space model type: <a href="matlab: doc dssm">dssm</a>

State vector length: 2
Observation vector length: 1
State disturbance vector length: 2
Observation innovation vector length: 0
Sample size supported by model: Unlimited
Unknown parameters for estimation: 4

State variables: x1, x2,...
State disturbances: u1, u2,...
Observation series: y1, y2,...
Observation innovations: e1, e2,...
Unknown parameters: c1, c2,...

State equations:
x1(t) = (c1)x1(t-1) + (c3)u1(t)
x2(t) = (c2)x2(t-1) + (c4)u2(t)

Observation equation:
y1(t) = x1(t) + x2(t)

Initial state distribution:

Initial state means
 x1  x2
  0   0

Initial state covariance matrix
     x1    x2
 x1  0.04  0
 x2   0    Inf

State types
     x1         x2
 Stationary  Diffuse
```

The software computes the initial state mean and variance of the stationary state using its stationary distribution.

Explicitly Create and Display Time-Varying Diffuse State-Space Model

From periods 1 through 50, the state model is a diffuse AR(2) and a stationary MA(1) model, and the observation model is the sum of the two states. From periods 51 through 100, the state model includes the first AR(2) model only. Symbolically, the state-space model is, for periods 1 through 50,

$$\begin{bmatrix} x_{1t} \\ x_{2t} \\ x_{3t} \\ x_{4t} \end{bmatrix} = \begin{bmatrix} \phi_1 & \phi_2 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \theta \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \\ x_{3,t-1} \\ x_{4,t-1} \end{bmatrix} + \begin{bmatrix} \sigma_1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} u_{1t} \\ u_{2t} \\ u_{3t} \\ u_{4t} \end{bmatrix},$$

$$y_t = a_1 (x_{1t} + x_{3t}) + \sigma_2 \varepsilon_t$$

for period 51,

$$\begin{bmatrix} x_{1t} \\ x_{2t} \end{bmatrix} = \begin{bmatrix} \phi_1 & \phi_2 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \\ x_{3,t-1} \\ x_{4,t-1} \end{bmatrix} + \begin{bmatrix} \sigma_1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} u_{1t} \\ u_{2t} \\ u_{3t} \\ u_{4t} \end{bmatrix}$$

$$y_t = a_2 x_{1t} + \sigma_3 \varepsilon_t$$

and for periods 52 through 100,

$$\begin{bmatrix} x_{1t} \\ x_{2t} \end{bmatrix} = \begin{bmatrix} \phi_1 & \phi_2 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \end{bmatrix} + \begin{bmatrix} \sigma_1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} u_{1t} \\ u_{2t} \end{bmatrix}$$

$$y_t = a_2 x_{1t} + \sigma_3 \varepsilon_t.$$

Specify the state-transition coefficient matrix.

```
A1 = {[NaN NaN 0 0; 1 0 0 0; 0 0 0 NaN; 0 0 0 0]};
A2 = {[NaN NaN 0 0; 1 0 0 0]};
A3 = {[NaN NaN; 1 0]};
A = [repmat(A1,50,1);A2;repmat(A3,49,1)];
```

Specify the state-disturbance-loading coefficient matrix.

```
B1 = {[NaN 0 0 0; 0 0 0 0; 0 0 1 0; 0 0 1 0]};
B2 = {[NaN 0 0 0; 0 0 0 0]};
B3 = {[NaN 0; 0 0]};
B = [repmat(B1,50,1);B2;repmat(B3,49,1)];
```

Specify the measurement-sensitivity coefficient matrix.

```
C1 = {[NaN 0 NaN 0]};
C3 = {[NaN 0]};
C = [repmat(C1,50,1);repmat(C3,50,1)];
```

Specify the observation-distrubance coefficient matrix.

```
D1 = {NaN};
D3 = {NaN};
D = [repmat(D1,50,1);repmat(D3,50,1)];
```

Create the diffuse state-space model. Specify that the initial state distributions are diffuse for the states composing the AR model and stationary for those composing the MA model.

```
StateType = [2; 2; 0; 0];

Mdl = dssm(A,B,C,D,'StateType',StateType);
```

`Mdl` is an `dssm` model object.

The model is large and contains a different set of parameters for each period. The software displays state and observation equations for the first 10 and last 10 periods. You can choose which periods to display the equations for using the `'Period'` name-value pair argument.

Display the diffuse state-space model, and use `'Period'` display the state and observation equations for the 50th, 51st, and 52nd periods.

```
disp(Mdl,'Period',50)
disp(Mdl,'Period',51)
disp(Mdl,'Period',52)
```

```
State-space model type: <a href="matlab: doc dssm">dssm</a>

State vector length: Time-varying
Observation vector length: 1
State disturbance vector length: Time-varying
Observation innovation vector length: 1
Sample size supported by model: 100
Unknown parameters for estimation: 600

State variables: x1, x2,...
State disturbances: u1, u2,...
```

```
Observation series: y1, y2,...
Observation innovations: e1, e2,...
Unknown parameters: c1, c2,...

State equations (in period 50):
x1(t) = (c148)x1(t-1) + (c149)x2(t-1) + (c300)u1(t)
x2(t) = x1(t-1)
x3(t) = (c150)x4(t-1) + u3(t)
x4(t) = u3(t)
Time-varying transition matrix A contains unknown parameters:
c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15 c16 c17 c18 c19 c20
c21 c22 c23 c24 c25 c26 c27 c28 c29 c30 c31 c32 c33 c34 c35 c36 c37 c38 c39 c40
c41 c42 c43 c44 c45 c46 c47 c48 c49 c50 c51 c52 c53 c54 c55 c56 c57 c58 c59 c60
c61 c62 c63 c64 c65 c66 c67 c68 c69 c70 c71 c72 c73 c74 c75 c76 c77 c78 c79 c80
c81 c82 c83 c84 c85 c86 c87 c88 c89 c90 c91 c92 c93 c94 c95 c96 c97 c98 c99 c100
c101 c102 c103 c104 c105 c106 c107 c108 c109 c110 c111 c112 c113 c114 c115 c116 c117 c1
c121 c122 c123 c124 c125 c126 c127 c128 c129 c130 c131 c132 c133 c134 c135 c136 c137 c1
c141 c142 c143 c144 c145 c146 c147 c148 c149 c150 c151 c152 c153 c154 c155 c156 c157 c1
c161 c162 c163 c164 c165 c166 c167 c168 c169 c170 c171 c172 c173 c174 c175 c176 c177 c1
c181 c182 c183 c184 c185 c186 c187 c188 c189 c190 c191 c192 c193 c194 c195 c196 c197 c1
c201 c202 c203 c204 c205 c206 c207 c208 c209 c210 c211 c212 c213 c214 c215 c216 c217 c2
c221 c222 c223 c224 c225 c226 c227 c228 c229 c230 c231 c232 c233 c234 c235 c236 c237 c2
c241 c242 c243 c244 c245 c246 c247 c248 c249 c250
Time-varying state disturbance loading matrix B contains unknown parameters:
c251 c252 c253 c254 c255 c256 c257 c258 c259 c260 c261 c262 c263 c264 c265 c266 c267 c2
c271 c272 c273 c274 c275 c276 c277 c278 c279 c280 c281 c282 c283 c284 c285 c286 c287 c2
c291 c292 c293 c294 c295 c296 c297 c298 c299 c300 c301 c302 c303 c304 c305 c306 c307 c3
c311 c312 c313 c314 c315 c316 c317 c318 c319 c320 c321 c322 c323 c324 c325 c326 c327 c3
c331 c332 c333 c334 c335 c336 c337 c338 c339 c340 c341 c342 c343 c344 c345 c346 c347 c3

Observation equation (in period 50):
y1(t) = (c449)x1(t) + (c450)x3(t) + (c550)e1(t)
Time-varying measurement sensitivity matrix C contains unknown parameters:
c351 c352 c353 c354 c355 c356 c357 c358 c359 c360 c361 c362 c363 c364 c365 c366 c367 c3
c371 c372 c373 c374 c375 c376 c377 c378 c379 c380 c381 c382 c383 c384 c385 c386 c387 c3
c391 c392 c393 c394 c395 c396 c397 c398 c399 c400 c401 c402 c403 c404 c405 c406 c407 c4
c411 c412 c413 c414 c415 c416 c417 c418 c419 c420 c421 c422 c423 c424 c425 c426 c427 c4
c431 c432 c433 c434 c435 c436 c437 c438 c439 c440 c441 c442 c443 c444 c445 c446 c447 c4
c451 c452 c453 c454 c455 c456 c457 c458 c459 c460 c461 c462 c463 c464 c465 c466 c467 c4
c471 c472 c473 c474 c475 c476 c477 c478 c479 c480 c481 c482 c483 c484 c485 c486 c487 c4
c491 c492 c493 c494 c495 c496 c497 c498 c499 c500
Time-varying observation innovation loading matrix D contains unknown parameters:
c501 c502 c503 c504 c505 c506 c507 c508 c509 c510 c511 c512 c513 c514 c515 c516 c517 c5
c521 c522 c523 c524 c525 c526 c527 c528 c529 c530 c531 c532 c533 c534 c535 c536 c537 c5
```

```
c541 c542 c543 c544 c545 c546 c547 c548 c549 c550 c551 c552 c553 c554 c555 c556 c557 c5
c561 c562 c563 c564 c565 c566 c567 c568 c569 c570 c571 c572 c573 c574 c575 c576 c577 c5
c581 c582 c583 c584 c585 c586 c587 c588 c589 c590 c591 c592 c593 c594 c595 c596 c597 c5

Initial state distribution:

Initial state means are not specified.
Initial state covariance matrix is not specified.
State types
    x1       x2        x3          x4
 Diffuse  Diffuse  Stationary  Stationary


State-space model type: <a href="matlab: doc dssm">dssm</a>

State vector length: Time-varying
Observation vector length: 1
State disturbance vector length: Time-varying
Observation innovation vector length: 1
Sample size supported by model: 100
Unknown parameters for estimation: 600

State variables: x1, x2,...
State disturbances: u1, u2,...
Observation series: y1, y2,...
Observation innovations: e1, e2,...
Unknown parameters: c1, c2,...

State equations (in period 51):
x1(t) = (c151)x1(t-1) + (c152)x2(t-1) + (c301)u1(t)
x2(t) = x1(t-1)
Time-varying transition matrix A contains unknown parameters:
c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15 c16 c17 c18 c19 c20
c21 c22 c23 c24 c25 c26 c27 c28 c29 c30 c31 c32 c33 c34 c35 c36 c37 c38 c39 c40
c41 c42 c43 c44 c45 c46 c47 c48 c49 c50 c51 c52 c53 c54 c55 c56 c57 c58 c59 c60
c61 c62 c63 c64 c65 c66 c67 c68 c69 c70 c71 c72 c73 c74 c75 c76 c77 c78 c79 c80
c81 c82 c83 c84 c85 c86 c87 c88 c89 c90 c91 c92 c93 c94 c95 c96 c97 c98 c99 c100
c101 c102 c103 c104 c105 c106 c107 c108 c109 c110 c111 c112 c113 c114 c115 c116 c117 c1
c121 c122 c123 c124 c125 c126 c127 c128 c129 c130 c131 c132 c133 c134 c135 c136 c137 c1
c141 c142 c143 c144 c145 c146 c147 c148 c149 c150 c151 c152 c153 c154 c155 c156 c157 c1
c161 c162 c163 c164 c165 c166 c167 c168 c169 c170 c171 c172 c173 c174 c175 c176 c177 c1
c181 c182 c183 c184 c185 c186 c187 c188 c189 c190 c191 c192 c193 c194 c195 c196 c197 c1
c201 c202 c203 c204 c205 c206 c207 c208 c209 c210 c211 c212 c213 c214 c215 c216 c217 c2
c221 c222 c223 c224 c225 c226 c227 c228 c229 c230 c231 c232 c233 c234 c235 c236 c237 c2
```

```
c241 c242 c243 c244 c245 c246 c247 c248 c249 c250
Time-varying state disturbance loading matrix B contains unknown parameters:
c251 c252 c253 c254 c255 c256 c257 c258 c259 c260 c261 c262 c263 c264 c265 c266 c267 c2
c271 c272 c273 c274 c275 c276 c277 c278 c279 c280 c281 c282 c283 c284 c285 c286 c287 c2
c291 c292 c293 c294 c295 c296 c297 c298 c299 c300 c301 c302 c303 c304 c305 c306 c307 c3
c311 c312 c313 c314 c315 c316 c317 c318 c319 c320 c321 c322 c323 c324 c325 c326 c327 c3
c331 c332 c333 c334 c335 c336 c337 c338 c339 c340 c341 c342 c343 c344 c345 c346 c347 c3

Observation equation (in period 51):
y1(t) = (c451)x1(t) + (c551)e1(t)
Time-varying measurement sensitivity matrix C contains unknown parameters:
c351 c352 c353 c354 c355 c356 c357 c358 c359 c360 c361 c362 c363 c364 c365 c366 c367 c3
c371 c372 c373 c374 c375 c376 c377 c378 c379 c380 c381 c382 c383 c384 c385 c386 c387 c3
c391 c392 c393 c394 c395 c396 c397 c398 c399 c400 c401 c402 c403 c404 c405 c406 c407 c4
c411 c412 c413 c414 c415 c416 c417 c418 c419 c420 c421 c422 c423 c424 c425 c426 c427 c4
c431 c432 c433 c434 c435 c436 c437 c438 c439 c440 c441 c442 c443 c444 c445 c446 c447 c4
c451 c452 c453 c454 c455 c456 c457 c458 c459 c460 c461 c462 c463 c464 c465 c466 c467 c4
c471 c472 c473 c474 c475 c476 c477 c478 c479 c480 c481 c482 c483 c484 c485 c486 c487 c4
c491 c492 c493 c494 c495 c496 c497 c498 c499 c500
Time-varying observation innovation loading matrix D contains unknown parameters:
c501 c502 c503 c504 c505 c506 c507 c508 c509 c510 c511 c512 c513 c514 c515 c516 c517 c5
c521 c522 c523 c524 c525 c526 c527 c528 c529 c530 c531 c532 c533 c534 c535 c536 c537 c5
c541 c542 c543 c544 c545 c546 c547 c548 c549 c550 c551 c552 c553 c554 c555 c556 c557 c5
c561 c562 c563 c564 c565 c566 c567 c568 c569 c570 c571 c572 c573 c574 c575 c576 c577 c5
c581 c582 c583 c584 c585 c586 c587 c588 c589 c590 c591 c592 c593 c594 c595 c596 c597 c5

Initial state distribution:

Initial state means are not specified.
Initial state covariance matrix is not specified.
State types
    x1       x2        x3          x4
 Diffuse  Diffuse  Stationary  Stationary


State-space model type: <a href="matlab: doc dssm">dssm</a>

State vector length: Time-varying
Observation vector length: 1
State disturbance vector length: Time-varying
Observation innovation vector length: 1
Sample size supported by model: 100
Unknown parameters for estimation: 600
```

```
State variables: x1, x2,...
State disturbances: u1, u2,...
Observation series: y1, y2,...
Observation innovations: e1, e2,...
Unknown parameters: c1, c2,...

State equations (in period 52):
x1(t) = (c153)x1(t-1) + (c154)x2(t-1) + (c302)u1(t)
x2(t) = x1(t-1)
Time-varying transition matrix A contains unknown parameters:
c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15 c16 c17 c18 c19 c20
c21 c22 c23 c24 c25 c26 c27 c28 c29 c30 c31 c32 c33 c34 c35 c36 c37 c38 c39 c40
c41 c42 c43 c44 c45 c46 c47 c48 c49 c50 c51 c52 c53 c54 c55 c56 c57 c58 c59 c60
c61 c62 c63 c64 c65 c66 c67 c68 c69 c70 c71 c72 c73 c74 c75 c76 c77 c78 c79 c80
c81 c82 c83 c84 c85 c86 c87 c88 c89 c90 c91 c92 c93 c94 c95 c96 c97 c98 c99 c100
c101 c102 c103 c104 c105 c106 c107 c108 c109 c110 c111 c112 c113 c114 c115 c116 c117 c
c121 c122 c123 c124 c125 c126 c127 c128 c129 c130 c131 c132 c133 c134 c135 c136 c137 c
c141 c142 c143 c144 c145 c146 c147 c148 c149 c150 c151 c152 c153 c154 c155 c156 c157 c
c161 c162 c163 c164 c165 c166 c167 c168 c169 c170 c171 c172 c173 c174 c175 c176 c177 c
c181 c182 c183 c184 c185 c186 c187 c188 c189 c190 c191 c192 c193 c194 c195 c196 c197 c
c201 c202 c203 c204 c205 c206 c207 c208 c209 c210 c211 c212 c213 c214 c215 c216 c217 c2
c221 c222 c223 c224 c225 c226 c227 c228 c229 c230 c231 c232 c233 c234 c235 c236 c237 c2
c241 c242 c243 c244 c245 c246 c247 c248 c249 c250
Time-varying state disturbance loading matrix B contains unknown parameters:
c251 c252 c253 c254 c255 c256 c257 c258 c259 c260 c261 c262 c263 c264 c265 c266 c267 c2
c271 c272 c273 c274 c275 c276 c277 c278 c279 c280 c281 c282 c283 c284 c285 c286 c287 c2
c291 c292 c293 c294 c295 c296 c297 c298 c299 c300 c301 c302 c303 c304 c305 c306 c307 c3
c311 c312 c313 c314 c315 c316 c317 c318 c319 c320 c321 c322 c323 c324 c325 c326 c327 c3
c331 c332 c333 c334 c335 c336 c337 c338 c339 c340 c341 c342 c343 c344 c345 c346 c347 c3

Observation equation (in period 52):
y1(t) = (c452)x1(t) + (c552)e1(t)
Time-varying measurement sensitivity matrix C contains unknown parameters:
c351 c352 c353 c354 c355 c356 c357 c358 c359 c360 c361 c362 c363 c364 c365 c366 c367 c3
c371 c372 c373 c374 c375 c376 c377 c378 c379 c380 c381 c382 c383 c384 c385 c386 c387 c3
c391 c392 c393 c394 c395 c396 c397 c398 c399 c400 c401 c402 c403 c404 c405 c406 c407 c4
c411 c412 c413 c414 c415 c416 c417 c418 c419 c420 c421 c422 c423 c424 c425 c426 c427 c4
c431 c432 c433 c434 c435 c436 c437 c438 c439 c440 c441 c442 c443 c444 c445 c446 c447 c4
c451 c452 c453 c454 c455 c456 c457 c458 c459 c460 c461 c462 c463 c464 c465 c466 c467 c4
c471 c472 c473 c474 c475 c476 c477 c478 c479 c480 c481 c482 c483 c484 c485 c486 c487 c4
c491 c492 c493 c494 c495 c496 c497 c498 c499 c500
Time-varying observation innovation loading matrix D contains unknown parameters:
c501 c502 c503 c504 c505 c506 c507 c508 c509 c510 c511 c512 c513 c514 c515 c516 c517 c5
c521 c522 c523 c524 c525 c526 c527 c528 c529 c530 c531 c532 c533 c534 c535 c536 c537 c5
```

```
c541 c542 c543 c544 c545 c546 c547 c548 c549 c550 c551 c552 c553 c554 c555 c556 c557 c5
c561 c562 c563 c564 c565 c566 c567 c568 c569 c570 c571 c572 c573 c574 c575 c576 c577 c5
c581 c582 c583 c584 c585 c586 c587 c588 c589 c590 c591 c592 c593 c594 c595 c596 c597 c5

Initial state distribution:

Initial state means are not specified.
Initial state covariance matrix is not specified.
State types
    x1       x2        x3          x4
 Diffuse  Diffuse  Stationary  Stationary
```

The software attributes a different set of coefficients for each period. You might experience numerical issues when you estimate such models. To reuse parameters among groups of periods, consider creating a parameter-to-matrix mapping function.

References

[1] Durbin J., and S. J. Koopman. *Time Series Analysis by State Space Methods*. 2nd ed. Oxford: Oxford University Press, 2012.

See Also

dssm | estimate | filter | forecast | smooth | ssm

More About

- "What Are State-Space Models?" on page 8-3

Introduced in R2015b

disp

Class: ssm

Display summary information for state-space model

Syntax

```
disp(Mdl)
disp(Mdl,params)
disp(___,Name,Value)
```

Description

`disp(Mdl)` displays summary information for the state-space model (ssm model object) `Mdl`. The display includes the state and observation equations as a system of scalar equations to facilitate model verification. The display also includes the coefficient dimensionalities, notation, and initial state distribution types.

The software displays unknown parameter values using `c1` for the first unknown parameter, `c2` for the second unknown parameter, and so on.

For time-varying models with more than 20 different sets of equations, the software displays the first and last 10 groups in terms of time (the last group is the latest).

`disp(Mdl,params)` displays the `ssm` model `Mdl` and applies initial values to the model parameters (`params`).

`disp(___,Name,Value)` displays the `ssm` model with additional options specified by one or more `Name,Value` pair arguments. For example, you can specify the number of digits to display after the decimal point for model coefficients, or the number of terms per row for state and observation equations. You can use any of the input arguments in the previous syntaxes.

Tips

- The software always displays explicitly specified state-space models (that is, models you create without using a parameter-to-matrix mapping function). Try explicitly specifying state-space models first so that you can verify them using `disp`.
- A parameter-to-matrix function that you specify to create `Mdl` is a black box to the software. Therefore, the software might not display complex, implicitly defined state-space models.

Input Arguments

`Mdl` — Standard state-space model

`ssm` model object

Standard state-space model, specified as an `ssm` model object returned by ssm or estimate.

`params` — Initial values for unknown parameters

`[ ]` (default) | numeric vector

Initial values for unknown parameters, specified as a numeric vector.

The elements of `params` correspond to the unknown parameters in the state-space model matrices `A`, `B`, `C`, and `D`, and, optionally, the initial state mean `Mean0` and covariance matrix `Cov0`.

- If you created `Mdl` explicitly (that is, by specifying the matrices without a parameter-to-matrix mapping function), then the software maps the elements of `params` to `NaN`s in the state-space model matrices and initial state values. The software searches for `NaN`s column-wise, following the order `A`, `B`, `C`, `D`, `Mean0`, `Cov0`.
- If you created `Mdl` implicitly (that is, by specifying the matrices with a parameter-to-matrix mapping function), then you must set initial parameter values for the state-space model matrices, initial state values, and state types within the parameter-to-matrices mapping function.

To set the type of initial state distribution, see ssm.

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'MaxStateEq'` — Maximum number of equations to display

`100` (default) | positive integer

Maximum number of equations to display, specified as the comma-separated pair consisting of `'MaxStateEq'` and a positive integer. If the maximum number of states among all periods is no larger than `MaxStateEq`, then the software displays the model equation by equation.

Example: `'MaxStateEq',10`

Data Types: `double`

`'NumDigits'` — Number of digits to display after decimal point

`2` (default) | nonnegative integer

Number of digits to display after the decimal point for known or estimated model coefficients, specified as the comma-separated pair consisting of `'NumDigits'` and a nonnegative integer.

Example: `'NumDigits',0`

Data Types: `double`

`'Period'` — Period to display state and observation equations

positive integer

Period to display state and observation equations for time-varying state-space models, specified as the comma-separated pair consisting of `'Period'` and a positive integer.

By default, the software displays state and observation equations for all periods.

If `Period` exceeds the maximum number of observations that the model supports, then the software displays state and observation equations for all periods. If the model has more than 20 different sets of equations, then the software displays the first and last 10 groups in terms of time (the last group is the latest).

Example: `'Period',120`

Data Types: `double`

`'PredictorsPerRow'` — Number of equation terms to display per row

`5` (default) | positive integer

Number of equation terms to display per row, specified as the comma-separated pair consisting of `'PredictorsPerRow'` and a positive integer.

Example: `'PredictorsPerRow',3`

Data Types: `double`

Examples

Verify Explicitly Created State-Space Model

An important step in state-space model analysis is to ensure that the software interpretes the state and observation equation matrices as you intend. Use `disp` to help you verify the state-space model.

Define a state-space model, where the state equation is an AR(2) model, and the observation equation is the difference between the current and previous state plus the observation error. Symbolically, the state-space model is

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \\ x_{3,t} \end{bmatrix} = \begin{bmatrix} 0.6 & 0.2 & 0.5 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \\ x_{3,t-1} \end{bmatrix} + \begin{bmatrix} 0.3 \\ 0 \\ 0 \end{bmatrix} u_{1,t}$$

$$y_t = \begin{bmatrix} 1 & -1 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t} \\ x_{2,t} \\ x_{3,t} \end{bmatrix} + 0.1\varepsilon_t.$$

There are three states: $x_{1,t}$ is the AR(2) process, $x_{2,t}$ represents $x_{1,t-1}$, and $x_{3,t}$ is the AR(2) model constant.

Define the state-transition matrix.

```
A = [0.6 0.2 0.5; 1 0 0; 0 0 1];
```

Define the state-disturbance-loading matrix.

```
B = [0.3; 0; 0];
```

Define the measurement-sensitivity matrix.

```
C = [1 -1 0];
```

Define the observation-innovation matrix.

```
D = 0.1;
```

Specify the state-space model using `ssm`. Set the initial-state mean (`Mean0`) and covariance matrix (`Cov0`). Identify the type of initial state distributions (`StateType`) by noting the following:

- $x_{1,t}$ is a stationary, AR(2) process.
- $x_{2,t}$ is also a stationary, AR(2) process.
- $x_{3,t}$ is the constant 1 for all periods.

```
Mean0 = [0; 0; 1];                                          % The mean of the AR(2)
varAR2 = 0.3*(1 - 0.2)/((1 + 0.2)*((1 - 0.2)^2 - 0.6^2)); % The variance of the AR(2)
Cov1AR2 = 0.6*0.3/((1 + 0.2)*((1 - 0.2)^2) - 0.6^2);      % The covariance of the AR(2
Cov0 = zeros(3);
Cov0(1:2,1:2) = varAR2*eye(2) + Cov1AR2*flip(eye(2));
StateType = [0; 0; 1];
Mdl = ssm(A,B,C,D,'Mean0',Mean0,'Cov0',Cov0,'StateType',StateType);
```

`Mdl` is an `ssm` model.

Verify the state-space model using `disp`.

```
disp(Mdl)
```

```
State-space model type: <a href="matlab: doc ssm">ssm</a>

State vector length: 3
Observation vector length: 1
State disturbance vector length: 1
Observation innovation vector length: 1
Sample size supported by model: Unlimited

State variables: x1, x2,...
State disturbances: u1, u2,...
```

```
Observation series: y1, y2,...
Observation innovations: e1, e2,...

State equations:
x1(t) = (0.60)x1(t-1) + (0.20)x2(t-1) + (0.50)x3(t-1) + (0.30)u1(t)
x2(t) = x1(t-1)
x3(t) = x3(t-1)

Observation equation:
y1(t) = x1(t) - x2(t) + (0.10)e1(t)

Initial state distribution:

Initial state means
 x1  x2  x3
  0   0   1

Initial state covariance matrix
     x1    x2    x3
 x1  0.71  0.44   0
 x2  0.44  0.71   0
 x3   0     0     0

State types
     x1          x2          x3
 Stationary  Stationary  Constant
```

Display State-Space Model and Initial Values

Define a state-space model containing two independent, autoregressive states, and where the observations are the deterministic sum of the two states. Symbolically, the system of equations is

$$\begin{bmatrix} x_{t,1} \\ x_{t,2} \end{bmatrix} = \begin{bmatrix} \phi_1 & 0 \\ 0 & \phi_2 \end{bmatrix} \begin{bmatrix} x_{t-1,1} \\ x_{t-1,2} \end{bmatrix} + \begin{bmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{bmatrix} \begin{bmatrix} u_{t,1} \\ u_{t,2} \end{bmatrix}$$

$$y_t = \begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} x_{t,1} \\ x_{t,2} \end{bmatrix}.$$

Specify the state-transition matrix.

```
A = [NaN 0; 0 NaN];
```

Specify the state-disturbance-loading matrix.

```
B = [NaN 0; 0 NaN];
```

Specify the measurement-sensitivity matrix.

```
C = [1 1];
```

Specify an empty matrix for the observation distrubance matrix.

```
D = [];
```

Use `ssm` to define the state-space model. Specify the initial state means and covariance matrix to as unknown parameters. Specify that the states are stationary.

```
Mean0 = nan(2,1);
Cov0 = nan(2,2);
StateType = zeros(2,1);
Mdl = ssm(A,B,C,D,'Mean0',Mean0,'Cov0',Cov0,'StateType',StateType);
```

`Mdl` is an `ssm` model containing unknown parameters.

Use `disp` to display the state-space model. Specify initial values for the unknown parameters and the initial state means and covariance matrix as follows:

- $\phi_{1,0} = \phi_{2,0} = 0.1$.
- $\sigma_{1,0} = \sigma_{2,0} = 0.2$.
- $x_{1,0} = 1$ and $x_{2,0} = 0.5$.
- $\Sigma_{x_{1,0},x_{2,0}} = I_2$.

```
params = [0.1; 0.1; 0.2; 0.2; 1; 0.5; 1; 0; 0; 1];
disp(Mdl,params)
```

```
State-space model type: <a href="matlab: doc ssm">ssm</a>

State vector length: 2
Observation vector length: 1
State disturbance vector length: 2
Observation innovation vector length: 0
Sample size supported by model: Unlimited
Unknown parameters for estimation: 10
```

```
State variables: x1, x2,...
State disturbances: u1, u2,...
Observation series: y1, y2,...
Observation innovations: e1, e2,...
Unknown parameters: c1, c2,...

State equations:
x1(t) = (c1)x1(t-1) + (c3)u1(t)
x2(t) = (c2)x2(t-1) + (c4)u2(t)

Observation equation:
y1(t) = x1(t) + x2(t)

Initial state distribution:

Initial state means
 x1   x2
  1  0.50

Initial state covariance matrix
     x1  x2
 x1   1   0
 x2   0   1

State types
     x1          x2
 Stationary  Stationary
```

Explicitly Create and Display Time-Varying State-Space Model

From periods 1 through 50, the state model is an AR(2) and an MA(1) model, and the observation model is the sum of the two states. From periods 51 through 100, the state model includes the first AR(2) model only. Symbolically, the state-space model is, for periods 1 through 50,

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \\ x_{3,t} \\ x_{4,t} \end{bmatrix} = \begin{bmatrix} \phi_1 & \phi_2 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \theta \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \\ x_{3,t-1} \\ x_{4,t-1} \end{bmatrix} + \begin{bmatrix} \sigma_1 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} u_{1,t} \\ u_{3,t} \end{bmatrix},$$

$$y_t = a_1 (x_{1,t} + x_{3,t}) + \sigma_2 \varepsilon_t$$

for period 51,

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix} = \begin{bmatrix} \phi_1 & \phi_2 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \\ x_{3,t-1} \\ x_{4,t-1} \end{bmatrix} + \begin{bmatrix} \sigma_1 \\ 0 \end{bmatrix} u_{1,t}$$

$$y_t = a_2 x_{1t} + \sigma_3 \varepsilon_t$$

and for periods 52 through 100,

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix} = \begin{bmatrix} \phi_1 & \phi_2 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \end{bmatrix} + \begin{bmatrix} \sigma_1 & 0 \\ 0 & 0 \end{bmatrix} u_{1,t}$$

$$y_t = a_2 x_{1,t} + \sigma_3 \varepsilon_t.$$

Specify the state-transition coefficient matrix.

```
A1 = {[NaN NaN 0 0; 1 0 0 0; 0 0 0 NaN; 0 0 0 0]};
A2 = {[NaN NaN 0 0; 1 0 0 0]};
A3 = {[NaN NaN; 1 0]};
A = [repmat(A1,50,1);A2;repmat(A3,49,1)];
```

Specify the state-disturbance-loading coefficient matrix.

```
B1 = {[NaN 0;0 0; 0 1; 0 1]};
B2 = {[NaN; 0]};
B3 = {[NaN; 0]};
B = [repmat(B1,50,1);B2;repmat(B3,49,1)];
```

Specify the measurement-sensitivity coefficient matrix.

```
C1 = {[NaN 0 NaN 0]};
C3 = {[NaN 0]};
C = [repmat(C1,50,1);repmat(C3,50,1)];
```

Specify the observation-distrubance coefficient matrix.

```
D1 = {NaN};
D3 = {NaN};
D = [repmat(D1,50,1);repmat(D3,50,1)];
```

Specify the state-space model. Set the initial state means and covariance matrix to unknown parameters. Specify that the initial state distributions are stationary.

```
Mean0 = nan(4,1);
Cov0 = nan(4,4);
StateType = [0; 0; 0; 0];
```

```
Mdl = ssm(A,B,C,D,'Mean0',Mean0,'Cov0',Cov0,'StateType',StateType);
```

`Mdl` is an `ssm` model.

The model is large and contains a different set of parameters for each period. The software displays state and observation equations for the first 10 and last 10 periods. You can choose which periods to display the equations for using the `'Period'` name-value pair argument.

Display the state-space model, and use `'Period'` to display the state and observation equations for the 50th, 51st, and 52nd periods.

```
disp(Mdl,'Period',50)
disp(Mdl,'Period',51)
disp(Mdl,'Period',52)
```

```
State-space model type: <a href="matlab: doc ssm">ssm</a>

State vector length: Time-varying
Observation vector length: 1
State disturbance vector length: Time-varying
Observation innovation vector length: 1
Sample size supported by model: 100
Unknown parameters for estimation: 620

State variables: x1, x2,...
State disturbances: u1, u2,...
Observation series: y1, y2,...
Observation innovations: e1, e2,...
Unknown parameters: c1, c2,...

State equations (in period 50):
x1(t) = (c148)x1(t-1) + (c149)x2(t-1) + (c300)u1(t)
x2(t) = x1(t-1)
x3(t) = (c150)x4(t-1) + u2(t)
x4(t) = u2(t)
Time-varying transition matrix A contains unknown parameters:
c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15 c16 c17 c18 c19 c20
c21 c22 c23 c24 c25 c26 c27 c28 c29 c30 c31 c32 c33 c34 c35 c36 c37 c38 c39 c40
c41 c42 c43 c44 c45 c46 c47 c48 c49 c50 c51 c52 c53 c54 c55 c56 c57 c58 c59 c60
c61 c62 c63 c64 c65 c66 c67 c68 c69 c70 c71 c72 c73 c74 c75 c76 c77 c78 c79 c80
c81 c82 c83 c84 c85 c86 c87 c88 c89 c90 c91 c92 c93 c94 c95 c96 c97 c98 c99 c100
c101 c102 c103 c104 c105 c106 c107 c108 c109 c110 c111 c112 c113 c114 c115 c116 c117 c
c121 c122 c123 c124 c125 c126 c127 c128 c129 c130 c131 c132 c133 c134 c135 c136 c137 c
```

```
c141 c142 c143 c144 c145 c146 c147 c148 c149 c150 c151 c152 c153 c154 c155 c156 c157 c1
c161 c162 c163 c164 c165 c166 c167 c168 c169 c170 c171 c172 c173 c174 c175 c176 c177 c1
c181 c182 c183 c184 c185 c186 c187 c188 c189 c190 c191 c192 c193 c194 c195 c196 c197 c1
c201 c202 c203 c204 c205 c206 c207 c208 c209 c210 c211 c212 c213 c214 c215 c216 c217 c2
c221 c222 c223 c224 c225 c226 c227 c228 c229 c230 c231 c232 c233 c234 c235 c236 c237 c2
c241 c242 c243 c244 c245 c246 c247 c248 c249 c250
Time-varying state disturbance loading matrix B contains unknown parameters:
c251 c252 c253 c254 c255 c256 c257 c258 c259 c260 c261 c262 c263 c264 c265 c266 c267 c2
c271 c272 c273 c274 c275 c276 c277 c278 c279 c280 c281 c282 c283 c284 c285 c286 c287 c2
c291 c292 c293 c294 c295 c296 c297 c298 c299 c300 c301 c302 c303 c304 c305 c306 c307 c3
c311 c312 c313 c314 c315 c316 c317 c318 c319 c320 c321 c322 c323 c324 c325 c326 c327 c3
c331 c332 c333 c334 c335 c336 c337 c338 c339 c340 c341 c342 c343 c344 c345 c346 c347 c3

Observation equation (in period 50):
y1(t) = (c449)x1(t) + (c450)x3(t) + (c550)e1(t)
Time-varying measurement sensitivity matrix C contains unknown parameters:
c351 c352 c353 c354 c355 c356 c357 c358 c359 c360 c361 c362 c363 c364 c365 c366 c367 c3
c371 c372 c373 c374 c375 c376 c377 c378 c379 c380 c381 c382 c383 c384 c385 c386 c387 c3
c391 c392 c393 c394 c395 c396 c397 c398 c399 c400 c401 c402 c403 c404 c405 c406 c407 c4
c411 c412 c413 c414 c415 c416 c417 c418 c419 c420 c421 c422 c423 c424 c425 c426 c427 c4
c431 c432 c433 c434 c435 c436 c437 c438 c439 c440 c441 c442 c443 c444 c445 c446 c447 c4
c451 c452 c453 c454 c455 c456 c457 c458 c459 c460 c461 c462 c463 c464 c465 c466 c467 c4
c471 c472 c473 c474 c475 c476 c477 c478 c479 c480 c481 c482 c483 c484 c485 c486 c487 c4
c491 c492 c493 c494 c495 c496 c497 c498 c499 c500
Time-varying observation innovation loading matrix D contains unknown parameters:
c501 c502 c503 c504 c505 c506 c507 c508 c509 c510 c511 c512 c513 c514 c515 c516 c517 c5
c521 c522 c523 c524 c525 c526 c527 c528 c529 c530 c531 c532 c533 c534 c535 c536 c537 c5
c541 c542 c543 c544 c545 c546 c547 c548 c549 c550 c551 c552 c553 c554 c555 c556 c557 c5
c561 c562 c563 c564 c565 c566 c567 c568 c569 c570 c571 c572 c573 c574 c575 c576 c577 c5
c581 c582 c583 c584 c585 c586 c587 c588 c589 c590 c591 c592 c593 c594 c595 c596 c597 c5

Initial state distribution:

Initial state means
  x1   x2   x3   x4
 NaN  NaN  NaN  NaN

Initial state covariance matrix
     x1   x2   x3   x4
 x1  NaN  NaN  NaN  NaN
 x2  NaN  NaN  NaN  NaN
 x3  NaN  NaN  NaN  NaN
 x4  NaN  NaN  NaN  NaN
```

```
State types
     x1          x2          x3          x4
 Stationary  Stationary  Stationary  Stationary


State-space model type: <a href="matlab: doc ssm">ssm</a>

State vector length: Time-varying
Observation vector length: 1
State disturbance vector length: Time-varying
Observation innovation vector length: 1
Sample size supported by model: 100
Unknown parameters for estimation: 620

State variables: x1, x2,...
State disturbances: u1, u2,...
Observation series: y1, y2,...
Observation innovations: e1, e2,...
Unknown parameters: c1, c2,...

State equations (in period 51):
x1(t) = (c151)x1(t-1) + (c152)x2(t-1) + (c301)u1(t)
x2(t) = x1(t-1)
Time-varying transition matrix A contains unknown parameters:
c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15 c16 c17 c18 c19 c20
c21 c22 c23 c24 c25 c26 c27 c28 c29 c30 c31 c32 c33 c34 c35 c36 c37 c38 c39 c40
c41 c42 c43 c44 c45 c46 c47 c48 c49 c50 c51 c52 c53 c54 c55 c56 c57 c58 c59 c60
c61 c62 c63 c64 c65 c66 c67 c68 c69 c70 c71 c72 c73 c74 c75 c76 c77 c78 c79 c80
c81 c82 c83 c84 c85 c86 c87 c88 c89 c90 c91 c92 c93 c94 c95 c96 c97 c98 c99 c100
c101 c102 c103 c104 c105 c106 c107 c108 c109 c110 c111 c112 c113 c114 c115 c116 c117 c
c121 c122 c123 c124 c125 c126 c127 c128 c129 c130 c131 c132 c133 c134 c135 c136 c137 c
c141 c142 c143 c144 c145 c146 c147 c148 c149 c150 c151 c152 c153 c154 c155 c156 c157 c
c161 c162 c163 c164 c165 c166 c167 c168 c169 c170 c171 c172 c173 c174 c175 c176 c177 c
c181 c182 c183 c184 c185 c186 c187 c188 c189 c190 c191 c192 c193 c194 c195 c196 c197 c
c201 c202 c203 c204 c205 c206 c207 c208 c209 c210 c211 c212 c213 c214 c215 c216 c217 c2
c221 c222 c223 c224 c225 c226 c227 c228 c229 c230 c231 c232 c233 c234 c235 c236 c237 c2
c241 c242 c243 c244 c245 c246 c247 c248 c249 c250
Time-varying state disturbance loading matrix B contains unknown parameters:
c251 c252 c253 c254 c255 c256 c257 c258 c259 c260 c261 c262 c263 c264 c265 c266 c267 c2
c271 c272 c273 c274 c275 c276 c277 c278 c279 c280 c281 c282 c283 c284 c285 c286 c287 c2
c291 c292 c293 c294 c295 c296 c297 c298 c299 c300 c301 c302 c303 c304 c305 c306 c307 c3
c311 c312 c313 c314 c315 c316 c317 c318 c319 c320 c321 c322 c323 c324 c325 c326 c327 c3
c331 c332 c333 c334 c335 c336 c337 c338 c339 c340 c341 c342 c343 c344 c345 c346 c347 c3
```

```
Observation equation (in period 51):
y1(t) = (c451)x1(t) + (c551)e1(t)
Time-varying measurement sensitivity matrix C contains unknown parameters:
c351 c352 c353 c354 c355 c356 c357 c358 c359 c360 c361 c362 c363 c364 c365 c366 c367 c3
c371 c372 c373 c374 c375 c376 c377 c378 c379 c380 c381 c382 c383 c384 c385 c386 c387 c3
c391 c392 c393 c394 c395 c396 c397 c398 c399 c400 c401 c402 c403 c404 c405 c406 c407 c4
c411 c412 c413 c414 c415 c416 c417 c418 c419 c420 c421 c422 c423 c424 c425 c426 c427 c4
c431 c432 c433 c434 c435 c436 c437 c438 c439 c440 c441 c442 c443 c444 c445 c446 c447 c4
c451 c452 c453 c454 c455 c456 c457 c458 c459 c460 c461 c462 c463 c464 c465 c466 c467 c4
c471 c472 c473 c474 c475 c476 c477 c478 c479 c480 c481 c482 c483 c484 c485 c486 c487 c4
c491 c492 c493 c494 c495 c496 c497 c498 c499 c500
Time-varying observation innovation loading matrix D contains unknown parameters:
c501 c502 c503 c504 c505 c506 c507 c508 c509 c510 c511 c512 c513 c514 c515 c516 c517 c5
c521 c522 c523 c524 c525 c526 c527 c528 c529 c530 c531 c532 c533 c534 c535 c536 c537 c5
c541 c542 c543 c544 c545 c546 c547 c548 c549 c550 c551 c552 c553 c554 c555 c556 c557 c5
c561 c562 c563 c564 c565 c566 c567 c568 c569 c570 c571 c572 c573 c574 c575 c576 c577 c5
c581 c582 c583 c584 c585 c586 c587 c588 c589 c590 c591 c592 c593 c594 c595 c596 c597 c5

Initial state distribution:

Initial state means
  x1   x2   x3   x4
 NaN  NaN  NaN  NaN

Initial state covariance matrix
     x1   x2   x3   x4
 x1  NaN  NaN  NaN  NaN
 x2  NaN  NaN  NaN  NaN
 x3  NaN  NaN  NaN  NaN
 x4  NaN  NaN  NaN  NaN

State types
     x1          x2          x3          x4
 Stationary  Stationary  Stationary  Stationary


State-space model type: <a href="matlab: doc ssm">ssm</a>

State vector length: Time-varying
Observation vector length: 1
State disturbance vector length: Time-varying
Observation innovation vector length: 1
Sample size supported by model: 100
Unknown parameters for estimation: 620
```

```
State variables: x1, x2,...
State disturbances: u1, u2,...
Observation series: y1, y2,...
Observation innovations: e1, e2,...
Unknown parameters: c1, c2,...

State equations (in period 52):
x1(t) = (c153)x1(t-1) + (c154)x2(t-1) + (c302)u1(t)
x2(t) = x1(t-1)
Time-varying transition matrix A contains unknown parameters:
c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15 c16 c17 c18 c19 c20
c21 c22 c23 c24 c25 c26 c27 c28 c29 c30 c31 c32 c33 c34 c35 c36 c37 c38 c39 c40
c41 c42 c43 c44 c45 c46 c47 c48 c49 c50 c51 c52 c53 c54 c55 c56 c57 c58 c59 c60
c61 c62 c63 c64 c65 c66 c67 c68 c69 c70 c71 c72 c73 c74 c75 c76 c77 c78 c79 c80
c81 c82 c83 c84 c85 c86 c87 c88 c89 c90 c91 c92 c93 c94 c95 c96 c97 c98 c99 c100
c101 c102 c103 c104 c105 c106 c107 c108 c109 c110 c111 c112 c113 c114 c115 c116 c117 c
c121 c122 c123 c124 c125 c126 c127 c128 c129 c130 c131 c132 c133 c134 c135 c136 c137 c
c141 c142 c143 c144 c145 c146 c147 c148 c149 c150 c151 c152 c153 c154 c155 c156 c157 c
c161 c162 c163 c164 c165 c166 c167 c168 c169 c170 c171 c172 c173 c174 c175 c176 c177 c
c181 c182 c183 c184 c185 c186 c187 c188 c189 c190 c191 c192 c193 c194 c195 c196 c197 c
c201 c202 c203 c204 c205 c206 c207 c208 c209 c210 c211 c212 c213 c214 c215 c216 c217 c2
c221 c222 c223 c224 c225 c226 c227 c228 c229 c230 c231 c232 c233 c234 c235 c236 c237 c2
c241 c242 c243 c244 c245 c246 c247 c248 c249 c250
Time-varying state disturbance loading matrix B contains unknown parameters:
c251 c252 c253 c254 c255 c256 c257 c258 c259 c260 c261 c262 c263 c264 c265 c266 c267 c2
c271 c272 c273 c274 c275 c276 c277 c278 c279 c280 c281 c282 c283 c284 c285 c286 c287 c2
c291 c292 c293 c294 c295 c296 c297 c298 c299 c300 c301 c302 c303 c304 c305 c306 c307 c3
c311 c312 c313 c314 c315 c316 c317 c318 c319 c320 c321 c322 c323 c324 c325 c326 c327 c3
c331 c332 c333 c334 c335 c336 c337 c338 c339 c340 c341 c342 c343 c344 c345 c346 c347 c3

Observation equation (in period 52):
y1(t) = (c452)x1(t) + (c552)e1(t)
Time-varying measurement sensitivity matrix C contains unknown parameters:
c351 c352 c353 c354 c355 c356 c357 c358 c359 c360 c361 c362 c363 c364 c365 c366 c367 c3
c371 c372 c373 c374 c375 c376 c377 c378 c379 c380 c381 c382 c383 c384 c385 c386 c387 c3
c391 c392 c393 c394 c395 c396 c397 c398 c399 c400 c401 c402 c403 c404 c405 c406 c407 c4
c411 c412 c413 c414 c415 c416 c417 c418 c419 c420 c421 c422 c423 c424 c425 c426 c427 c4
c431 c432 c433 c434 c435 c436 c437 c438 c439 c440 c441 c442 c443 c444 c445 c446 c447 c4
c451 c452 c453 c454 c455 c456 c457 c458 c459 c460 c461 c462 c463 c464 c465 c466 c467 c4
c471 c472 c473 c474 c475 c476 c477 c478 c479 c480 c481 c482 c483 c484 c485 c486 c487 c4
c491 c492 c493 c494 c495 c496 c497 c498 c499 c500
Time-varying observation innovation loading matrix D contains unknown parameters:
c501 c502 c503 c504 c505 c506 c507 c508 c509 c510 c511 c512 c513 c514 c515 c516 c517 c5
```

```
c521 c522 c523 c524 c525 c526 c527 c528 c529 c530 c531 c532 c533 c534 c535 c536 c537 c5
c541 c542 c543 c544 c545 c546 c547 c548 c549 c550 c551 c552 c553 c554 c555 c556 c557 c5
c561 c562 c563 c564 c565 c566 c567 c568 c569 c570 c571 c572 c573 c574 c575 c576 c577 c5
c581 c582 c583 c584 c585 c586 c587 c588 c589 c590 c591 c592 c593 c594 c595 c596 c597 c5

Initial state distribution:

Initial state means
  x1   x2   x3   x4
 NaN  NaN  NaN  NaN

Initial state covariance matrix
     x1   x2   x3   x4
 x1  NaN  NaN  NaN  NaN
 x2  NaN  NaN  NaN  NaN
 x3  NaN  NaN  NaN  NaN
 x4  NaN  NaN  NaN  NaN

State types
     x1          x2          x3          x4
 Stationary  Stationary  Stationary  Stationary
```

The software attributes a different set of coefficients for each period. You might experience numerical issues when you estimate such models. To reuse parameters among groups of periods, consider creating a parameter-to-matrix mapping function.

- "Create State-Space Model with Random State Coefficient" on page 8-38

Algorithms

- If you implicitly create `Mdl`, and if the software cannot infer locations for unknown parameters from the parameter-to-matrix function, then the software evaluates these parameters using their initial values and displays them as numeric values. This evaluation can occur when the parameter-to-matrix function has a random, unknown coefficient, which is a convenient form for a Monte Carlo study.
- The software displays the initial state distributions as numeric values. This type of display occurs because, in many cases, the initial distribution depends on the values of the state equation matrices `A` and `B`. These values are often a complicated function of unknown parameters. In such situations, the software does not display the initial distribution symbolically. Additionally, if `Mean0` and `Cov0` contain

unknown parameters, then the software evaluates and displays numeric values for the unknown parameters.

References

[1] Durbin J., and S. J. Koopman. *Time Series Analysis by State Space Methods*. 2nd ed. Oxford: Oxford University Press, 2012.

See Also

estimate | filter | forecast | simulate | smooth | ssm

More About

- "What Are State-Space Models?" on page 8-3

dssm class

Create diffuse state-space model

Description

`dssm` creates a linear diffuse state-space model with independent Gaussian state disturbances and observation innovations. A diffuse state-space model contains diffuse states, and variances of the initial distributions of diffuse states are `Inf`. All diffuse states are independent of each other and all other states. The software implements the diffuse Kalman filter for filtering, smoothing, and parameter estimation.

You can:

- Specify a time-invariant or time-varying model.
- Specify whether states are stationary, static, or nonstationary.
- Specify the state-transition, state-disturbance-loading, measurement-sensitivity, or observation-innovation matrices:
  - Explicitly by providing the matrices
  - Implicitly by providing a function that maps the parameters to the matrices, that is, a parameter-to-matrix mapping function

After creating a diffuse state-space model containing unknown parameters, you can estimate its parameters by passing the created `dssm` model object and data to estimate. The `estimate` function builds the likelihood function using the diffuse Kalman filter.

Use a completely specified model (that is, all parameter values of the model are known) to:

- Filter or smooth states using filter or smooth, respectively. These functions apply the diffuse Kalman filter and data to the state-space model.
- Forecast states or observations using forecast.

Construction

`Mdl = dssm(A,B,C)` creates a diffuse state-space model (`Mdl`) using state-transition matrix `A`, state-disturbance-loading matrix `B`, and measurement-sensitivity matrix `C`.

`Mdl = dssm(A,B,C,D)` creates a diffuse state-space model using state-transition matrix `A`, state-disturbance-loading matrix `B`, measurement-sensitivity matrix `C`, and observation-innovation matrix `D`.

`Mdl = dssm(___,Name,Value)` uses any of the input arguments in the previous syntaxes and additional options that you specify by one or more `Name,Value` pair arguments.

`Mdl = dssm(ParamMap)` creates a diffuse state-space model using a parameter-to-matrix mapping function (`ParamMap`) that you write. The function maps a vector of parameters to the matrices `A`, `B`, and `C`. Optionally, `ParamMap` can map parameters to `D`, `Mean0`, `Cov0`. To specify the types of states, the function can return `StateType`. To accommodate a regression component in the observation equation, `ParamMap` can also return deflated observation data.

`Mdl = dssm(SSMMdl)` converts a state-space model object (`SSMMdl`) to a diffuse state-space model object (`Mdl`). `dssm` sets all initial variances of diffuse states in `SSMMdl.Cov0` to `Inf`.

Input Arguments

A — State-transition coefficient matrix

matrix | cell vector of matrices

State-transition coefficient matrix for explicit state-space model creation, specified as a matrix or cell vector of matrices.

The state-transition coefficient matrix, A_t, specifies how the states, x_t, are expected to transition from period $t-1$ to t, for all $t = 1,\ldots,T$. That is, the expected state-transition equation at period t is $E(x_t \mid x_{t-1}) = A_t x_{t-1}$.

For time-invariant state-space models, specify `A` as an m-by-m matrix, where m is the number of states per period.

For time-varying state-space models, specify `A` as a T-dimensional cell array, where `A{t}` contains an m_t-by-m_{t-1} state-transition coefficient matrix. If the number of states changes from period $t-1$ to t, then $m_t \neq m_{t-1}$.

`NaN` values in any coefficient matrix indicate unique, unknown parameters in the state-space model. `A` contributes:

- `sum(isnan(A(:)))` unknown parameters to time-invariant state-space models. In other words, if the state-space model is time invariant, then the software uses the same unknown parameters defined in `A` at each period.
- `numParamsA` unknown parameters to time-varying state-space models, where `numParamsA = sum(cell2mat(cellfun(@(x)sum(sum(isnan(x))),A,'UniformOutput',0`. In other words, if the state-space model is time varying, then the software assigns a new set of parameters for each matrix in `A`.

You cannot specify `A` and `ParamMap` simultaneously.

Data Types: `double | cell`

B — State-disturbance-loading coefficient matrix

matrix | cell vector of matrices

State-disturbance-loading coefficient matrix for explicit state-space model creation, specified as a matrix or cell vector of matrices.

The state disturbances, u_t, are independent Gaussian random variables with mean 0 and standard deviation 1. The state-disturbance-loading coefficient matrix, B_t, specifies the additive error structure in the state-transition equation from period $t - 1$ to t, for all $t = 1,...,T$. That is, the state-transition equation at period t is $x_t = A_t x_{t-1} + B_t u_t$.

For time-invariant state-space models, specify `B` as an m-by-k matrix, where m is the number of states and k is the number of state disturbances per period. `B*B'` is the state-disturbance covariance matrix for all periods.

For time-varying state-space models, specify `B` as a T-dimensional cell array, where `B{t}` contains an m_t-by-k_t state-disturbance-loading coefficient matrix. If the number of states or state disturbances changes at period t, then the matrix dimensions between `B{t-1}` and `B{t}` vary. `B{t}*B{t}'` is the state-disturbance covariance matrix for period `t`.

`NaN` values in any coefficient matrix indicate unique, unknown parameters in the state-space model. `B` contributes:

- `sum(isnan(B(:)))` unknown parameters to time-invariant state-space models. In other words, if the state-space model is time invariant, then the software uses the same unknown parameters defined in `B` at each period.
- `numParamsB` unknown parameters to time-varying state-space models, where `numParamsB = sum(cell2mat(cellfun(@(x)sum(sum(isnan(x))),B,'UniformOutput',0)))`.

In other words, if the state-space model is time varying, then the software assigns a new set of parameters for each matrix in `B`.

You cannot specify `B` and `ParamMap` simultaneously.

Data Types: `double` | `cell`

`C` — Measurement-sensitivity coefficient matrix

matrix | cell vector of matrices

Measurement-sensitivity coefficient matrix for explicit state-space model creation, specified as a matrix or cell vector of matrices.

The measurement-sensitivity coefficient matrix, C_t, specifies how the states are expected to linearly combine at period t to form the observations, y_t, for all $t = 1,...,T$. That is, the expected observation equation at period t is $E(y_t | x_t) = C_t x_t$.

For time-invariant state-space models, specify `C` as an n-by-m matrix, where n is the number of observations and m is the number of states per period.

For time-varying state-space models, specify `C` as a T-dimensional cell array, where `C{t}` contains an n_t-by-m_t measurement-sensitivity coefficient matrix. If the number of states or observations changes at period t, then the matrix dimensions between `C{t-1}` and `C{t}` vary.

`NaN` values in any coefficient matrix indicate unique, unknown parameters in the state-space model. `C` contributes:

- `sum(isnan(C(:)))` unknown parameters to time-invariant state-space models. In other words, if the state-space model is time invariant, then the software uses the same unknown parameters defined in `C` at each period.
- `numParamsC` unknown parameters to time-varying state-space models, where `numParamsC = sum(cell2mat(cellfun(@(x)sum(sum(isnan(x))),C,'UniformOutput',0)))`. In other words, if the state-space model is time varying, then the software assigns a new set of parameters for each matrix in `C`.

You cannot specify `C` and `ParamMap` simultaneously.

Data Types: `double` | `cell`

`D` — Observation-innovation coefficient matrix

`[]` (default) | matrix | cell vector of matrices

Observation-innovation coefficient matrix for explicit state-space model creation, specified as a matrix or cell vector of matrices.

The observation innovations, ε_t, are independent Gaussian random variables with mean 0 and standard deviation 1. The observation-innovation coefficient matrix, D_t, specifies the additive error structure in the observation equation at period t, for all $t = 1,...,T$. That is, the observation equation at period t is $y_t = C_t x_t + D_t \varepsilon_t$.

For time-invariant state-space models, specify `D` as an n-by-h matrix, where n is the number of observations and h is the number of observation innovations per period. `D*D'` is the observation-innovation covariance matrix for all periods.

For time-varying state-space models, specify `D` as a T-dimensional cell array, where `D{t}` contains an n_t-by-h_t matrix. If the number of observations or observation innovations changes at period t, then the matrix dimensions between `D{t-1}` and `D{t}` vary. `D{t}*D{t}'` is the observation-innovation covariance matrix for period `t`.

`NaN` values in any coefficient matrix indicate unique, unknown parameters in the state-space model. `D` contributes:

- `sum(isnan(D(:)))` unknown parameters to time-invariant state-space models. In other words, if the state-space model is time invariant, then the software uses the same unknown parameters defined in `D` at each period.
- `numParamsD` unknown parameters to time-varying state-space models, where `numParamsD = sum(cell2mat(cellfun(@(x)sum(sum(isnan(x))),D,'UniformOutput',0)))`. In other words, if the state-space model is time varying, then the software assigns a new set of parameters for each matrix in `D`.

By default, `D` is an empty matrix indicating no observation innovations in the state-space model.

You cannot specify `D` and `ParamMap` simultaneously.

Data Types: `double | cell`

`ParamMap` — Parameter-to-matrix mapping function

empty array (`[ ]`) (default) | function handle

Parameter-to-matrix mapping function for implicit state-space model creation, specified as a function handle.

`ParamMap` must be a function that takes at least one input argument and returns at least three output arguments. The requisite input argument is a vector of unknown parameters, and the requisite output arguments correspond to the coefficient matrices `A`, `B`, and `C`, respectively. If your parameter-to-mapping function requires the input parameter vector argument only, then implicitly create a diffuse state-space model by entering the following:

```
Mdl = dssm(@ParamMap)
```

In general, you can write an intermediate function, for example, `ParamFun`, using this syntax:

```
function [A,B,C,D,Mean0,Cov0,StateType,DeflateY] = ...
    ParamFun(params,...otherInputArgs...)
```

In this general case, create the diffuse state-space model by entering

```
Mdl = dssm(@(params)ParamMap(params,...otherInputArgs...))
```

However:

- Follow the order of the output arguments.
- `params` is a vector, and each element corresponds to an unknown parameter.
- `ParamFun` must return `A`, `B`, and `C`, which correspond to the state-transition, state-disturbance-loading, and measurement-sensitivity coefficient matrices, respectively.
- If you specify more input arguments than the parameter vector (`params`), such as observed responses and predictors, then implicitly create the diffuse state-space model using the syntax pattern

  ```
  Mdl = dssm(@(params)ParamFun(params,y,z))
  ```
- For the optional output arguments `D`, `Mean0`, `Cov0`, `StateType`, and `DeflateY`:
  - The optional output arguments correspond to the observation-innovation coefficient matrix `D` and the name-value pair arguments `Mean0`, `Cov0`, and `StateType`.
  - To skip specifying an optional output argument, set the argument to `[]` in the function body. For example, to skip specifying `D`, then set `D = [];` in the function.
  - `DeflateY` is the deflated-observation data, which accommodates a regression component in the observation equation. For example, in this function, which has a linear regression component, `Y` is the vector of observed responses and `Z` is the vector of predictor data.

```
function [A,B,C,D,Mean0,Cov0,StateType,DeflateY] = ParamFun(params,Y,Z)
 ...
 DeflateY = Y - params(9) - params(10)*Z;
 ...
end
```

  - For the default values of `Mean0`, `Cov0`, and `StateType`, see "Algorithms" on page 9-996.

- It is best practice to:
  - Load the data to the MATLAB Workspace before specifying the model.
  - Create the parameter-to-matrix mapping function as its own file.

If you specify `ParamMap`, then you cannot specify any name-value pair arguments or any other input arguments.

Data Types: `function_handle`

`SSMMdl` — State-space model

`ssm` model object

State-space model to convert to a diffuse state-space model, specified as an ssm model object.

`dssm` sets all initial variances of diffuse states in `SSMMdl.Cov0` to `Inf`.

To use the diffuse Kalman filter for filtering, smoothing, and parameter estimation instead of the standard Kalman filter, convert a state-space model to a diffuse state-space model.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'Mean0'` — Initial state mean

numeric vector

Initial state mean for explicit state-space model creation, specified as the comma-separated pair consisting of `'Mean0'` and a numeric vector with length equal to the number of initial states. For the default values, see "Algorithms" on page 9-996.

If you specify `ParamMap`, then you cannot specify `Mean0`. Instead, specify the initial state mean in the parameter-to-matrix mapping function.

Data Types: `double`

`'Cov0'` — Initial state covariance matrix

square matrix

Initial state covariance matrix for explicit state-space model creation, specified as the comma-separated pair consisting of `'Cov0'` and a square matrix with dimensions equal to the number of initial states. For the default values, see “Algorithms” on page 9-996.

If you specify `ParamMap`, then you cannot specify `Cov0`. Instead, specify the initial state covariance in the parameter-to-matrix mapping function.

Data Types: `double`

`'StateType'` — Initial state distribution indicator

0 | 1 | 2

Initial state distribution indicator for explicit state-space model creation, specified as the comma-separated pair consisting of `'StateType'` and a numeric vector with length equal to the number of initial states. This table summarizes the available types of initial state distributions.

Value	Initial State Distribution Type
0	Stationary (for example, ARMA models)
1	The constant 1 (that is, the state is 1 with probability 1)
2	Diffuse or nonstationary (for example, random walk model, seasonal linear time series) or static state

For example, suppose that the state equation has two state variables: The first state variable is an AR(1) process, and the second state variable is a random walk. Specify the initial distribution types by setting `'StateType',[0; 2]`.

If you specify `ParamMap`, then you cannot specify `Mean0`. Instead, specify the initial state distribution indicator in the parameter-to-matrix mapping function.

For the default values, see “Algorithms” on page 9-996.

Data Types: `double`

Properties

A — State-transition coefficient matrix

matrix | cell vector of matrices | empty array ([ ])

State-transition coefficient matrix for explicitly created state-space models, specified as a matrix, a cell vector of matrices, or an empty array ([ ]). For implicitly created state-space models and before estimation, `A` is `[ ]` and read only.

The state-transition coefficient matrix, A_t, specifies how the states, x_t, are expected to transition from period $t - 1$ to t, for all $t = 1,...,T$. That is, the expected state-transition equation at period t is $E(x_t \mid x_{t-1}) = A_t x_{t-1}$.

For time-invariant state-space models, `A` is an m-by-m matrix, where m is the number of states per period.

For time-varying state-space models, `A` is a T-dimensional cell array, where `A{t}` contains an m_t-by-m_{t-1} state-transition coefficient matrix. If the number of states changes from period $t - 1$ to t, then $m_t \neq m_{t-1}$.

`NaN` values in any coefficient matrix indicate unknown parameters in the state-space model. `A` contributes:

- `sum(isnan(A(:)))` unknown parameters to time-invariant state-space models. In other words, if the state-space model is time invariant, then the software uses the same unknown parameters defined in `A` at each period.
- `numParamsA` unknown parameters to time-varying state-space models, where `numParamsA = sum(cell2mat(cellfun(@(x)sum(sum(isnan(x))),A,'UniformOutput',(`
  In other words, if the state-space model is time varying, then the software assigns a new set of parameters for each matrix in `A`.

Data Types: `double` | `cell`

B — State-disturbance-loading coefficient matrix

matrix | cell vector of matrices | empty array ([ ])

State-disturbance-loading coefficient matrix for explicitly created state-space models, specified as a matrix, a cell vector of matrices, or an empty array ([ ]). For implicitly created state-space models and before estimation, `B` is `[ ]` and read only.

The state disturbances, u_t, are independent Gaussian random variables with mean 0 and standard deviation 1. The state-disturbance-loading coefficient matrix, B_t, specifies the additive error structure in the state-transition equation from period $t - 1$ to t, for all $t = 1,...,T$. That is, the state-transition equation at period t is $x_t = A_t x_{t-1} + B_t u_t$.

For time-invariant state-space models, `B` is an m-by-k matrix, where m is the number of states and k is the number of state disturbances per period. `B*B'` is the state-disturbance covariance matrix for all periods.

For time-varying state-space models, `B` is a T-dimensional cell array, where `B{t}` contains an m_t-by-k_t state-disturbance-loading coefficient matrix. If the number of states or state disturbances changes at period t, then the matrix dimensions between `B{t-1}` and `B{t}` vary. `B{t}*B{t}'` is the state-disturbance covariance matrix for period `t`.

`NaN` values in any coefficient matrix indicate unknown parameters in the state-space model. `B` contributes:

- `sum(isnan(B(:)))` unknown parameters to time-invariant state-space models. In other words, if the state-space model is time invariant, then the software uses the same unknown parameters defined in `B` at each period.
- `numParamsB` unknown parameters to time-varying state-space models, where `numParamsB = sum(cell2mat(cellfun(@(x)sum(sum(isnan(x))),B,'UniformOutput',0)))`. In other words, if the state-space model is time varying, then the software assigns a new set of parameters for each matrix in `B`.

Data Types: `double` | `cell`

`C` — Measurement-sensitivity coefficient matrix

matrix | cell vector of matrices | empty array (`[]`)

Measurement-sensitivity coefficient matrix for explicitly created state-space models, specified as a matrix, a cell vector of matrices, or an empty array (`[]`). For implicitly created state-space models and before estimation, `C` is `[]` and read only.

The measurement-sensitivity coefficient matrix, C_t, specifies how the states are expected to combine linearly at period t to form the observations, y_t, for all $t = 1,...,T$. That is, the expected observation equation at period t is $E(y_t | x_t) = C_t x_t$.

For time-invariant state-space models, `C` is an n-by-m matrix, where n is the number of observations and m is the number of states per period.

For time-varying state-space models, `C` is a T-dimensional cell array, where `C{t}` contains an n_t-by-m_t measurement-sensitivity coefficient matrix. If the number of states or observations changes at period t, then the matrix dimensions between `C{t-1}` and `C{t}` vary.

`NaN` values in any coefficient matrix indicate unknown parameters in the state-space model. `C` contributes:

- `sum(isnan(C(:)))` unknown parameters to time-invariant state-space models. In other words, if the state-space model is time invariant, then the software uses the same unknown parameters defined in `C` at each period.
- `numParamsC` unknown parameters to time-varying state-space models, where `numParamsC = sum(cell2mat(cellfun(@(x)sum(sum(isnan(x))),C,'UniformOutput',0)))`. In other words, if the state-space model is time varying, then the software assigns a new set of parameters for each matrix in `C`.

Data Types: `double` | `cell`

`D` — Observation-innovation coefficient matrix

matrix | cell vector of matrices | empty array (`[]`)

Observation-innovation coefficient matrix for explicitly created state-space models, specified as a matrix, a cell vector of matrices, or an empty array (`[]`). For implicitly created state-space models and before estimation, `D` is `[]` and read only.

The observation innovations, ε_t, are independent Gaussian random variables with mean 0 and standard deviation 1. The observation-innovation coefficient matrix, D_t, specifies the additive error structure in the observation equation at period t, for all $t = 1,\ldots,T$. That is, the observation equation at period t is $y_t = C_t x_t + D_t \varepsilon_t$.

For time-invariant state-space models, `D` is an n-by-h matrix, where n is the number of observations and h is the number of observation innovations per period. `D*D'` is the observation-innovation covariance matrix for all peroids.

For time-varying state-space models, `D` is a T-dimensional cell array, where `D{t}` contains an n_t-by-h_t matrix. If the number of observations or observation innovations changes at period t, then the matrix dimensions between `D{t-1}` and `D{t}` vary. `D{t}*D{t}'` is the state-disturbance covariance matrix for period `t`.

`NaN` values in any coefficient matrix indicate unknown parameters in the state-space model. `D` contributes:

- `sum(isnan(D(:)))` unknown parameters to time-invariant state-space models. In other words, if the state-space model is time invariant, then the software uses the same unknown parameters defined in `D` at each period.
- `numParamsD` unknown parameters to time-varying state-space models, where `numParamsD = sum(cell2mat(cellfun(@(x)sum(sum(isnan(x))),D,'UniformOutput',0)))`. In other words, if the state-space model is time varying, then the software assigns a new set of parameters for each matrix in `D`.

Data Types: `double` | `cell`

`Mean0` — Initial state mean

numeric vector | empty array (`[]`)

Initial state mean, specified as a numeric vector or an empty array (`[]`). `Mean0` has length equal to the number of initial states (`size(A,1)` or `size(A{1},1)`).

`Mean0` is the mean of the Gaussian distribution of the states at period 0.

For implicitly created state-space models and before estimation, `Mean0` is `[]` and read only. However, `estimate` specifies `Mean0` after estimation.

Data Types: `double`

`Cov0` — Initial state covariance matrix

square matrix | empty array (`[]`)

Initial state covariance matrix, specified as a square matrix or an empty array (`[]`). `Cov0` has dimensions equal to the number of initial states (`size(A,1)` or `size(A{1},1)`).

`Cov0` is the covariance of the Gaussian distribution of the states at period 0.

For implicitly created state-space models and before estimation, `Cov0` is `[]` and read only. However, `estimate` specifies `Cov0` after estimation.

Diagonal elements of `Cov0` that have value `Inf` correspond to diffuse initial state distributions. This specification indicates complete ignorance or no prior knowledge of the initial state value. Subsequently, the software filters, smooths, and estimates parameters in the presence of diffuse initial state distributions using the diffuse Kalman filter. To use the standard Kalman filter for diffuse states instead, set each diagonal element of `Cov0` to a large, positive value, for example, `1e7`. This specification suggests relatively weak knowledge of the initial state value.

Data Types: `double`

`StateType` — Initial state distribution type

numeric vector | empty array ([])

Initial state distribution indicator, specified as a numeric vector or empty array ([]). `StateType` has length equal to the number of initial states.

For implicitly created state-space models or models with unknown parameters, `StateType` is `[]` and read only.

This table summarizes the available types of initial state distributions.

Value	Initial State Distribution Type
0	Stationary (e.g., ARMA models)
1	The constant 1 (that is, the state is 1 with probability 1)
2	Nonstationary (e.g., random walk model, seasonal linear time series) or static state.

For example, suppose that the state equation has two state variables: The first state variable is an AR(1) process, and the second state variable is a random walk. Then, `StateType` is `[0; 2]`.

For nonstationary states, `dssm` sets `Cov0` to `Inf` by default. Subsequently, the software assumes that diffuse states are uncorrelated and implements the diffuse Kalman filter for filtering, smoothing, and parameter estimation. This specification imposes no prior knowledge on the initial state values of diffuse states.

Data Types: `double`

`ParamMap` — Parameter-to-matrix mapping function

function handle | empty array ([])

Parameter-to-matrix mapping function, specified as a function handle or an empty array ([]). `ParamMap` completely specifies the structure of the state-space model. That is, `ParamMap` defines `A`, `B`, `C`, `D`, and, optionally, `Mean0`, `Cov0`, and `StateType`. For explicitly created state-space models, `ParamMap` is `[]` and read only.

Data Types: `function_handle`

Methods

disp	Display summary information for diffuse state-space model
estimate	Maximum likelihood parameter estimation of diffuse state-space models
filter	Forward recursion of diffuse state-space models
forecast	Forecast states and observations of diffuse state-space models
refine	Refine initial parameters to aid diffuse state-space model estimation
smooth	Backward recursion of diffuse state-space models

Definitions

Static State

A *static state* does not change in value throughout the sample, that is, $P(x_{t+1} = x_t) = 1$ for all $t = 1,...,T$.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Explicitly Create Diffuse State-Space Model Containing Known and Unknown Parameters

Create a diffuse state-space model containing two independent states, $x_{1,t}$ and $x_{3,t}$, and an observation, y_t, that is the deterministic sum of the two states at time t. x_1 is an AR(1) model with a constant and x_3 is a random walk. Symbolically, the state-space model is

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \\ x_{3,t} \end{bmatrix} = \begin{bmatrix} \phi_1 & c_1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{t-1,1} \\ x_{t-1,2} \\ x_{t-1,3} \end{bmatrix} + \begin{bmatrix} \sigma_1 & 0 \\ 0 & 0 \\ 0 & \sigma_2 \end{bmatrix} \begin{bmatrix} u_{t,1} \\ u_{t,3} \end{bmatrix}$$

$$y_t = \begin{bmatrix} 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{t,1} \\ x_{t,2} \\ x_{t,3} \end{bmatrix}.$$

The state disturbances, $u_{1,t}$ and $u_{3,t}$, are standard Gaussian random variables.

Specify the state-transition matrix.

```
A = [NaN NaN 0; 0 1 0; 0 0 1];
```

The `NaN` values indicate unknown parameters.

Specify the state-disturbance-loading matrix.

```
B = [NaN 0; 0 0; 0 NaN];
```

Specify the measurement-sensitivity matrix.

```
C = [1 0 1];
```

Create a vector that specifies the state types. In this example:

- $x_{1,t}$ is a stationary AR(1) model, so its state type is `0`.
- $x_{2,t}$ is a placeholder for the constant in the AR(1) model. Because the constant is unknown and is expressed in the first equation, $x_{2,t}$ is `1` for the entire series. Therefore, its state type is `1`.
- $x_{3,t}$ is a nonstationary, random walk with drift, so its state type is `2`.

```
StateType = [0 1 2];
```

Create the state-space model using `dssm`.

```
Mdl = dssm(A,B,C,'StateType',StateType)
```

```
Mdl =
```

```
State-space model type: <a href="matlab: doc dssm">dssm</a>

State vector length: 3
Observation vector length: 1
State disturbance vector length: 2
Observation innovation vector length: 0
Sample size supported by model: Unlimited
Unknown parameters for estimation: 4

State variables: x1, x2,...
State disturbances: u1, u2,...
Observation series: y1, y2,...
Observation innovations: e1, e2,...
Unknown parameters: c1, c2,...

State equations:
x1(t) = (c1)x1(t-1) + (c2)x2(t-1) + (c3)u1(t)
x2(t) = x2(t-1)
x3(t) = x3(t-1) + (c4)u2(t)

Observation equation:
y1(t) = x1(t) + x3(t)

Initial state distribution:

Initial state means are not specified.
Initial state covariance matrix is not specified.
State types
     x1         x2        x3
 Stationary  Constant  Diffuse
```

`Mdl` is a `dssm` model object containing unknown parameters. A detailed summary of `Mdl` prints to the Command Window. If you do not specify the initial state covariance matrix, then the initial variance of $x_{3,t}$ is `Inf`.

It is good practice to verify that the state and observation equations are correct. If the equations are not correct, then expand the state-space equation and verify it manually.

Explicitly Create Diffuse State-Space Model Containing Observation Error

Create a diffuse state-space model containing two random walk states. The observations are the sum of the two states, plus Gaussian error. Symbolically, the equation is

$$\begin{bmatrix} x_{t,1} \\ x_{t,2} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_{t-1,1} \\ x_{t-1,2} \end{bmatrix} + \begin{bmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{bmatrix} \begin{bmatrix} u_{t,1} \\ u_{t,2} \end{bmatrix}$$

$$y_t = \begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} x_{t,1} \\ x_{t,2} \end{bmatrix} + \sigma_3 \varepsilon_t.$$

Define the state-transition matrix.

```
A = [1 0; 0 1];
```

Define the state-disturbance-loading matrix.

```
B = [NaN 0; 0 NaN];
```

Define the measurement-sensitivity matrix.

```
C = [1 1];
```

Define the observation-innovation matrix.

```
D = NaN;
```

Create a vector that specifies that both states are nonstationary.

```
StateType = [2; 2];
```

Create the state-space model using `dssm`.

```
Mdl = dssm(A,B,C,D,'StateType',StateType)
```

```
Mdl =


State-space model type: <a href="matlab: doc dssm">dssm</a>

State vector length: 2
Observation vector length: 1
State disturbance vector length: 2
Observation innovation vector length: 1
Sample size supported by model: Unlimited
Unknown parameters for estimation: 3

State variables: x1, x2,...
State disturbances: u1, u2,...
```

```
Observation series: y1, y2,...
Observation innovations: e1, e2,...
Unknown parameters: c1, c2,...

State equations:
x1(t) = x1(t-1) + (c1)u1(t)
x2(t) = x2(t-1) + (c2)u2(t)

Observation equation:
y1(t) = x1(t) + x2(t) + (c3)e1(t)

Initial state distribution:

Initial state means are not specified.
Initial state covariance matrix is not specified.
State types
    x1       x2
 Diffuse  Diffuse
```

`Mdl` is an `dssm` model containing unknown parameters. A detailed summary of `Mdl` prints to the Command Window.

Pass the data and `Mdl` to `estimate` to estimate the parameters. During estimation, the initial state variances are `Inf`, and `estimate` implements the diffuse Kalman filter.

Create Known Diffuse State-Space Model with Initial State Values

Create a diffuse state-space model, where:

- The state $x_{1,t}$ is a stationary AR(2) model with $\phi_1 = 0.6$, $\phi_2 = 0.2$, and a constant 0.5. The state disturbance is a mean zero Gaussian random variable with standard deviation 0.3.
- The state $x_{4,t}$ is a random walk. The state disturbance is a mean zero Gaussian random variable with standard deviation 0.05.
- The observation $y_{1,t}$ is the difference between the current and previous value in the AR(2) state, plus a mean 0 Gaussian observation innovation with standard deviation 0.1.
- The observation $y_{2,t}$ is the random walk state plus a mean 0 Gaussian observation innovation with standard deviation 0.02.

Symbolically, the state-space model is

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \\ x_{3,t} \\ x_{4,t} \end{bmatrix} = \begin{bmatrix} 0.6 & 0.2 & 0.5 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \\ x_{3,t-1} \\ x_{4,t-1} \end{bmatrix} + \begin{bmatrix} 0.3 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0.05 \end{bmatrix} \begin{bmatrix} u_{1,t} \\ u_{4,t} \end{bmatrix}$$

$$\begin{bmatrix} y_{1,t} \\ y_{2,t} \end{bmatrix} = \begin{bmatrix} 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{1,t} \\ x_{2,t} \\ x_{3,t} \\ x_{4,t} \end{bmatrix} + \begin{bmatrix} 0.1 & 0 \\ 0 & 0.02 \end{bmatrix} \begin{bmatrix} \varepsilon_{1,t} \\ \varepsilon_{2,t} \end{bmatrix}.$$

The model has four states: $x_{1,t}$ is the AR(2) process, $x_{2,t}$ represents $x_{1,t-1}$, $x_{3,t}$ is the AR(2) model constant, and $x_{4,t}$ is the random walk.

Define the state-transition matrix.

```
A = [0.6 0.2 0.5 0; 1 0 0 0; 0 0 1 0; 0 0 0 1];
```

Define the state-disturbance-loading matrix.

```
B = [0.3 0; 0 0; 0 0; 0 0.05];
```

Define the measurement-sensitivity matrix.

```
C = [1 -1 0 0; 0 0 0 1];
```

Define the observation-innovation matrix.

```
D = [0.1; 0.02];
```

Use `dssm` to create the state-space model. Identify the type of initial state distributions (`StateType`) by noting the following:

- $x_{1,t}$ is a stationary AR(2) process.
- $x_{2,t}$ is also a stationary AR(2) process.
- $x_{3,t}$ is the constant 1 for all periods.
- $x_{4,t}$ is nonstationary.

Set the initial state means (`Mean0`) to 0. The initial state mean for constant states must be 1.

```
Mean0 = [0; 0; 1; 0];
StateType = [0; 0; 1; 2];
```

```
Mdl = dssm(A,B,C,D,'MeanO',MeanO,'StateType',StateType)


Mdl =


State-space model type: <a href="matlab: doc dssm">dssm</a>

State vector length: 4
Observation vector length: 2
State disturbance vector length: 2
Observation innovation vector length: 1
Sample size supported by model: Unlimited

State variables: x1, x2,...
State disturbances: u1, u2,...
Observation series: y1, y2,...
Observation innovations: e1, e2,...

State equations:
x1(t) = (0.60)x1(t-1) + (0.20)x2(t-1) + (0.50)x3(t-1) + (0.30)u1(t)
x2(t) = x1(t-1)
x3(t) = x3(t-1)
x4(t) = x4(t-1) + (0.05)u2(t)

Observation equations:
y1(t) = x1(t) - x2(t) + (0.10)e1(t)
y2(t) = x4(t) + (0.02)e1(t)

Initial state distribution:

Initial state means
 x1  x2  x3  x4
  0   0   1   0

Initial state covariance matrix
     x1    x2    x3  x4
 x1  0.21  0.16   0  0
 x2  0.16  0.21   0  0
 x3   0     0     0  0
 x4   0     0     0  Inf

State types
     x1          x2         x3        x4
 Stationary  Stationary  Constant  Diffuse
```

`Mdl` is a `dssm` model object. `dssm` sets the initial state:

- Covariance matrix for the stationary states to the asymptotic covariance of the AR(2) model
- Variance for constant states to `0`
- Variance for diffuse states to `Inf`

You can display or modify properties of `Mdl` using dot notation. For example, display the initial state covariance matrix.

```
Mdl.Cov0
```

```
ans =

    0.2143    0.1607         0         0
    0.1607    0.2143         0         0
         0         0         0         0
         0         0         0       Inf
```

Reset the initial state means for the stationary states to their asymptotic values.

```
Mdl.Mean0(1:2) = 0.5/(1-0.2-0.6);
Mdl.Mean0
```

```
ans =

    2.5000
    2.5000
    1.0000
         0
```

Implicitly Create Time-Invariant State-Space Model

Use a parameter mapping function to create a time-invariant state-space model, where the state model is AR(1) model. The states are observed with bias, but without random error. Set the initial state mean and variance, and specify that the state is stationary.

Write a function that specifies how the parameters in `params` map to the state-space model matrices, the initial state values, and the type of state. Symbolically, the model is

$$\begin{aligned} x_t &= \phi x_{t-1} + \sigma u_t \\ y_t &= a x_t \end{aligned}.$$

```
% Copyright 2015 The MathWorks, Inc.

function [A,B,C,D,Mean0,Cov0,StateType] = timeInvariantParamMap(params)
% Time-invariant state-space model parameter mapping function example. This
% function maps the vector params to the state-space matrices (A, B, C, and
% D), the initial state value and the initial state variance (Mean0 and
% Cov0), and the type of state (StateType). The state model is AR(1)
% without observation error.
    varu1 = exp(params(2)); % Positive variance constraint
    A = params(1);
    B = sqrt(varu1);
    C = params(3);
    D = [];
    Mean0 = 0.5;
    Cov0 = 100;
    StateType = 0;
end
```

Save this code as a file named `timeInvariantParamMap.m` to a folder on your MATLAB® path.

Create the state-space model by passing the function `timeInvariantParamMap` as a function handle to `ssm`.

```
Mdl = ssm(@timeInvariantParamMap);
```

`ssm` implicitly creates the state-space model. Usually, you cannot verify implicitly defined state-space models.

Convert Standard to Diffuse State-Space Model

By default, `ssm` assigns a large scalar (`1e7`) to the initial state variance of all diffuse states in a standard state-space model. Using this specification, the software subsequently estimates, filters, and smooths a standard state-space model using the standard Kalman filter. A standard state-space model treatment is an approximation to results from an analysis that treats diffuse states using infinite variance. To implement

the diffuse Kalman filter instead, convert the standard state-space model to a diffuse state-space model. This conversion attributes infinite variance to all diffuse states.

Explicitly create a two-dimensional standard state-space model. Specify that the first state equation is $x_{1,t} = x_{1,t-1} + u_{1,t}$ and that the second state equation is $x_{2,t} = 0.2x_{2,t-1} + u_{2,t}$. Specify that the first observation equation is $y_{1,t} = x_{1,t} + \varepsilon_{1,t}$ and that the second observation equation is $y_{2,t} = x_{2,t} + \varepsilon_{2,t}$. Specify that the states are diffuse and nonstationary, respectively.

```
A = [1 0; 0 0.2];
B = [1 0; 0 1];
C = [1 0;0 1];
D = [1 0; 0 1];
StateType = [2 0];
SSMMdl = ssm(A,B,C,D,'StateType',StateType)
```

```
SSMMdl =


State-space model type: <a href="matlab: doc ssm">ssm</a>

State vector length: 2
Observation vector length: 2
State disturbance vector length: 2
Observation innovation vector length: 2
Sample size supported by model: Unlimited

State variables: x1, x2,...
State disturbances: u1, u2,...
Observation series: y1, y2,...
Observation innovations: e1, e2,...

State equations:
x1(t) = x1(t-1) + u1(t)
x2(t) = (0.20)x2(t-1) + u2(t)

Observation equations:
y1(t) = x1(t) + e1(t)
y2(t) = x2(t) + e2(t)

Initial state distribution:

Initial state means
```

```
 x1  x2
  0   0

Initial state covariance matrix
     x1        x2
 x1  1.00e+07   0
 x2   0        1.04

State types
    x1        x2
 Diffuse  Stationary
```

`SSMMdl` is an `ssm` model object. In some cases, `ssm` can detect the state type, but it is good practice to specify whether the state is stationary, diffuse, or the constant `1`. Because the model does not contain any unknown parameters, `ssm` infers the initial state distributions.

Convert `SSMMdl` to a diffuse state-space model.

```
Mdl = dssm(SSMMdl)
```

```
Mdl =


State-space model type: <a href="matlab: doc dssm">dssm</a>

State vector length: 2
Observation vector length: 2
State disturbance vector length: 2
Observation innovation vector length: 2
Sample size supported by model: Unlimited

State variables: x1, x2,...
State disturbances: u1, u2,...
Observation series: y1, y2,...
Observation innovations: e1, e2,...

State equations:
x1(t) = x1(t-1) + u1(t)
x2(t) = (0.20)x2(t-1) + u2(t)

Observation equations:
y1(t) = x1(t) + e1(t)
```

```
y2(t) = x2(t) + e2(t)

Initial state distribution:

Initial state means
 x1  x2
  0   0

Initial state covariance matrix
     x1   x2
 x1  Inf   0
 x2  0    1.04

State types
    x1        x2
 Diffuse  Stationary
```

`Mdl` is a `dssm` model object. The structures of `Mdl` and `SSMMdl` are equivalent, except that the initial state variance of the state in `Mdl` is `Inf` rather than `1e7`.

To see the difference between the two models, simulate 10 periods of data from a state-space model that is similar to `SSMMdl`. Set the initial state covariance matrix to I_2.

```
SimMdl = SSMMdl;
SimMdl.Cov0 = eye(2);
T = 10;
rng(1); % For reproducibility
y = simulate(SimMdl,T);
```

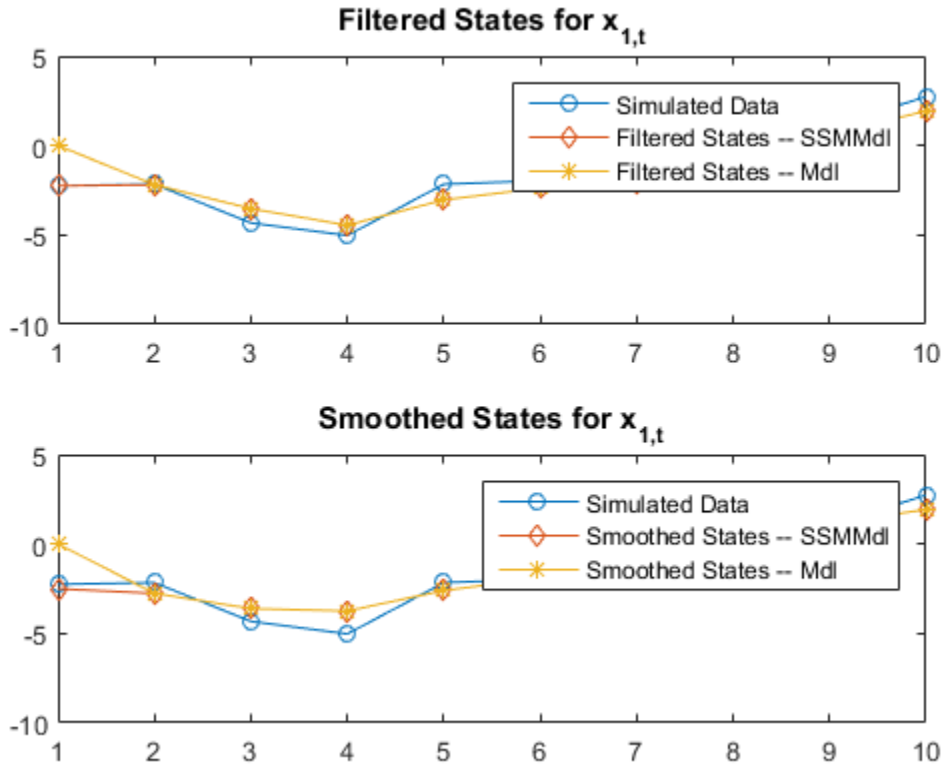
Obtain filtered and smoothed states from `Mdl` and `DSSMMdl` using the simulated data.

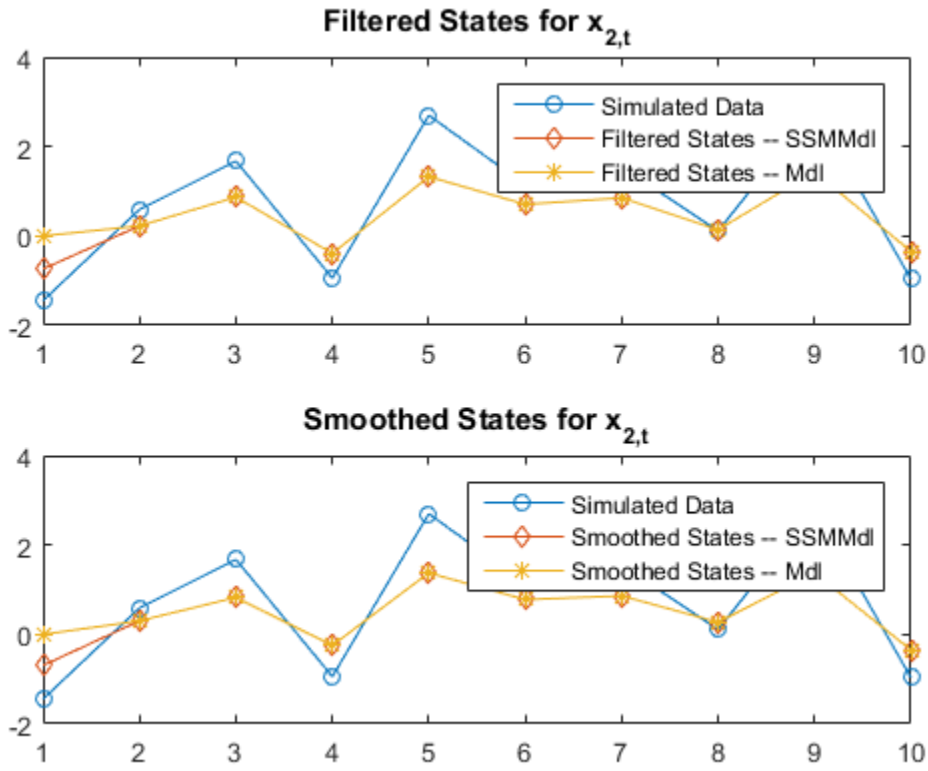
```
fSSMMdl = filter(SSMMdl,y);
fMdl = filter(Mdl,y);
sSSMMdl = smooth(SSMMdl,y);
sMdl = smooth(Mdl,y);
```

Plot the filtered and smoothed states.

```
figure;
subplot(2,1,1)
plot(1:T,y(:,1),'-o',1:T,fSSMMdl(:,1),'-d',1:T,fMdl(:,1),'-*');
title('Filtered States for x_{1,t}')
legend('Simulated Data','Filtered States -- SSMMdl','Filtered States -- Mdl');
```

```
subplot(2,1,2)
plot(1:T,y(:,1),'-o',1:T,sSSMMdl(:,1),'-d',1:T,sMdl(:,1),'-*');
title('Smoothed States for x_{1,t}')
legend('Simulated Data','Smoothed States -- SSMMdl','Smoothed States -- Mdl');

figure;
subplot(2,1,1)
plot(1:T,y(:,2),'-o',1:T,fSSMMdl(:,2),'-d',1:T,fMdl(:,2),'-*');
title('Filtered States for x_{2,t}')
legend('Simulated Data','Filtered States -- SSMMdl','Filtered States -- Mdl');
subplot(2,1,2)
plot(1:T,y(:,2),'-o',1:T,sSSMMdl(:,2),'-d',1:T,sMdl(:,2),'-*');
title('Smoothed States for x_{2,t}')
legend('Simulated Data','Smoothed States -- SSMMdl','Smoothed States -- Mdl');
```





Besides apparent transient behavior in the random walk, the filtered and smoothed states between the standard and diffuse state-space models appear nearly equivalent. The slight difference occurs because `filter` and `smooth` set all diffuse state estimates in the diffuse state-space model to 0 while they implement the diffuse Kalman filter. Once the covariance matrices of the smoothed states attain full rank, `filter` and `smooth` switch to using the standard Kalman filter. In this case, the switching time occurs after the first period.

- “Implicitly Create Time-Varying Diffuse State-Space Model” on page 8-35
- “Implicitly Create Diffuse State-Space Model Containing Regression Component” on page 8-30

Tip

- Specify `ParamMap` in a more general or complex setting, where, for example:
  - The initial state values are parameters.
  - In time-varying models, you want to use the same parameters for more than one period.
  - You want to impose parameter constraints.
- You can create a `dssm` model object that does not contain any diffuse states. However, subsequent computations, for example, filtering and parameter estimation, can be inefficient. If all states have stationary distributions or are the constant 1, then create an ssm model object instead.

Algorithms

- Default values for `Mean0` and `Cov0`:
  - If you explicitly specify the state-space model (that is, you provide the coefficient matrices `A`, `B`, `C`, and optionally `D`), then:
    - For stationary states, the software generates the initial value using the stationary distribution. If you provide all values in the coefficient matrices (that is, your model has no unknown parameters), then `dssm` generates the initial values. Otherwise, the software generates the initial values during estimation.
    - For states that are always the constant 1, `dssm` sets `Mean0` to 1 and `Cov0` to `0`.
    - For diffuse states, the software sets `Mean0` to 0 and `Cov0` to `Inf` by default.
  - If you implicitly specify the state-space model (that is, you provide the parameter vector to the coefficient-matrices-mapping function `ParamMap`), then the software generates the initial values during estimation.
- For static states that do not equal 1 throughout the sample, the software cannot assign a value to the degenerate, initial state distribution. Therefore, set static states to `2` using the name-value pair argument `StateType`. Subsequently, the software treats static states as nonstationary and assigns the static state a diffuse initial distribution.

- It is best practice to set `StateType` for each state. By default, the software generates `StateType`, but this behavior might not be accurate. For example, the software cannot distinguish between a constant 1 state and a static state.
- The software cannot infer `StateType` from data because the data theoretically comes from the observation equation. The realizations of the state equation are unobservable.
- `dssm` models do not store observed responses or predictor data. Supply the data wherever necessary using the appropriate input or name-value pair arguments.
- Suppose that you want to create a diffuse state-space model using a parameter-to-matrix mapping function with this signature:

  ```
  [A,B,C,D,Mean0,Cov0,StateType,DeflateY] = paramMap(params,Y,Z)
  ```
  and you specify the model using an anonymous function

  ```
  Mdl = dssm(@(params)paramMap(params,Y,Z))
  ```
  The observed responses `Y` and predictor data `Z` are not input arguments in the anonymous function. If `Y` and `Z` exist in the MATLAB Workspace before you create `Mdl`, then the software establishes a link to them. Otherwise, if you pass `Mdl` to estimate, the software throws an error.

  The link to the data established by the anonymous function overrides all other corresponding input argument values of `estimate`. This distinction is important particularly when conducting a rolling window analysis. For details, see "Rolling-Window Analysis of Time-Series Models" on page 8-168.

Alternatives

Create an ssm model object instead of a `dssm` model object when:

- The model does not contain any diffuse states.
- The diffuse states are correlated with each other or to other states.
- You want to implement the standard Kalman filter.

References

[1] Durbin J., and S. J. Koopman. *Time Series Analysis by State Space Methods*. 2nd ed. Oxford: Oxford University Press, 2012.

See Also

ssm

More About

- "What Are State-Space Models?" on page 8-3
- "Rolling-Window Analysis of Time-Series Models" on page 8-168

Introduced in R2015b

egarch

Create EGARCH conditional variance model object

Create an `egarch` model object to represent an exponential generalized autoregressive conditional heteroscedastic (EGARCH) model. The EGARCH(P,Q) conditional variance model includes P past log conditional variances composing the GARCH polynomial, and Q past standardized innovations composing the ARCH and leverage polynomials.

Use `egarch` to create a model with known or unknown coefficients, and then estimate any unknown coefficients from data using `estimate`. You can also simulate or forecast conditional variances from fully specified models using `simulate` or `forecast`, respectively.

For more information about `egarch` model objects, see Using egarch Objects.

Syntax

```
Mdl = egarch
Mdl = egarch(P,Q)
Mdl = egarch(Name,Value)
```

Description

`Mdl = egarch` creates a zero-degree conditional variance EGARCH model object.

`Mdl = egarch(P,Q)` creates an EGARCH model with GARCH polynomial degree `P`, and ARCH and leverage polynomials having degree `Q`.

`Mdl = egarch(Name,Value)` creates an EGARCH model with additional options specified by one or more `Name,Value` pair arguments. For example, you can specify a conditional variance model constant, the number of ARCH polynomial lags, and the innovation distribution.

Examples

Create Default EGARCH Model

Create a default `egarch` model object and specify its parameter values using dot notation.

Create an EGARCH(0,0) model.

```
Mdl = egarch
```

```
Mdl =

    EGARCH(0,0) Conditional Variance Model:
    ----------------------------------------
    Distribution: Name = 'Gaussian'
               P: 0
               Q: 0
        Constant: NaN
           GARCH: {}
            ARCH: {}
        Leverage: {}
```

`Mdl` is an `egarch` model. It contains an unknown constant, its offset is `0`, and the innovation distribution is `'Gaussian'`. The model does not have GARCH, ARCH, or leverage polynomials.

Specify two unknown ARCH and leverage coefficients for lags one and two using dot notation.

```
Mdl.ARCH = {NaN NaN};
Mdl.Leverage = {NaN NaN};
Mdl
```

```
Mdl =

    EGARCH(0,2) Conditional Variance Model:
    ----------------------------------------
    Distribution: Name = 'Gaussian'
               P: 0
               Q: 2
        Constant: NaN
           GARCH: {}
```

```
           ARCH: {NaN NaN} at Lags [1 2]
      Leverage: {NaN NaN} at Lags [1 2]
```

The `Q`, `ARCH`, and `Leverage` properties update to `2`, `{NaN NaN}`, `{NaN NaN}`, respectively. The two ARCH and leverage coefficients are associated with lags 1 and 2.

Create EGARCH Model Using Shorthand Syntax

Create an `egarch` model object using the shorthand notation `egarch(P,Q)`, where `P` is the degree of the GARCH polynomial and `Q` is the degree of the ARCH and leverage polynomial.

Create an EGARCH(3,2) model.

```
Mdl = egarch(3,2)
```

```
Mdl =

    EGARCH(3,2) Conditional Variance Model:
    ----------------------------------------
    Distribution: Name = 'Gaussian'
               P: 3
               Q: 2
        Constant: NaN
           GARCH: {NaN NaN NaN} at Lags [1 2 3]
            ARCH: {NaN NaN} at Lags [1 2]
        Leverage: {NaN NaN} at Lags [1 2]
```

`Mdl` is an `egarch` model object. All properties of `Mdl`, except `P`, `Q`, and `Distribution`, are `NaN` values. By default, the software:

- Includes a conditional variance model constant
- Excludes a conditional mean model offset (i.e., the offset is `0`)
- Includes all lag terms in the GARCH polynomial up to lag `P`
- Includes all lag terms in the ARCH and leverage polynomials up to lag `Q`

`Mdl` specifies only the functional form of an EGARCH model. Because it contains unknown parameter values, you can pass `Mdl` and time-series data to `estimate` to estimate the parameters.

Create EGARCH Model

Create an `egarch` model object using name-value pair arguments.

Specify an EGARCH(1,1) model. By default, the conditional mean model offset is zero. Specify that the offset is `NaN`. Include a leverage term.

```
Mdl = egarch('GARCHLags',1,'ARCHLags',1,'LeverageLags',1,'Offset',NaN)
```

```
Mdl =

    EGARCH(1,1) Conditional Variance Model with Offset:
    ---------------------------------------------------
    Distribution: Name = 'Gaussian'
               P: 1
               Q: 1
        Constant: NaN
           GARCH: {NaN} at Lags [1]
            ARCH: {NaN} at Lags [1]
        Leverage: {NaN} at Lags [1]
          Offset: NaN
```

`Mdl` is an `egarch` model object. The software sets all parameters to `NaN`, except `P`, `Q`, and `Distribution`.

Since `Mdl` contains `NaN` values, `Mdl` is appropriate for estimation only. Pass `Mdl` and time-series data to `estimate`. For a continuation of this example, see "Estimate EGARCH Model".

Create EGARCH Model with Known Coefficients

Create an EGARCH(1,1) model with mean offset,

$$y_t = 0.5 + \varepsilon_t,$$

where $\varepsilon_t = \sigma_t z_t$,

$$\sigma_t^2 = 0.0001 + 0.75 \log \sigma_{t-1}^2 + 0.1 \left(\frac{|\varepsilon_{t-1}|}{\sigma_{t-1}} - \sqrt{\frac{2}{\pi}} \right) - 0.3 \frac{\varepsilon_{t-1}}{\sigma_{t-1}} + 0.01 \frac{\varepsilon_{t-3}}{\sigma_{t-3}},$$

and z_t is an independent and identically distributed standard Gaussian process.

```
Mdl = egarch('Constant',0.0001,'GARCH',0.75,...
    'ARCH',0.1,'Offset',0.5,'Leverage',{-0.3 0 0.01})
```

```
Mdl =

    EGARCH(1,3) Conditional Variance Model with Offset:
    ---------------------------------------------------
    Distribution: Name = 'Gaussian'
               P: 1
               Q: 3
        Constant: 0.0001
           GARCH: {0.75} at Lags [1]
            ARCH: {0.1} at Lags [1]
        Leverage: {-0.3 0.01} at Lags [1 3]
          Offset: 0.5
```

`egarch` assigns default values to any properties you do not specify with name-value pair arguments. An alternative way to specify the leverage component is `'Leverage',{-0.3 0.01},'LeverageLags',[1 3]`.

- "Specify EGARCH Models Using egarch" on page 6-19
- "Modify Properties of Conditional Variance Models" on page 6-42
- "Specify the Conditional Variance Model Innovation Distribution" on page 6-48
- "Specify Conditional Mean and Variance Models" on page 5-79
- "Specify Conditional Variance Model For Exchange Rates" on page 6-53

Input Arguments

`P` — Number of past consecutive, logged conditional variance terms

nonnegative integer

Number of past consecutive, logged conditional variance terms to include in the GARCH polynomial, specified as a nonnegative integer. That is, `P` is the degree of the GARCH polynomial, where the polynomial includes each lag term from $t - 1$ to $t - P$.

You can specify `P` using the `egarch(P,Q)` shorthand syntax only. You cannot specify `P` in conjunction with `Name,Value` pair arguments.

If `P` > 0, then you must specify `Q` as a positive integer.

Example: `egarch(3,2)`

Data Types: `double`

`Q` — Number of past consecutive standardized innovation terms

nonnegative integer

Number of past consecutive standardized innovation terms to include in the ARCH and leverage polynomials, specified as a nonnegative integer. That is, `Q` is the degree of the ARCH and leverage polynomials, where each polynomial includes each lag term from $t - 1$ to $t - Q$. Also, `Q` specifies the minimum number of presample innovations the software requires to initiate the model.

You can specify this property when using the `egarch(P,Q)` shorthand syntax only. You cannot specify `Q` in conjunction with `Name,Value` pair arguments.

If `P` > 0, then you must specify `Q` as a positive integer.

Example: `egarch(3,2)`

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example:
`'Constant',0.5,'ARCHLags',2,'Distribution',struct('Name','t','DoF',5)`
specifies a conditional variance model constant of 0.5, two standardized innovation terms at lags 1 and 2 of the ARCH polynomial (but no leverage terms), and a t distribution with 5 degrees of freedom for the innovations.

`'Constant'` — Conditional variance model constant

`NaN` (default) | scalar

Conditional variance model constant, specified as the comma-separated pair consisting of `'Constant'` and a scalar.

Example: `'Constant',-0.5`

Data Types: `double`

`'GARCH'` — Coefficients corresponding to past logged conditional variance terms

cell vector of `NaN`s (default) | cell vector of scalars

Coefficients corresponding to the past logged conditional variance terms that compose the GARCH polynomial, specified as the comma-separated pair consisting of `'GARCH'` and a cell vector of scalars.

If you specify `GARCHLags`, then `GARCH` is an equivalent-length cell vector of coefficients associated with the lags in `GARCHLags`. Otherwise, `GARCH` is a P-element cell vector of coefficients corresponding to lags 1, 2,..., P.

The coefficients must compose a stationary GARCH polynomial. For details, see "EGARCH Model" on page 9-232.

By default, `GARCH` is a cell vector of `NaN`s of length P (the degree of the GARCH polynomial) or `numel(GARCHLags)`.

Example: `'GARCH',{0.1 0 0 0.02}`

Data Types: `cell`

`'ARCH'` — Coefficients corresponding to magnitude of past standardized innovation terms
cell vector of `NaN`s (default) | cell vector of scalars

Coefficients corresponding to the magnitude of the past standardized innovation terms that compose the ARCH polynomial, specified as the comma-separated pair consisting of `'ARCH'` and a cell vector of scalars.

If you do not specify `ARCHLags`, then `ARCH` is a cell vector of coefficients corresponding to lags 1 through the number of elements in `ARCH`.

If you specify `ARCHLags`, then `ARCH` is an equivalent-length cell vector of coefficients associated with the lags in `ARCHLags`.

By default, `ARCH` is a cell vector of `NaN`s with the same length as the ARCH polynomial degree or `numel(ARCHLags)`.

Example: `'ARCH',{0.5 0 0.2}`

Data Types: `cell`

`'Leverage'` — Coefficients corresponding to past standardized innovation terms
cell vector of `NaN`s (default) | cell vector of scalars

Coefficients corresponding to the past standardized innovation terms that compose the leverage polynomial, specified as the comma-separated pair consisting of `'Leverage'` and a cell vector of scalars.

If you specify `LeverageLags`, then `Leverage` is an equivalent-length cell vector of coefficients associated with the lags in `LeverageLags`. Otherwise, `Leverage` is a cell vector of coefficients corresponding to lags 1 through the number of elements in `Leverage`.

By default, `Leverage` is a cell vector of `NaN`s with the same length as the leverage polynomial degree or `numel(LeverageLags)`.

Example: `'Leverage',{-0.1 0 0 0.03}`

`'Offset'` — Innovation mean model offset

`0` (default) | scalar

Innovation mean model offset or additive constant, specified as the comma-separated pair consisting of `'Offset'` and a scalar.

Example: `'Offset',0.1`

Data Types: `double`

`'GARCHLags'` — Lags associated with GARCH polynomial coefficients

vector of positive integers

Lags associated with the GARCH polynomial coefficients, specified as the comma-separated pair consisting of `'GARCHLags'` and a vector of positive integers. The maximum value of `GARCHLags` determines *P*, the GARCH polynomial degree.

If you specify `GARCH`, then `GARCHLags` is an equivalent-length vector of positive integers specifying the lags of the corresponding coefficients in `GARCH`. Otherwise, `GARCHLags` indicates the lags of unknown coefficients in the GARCH polynomial.

By default, `GARCHLags` is a vector containing the integers 1 through *P*.

Example: `'GARCHLags',[1 2 4 3]`

Data Types: `double`

`'ARCHLags'` — Lags associated with ARCH polynomial coefficients

vector of positive integers

Lags associated with the ARCH polynomial coefficients, specified as the comma-separated pair consisting of `'ARCHLags'` and a vector of positive integers. The maximum value of `ARCHLags` determines the ARCH polynomial degree.

If you specify ARCH, then ARCHLags is an equivalent-length vector of positive integers specifying the lags of the corresponding coefficients in ARCH. Otherwise, ARCHLags indicates the lags of unknown coefficients in the ARCH polynomial.

By default, ARCHLags is a vector containing the integers 1 through the ARCH polynomial degree.

Example: 'ARCHLags',[3 1 2]

Data Types: double

'LeverageLags' — Lags associated with leverage polynomial coefficients
vector of positive integers

Lags associated with the leverage polynomial coefficients, specified as the comma-separated pair consisting of 'LeverageLags' and a vector of positive integers. The maximum value of LeverageLags determines the leverage polynomial degree.

If you specify Leverage, then LeverageLags is an equivalent-length vector of positive integers specifying the lags of the corresponding coefficients in LeverageLags. Otherwise, LeverageLags indicates the lags of unknown coefficients in the leverage polynomial.

By default, LeverageLags is a vector containing the integers 1 through the leverage polynomial degree.

Example: 'LeverageLags',1:4

Data Types: double

'Distribution' — Conditional probability distribution of innovation process
'Gaussian' (default) | string | structure array

Conditional probability distribution of the innovation process, specified as the comma-separated pair consisting of 'Distribution' and a string or a structure array.

This table contains the available distributions.

Distribution	String	Structure Array
Gaussian	'Gaussian'	struct('Name','Gaussian')
t	't' By default, DoF is NaN.	struct('Name','t','DoF',DoF) DoF > 2 or DoF = NaN

Example: `'Distribution',struct('Name','t','DoF',10)`

Data Types: `char` | `struct`

Notes:

- All `GARCH`, `ARCH` and `Leverage` coefficients are subject to a near-zero tolerance exclusion test. That is, the software:

  1. Creates lag operator polynomials for each of the `GARCH`, `ARCH` and `Leverage` components.
  2. Compares each coefficient to the default lag operator zero tolerance, `1e-12`.
  3. Includes a coefficient in the model if its magnitude is greater than `1e-12`, and excludes the coefficient otherwise. In other words, the software considers excluded coefficients to be sufficiently close to zero.

  For details, see LagOp.
- The lengths of `ARCH` and `Leverage` might differ. The difference can occur because the software defines the property `Q` as the largest lag associated with nonzero `ARCH` and `Leverage` coefficients, or `max(ARCHLags,LeverageLags)`. Typically, the number and corresponding lags of nonzero `ARCH` and `Leverage` coefficients are equivalent, but this is not a requirement.

Output Arguments

`Mdl` — EGARCH model

`egarch` model object

EGARCH model, returned as an `egarch` model object.

For the property descriptions of `Mdl`, see Conditional Variance Model Properties.

If `Mdl` contains unknown parameters (indicated by `NaN`s), then you can specify them using dot notation. Alternatively, you can pass `Mdl` and time series data to `estimate` to obtain estimates.

If `Mdl` is fully specified, then you can simulate or forecast conditional variances using `simulate` or `forecast`, respectively.

More About

EGARCH Model

An *EGARCH model* is an innovations process that addresses conditional heteroscedasticity. Specifically, the model posits that the current conditional variance is the sum of these linear processes:

- Past logged conditional variances (the GARCH component or polynomial)
- Magnitudes of past standardized innovations (the ARCH component or polynomial)
- Past standardized innovations (the leverage component or polynomial)

Consider the time series

$$y_t = \mu + \varepsilon_t,$$

where $\varepsilon_t = \sigma_t z_t$. The EGARCH($P$,$Q$) conditional variance process, σ_t^2, has the form

$$\log \sigma_t^2 = \kappa + \sum_{i=1}^{P} \gamma_i \log \sigma_{t-i}^2 + \sum_{j=1}^{Q} \alpha_j \left[\frac{|\varepsilon_{t-j}|}{\sigma_{t-j}} - E\left\{ \frac{|\varepsilon_{t-j}|}{\sigma_{t-j}} \right\} \right] + \sum_{j=1}^{Q} \xi_j \left(\frac{\varepsilon_{t-j}}{\sigma_{t-j}} \right).$$

The table shows how the variables correspond to the properties of the `garch` model object.

Variable	Description	Property
μ	Innovation mean model constant offset	`'Offset'`
$\kappa > 0$	Conditional variance model constant	`'Constant'`
γ_j	GARCH component coefficients	`'GARCH'`
α_j	ARCH component coefficients	`'ARCH'`
ξ_j	Leverage component coefficients	`'Leverage'`

Variable	Description	Property
z_t	Series of independent random variables with mean 0 and variance 1	`'Distribution'`

If z_t is Gaussian, then

$$E\left\{\frac{|\varepsilon_{t-j}|}{\sigma_{t-j}}\right\} = E\{|z_{t-j}|\} = \sqrt{\frac{2}{\pi}}.$$

If z_t is t distributed with $\nu > 2$ degrees of freedom, then

$$E\left\{\frac{|\varepsilon_{t-j}|}{\sigma_{t-j}}\right\} = E\{|z_{t-j}|\} = \sqrt{\frac{\nu-2}{\pi}}\frac{\Gamma\left(\frac{\nu-1}{2}\right)}{\Gamma\left(\frac{\nu}{2}\right)}.$$

To ensure a stationary EGARCH model, all roots of the GARCH lag operator polynomial, $(1-\gamma_1 L-\ldots-\gamma_P L^P)$, must lie outside of the unit circle.

The EGARCH model is unique from the GARCH and GJR models because it models the logarithm of the variance. By modeling the logarithm, positivity constraints on the model parameters are relaxed. However, forecasts of conditional variances from an EGARCH model are biased, because by Jensen's inequality,

$$E(\sigma_t^2) \geq \exp\{E(\log \sigma_t^2)\}.$$

EGARCH models are appropriate when positive and negative shocks of equal magnitude do not contribute equally to volatility [1].

Tips

- An EGARCH(1,1) specification is complex enough for most applications. Typically in these models, the GARCH and ARCH coefficients are positive, and the leverage coefficients are negative. If you get these signs, then large unanticipated downward shocks increase the variance. If you get signs opposite to those expected, you might

encounter difficulties inferring volatility sequences and forecasting. A negative ARCH coefficient is particularly problematic. In this case, an EGARCH model might not be the best choice for your application.

- "Conditional Variance Models" on page 6-2
- "EGARCH Model" on page 6-4

References

[1] Tsay, R. S. *Analysis of Financial Time Series*. 3rd ed. Hoboken, NJ: John Wiley & Sons, Inc., 2010.

See Also

`estimate` | `filter` | `forecast` | `infer` | `print` | `simulate`

Introduced in R2012a

Using egarch Objects

EGARCH conditional variance time series model

Description

An `egarch` model object specifies the functional form and stores the parameter values of a exponential generalized autoregressive conditional heteroscedastic (EGARCH) model. "Definitions" on page 9- attempt to address volatility clustering in an innovations process. Volatility clustering occurs when an innovations process does not exhibit significant autocorrelation, but the variance of the process changes with time. EGARCH models are appropriate when positive and negative shocks of equal magnitude might not contribute equally to volatility [1].

The EGARCH(P,Q) conditional variance model includes:

- P past log conditional variances that compose the GARCH component polynomial
- Q past standardized innovations that compose the ARCH and leverage component polynomials

To create an `egarch` model object, use `egarch`. Specify only the GARCH and ARCH (and leverage) polynomial degrees P and Q, respectively, using the shorthand syntax `egarch(P,Q)`. Then, pass the model and time series data to `estimate` to fit the model to the data. Or, specify the values of some parameters, and then estimate others.

Use a completely specified model (i.e., all parameter values of the model are known) to:

- Simulate conditional variances or responses using `simulate`
- Forecast conditional variances using `forecast`

Examples

Create EGARCH Model

Create an `egarch` model object using name-value pair arguments.

Specify an EGARCH(1,1) model. By default, the conditional mean model offset is zero. Specify that the offset is `NaN`. Include a leverage term.

```
Mdl = egarch('GARCHLags',1,'ARCHLags',1,'LeverageLags',1,'Offset',NaN)
```

```
Mdl =

    EGARCH(1,1) Conditional Variance Model with Offset:
    ---------------------------------------------------
    Distribution: Name = 'Gaussian'
               P: 1
               Q: 1
        Constant: NaN
           GARCH: {NaN} at Lags [1]
            ARCH: {NaN} at Lags [1]
        Leverage: {NaN} at Lags [1]
          Offset: NaN
```

`Mdl` is an `egarch` model object. The software sets all parameters to `NaN`, except `P`, `Q`, and `Distribution`.

Since `Mdl` contains `NaN` values, `Mdl` is appropriate for estimation only. Pass `Mdl` and time-series data to `estimate`. For a continuation of this example, see "Estimate EGARCH Model".

Create EGARCH Model Using Shorthand Syntax

Create an `egarch` model object using the shorthand notation `egarch(P,Q)`, where `P` is the degree of the GARCH polynomial and `Q` is the degree of the ARCH and leverage polynomial.

Create an EGARCH(3,2) model.

```
Mdl = egarch(3,2)
```

```
Mdl =

    EGARCH(3,2) Conditional Variance Model:
    ---------------------------------------
    Distribution: Name = 'Gaussian'
               P: 3
               Q: 2
        Constant: NaN
           GARCH: {NaN NaN NaN} at Lags [1 2 3]
            ARCH: {NaN NaN} at Lags [1 2]
```

```
        Leverage: {NaN NaN} at Lags [1 2]
```

`Mdl` is an `egarch` model object. All properties of `Mdl`, except `P`, `Q`, and `Distribution`, are `NaN` values. By default, the software:

- Includes a conditional variance model constant
- Excludes a conditional mean model offset (i.e., the offset is `0`)
- Includes all lag terms in the GARCH polynomial up to lag `P`
- Includes all lag terms in the ARCH and leverage polynomials up to lag `Q`

`Mdl` specifies only the functional form of an EGARCH model. Because it contains unknown parameter values, you can pass `Mdl` and time-series data to `estimate` to estimate the parameters.

Access EGARCH Model Properties

Access the properties of a created `egarch` model object using dot notation.

Create an `egarch` model object.

```
Mdl = egarch(3,2)
```

```
Mdl =

    EGARCH(3,2) Conditional Variance Model:
    ----------------------------------------
    Distribution: Name = 'Gaussian'
               P: 3
               Q: 2
        Constant: NaN
           GARCH: {NaN NaN NaN} at Lags [1 2 3]
            ARCH: {NaN NaN} at Lags [1 2]
        Leverage: {NaN NaN} at Lags [1 2]
```

Remove the second GARCH term from the model. That is, specify that the GARCH coefficient of the second lagged conditional variance is `0`.

```
Mdl.GARCH{2} = 0
```

```
Mdl =
```

```
    EGARCH(3,2) Conditional Variance Model:
    ----------------------------------------
    Distribution: Name = 'Gaussian'
               P: 3
               Q: 2
        Constant: NaN
           GARCH: {NaN NaN} at Lags [1 3]
            ARCH: {NaN NaN} at Lags [1 2]
        Leverage: {NaN NaN} at Lags [1 2]
```

The GARCH polynomial has two unknown parameters corresponding to lags 1 and 3.

Display the distribution of the disturbances.

```
Mdl.Distribution
```

```
ans =

    Name: 'Gaussian'
```

The disturbances are Gaussian with mean 0 and variance 1.

Specify that the underlying disturbances have a t distribution with five degrees of freedom.

```
Mdl.Distribution = struct('Name','t','DoF',5)
```

```
Mdl =

    EGARCH(3,2) Conditional Variance Model:
    ----------------------------------------
    Distribution: Name = 't', DoF = 5
               P: 3
               Q: 2
        Constant: NaN
           GARCH: {NaN NaN} at Lags [1 3]
            ARCH: {NaN NaN} at Lags [1 2]
        Leverage: {NaN NaN} at Lags [1 2]
```

Specify that the ARCH coefficients are 0.2 for the first lag and 0.1 for the second lag.

```
Mdl.ARCH = {0.2 0.1}


Mdl =

    EGARCH(3,2) Conditional Variance Model:
    ----------------------------------------
    Distribution: Name = 't', DoF = 5
               P: 3
               Q: 2
        Constant: NaN
           GARCH: {NaN NaN} at Lags [1 3]
            ARCH: {0.2 0.1} at Lags [1 2]
        Leverage: {NaN NaN} at Lags [1 2]
```

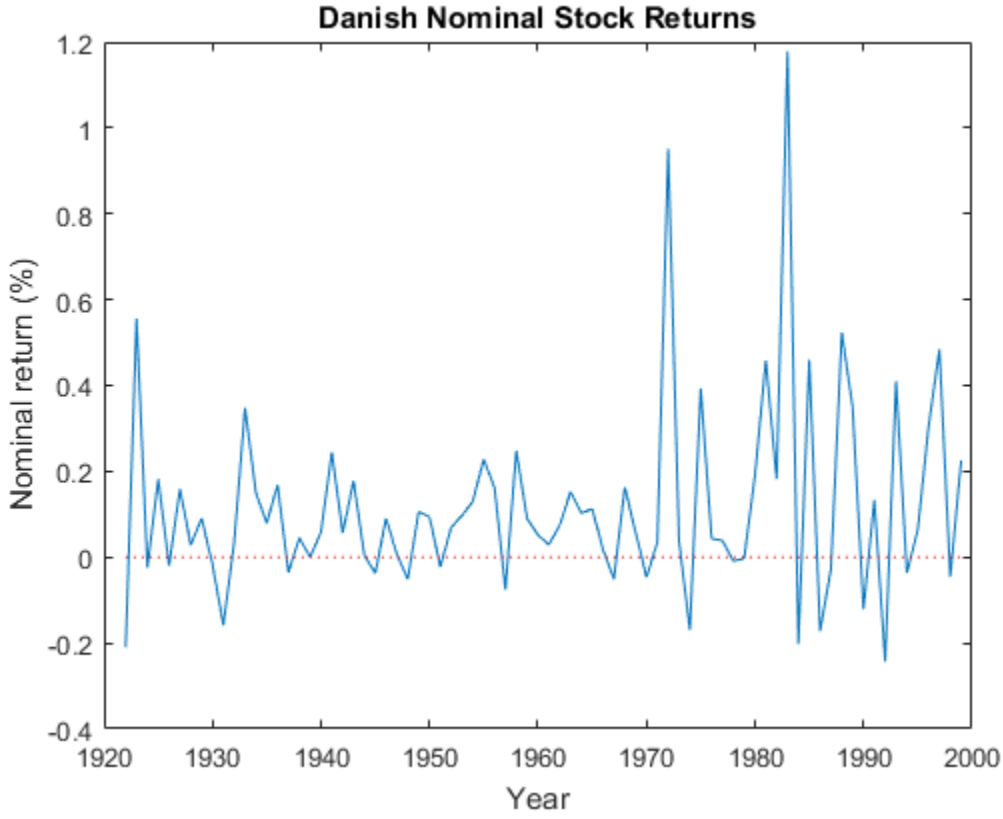
To estimate the remaining parameters, you can pass `Mdl` and your data to estimate and use the specified parameters as equality constraints. Or, you can specify the rest of the parameter values, and then simulate or forecast conditional variances from the GARCH model by passing the fully specified model to `simulate` or `forecast`, respectively.

Estimate EGARCH Model

Fit an EGARCH model to an annual time series of Danish nominal stock returns from 1922-1999. The example follows from “Create EGARCH Model”.

Load the `Data_Danish` data set. Plot the nominal returns (`RN`).

```
load Data_Danish;
nr = DataTable.RN;

figure;
plot(dates,nr);
hold on;
plot([dates(1) dates(end)],[0 0],'r:'); % Plot y = 0
hold off;
title('Danish Nominal Stock Returns');
ylabel('Nominal return (%)');
xlabel('Year');
```



The nominal return series seems to have a nonzero conditional mean offset and seems to exhibit volatility clustering. That is, the variability is smaller for earlier years than it is for later years. For this example, assume that an EGARCH(1,1) model is appropriate for this series.

Create an EGARCH(1,1) model. The conditional mean offset is zero by default. To estimate the offset, specify that it is `NaN`. Include a leverage lag.

```
Mdl = egarch('GARCHLags',1,'ARCHLags',1,'LeverageLags',1,'Offset',NaN);
```

Fit the EGARCH(1,1) model to the data.

```
EstMdl = estimate(Mdl,nr);
```

```
    EGARCH(1,1) Conditional Variance Model:
    ----------------------------------------
    Conditional Probability Distribution: Gaussian

                                Standard          t
     Parameter       Value        Error       Statistic
    -----------   -----------   ------------   -----------
     Constant       -0.62723      0.744007      -0.843043
     GARCH{1}       0.774189       0.23628        3.27658
      ARCH{1}       0.386361      0.373606        1.03414
  Leverage{1}    -0.00249918       0.19222     -0.0130016
      Offset         0.10325     0.0377269        2.73676
```

`EstMdl` is a fully specified `egarch` model object. That is, it does not contain `NaN` values. You can assess the adequacy of the model by generating residuals using `infer`, and then analyzing them.

To simulate conditional variances or responses, pass `EstMdl` to `simulate`. See "Simulate EGARCH Model Observations and Conditional Variances".

To forecast innovations, pass `EstMdl` to `forecast`. See "Forecast EGARCH Model Conditional Variances".

Simulate EGARCH Model Observations and Conditional Variances

Simulate conditional variance or response paths from a fully specified `egarch` model object. That is, simulate from an estimated `egarch` model or a known `egarch` model in which you specify all parameter values. This example follows from "Estimate EGARCH Model".

Load the `Data_Danish` data set.

```
load Data_Danish;
rn = DataTable.RN;
```

Create an EGARCH(1,1) model with an unknown conditional mean offset. Fit the model to the annual, nominal return series. Include a leverage term.

```
Mdl = egarch('GARCHLags',1,'ARCHLags',1,'LeverageLags',1,'Offset',NaN);
EstMdl = estimate(Mdl,rn);
```

```
    EGARCH(1,1) Conditional Variance Model:
```

```
    --------------------------------------
    Conditional Probability Distribution: Gaussian

                               Standard          t
     Parameter       Value       Error       Statistic
    -----------   -----------   ------------   -----------
     Constant       -0.62723      0.744007      -0.843043
     GARCH{1}       0.774189       0.23628        3.27658
      ARCH{1}       0.386361      0.373606        1.03414
  Leverage{1}    -0.00249918       0.19222     -0.0130016
      Offset        0.10325     0.0377269        2.73676
```

Simulate 100 paths of conditional variances and responses from the estimated EGARCH model.

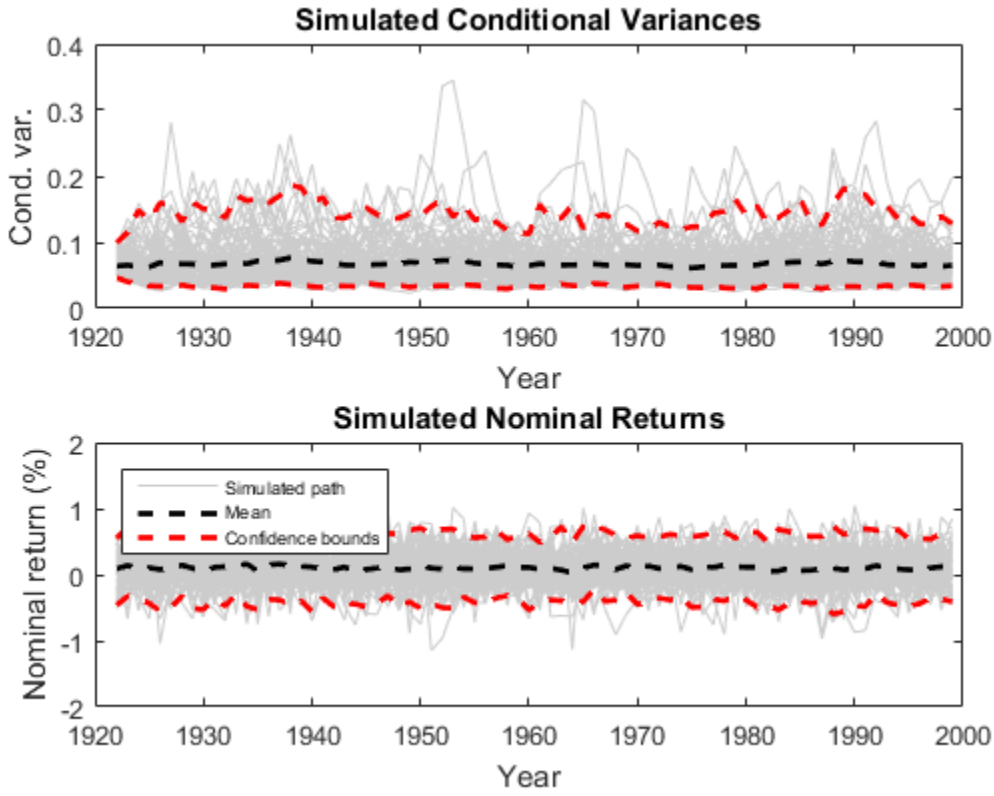
```
numObs = numel(rn); % Sample size (T)
numPaths = 100;     % Number of paths to simulate
rng(1);             % For reproducibility
[VSim,YSim] = simulate(EstMdl,numObs,'NumPaths',numPaths);
```

`VSim` and `YSim` are `T`-by-`numPaths` matrices. Rows correspond to a sample period, and columns correspond to a simulated path.

Plot the average and the 97.5% and 2.5% percentiles of the simulate paths. Compare the simulation statistics to the original data.

```
VSimBar = mean(VSim,2);
VSimCI = quantile(VSim,[0.025 0.975],2);
YSimBar = mean(YSim,2);
YSimCI = quantile(YSim,[0.025 0.975],2);

figure;
subplot(2,1,1);
h1 = plot(dates,VSim,'Color',0.8*ones(1,3));
hold on;
h2 = plot(dates,VSimBar,'k--','LineWidth',2);
h3 = plot(dates,VSimCI,'r--','LineWidth',2);
hold off;
title('Simulated Conditional Variances');
ylabel('Cond. var.');
xlabel('Year');

subplot(2,1,2);
h1 = plot(dates,YSim,'Color',0.8*ones(1,3));
hold on;
```

```
h2 = plot(dates,YSimBar,'k--','LineWidth',2);
h3 = plot(dates,YSimCI,'r--','LineWidth',2);
hold off;
title('Simulated Nominal Returns');
ylabel('Nominal return (%)');
xlabel('Year');
legend([h1(1) h2 h3(1)],{'Simulated path' 'Mean' 'Confidence bounds'},...
    'FontSize',7,'Location','NorthWest');
```



Forecast EGARCH Model Conditional Variances

Forecast conditional variances from a fully specified `egarch` model object. That is, forecast from an estimated `egarch` model or a known `egarch` model in which you specify all parameter values. The example follows from “Estimate EGARCH Model”.

Load the `Data_Danish` data set.

```
load Data_Danish;
nr = DataTable.RN;
```

Create an EGARCH(1,1) model with an unknown conditional mean offset and include a leverage term. Fit the model to the annual nominal return series.

```
Mdl = egarch('GARCHLags',1,'ARCHLags',1,'LeverageLags',1,'Offset',NaN);
EstMdl = estimate(Mdl,nr);
```

```
    EGARCH(1,1) Conditional Variance Model:
    ----------------------------------------
    Conditional Probability Distribution: Gaussian

                                Standard          t
     Parameter       Value        Error       Statistic
    -----------   -----------   ------------   -----------
     Constant       -0.62723      0.744007      -0.843043
     GARCH{1}       0.774189       0.23628        3.27658
      ARCH{1}       0.386361      0.373606        1.03414
  Leverage{1}    -0.00249918       0.19222     -0.0130016
      Offset         0.10325     0.0377269        2.73676
```

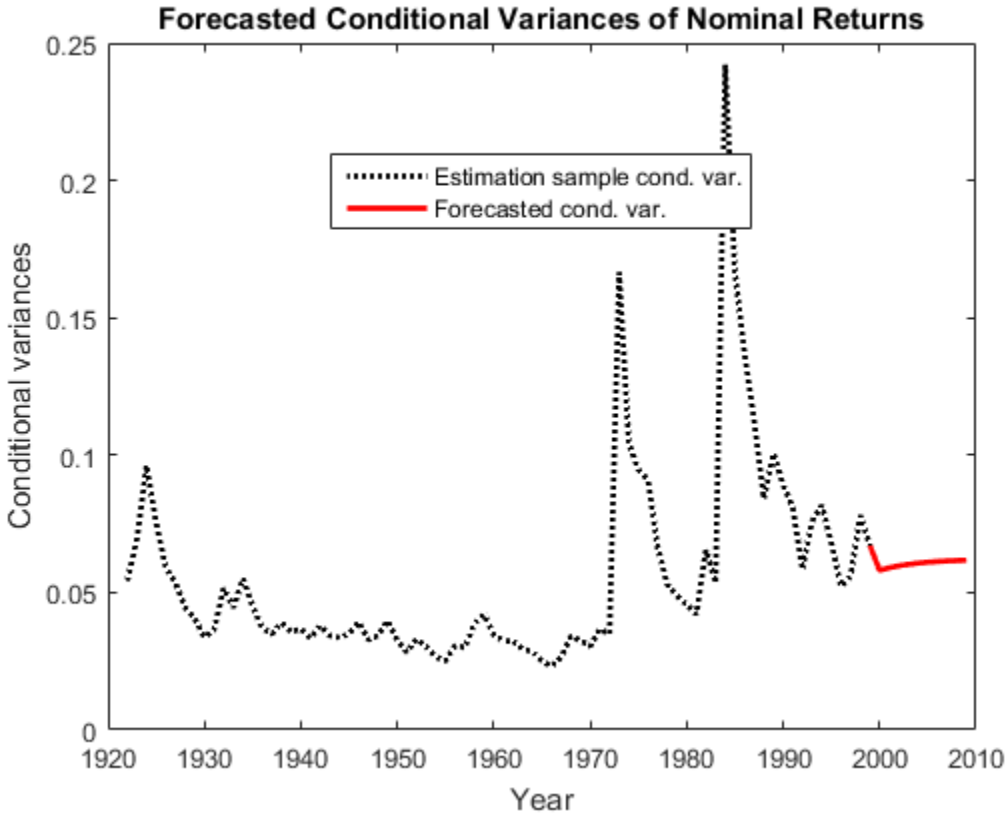
Forecast the conditional variance of the nominal return series 10 years into the future using the estimated EGARCH model. Specify the entire returns series as presample observations. The software infers presample conditional variances using the presample observations and the model.

```
numPeriods = 10;
vF = forecast(EstMdl,numPeriods,'Y0',nr);
```

Plot the forecasted conditional variances of the nominal returns. Compare the forecasts to the observed conditional variances.

```
v = infer(EstMdl,nr);

figure;
plot(dates,v,'k:','LineWidth',2);
hold on;
plot(dates(end):dates(end) + 10,[v(end);vF],'r','LineWidth',2);
title('Forecasted Conditional Variances of Nominal Returns');
ylabel('Conditional variances');
```

```
xlabel('Year');
legend({'Estimation sample cond. var.','Forecasted cond. var.'},...
    'Location','Best');
```



- "Specify EGARCH Models Using egarch" on page 6-19
- "Modify Properties of Conditional Variance Models" on page 6-42
- "Specify Conditional Mean and Variance Models" on page 5-79
- "Infer Conditional Variances and Residuals" on page 6-77
- "Compare Conditional Variance Models Using Information Criteria" on page 6-87
- "Assess EGARCH Forecast Bias Using Simulations" on page 6-104
- "Forecast a Conditional Variance Model" on page 6-126

Properties

Conditional Variance Model Properties	Specify conditional variance model functional form and parameter values

Object Functions

estimate	Fit conditional variance model to data
filter	Filter disturbances through conditional variance model
forecast	Forecast conditional variances from conditional variance models
infer	Infer conditional variances of conditional variance models
print	Display parameter estimation results for conditional variance models
simulate	Monte Carlo simulation of conditional variance models

Create Object

Create `egarch` models using `egarch`.

You can specify an `egarch` model as part of a composition of conditional mean and variance models. For details, see arima.

See Also

arima | `garch` | `gjr`

More About

- "Conditional Variance Models" on page 6-2
- "EGARCH Model" on page 6-4

Introduced in R2012a

egcitest

Engle-Granger cointegration test

Syntax

```
[h,pValue,stat,cValue,reg1,reg2] = egcitest(Y)
[h,pValue,stat,cValue,reg1,reg2] = egcitest(Y,Name,Value)
```

Description

Engle-Granger tests assess the null hypothesis of no cointegration among the time series in `Y`. The test regresses `Y(:,1)` on `Y(:,2:end)`, then tests the residuals for a unit root.

`[h,pValue,stat,cValue,reg1,reg2] = egcitest(Y)` performs the Engle-Granger test on a data matrix `Y`.

`[h,pValue,stat,cValue,reg1,reg2] = egcitest(Y,Name,Value)` performs the Engle-Granger test on a data matrix `Y` with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

`Y`

numObs-by-*numDims* matrix representing *numObs* observations of a *numDims*-dimensional time series $y(t)$, with the last observation the most recent. Y cannot have more than 12 columns. Observations containing `NaN` values are removed.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

'creg'

String or cell vector of strings indicating the form of the cointegrating regression, where y_1 = `Y(:,1)` is regressed on Y_2 = `Y(:,2:end)` and optional deterministic terms in X:
$y_1 = Xa + Y_2b + \varepsilon$
Values are

- `nc`—no constant or trend in X
- `c`—constant but no trend in X
- `ct` —constant and linear trend in X
- `ctt` —constant, linear trend, and quadratic trend in X

Default: `c`

'cvec'

Vector or cell vector of vectors containing coefficients $[a;b]$ to be held fixed in the cointegrating regression. The length of a is 0, 1, 2 or 3, depending on `creg`, with coefficient order: constant, linear trend, quadratic trend. The length of b is $numDims - 1$. It is assumed that the coefficient of y_1 = `Y(:,1)` has been normalized to 1. `NaN` values indicate coefficients to be estimated. If `cvec` is completely specified (no `NaN` values), no cointegrating regression is performed.

Default: Completely unspecified cointegrating vector (all NaN values).

'rreg'

String or cell vector of strings indicating the form of the residual regression. Values are `ADF`, for an augmented Dickey-Fuller test of residuals from the cointegrating regression, or `PP`, for a Phillips-Perron test. Test statistics are computed by calling `adftest` and `pptest` with the model parameter set to `AR`, assuming data have been demeaned or detrended, as necessary, in the cointegrating regression.

Default: `ADF`

'lags'

Scalar or vector of nonnegative integers indicating the number of lags used in the residual regression. The meaning of the parameter depends on the value of `rreg` (see the documentation for the `lags` parameter in `adftest` and `pptest`).

Default: 0

`'test'`

String or cell vector of strings indicating the type of test statistic computed from the residual regression. Values are `t1` (a "τ test") or `t2` (a "z test"). The meaning of the parameter depends on the value of `rreg` (see the documentation for the test parameter in `adftest` and `pptest`).

Default: `t1`

`'alpha'`

Scalar or vector of nominal significance levels for the tests. Values must be between 0.001 and 0.999.

Default: 0.05

Single-element parameter values are expanded to the length of any vector value (the number of tests). Vector values must have equal length. If any value is a row vector, all outputs are row vectors.

Output Arguments

`h`

Vector of Boolean decisions for the tests, with length equal to the number of tests. Values of `h` equal to `1` (`true`) indicate rejection of the null in favor of the alternative of cointegration. Values of `h` equal to `0` (`false`) indicate a failure to reject the null.

`pValue`

Vector of p-values of the test statistics, with length equal to the number of tests. p-values are left-tail probabilities.

`stat`

Vector of test statistics, with length equal to the number of tests. The statistic depends on the `rreg` and `test` values (see the documentation for `adftest` and `pptest`).

cValue

Vector of critical values for the tests, with length equal to the number of tests. Values are for left-tail probabilities. Since residuals are estimated rather than observed, critical values are different from those used in `adftest` or `pptest` (unless the cointegrating vector is completely specified by `cvec`). `egcitest` loads tables of critical values from the file `Data_EGCITest.mat`, then linearly interpolates test values from the tables. Critical values in the tables were computed using methods described in [3].

reg1

Structure of regression statistics from the cointegrating regression.

reg2

Structure of regression statistics from the residual regression.

The number of records in `reg1` and `reg2` equals the number of tests. Each record has the following fields:

`num`	Length of the regression response *y*, with `NaN`s removed
`size`	Effective sample size, adjusted for lags, difference*
`names`	Regression coefficient names
`coeff`	Estimated coefficient values
`se`	Estimated coefficient standard errors
`Cov`	Estimated coefficient covariance matrix
`tStats`	*t* statistics of coefficients and *p*-values
`FStat`	*F* statistic and *p*-value
`yMu`	Mean of *y*, adjusted for lags, difference*
`ySigma`	Standard deviation of *y*, adjusted for lags, difference*
`yHat`	Fitted values of *y*, adjusted for lags, difference*
`res`	Regression residuals
`DWStat`	Durbin-Watson statistic
`SSR`	Regression sum of squares
`SSE`	Error sum of squares

`SST`	Total sum of squares
`MSE`	Mean squared error
`RMSE`	Standard error of the regression
`RSq`	R^2 statistic
`aRSq`	Adjusted R^2 statistic
`LL`	Loglikelihood of data under Gaussian innovations
`AIC`	Akaike information criterion
`BIC`	Bayesian (Schwarz) information criterion
`HQC`	Hannan-Quinn information criterion

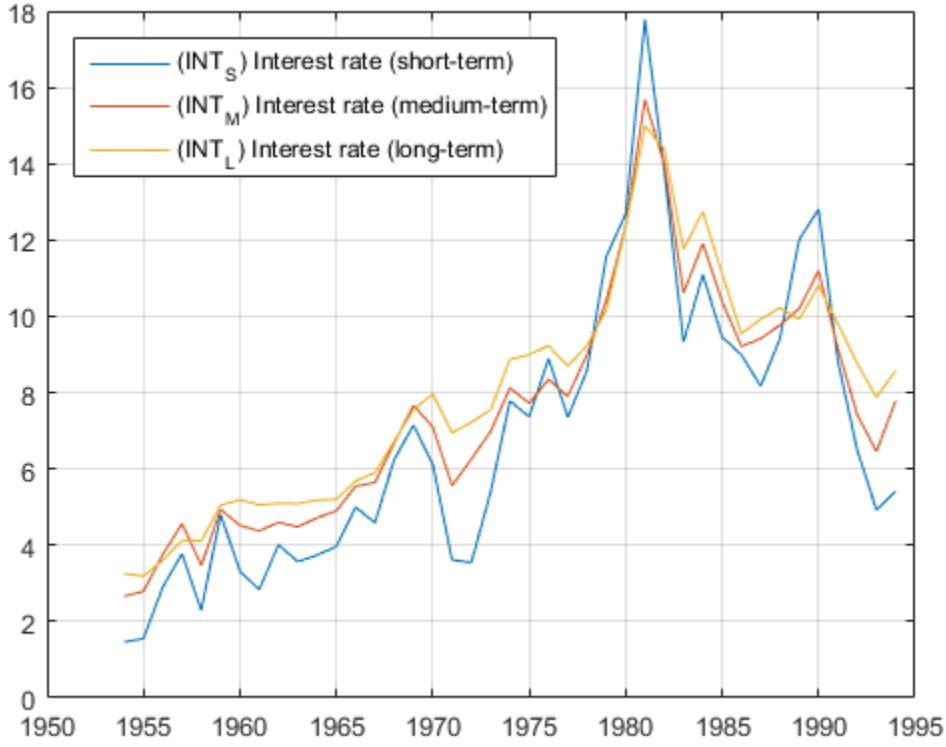
*Lagging and differencing a time series reduces the sample size. Absent any presample values, if $y(t)$ is defined for $t = 1{:}N$, then the lagged series $y(t-k)$ is defined for $t = k+1{:}N$. Differencing reduces the time base to $k+2{:}N$. With p lagged differences, the common time base is $p+2{:}N$ and the effective sample size is $N-(p+1)$.

Examples

Test Multiple Time Series for Cointegration Using egcitest

Load data on term structure of interest rates in Canada:

```
load Data_Canada
Y = Data(:,3:end);
names = series(3:end);
plot(dates,Y)
legend(names,'location','NW')
grid on
```



Test for cointegration (and reproduce row 1 of Table II in [3]):

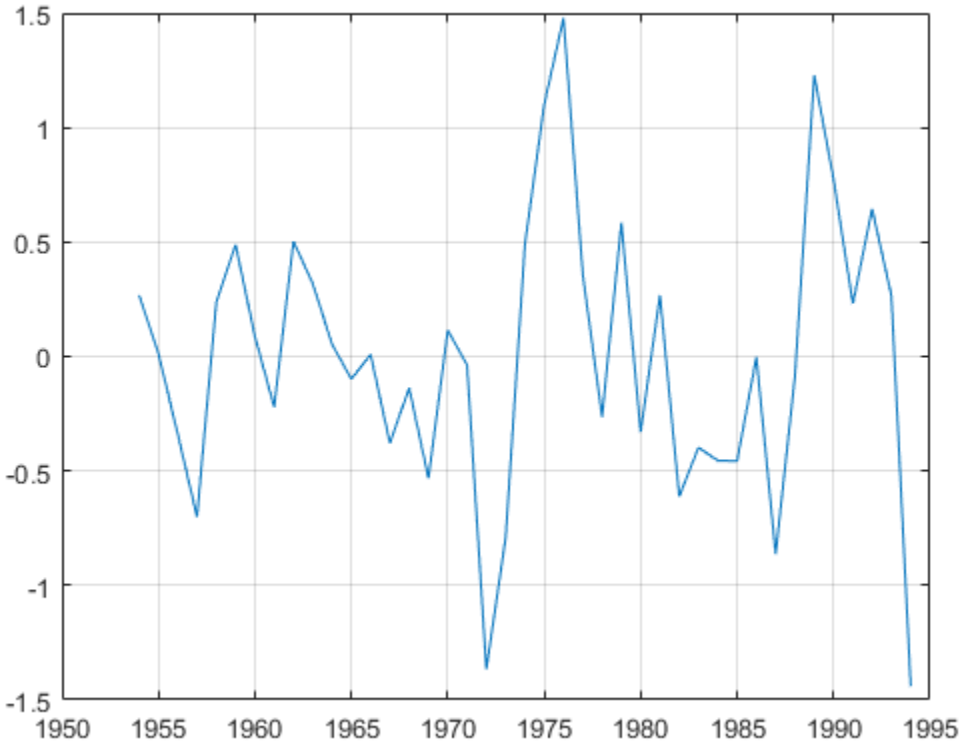
```
[h,pValue,stat,cValue,reg] = egcitest(Y,'test',...
    {'t1','t2'});
h,pValue
```

```
h =

     0     1


pValue =
```

```
    0.0526    0.0202
```

Plot the estimated cointegrating relation $y_1 - Y_2 b - Xa$:

```
a = reg(2).coeff(1);
b = reg(2).coeff(2:3);
plot(dates,Y*[1;-b]-a)
grid on
```



More About

Algorithms

A suitable value for `lags` must be determined in order to draw valid inferences from the test. See notes on the `lags` parameter in the documentation for `adftest` and `pptest`.

Samples with less than ~20 to 40 observations (depending on the dimension of the data) can yield unreliable critical values, and so unreliable inferences. See [3].

If cointegration is inferred, residuals from the `reg1` output can be used as data for the error-correction term in a VEC representation of $y(t)$. See [1]. Estimation of autoregressive model components can then be performed with `vgxvarx`, treating the residual series as exogenous.

- "Cointegration and Error Correction Analysis" on page 7-108

References

[1] Engle, R. F. and C. W. J. Granger. "Co-Integration and Error-Correction: Representation, Estimation, and Testing." *Econometrica*. v. 55, 1987, pp. 251–276.

[2] Hamilton, J. D. *Time Series Analysis*. Princeton, NJ: Princeton University Press, 1994.

[3] MacKinnon, J. G. "Numerical Distribution Functions for Unit Root and Cointegration Tests." *Journal of Applied Econometrics*. v. 11, 1996, pp. 601–618.

See Also

`jcitest` | `adftest` | `pptest` | `vec2var`

Introduced in R2011a

estimate

Fit conditional variance model to data

Syntax

```
EstMdl = estimate(Mdl,y)
EstMdl = estimate(Mdl,y,Name,Value)
[EstMdl,EstParamCov,logL,info] = estimate( ___ )
```

Description

`EstMdl = estimate(Mdl,y)` estimates the unknown parameters of the conditional variance model object `Mdl` with the observed univariate time series `y`, using maximum likelihood. `EstMdl` is a fully specified conditional variance model object that stores the results. It is the same model type as `Mdl` (see `garch`, `egarch`, and `gjr`).

`EstMdl = estimate(Mdl,y,Name,Value)` estimates the conditional variance model with additional options specified by one or more `Name,Value` pair arguments. For example, you can specify to display iterative optimization information or presample innovations.

`[EstMdl,EstParamCov,logL,info] = estimate( ___ )` additionally returns:

- `EstParamCov`, the variance-covariance matrix associated with estimated parameters.
- `logL`, the optimized loglikelihood objective function.
- `info`, a data structure of summary information using any of the input arguments in the previous syntaxes.

Examples

Estimate GARCH Model Parameters Without Initial Values

Fit a GARCH(1,1) model to simulated data.

Simulate 500 data points from the GARCH(1,1) model

$$y_t = \varepsilon_t,$$

where $\varepsilon_t = \sigma_t z_t$ and

$$\sigma_t^2 = 0.0001 + 0.5\sigma_{t-1}^2 + 0.2\varepsilon_{t-1}^2.$$

Use the default Gaussian innovation distribution for z_t.

```
Mdl = garch('Constant',0.0001,'GARCH',0.5,...
    'ARCH',0.2);
rng default; % For reproducibility
[v,y] = simulate(Mdl,500);
```

The output `v` contains simulated conditional variances. `y` is a column vector of simulated responses (innovations).

Specify a GARCH(1,1) model with unknown coefficients, and fit it to the series `y`.

```
ToEstMdl = garch(1,1);
EstMdl = estimate(ToEstMdl,y)
```

```
    GARCH(1,1) Conditional Variance Model:
    ----------------------------------------
    Conditional Probability Distribution: Gaussian

                                Standard          t
     Parameter       Value          Error       Statistic
    -----------   -----------   ------------   -----------
     Constant    9.89133e-05   3.07271e-05        3.21909
     GARCH{1}       0.453925      0.111928        4.05552
      ARCH{1}       0.263743     0.0569322        4.63258

EstMdl =

    GARCH(1,1) Conditional Variance Model:
    ----------------------------------------
    Distribution: Name = 'Gaussian'
               P: 1
               Q: 1
        Constant: 9.89133e-05
```

```
          GARCH: {0.453925} at Lags [1]
           ARCH: {0.263743} at Lags [1]
```

The result is a new `garch` model called `EstMdl`. The parameter estimates in `EstMdl` resemble the parameter values that generated the simulated data.

Estimate EGARCH Model Parameters Without Initial Values

Fit an EGARCH(1,1) model to simulated data.

Simulate 500 data points from an EGARCH(1,1) model

$$y_t = \varepsilon_t,$$

where $\varepsilon_t = \sigma_t z_t$, and

$$\log \sigma_t^2 = 0.001 + 0.7 \log \sigma_{t-1}^2 + 0.5 \left[\frac{|\varepsilon_{t-1}|}{\sigma_{t-1}} - \sqrt{\frac{2}{\pi}} \right] - 0.3 \left(\frac{\varepsilon_{t-1}}{\sigma_{t-1}} \right)$$

(the distribution of z_t is Gaussian).

```
Mdl = egarch('Constant',0.001,'GARCH',0.7,...
               'ARCH',0.5,'Leverage',-0.3);

rng default % For reproducibility
[v,y] = simulate(Mdl,500);
```

The output `v` contains simulated conditional variances. `y` is a column vector of simulated responses (innovations).

Specify an EGARCH(1,1) model with unknown coefficients, and fit it to the series `y`.

```
ToEstMdl = egarch(1,1);
EstMdl = estimate(ToEstMdl,y)
```

```
    EGARCH(1,1) Conditional Variance Model:
    ----------------------------------------
    Conditional Probability Distribution: Gaussian

                                 Standard          t
     Parameter       Value          Error       Statistic
```

```
    -----------   -----------   ------------   -----------
     Constant   -0.000638689     0.0316977     -0.0201494
     GARCH{1}       0.705065     0.0673594        10.4672
      ARCH{1}       0.567741     0.0747457        7.59563
  Leverage{1}      -0.321158     0.0533449         -6.0204

EstMdl =

    EGARCH(1,1) Conditional Variance Model:
    ----------------------------------------
    Distribution: Name = 'Gaussian'
               P: 1
               Q: 1
        Constant: -0.000638689
           GARCH: {0.705065} at Lags [1]
            ARCH: {0.567741} at Lags [1]
        Leverage: {-0.321158} at Lags [1]
```

The result is a new `egarch` model called `EstMdl`. The parameter estimates in `EstMdl` resemble the parameter values that generated the simulated data.

Estimate GJR Model Parameters Without Initial Values

Fit a GJR(1,1) model to simulated data.

Simulate 500 data points from a GJR(1,1) model.

$$y_t = \varepsilon_t,$$

where $\varepsilon_t = \sigma_t z_t$ and

$$\sigma_t^2 = 0.001 + 0.5\sigma_{t-1}^2 + 0.2\varepsilon_{t-1}^2 + 0.2I\left[\varepsilon_{t-1} < 0\right]\varepsilon_{t-1}^2.$$

Use the default Gaussian innovation distribution for z_t.

```
Mdl = gjr('Constant',0.001,'GARCH',0.5,...
    'ARCH',0.2,'Leverage',0.2);

rng default; % For reproducibility
[v,y] = simulate(Mdl,500);
```

The output `v` contains simulated conditional variances. `y` is a column vector of simulated responses (innovations).

Specify a GJR(1,1) model with unknown coefficients, and fit it to the series `y`.

```
ToEstMdl = gjr(1,1);
EstMdl = estimate(ToEstMdl,y)
```

```
    GJR(1,1) Conditional Variance Model:
    ------------------------------------
    Conditional Probability Distribution: Gaussian

                                Standard          t
     Parameter       Value          Error       Statistic
    -----------   -----------   ------------   -----------
     Constant    0.000973819   0.000251354        3.87429
     GARCH{1}       0.460555     0.0717928        6.41505
      ARCH{1}       0.241255     0.0634092        3.80474
  Leverage{1}       0.250508      0.112655        2.22368

EstMdl =

    GJR(1,1) Conditional Variance Model:
    ------------------------------------
    Distribution: Name = 'Gaussian'
               P: 1
               Q: 1
        Constant: 0.000973819
           GARCH: {0.460555} at Lags [1]
            ARCH: {0.241255} at Lags [1]
        Leverage: {0.250508} at Lags [1]
```

The result is a new `gjr` model called `EstMdl`. The parameter estimates in `EstMdl` resemble the parameter values that generated the simulated data.

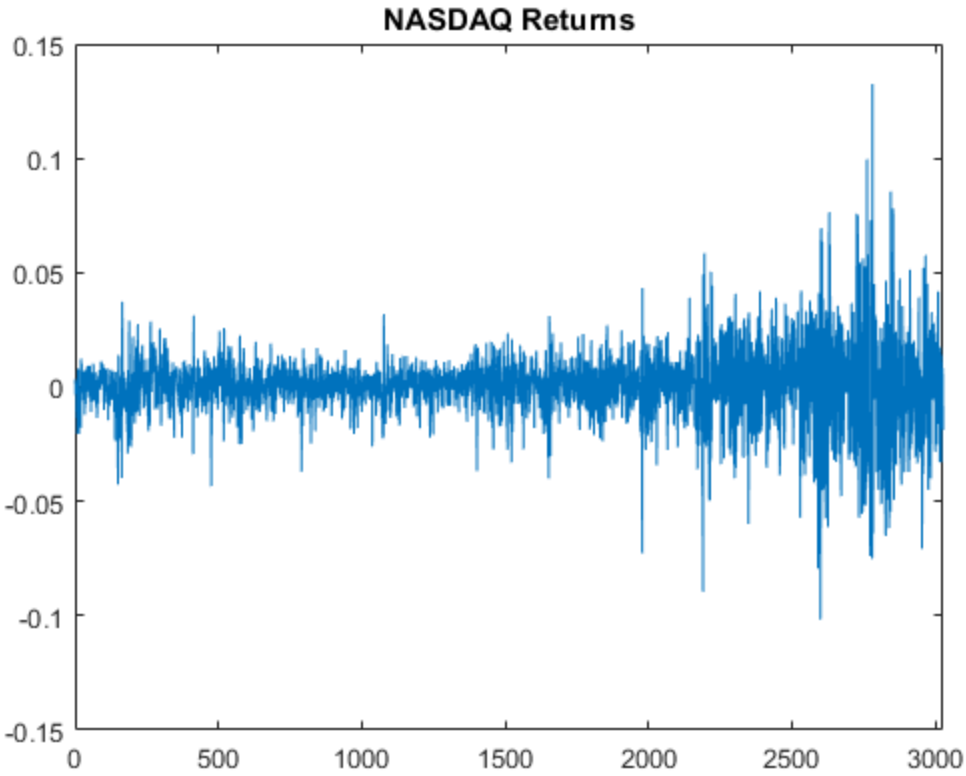
Estimate GARCH Model Parameters Using Presample Data

Fit a GARCH(1,1) model to the daily close NASDAQ Composite Index returns.

Load the NASDAQ data included with the toolbox. Convert the index to returns.

```
load Data_EquityIdx
nasdaq = DataTable.NASDAQ;
y = price2ret(nasdaq);
T = length(y);

figure
```

```
plot(y)
xlim([0,T])
title('NASDAQ Returns')
```



The returns exhibit volatility clustering.

Specify a GARCH(1,1) model, and fit it to the series. One presample innovation is required to initialize this model. Use the first observation of `y` as the necessary presample innovation.

```
Mdl = garch(1,1);
[EstMdl,EstParamCov] = estimate(Mdl,y(2:end),'E0',y(1))
```

```
    GARCH(1,1) Conditional Variance Model:
    --------------------------------------
    Conditional Probability Distribution: Gaussian

                                Standard          t
     Parameter       Value        Error       Statistic
    -----------   -----------   ------------   -----------
     Constant    1.99864e-06   5.42273e-07        3.68567
     GARCH{1}       0.883564    0.00843403        104.762
      ARCH{1}       0.109026    0.00764706        14.2573

EstMdl =

    GARCH(1,1) Conditional Variance Model:
    --------------------------------------
    Distribution: Name = 'Gaussian'
               P: 1
               Q: 1
        Constant: 1.99864e-06
           GARCH: {0.883564} at Lags [1]
            ARCH: {0.109026} at Lags [1]

EstParamCov =

   1.0e-04 *

    0.0000   -0.0000    0.0000
   -0.0000    0.7113   -0.5343
    0.0000   -0.5343    0.5848
```

The output `EstMdl` is a new `garch` model with estimated parameters.

Use the output variance-covariance matrix to calculate the estimate standard errors.

```
se = sqrt(diag(EstParamCov))
```

```
se =

    0.0000
    0.0084
    0.0076
```

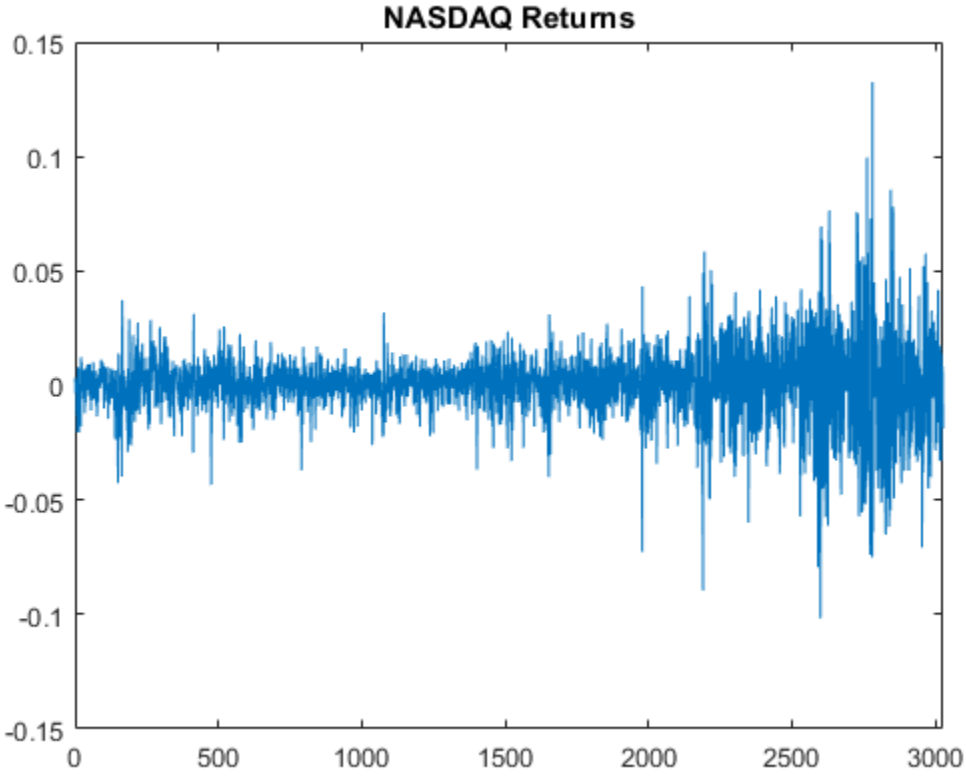
These are the standard errors shown in the estimation output display. They correspond (in order) to the constant, GARCH coefficient, and ARCH coefficient.

Estimate EGARCH Model Parameters Using Presample Data

Fit an EGARCH(1,1) model to the daily close NASDAQ Composite Index returns.

Load the NASDAQ data included with the toolbox. Convert the index to returns.

```
load Data_EquityIdx
nasdaq = DataTable.NASDAQ;
y = price2ret(nasdaq);
T = length(y);

figure
plot(y)
xlim([0,T])
title('NASDAQ Returns')
```



The returns exhibit volatility clustering.

Specify an EGARCH(1,1) model, and fit it to the series. One presample innovation is required to initialize this model. Use the first observation of `y` as the necessary presample innovation.

```
Mdl = egarch(1,1);
[EstMdl,EstParamCov] = estimate(Mdl,y(2:end),'E0',y(1))
```

```
    EGARCH(1,1) Conditional Variance Model:
    ----------------------------------------
    Conditional Probability Distribution: Gaussian
```

```
                              Standard          t
     Parameter       Value      Error       Statistic
    -----------   -----------   ------------   -----------
     Constant      -0.134783      0.022092       -6.10101
     GARCH{1}       0.983909    0.00242211         406.22
      ARCH{1}       0.199644     0.0139654        14.2955
  Leverage{1}     -0.0602429    0.00564702       -10.6681

EstMdl =

    EGARCH(1,1) Conditional Variance Model:
    ----------------------------------------
    Distribution: Name = 'Gaussian'
               P: 1
               Q: 1
        Constant: -0.134783
           GARCH: {0.983909} at Lags [1]
            ARCH: {0.199644} at Lags [1]
        Leverage: {-0.0602429} at Lags [1]

EstParamCov =

   1.0e-03 *

    0.4881    0.0533   -0.1018    0.0106
    0.0533    0.0059   -0.0118    0.0017
   -0.1018   -0.0118    0.1950    0.0016
    0.0106    0.0017    0.0016    0.0319
```

The output `EstMdl` is a new `egarch` model with estimated parameters.

Use the output variance-covariance matrix to calculate the estimate standard errors.

```
se = sqrt(diag(EstParamCov))
```

```
se =

    0.0221
    0.0024
    0.0140
    0.0056
```

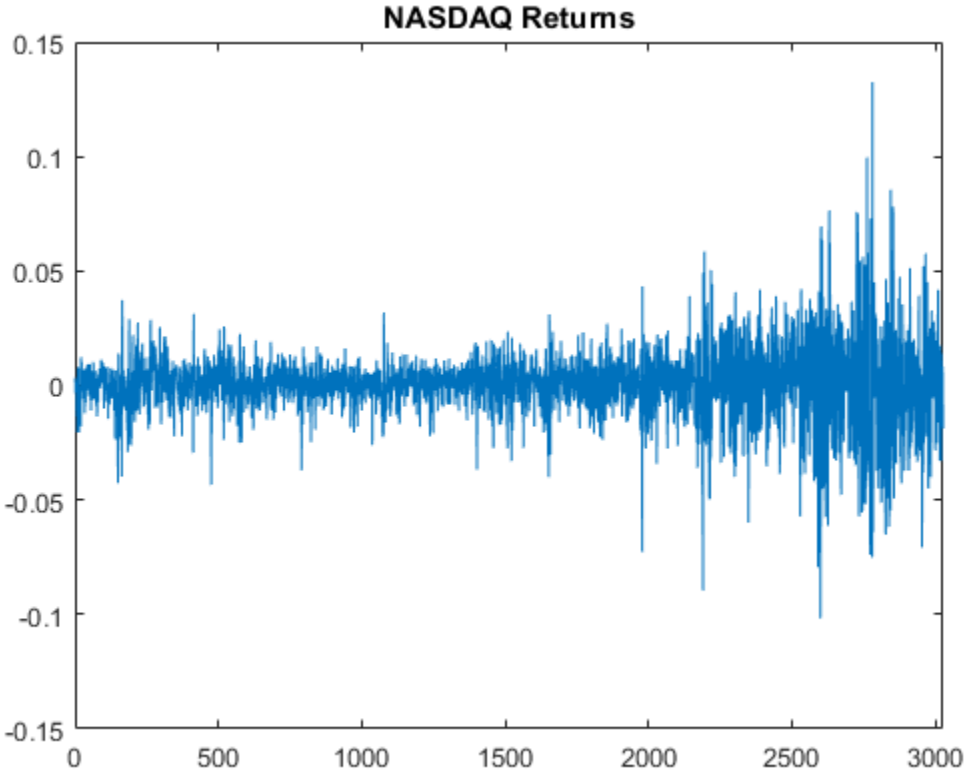
These are the standard errors shown in the estimation output display. They correspond (in order) to the constant, GARCH coefficient, ARCH coefficient, and leverage coefficient.

Estimate GJR Model Parameters Using Presample Data

Fit a GJR(1,1) model to the daily close NASDAQ Composite Index returns.

Load the NASDAQ data included with the toolbox. Convert the index to returns.

```
load Data_EquityIdx
nasdaq = DataTable.NASDAQ;
y = price2ret(nasdaq);
T = length(y);

figure
plot(y)
xlim([0,T])
title('NASDAQ Returns')
```



The returns exhibit volatility clustering.

Specify a GJR(1,1) model, and fit it to the series. One presample innovation is required to initialize this model. Use the first observation of `y` as the necessary presample innovation.

```
Mdl = gjr(1,1);
[EstMdl,EstParamCov] = estimate(Mdl,y(2:end),'E0',y(1))
```

```
    GJR(1,1) Conditional Variance Model:
    ------------------------------------
    Conditional Probability Distribution: Gaussian
```

```
                              Standard          t
    Parameter       Value       Error       Statistic
    -----------   -----------   ------------   -----------
     Constant    2.45647e-06   5.68527e-07        4.32076
     GARCH{1}       0.881379    0.00948646        92.9092
      ARCH{1}      0.0640741    0.00919501        6.96836
  Leverage{1}      0.0888268     0.0099137           8.96

EstMdl =

    GJR(1,1) Conditional Variance Model:
    ------------------------------------
    Distribution: Name = 'Gaussian'
               P: 1
               Q: 1
        Constant: 2.45647e-06
           GARCH: {0.881379} at Lags [1]
            ARCH: {0.0640741} at Lags [1]
        Leverage: {0.0888268} at Lags [1]

EstParamCov =

   1.0e-04 *

    0.0000   -0.0000    0.0000    0.0000
   -0.0000    0.8999   -0.6930   -0.0002
    0.0000   -0.6930    0.8455   -0.3605
    0.0000   -0.0002   -0.3605    0.9828
```

The output `EstMdl` is a new `gjr` model with estimated parameters.

Use the output variance-covariance matrix to calculate the estimate standard errors.

```
se = sqrt(diag(EstParamCov))


se =

    0.0000
    0.0095
    0.0092
    0.0099
```

These are the standard errors shown in the estimation output display. They correspond (in order) to the constant, GARCH coefficient, ARCH coefficient, and leverage coefficient.

- "Compare Conditional Variance Models Using Information Criteria" on page 6-87
- "Likelihood Ratio Test for Conditional Variance Models" on page 6-83
- "Estimate Conditional Mean and Variance Models" on page 5-129

Input Arguments

`Mdl` — Conditional variance model

`garch` model object | `egarch` model object | `gjr` model object

Conditional variance model containing unknown parameters, specified as a `garch`, `egarch`, or `gjr` model object.

`estimate` treats non-`NaN` elements in `Mdl` as equality constraints, and does not estimate the corresponding parameters.

`y` — Single path of response data

numeric column vector

Single path of response data, specified as a numeric column vector. The software infers the conditional variances from `y`, i.e., the data to which the model is fit.

`y` is usually an innovation series with mean 0 and conditional variance characterized by the model specified in `Mdl`. In this case, `y` is a continuation of the innovation series `E0`.

`y` can also represent an innovation series with mean 0 plus an offset. A nonzero `Offset` signals the inclusion of an offset in `Mdl`.

The last observation of `y` is the latest observation.

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Display','iter','E0',[0.1; 0.05]` specifies to display iterative optimization information, and `[0.05; 0.1]` as presample innovations.

For GARCH, EGARCH, and GJR Models

`'ARCH0'` — Initial coefficient estimates corresponding to past innovation terms

numeric vector

Initial coefficient estimates corresponding to past innovation terms, specified as the comma-separated pair consisting of `'ARCH0'` and a numeric vector.

- For GARCH(P,Q) and GJR(P,Q) models:
  - `ARCH0` must be a numeric vector containing nonnegative elements.
  - `ARCH0` contains the initial coefficient estimates associated with the past squared innovation terms that compose the ARCH polynomial.
  - By default, `estimate` derives initial estimates using standard time series techniques.
- For EGARCH(P,Q) models:
  - `ARCH0` contains the initial coefficient estimates associated with the magnitude of the past standardized innovations that compose the ARCH polynomial.
  - By default, `estimate` sets the initial coefficient estimate associated with the first nonzero lag in the model to a small positive value. All other values are zero.

The number of coefficients in `ARCH0` must equal the number of lags associated with nonzero coefficients in the ARCH polynomial, as specified in the `ARCHLags` property of `Mdl`.

Data Types: `double`

`'Constant0'` — Initial conditional variance model constant estimate

scalar

Initial conditional variance model constant estimate, specified as the comma-separated pair consisting of `'Constant0'` and a scalar.

For GARCH(P,Q) and GJR(P,Q) models, `Constant0` must be a positive scalar.

By default, `estimate` derives initial estimates using standard time series techniques.

Data Types: `double`

`'Display'` — Command Window display option

`'params'` (default) | `'diagnostics'` | `'full'` | `'iter'` | `'off'` | cell vector of strings

Command Window display option, specified as the comma-separated pair consisting of `'Display'` and a string or cell vector of strings.

Set `Display` using any combination of values in this table.

Value	estimate Displays
`'diagnostics'`	Optimization diagnostics
`'full'`	Maximum likelihood parameter estimates, standard errors, *t* statistics, iterative optimization information, and optimization diagnostics
`'iter'`	Iterative optimization information
`'off'`	No display in the Command Window
`'params'`	Maximum likelihood parameter estimates, standard errors, and *t* statistics

For example:

- To run a simulation where you are fitting many models, and therefore want to suppress all output, use `'Display','off'`.
- To display all estimation results and the optimization diagnostics, use `'Display',{'params','diagnostics'}`.

Data Types: `char` | `cell`

`'DoF0'` — Initial *t*-distribution degrees-of-freedom parameter estimate

`10` (default) | positive scalar

Initial *t*-distribution degrees-of-freedom parameter estimate, specified as the comma-separated pair consisting of `'DoF0'` and a positive scalar. `DoF0` must exceed 2.

Data Types: `double`

`'E0'` — Presample innovations

numeric column vector

Presample innovations, specified as the comma-separated pair consisting of `'E0'` and a numeric column vector. The presample innovations provide initial values for the innovations process of the conditional variance model `Mdl`. The presample innovations derive from a distribution with mean 0.

`E0` must contain at least `Mdl.Q` rows. If `E0` contains extra rows, then `estimate` uses the latest `Mdl.Q` presample innovations. The last row contains the latest presample innovation.

The defaults are:

- For GARCH(P,Q) and GJR(P,Q) models, `estimate` sets any necessary presample innovations to the square root of the average squared value of the offset-adjusted response series `y`.
- For EGARCH(P,Q) models, `estimate` sets any necessary presample innovations to zero.

Data Types: `double`

`'GARCH0'` — Initial coefficient estimates for past conditional variance terms

numeric vector

Initial coefficient estimates for past conditional variance terms, specified as the comma-separated pair consisting of `'GARCH0'` and a numeric vector.

- For GARCH(P,Q) and GJR(P,Q) models:
  - `GARCH0` must be a numeric vector containing nonnegative elements.
  - `GARCH0` contains the initial coefficient estimates associated with the past conditional variance terms that compose the GARCH polynomial.
- For EGARCH(P,Q) models,`GARCH0` contains the initial coefficient estimates associated with past log conditional variance terms that compose the GARCH polynomial.

The number of coefficients in `GARCH0` must equal the number of lags associated with nonzero coefficients in the GARCH polynomial, as specified in the `GARCHLags` property of `Mdl`.

By default, `estimate` derives initial estimates using standard time series techniques.

Data Types: `double`

'Offset0' — Initial innovation mean model offset estimate

scalar

Initial innovation mean model offset estimate, specified as the comma-separated pair consisting of 'Offset0' and a scalar.

By default, estimate sets the initial estimate to the sample mean of y.

Data Types: double

'Options' — Optimization options

optimoptions optimization controller | optimset optimization controller

Optimization options, specified as the comma-separated pair consisting of 'Options' and an optimoptions or optimset optimization controller. For details on altering the default values of the optimizer, see optimoptions, optimset, or fmincon in Optimization Toolbox.

Suppose that you want to change the constraint tolerance to 1e-6. Set Options = optimoptions(@fmincon,'TolCon',1e-6,'Algorithm','sqp'), and then pass Options into estimate using 'Options',Options.

By default, estimate uses the same default options as fmincon, except Algorithm = sqp and TolCon = 1e-7.

'V0' — Presample conditional variances

numeric column vector with positive entries

Presample conditional variances, specified as the comma-separated pair consisting of 'V0' and numeric column vector with positive entries. V0 provide initial values for conditional variance process of the conditional variance model Mdl.

For GARCH(P,Q) and GJR(P,Q) models, V0 must have at least Mdl.P rows.

For EGARCH(P,Q) models,V0 must have at least max(Mdl.P,Mdl.Q) rows.

If the number of rows in V0 exceeds the necessary number, only the latest observations are used. The last row contains the latest observation.

By default, estimate sets the necessary presample conditional variances to the average squared value of the offset-adjusted response series y.

Data Types: double

For EGARCH and GJR Models

`'Leverage0'` — Initial coefficient estimates past leverage terms

`0` (default) | numeric vector

Initial coefficient estimates past leverage terms, specified as the comma-separated pair consisting of `'Leverage0'` and a numeric vector.

For EGARCH(P,Q) models, `Leverage0` contains the initial coefficient estimates associated with past standardized innovation terms that compose the leverage polynomial.

For GJR(P,Q) models, `Leverage0` contains the initial coefficient estimates associated with past, squared, negative innovations that compose the leverage polynomial.

The number of coefficients in `Leverage0` must equal the number of lags associated with nonzero coefficients in the leverage polynomial (`Leverage`), as specified in `LeverageLags`.

Data Types: `double`

Notes

- `NaN`s indicate missing values. `estimate` removes them. The software merges the presample data (`E0` and `V0`) separately from the effective sample data (`y`), and then uses list-wise deletion to remove rows containing at least one `NaN`. Removing `NaN`s in the data reduces the sample size, and can also create irregular time series.
- `estimate` assumes that you synchronize the presample data such that the latest observations occur simultaneously.
- If you specify a value for `Display`, then it takes precedence over the specifications of the optimization options `Diagnostics` and `Display`. Otherwise, `estimate` honors all selections related to the display of optimization information in the optimization options.
- If you do not specify `E0` and `V0`, then `estimate` derives the necessary presample observations from the unconditional, or long-run, variance of the offset-adjusted response process.
  - For all conditional variance models, `V0` is the sample average of the squared disturbances of the offset-adjusted response data `y`.

- For GARCH(P,Q) and GJR(P,Q) models, `E0` is the square root of the average squared value of the offset-adjusted response series `y`.
- For EGARCH(P,Q) models, `E0` is `0`.

These specifications minimize initial transient effects.

Output Arguments

`EstMdl` — Conditional variance model containing parameter estimates

`garch` model object | `egarch` model object | `gjr` model object

Conditional variance model containing parameter estimates, returned as a `garch`, `egarch`, or `gjr` model object. `estimate` uses maximum likelihood to calculate all parameter estimates not constrained by `Mdl` (i.e., constrained parameters have known values).

`EstMdl` is a fully specified conditional variance model. To infer conditional variances for diagnostic checking, pass `EstMdl` to `infer`. To simulate or forecast conditional variances, pass `EstMdl` to `simulate` or `forecast`, respectively.

`EstParamCov` — Variance-covariance matrix of maximum likelihood estimates

numeric matrix

Variance-covariance matrix of maximum likelihood estimates of model parameters known to the optimizer, returned as a numeric matrix.

The rows and columns associated with any parameters estimated by maximum likelihood contain the covariances of estimation error. The standard errors of the parameter estimates are the square root of the entries along the main diagonal.

The rows and columns associated with any parameters that are held fixed as equality constraints contain `0`s.

`estimate` uses the outer product of gradients (OPG) method to perform covariance matrix estimation.

`estimate` orders the parameters in `EstParamCov` as follows:

- Constant

- Nonzero GARCH coefficients at positive lags
- Nonzero ARCH coefficients at positive lags
- For EGARCH and GJR models, nonzero leverage coefficients at positive lags
- Degrees of freedom (t innovation distribution only)
- Offset (models with nonzero offset only)

Data Types: `double`

`logL` — Optimized loglikelihood objective function value

scalar

Optimized loglikelihood objective function value, returned as a scalar.

Data Types: `double`

`info` — Summary information

structure array

Summary information, returned as a structure.

Field	Description
`exitflag`	Optimization exit flag (see `fmincon` in Optimization Toolbox)
`options`	Optimization options controller (see `optimoptions` and `fmincon` in Optimization Toolbox)
`X`	Vector of final parameter estimates
`X0`	Vector of initial parameter estimates

For example, you can display the vector of final estimates by typing `info.X` in the Command Window.

Data Types: `struct`

More About

Tips

Suppose `EstParamCov` is an estimated parameter covariance matrix returned by `estimate`. The software sets the variances and covariances of parameters fixed

during estimation to `0`. Enter this command to count the number of free parameters (`numParams`) in a fitted model.

```
numParams = sum(any(EstParamCov))
```

This command counts the number of columns (or equivalently, rows) with any nonzero values.

- Using garch Objects
- Using egarch Objects
- Using gjr Objects
- "Maximum Likelihood Estimation for Conditional Variance Models" on page 6-62
- "Conditional Variance Model Estimation with Equality Constraints" on page 6-65
- "Presample Data for Conditional Variance Model Estimation" on page 6-67
- "Initial Values for Conditional Variance Model Estimation" on page 6-69
- "Optimization Settings for Conditional Variance Model Estimation" on page 6-71

References

[1] Bollerslev, T. "Generalized Autoregressive Conditional Heteroskedasticity." *Journal of Econometrics*. Vol. 31, 1986, pp. 307–327.

[2] Bollerslev, T. "A Conditionally Heteroskedastic Time Series Model for Speculative Prices and Rates of Return." *The Review of Economics and Statistics*. Vol. 69, 1987, pp. 542–547.

[3] Box, G. E. P., G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis: Forecasting and Control*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

[4] Enders, W. *Applied Econometric Time Series*. Hoboken, NJ: John Wiley & Sons, 1995.

[5] Engle, R. F. "Autoregressive Conditional Heteroskedasticity with Estimates of the Variance of United Kingdom Inflation." *Econometrica*. Vol. 50, 1982, pp. 987–1007.

[6] Glosten, L. R., R. Jagannathan, and D. E. Runkle. "On the Relation between the Expected Value and the Volatility of the Nominal Excess Return on Stocks." *The Journal of Finance*. Vol. 48, No. 5, 1993, pp. 1779–1801.

[7] Greene, W. H. *Econometric Analysis*. 3rd ed. Upper Saddle River, NJ: Prentice Hall, 1997.

[8] Hamilton, J. D. *Time Series Analysis*. Princeton, NJ: Princeton University Press, 1994.

See Also

`egarch` | `filter` | `forecast` | `garch` | `gjr` | `infer` | `print` | `simulate`

Introduced in R2012a

estimate

Class: arima

Estimate ARIMA or ARIMAX model parameters

Syntax

```
EstMdl = estimate(Mdl,y)
[EstMdl,EstParamCov,logL,info] = estimate(Mdl,y)
[EstMdl,EstParamCov,logL,info] = estimate(Mdl,y,Name,Value)
```

Description

`EstMdl = estimate(Mdl,y)` uses maximum likelihood to estimate the parameters of the ARIMA(p,D,q) model `Mdl` given the observed univariate time series `y`. `EstMdl` is an arima model that stores the results.

`[EstMdl,EstParamCov,logL,info] = estimate(Mdl,y)` additionally returns `EstParamCov`, the variance-covariance matrix associated with estimated parameters, `logL`, the optimized loglikelihood objective function, and `info`, a data structure of summary information.

`[EstMdl,EstParamCov,logL,info] = estimate(Mdl,y,Name,Value)` estimates the model with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

`Mdl` — ARIMA or ARIMAX model

`arima` model

ARIMA or ARIMAX model, specified as an arima model returned by `arima` or `estimate`.

`estimate` treats non-`NaN` elements in `Mdl` as equality constraints and does not estimate the corresponding parameters.

`y` — Single path of response data

numeric column vector

Single path of response data to which the model is fit, specified as a numeric column vector. The last observation of `y` is the latest.

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'AR0'` — Initial estimates of nonseasonal autoregressive coefficients

numeric vector

Initial estimates of the nonseasonal autoregressive coefficients for the ARIMA model, specified as the comma-separated pair consisting of `'AR0'` and a numeric vector.

The number of coefficients in `AR0` must equal the number of lags associated with nonzero coefficients in the nonseasonal autoregressive polynomial, `ARLags`.

By default, `estimate` derives initial estimates using standard time series techniques.

Data Types: `double`

`'Beta0'` — Initial estimates of regression coefficients

numeric vector

Initial estimates of regression coefficients for the regression component, specified as the comma-separated pair consisting of `'Beta0'` and a numeric vector.

The number of coefficients in `Beta0` must equal the number of columns of `X`.

By default, `estimate` derives initial estimates using standard time series techniques.

Data Types: `double`

`'Constant0'` — Initial ARIMA model constant estimate

scalar

Initial ARIMA model constant estimate, specified as the comma-separated pair consisting of `'Constant0'` and a scalar.

By default, `estimate` derives initial estimates using standard time series techniques.

Data Types: `double`

`'Display'` — Command Window display option

`'params'` (default) | `'diagnostics'` | `'full'` | `'iter'` | `'off'` | cell vector of strings

Command Window display option, specified as the comma-separated pair consisting of `'Display'` and a string or cell vector of strings.

Set `Display` using any combination of values in this table.

Value	estimate Displays
`'diagnostics'`	Optimization diagnostics
`'full'`	Maximum likelihood parameter estimates, standard errors, *t* statistics, iterative optimization information, and optimization diagnostics
`'iter'`	Iterative optimization information
`'off'`	No display in the Command Window
`'params'`	Maximum likelihood parameter estimates, standard errors, and *t* statistics

For example:

- To run a simulation where you are fitting many models, and therefore want to suppress all output, use `'Display','off'`.
- To display all estimation results and the optimization diagnostics, use `'Display', {'params','diagnostics'}`.

Data Types: `char` | `cell`

`'DoF0'` — Initial *t*-distribution degrees-of-freedom parameter estimate

`10` (default) | positive scalar

Initial *t*-distribution degrees-of-freedom parameter estimate, specified as the comma-separated pair consisting of `'DoF0'` and a positive scalar. `DoF0` must exceed 2.

Data Types: `double`

`'E0'` — Presample innovations

numeric column vector

Presample innovations that have mean 0 and provide initial values for the ARIMA(p,$D$$q$) model, specified as the comma-separated pair consisting of `'E0'` and a numeric column vector.

`E0` must contain at least `Mdl.Q` rows. If you use a conditional variance model, such as a `garch` model, then the software might require more than `Mdl.Q` presample innovations.

If `E0` contains extra rows, then `estimate` uses the latest `Mdl.Q` presample innovations. The last row contains the latest presample innovation.

By default, `estimate` sets the necessary presample innovations to `0`.

Data Types: `double`

`'MA0'` — Initial estimates of nonseasonal moving average coefficients

numeric vector

Initial estimates of nonseasonal moving average coefficients for the ARIMA(p,$D$$q$) model, specified as the comma-separated pair consisting of `'MA0'` and a numeric vector.

The number of coefficients in `MA0` must equal the number of lags associated with nonzero coefficients in the nonseasonal moving average polynomial, `MALags`.

By default, `estimate` derives initial estimates using standard time series techniques.

Data Types: `double`

`'Options'` — Optimization options

`optimoptions` optimization controller | `optimset` optimization controller

Optimization options, specified as the comma-separated pair consisting of `'Options'` and an `optimoptions` or `optimset` optimization controller. For details on altering the default values of the optimizer, see `optimoptions`, `optimset`, or `fmincon` in Optimization Toolbox.

Suppose that you want to change the constraint tolerance to `1e-6`. Set `Options = optimoptions(@fmincon,'TolCon',1e-6,'Algorithm','sqp')`, and then pass `Options` into `estimate` using `'Options',Options`.

By default, `estimate` uses the same default options as `fmincon`, except `Algorithm` = `sqp` and `TolCon` = `1e-7`.

`'SAR0'` — Initial estimates of seasonal autoregressive coefficients

numeric vector

Initial estimates of seasonal autoregressive coefficients for the ARIMA(p,$D$$q$) model, specified as the comma-separated pair consisting of `'SAR0'` and a numeric vector.

The number of coefficients in `SAR0` must equal the number of lags associated with nonzero coefficients in the seasonal autoregressive polynomial, `SARLags`.

By default, `estimate` derives initial estimates using standard time series techniques.

Data Types: `double`

`'SMA0'` — Initial estimates of seasonal moving average coefficients
numeric vector

Initial estimates of seasonal moving average coefficients for the ARIMA(p,$D$$q$) model, specified as the comma-separated pair consisting of `'SMA0'` and a vector.

The number of coefficients in `SMA0` must equal the number of lags with nonzero coefficients in the seasonal moving average polynomial, `SMALags`.

By default, `estimate` derives initial estimates using standard time series techniques.

Data Types: `double`

`'V0'` — Presample conditional variances
numeric column vector with positive entries

Presample conditional variances that provide initial values for any conditional variance model, specified as the comma-separated pair consisting of `'V0'` and a numeric column vector with positive entries.

The software requires `V0` to have at least the number of observations required to initialize the variance model. If the number of rows in `V0` exceeds the number necessary, then `estimate` only uses the latest observations. The last row contains the latest observation.

If the variance of the model is constant, then `V0` is unnecessary.

By default, `estimate` sets the necessary presample conditional variances to the average of the squared inferred residuals.

Data Types: `double`

`'Variance0'` — Initial estimates of variances of innovations
positive scalar | cell vector of positive scalars

Initial estimates of variances of innovations for the ARIMA(p,Dq) model, specified as the comma-separated pair consisting of `'Variance0'` and a positive scalar or a cell vector of positive scalars. If `Variance0` is a cell vector, then the conditional variance model must recognize the parameter names as valid coefficients.

By default, `estimate` derives initial estimates using standard time series techniques.

Data Types: `double | cell`

`'X'` — Exogenous predictors

matrix

Exogenous predictors in the regression model, specified as the comma-separated pair consisting of `'X'` and a matrix.

The columns of `X` are separate, synchronized time series, with the last row containing the latest observations.

If you do not specify `Y0`, then the number of rows of `X` must be at least `numel(y,2) + Mdl.P`. Otherwise, the number of rows of `X` should be at least the length of `y`.

If the number of rows of `X` exceeds the number necessary, then `estimate` uses the latest observations and synchronizes `X` with the response series `y`.

By default, `estimate` does not estimate the regression coefficients regardless of their presence in `Mdl`.

Data Types: `double`

`'Y0'` — Presample response data

numeric column vector

Presample response data that provides initial values for the ARIMA(p,Dq) model, specified as the comma-separated pair consisting of `'Y0'` and a numeric column vector.

`Y0` is a column vector with at least `Mdl.P` rows. If the number of rows in `Y0` exceeds `Mdl.P`, `estimate` only uses the latest `Mdl.P` observations. The last row contains the latest observation.

By default, `estimate` backward forecasts for the necessary amount of presample observations.

Data Types: `double`

Notes

- `NaN`s indicate missing values, and `estimate` removes them. The software merges the presample data (`E0`, `V0`, and `Y0`) separately from the effective sample data (`X` and `y`),

  then uses list-wise deletion to remove any `NaN`s. Removing `NaN`s in the data reduces the sample size, and can also create irregular time series.
- Removing `NaN`s in the data reduces the sample size, and can also create irregular time series.
- `estimate` assumes that you synchronize the response and exogenous predictors such that the last (latest) observation of each occurs simultaneously. The software also assumes that you synchronize the presample series similarly.
- If you specify a value for `Display`, then it takes precedence over the specifications of the optimization options `Diagnostics` and `Display`. Otherwise, `estimate` honors all selections related to the display of optimization information in the optimization options.

Output Arguments

`EstMdl` — Model containing parameter estimates

`arima` model

Model containing parameter estimates, returned as an `arima` model. `estimate` uses maximum likelihood to calculate all parameter estimates not constrained by `Mdl` (that is, all parameters in `Mdl` that you set to `NaN`).

`EstParamCov` — Variance-covariance matrix of maximum likelihood estimates

matrix

Variance-covariance matrix of maximum likelihood estimates of model parameters known to the optimizer, returned as a matrix.

The rows and columns contain the covariances of the parameter estimates. The standard errors of the parameter estimates are the square root of the entries along the main diagonal.

The rows and columns associated with any parameters held fixed as equality constraints contain `0`s.

`estimate` uses the outer product of gradients (OPG) method to perform covariance matrix estimation.

`estimate` orders the parameters in `EstParamCov` as follows:

- Constant
- Nonzero `AR` coefficients at positive lags
- Nonzero `SAR` coefficients at positive lags
- Nonzero `MA` coefficients at positive lags
- Nonzero `SMA` coefficients at positive lags
- Regression coefficients (when you specify `X` in `estimate`)
- Variance parameters (scalar for constant-variance models, vector of additional parameters otherwise)
- Degrees of freedom (t innovation distribution only)

Data Types: `double`

`logL` — Optimized loglikelihood objective function value

scalar

Optimized loglikelihood objective function value, returned as a scalar.

Data Types: `double`

`info` — Summary information

structure array

Summary information, returned as a structure.

Field	Description
`exitflag`	Optimization exit flag (see `fmincon` in Optimization Toolbox)
`options`	Optimization options controller (see `optimoptions` and `fmincon` in Optimization Toolbox)
`X`	Vector of final parameter estimates
`X0`	Vector of initial parameter estimates

For example, you can display the vector of final estimates by typing `info.X` in the Command Window.

Data Types: `struct`

Examples

Estimate ARIMA Model Parameters Without Initial Values

Fit an ARMA(2,1) model to simulated data.

Simulate 500 data points from the ARMA(2,1) model

$$y_t = 0.5y_{t-1} - 0.3y_{t-2} + \varepsilon_t + 0.2\varepsilon_{t-1},$$

where ε_t follows a Gaussian distribution with mean 0 and variance 0.1.

```
Mdl = arima('AR',{0.5,-0.3},'MA',0.2,...
 'Constant',0,'Variance',0.1);

rng(5); % For reproducibility
y = simulate(Mdl,500);
```

The simulated data is stored in the column vector `Y`.

Specify an ARMA(2,1) model with no constant and unknown coefficients and variance.

```
ToEstMdl = arima(2,0,1);
ToEstMdl.Constant = 0
```

```
ToEstMdl =

    ARIMA(2,0,1) Model:
    --------------------
    Distribution: Name = 'Gaussian'
               P: 2
               D: 0
               Q: 1
        Constant: 0
              AR: {NaN NaN} at Lags [1 2]
             SAR: {}
              MA: {NaN} at Lags [1]
             SMA: {}
        Variance: NaN
```

Fit the ARMA(2,1) model to `y`.

```
EstMdl = estimate(ToEstMdl,y);
```

```
    ARIMA(2,0,1) Model:
    --------------------
    Conditional Probability Distribution: Gaussian

                                Standard          t
     Parameter       Value        Error       Statistic
    -----------   -----------   ------------   -----------
     Constant             0         Fixed          Fixed
        AR{1}      0.494039      0.103213        4.78662
        AR{2}      -0.25348     0.0699297       -3.62478
        MA{1}      0.279583      0.107211        2.60778
     Variance      0.100092     0.0066403        15.0734
```

The result is a new `arima` model called `EstMdl`. The estimates in `EstMdl` resemble the parameter values that generated the simulated data.

Estimate ARIMA Model Parameters Using Initial Values

Fit an integrated ARIMA(1,1,1) model to the daily close of the NASDAQ Composite Index.

Load the NASDAQ data included with the toolbox. Extract the first 1500 observations of the Composite Index (January 1990 to December 1995).

```
load Data_EquityIdx
nasdaq = DataTable.NASDAQ(1:1500);
```

Specify an ARIMA(1,1,1) model for fitting.

```
Mdl = arima(1,1,1);
```

The model is nonseasonal, so you can use shorthand syntax.

Fit the model to the first half of the data.

```
EstMdl = estimate(Mdl,nasdaq(1:750));
```

```
    ARIMA(1,1,1) Model:
    --------------------
```

```
    Conditional Probability Distribution: Gaussian

                               Standard          t
     Parameter       Value       Error       Statistic
    -----------   -----------   ------------   -----------
     Constant       0.223403      0.184177        1.21298
        AR{1}       0.114341      0.119438       0.957319
        MA{1}       0.127638      0.119251        1.07032
     Variance        18.9833      0.689994        27.5122
```

The result is a new `arima` model (`EstMdl`). The estimated parameters, their standard errors, and t statistics display in the Command Window.

Use the estimated parameters as initial values for fitting the second half of the data.

```
con0 = EstMdl.Constant;
ar0 = EstMdl.AR{1};
ma0 = EstMdl.MA{1};
var0 = EstMdl.Variance;

[EstMdl2,EstParamCov2,logL2,info2] = estimate(Mdl,....
   nasdaq(751:end),'Constant0',con0,'AR0',ar0,...
   'MA0',ma0,'Variance0',var0);


    ARIMA(1,1,1) Model:
    --------------------
    Conditional Probability Distribution: Gaussian

                               Standard          t
     Parameter       Value       Error       Statistic
    -----------   -----------   ------------   -----------
     Constant       0.611432      0.326754        1.87123
        AR{1}      -0.150712      0.117818       -1.27919
        MA{1}       0.385689      0.109055        3.53666
     Variance        36.4933       1.22699        29.7421
```

The parameter estimates are stored in the `info` data structure. Display the final parameter estimates.

```
info2.X


ans =
```

```
    0.6114
   -0.1507
    0.3857
   36.4933
```

Estimate ARIMAX Model Parameters Without Initial Values

Fit an ARIMAX model to a simulated time series without specifying initial values for the response or the parameters.

Define the ARIMAX(2,1,1) model

$$(1 - 0.5L + 0.3L^2)(1 - L)^1 y_t = 1.5x_{1,t} + 2.6x_{2,t} - 0.3x_{3,t} + \varepsilon_t + 0.2\varepsilon_{t-1}$$

to eventually simulate a time series of length 500, where ε_t follows a Gaussian distribution with mean 0 and variance 0.1.

```
Mdl = arima('AR',{0.5,-0.3},'MA',0.2,'D',1,...
    'Constant',0,'Variance',0.1,'Beta',[1.5 2.6 -0.3]);
T = 500;
```

Simulate three stationary AR(1) series and presample values:

$$\begin{aligned} x_{1,t} &= 0.1x_{1,t-1} + \eta_{1,t} \\ x_{2,t} &= 0.2x_{2,t-1} + \eta_{2,t} \\ x_{3,t} &= 0.3x_{3,t-1} + \eta_{3,t}, \end{aligned}$$

where $\eta_{i,t}$ follows a Gaussian distribution with mean 0 and variance 0.01 for $i = \{1,2,3\}$.

```
numObs = Mdl.P + T;
MdlX1 = arima('AR',0.1,'Constant',0,'Variance',0.01);
MdlX2 = arima('AR',0.2,'Constant',0,'Variance',0.01);
MdlX3 = arima('AR',0.3,'Constant',0,'Variance',0.01);
X1 = simulate(MdlX1,numObs);
X2 = simulate(MdlX2,numObs);
X3 = simulate(MdlX3,numObs);
Xmat = [X1 X2 X3];
```

The simulated exogenous predictors are stored in the `numObs`-by-3 matrix `Xmat`.

Simulate 500 data points from the ARIMA(2,1,1) model.

```
y = simulate(Mdl,T,'X',Xmat);
```

The simulated response is stored in the column vector `y`.

Create an ARIMA(2,1,1) model with known `0`-valued constant and unknown coefficients and variance.

```
ToEstMdl = arima(2,1,1);
ToEstMdl.Constant = 0
```

```
ToEstMdl =

    ARIMA(2,1,1) Model:
    --------------------
    Distribution: Name = 'Gaussian'
               P: 3
               D: 1
               Q: 1
        Constant: 0
              AR: {NaN NaN} at Lags [1 2]
             SAR: {}
              MA: {NaN} at Lags [1]
             SMA: {}
        Variance: NaN
```

`ToEstMdl` is an ARIMA(2,1,1) model. `estimate` changes this designation to ARIMAX(2,1,1) when you pass the exogenous predictors into the `X` argument. `estimate` estimates all parameters with the value `NaN` in `ToEstMdl`.

Fit the ARIMAX(2,1,1) model to `y` including regression matrix `Xmat`.

```
EstMdl = estimate(ToEstMdl,y,'X',Xmat);
```

```
    ARIMAX(2,1,1) Model:
    ---------------------
    Conditional Probability Distribution: Gaussian

                                Standard          t
     Parameter       Value        Error       Statistic
    -----------   -----------   ------------   -----------
     Constant             0         Fixed          Fixed
        AR{1}      0.416338     0.0460672        9.03763
        AR{2}     -0.274052     0.0406445       -6.74265
        MA{1}      0.334598     0.0572075        5.84885
        Beta1        1.4194      0.142422        9.96619
```

```
       Beta2         2.54199       0.133102         19.0981
       Beta3        -0.287669        0.14035        -2.04965
    Variance        0.0967773     0.00579104         16.7115
```

`ToEstMdl` is a new `arima` model designated as ARIMAX(2,1,1) since exogenous predictors enter the model. The estimates in `ToEstMdl` resemble the parameter values that generated the simulated data.

Estimate ARIMAX Model Parameters Using Initial Values

Fit an ARIMAX model to a time series specifying initial values for the response and the parameters.

The Credit Defaults data set contains four variables:

- Default rate on investment-grade corporate bonds (IGD)
- Percentage of investment-grade bond issuers first rated 3 years ago (AGE)
- One-year-ahead forecast of the change in corporate profits, adjusted for inflation (CPF)
- Spread between corporate bond yields and those of comparable government bonds (SPR)

Assume that an ARIMAX(1,0,0) model is appropriate to fit IGD using AGE, CPF, and SPR as exogenous predictors. Load the Credit Defaults data set. Assign the response IGD to `y`. Assign the predictors AGE, CPF, and SPR to the matrix `X`.

```
load Data_CreditDefaults
X = Data(:,[1 3:4]);
T = size(X,1);
y = Data(:,5);
```

The response and exogenous predictor series should be stationary before you continue. If your response is not stationary, then specify the degree of integration in the `arima` statement. If your exogenous predictors are not stationary, then you must difference them using `diff`. The series in this example are stationary to not distract from its main purpose.

Separate the initial values from the main response and exogenous predictors. Choose initial values for the regression coefficients `Beta0`.

```
y0 = y(1);
yEst = y(2:T);
```

```
XEst = X(2:end,:);
Beta0 = [0.5 0.5 0.5];
```

`y0` initializes the response series and `yest` is the main response series for estimation. `XEst` is the main exogenous predictor matrix for estimation.

Specify the model `Mdl` to fit to the data.

```
Mdl = arima(1,0,0);
```

Fit the model to the data and specify the initial values.

```
EstMdl = estimate(Mdl,yEst,'X',XEst,...
    'Y0',y0,'Beta0',Beta0);
```

```
    ARIMAX(1,0,0) Model:
    --------------------
    Conditional Probability Distribution: Gaussian

                               Standard          t
     Parameter       Value       Error       Statistic
    -----------   -----------   ------------   -----------
     Constant      -0.204768      0.266078      -0.769581
        AR{1}     -0.0173106      0.565618     -0.0306048
        Beta1      0.0239329     0.0218417        1.09574
        Beta2     -0.0124602    0.00749916       -1.66155
        Beta3      0.0680873     0.0745041       0.913874
     Variance     0.00539462    0.00224393          2.4041
```

- "Estimate Multiplicative ARIMA Model" on page 5-113
- "Estimate Conditional Mean and Variance Models" on page 5-129
- "Model Seasonal Lag Effects Using Indicator Variables" on page 5-117

Tip

Suppose `EstParamCov` is an estimated parameter covariance matrix returned by `estimate`. The software sets the variances and covariances of parameters fixed during estimation to `0`. Enter this command to count the number of free parameters (`numParams`) in a fitted model.

```
numParams = sum(any(EstParamCov))
```

This command counts the number of columns (or equivalently, rows) with any nonzero values.

References

[1] Box, G. E. P., G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis: Forecasting and Control* 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

[2] Enders, W. *Applied Econometric Time Series*. Hoboken, NJ: John Wiley & Sons, 1995.

[3] Greene, W. H. *Econometric Analysis.* 3rd ed. Upper Saddle River, NJ: Prentice Hall, 1997.

[4] Hamilton, J. D. *Time Series Analysis*. Princeton, NJ: Princeton University Press, 1994.

See Also

arima | filter | forecast | impulse | infer | print | simulate

More About

- "Maximum Likelihood Estimation for Conditional Mean Models" on page 5-98
- "Conditional Mean Model Estimation with Equality Constraints" on page 5-101
- "Presample Data for Conditional Mean Model Estimation" on page 5-103
- "Initial Values for Conditional Mean Model Estimation" on page 5-106
- "Optimization Settings for Conditional Mean Model Estimation" on page 5-108

estimate

Class: regARIMA

Estimate parameters of regression models with ARIMA errors

Syntax

```
EstMdl = estimate(Mdl,y)
[EstMdl,EstParamCov,logL,info] = estimate(Mdl,y)
[EstMdl,EstParamCov,logL,info] = estimate(Mdl,y,Name,Value)
```

Description

`EstMdl = estimate(Mdl,y)` uses maximum likelihood to estimate the parameters of the regression model with ARIMA time series errors, `Mdl`, given the response series `y`. `EstMdl` is a regARIMA model that stores the results.

`[EstMdl,EstParamCov,logL,info] = estimate(Mdl,y)` additionally returns `EstParamCov`, the variance-covariance matrix associated with estimated parameters, `logL`, the optimized loglikelihood objective function, and `info`, a data structure of summary information.

`[EstMdl,EstParamCov,logL,info] = estimate(Mdl,y,Name,Value)` estimates the model using additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

`Mdl` — Regression model with ARIMA errors

`regARIMA` model

Regression model with ARIMA errors, specified as a regARIMA model returned by `regARIMA` or `estimate`.

`estimate` treats non-`NaN` elements in `Mdl` as equality constraints, and does not estimate the corresponding parameters.

y — Single path of response data

numeric column vector

Single path of response data to which the model is fit, specified as a numeric column vector. The last observation of `y` is the latest.

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

'AR0' — Initial estimates of ARIMA error model nonseasonal autoregressive coefficients

numeric vector

Initial estimates of ARIMA error model nonseasonal autoregressive coefficients, specified as the comma-separated pair consisting of `'AR0'` and a numeric vector.

The number of coefficients in `AR0` must equal the number of lags associated with nonzero coefficients in the nonseasonal autoregressive polynomial.

By default, `estimate` derives initial estimates using standard time series techniques.

Data Types: `double`

'Beta0' — Initial estimates of regression coefficients

numeric vector

Initial estimates of regression coefficients, specified as the comma-separated pair consisting of `'Beta0'` and a numeric vector.

The number of coefficients in `Beta0` must equal the number of columns of `X`.

By default, `estimate` derives initial estimates using standard time series techniques.

Data Types: `double`

'Display' — Command Window display option

`'params'` (default) | `'diagnostics'` | `'full'` | `'iter'` | `'off'` | cell vector of strings

Command Window display option, specified as the comma-separated pair consisting of `'Display'` and a string or cell vector of strings.

Set `Display` using any combination of values in this table.

Value	estimate Displays
`'diagnostics'`	Optimization diagnostics
`'full'`	Maximum likelihood parameter estimates, standard errors, *t* statistics, iterative optimization information, and optimization diagnostics
`'iter'`	Iterative optimization information
`'off'`	No display in the Command Window
`'params'`	Maximum likelihood parameter estimates, standard errors, and *t* statistics

For example:

- To run a simulation where you are fitting many models, and therefore want to suppress all output, use `'Display','off'`.
- To display all estimation results and the optimization diagnostics, use `'Display',{'params','diagnostics'}`.

Data Types: `char` | `cell`

`'DoF0'` — Initial *t*-distribution degree-of-freedom estimate

`10` (default) | positive scalar

Initial *t*-distribution degree-of-freedom estimate, specified as the comma-separated pair consisting of `'DoF0'` and a positive scalar. `DoF0` must exceed 2.

Data Types: `double`

`'E0'` — Presample innovations

numeric column vector

Presample innovations that have mean 0 and provide initial values for the ARIMA error model, specified as the comma-separated pair consisting of `'E0'` and a numeric column

vector. `E0` must contain at least `Mdl.Q` rows. If `E0` contains extra rows, then `estimate` uses the latest `Mdl.Q` presample innovations. The last row contains the latest presample innovation.

By default, `estimate` sets the necessary presample innovations to `0`.

Data Types: `double`

`'Intercept0'` — Initial regression model intercept estimate

scalar

Initial regression model intercept estimate, specified as the comma-separated pair consisting of `'Intercept0'` and a scalar.

By default, `estimate` derives initial estimates using standard time series techniques.

Data Types: `double`

`'MA0'` — Initial estimates of ARIMA error model nonseasonal moving average coefficients

numeric vector

Initial estimates of ARIMA error model nonseasonal moving average coefficients, specified as the comma-separated pair consisting of `'MA0'` and a numeric vector.

The number of coefficients in `MA0` must equal the number of lags associated with nonzero coefficients in the nonseasonal moving average polynomial.

By default, `estimate` derives initial estimates using standard time series techniques.

Data Types: `double`

`'Options'` — Optimization options

`optimoptions` optimization controller | `optimset` optimization controller

Optimization options, specified as the comma-separated pair consisting of `'Options'` and an `optimoptions` or `optimset` optimization controller. For details on altering the default values of the optimizer, see `optimoptions`, `optimset`, or `fmincon` in Optimization Toolbox.

Suppose that you want to change the constraint tolerance to `1e-6`. Set `Options = optimoptions(@fmincon,'TolCon',1e-6,'Algorithm','sqp')`, and then pass `Options` into `estimate` using `'Options',Options`.

By default, `estimate` uses the same default options as `fmincon`, except `Algorithm` = `sqp` and `TolCon` = `1e-7`.

`'SAR0'` — Initial estimates of ARIMA error model seasonal autoregressive coefficients
numeric vector

Initial estimates of ARIMA error model seasonal autoregressive coefficients, specified as the comma-separated pair consisting of `'SAR0'` and a numeric vector.

The number of coefficients in `SAR0` must equal the number of lags associated with nonzero coefficients in the seasonal autoregressive polynomial.

By default, `estimate` derives initial estimates using standard time series techniques.

Data Types: `double`

`'SMA0'` — Initial estimates of ARIMA error model seasonal moving average coefficients
numeric vector

Initial estimates of ARIMA error model seasonal moving average coefficients, specified as the comma-separated pair consisting of `'SMA0'` and a numeric vector.

The number of coefficients in `SMA0` must equal the number of lags with nonzero coefficients in the seasonal moving average polynomial.

By default, `estimate` derives initial estimates using standard time series techniques.

Data Types: `double`

`'U0'` — Presample unconditional disturbances
numeric column vector

Presample unconditional disturbances that provide initial values for the ARIMA error model, specified as the comma-separated pair consisting of `'U0'` and a numeric column vector. `U0` must contain at least `Mdl.P` rows. If `U0` contains extra rows, then `estimate` uses the latest presample unconditional disturbances. The last row contains the latest presample unconditional disturbance.

By default, `estimate` backcasts for the necessary amount of presample unconditional disturbances.

Data Types: `double`

`'Variance0'` — Initial estimate of ARIMA error model innovation variance

positive scalar

Initial estimate of ARIMA error model innovation variance, specified as the comma-separated pair consisting of `'Variance0'` and a positive scalar.

By default, `estimate` derives initial estimates using standard time series techniques.

Data Types: `double`

`'X'` — Predictor data

matrix

Predictor data in the regression model, specified as the comma-separated pair consisting of `'X'` and a matrix.

The columns of `X` are separate, synchronized time series, with the last row containing the latest observations. The number of rows of `X` must be at least the length of `y`. If the number of rows of `X` exceeds the number required, then `estimate` uses the latest observations.

By default, `estimate` does not estimate the regression coefficients regardless of their presence in `Mdl`.

Data Types: `double`

Notes

- `NaN`s in `y`, `E0`, `U0`, and `X` indicate missing values, and `estimate` removes them. The software merges the presample data (`E0` and `U0`) separately from the effective sample data (`X` and `y`), then uses list-wise deletion to remove any `NaN`s. Removing `NaN`s in the data reduces the sample size, and can also create irregular time series.
- `estimate` assumes that you synchronize the data (presample separately from effective sample) such that the latest observations occur simultaneously.
- The intercept of a regression model with ARIMA errors having nonzero degrees of seasonal or nonseasonal integration is not identifiable. In other words, `estimate` cannot estimate an intercept of a regression model with ARIMA errors that has nonzero degrees of seasonal or nonseasonal integration. If you pass in such a model for estimation, `estimate` displays a warning in the Command Window and sets `EstMdl.Intercept` to `NaN`.

- If you specify a value for `Display`, then it takes precedence over the specifications of the optimization options `Diagnostics` and `Display`. Otherwise, `estimate` honors

  all selections related to the display of optimization information in the optimization options.

Output Arguments

`EstMdl` — Model containing parameter estimates

`regARIMA` model

Model containing the parameter estimates, returned as a `regARIMA` model. `estimate` uses maximum likelihood to calculate all parameter estimates not constrained by `Mdl` (that is, all parameters in `Mdl` that you set to `NaN`).

`EstParamCov` — Variance-covariance matrix of maximum likelihood estimates

matrix

Variance-covariance matrix of maximum likelihood estimates of model parameters known to the optimizer, returned as a matrix.

The rows and columns contain the covariances of the parameter estimates. The standard errors of the parameter estimates are the square root of the entries along the main diagonal. The rows and columns associated with any parameters held fixed as equality constraints contain `0`s.

`estimate` uses the outer product of gradients (OPG) method to perform covariance matrix estimation.

`estimate` orders the parameters in `EstParamCov` as follows:

- Intercept
- Nonzero `AR` coefficients at positive lags
- Nonzero `SAR` coefficients at positive lags
- Nonzero `MA` coefficients at positive lags
- Nonzero `SMA` coefficients at positive lags
- Regression coefficients (when you specify `X` in `estimate`)
- Innovations variance
- Degrees of freedom for the *t* distribution

Data Types: `double`

`logL` — Optimized loglikelihood objective function value

scalar

Optimized loglikelihood objective function value, returned as a scalar.

Data Types: `double`

`info` — Summary information

structure

Summary information, returned as a structure.

Field	Description
`exitflag`	Optimization exit flag (see `fmincon` in Optimization Toolbox)
`options`	Optimization options controller (see `optimoptions` and `fmincon` in Optimization Toolbox)
`X`	Vector of final parameter estimates
`X0`	Vector of initial parameter estimates

For example, you can display the vector of final estimates by typing `info.X` in the Command Window.

Data Types: `struct`

Examples

Estimate Parameters of Regression Model Containing ARIMA Errors Without Initial Values

Fit this regression model with ARMA(2,1) errors to simulated data:

$$\begin{aligned} y_t &= X_t \begin{bmatrix} 0.1 \\ -0.2 \end{bmatrix} + u_t \\ u_t &= 0.5u_{t-1} - 0.8u_{t-2} + \varepsilon_t - 0.5\varepsilon_{t-1}, \end{aligned}$$

where ε_t is Gaussian with variance 0.1.

Specify the regression model ARMA(2,1) errors. Simulate responses from the model and two predictor series.

```
Mdl = regARIMA('Intercept',0,'AR',{0.5 -0.8}, ...
    'MA',-0.5,'Beta',[0.1 -0.2],'Variance',0.1);
rng(1);
X =  randn(100,2);
y = simulate(Mdl,100,'X',X);
```

Specify a regression model with ARMA(2,1) errors with no intercept, and unknown coefficients and variance.

```
ToEstMdl = regARIMA(2,0,1);
ToEstMdl.Intercept = 0 % Exclude the intercept
```

```
ToEstMdl =

    ARIMA(2,0,1) Error Model:
    -------------------------
    Distribution: Name = 'Gaussian'
       Intercept: 0
               P: 2
               D: 0
               Q: 1
              AR: {NaN NaN} at Lags [1 2]
             SAR: {}
              MA: {NaN} at Lags [1]
             SMA: {}
        Variance: NaN
```

The AR coefficients, MA coefficients, and the innovation variance are `NaN` values. `estimate` estimates those parameters, but not the intercept. The intercept is held fixed at 0.

Fit the regression model with ARMA(2,1) errors to the data.

```
EstMdl = estimate(ToEstMdl,y,'X',X,'Display','params');
```

```
    Regression with ARIMA(2,0,1) Error Model:
    ----------------------------------------
    Conditional Probability Distribution: Gaussian

                                 Standard          t
     Parameter       Value          Error       Statistic
    -----------   -----------   ------------   -----------
    Intercept             0         Fixed          Fixed
```

```
        AR{1}       0.620303      0.104194       5.95338
        AR{2}      -0.697172     0.0795748      -8.76122
        MA{1}      -0.558083      0.131897      -4.23122
        Beta1       0.103667     0.0217347       4.76964
        Beta2      -0.209448     0.0241883      -8.65904
     Variance      0.0748852    0.00903584       8.28758
```

The result, `EstMdl`, is a new `regARIMA` model. The estimates in `EstMdl` resemble the parameter values that generated the simulated data.

Estimate Parameters of a Regression Model with ARIMA Errors Using Initial Values

Fit a regression model with ARMA(1,1) errors by regressing the log GDP onto the CPI and using initial values.

Load the US Macroeconomic data set and preprocess the data.

```
load Data_USEconModel;
logGDP = log(DataTable.GDP);
dlogGDP = diff(logGDP);         % For stationarity
dCPI = diff(DataTable.CPIAUCSL); % For stationarity
T = length(dlogGDP);            % Effective sample size
```

Specify an "empty" regression model with ARMA(1,1) errors.

```
ToEstMdl = regARIMA(1,0,1);
```

Fit the model to the first half of the data.

```
EstMdl0 = estimate(ToEstMdl,dlogGDP(1:ceil(T/2)),...
    'X',dCPI(1:ceil(T/2)),'Display','off');
```

The result is a new `regARIMA` model with the estimated parameters.

Use the estimated parameters as initial values for fitting the second half of the data.

```
Intercept0 = EstMdl0.Intercept;
AR0        = EstMdl0.AR{1};
MA0        = EstMdl0.MA{1};
Variance0  = EstMdl0.Variance;
Beta0      = EstMdl0.Beta;

[EstMdl,~,~,info] = estimate(ToEstMdl,...
   dlogGDP(floor(T/2)+1:end),'X',...
   dCPI(floor(T/2)+1:end),'Display','params',...
```

```
'Intercept0',Intercept0,'AR0',AR0,'MA0',MA0,...
'Variance0',Variance0,'Beta0',Beta0);
```

```
Regression with ARIMA(1,0,1) Error Model:
----------------------------------------
Conditional Probability Distribution: Gaussian

                           Standard          t
 Parameter       Value       Error       Statistic
-----------   -----------   ------------   -----------
Intercept      0.0111738    0.00210199         5.3158
    AR{1}       0.786836     0.0362291        21.7184
    MA{1}      -0.473619     0.0655402       -7.22639
    Beta1     0.00219331   0.000583268        3.76038
 Variance    4.83486e-05    4.1705e-06         11.593
```

Display all of the parameter estimates using `info.X`.

```
info.X
```

```
ans =

    0.0112
    0.7868
   -0.4736
    0.0022
    0.0000
```

The order of the parameter estimates in `info.X` matches the order that `estimate` displays in its output table.

- "Estimate a Regression Model with ARIMA Errors" on page 4-105
- "Intercept Identifiability in Regression Models with ARIMA Errors" on page 4-130
- "Compare Alternative ARIMA Model Representations" on page 4-136

Tip

Suppose `EstParamCov` is an estimated parameter covariance matrix returned by `estimate`. The software sets the variances and covariances of parameters fixed

during estimation to `0`. Enter this command to count the number of free parameters (`numParams`) in a fitted model.

```
numParams = sum(any(EstParamCov))
```

This command counts the number of columns (or equivalently, rows) with any nonzero values.

Algorithms

`estimate` estimates the parameters as follows:

1. Infer the unconditional disturbances from the regression model.
2. Infer the residuals of the ARIMA error model.
3. Use the distribution of the innovations to build the likelihood function.
4. Maximize the loglikelihood function with respect to the parameters using `fmincon`.

References

[1] Box, G. E. P., G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis: Forecasting and Control*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

[2] Davidson, R., and J. G. MacKinnon. *Econometric Theory and Methods*. Oxford, UK: Oxford University Press, 2004.

[3] Enders, W. *Applied Econometric Time Series*. Hoboken, NJ: John Wiley & Sons, Inc., 1995.

[4] Hamilton, J. D. *Time Series Analysis*. Princeton, NJ: Princeton University Press, 1994.

[5] Pankratz, A. *Forecasting with Dynamic Regression Models*. John Wiley & Sons, Inc., 1991.

[6] Tsay, R. S. *Analysis of Financial Time Series*. 2nd ed. Hoboken, NJ: John Wiley & Sons, Inc., 2005.

See Also

regARIMA | forecast | infer | simulate

More About

- "Maximum Likelihood Estimation for Conditional Mean Models" on page 5-98
- "Conditional Mean Model Estimation with Equality Constraints" on page 5-101
- "Presample Data for Conditional Mean Model Estimation" on page 5-103
- "Initial Values for Conditional Mean Model Estimation" on page 5-106
- "Optimization Settings for Conditional Mean Model Estimation" on page 5-108

estimate

Class: dssm

Maximum likelihood parameter estimation of diffuse state-space models

Syntax

```
EstMdl = estimate(Mdl,Y,params0)
EstMdl = estimate(Mdl,Y,params0,Name,Value)
[EstMdl,estParams,EstParamCov,logL,Output] = estimate( ___ )
```

Description

`EstMdl = estimate(Mdl,Y,params0)` estimates the unknown parameters of the diffuse state-space model `Mdl` using the diffuse Kalman filter and maximum likelihood.

- `Y` is the observed response series to which `estimate` fits `Mdl`.
- `params0` is a set of initial values for the unknown parameters.

`estimate` returns an estimated diffuse state-space model (`EstMdl`), which stores the estimated coefficient matrices and initial state means, covariance matrices, and distributions.

- For explicitly created state-space models, the software estimates all `NaN` values in the coefficient matrices (`Mdl.A`, `Mdl.B`, `Mdl.C`, and `Mdl.D`) and the initial state means and covariance matrix (`Mdl.Mean0` and `Mdl.Cov0`). For details on explicit and implicit model creation, see dssm.
- For implicitly created state-space models, you specify the model structure and the location of the unknown parameters using the parameter-to-matrix mapping function. Implicitly create a state-space model to estimate complex models, impose parameter constraints, and estimate initial states. The parameter-to-mapping function can also accommodate additional output arguments.

`EstMdl = estimate(Mdl,Y,params0,Name,Value)` estimates the diffuse state-space model with additional options specified by one or more `Name,Value` pair arguments. For example, you can specify to deflate the observations by a linear regression using

predictor data, control how the results appear in the Command Window, and indicate which estimation method to use for the parameter covariance matrix.

`[EstMdl,estParams,EstParamCov,logL,Output] = estimate( ___ )` additionally returns these arguments using any of the input arguments in the previous syntaxes.

- `estParams`, a vector containing the estimated parameters
- `EstParamCov`, the estimated variance-covariance matrix of the estimated parameters
- `logL`, the optimized loglikelihood value
- `Output`, optimization diagnostic information structure

Tips

Constrained likelihood objective function maximization

- You can specify any combination of linear inequality, linear equality, and upper and lower bound constraints on the parameters.
- If a parameter is unbounded below, then set `'lb',-Inf`.
- If a parameter is unbounded above, then set `'ub',Inf`.
- It is good practice to avoid equality and inequality constraints during optimization. For example, if you want to constrain the parameter w to be positive, then implicitly specify the state-space model using a parameter-to-matrix mapping function, set $w = \exp(s)$ within the function, and use unconstrained optimization to estimate s. Subsequently, s can assume any real value, but w must be positive.

Predictors and corresponding coefficients

- The state-space model `Mdl` does not store the predictors (Z_t) nor their corresponding regression coefficients (β). Supply the predictors and their corresponding coefficients wherever necessary using the appropriate name-value pair arguments.
- The predictor series serve as observation deflators. Subsequently, the deflated data set is $Y_t - Z_t\beta$, where:
  - $Z_t = \begin{pmatrix} z_{1t} & z_{2t} & \cdots & z_{dt} \end{pmatrix}$., that is, Z is a T-by$d$ matrix.
  - z_{jt} is the period t value of predictor j.
  - β is a d-by-n matrix of regression coefficients.

- To include an overall mean to the observation model, include a column of 1s in Z_t.
- If you want to account for predictor effects when you simulate (simulate), then you must deflate the observations manually. To deflate the observations, use $W_t = Y_t - Z_t\hat{\beta}$.
- If the state equation requires predictors, then expand the states by the constant 1 and the predictors.
- If the regression model is complex, then consider implicitly defining the state space model. For example, define the parameter-to-matrix mapping function using the following syntax pattern.

```
function [A,B,C,D,Mean0,Cov0,StateType,DeflateY] = ParamMap(params,Y,Z)
  ...
  DeflateY = Y - exp(params(9) + params(10)*Z);
  ...
end
```

  In this example, `Y` is the matrix of observations and `Z` is the matrix of predictors. The function returns `DeflateY`, which is the matrix of deflated observations. Specify `Y` and `Z` in the MATLAB Workspace before, and then pass `ParamMap` to `ssm` using the following syntax pattern.

```
Mdl = ssm(@(params)ParamMap(params,Y,Z))
```

  This is also useful if each response series requires a distinct set of predictors.
- If the state equation requires known predictors, then include the predictors as additional state variables. Since predictor data varies with time, a state-space model with predictors as states is time varying.

Diffuse State-Space Models

- You cannot use the square root method to filter and smooth diffuse state-space models (i.e., dssm model objects). As a workaround, you can convert a diffuse state-space model to a standard state-space model using ssm, and then you can filter using the square root method. Upon conversion, all diffuse states have a finite, albeit large, initial distribution variance of `1e7`.
- It is a best practice to let `estimate` determine the value of `SwitchTime`. However, in rare cases, you might experience numerical issues during estimation, filtering, or smoothing diffuse state-space models. For such cases, try experimenting with various `SwitchTime` specifications, or consider a different model structure (e.g., simplify the model or verify that the model is identifiable). For example, convert the diffuse state-space model to a standard state-space model using `ssm`.

Additional Tips

- The software accommodates missing data. Indicate missing data using `NaN` values in the observed responses (`Y`).
- It is good practice to check the convergence status of the optimization routine by displaying `Output.ExitFlag`.
- If the optimization algorithm does not converge, then you can increase the number of iterations using the `'Options'` name-value pair argument.
- If the optimization algorithm does not converge, then consider using refine, which might help you obtain better initial parameter values for optimization.

Input Arguments

`Mdl` — Diffuse state-space model

`dssm` model object

Diffuse state-space model containing unknown parameters, specified as a `dssm` model object returned by dssm.

`Mdl` does not store observed responses or predictor data. Supply the data wherever necessary using, the appropriate input and name-value pair arguments.

`Y` — Observed response data

numeric matrix | cell vector of numeric vectors

Observed response data to which `Mdl` is fit, specified as a numeric matrix or a cell vector of numeric vectors.

- If `Mdl` is time invariant with respect to the observation equation, then `Y` is a T-by-n matrix. Each row of the matrix corresponds to a period and each column corresponds to a particular observation in the model. Therefore, T is the sample size and n is the number of observations per period. The last row of `Y` contains the latest observations.
- If `Mdl` is time varying with respect to the observation equation, then `Y` is a T-by-1 cell vector. `Y{t}` contains an n_t-dimensional vector of observations for period t, where $t = 1,...,T$. The corresponding dimensions of the coefficient matrices in `Mdl.C{t}` and `Mdl.D{t}` must be consistent with the matrix in `Y{t}` for all periods. The last cell of `Y` contains the latest observations.
- Suppose that you created `Mdl` implicitly by specifying a parameter-to-matrix mapping function, and the function has input arguments for the observed responses or

predictors. The mapping function establishes a link to observed responses and the predictor data in the MATLAB workspace, which overrides the value of `Y`.

`NaN` elements indicate missing observations. For details on how the Kalman filter accommodates missing observations, see "Algorithms" on page 9-829.

Data Types: `double | cell`

`params0` — Initial values of unknown parameters

numeric vector

Initial values of unknown parameters for numeric maximum likelihood estimation, specified as a numeric vector.

The elements of `params0` correspond to the unknown parameters in the state-space model matrices `A`, `B`, `C`, and `D`, and, optionally, the initial state mean `Mean0` and covariance matrix `Cov0`.

- If you created `Mdl` explicitly (that is, by specifying the matrices without a parameter-to-matrix mapping function), then the software maps the elements of `params` to `NaN`s in the state-space model matrices and initial state values. The software searches for `NaN`s column-wise, following the order `A`, `B`, `C`, `D`, `Mean0`, `Cov0`.
- If you created `Mdl` implicitly (that is, by specifying the matrices with a parameter-to-matrix mapping function), then set initial parameter values for the state-space model matrices, initial state values, and state types within the parameter-to-matrix mapping function.

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Estimation Options

`'Beta0'` — Initial values of regression coefficients

numeric matrix

Initial values of regression coefficients, specified as the comma-separated pair consisting of `'Beta0'` and a d-by-n numeric matrix. d is the number of predictor variables (see `Predictors`) and n is the number of observed response series (see `Y`).

By default, `Beta0` is the ordinary least-squares estimate of `Y` onto `Predictors`.

Data Types: `double`

`'CovMethod'` — Asymptotic covariance estimation method

`'opg'` (default) | `'hessian'` | `'sandwich'`

Asymptotic covariance estimation method, specified as the comma-separated pair consisting of `'CovMethod'` and a string.

Set `CovMethod` using a value in this table.

Value	Description
`'Hessian'`	Negative, inverted Hessian matrix
`'OPG'`	Outer product of gradients (OPG)
`'Sandwich'`	Both Hessian and OPG

Example: `'CovMethod','Sandwich'`

Data Types: `char`

`'Display'` — Command Window display option

`'params'` (default) | `'diagnostics'` | `'full'` | `'iter'` | `'off'` | cell vector of strings

Command Window display option, specified as the comma-separated pair consisting of `'Display'` and a string or cell vector of strings.

Set `Display` using any combination of values in this table.

Value	estimate Displays
`'diagnostics'`	Optimization diagnostics
`'full'`	Maximum likelihood parameter estimates, standard errors, t statistics, iterative optimization information, and optimization diagnostics
`'iter'`	Iterative optimization information

Value	estimate Displays
`'off'`	No display in the Command Window
`'params'`	Maximum likelihood parameter estimates, standard errors, and t statistics

For example:

- To run a simulation where you are fitting many models, and therefore want to suppress all output, use `'Display','off'`.
- To display all estimation results and the optimization diagnostics, use `'Display',{'params','diagnostics'}`.

Data Types: `char` | `cell`

`'Options'` — Optimization options

`optimoptions` optimization controller

Optimization options, specified as the comma-separated pair consisting of `'Options'` and an `optimoptions` optimization controller. `Options` replaces default optimization options of the optimizer. For details on altering default values of the optimizer, see the optimization controller `optimoptions`, the constrained optimization function `fmincon`, or the unconstrained optimization function `fminunc` in Optimization Toolbox.

For example, suppose that you want to change the constraint tolerance to `1e-6`. Set `Options = optimoptions(@fmincon,'TolCon',1e-6,'Algorithm','sqp')` and then pass `Options` into `estimate` using `'Options',Options`.

By default:

- For constrained optimization, `estimate` maximizes the likelihood objective function using `fmincon` and its default options, but sets `'Algorithm','interior-point'`.
- For unconstrained optimization, `estimate` maximizes the likelihood objective function using `fminunc` and its default options, but sets `'Algorithm','quasi-newton'`.

`'Predictors'` — Predictor data

`[ ]` (default) | numeric matrix

Predictor data used to deflate the observations in a time-invariant state-space model, specified as the comma-separated pair consisting of `'Predictors'` and a T-by-d numeric matrix. T is the number of periods and d is the number of predictor variables.

Row t corresponds to the observed predictors at period t (Z_t) in the expanded observation equation

$$y_t - Z_t\beta = Cx_t + Du_t.$$

That is, the software deflates the observations using the regression component. β is the time-invariant vector of regression coefficients that the software estimates with all other parameters. `Predictors` and `Y` must have the same number of rows.

For n observations per period, the software regresses all predictor series onto each observation. Then, the software returns a d-by-n matrix of fitted regression coefficient vectors for each observation series.

If you specify `Predictors`, then `Mdl` must be time invariant. Otherwise, the software returns an error.

By default, the software excludes a regression component from the state-space model.

Data Types: `double`

`'SwitchTime'` — Final period for diffuse state initialization
positive integer

Final period for diffuse state initialization, specified as the comma-separated pair consisting of `'SwitchTime'` and a positive integer. That is, `estimate` uses the observations from period 1 to period `SwitchTime` as a presample to implement the *exact initial Kalman filter* (see "Diffuse Kalman Filter" on page 8-15 and [1]). After initializing the diffuse states, `estimate` applies the standard Kalman filter to the observations from periods `SwitchTime` + 1 to T.

The default value for `SwitchTime` is the last period in which the estimated smoothed state precision matrix is singular (i.e., the inverse of the covariance matrix). This specification represents the fewest number of observations required to initialize the diffuse states. Therefore, it is a best practice to use the default value.

If you set `SwitchTime` to a value greater than the default, then the effective sample size decreases. If you set `SwitchTime` to a value that is fewer than the default, then `estimate` might not have enough observations to initialize the diffuse states, which can result in an error or improper values.

In general, estimating, filtering, and smoothing state-space models with at least one diffuse state requires `SwitchTime` to be at least one. The default estimation display contains the effective sample size.

Data Types: `double`

`'Tolerance'` — Forecast uncertainty threshold

`0` (default) | nonnegative scalar

Forecast uncertainty threshold, specified as the comma-separated pair consisting of `'Tolerance'` and a nonnegative scalar.

If the forecast uncertainty for a particular observation is less than `Tolerance` during numerical estimation, then the software removes the uncertainty corresponding to the observation from the forecast covariance matrix before its inversion.

It is best practice to set `Tolerance` to a small number, for example, `1e-15`, to overcome numerical obstacles during estimation.

Example: `'Tolerance',1e-15`

Data Types: `double`

`'Univariate'` — Univariate treatment of multivariate series flag

`false` (default) | `true`

Univariate treatment of a multivariate series flag, specified as the comma-separated pair consisting of `'Univariate'` and `true` or `false`. Univariate treatment of a multivariate series is also known as *sequential filtering*.

The univariate treatment can accelerate and improve numerical stability of the Kalman filter. However, all observation innovations must be uncorrelated. That is, D_tD_t' must be diagonal, where D_t, $t = 1,...,T$, is one of the following:

- The matrix `D{t}` in a time-varying state-space model
- The matrix `D` in a time-invariant state-space model

Example: `'Univariate',true`

Data Types: `logical`

Constrained Optimization Options for fmincon

`'Aeq'` — Linear equality constraint parameter transformer

matrix

Linear equality constraint parameter transformer for constrained likelihood objective function maximization, specified as the comma-separated pair consisting of `'Aeq'` and a matrix.

If you specify `Aeq` and `beq`, then `estimate` maximizes the likelihood objective function using the equality constraint $\texttt{Aeq}\theta = \texttt{beq}$, where θ is a vector containing every `Mdl` parameter.

The number of rows of `Aeq` is the number of constraints, and the number of columns is the number of parameters that the software estimates. Order the columns of `Aeq` by `Mdl.A`, `Mdl.B`, `Mdl.C`, `Mdl.D`, `Mdl.Mean0`, `Mdl.Cov0`, and the regression coefficient (if the model has one).

Specify `Aeq` and `beq` together, otherwise `estimate` returns an error.

`Aeq` directly corresponds to the input argument `Aeq` of `fmincon`, not to the state-transition coefficient matrix `Mdl.A`.

By default, if you did not specify any constraint (linear inequality, linear equality, or upper and lower bound), then `estimate` maximizes the likelihood objective function using unconstrained maximization.

`'Aineq'` — Linear inequality constraint parameter transformer

matrix

Linear inequality constraint parameter transformer for constrained likelihood objective function maximization, specified as the comma-separated pair consisting of `'Aineq'` and a matrix.

If you specify `Aineq` and `bineq`, then `estimate` maximizes the likelihood objective function using the inequality constraint $\texttt{Aineq}\theta \le \texttt{bineq}$, where θ is a vector containing every `Mdl` parameter.

The number of rows of `Aineq` is the number of constraints, and the number of columns is the number of parameters that the software estimates. Order the columns of `Aineq` by `Mdl.A`, `Mdl.B`, `Mdl.C`, `Mdl.D`, `Mdl.Mean0`, `Mdl.Cov0`, and the regression coefficient (if the model has one).

Specify `Aineq` and `bineq` together, otherwise `estimate` returns an error.

`Aineq` directly corresponds to the input argument `A` of `fmincon`, not to the state-transition coefficient matrix `Mdl.A`.

By default, if you did not specify any constraint (linear inequality, linear equality, or upper and lower bound), then `estimate` maximizes the likelihood objective function using unconstrained maximization.

Data Types: `double`

`'beq'` — Linear equality constraints of transformed parameters

numeric vector

Linear equality constraints of the transformed parameters for constrained likelihood objective function maximization, specified as the comma-separated pair consisting of `'beq'` and a numeric vector.

If you specify `Aeq` and `beq`, then `estimate` maximizes the likelihood objective function using the equality constraint $\texttt{Aeq}\theta = \texttt{beq}$, where θ is a vector containing every `Mdl` parameter..

Specify `Aeq` and `beq` together, otherwise `estimate` returns an error.

`beq` directly corresponds to the input argument `beq` of `fmincon`, and is not associated with any component of `Mdl`.

By default, if you did not specify any constraint (linear inequality, linear equality, or upper and lower bound), then `estimate` maximizes the likelihood objective function using unconstrained maximization.

Data Types: `double`

`'bineq'` — Linear inequality constraint upper bounds

numeric vector

Linear inequality constraint upper bounds of the transformed parameters for constrained likelihood objective function maximization, specified as the comma-separated pair consisting of `'bineq'` and a numeric vector.

If you specify `Aineq` and `bineq`, then `estimate` maximizes the likelihood objective function using the inequality constraint $\texttt{Aineq}\theta \le \texttt{bineq}$, where θ is a vector containing every `Mdl` parameter.

Specify `Aineq` and `bineq` together, otherwise `estimate` returns an error.

`bineq` directly corresponds to the input argument `b` of `fmincon`, and is not associated with any component of `Mdl`.

By default, if you did not specify any constraint (linear inequality, linear equality, or upper and lower bound), then `estimate` maximizes the likelihood objective function using unconstrained maximization.

Data Types: `double`

`'lb'` — Lower bounds of parameters

numeric vector

Lower bounds of the parameters for constrained likelihood objective function maximization, specified as the comma-separated pair consisting of `'lb'` and a numeric vector.

If you specify `lb` and `ub`, then `estimate` maximizes the likelihood objective function subject to $\mathtt{lb} \le \theta \le \mathtt{ub}$, where θ is a vector containing every `Mdl` parameter.

Order the elements of `lb` by `Mdl.A`, `Mdl.B`, `Mdl.C`, `Mdl.D`, `Mdl.Mean0`, `Mdl.Cov0`, and the regression coefficient (if the model has one).

By default, if you did not specify any constraint (linear inequality, linear equality, or upper and lower bound), then `estimate` maximizes the likelihood objective function using unconstrained maximization.

Data Types: `double`

`'ub'` — Upper bounds of parameters

numeric vector

Upper bounds of the parameters for constrained likelihood objective function maximization, specified as the comma-separated pair consisting of `'ub'` and a numeric vector.

If you specify `lb` and `ub`, then `estimate` maximizes the likelihood objective function subject to $\mathtt{lb} \le \theta \le \mathtt{ub}$, where θ is a vector every `Mdl` parameter.

Order the elements of `ub` by `Mdl.A`, `Mdl.B`, `Mdl.C`, `Mdl.D`, `Mdl.Mean0`, `Mdl.Cov0`, and the regression coefficient (if the model has one).

By default, if you did not specify any constraint (linear inequality, linear equality, or upper and lower bound), then `estimate` maximizes the likelihood objective function using unconstrained maximization.

Data Types: `double`

Output Arguments

`EstMdl` — Diffuse state-space model containing parameter estimates

`dssm` model object

Diffuse state-space model containing the parameter estimates, returned as a dssm model object.

`estimate` uses maximum likelihood to calculate all parameter estimates. `EstMdl` stores the parameter estimates in the coefficient matrices (`EstMdl.A`, `EstMdl.B`, `EstMdl.C`, and `EstMdl.D`), and the initial state means and covariance matrix (`EstMdl.Mean0` and `EstMdl.Cov0`), regardless of specifying `Mdl` explicitly. For the estimated regression coefficient, see `estParams`.

`EstMdl` does not store observed responses or predictor data. If you plan to filter (using filter), forecast (using forecast), or smooth (using smooth) using `EstMdl`, then you need to supply the appropriate data.

`estParams` — Maximum likelihood estimates of model parameters

numeric vector

Maximum likelihood estimates of the model parameters known to the optimizer, returned as a numeric vector. `estParams` has the same dimensions as `params0`.

`estimate` arranges the estimates in `estParams` corresponding to unknown parameters in this order.

1. `EstMdl.A(:)`, that is, estimates in `EstMdl.A` listed column-wise
2. `EstMdl.B(:)`
3. `EstMdl.C(:)`
4. `EstMdl.D(:)`
5. `EstMdl.Mean0`
6. `EstMdl.Cov0(:)`
7. In models with predictors, estimated regression coefficients listed column-wise

`EstParamCov` — Variance-covariance matrix of maximum likelihood estimates

numeric matrix

Variance-covariance matrix of maximum likelihood estimates of the model parameters known to the optimizer, returned as a numeric matrix.

The rows and columns contain the covariances of the parameter estimates. The standard errors of the parameter estimates are the square root of the entries along the main diagonal.

`estimate` arranges the estimates in the rows and columns of `EstParamCov` corresponding to unknown parameters in this order.

1. `EstMdl.A(:)`, that is, estimates in `EstMdl.A` listed column-wise
2. `EstMdl.B(:)`
3. `EstMdl.C(:)`
4. `EstMdl.D(:)`
5. `EstMdl.Mean0`
6. `EstMdl.Cov0(:)`
7. In models with predictors, estimated regression coefficients listed column-wise

`logL` — Optimized loglikelihood value

scalar

Optimized loglikelihood value, returned as a scalar.

Missing observations do not contribute to the loglikelihood. The observations after the first `SwitchTime` periods contribute to the loglikelihood only.

`Output` — Optimization information

structure array

Optimization information, returned as a structure array.

This table describes the fields of `Output`.

Field	Description
`ExitFlag`	Optimization exit flag that describes the exit condition. For details, see `fmincon` and `fminunc`.
`Options`	Optimization options that the optimizer used for numerical estimation. For details, see `optimoptions`.

Data Types: `struct`

Examples

Fit Time-Invariant Diffuse State-Space Model to Data

This example generates data from a known model, and then fits a diffuse state-space model to the data.

Suppose that a latent process is this AR(1) process

$$x_t = 0.5x_{t-1} + u_t,$$

where u_t is Gaussian with mean 0 and standard deviation 1.

Generate a random series of 100 observations from x_t, assuming that the series starts at 1.5.

```
T = 100;
ARMdl = arima('AR',0.5,'Constant',0,'Variance',1);
x0 = 1.5;
rng(1); % For reproducibility
x = simulate(ARMdl,T,'Y0',x0);
```

Suppose further that the latent process is subject to additive measurement error as indicated in the equation

$$y_t = x_t + \varepsilon_t,$$

where ε_t is Gaussian with mean 0 and standard deviation 0.1.

Use the random latent state process (`x`) and the observation equation to generate observations.

```
y = x + 0.1*randn(T,1);
```

Together, the latent process and observation equations compose a state-space model. Supposing that the coefficients and variances are unknown parameters, the state-space model is

$$\begin{aligned} x_t &= \phi x_{t-1} + \sigma_1 u_t \\ y_t &= x_t + \sigma_2 \varepsilon_t. \end{aligned}$$

Specify the state-transition matrix. Use `NaN` values for unknown parameters.

```
A = NaN;
```

Specify the state-disturbance-loading coefficient matrix.

```
B = NaN;
```

Specify the measurement-sensitivity coefficient matrix.

```
C = 1;
```

Specify the observation-innovation coefficient matrix

```
D = NaN;
```

Create the state-space model using the coefficient matrices and specify that the state variable is diffuse. A diffuse state specification indicates complete ignorance on the values of the states.

```
StateType = 2;
Mdl = dssm(A,B,C,D,'StateType',StateType);
```

`Mdl` is a `dssm` model. Verify that the model is correctly specified by viewing its display in the Command Window.

Pass the observations to `estimate` to estimate the parameter. Set a starting value for the parameter to `params0`. σ_1 and σ_2 must be positive, so set the lower bound constraints using the `'lb'` name-value pair argument. Specify that the lower bound of ϕ is `-Inf`.

```
params0 = [0.9; 0.5; 0.1];
EstMdl = estimate(Mdl,y,params0,'lb',[-Inf; 0; 0])
```

```
Method: Maximum likelihood (fmincon)
Effective Sample size:             99
Logarithmic  likelihood:     -138.968
Akaike   info criterion:      283.936
Bayesian info criterion:      291.752
      |     Coeff      Std Err   t Stat    Prob
-------------------------------------------------
 c(1) | 0.56114       0.18045   3.10975  0.00187
 c(2) | 0.75836       0.24569   3.08661  0.00202
 c(3) | 0.57129       0.27455   2.08086  0.03745
      |
      |  Final State   Std Dev   t Stat    Prob
 x(1) | 1.24096       0.46532   2.66690  0.00766

EstMdl =


State-space model type: <a href="matlab: doc dssm">dssm</a>
```

```
State vector length: 1
Observation vector length: 1
State disturbance vector length: 1
Observation innovation vector length: 1
Sample size supported by model: Unlimited

State variables: x1, x2,...
State disturbances: u1, u2,...
Observation series: y1, y2,...
Observation innovations: e1, e2,...

State equation:
x1(t) = (0.56)x1(t-1) + (0.76)u1(t)

Observation equation:
y1(t) = x1(t) + (0.57)e1(t)

Initial state distribution:

Initial state means
 x1
  0

Initial state covariance matrix
     x1
 x1  Inf

State types
    x1
 Diffuse
```

`EstMdl` is a `dssm` model. The results of the estimation appear in the Command Window, contain the fitted state-space equations, and contain a table of parameter estimates, their standard errors, t statistics, and p-values.

You can use or display, for example the fitted state-transition matrix using dot notation.

```
EstMdl.A
```

```
ans =

    0.5611
```

Pass `EstMdl` to `forecast` to forecast observations, or to `simulate` to conduct a Monte Carlo study.

Estimate Diffuse State-Space Model Containing Regression Component

Suppose that the linear relationship between unemployment rate and the nominal gross national product (nGNP) is of interest. Suppose further that unemployment rate is an AR(1) series. Symbolically, and in state-space form, the model is

$$\begin{aligned} x_t &= \phi x_{t-1} + \sigma u_t \\ y_t - \beta Z_t &= x_t, \end{aligned}$$

where:

- x_t is the unemployment rate at time t.
- y_t is the observed change in the unemployment rate being deflated by the return of nGNP (Z_t).
- u_t is the Gaussian series of state disturbances having mean 0 and unknown standard deviation σ.

Load the Nelson-Plosser data set, which contains the unemployment rate and nGNP series, among other things.

```
load Data_NelsonPlosser
```

Preprocess the data by taking the natural logarithm of the nGNP series, and removing the starting `NaN` values from each series.

```
isNaN = any(ismissing(DataTable),2);       % Flag periods containing NaNs
gnpn = DataTable.GNPN(~isNaN);
y = diff(DataTable.UR(~isNaN));
T = size(gnpn,1);                          % The sample size
Z = price2ret(gnpn);
```

This example continues using the series without `NaN` values. However, using the Kalman filter framework, the software can accommodate series containing missing values.

Specify the coefficient matrices.

```
A = NaN;
B = NaN;
C = 1;
```

Create the state-space model using `dssm` by supplying the coefficient matrices and specifying that the state values come from a diffuse distribution. The diffuse specification indicates complete ignorance about the moments of the initial distribution.

```
StateType = 2;
Mdl = dssm(A,B,C,'StateType',StateType);
```

Estimate the parameters. Specify the regression component and its initial value for optimization using the `'Predictors'` and `'Beta0'` name-value pair arguments, respectively. Display the estimates and all optimization diagnostic information. Restrict the estimate of σ to all positive, real numbers.

```
params0 = [0.3 0.2]; % Initial values chosen arbitrarily
Beta0 = 0.1;
EstMdl = estimate(Mdl,y,params0,'Predictors',Z,'Display','full',...
    'Beta0',Beta0,'lb',[-Inf 0 -Inf]);
```

```
                                                          
   Diagnostic Information

Number of variables: 3

Functions
Objective:                               @(c)-fML(c,Mdl,Y,Predictors,unitFlag,sqrtFlag,me
Gradient:                                finite-differencing
Hessian:                                 finite-differencing (or Quasi-Newton)

Constraints
Nonlinear constraints:                   do not exist

Number of linear inequality constraints:    0
Number of linear equality constraints:      0
Number of lower bound constraints:          1
Number of upper bound constraints:          0

Algorithm selected
   interior-point


                                                          
   End diagnostic information
                                                First-order      Norm of
 Iter F-count            f(x)  Feasibility   optimality         step
    0       4    5.084683e+03    0.000e+00    5.096e+04
```

```
  1       8    6.405732e+02    0.000e+00    7.720e-02    1.457e+04
  2      12    6.405620e+02    0.000e+00    7.713e-02    1.058e-01
  3      16    6.405063e+02    0.000e+00    7.683e-02    5.285e-01
  4      20    6.402322e+02    0.000e+00    7.531e-02    2.632e+00
  5      24    6.389682e+02    0.000e+00    6.816e-02    1.289e+01
  6      28    6.346900e+02    0.000e+00    4.146e-02    5.821e+01
  7      32    6.314789e+02    0.000e+00    1.601e-02    8.771e+01
  8      36    6.307024e+02    0.000e+00    7.462e-03    5.266e+01
  9      40    6.304200e+02    0.000e+00    4.104e-03    4.351e+01
 10      44    6.303324e+02    0.000e+00    4.116e-03    3.168e+01
 11      48    6.303036e+02    0.000e+00    4.120e-03    2.417e+01
 12      52    6.302943e+02    0.000e+00    4.121e-03    1.816e+01
 13      56    6.302913e+02    0.000e+00    4.121e-03    1.375e+01
 14      60    6.302903e+02    0.000e+00    4.121e-03    1.062e+01
 15      64    6.302899e+02    0.000e+00    4.121e-03    9.300e+00
 16      68    6.302897e+02    0.000e+00    4.121e-03    9.121e+00
 17      72    6.302894e+02    0.000e+00    4.121e-03    1.313e+01
 18      77    6.302888e+02    0.000e+00    4.121e-03    3.413e+01
 19      82    6.302888e+02    0.000e+00    4.121e-03    7.903e+00
 20      86    6.302888e+02    0.000e+00    4.121e-03    3.076e+00
 21      90    6.302888e+02    0.000e+00    4.121e-03    1.375e+00
 22      94    6.302888e+02    0.000e+00    4.121e-03    6.476e-01
 23      98    6.302887e+02    0.000e+00    4.121e-03    1.619e+00
 24     102    6.302883e+02    0.000e+00    4.121e-03    4.675e+00
 25     106    6.302863e+02    0.000e+00    4.121e-03    1.421e+01
 26     110    6.302820e+02    0.000e+00    4.122e-03    6.846e+00
 27     114    6.302455e+02    0.000e+00    4.124e-03    1.644e+01
 28     118    6.299840e+02    0.000e+00    4.141e-03    6.680e+01
 29     122    6.285305e+02    0.000e+00    4.243e-03    3.447e+02
 30     126    6.203531e+02    0.000e+00    4.862e-03    1.802e+03

                                            First-order      Norm of
Iter F-count            f(x)  Feasibility   optimality         step
 31     130    5.202166e+02    0.000e+00    2.379e-01    1.034e+04
 32     135    5.092568e+02    0.000e+00    1.207e+00    9.939e+02
 33     140    4.768740e+02    0.000e+00    3.401e+00    5.025e+02
 34     145    4.133588e+02    0.000e+00    5.513e+00    2.526e+02
 35     151    3.760473e+02    0.000e+00    3.601e+00    6.555e+01
 36     156    3.568227e+02    0.000e+00    1.572e+01    9.471e+01
 37     161    3.278483e+02    0.000e+00    3.534e+01    4.761e+01
 38     166    2.716782e+02    0.000e+00    5.998e+00    2.404e+01
 39     173    2.711253e+02    0.000e+00    2.616e+00    2.938e+00
 40     180    2.711246e+02    0.000e+00    5.724e+00    6.473e-01
 41     189    2.710580e+02    0.000e+00    6.609e+00    6.477e-01
```

```
   42      194    2.709747e+02    0.000e+00    5.940e+00    6.475e-01
   43      198    2.705427e+02    0.000e+00    1.010e+00    9.036e-01
   44      202    2.699854e+02    0.000e+00    3.508e+00    2.566e+00
   45      206    2.684256e+02    0.000e+00    1.084e+01    9.181e+00
   46      210    2.635256e+02    0.000e+00    2.192e+01    3.090e+01
   47      214    2.062327e+02    0.000e+00    4.639e+00    1.836e+02
   48      221    1.998224e+02    0.000e+00    4.999e+00    2.413e+01
   49      226    1.798855e+02    0.000e+00    2.143e+01    7.639e+00
   50      232    1.741133e+02    0.000e+00    3.585e+01    1.675e+00
   51      237    1.656338e+02    0.000e+00    7.790e+01    9.176e+00
   52      241    1.506502e+02    0.000e+00    4.755e+01    1.536e+01
   53      246    1.477040e+02    0.000e+00    1.853e+01    6.053e+00
   54      250    1.427015e+02    0.000e+00    2.339e+01    6.196e-01
   55      255    1.349762e+02    0.000e+00    5.112e+01    1.266e+00
   56      259    1.278666e+02    0.000e+00    4.088e+01    1.088e+00
   57      265    1.127098e+02    0.000e+00    1.763e+01    2.881e+00
   58      269    1.116068e+02    0.000e+00    8.076e+00    6.365e-01
   59      273    1.108242e+02    0.000e+00    5.216e+00    2.060e+00
   60      277    1.105218e+02    0.000e+00    1.842e+00    1.561e-01

                                                First-order      Norm of
 Iter F-count            f(x)  Feasibility   optimality         step
   61      281    1.104810e+02    0.000e+00    3.733e-01    1.476e-01
   62      285    1.104776e+02    0.000e+00    3.632e-02    9.569e-02
   63      289    1.104771e+02    0.000e+00    1.016e-02    6.805e-02
   64      293    1.104771e+02    0.000e+00    1.520e-03    5.361e-03
   65      297    1.104771e+02    0.000e+00    6.555e-04    5.119e-05
   66      301    1.104771e+02    0.000e+00    1.311e-04    2.297e-05
   67      305    1.104771e+02    0.000e+00    8.583e-06    2.968e-06
   68      309    1.104771e+02    0.000e+00    4.768e-06    4.252e-07
   69      314    1.104771e+02    0.000e+00    1.311e-06    2.940e-07
   70      318    1.104771e+02    0.000e+00    2.861e-06    6.425e-07
   71      322    1.104771e+02    0.000e+00    9.881e-07    3.253e-08

Local minimum found that satisfies the constraints.

Optimization completed because the objective function is non-decreasing in
feasible directions, to within the default value of the function tolerance,
and constraints are satisfied to within the default value of the constraint tolerance.



Method: Maximum likelihood (fmincon)
```

```
Effective Sample size:             60
Logarithmic  likelihood:     -110.477
Akaike   info criterion:      226.954
Bayesian info criterion:      233.287
           |      Coeff       Std Err    t Stat    Prob
-------------------------------------------------------
 c(1)      |   0.59436       0.09408     6.31738  0
 c(2)      |   1.52554       0.10758    14.17991  0
 y <- z(1) | -24.26161       1.55730   -15.57930  0
           |
           |    Final State   Std Dev     t Stat   Prob
 x(1)      |   2.54764        0           Inf     0
```

Optimization information and a table of estimates and statistics output to the Command Window. `EstMdl` is an `ssm` model, and you can access its properties using dot notation.

Change Switching Time for Diffuse Model Estimation

For diffuse state-space models, the software implements the diffuse Kalman filter until it satisfies a regularity condition. To meet the condition, the software requires enough presample data. Once the software satisfies the condition, it switches to using the standard Kalman filter. By default, the software determines how much presample data it requires, and only uses as much as it needs. However, you can experiment by specifying other values for the `SwitchTime` name-value pair argument.

For this example, use the same data and model as in "Estimate Diffuse State-Space Model Containing Regression Component".

Load the Nelson-Plosser data set.

```
load Data_NelsonPlosser

isNaN = any(ismissing(DataTable),2);
gnpn = DataTable.GNPN(~isNaN);
y = DataTable.UR(~isNaN);
T = size(gnpn,1);
Z = log(gnpn);
```

Specify the coefficient matrices.

```
A = NaN;
B = NaN;
C = 1;
```

Create the state-space model using `dssm` by supplying the coefficient matrices and specifying that the state values come from a diffuse distribution. The diffuse specification indicates complete ignorance about the moments of the initial distribution.

```
StateType = 2;
Mdl = dssm(A,B,C,'StateType',StateType);
```

Estimate the parameters several times. For each time, change the period to switch to the standard Kalman filter.

```
params0 = [0.3 0.2]; % Initial values chosen arbitrarily
Beta0 = 0.1;

[~,estParams1] = estimate(Mdl,y,params0,'Predictors',Z,'Display','off',...
    'Beta0',Beta0,'lb',[-Inf 0 -Inf],'SwitchTime',1);
[~,estParams5] = estimate(Mdl,y,params0,'Predictors',Z,'Display','off',...
    'Beta0',Beta0,'lb',[-Inf 0 -Inf],'SwitchTime',5);
[~,estParams10] = estimate(Mdl,y,params0,'Predictors',Z,'Display','off',...
    'Beta0',Beta0,'lb',[-Inf 0 -Inf],'SwitchTime',10);
```

Compare the parameter estimates.

```
estParams1
estParams5
estParams10
```

```
estParams1 =

    1.0101    1.3574  -24.4585


estParams5 =

    1.0102    1.3832  -24.4852


estParams10 =

    1.0094    1.2735  -26.4448
```

Because `estimate` uses fewer data points for subsequent estimations, the estimates are slightly different.

- “Estimate Time-Varying Diffuse State-Space Model” on page 8-50

- "Estimate Random Parameter of State-Space Model" on page 8-116
- "Assess State-Space Model Stability Using Rolling Window Analysis" on page 8-172
- "Choose State-Space Model Specification Using Backtesting" on page 8-181

Algorithms

- The Kalman filter accommodates missing data by not updating filtered state estimates corresponding to missing observations. In other words, suppose there is a missing observation at period t. Then, the state forecast for period t based on the previous $t-1$ observations and filtered state for period t are equivalent.
- The diffuse Kalman filter requires presample data. If missing observations begin the time series, then the diffuse Kalman filter must gather enough nonmissing observations to initialize the diffuse states.
- For explicitly created state-space models, `estimate` applies all predictors to each response series. However, each response series has its own set of regression coefficients.
- If you do not specify optimization constraints, then `estimate` uses `fminunc` for unconstrained numerical estimation. If you specify any pair of optimization constraints, then `estimate` uses `fmincon` for constrained numerical estimation. For either type of optimization, optimization options you set using the name-value pair argument `Options` must be consistent with the options of the optimization algorithm.
- `estimate` passes the name-value pair arguments `Options`, `Aineq`, `bineq`, `Aeq`, `beq`, `lb`, and `ub` directly to the optimizer `fmincon` or `fminunc`.
- `estimate` fits regression coefficients along with all other state-space model parameters. The software is flexible enough to allow applying constraints to the regression coefficients using constrained optimization options. For more details, see the `Name,Value` pair arguments and `fmincon`.
- If you set `'Univariate',true`, then, during the filtering algorithm, the software sequentially updates rather then updating all at once. This might accelerate parameter estimation, especially for a low-dimensional, time-invariant model.
- Suppose that you want to create a state-space model using a parameter-to-matrix mapping function with this signature

  ```
  [A,B,C,D,Mean0,Cov0,StateType,DeflateY] = paramMap(params,Y,Z)
  ```
  and you specify the model using an anonymous function

  ```
  Mdl = dssm(@(params)paramMap(params,Y,Z))
  ```

The observed responses `Y` and predictor data `Z` are not input arguments in the anonymous function. If `Y` and `Z` exist in the MATLAB Workspace before creating `Mdl`, then the software establishes a link to them. Otherwise, if you pass `Mdl` to `estimate`, the software throws an error.

The link to the data established by the anonymous function overrides all other corresponding input argument values of `estimate`. This distinction is important particularly when conducting a rolling window analysis. For details, see "Rolling-Window Analysis of Time-Series Models" on page 8-168.

- For diffuse state-space models, `estimate` usually switches from the diffuse Kalman filter to the standard Kalman filter when the number of cumulative observations and the number of diffuse states are equal.

Limitations

- If the model is time varying with respect the observed responses, then the software does not support including predictors. If the observation vectors among different periods vary in length, then the software cannot determine which coefficients to use to deflate the observed responses.
- If a diffuse state-space model has identifiability issues (i.e., at least two sets of distinct parameters values yield the same likelihood value for all observations), then `estimate` cannot properly initialize the diffuse states, and results are not predictable.

References

[1] Durbin J., and S. J. Koopman. *Time Series Analysis by State Space Methods*. 2nd ed. Oxford: Oxford University Press, 2012.

See Also

dssm | filter | `fmincon` | `fminunc` | `optimoptions` | refine | smooth

More About

- "What Are State-Space Models?" on page 8-3
- "What Is the Kalman Filter?" on page 8-8
- "Rolling-Window Analysis of Time-Series Models" on page 8-168

Introduced in R2015b

estimate

Class: ssm

Maximum likelihood parameter estimation of state-space models

Syntax

```
EstMdl = estimate(Mdl,Y,params0)
EstMdl = estimate(Mdl,Y,params0,Name,Value)
[EstMdl,estParams,EstParamCov,logL,Output] = estimate( ___ )
```

Description

`EstMdl = estimate(Mdl,Y,params0)` estimates the parameters of `Mdl` using the Kalman filter and maximum likelihood, where:

- `Mdl` is a state-space model (ssm).
- `Y` is the observed response series.
- `params0` is the vector of initial values for unknown parameters.

`estimate` returns an estimated state-space model (`EstMdl`), which stores the estimated coefficient matrices and initial state means, covariance matrices, and distributions.

- For explicitly created state-space models, the software estimates all `NaN` values in the coefficient matrices (`Mdl.A`, `Mdl.B`, `Mdl.C`, and `Mdl.D`) and the initial state means and covariance matrix (`Mdl.Mean0` and `Mdl.Cov0`). For details on explicit and implicit model creation, see ssm.
- For implicitly created state-space models, you specify the model structure and the location of the unknown parameters using the parameter-to-matrix mapping function. Implicitly create a state-space model to estimate complex models, impose parameter constraints, and estimate initial states. The parameter-to-mapping function can also accommodate additional output arguments.

`EstMdl = estimate(Mdl,Y,params0,Name,Value)` estimates the state-space model with additional options specified by one or more `Name,Value` pair arguments. For example, you can specify to deflate the observations by a linear regression using

predictor data, control how the results appear in the Command Window, and indicate which estimation method to use for the parameter covariance matrix.

`[EstMdl,estParams,EstParamCov,logL,Output] = estimate( ___ )` additionally returns:

- `estParams`, a vector containing the estimated parameters
- `EstParamCov`, the estimated variance-covariance matrix of the estimated parameters
- `logL`, the optimized loglikelihood value
- `Output`, optimization diagnostic information structure

using any of the input arguments in the previous syntaxes.

Tips

Constrained likelihood objective function maximization

- You can specify any combination of linear inequality, linear equality, and upper and lower bound constraints on the parameters.
- If a parameter is unbounded below, then set `'lb',-Inf`.
- If a parameter is unbounded above, then set `'ub',Inf`.
- It is good practice to avoid equality and inequality constraints during optimization. For example, if you want to constrain the parameter w to be positive, then implicitly specify the state-space model using a parameter-to-matrix mapping function, set $w = \exp(s)$ within the function, and use unconstrained optimization to estimate s. Subsequently, s can assume any real value, but w must be positive.

Predictors and corresponding coefficients

- The state-space model `Mdl` does not store the predictors (Z_t) nor their corresponding regression coefficients (β). Supply the predictors and their corresponding coefficients wherever necessary using the appropriate name-value pair arguments.
- The predictor series serve as observation deflators. Subsequently, the deflated data set is $Y_t - Z_t\beta$, where:
  - $Z_t = \begin{pmatrix} z_{1t} & z_{2t} & \cdots & z_{dt} \end{pmatrix}$., that is, Z is a T-by$d$ matrix.
  - z_{jt} is the period t value of predictor j.

  - β is a d-by-n matrix of regression coefficients.
- To include an overall mean to the observation model, include a column of 1s in Z_t.
- If you want to account for predictor effects when you simulate (simulate), then you must deflate the observations manually. To deflate the observations, use $W_t = Y_t - Z_t\hat{\beta}$.
- If the state equation requires predictors, then expand the states by the constant 1 and the predictors.
- If the regression model is complex, then consider implicitly defining the state space model. For example, define the parameter-to-matrix mapping function using the following syntax pattern.

```
function [A,B,C,D,Mean0,Cov0,StateType,DeflateY] = ParamMap(params,Y,Z)
  ...
  DeflateY = Y - exp(params(9) + params(10)*Z);
  ...
end
```

  In this example, `Y` is the matrix of observations and `Z` is the matrix of predictors. The function returns `DeflateY`, which is the matrix of deflated observations. Specify `Y` and `Z` in the MATLAB Workspace before, and then pass `ParamMap` to `ssm` using the following syntax pattern.

```
Mdl = ssm(@(params)ParamMap(params,Y,Z))
```

  This is also useful if each response series requires a distinct set of predictors.
- If the state equation requires known predictors, then include the predictors as additional state variables. Since predictor data varies with time, a state-space model with predictors as states is time varying.

Additional Tips

- The software accommodates missing data. Indicate missing data using `NaN` values in the observed responses (`Y`).
- It is good practice to check the convergence status of the optimization routine by displaying `Output.ExitFlag`.
- If the optimization algorithm does not converge, then you can increase the number of iterations using the `'Options'` name-value pair argument.
- If the optimization algorithm does not converge, then consider using refine, which might help you obtain better initial parameter values for optimization.

Input Arguments

`Mdl` — Standard state-space model

`ssm` model object

Standard state-space model containing unknown parameters, specified as an `ssm` model object returned by ssm.

`Mdl` does not store observed responses or predictor data. Supply the data wherever necessary, using the appropriate input and name-value pair arguments.

`Y` — Observed response data

numeric matrix | cell vector of numeric vectors

Observed response data to which `Mdl` is fit, specified as a numeric matrix or a cell vector of numeric vectors.

- If `Mdl` is time invariant with respect to the observation equation, then `Y` is a T-by-n matrix. Each row of the matrix corresponds to a period and each column corresponds to a particular observation in the model. Therefore, T is the sample size and n is the number of observations per period. The last row of `Y` contains the latest observations.
- If `Mdl` is time varying with respect to the observation equation, then `Y` is a T-by-1 cell vector. `Y{t}` contains an n_t-dimensional vector of observations for period t, where $t = 1,...,T$. The corresponding dimensions of the coefficient matrices in `Mdl.C{t}` and `Mdl.D{t}` must be consistent with the matrix in `Y{t}` for all periods. The last cell of `Y` contains the latest observations.
- Suppose that you created `Mdl` implicitly by specifying a parameter-to-matrix mapping function, and the function has input arguments for the observed responses or predictors. The mapping function establishes a link to observed responses and the predictor data in the MATLAB workspace, which overrides the value of `Y`.

`NaN` elements indicate missing observations. For details on how the Kalman filter accommodates missing observations, see "Algorithms" on page 9-829.

Data Types: `double | cell`

`params0` — Initial values of unknown parameters

numeric vector

Initial values of unknown parameters for numeric maximum likelihood estimation, specified as a numeric vector.

The elements of `params0` correspond to the unknown parameters in the state-space model matrices `A`, `B`, `C`, and `D`, and, optionally, the initial state mean `Mean0` and covariance matrix `Cov0`.

- If you created `Mdl` explicitly (that is, by specifying the matrices without a parameter-to-matrix mapping function), then the software maps the elements of `params` to `NaN`s in the state-space model matrices and initial state values. The software searches for `NaN`s column-wise, following the order `A`, `B`, `C`, `D`, `Mean0`, `Cov0`.
- If you created `Mdl` implicitly (that is, by specifying the matrices with a parameter-to-matrix mapping function), then set initial parameter values for the state-space model matrices, initial state values, and state types within the parameter-to-matrix mapping function.

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Estimation Options

`'Beta0'` — Initial values of regression coefficients

numeric matrix

Initial values of regression coefficients, specified as the comma-separated pair consisting of `'Beta0'` and a d-by-n numeric matrix. d is the number of predictor variables (see `Predictors`) and n is the number of observed response series (see `Y`).

By default, `Beta0` is the ordinary least-squares estimate of `Y` onto `Predictors`.

Data Types: `double`

`'CovMethod'` — Asymptotic covariance estimation method

`'opg'` (default) | `'hessian'` | `'sandwich'`

Asymptotic covariance estimation method, specified as the comma-separated pair consisting of `'CovMethod'` and a string.

Set `CovMethod` using a value in this table.

Value	Description
`'Hessian'`	Negative, inverted Hessian matrix
`'OPG'`	Outer product of gradients (OPG)
`'Sandwich'`	Both Hessian and OPG

Example: `'CovMethod','Sandwich'`

Data Types: `char`

`'Display'` — Command Window display option
`'params'` (default) | `'diagnostics'` | `'full'` | `'iter'` | `'off'` | cell vector of strings

Command Window display option, specified as the comma-separated pair consisting of `'Display'` and a string or cell vector of strings.

Set `Display` using any combination of values in this table.

Value	estimate Displays
`'diagnostics'`	Optimization diagnostics
`'full'`	Maximum likelihood parameter estimates, standard errors, *t* statistics, iterative optimization information, and optimization diagnostics
`'iter'`	Iterative optimization information
`'off'`	No display in the Command Window
`'params'`	Maximum likelihood parameter estimates, standard errors, and *t* statistics

For example:

- To run a simulation where you are fitting many models, and therefore want to suppress all output, use `'Display','off'`.
- To display all estimation results and the optimization diagnostics, use `'Display',{'params','diagnostics'}`.

Data Types: `char` | `cell`

`'Options'` — Optimization options

`optimoptions` optimization controller

Optimization options, specified as the comma-separated pair consisting of `'Options'` and an `optimoptions` optimization controller. `Options` replaces default optimization options of the optimizer. For details on altering default values of the optimizer, see the optimization controller `optimoptions`, the constrained optimization function `fmincon`, or the unconstrained optimization function `fminunc` in Optimization Toolbox.

For example, suppose that you want to change the constraint tolerance to `1e-6`. Set `Options = optimoptions(@fmincon,'TolCon',1e-6,'Algorithm','sqp')` and then pass `Options` into `estimate` using `'Options',Options`.

By default:

- For constrained optimization, `estimate` maximizes the likelihood objective function using `fmincon` and its default options, but sets `'Algorithm','interior-point'`.
- For unconstrained optimization, `estimate` maximizes the likelihood objective function using `fminunc` and its default options, but sets `'Algorithm','quasi-newton'`.

`'Predictors'` — Predictor data

[ ] (default) | numeric matrix

Predictor data used to deflate the observations in a time-invariant state-space model, specified as the comma-separated pair consisting of `'Predictors'` and a T-by-d numeric matrix. T is the number of periods and d is the number of predictor variables. Row t corresponds to the observed predictors at period t (Z_t) in the expanded observation equation

$$y_t - Z_t\beta = Cx_t + Du_t.$$

That is, the software deflates the observations using the regression component. β is the time-invariant vector of regression coefficients that the software estimates with all other parameters. `Predictors` and `Y` must have the same number of rows.

For n observations per period, the software regresses all predictor series onto each observation. Then, the software returns a d-by-n matrix of fitted regression coefficient vectors for each observation series.

If you specify `Predictors`, then `Mdl` must be time invariant. Otherwise, the software returns an error.

By default, the software excludes a regression component from the state-space model.

Data Types: `double`

`'SquareRoot'` — Square root filter method flag

`false` (default) | `true`

Square root filter method flag, specified as the comma-separated pair consisting of `'SquareRoot'` and `true` or `false`. If `true`, then `estimate` applies the square root filter method when implementing the Kalman filter.

If you suspect that the eigenvalues of the filtered state or forecasted observation covariance matrices are close to zero, then specify `'SquareRoot',true`. The square root filter is robust to numerical issues arising from finite the precision of calculations, but requires more computational resources.

Example: `'SquareRoot',true`

Data Types: `logical`

`'Tolerance'` — Forecast uncertainty threshold

`0` (default) | nonnegative scalar

Forecast uncertainty threshold, specified as the comma-separated pair consisting of `'Tolerance'` and a nonnegative scalar.

If the forecast uncertainty for a particular observation is less than `Tolerance` during numerical estimation, then the software removes the uncertainty corresponding to the observation from the forecast covariance matrix before its inversion.

It is best practice to set `Tolerance` to a small number, for example, `1e-15`, to overcome numerical obstacles during estimation.

Example: `'Tolerance',1e-15`

Data Types: `double`

`'Univariate'` — Univariate treatment of multivariate series flag

`false` (default) | `true`

Univariate treatment of a multivariate series flag, specified as the comma-separated pair consisting of `'Univariate'` and `true` or `false`. Univariate treatment of a multivariate series is also known as *sequential filtering*.

The univariate treatment can accelerate and improve numerical stability of the Kalman filter. However, all observation innovations must be uncorrelated. That is, D_tD_t' must be diagonal, where D_t, $t = 1,...,T$, is one of the following:

- The matrix `D{t}` in a time-varying state-space model
- The matrix `D` in a time-invariant state-space model

Example: `'Univariate',true`

Data Types: `logical`

Constrained Optimization Options for fmincon

`'Aeq'` — Linear equality constraint parameter transformer

matrix

Linear equality constraint parameter transformer for constrained likelihood objective function maximization, specified as the comma-separated pair consisting of `'Aeq'` and a matrix.

If you specify `Aeq` and `beq`, then `estimate` maximizes the likelihood objective function using the equality constraint $\texttt{Aeq}\theta = \texttt{beq}$, where θ is a vector containing every `Mdl` parameter.

The number of rows of `Aeq` is the number of constraints, and the number of columns is the number of parameters that the software estimates. Order the columns of `Aeq` by `Mdl.A`, `Mdl.B`, `Mdl.C`, `Mdl.D`, `Mdl.Mean0`, `Mdl.Cov0`, and the regression coefficient (if the model has one).

Specify `Aeq` and `beq` together, otherwise `estimate` returns an error.

`Aeq` directly corresponds to the input argument `Aeq` of `fmincon`, not to the state-transition coefficient matrix `Mdl.A`.

By default, if you did not specify any constraint (linear inequality, linear equality, or upper and lower bound), then `estimate` maximizes the likelihood objective function using unconstrained maximization.

`'Aineq'` — Linear inequality constraint parameter transformer

matrix

Linear inequality constraint parameter transformer for constrained likelihood objective function maximization, specified as the comma-separated pair consisting of `'Aineq'` and a matrix.

If you specify `Aineq` and `bineq`, then `estimate` maximizes the likelihood objective function using the inequality constraint $\texttt{Aineq}\theta \le \texttt{bineq}$, where θ is a vector containing every `Mdl` parameter.

The number of rows of `Aineq` is the number of constraints, and the number of columns is the number of parameters that the software estimates. Order the columns of `Aineq` by `Mdl.A`, `Mdl.B`, `Mdl.C`, `Mdl.D`, `Mdl.Mean0`, `Mdl.Cov0`, and the regression coefficient (if the model has one).

Specify `Aineq` and `bineq` together, otherwise `estimate` returns an error.

`Aineq` directly corresponds to the input argument `A` of `fmincon`, not to the state-transition coefficient matrix `Mdl.A`.

By default, if you did not specify any constraint (linear inequality, linear equality, or upper and lower bound), then `estimate` maximizes the likelihood objective function using unconstrained maximization.

Data Types: `double`

`'beq'` — Linear equality constraints of transformed parameters

numeric vector

Linear equality constraints of the transformed parameters for constrained likelihood objective function maximization, specified as the comma-separated pair consisting of `'beq'` and a numeric vector.

If you specify `Aeq` and `beq`, then `estimate` maximizes the likelihood objective function using the equality constraint $\texttt{Aeq}\theta = \texttt{beq}$, where θ is a vector containing every `Mdl` parameter..

Specify `Aeq` and `beq` together, otherwise `estimate` returns an error.

`beq` directly corresponds to the input argument `beq` of `fmincon`, and is not associated with any component of `Mdl`.

By default, if you did not specify any constraint (linear inequality, linear equality, or upper and lower bound), then `estimate` maximizes the likelihood objective function using unconstrained maximization.

Data Types: `double`

`'bineq'` — Linear inequality constraint upper bounds

numeric vector

Linear inequality constraint upper bounds of the transformed parameters for constrained likelihood objective function maximization, specified as the comma-separated pair consisting of `'bineq'` and a numeric vector.

If you specify `Aineq` and `bineq`, then `estimate` maximizes the likelihood objective function using the inequality constraint $\texttt{Aineq}\theta \le \texttt{bineq}$, where θ is a vector containing every `Mdl` parameter.

Specify `Aineq` and `bineq` together, otherwise `estimate` returns an error.

`bineq` directly corresponds to the input argument `b` of `fmincon`, and is not associated with any component of `Mdl`.

By default, if you did not specify any constraint (linear inequality, linear equality, or upper and lower bound), then `estimate` maximizes the likelihood objective function using unconstrained maximization.

Data Types: `double`

`'lb'` — Lower bounds of parameters

numeric vector

Lower bounds of the parameters for constrained likelihood objective function maximization, specified as the comma-separated pair consisting of `'lb'` and a numeric vector.

If you specify `lb` and `ub`, then `estimate` maximizes the likelihood objective function subject to $\texttt{lb} \le \theta \le \texttt{ub}$, where θ is a vector containing every `Mdl` parameter.

Order the elements of `lb` by `Mdl.A`, `Mdl.B`, `Mdl.C`, `Mdl.D`, `Mdl.Mean0`, `Mdl.Cov0`, and the regression coefficient (if the model has one).

By default, if you did not specify any constraint (linear inequality, linear equality, or upper and lower bound), then `estimate` maximizes the likelihood objective function using unconstrained maximization.

Data Types: `double`

'ub' — Upper bounds of parameters

numeric vector

Upper bounds of the parameters for constrained likelihood objective function maximization, specified as the comma-separated pair consisting of 'ub' and a numeric vector.

If you specify lb and ub, then estimate maximizes the likelihood objective function subject to $lb \le \theta \le ub$, where θ is a vector every Mdl parameter.

Order the elements of ub by Mdl.A, Mdl.B, Mdl.C, Mdl.D, Mdl.Mean0, Mdl.Cov0, and the regression coefficient (if the model has one).

By default, if you did not specify any constraint (linear inequality, linear equality, or upper and lower bound), then estimate maximizes the likelihood objective function using unconstrained maximization.

Data Types: double

Output Arguments

EstMdl — State-space model containing parameter estimates

ssm model object

State-space model containing the parameter estimates, returned as an ssm model object.

estimate uses maximum likelihood to calculate all parameter estimates. EstMdl stores the parameter estimates in the coefficient matrices (EstMdl.A, EstMdl.B, EstMdl.C, and EstMdl.D), and the initial state means and covariance matrix (EstMdl.Mean0 and EstMdl.Cov0), regardless of specifying Mdl explicitly. For the estimated regression coefficient, see estParams.

EstMdl does not store observed responses or predictor data. If you plan to filter (using filter), forecast (using forecast), or smooth (using smooth) using EstMdl, then you might need to supply the appropriate data.

estParams — Maximum likelihood estimates of model parameters

numeric vector

Maximum likelihood estimates of the model parameters known to the optimizer, returned as a numeric vector. estParams has the same dimensions as params0.

`estimate` arranges the estimates in `estParams` corresponding to unknown parameters in this order.

1. `EstMdl.A(:)`, that is, estimates in `EstMdl.A` listed column-wise
2. `EstMdl.B(:)`
3. `EstMdl.C(:)`
4. `EstMdl.D(:)`
5. `EstMdl.Mean0`
6. `EstMdl.Cov0(:)`
7. In models with predictors, estimated regression coefficients listed column-wise

`EstParamCov` — Variance-covariance matrix of maximum likelihood estimates

numeric matrix

Variance-covariance matrix of maximum likelihood estimates of the model parameters known to the optimizer, returned as a numeric matrix.

The rows and columns contain the covariances of the parameter estimates. The standard errors of the parameter estimates are the square root of the entries along the main diagonal.

`estimate` arranges the estimates in the rows and columns of `EstParamCov` corresponding to unknown parameters in this order.

1. `EstMdl.A(:)`, that is, estimates in `EstMdl.A` listed column-wise
2. `EstMdl.B(:)`
3. `EstMdl.C(:)`
4. `EstMdl.D(:)`
5. `EstMdl.Mean0`
6. `EstMdl.Cov0(:)`
7. In models with predictors, estimated regression coefficients listed column-wise

`logL` — Optimized loglikelihood value

numeric scalar

Optimized loglikelihood value, returned as a scalar.

Missing observations do not contribute to the loglikelihood.

`Output` — Optimization information

structure array

Optimization information, returned as a structure array.

This table describes the fields of `Output`.

Field	Description
`ExitFlag`	Optimization exit flag that describes the exit condition. For details, see `fmincon` and `fminunc`.
`Options`	Optimization options that the optimizer used for numerical estimation. For details, see `optimoptions`.

Data Types: `struct`

Examples

Fit Time-Invariant State-Space Model to Data

This example generates data from a known model, and then fits a state-space model to the data.

Suppose that a latent process is this AR(1) process

$$x_t = 0.5x_{t-1} + u_t,$$

where u_t is Gaussian with mean 0 and standard deviation 1.

Generate a random series of 100 observations from x_t, assuming that the series starts at 1.5.

```
T = 100;
ARMdl = arima('AR',0.5,'Constant',0,'Variance',1);
x0 = 1.5;
rng(1); % For reproducibility
x = simulate(ARMdl,T,'Y0',x0);
```

Suppose further that the latent process is subject to additive measurement error as indicated in the equation

$$y_t = x_t + \varepsilon_t,$$

where ε_t is Gaussian with mean 0 and standard deviation 0.1.

Use the random latent state process (`x`) and the observation equation to generate observations.

```
y = x + 0.1*randn(T,1);
```

Together, the latent process and observation equations compose a state-space model. Supposing that the coefficients and variances are unknown parameters, the state-space model is

$$\begin{aligned} x_t &= \phi x_{t-1} + \sigma_1 u_t \\ y_t &= x_t + \sigma_2 \varepsilon_t. \end{aligned}$$

Specify the state-transition matrix. Use `NaN` values for unknown parameters.

```
A = NaN;
```

Specify the state-disturbance-loading coefficient matrix.

```
B = NaN;
```

Specify the measurement-sensitivity coefficient matrix.

```
C = 1;
```

Specify the observation-innovation coefficient matrix

```
D = NaN;
```

Specify the state-space model using the coefficient matrices. Also, specify the initial state mean, variance, and distribution (which is stationary).

```
Mean0 = 0;
Cov0 = 10;
StateType = 0;
Mdl = ssm(A,B,C,D,'Mean0',Mean0,'Cov0',Cov0,'StateType',StateType);
```

`Mdl` is an `ssm` model. Verify that the model is correctly specified using the display in the Command Window.

Pass the observations to `estimate` to estimate the parameter. Set a starting value for the parameter to `params0`. σ_1 and σ_2 must be positive, so set the lower bound constraints using the `'lb'` name-value pair argument. Specify that the lower bound of ϕ is `-Inf`.

```
params0 = [0.9; 0.5; 0.1];
```

```
EstMdl = estimate(Mdl,y,params0,'lb',[-Inf; 0; 0])

Method: Maximum likelihood (fmincon)
Sample size: 100
Logarithmic  likelihood:     -140.532
Akaike   info criterion:      287.064
Bayesian info criterion:      294.879
      |     Coeff      Std Err   t Stat    Prob
-------------------------------------------------
 c(1) | 0.45425       0.19870   2.28612  0.02225
 c(2) | 0.89013       0.30359   2.93205  0.00337
 c(3) | 0.38750       0.57857   0.66976  0.50302
      |
      |  Final State   Std Dev   t Stat    Prob
 x(1) | 1.52989       0.35621   4.29496  0.00002

EstMdl =


State-space model type: <a href="matlab: doc ssm">ssm</a>

State vector length: 1
Observation vector length: 1
State disturbance vector length: 1
Observation innovation vector length: 1
Sample size supported by model: Unlimited

State variables: x1, x2,...
State disturbances: u1, u2,...
Observation series: y1, y2,...
Observation innovations: e1, e2,...

State equation:
x1(t) = (0.45)x1(t-1) + (0.89)u1(t)

Observation equation:
y1(t) = x1(t) + (0.39)e1(t)

Initial state distribution:

Initial state means
 x1
  0
```

```
Initial state covariance matrix
     x1
 x1  10

State types
     x1
 Stationary
```

`EstMdl` is an `ssm` model. The results of the estimation appear in the Command Window, contain the fitted state-space equations, and contain a table of parameter estimates, their standard errors, t statistics, and p-values.

You can use or display, for example the fitted state-transition matrix using dot notation.

```
EstMdl.A
```

```
ans =

    0.4543
```

Pass `EstMdl` to `forecast` to forecast observations, or to `simulate` to conduct a Monte Carlo study.

Estimate State-Space Model Containing Regression Component

Suppose that the linear relationship between the change in the unemployment rate and the nominal gross national product (nGNP) growth rate is of interest. Suppose further that the first difference of the unemployment rate is an ARMA(1,1) series. Symbolically, and in state-space form, the model is

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix} = \begin{bmatrix} \phi & \theta \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix} u_{1,t}$$
$$y_t - \beta Z_t = x_{1,t} + \sigma \varepsilon_t,$$

where:

- $x_{1,t}$ is the change in the unemployment rate at time t.
- $x_{2,t}$ is a dummy state for the MA(1) effect.
- $y_{1,t}$ is the observed change in the unemployment being deflated by the growth rate of nGNP (Z_t).

- $u_{1,t}$ is the Gaussian series of state disturbances having mean 0 and standard deviation 1.
- ε_t is the Gaussian series of observation innovations having mean 0 and standard deviation σ.

Load the Nelson-Plosser data set, which contains the unemployment rate and nGNP series, among other things.

```
load Data_NelsonPlosser
```

Preprocess the data by taking the natural logarithm of the nGNP series, and the first difference of each. Also, remove the starting `NaN` values from each series.

```
isNaN = any(ismissing(DataTable),2);       % Flag periods containing NaNs
gnpn = DataTable.GNPN(~isNaN);
u = DataTable.UR(~isNaN);
T = size(gnpn,1);                          % Sample size
Z = [ones(T-1,1) diff(log(gnpn))];
y = diff(u);
```

This example proceeds using series without `NaN` values. However, using the Kalman filter framework, the software can accommodate series containing missing values.

Specify the state-transition coefficient matrix.

```
A = [NaN NaN; 0 0];
```

Specify the state-disturbance-loading coefficient matrix.

```
B = [1; 1];
```

Specify the measurement-sensitivity coefficient matrix.

```
C = [1 0];
```

Specify the observation-innovation coefficient matrix.

```
D = NaN;
```

Specify the state-space model using `ssm`.

```
Mdl = ssm(A,B,C,D);
```

Estimate the model parameters. Specify the regression component and its initial value for optimization using the `'Predictors'` and `'Beta0'` name-value pair arguments,

respectively. Display the estimates and all optimization diagnostic information. Restrict the estimate of σ to all positive, real numbers.

```
params0 = [0.3 0.2 0.1]; % Chosen arbitrarily
EstMdl = estimate(Mdl,y,params0,'Predictors',Z,'Display','full',...
    'Beta0',[0.1 0.2],'lb',[-Inf,-Inf,0,-Inf,-Inf]);
```

```
_____________________________________________________________
   Diagnostic Information

Number of variables: 5

Functions
Objective:                              @(c)-fML(c,Mdl,Y,Predictors,unitFlag,sqrtFlag,mex
Gradient:                               finite-differencing
Hessian:                                finite-differencing (or Quasi-Newton)

Constraints
Nonlinear constraints:                  do not exist

Number of linear inequality constraints:    0
Number of linear equality constraints:      0
Number of lower bound constraints:          1
Number of upper bound constraints:          0

Algorithm selected
   interior-point


_____________________________________________________________
   End diagnostic information
                                                First-order      Norm of
 Iter F-count            f(x)  Feasibility   optimality         step
    0       6    2.579611e+02    0.000e+00    4.601e+01
    1      20    2.556482e+02    0.000e+00    3.652e+01    1.392e-01
    2      27    2.503349e+02    0.000e+00    4.319e+01    1.908e-01
    3      35    2.379649e+02    0.000e+00    1.294e+01    1.083e+01
    4      41    1.946860e+02    0.000e+00    1.948e+01    7.164e+00
    5      47    1.602292e+02    0.000e+00    2.126e+02    1.184e+01
    6      53    1.258094e+02    0.000e+00    9.559e+01    1.590e+00
    7      59    1.107064e+02    0.000e+00    1.145e+01    2.533e+00
    8      65    1.040826e+02    0.000e+00    8.196e+00    1.591e+00
    9      72    1.034635e+02    0.000e+00    8.882e+00    1.003e+00
   10      79    1.013796e+02    0.000e+00    2.783e+00    1.840e+00
```

```
   11       85    1.004734e+02    0.000e+00    2.993e+00    1.105e+00
   12       91    9.981209e+01    0.000e+00    8.294e-01    2.173e+00
   13       97    9.974121e+01    0.000e+00    8.724e-01    5.833e-01
   14      103    9.973856e+01    0.000e+00    9.659e-01    9.537e-02
   15      109    9.973481e+01    0.000e+00    7.712e-01    5.607e-02
   16      115    9.973404e+01    0.000e+00    7.017e-01    3.579e-03
   17      121    9.973202e+01    0.000e+00    5.113e-01    2.047e-02
   18      127    9.973083e+01    0.000e+00    4.082e-01    1.825e-02
   19      133    9.972930e+01    0.000e+00    2.866e-01    2.316e-02
   20      139    9.972742e+01    0.000e+00    3.040e-01    2.026e-02
   21      145    9.972574e+01    0.000e+00    2.568e-01    1.648e-02
   22      151    9.972535e+01    0.000e+00    1.275e-01    3.594e-02
   23      157    9.972524e+01    0.000e+00    1.000e-01    2.590e-02
   24      163    9.972456e+01    0.000e+00    2.317e-02    1.184e-02
   25      169    9.972457e+01    0.000e+00    2.000e-02    2.439e-03
   26      175    9.972454e+01    0.000e+00    1.295e-03    1.943e-03
   27      181    9.972454e+01    0.000e+00    2.000e-04    2.015e-04
   28      187    9.972454e+01    0.000e+00    2.575e-05    2.796e-05
   29      193    9.972454e+01    0.000e+00    6.676e-06    2.778e-06
   30      199    9.972454e+01    0.000e+00    6.676e-06    7.382e-07

                                                First-order      Norm of
 Iter F-count            f(x)  Feasibility   optimality         step
   31      205    9.972454e+01    0.000e+00    3.633e-06    1.771e-07
   32      211    9.972454e+01    0.000e+00    2.000e-06    3.501e-07
   33      217    9.972454e+01    0.000e+00    1.907e-06    1.416e-07
   34      225    9.972454e+01    0.000e+00    2.861e-06    1.425e-07
   35      232    9.972454e+01    0.000e+00    3.815e-06    1.567e-07
   36      245    9.972454e+01    0.000e+00    2.802e-06    3.963e-08

Local minimum possible. Constraints satisfied.

fmincon stopped because the size of the current step is less than
the default value of the step size tolerance and constraints are
satisfied to within the default value of the constraint tolerance.



Method: Maximum likelihood (fmincon)
Sample size: 61
Logarithmic  likelihood:     -99.7245
Akaike   info criterion:      209.449
Bayesian info criterion:      220.003
```

```
           |      Coeff       Std Err    t Stat     Prob
-------------------------------------------------------------
 c(1)      |   -0.34098       0.29608    -1.15164  0.24948
 c(2)      |    1.05003       0.41377     2.53771  0.01116
 c(3)      |    0.48592       0.36790     1.32079  0.18657
 y <- z(1) |    1.36121       0.22338     6.09358   0
 y <- z(2) |  -24.46711       1.60018   -15.29024   0
           |
           |    Final State   Std Dev     t Stat    Prob
 x(1)      |    1.01264       0.44690     2.26592  0.02346
 x(2)      |    0.77718       0.58917     1.31912  0.18713
```

Optimization information and a table of estimates and statistics output to the Command Window. `EstMdl` is an `ssm` model, and you can access its properties using dot notation.

Compare Estimates from State-Space Model Filtering Methods

The software implements the Kalman filter using the covariance filter by default, but you can specify to use the square-root filter instead. This example compares estimates from each method using simulated data.

Suppose that a latent process is an AR(1). Subsequently, the state equation is

$$x_t = 0.5x_{t-1} + u_t,$$

where u_t is Gaussian with mean 0 and standard deviation 0.3.

Generate a random series of 100 observations from x_t, assuming that the series starts at 1.5.

```
T = 100;
ARMdl = arima('AR',0.5,'Constant',0,'Variance',0.3^2);
x0 = 1.5;
rng(1); % For reproducibility
x = simulate(ARMdl,T,'Y0',x0);
```

Suppose further that the latent process is subject to additive measurement error. Subsequently, the observation equation is

$$y_t = x_t + \varepsilon_t,$$

where ε_t is Gaussian with mean 0 and standard deviation 0.1.

Use the random latent state process (`x`) and the observation equation to generate observations.

```
y = x + 0.1*randn(T,1);
```

Together, the latent process and observation equations compose a state-space model. Supposing that the coefficients and variances are unknown parameter, the state-space model is

$$x_t = \phi x_{t-1} + \sigma_1 u_t$$
$$y_t = x_t + \sigma_2 \varepsilon_t$$

Specify the state-transition coefficient matrix. Use `NaN` values for unknown parameters.

```
A = NaN;
```

Specify the state-disturbance-loading coefficient matrix.

```
B = NaN;
```

Specify the measurement-sensitivity coefficient matrix.

```
C = 1;
```

Specify the observation-innovation coefficient matrix.

```
D = NaN;
```

Specify the state-space model using the coefficient matrices. Also, specify the initial state mean, variance, and distribution (which is stationary).

```
Mean0 = 0;
Cov0 = 10;
StateType = 0;
Mdl = ssm(A,B,C,D,'Mean0',Mean0,'Cov0',Cov0,'StateType',StateType);
```

`Mdl` is an `ssm` model.

Estimate the parameters using `estimate` two ways:

- Using the default, simple Kalman filter
- Using the square root filter variation

In both cases, specify that no output should be returned to the Command Window. This is good practice if you plan on running `estimate` multiple times (such as a Monte Carlo simulation).

```
params0 = [10,10,10];
[~,estParamsSKF,EstParamCovSKF,logLSKF,OutputSKF] = estimate(Mdl,y,params0,...
```

```
    'Display','off');
[~,estParamsSR,EstParamCovSR,logLSR,OutputSR] = estimate(Mdl,y,params0,...
    'Squareroot',true,'Display','off');
```

Check that the algorithms converged properly by printing the exit flag properties of `OutputSKF` and `OutputSR`.

```
exitFlagSKF = OutputSKF.ExitFlag
exitFlagSR = OutputSR.ExitFlag
```

```
exitFlagSKF =

     1


exitFlagSR =

     1
```

Both algorithms have an exit flag of `1`, which indicates that the software met the convergence criteria.

Compare the estimates from each algorithm.

```
fprintf('\n Parameter Estimates\n')
table(estParamsSKF',estParamsSR','VariableNames',...
    {'SimpleKalmanFilter','SquarerootFilter'})
fprintf('\nEstimated Parameter Covariance Matrix\n')
table(EstParamCovSKF,EstParamCovSR,'VariableNames',...
    {'SimpleKalmanFilter','SquarerootFilter'})
```

```
 Parameter Estimates

ans =

    SimpleKalmanFilter    SquarerootFilter
    __________________    ________________

     0.51057               0.51057
     0.23436               0.23436
    -0.17904              -0.17904
```

```
Estimated Parameter Covariance Matrix

ans =

           SimpleKalmanFilter                        SquarerootFilter
    ___________________________________    ___________________________________

     0.036669     -0.013302     -0.014012     0.036669     -0.013302     -0.014012
    -0.013302      0.0070187     0.0072533   -0.013302      0.0070187     0.0072533
    -0.014012      0.0072533     0.0089019   -0.014012      0.0072533     0.0089019
```

In this case, the results are the same.

If you use the default, covariance filter method, and you run into numerical problems during estimation, filtering, or smoothing, try using the squareroot method.

- "Estimate Time-Varying State-Space Model" on page 8-45
- "Estimate Random Parameter of State-Space Model" on page 8-116
- "Assess State-Space Model Stability Using Rolling Window Analysis" on page 8-172
- "Choose State-Space Model Specification Using Backtesting" on page 8-181

Algorithms

- The Kalman filter accommodates missing data by not updating filtered state estimates corresponding to missing observations. In other words, suppose there is a missing observation at period t. Then, the state forecast for period t based on the previous $t-1$ observations and filtered state for period t are equivalent.
- For explicitly created state-space models, `estimate` applies all predictors to each response series. However, each response series has its own set of regression coefficients.
- If you do not specify optimization constraints, then `estimate` uses `fminunc` for unconstrained numerical estimation. If you specify any pair of optimization constraints, then `estimate` uses `fmincon` for constrained numerical estimation. For either type of optimization, optimization options you set using the name-value pair argument `Options` must be consistent with the options of the optimization algorithm.
- `estimate` passes the name-value pair arguments `Options`, `Aineq`, `bineq`, `Aeq`, `beq`, `lb`, and `ub` directly to the optimizer `fmincon` or `fminunc`.

- `estimate` fits regression coefficients along with all other state-space model parameters. The software is flexible enough to allow applying constraints to the regression coefficients using constrained optimization options. For more details, see the `Name,Value` pair arguments and `fmincon`.
- If you set `'Univariate',true`, then, during the filtering algorithm, the software sequentially updates rather then updating all at once. This might accelerate parameter estimation, especially for a low-dimensional, time-invariant model.
- Suppose that you want to create a state-space model using a parameter-to-matrix mapping function with this signature

  ```
  [A,B,C,D,Mean0,Cov0,StateType,DeflateY] = paramMap(params,Y,Z)
  ```
  and you specify the model using an anonymous function

  ```
  Mdl = ssm(@(params)paramMap(params,Y,Z))
  ```
  The observed responses `Y` and predictor data `Z` are not input arguments in the anonymous function. If `Y` and `Z` exist in the MATLAB Workspace before creating `Mdl`, then the software establishes a link to them. Otherwise, if you pass `Mdl` to `estimate`, the software throws an error.

  The link to the data established by the anonymous function overrides all other corresponding input argument values of `estimate`. This distinction is important particularly when conducting a rolling window analysis. For details, see "Rolling-Window Analysis of Time-Series Models" on page 8-168.

Limitations

If the model is time varying with respect the observed responses, then the software does not support including predictors. If the observation vectors among different periods vary in length, then the software cannot determine which coefficients to use to deflate the observed responses.

References

[1] Durbin J., and S. J. Koopman. *Time Series Analysis by State Space Methods*. 2nd ed. Oxford: Oxford University Press, 2012.

See Also

filter | `fmincon` | `fminunc` | forecast | `optimoptions` | refine | simulate | smooth | ssm

More About

- "What Are State-Space Models?" on page 8-3
- "What Is the Kalman Filter?" on page 8-8
- "Rolling-Window Analysis of Time-Series Models" on page 8-168

fgls

Feasible generalized least squares

Syntax

```
coeff = fgls(X,y)

coeff = fgls(Tbl)

coeff = fgls( ___ ,Name,Value)

[coeff,se,EstCoeffCov] = fgls( ___ )
```

Description

`coeff = fgls(X,y)` returns coefficient estimates (`coeff`) of multiple linear regression models $y = X\beta + \varepsilon$ using feasible generalized least squares (FGLS) by first estimating the covariance of the innovations process ε.

`NaN`s in the data indicate missing values, which `fgls` removes using list-wise deletion. `fgls` sets `Data = [X y]`, then it removes any row in `Data` containing at least one `NaN`. This reduces the effective sample size, and changes the time base of the series.

`coeff = fgls(Tbl)` returns FGLS coefficient estimates (`coeff`), with predictor data in the first `numPreds` columns of the tabular array, `Tbl`, and response data in the last column.

`fgls` removes all missing values in `Tbl`, indicated by `NaN`s, using list-wise deletion. In other words, `fgls` removes all rows in `Tbl` containing at least one `NaN`. This reduces the effective sample size, and changes the time base of the series.

`coeff = fgls( ___ ,Name,Value)` uses any of the input arguments in the previous syntaxes and additional options specified by one or more `Name,Value` pair arguments.

For example, use `Name,Value` pair arguments to choose the innovations covariance model, number of iterations, or to plot estimates after each iteration.

`[coeff,se,EstCoeffCov] = fgls( ___ )` additionally returns a vector of FGLS coefficient standard errors, `se` = `sqrt(diag(EstCov))`, and the FGLS estimated coefficient covariance matrix (`EstCoeffCov`).

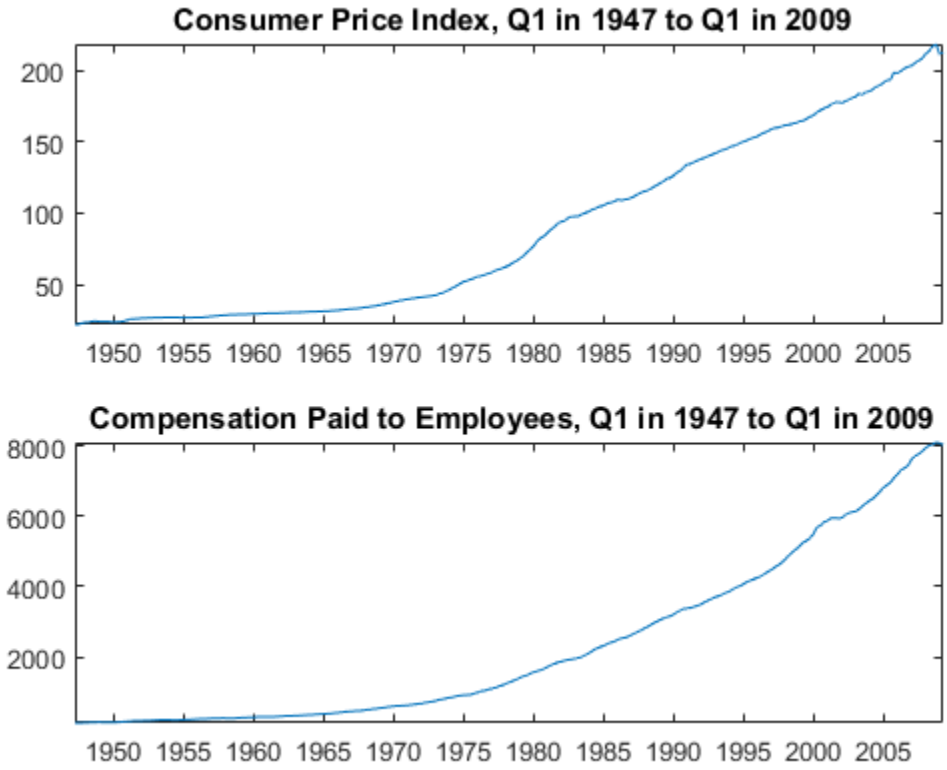
Examples

Estimate FGLS Coefficients Using Default Options

Suppose the sensitivity of the U.S. Consumer Price Index (CPI) to changes in the paid compensation of employees (COE) is of interest.

Load the US macroeconomic data set. Plot the CPI and COE series.

```
load Data_USEconModel

figure;
subplot(2,1,1)
plot(dates,DataTable.CPIAUCSL);
title '{\bf Consumer Price Index, Q1 in 1947 to Q1 in 2009}';
datetick;
axis tight;
subplot(2,1,2);
plot(dates,DataTable.COE);
title '{\bf Compensation Paid to Employees, Q1 in 1947 to Q1 in 2009}';
datetick;
axis tight;
```

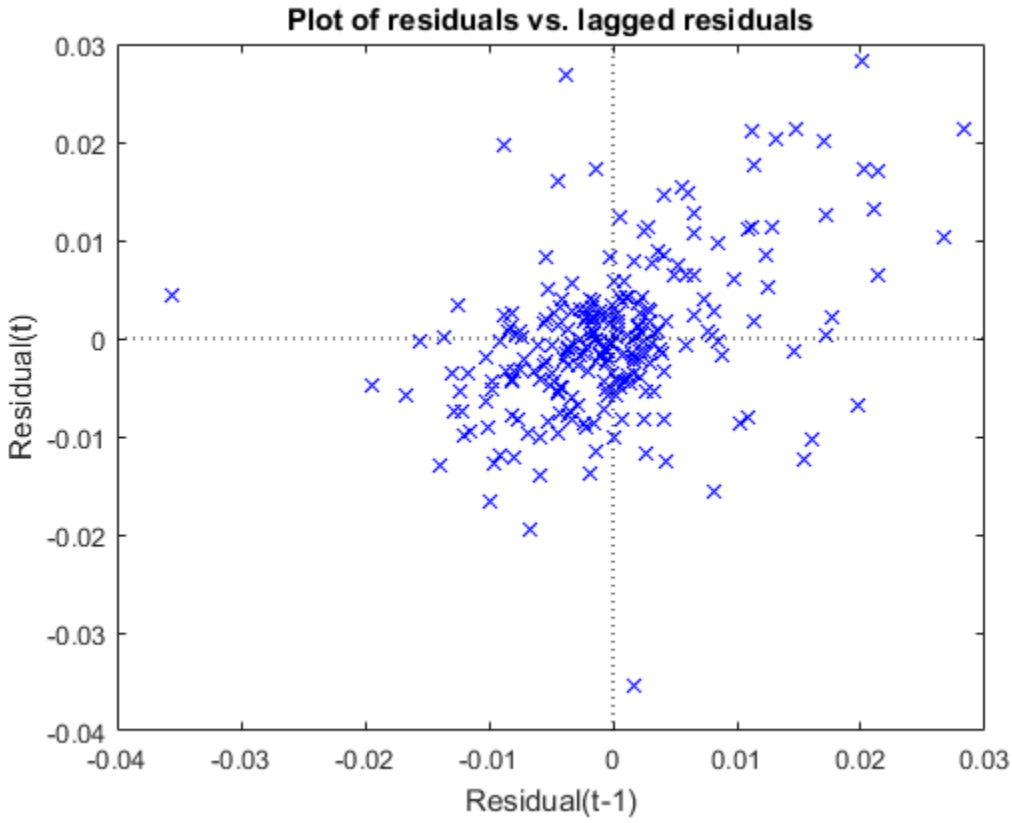


The series are nonstationary. Stabilize them by applying the log, and then the first difference.

```
CPI = diff(log(DataTable.CPIAUCSL));
COE = diff(log(DataTable.COE));
```

Regress `CPI` onto `COE` including an intercept to obtain ordinary least squares (OLS) estimates. Generate a lagged residual plot.

```
Mdl = fitlm(COE,CPI);

figure;
plotResiduals(Mdl,'lagged')
```



There is an upward trend in the residual plot, which suggests that the innovations comprise an autoregressive process. This violates one of the classical linear model assumptions. Consequently, hypothesis tests based on the regression coefficients are incorrect, even asymptotically.

Estimate the regression coefficients using FGLS. By default, `fgls` includes an intercept in the regression model and imposes an AR(1) model on the innovations. Optionally, display the OLS and FGLS estimates by specifying `'final'` for the `'display'` name-value pair argument.

```
coeff = fgls(CPI,COE,'display','final');
```

```
OLS Estimates:

       |  Coeff    SE
-----------------------
 Const | 0.0122  0.0009
 x1    | 0.4915  0.0686

FGLS Estimates:

       |  Coeff    SE
-----------------------
 Const | 0.0148  0.0012
 x1    | 0.1961  0.0685
```

If the COE series is exogenous with respect to the CPI, then the FGLS estimates (`coeff`) are consistent and asymptotically more efficient than the OLS estimates.

Specify AR Lags When Estimating FGLS Coefficients and Standard Errors

Suppose the sensitivity of the U.S. Consumer Price Index (CPI) to changes in the paid compensation of employees (COE) is of interest. This example enhances the analysis outlined in the example "Estimate FGLS Coefficients Using Default Options".

Load the U.S. macroeconomic data set.

```
load Data_USEconModel
```

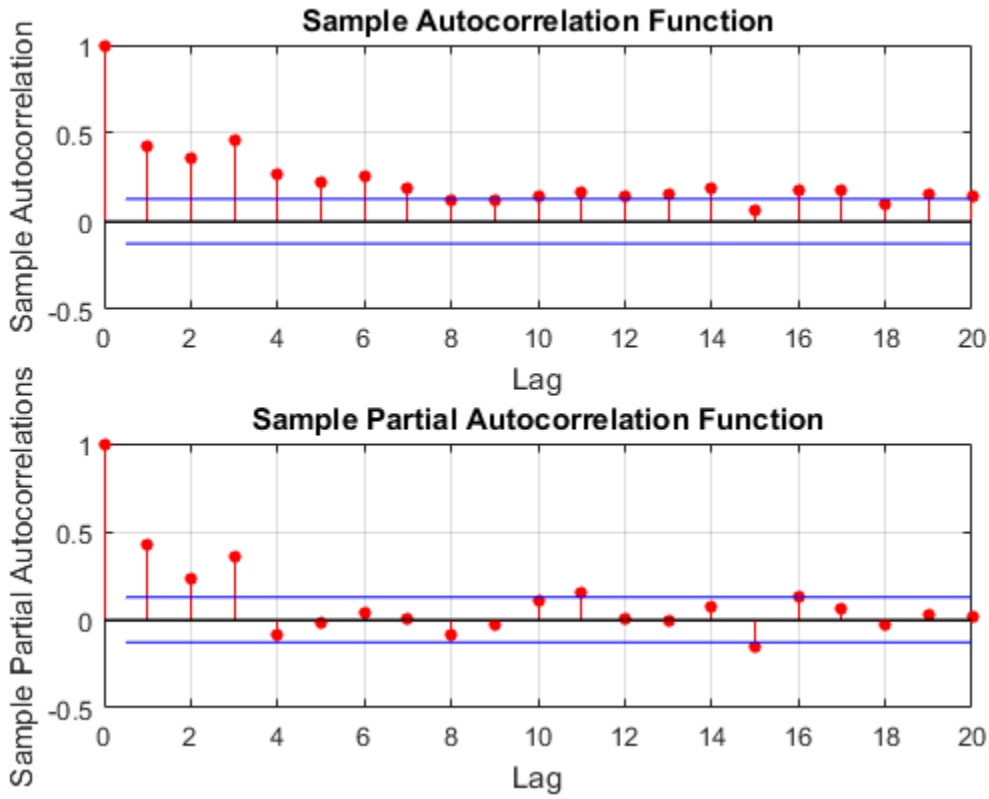
The series are nonstationary. Stabilize them by applying the log, and then the first difference.

```
CPI = diff(log(DataTable.CPIAUCSL));
COE = diff(log(DataTable.COE));
```

Regress `CPI` onto `COE` including an intercept to obtain OLS estimates. Plot correlograms for the residuals.

```
Mdl = fitlm(COE,CPI);
u = Mdl.Residuals.Raw;

figure;
subplot(2,1,1)
autocorr(u);
subplot(2,1,2);
```

```
parcorr(u);
```



The correlograms suggest that the innovations have significant AR effects. According to "Box-Jenkins Methodology", the innovations seem to comprise an AR(3) series.

Estimate the regression coefficients using FGLS. By default, `fgls` assumes that the innovations are autoregressive. Specify that the innovations are AR(3) using the `'arLags'` name-value pair argument.

```
[coeff,se] = fgls(CPI,COE,'arLags',3,'display','final');
```

```
OLS Estimates:
```

```
       |  Coeff    SE
-----------------------
 Const | 0.0122  0.0009
 x1    | 0.4915  0.0686

FGLS Estimates:

       |  Coeff    SE
-----------------------
 Const | 0.0148  0.0012
 x1    | 0.1972  0.0684
```

If the COE series is exogenous with respect to the CPI, then the FGLS estimates (`coeff`) are consistent and asymptotically more efficient than the OLS estimates.

Account for Residual Heteroscedasticity Using FGLS Estimation

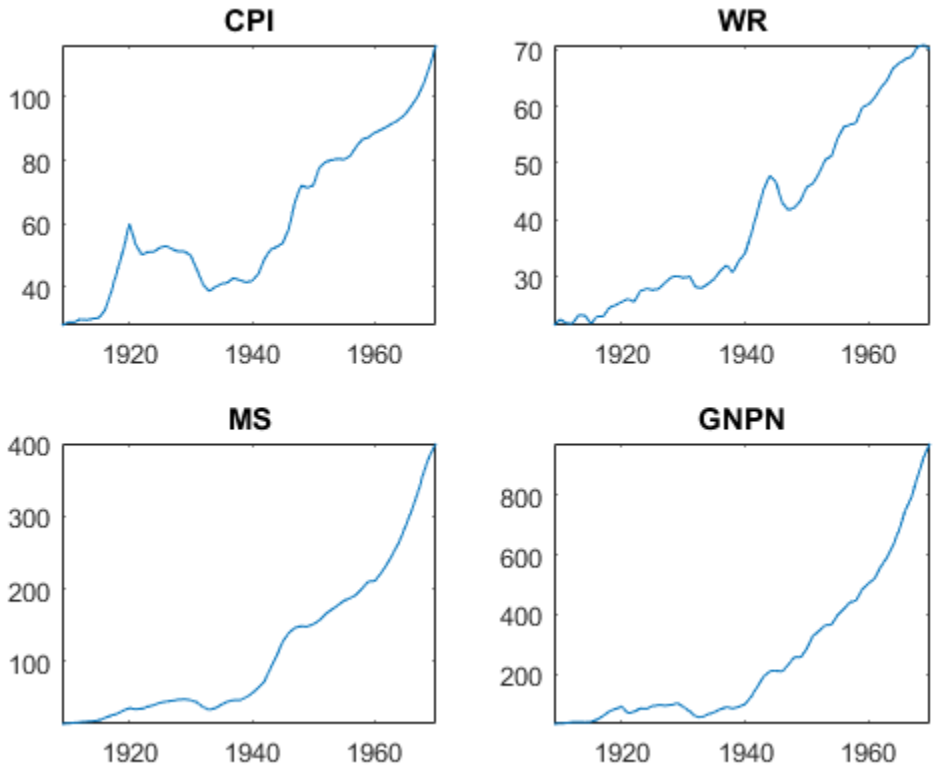
Model the nominal GNP (`GNPN`) growth rate accounting for the effects of the growth rates of the consumer price index (`CPI`), real wages (`WR`), and the money stock (`MS`). Account for classical linear model departures.

Load the Nelson Plosser data set.

```
load Data_NelsonPlosser
varIdx = [8,10,11,2];               % Variable indices
idx = ~any(ismissing(DataTable),2); % Identify nonmissing values
Tbl = DataTable(idx,varIdx);        % Tabular array of variables
T = sum(idx);                       % Sample size
```

Plot the series.

```
figure;
for j = 1:4;
    subplot(2,2,j);
    plot(dates(idx),Tbl{:,j});
    title(Tbl.Properties.VariableNames{j});
    axis tight;
end;
```



All series appear nonstationary.

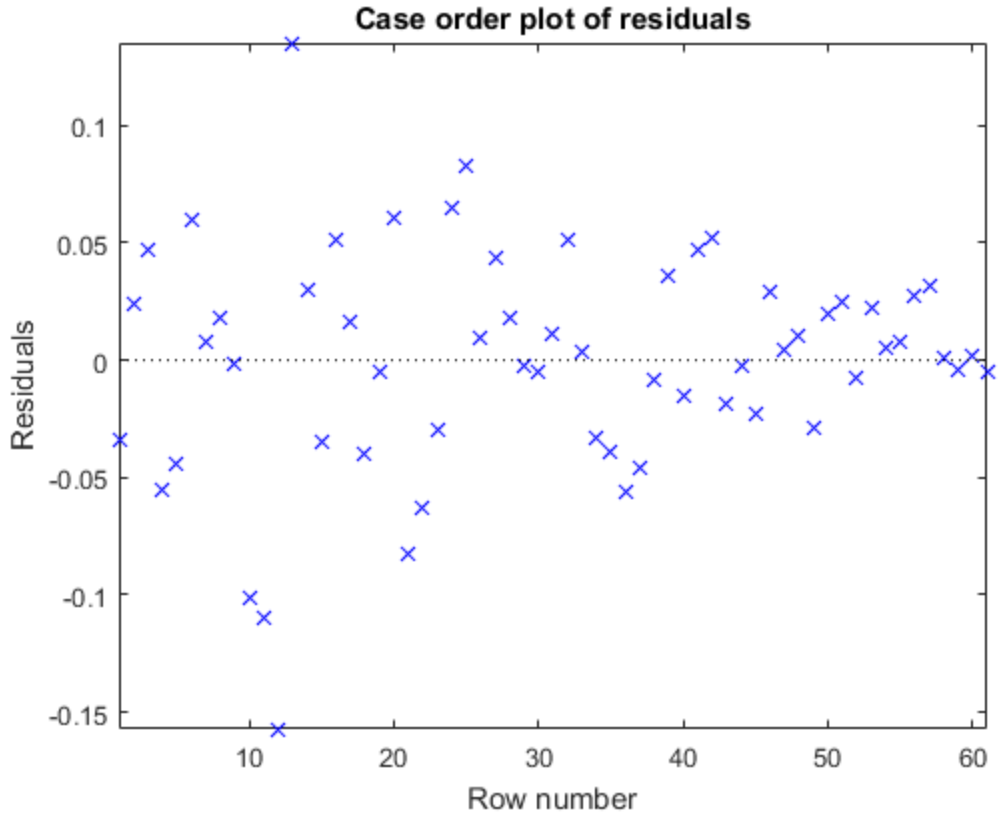
Apply the log, and then the first difference to each series.

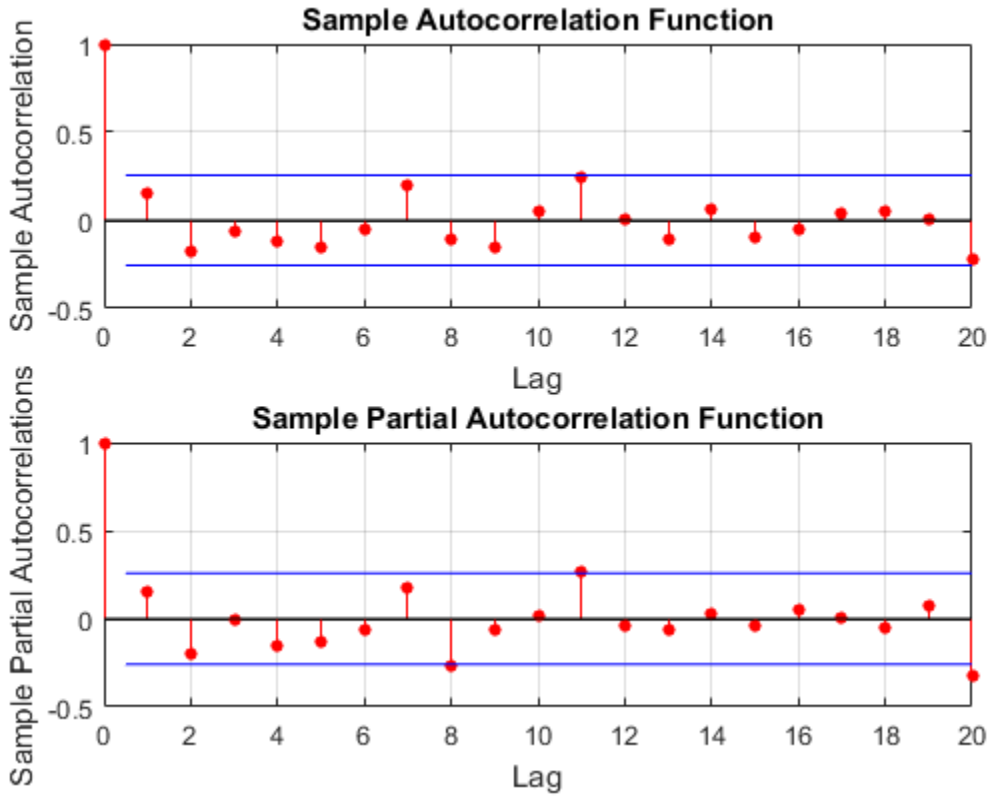
```
dLogTbl = array2table(diff(log(Tbl{:,:})),...
    'VariableNames',strcat(Tbl.Properties.VariableNames,'Rate'));
```

Regress `GNPNRate` onto the other variables in `dLogTbl`. Examine a scatter plot and correlograms of the residuals.

```
Mdl = fitlm(dLogTbl);

figure;
plotResiduals(Mdl,'caseorder');
```

```
axis tight;

figure;
subplot(2,1,1);
autocorr(Mdl.Residuals.Raw);
subplot(2,1,2);
parcorr(Mdl.Residuals.Raw);
```





The residuals appear to flare in, and so they exhibit heteroscedasticity. The correlograms suggest that there is no autocorrelation.

Estimate FGLS coefficients by accounting for the heteroscedasticity of the residuals. Specify that the estimated innovation covariance is diagonal with the squared residuals as weights.

```
fgls(dLogTbl,'innovMdl','HC0','display','final');
```

```
OLS Estimates:

          |  Coeff     SE
--------------------------
```

```
 Const    | -0.0076  0.0085
 CPIRate  |  0.9037  0.1544
 WRRate   |  0.9036  0.1906
 MSRate   |  0.4285  0.1379

FGLS Estimates:

          |  Coeff     SE
---------------------------
 Const    | -0.0102  0.0017
 CPIRate  |  0.8853  0.0169
 WRRate   |  0.8897  0.0294
 MSRate   |  0.4874  0.0291
```

Estimate FGLS Coefficients of Models Containing ARMA Errors

Create this regression model with ARMA(1,2) errors, where ε_t is Gaussian with mean 0 and variance 1.

$$y_t = 1 + x_t \begin{bmatrix} 2 \\ 3 \end{bmatrix} + u_t$$
$$u_t = 0.6u_{t-1} + \varepsilon_t - 0.3\varepsilon_{t-1} + 0.1\varepsilon_{t-1}.$$

```
beta = [2 3];
phi = 0.2;
theta = [-0.3 0.1];
Mdl = regARIMA('AR',phi,'MA',theta,'Intercept',1,'Beta',beta,'Variance',1);
```

`Mdl` is a `regARIMA` model. You can access its properties using dot notation.

Simulate 500 periods of 2-D standard Gaussian values for x_t, and then simulate responses using `Mdl`.

```
numObs = 500;
rng(1); % For reproducibility
X = randn(numObs,2);
y = simulate(Mdl,numObs,'X',X);
```

`fgls` supports AR(p) innovation models. You can convert an ARMA model polynomial to an infinite-lag AR model polynomial using arma2ar. By default, `arma2ar` returns the coefficients for the first 10 terms. After the conversion, determine how many lags of the resulting AR model are practically significant by checking the length of the returned vector of coefficients. Choose the number of terms that exceed 0.00001.

```
format long
arParams = arma2ar(phi,theta)
arLags = sum(abs(arParams) > 0.00001);
format short
```

```
arParams =

  -0.100000000000000   0.070000000000000   0.031000000000000
```

Some of the parameters have small magnitude. You might want to reduce the number of lags to include in the innovations model for `fgls`.

Estimate the coefficients and their standard errors using FGLS and the simulated data. Specify that the innovations comprise an AR(`arLags`) process.

```
[coeff,~,EstCoeffCov] = fgls(X,y,'innovMdl','AR','arLags',arLags)
```

```
coeff =

    1.0372
    2.0366
    2.9918


EstCoeffCov =

    0.0026   -0.0000    0.0001
   -0.0000    0.0022    0.0000
    0.0001    0.0000    0.0024
```

The estimated coefficients are close to their true values.

Estimate Linear Model Coefficients Using Iterative FGLS

This example expands on the analysis in "Estimate FGLS Coefficients of Models Containing ARMA Errors". Create this regression model with ARMA(1,2) errors, where ε_t is Gaussian with mean 0 and variance 1.

$$y_t = 1 + x_t \begin{bmatrix} 2 \\ 3 \end{bmatrix} + u_t$$
$$u_t = 0.6u_{t-1} + \varepsilon_t - 0.3\varepsilon_{t-1} + 0.1\varepsilon_{t-1}.$$

```
beta = [2 3];
phi = 0.2;
theta = [-0.3 0.1];
Mdl = regARIMA('AR',phi,'MA',theta,'Intercept',1,'Beta',beta,'Variance',1);
```

Simulate 500 periods of 2-D standard Gaussian values for x_t, and then simulate responses using `Mdl`.

```
numObs = 500;
rng(1); % For reproducibility
X = randn(numObs,2);
y = simulate(Mdl,numObs,'X',X);
```

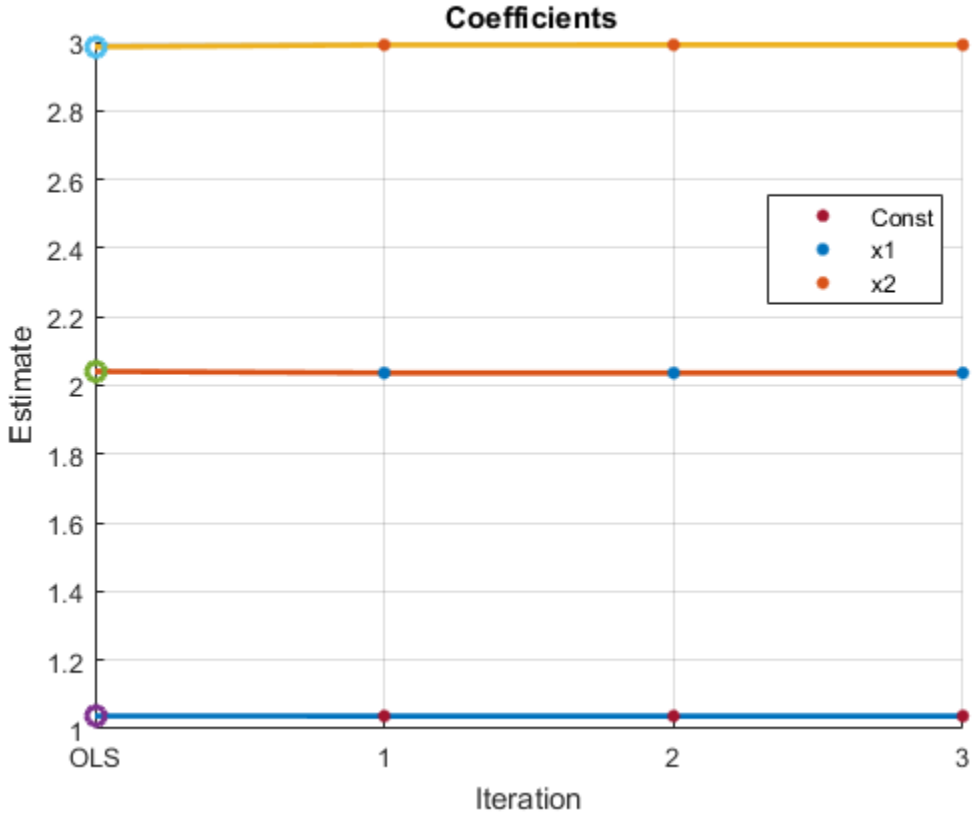
Convert the ARMA model polynomial to an infinite-lag AR model polynomial using arma2ar. By default, `arma2ar` returns the coefficients for the first 10 terms. Find the number of terms that exceed 0.00001.

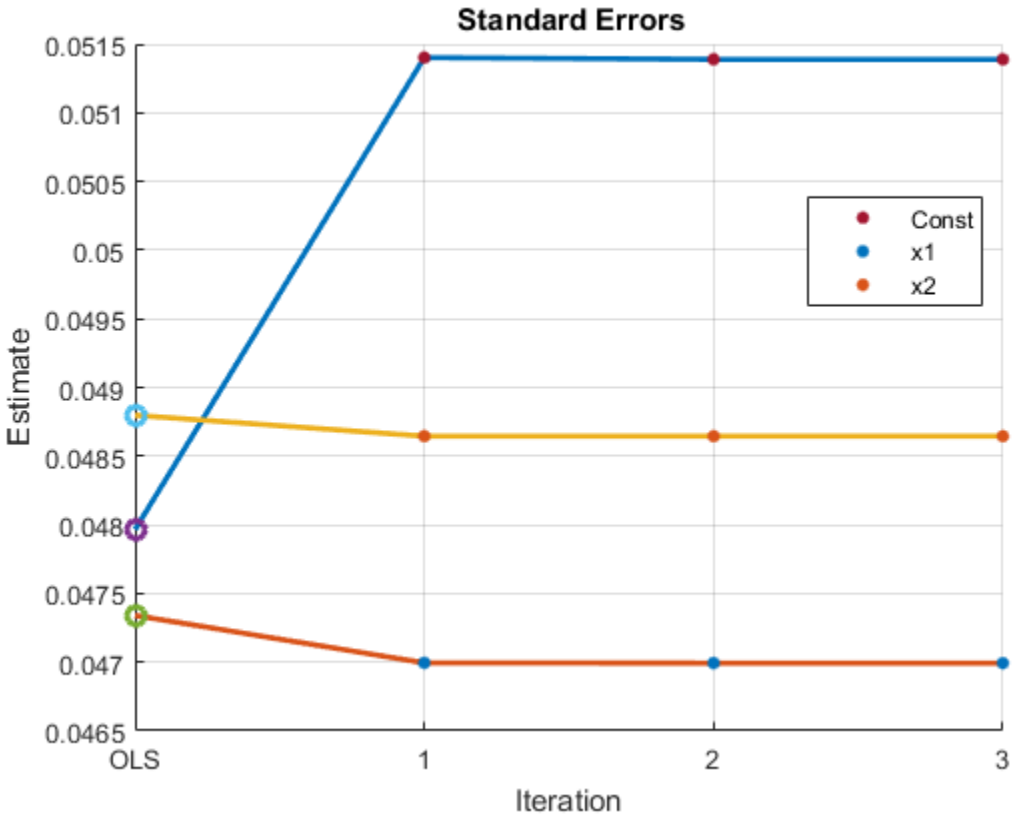
```
arParams = arma2ar(phi,theta);
arLags = sum(abs(arParams) > 0.00001);
```

Estimate the regression coefficients using three iterations of FGLS, and specify the number of lags in the AR innovation model (`arLags`). Also, specify to plot the coefficient estimates and their standard errors for each iteration, and to display the final estimates and the OLS estimates in tabular form.

```
[coeff,~,EstCoeffCov] = fgls(X,y,'innovMdl','AR','arLags',arLags,...
    'numIter',3,'plot',{'coeff','se'},'display','final');
```

```
OLS Estimates:

       |  Coeff    SE
------------------------
 Const | 1.0375  0.0480
 x1    | 2.0409  0.0473
 x2    | 2.9860  0.0488

FGLS Estimates:

       |  Coeff    SE
------------------------
 Const | 1.0372  0.0514
 x1    | 2.0366  0.0470
 x2    | 2.9919  0.0486
```


Coefficients
Estimate
Iteration
Const
x1
x2
3
2.8
2.6
2.4
2.2
2
1.8
1.6
1.4
1.2
1
OLS
1
2
3



The algorithm seems to converge after the first iteration, and the estimates are close to the OLS estimates, with the standard errors being slightly smaller.

Properties of iterative FGLS estimates in finite samples are difficult to establish. For asymptotic properties, one iteration of FGLS is sufficient. `fgls` supports iterative FGLS for experimentation.

If the estimates or standard errors show instability after successive iterations, then the estimated innovations covariance might be ill conditioned. Consider scaling the residuals using the `'resCond'` name-value pair argument to improve the conditioning of the estimated innovations covariance.

- “Classical Model Misspecification Tests”

- “Time Series Regression I: Linear Models”
- “Time Series Regression VI: Residual Diagnostics”
- “Time Series Regression X: Generalized Least Squares and HAC Estimators”

Input Arguments

`X` — Predictor data

numeric matrix

Predictor data for the multiple linear regression model, specified as a `numObs`-by-`numPreds` numeric matrix.

`numObs` is the number of observations and `numPreds` is the number of predictor variables.

Data Types: `double`

`y` — Response data

vector

Response data for the multiple linear regression model, specified as a `numObs`-by-1 vector with numeric or logical entries.

Data Types: `double | logical`

`Tbl` — Predictor and response data

tabular array

Predictor and response data for the multiple linear regression model, specified as a `numObs`-by-`numPreds` + 1 tabular array.

The first `numPreds` variables of `Tbl` are the predictor data, and the last variable is the response data.

The predictor data must be numeric, and the response data must be numeric or logical.

Data Types: `table`

Note: `NaN`s in `X`, `y`, or `Tbl` indicate missing values, and `fgls` removes observations containing at least one `NaN`. That is, to remove `NaN`s in `X` or `y`, the software merges them

([X y]), and then uses list-wise deletion to remove any row that contains at least one NaN. The software also removes any row of Tbl containing at least one NaN. Removing NaNs in the data reduces the sample size, and can also create irregular time series.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: 'innovMdl','HC0','numIter',10,'plot','coeff' specifies White's robust innovations covariance model, 10 iterations of FGLS, and to plot coefficient estimates after each iteration.

'varNames' — Variable names

cell vector of strings

Variable names used in displays and plots of the results, specified as the comma-separated pair consisting of 'varNames' and a cell vector of strings. varNames must have length numPreds, and each cell corresponds to a variable name. The software truncates all variable names to the first five characters.

varNames must include variable names for all variables in the model, such as an intercept term (e.g., 'Const') or higher-order terms (e.g., 'x1^2' or 'x1:x2').

The default variable names for:

- The matrix X is the cell vector of strings {'x1','x2',...}
- The tabular array Tbl is the property Tbl.Properties.VariableNames

Example: 'varNames',{'Const','AGE','BBD'}

Data Types: cell

'intercept' — Indicate whether to include model intercept

true (default) | false

Indicate whether to include model intercept when fgls fits the model, specified as the comma-separated pair consisting of 'intercept' and true or false.

Value	Description
`true`	Include an intercept in the model.
`false`	Exclude an intercept from the model.

Example: `'intercept',false`

Data Types: `logical`

`'innovMdl'` — Model for innovations covariance estimate

`'AR'` (default) | `'CLM'` | `'HC0'` | `'HC1'` | `'HC2'` | `'HC3'` | `'HC4'`

Model for the innovations covariance estimate, specified as the comma-separated pair consisting of `'innovMdl'` and a string.

Set `'innovMdl'` to specify the structure of the innovations covariance estimator $\hat{\Omega}$.

- For diagonal innovations covariance models (i.e., models with heteroscedasticity), $\hat{\Omega} = \mathrm{diag}(\omega)$, where $\omega = \{\omega_i;\ i = 1,...,T\}$ is a vector of innovation variance estimates for the observations, and $T = \mathtt{numObs}$.

  `fgls` estimates the data-driven vector ω using the corresponding model residuals (ε), their leverages $h_i = x_i(X'X)^{-1}x_i'$, and the degrees of freedom *dfe*.

  Use this table to choose `'innovMdl'`.

Value	Weight	Reference
`'CLM'`	$\omega_i = \frac{1}{dfe}\sum_{i=1}^{T}\varepsilon_i^2$	[4]
`'HC0'`	$\omega_i = \varepsilon_i^2$	[6]
`'HC1'`	$\omega_i = \frac{T}{dfe}\varepsilon_i^2$	[5]
`'HC2'`	$\omega_i = \frac{\varepsilon_i^2}{1-h_i}$	[5]

Value	Weight	Reference
`'HC3'`	$\omega_i = \frac{\varepsilon_i^2}{(1-h_i)^2}$	[5]
`'HC4'`	$\omega_i = \frac{\varepsilon_i^2}{(1-h_i)^{d_i}}$ where $d_i = \min\left(4, \frac{h_i}{\bar{h}}\right)$,	[1]

- For full innovation covariance models (i.e., models having heteroscedasticity and autocorrelation), specify `'AR'`. The software imposes an AR(p) model on the innovations, and constructs $\hat{\Omega}$ using the number of lags, p, specified by the name-value pair argument `arLags` and the Yule-Walker equations.

If `numIter` is `1` and you specify `InnovCov0`, then `fgls` ignores `InnovMdl`.

Example: `'innovMdl',HC0`

Data Types: `char`

`'arLags'` — Number of lags

1 (default) | positive integer

Number of lags to include in the AR innovations model, specified as the comma-separated pair consisting of `'arLags'` and a positive integer.

If `innovMdl` is not `'AR'` (i.e., for diagonal models), then the software ignores the value of `'arLags'`.

For general ARMA innovations models, convert to the equivalent AR form by:

- Constructing the ARMA innovations model lag operator polynomial using LagOp. Then, divide the AR polynomial by the MA polynomial using, e.g., mrdivide. The result is the inifinite-order, AR representation of the ARMA model.
- Using `arma2ar`, which returns the coefficients of the inifinite-order, AR representation of the ARMA model.

Example: `'arLags',4`

Data Types: `double`

`'InnovCov0'` — Initial innovations covariance

`[ ]` (default) | vector of positive scalars | positive definite matrix | positive semidefinite matrix

Initial innovations covariance, specified as the comma-specified pair consisting of `'InnovCov0'` and a vector of positive scalars, positive semidefinite matrix, or a positive definite matrix.

`InnovCov0` replaces the data-driven estimate of the innovations covariance ($\hat{\Omega}$) in the first iteration of GLS.

- For diagonal innovations covariance models (i.e., models with heteroscedasticity), specify a `numObs`-by-1 vector. `InnovCov0(`j`)` is the variance of innovation j.
- For full innovation covariance models (i.e., models having heteroscedasticity and autocorrelation), specify a `numObs`-by-`numObs` matrix. `InnovCov0(`j`,`k`)` is the covariance of innovations j and k.
- By default, `fgls` uses a data-driven $\hat{\Omega}$ (see `innovMdl`).

Data Types: `double`

`'numIter'` — Number of iterations

1 (default) | positive integer

Number of iterations to implement for the FGLS algorithm, specified as the comma-separated pair consisting of `'numIter'` and a positive integer.

`fgls` estimates the innovations covariance ($\hat{\Omega}$) at each iteration from the residual series according to the innovations covariance model (`innovMdl`). Then, the software computes the GLS estimates of the model coefficients.

Example: `'numIter',10`

Data Types: `double`

`'resCond'` — Flag indicating to scale residuals

`false` (default) | `true`

Flag indicating to scale the residuals at each iteration of FGLS, specified as the comma-separated pair consisting of `'resCond'` and `true` or `false`.

Value	Description
`true`	`fgls` scales the residuals at each iteration.
`false`	`fgls` does not scale the residuals at each iteration.

Scaling the residuals at each iteration of FGLS tends to improve the conditioning of the estimation of the innovations covariance ($\hat{\Omega}$).

Data Types: `logical`

`'display'` — Command Window display control

`'off'` (default) | `'final'` | `'iter'`

Command Window display control, specified as the comma-separated pair consisting of `'display'` and a string in the following table.

Value	Description
`'final'`	Display the final estimates.
`'iter'`	Display the estimates after each iteration.
`'off'`	Suppress displaying to the Command Window.

`fgls` shows estimation results in tabular form.

Example: `'display','iter'`

`'plot'` — Control for plotting results

`'off'` (default) | `'all` | `'coeff'` | `'mse'` | `'se'` | cell array of strings

Control for plotting results after each iteration, specified as the comma-separated pair consisting of `'plot'` and a string or cell array of strings.

To examine the convergence of the FGLS algorithm, it is good practice to specify plotting the estimates for each iteration. This table contains the available plot-control strings.

Value	Description
`'all`	Plot the estimated coefficients, their standard errors, and the residual mean-squared error (MSE) on separate plots.

Value	Description
`'coeff'`	Plot the estimated coefficients.
`'mse'`	Plot the MSE.
`'off'`	Do not plot the results.
`'se'`	Plot the estimated coefficient.

Output Arguments

`coeff` — FGLS coefficient estimates

numeric vector

FGLS coefficient estimates, returned as a `numPreds`-by-1 numeric vector.

The order of the estimates corresponds to the order of the predictor matrix columns or `Tbl.VariableNames`. For example, in a model with an intercept, the value of $\hat{\beta}_1$ (corresponding to the predictor x_1) is in position 2 of `coeff`.

`se` — Coefficient standard error estimates

numeric vector

Coefficient standard error estimates, returned as a `numPreds`-by-1 numeric. The elements of `se` are `sqrt(diag(EstCoeffCov))`.

The order of the estimates corresponds to the order of the predictor matrix columns or `Tbl.VariableNames`. For example, in a model with an intercept, the estimated standard error of $\hat{\beta}_1$ (corresponding to the predictor x_1) is in position 2 of `se`, and is the square root of the value in position (2,2) of `EstCoeffCov`.

`EstCoeffCov` — Coefficient covariance estimate

numeric matrix

Coefficient covariance estimate, returned as a `numPreds`-by-`numPreds` numeric matrix.

The order of the rows and columns of `EstCoeffCov` corresponds to the order of the predictor matrix columns or `Tbl.VariableNames`. For example, in a model with an intercept, the estimated covariance of $\hat{\beta}_1$ (corresponding to the predictor x_1) and $\hat{\beta}_2$

(corresponding to the predictor x_2) are in positions (2,3) and (3,2) of `EstCoeffCov`, respectively.

More About

Feasible Generalized Least Squares

Feasible generalized least squares (FGLS) estimates the coefficients of a multiple linear regression model and their covariance matrix in the presence of nonspherical innovations with an unknown covariance matrix.

Let $y_t = X_t\beta + \varepsilon_t$ be a multiple linear regression model, where the innovations process ε_t is Gaussian with mean 0, but with true, nonspherical covariance matrix Ω (e.g., the innovations are heteroscedastic or autocorrelated). Also, suppose that the sample size is T and there are p predictors (including an intercept). Then, the FGLS estimator of β is

$$\hat{\beta}_{FGLS} = \left(X \;\; \hat{\Omega}^{-1}X\right)^{-1} X \;\; \hat{\Omega}^{-1}y,$$

where $\hat{\Omega}$ is an innovations covariance estimate based on a model (e.g., innovations process forms an AR(1) mdoel). The estimated coefficient covariance matrix is

$$\hat{\Sigma}_{FGLS} = \hat{\sigma}^2_{FGLS}\left(X \;\; \hat{\Omega}^{-1}X\right)^{-1},$$

where

$$\hat{\sigma}^2_{FGLS} = y \;\; \left[\hat{\Omega}^{-1} - \hat{\Omega}^{-1}X\left(X \;\; \hat{\Omega}^{-1}X\right)^{-1} X \;\; \hat{\Omega}^{-1}\right] y/(T-p).$$

FGLS estimates are computed as follows:

1. OLS is applied to the data, and then residuals $(\hat{\varepsilon}_t)$ are computed.
2. $\hat{\Omega}$ is estimated based on a model for the innovations covariance.

3 $\hat{\beta}_{FGLS}$ is estimated, along with its covariance matrix $\hat{\Sigma}_{FGLS}$.

4 Optional: This process can be iterated by performing the following steps until $\hat{\beta}_{FGLS}$ converges.

   a Compute the residuals of the fitted model using the FGLS estimates.

   b Apply steps 2–3.

If $\hat{\Omega}$ is a consistent estimator of Ω and the predictors that comprise X are exogenous, then FGLS estimators are consistent and efficient.

Asymptotic distributions of FGLS estimators are unchanged by repeated iteration. However, iterations might change finite sample distributions.

Generalized Least Squares

Generalized least squares (GLS) estimates the coefficients of a multiple linear regression model and their covariance matrix in the presence of nonspherical innovations with known covariance matrix.

The setup and process for obtaining GLS estimates is the same as in FGLS, but replace $\hat{\Omega}$ with the known innovations covariance matrix Ω.

In the presence of nonspherical innovations and with known innovations covariance, GLS estimators are unbiased, efficient, consistent, and hypothesis tests based on the estimates are valid.

Weighted Least Squares

Weighted least squares (WLS) estimates the coefficients of a multiple linear regression model and their covariance matrix in the presence of uncorrelated, but heteroscedastic innovations with known, diagonal covariance matrix.

The setup and process to obtain WLS estimates is the same as in FGLS, but replace $\hat{\Omega}$ with the known, diagonal matrix of weights, typically the diagonal elements are the inverse of the variances of the innovations.

In the presence of heteroscedastic innovations and when the variances of the innovations are known, WLS estimators are unbiased, efficient, consistent, and hypothesis tests based on the estimates are valid.

Tips

- To obtain standard generalized least squares (GLS) estimates:
  - Set the `InnovCov0` name-value pair argument to the known innovations covariance.
  - Set the `numIter` name-value pair argument to `1`.
- To obtain WLS estimates, set the `InnovCov0` name-value pair argument to a vector of inverse weights (e.g., innovations variance estimates).
- In specific models and with repeated iterations, scale differences in the residuals might produce a badly conditioned estimated innovations covariance and induce numerical instability. If you set `'resCond',true`, then conditioning improves.

Algorithms

- In the presence of nonspherical innovations, GLS produces efficient estimates relative to OLS, and consistent coefficient covariances, conditional on the innovations covariance. The degree to which `fgls` maintains these properties depends on the accuracy of both the model and estimation of the innovations covariance.
- Rather than estimate FGLS estimates the usual way, `fgls` uses methods that are faster and more stable, and are applicable to rank-deficient cases.
- Traditional FGLS methods, such as the Cochrane-Orcutt procedure, use low-order, autoregressive models. These methods, however, estimate parameters in the innovations covariance matrix using OLS, where `fgls` uses maximum likelihood estimation (MLE) [2].

- "Autocorrelation and Partial Autocorrelation" on page 3-13
- "Engle's ARCH Test" on page 3-25
- "Nonspherical Models" on page 3-94
- "Time Series Regression Models" on page 4-3

References

[1] Cribari-Neto, F. "Asymptotic Inference Under Heteroskedasticity of Unknown Form." *Computational Statistics & Data Analysis*. Vol. 45, 2004, pp. 215–233.

[2] Hamilton, J. D. *Time Series Analysis*. Princeton, NJ: Princeton University Press, 1994.

[3] Judge, G. G., W. E. Griffiths, R. C. Hill, H. Lutkepohl, and T. C. Lee. *The Theory and Practice of Econometrics*. New York, NY: John Wiley & Sons, Inc., 1985.

[4] Kutner, M. H., C. J. Nachtsheim, J. Neter, and W. Li. *Applied Linear Statistical Models*. 5th ed. New York: McGraw-Hill/Irwin, 2005.

[5] MacKinnon, J. G., and H. White. "Some Heteroskedasticity-Consistent Covariance Matrix Estimators with Improved Finite Sample Properties." *Journal of Econometrics*. Vol. 29, 1985, pp. 305–325.

[6] White, H. "A Heteroskedasticity-Consistent Covariance Matrix and a Direct Test for Heteroskedasticity." *Econometrica*. Vol. 48, 1980, pp. 817–838.

See Also

`arma2ar` | `fitlm` | `hac` | `lscov` | regARIMA

Introduced in R2014b

filter

Filter disturbances through conditional variance model

Syntax

```
[V,Y] = filter(Mdl,Z)
[V,Y] = filter(Mdl,Z,Name,Value)
```

Description

`[V,Y] = filter(Mdl,Z)` filters disturbances (`Z`) through the fully specified conditional variance model (`Mdl`) to produce conditional variances (`V`) and responses (`Y`). `Mdl` can be a `garch`, `egarch`, or `gjr` model.

`[V,Y] = filter(Mdl,Z,Name,Value)` filters disturbances using additional options specified by one or more `Name,Value` pair arguments. For example, you can specify presample disturbance and conditional variance paths.

Examples

Simulate and Filter Disturbance Path Through GARCH Model

Specify a GARCH(1,1) model with Gaussian innovations.

```
Mdl = garch('Constant',0.005,'GARCH',0.8,'ARCH',0.1);
```

Simulate the model using Monte Carlo simulation. Then, standardize the simulated innovations and filter them.

```
rng(1); % For reproducibility
[v,e] = simulate(Mdl,100,'E0',0,'V0',0.05);
Z = e./sqrt(v);
[V,E] = filter(Mdl,Z,'Z0',0,'V0',0.05);
```

Confirm that the outputs of `simulate` and `filter` are identical.

```
isequal(v,V)
```

```
ans =

     1
```

The logical value 1 confirms that the two outputs are identical.

Filter Multiple Disturbance Paths Through EGARCH Model

Specify an EGARCH(1,1) model with Gaussian innovations.

```
Mdl = egarch('Constant',-0.1,'GARCH',0.8,'ARCH',0.3,...
    'Leverage',-0.05);
```

Simulate 25 series of standard Gaussian observations for 100 periods.

```
rng(1); % For reproducibility
Z = randn(100,25);
```

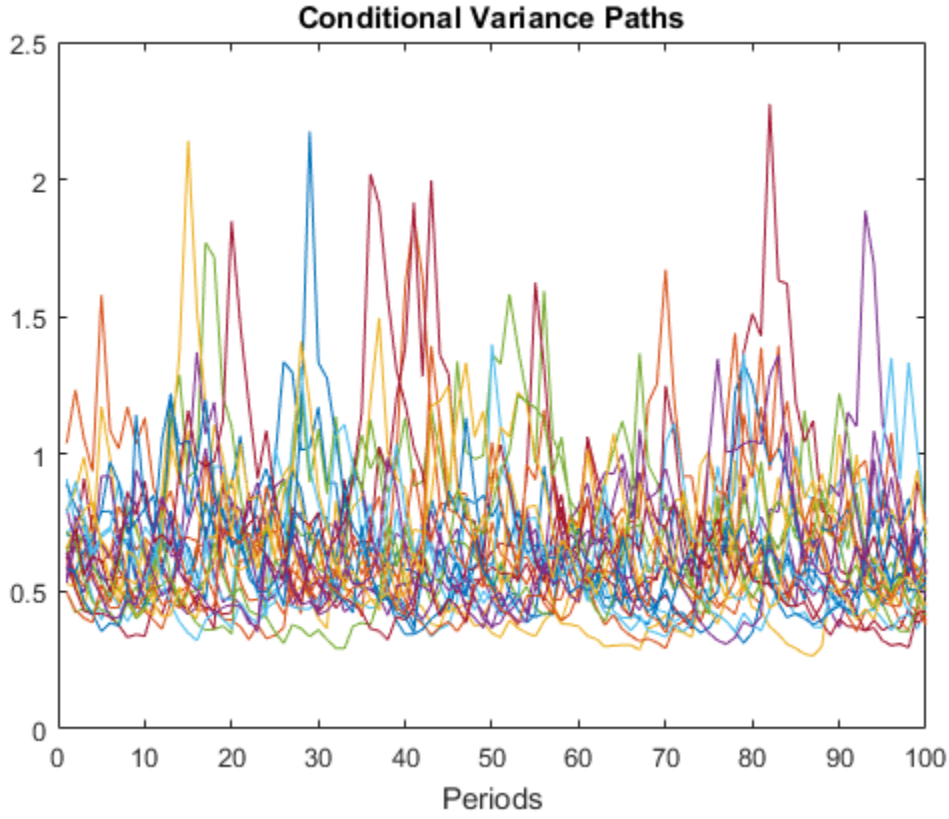
Z represents 25 paths of synchronized disturbances for 100 periods.

Obtain 25 paths of conditional variances by filtering the disturbance paths through the EGARCH(1,1) model.

```
V = filter(Mdl,Z);
```

Plot the paths of conditional variances.

```
figure;
plot(V);
title('Conditional Variance Paths');
xlabel('Periods');
```



Filter Disturbances Through GJR Model Specifying Presample Observations

Specify a GJR(1,2) model with Gaussian innovations.

```
Mdl = gjr('Constant',0.005,'GARCH',0.8,'ARCH',{0.1 0.01},...
    'Leverage',{0.05 0.01});
```

Simulate 25 series of standard Gaussian observations for 102 periods.

```
rng(1); % For reproducibility
Z = randn(102,25);
```

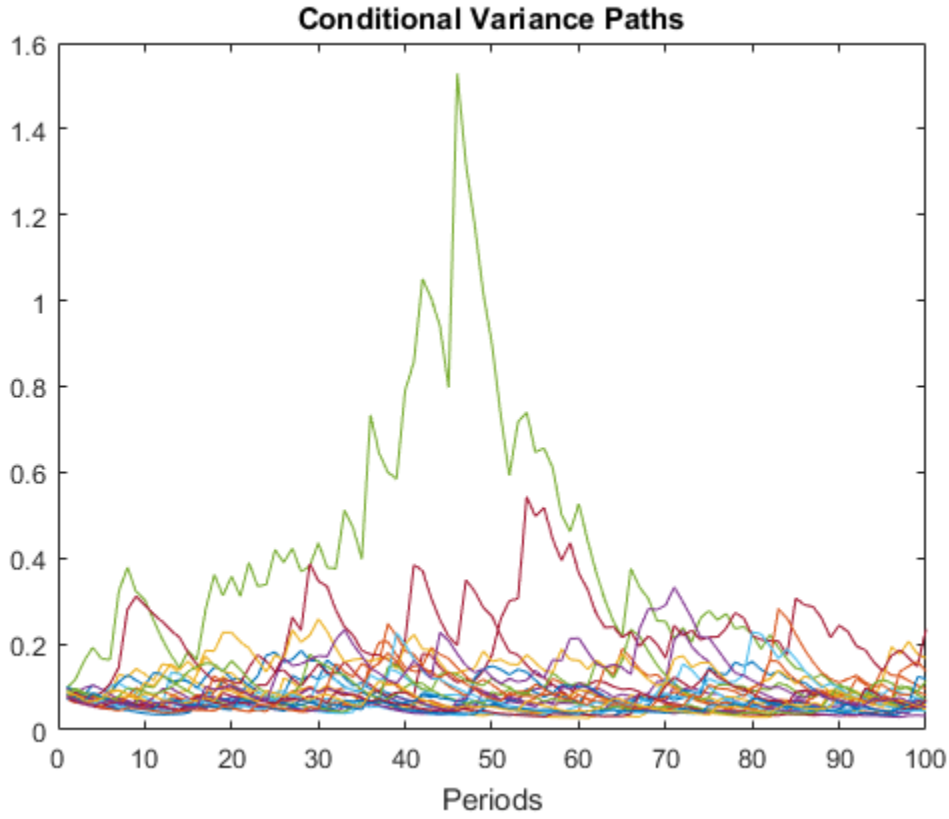
`Z` represents 25 paths of synchronized disturbances for 102 periods.

Obtain 25, 100 period paths of conditional variances by filtering the disturbance paths through the GJR(1,2) model. Specify the first two disturbances as presample observations.

```
V = filter(Mdl,Z(3:end,:),'Z0',Z(1:2,:));
```

Plot the paths of conditional variances.

```
figure;
plot(V);
title('Conditional Variance Paths');
xlabel('Periods');
```



- “Simulate Conditional Variance Model” on page 6-111

- “Simulate GARCH Models” on page 6-97

Input Arguments

`Mdl` — Conditional variance model

`garch` model object | `egarch` model object | `gjr` model object

Conditional variance model without any unknown parameters, specified as a `garch`, `egarch`, or `gjr` model object.

`Mdl` cannot contain any properties that have `NaN` value.

`Z` — Disturbance paths

numeric column vector | numeric matrix

Disturbance paths used to drive the innovation process, specified as a numeric vector or matrix. Given the variance process, σ_t^2, and the disturbance process z_t, the innovation process is

$$\varepsilon_t = \sigma_t z_t.$$

As a column vector, `Z` represents a single path of the underlying disturbance series.

As a matrix, the rows of `Z` correspond to periods. The columns correspond to separate paths. The observations across any row occur simultaneously.

The last element or row of `Z` contains the latest observation.

Note: `NaN`s indicate missing values. `filter` removes these values from `Z` by listwise deletion. The software removes any row of `Z` with at least one `NaN`. Removing `NaN`s in the data reduces the sample size, and can also create irregular time series.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: 'Z0',[1 1;0.5 0.5],'V0',[1 0.5;1 0.5] specifies two equivalent presample paths of innovations and two, different presample paths of conditional variances.

'Z0' — Presample disturbance paths

numeric column vector | numeric matrix

Presample disturbance paths, specified as a numeric vector or matrix. Z0 provides initial values for the input disturbance series, Z.

- If Z0 is a column vector, then filter applies it to each output path.
- If Z0 is a matrix, then it must have at least as many columns as Z. If Z0 has more columns than Z, then filter uses the first size(Z,2) columns only.

Z0 must have at least Mdl.Q rows to initialize the conditional variance model. If the number of rows in Z0 exceeds Mdl.Q, then filter uses the latest, required number of observations only.

The last element or row contains the latest observation.

By default, filter sets any necessary presample disturbances to an independent sequence of standardized disturbances drawn from Mdl.Distribution.

Data Types: double

'V0' — Positive presample conditional variance paths

numeric column vector | numeric matrix

Positive presample conditional variance paths, specified as a numeric vector or matrix. V0 provides initial values for the conditional variances in the model.

- If V0 is a column vector, then filter applies it to each output path.
- If V0 is a matrix, then it must have at least as many columns as Z. If V0 has more columns than Z, then filter uses the first size(Z,2) columns only.

V0 must have at least max(Mdl.P,Mdl.Q) rows to initialize the variance equation. If the number of rows in V0 exceeds the necessary number, then filter uses the latest, required number of observations only.

The last element or row contains the latest observation.

By default, `filter` sets any necessary presample conditional variances to the unconditional variance of the process.

Data Types: `double`

Notes

- `NaN`s indicate missing values. `filter` removes missing values. The software merges the presample data (`Z0` and `V0`) separately from the disturbances (`Z`), and then uses

  list-wise deletion to remove rows containing at least one `NaN`. Removing `NaN`s in the data reduces the sample size. Removing `NaN`s can also create irregular time series.
- `filter` assumes that you synchronize presample data such that the latest observation of each presample series occurs simultaneously.

Output Arguments

`V` — Conditional variance paths

numeric column vector | numeric matrix

Conditional variance paths, returned as a column vector or matrix. `V` represents the conditional variances of the mean-zero, heteroscedastic innovations associated with `Y`.

The dimensions of `V` and `Z` are equivalent. If `Z` is a matrix, then the columns of `V` are the filtered conditional variance paths corresponding to the columns of `Z`.

Rows of `V` are periods corresponding to the periodicity of `Z`.

`Y` — Response paths

numeric column vector | numeric matrix

Response paths, returned as a numeric column vector or matrix. `Y` usually represents a mean-zero, heteroscedastic time series of innovations with conditional variances given in `V`.

`Y` can also represent a time series of mean-zero, heteroscedastic innovations plus an offset. If `Mdl` includes an offset, then `filter` adds the offset to the underlying mean-zero, heteroscedastic innovations. Therefore, `Y` represents a time series of offset-adjusted innovations.

If `Z` is a matrix, then the columns of `Y` are the filtered response paths corresponding to the columns of `Z`.

Rows of `Y` are periods corresponding to the periodicity of `Z`.

Alternatives

`filter` generalizes `simulate`. Both function filter a series of disturbances to produce output responses and conditional variances. However, `simulate` autogenerates a series of mean-zero, unit-variance, independent and identically distributed (iid) disturbances according to the distribution in the conditional variance model object, `Mdl`. In contrast, `filter` lets you directly specify your own disturbances.

More About

- Using garch Objects
- Using egarch Objects
- Using gjr Objects
- “Monte Carlo Simulation of Conditional Variance Models” on page 6-92
- “Presample Data for Conditional Variance Model Simulation” on page 6-95

References

[1] Bollerslev, T. “Generalized Autoregressive Conditional Heteroskedasticity.” *Journal of Econometrics*. Vol. 31, 1986, pp. 307–327.

[2] Bollerslev, T. “A Conditionally Heteroskedastic Time Series Model for Speculative Prices and Rates of Return.” *The Review of Economics and Statistics*. Vol. 69, 1987, pp. 542–547.

[3] Box, G. E. P., G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis: Forecasting and Control*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

[4] Enders, W. *Applied Econometric Time Series*. Hoboken, NJ: John Wiley & Sons, 1995.

[5] Engle, R. F. “Autoregressive Conditional Heteroskedasticity with Estimates of the Variance of United Kingdom Inflation.” *Econometrica*. Vol. 50, 1982, pp. 987–1007.

[6] Hamilton, J. D. *Time Series Analysis*. Princeton, NJ: Princeton University Press, 1994.

See Also

`egarch` | `estimate` | `forecast` | `garch` | `gjr` | `infer` | `print` | `simulate`

Introduced in R2012a

filter

Class: arima

Filter disturbances using ARIMA or ARIMAX model

Syntax

```
[Y,E,V] = filter(Mdl,Z)
[Y,E,V] = filter(Mdl,Z,Name,Value)
```

Description

`[Y,E,V] = filter(Mdl,Z)` filters disturbances, `Z`, to produce responses, innovations, and conditional variances of a univariate ARIMA(p,D,q) model.

`[Y,E,V] = filter(Mdl,Z,Name,Value)` filters disturbances using additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

`Mdl`

ARIMA model, as created by arima or estimate. The input model cannot have any `NaN` values.

`Z`

`numObs`-by-`NumPaths` matrix of disturbances, z_t, that drives the innovation process, ε_t.

For a variance process σ_t^2, the innovation process is given by

$$\varepsilon_t = \sigma_t z_t.$$

As a column vector, `Z` represents a path of the underlying disturbance series. As a matrix, `Z` represents `numObs` observations of `NumPaths` paths of the underlying

disturbance series. `filter` assumes that observations across any row occur simultaneously. The last row contains the most recent observation.

Note: `NaN`s indicate missing values. `filter` removes them from `Z` using list-wise deletion. That is, `filter` removes any row of `Z` containing at least one `NaN`. This deletion reduces the effective sample size and can cause irregular time series.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'V0'`

Positive presample conditional variances that provide initial values for the model. If `V0` is a column vector, then `filter` applies it to each output path. If `V0` is a matrix, then it requires at least `NumPaths` columns. If the number of columns exceeds `NumPaths`, then `filter` uses the first `NumPaths` columns.

`V0` requires enough rows to initialize the moving average component of the ARIMA model and any conditional variance model. The required number of rows is at least `Mdl.Q`. If you include a conditional variance model, then `filter` might require more than `Mdl.Q` rows. If the number of rows in `V0` exceeds the number necessary, then `filter` uses the most recent observations. The last row contains the most recent observation.

Default: `filter` sets necessary presample observations to the unconditional variance of the conditional variance process.

`'X'`

Matrix of predictor data corresponding to a regression component in the conditional mean model. The columns of `X` are separate, synchronized time series, with the last row containing the most recent observations. The number of rows of `X` must be at least the number of rows of `Z`. When the number of rows of `X` exceeds the number necessary, then `filter` uses the most recent observations.

Default: `filter` does not include a regression component in the conditional mean model regardless of the presence of regression coefficients in `Mdl`.

`'Y0'`

Presample response data, providing initial values for the model. If `Y0` is a column vector, then `filter` applies it to each output path. If `Y0` is a matrix, then it requires at least `NumPaths` columns. If the number of columns in `Y0` exceeds `NumPaths`, then `filter` uses the first `NumPaths` columns.

`Y0` requires at least `Mdl.P` rows to initialize the model. If the number of rows in `Y0` exceeds `Mdl.P`, then `filter` uses the most recent `Mdl.P` observations. The last row contains the most recent observation.

Default: `filter` sets the necessary presample observations to the unconditional mean for stationary processes, and to 0 for nonstationary processes or processes with a regression component.

`'Z0'`

Presample disturbances, providing initial values for the input disturbance series, `Z`. If `Z0` is a column vector, then `filter` applies it to each output path. If `Z0` is a matrix, then it requires at least `NumPaths` columns. If the number of columns exceeds `NumPaths`, then `filter` uses the first `NumPaths` columns.

`Z0` requires a sufficient number of rows to initialize the moving average component of the ARIMA model and any conditional variance model. The required number of rows is at least `Mdl.Q`, but might be more if a conditional variance model is included. If the number of rows in `Z0` exceeds the number necessary, then `filter` uses the most recent observations. The last row contains the most recent observation.

Default: `filter` sets the necessary presample observations to 0.

Notes

- `NaN`s in the data indicate missing values and `filter` removes them. The software merges the presample data and main data sets separately, then uses list-wise deletion to remove any `NaN`s. That is, `filter` sets `PreSample` = `[Y0 Z0 V0]` and `Data` = `[Z X]`, then it removes any row in `PreSample` or `Data` that contains at least one `NaN`.
- Removing `NaN`s in the main data reduces the effective sample size. Such removal can also create irregular time series.
- `filter` assumes that you synchronize presample data such that the most recent observation of each presample series occurs simultaneously.

- All predictor series in `X` (i.e., columns of `X`) are applied to each disturbance series in `Z` to produce `NumPaths` response series `Y`.

Output Arguments

`Y`

`numObs`-by-`NumPaths` matrix of simulated responses. `Y` is the continuation of the presample series `Y0`.

`E`

`numObs`-by-`NumPaths` matrix of simulated innovations with conditional variances, `V`. Each column is a scaled series of innovations (or disturbances) such that `E = sqrt(V)*Z`.

`V`

`numObs`-by-`NumPaths` matrix of conditional variances of the innovations in `E` such that `E = sqrt(V)*Z`. `V` is the continuation of the presample series `V0`.

Examples

Simulate and Filter

Specify a mean zero ARIMA(2,0,1) model.

```
Mdl = arima('Constant',0,'AR',{0.5,-0.8},'MA',-0.5,...
    'Variance',0.1);
```

Simulate the model using Monte Carlo simulation. Then, standardize the simulated innovations and filter them.

```
rng(1); % For reproducibility
[y,e,v] = simulate(Mdl,100);
Z = e./sqrt(v);
[Y,E,V] = filter(Mdl,Z);
```

Confirm that the outputs of `simulate` and `filter` are identical.

```
isequal(y,Y)
```

```
ans =

     1
```

The logical value `1` confirms the two outputs are identical.

Simulate an Impulse Response Function

Specify a mean zero ARIMA(2,0,1) model.

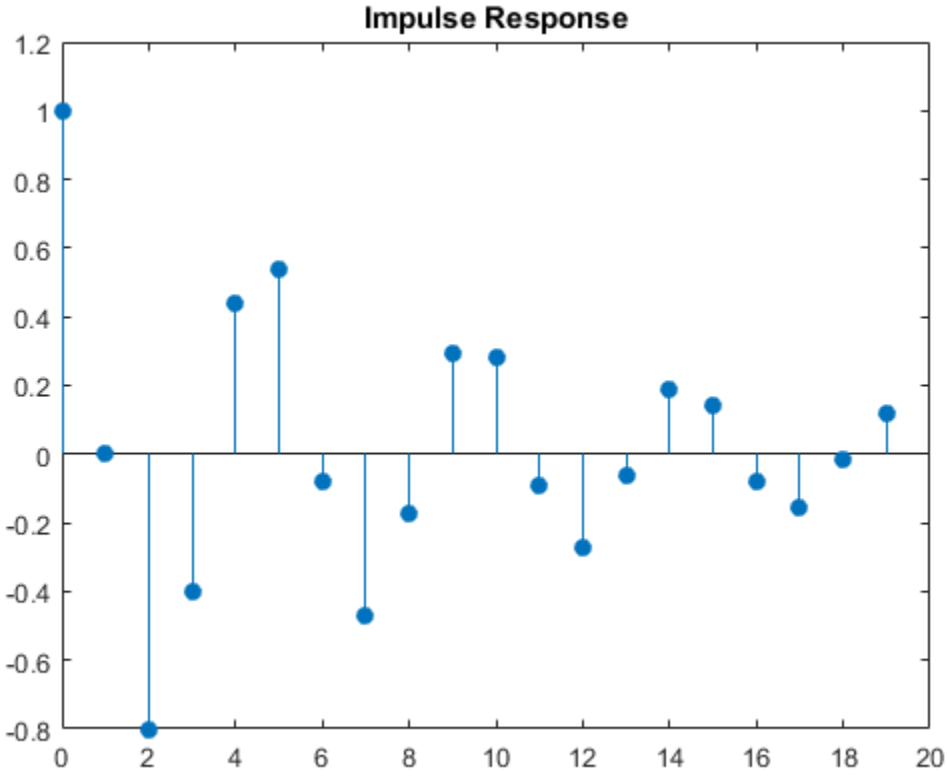
```
Mdl = arima('Constant',0,'AR',{0.5,-0.8},'MA',-0.5,...
    'Variance',0.1);
```

Simulate the first 20 responses of the impulse response function. Generate a disturbance series with a one-time, unit impulse, and then filter it. Set all presample observations equal to zero. Normalize the impulse response function to ensure that the first element is 1.

```
Z = [1;zeros(19,1)];
Y = filter(Mdl,Z,'Y0',zeros(Mdl.P,1));
Y = Y/Y(1);
```

Plot the impulse response function.

```
figure;
stem((0:numel(Y)-1)',Y,'filled');
title 'Impulse Response';
```



The impulse response assesses the dynamic behavior of a system to a one-time, unit impulse. You can also use the `impulse` method to plot the impulse response function for an ARIMA process.

Simulate a Step Response

Specify a mean zero ARIMA(2,0,1) process.

```
Mdl = arima('Constant',0,'AR',{0.5,-0.8},'MA',-0.5,...
    'Variance',0.1);
```

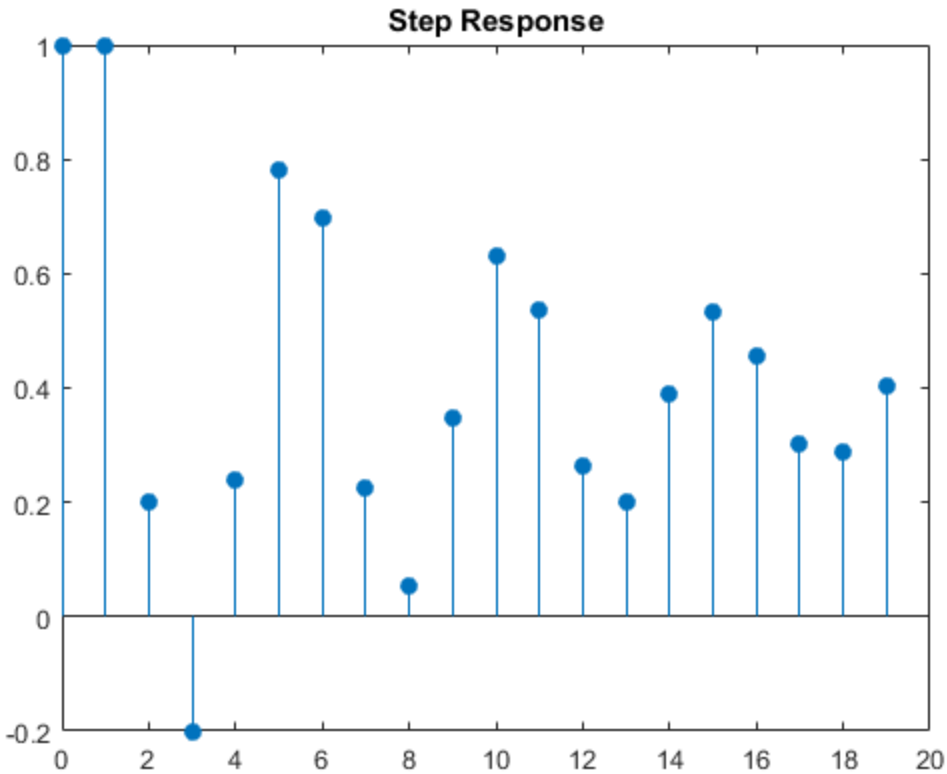
Simulate the first 20 responses to a sequence of unit disturbances. Generate a disturbance series of ones, and then filter it. Set all presample observations equal to zero.

```
Z = ones(20,1);
Y = filter(Mdl,Z,'YO',zeros(Mdl.P,1));
Y = Y/Y(1);
```

The last step normalizes the step response function to ensure that the first element is 1.

Plot the step response function.

```
figure;
stem((0:numel(Y)-1)',Y,'filled');
title 'Step Response';
```



The step response assess the dynamic behavior of a system to a persistent change in a variable.

Simulate a Response with Predictor Data

Create models for the response and predictor series. Set an ARIMAX(2,1,3) model to the response `MdlY`, and an AR(1) model to the `MdlX`.

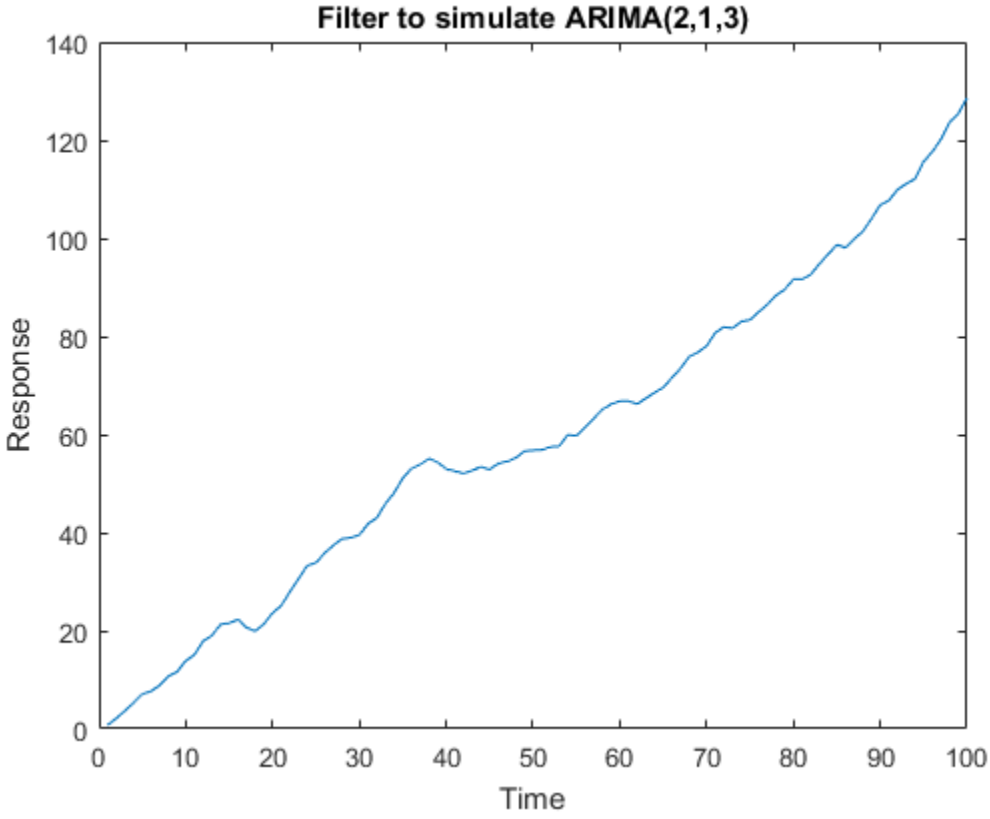
```
MdlY = arima('AR',{0.1 0.2},'D',1,'MA',{-0.1 0.1 0.05},...
'Constant',1,'Variance',0.5, 'Beta',2);
MdlX = arima('AR',0.5,'Constant',0,'Variance',0.1);
```

Simulate a length 100 predictor series `x` and a series of iid normal disturbances `z` having mean zero and variance 1.

```
rng(1);
z = randn(100,1);
x = simulate(MdlX,100);
```

Filter the disturbances `z` using `MdlY` to produce the response series `y`, and plot `y`.

```
y = filter(MdlY,z,'X',x);
figure;
plot(y);
title 'Filter to simulate ARIMA(2,1,3)';
xlabel 'Time';
ylabel 'Response';
```



- "Simulate Conditional Mean and Variance Models" on page 5-175
- "Plot the Impulse Response Function" on page 5-88

References

[1] Box, G. E. P., G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis: Forecasting and Control* 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

[2] Enders, W. *Applied Econometric Time Series*. Hoboken, NJ: John Wiley & Sons, 1995.

[3] Hamilton, J. D. *Time Series Analysis*. Princeton, NJ: Princeton University Press, 1994.

Alternatives

- `filter` generalizes simulate. That is, both filter a series of disturbances to produce output responses, innovations, and conditional variances. However, `simulate` autogenerates a series of mean zero, unit variance, independent and identically distributed (iid) disturbances according to the distribution in `Mdl`. In contrast, `filter` lets you directly specify your own disturbances.

See Also

arima | estimate | forecast | impulse | infer | print | simulate

More About

- "Monte Carlo Simulation of Conditional Mean Models" on page 5-146
- "Presample Data for Conditional Mean Model Simulation" on page 5-149

filter

Class: LagOp

Apply lag operator polynomial to filter time series

Syntax

```
[Y,times] = filter(A,X)
[Y,times] = filter(A,X,'Initial',X0)
```

Description

Given a lag operator polynomial *A(L)*, `[Y,times] = filter(A,X)` applies *A(L)* to time series data *X(t)*. This is equivalent to applying a linear filter to *X(t)*, producing the filtered output series *Y(t) = A(L)X(t)*.

`[Y,times] = filter(A,X,'Initial',X0)` applies *A(L)* to time series data *X(t)* with specified presample values of the input time series *X(t)*.

Input Arguments

`A`

Lag operator polynomial object, as produced by `LagOp`.

`X`

`numObs`-by-*`numDims`* matrix of time series data to which the lag operator polynomial *`A`* is applied. The last observation is assumed to be the most recent. *`numDims`* is the dimension of *`A`*, unless *`X`* is a row vector, in which case *`X`* is treated as a univariate series. For univariate *`X`*, the orientation of the output *`Y`* is determined by the orientation of the input X.

`'Initial'`

Presample values of the input time series *X(t)*. If `'Initial'` is unspecified, or if the number of presample values is insufficient to initialize filtering, values are taken from

the beginning of `X`, reducing the effective sample size of the output `Y`. For convenience, scalar presample values are expanded to provide all *`numPresampleObs`*-by-*`numDims`* presample values, and data is not taken from `X`. If more presample values are specified than necessary, only the most recent values are used. For univariate `X`, presample values can be a row or a column vector.

Output Arguments

`Y`

Filtered input time series, $Y(t) = A(L)X(t)$.

`times`

Vector of relative time indices the same length as `Y`. Times are expressed relative to, or as an offset from, observations times 0, 1, 2,...,`numObs`–1 for the input series $X(t)$. For a polynomial of degree p, $Y(0)$ is a linear combination of $X(t)$ for times $t = 0, -1, -2,...,-p$ (presample data). $Y(t)$ for $t > 0$ is a linear combination of $X(t)$ for times $t = t, t-1, t-2,...,t-p$.

Examples

Filter a Series Through a Lag Polynomial

Create a `LagOp` polynomial and a random time series:

```
rng('default')                % Make output reproducible

A = LagOp({1 -0.6 0.08 0.2}, 'Lags', [0 1 2 4]);
X = randn(10, A.Dimension);
```

Filter the input time series with no explicit initial observations, allowing the `filter` method to automatically strip all required initial data from the beginning of the input time series $X(t)$.

```
[Y1,T1] = filter(A, X);
```

Manually strip all required presample observations directly from the beginning of $X(t)$, then pass in the reduced-length $X(t)$ and the stripped presample observations directly to

the `filter` method. In this case, the first 4 observations of $X(t)$ are stripped because the degree of the lag operator polynomial created below is 4.

```
[Y2,T2] = filter(A, X((A.Degree + 1):end,:), ...
     'Initial', X(1:A.Degree,:));
```

Manually strip part of the required presample observations from the beginning of $X(t)$ and let the `filter` method automatically strip the remaining observations from $X(t)$.

```
[Y3,T3] = filter(A, X((A.Degree - 1):end,:), ...
     'Initial', X(1:A.Degree - 2,:));
```

The filtered output series are all the same. However, the associated time vectors are not.

```
disp([T1 T2 T3])
```

```
     4     0     2
     5     1     3
     6     2     4
     7     3     5
     8     4     6
     9     5     7
```

Algorithms

Filtering is limited to single paths, so matrix data are assumed to be a single path of a multidimensional process, and 3-D data (multiple paths of a multidimensional process) are not allowed.

See Also

`mldivide`

filter

Class: regARIMA

Filter disturbances through regression model with ARIMA errors

Syntax

```
[Y,E,U] = filter(Mdl,Z)
[Y,E,U] = filter(Mdl,Z,Name,Value)
```

Description

`[Y,E,U] = filter(Mdl,Z)` filters errors to produce responses, innovations, and unconditional disturbances of a univariate regression model with ARIMA time series errors.

`[Y,E,U] = filter(Mdl,Z,Name,Value)` filters errors using additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

`Mdl`

Regression model with ARIMA errors, specified as a model returned by regARIMA or estimate.

The parameters of `Mdl` cannot contain `NaN`s.

`Z`

Errors that drive the innovation process, specified as a `numObs`-by-`numPaths` matrix. That is, $\varepsilon_t = \sigma z_t$ is the innovations process, where σ is the innovation standard deviation and z_t are the errors for $t = 1,...,T$.

As a column vector, `Z` represents a path of the underlying error series. As a matrix, `Z` represents `numObs` observations of `numPaths` paths of the underlying errors. `filter` assumes that observations across any row occur simultaneously. The last row contains the latest observation. `Z` is a continuation of the presample errors, `Z0`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'U0'`

Presample unconditional disturbances that provide initial values for the ARIMA error model, specified as the comma-separated pair consisting of `'U0'` and a column vector or matrix.

- If `U0` is a column vector, then `filter` applies it to each output path.
- If `U0` is a matrix, then it requires at least `numPaths` columns. If the number of columns exceeds `numPaths`, then `filter` uses the first `numPaths` columns.
- `U0` requires enough rows to initialize the compound autoregressive component of the ARIMA error model. The required number of rows is at least `Mdl.P`. If the number of rows in `U0` exceeds the number necessary, then `filter` uses the latest `Mdl.P` presample unconditional disturbances. The last row contains the latest presample unconditional disturbance.

Default: `filter` sets the necessary presample unconditional disturbances to 0.

`'X'`

Predictor data in the regression model, specified as the comma-separated pair consisting of `'X'` and a matrix.

The columns of `X` are separate, synchronized time series, with the last row containing the latest observations. The number of rows of `X` must be at least `numObs`. If the number of rows of `X` exceeds the number necessary, then `filter` uses the latest observations.

Default: `filter` does not include a regression component in the model regardless of the presence of regression coefficients in `Mdl`.

`'Z0'`

Presample errors providing initial values for the input error series, `Z`, specified as the comma-separated pair consisting of `'Z0'` and a vector or matrix.

- If `Z0` is a column vector, then `filter` applies it to each output path.

- If `Z0` is a matrix, then it requires at least `numPaths` columns. If the number of columns exceeds `numPaths`, then `filter` uses the first `numPaths` columns.
- `Z0` requires enough rows to initialize the compound moving average component of the ARIMA error model. The required number of rows is at least `model.Q`. If the number of rows in `Z0` exceeds the number necessary, then `filter` uses the latest `Mdl.Q` observations. The last row contains the latest observation.

Default: `filter` sets the necessary presample errors to 0.

Notes

- `NaN`s in `Z`, `U0`, `X`, and `Z0` indicate missing values and `filter` removes them. The software merges the presample data sets (`U0` and `Z0`), then uses list-wise deletion to remove any `NaN`s. `filter` similarly removes `NaN`s from the effective sample data (`Z` and `X`). Removing `NaN`s in the data reduces the sample size. Such removal can also create irregular time series.
- Removing `NaN`s in the main data reduces the effective sample size. Such removal can also create irregular time series.
- `filter` assumes that you synchronize presample data such that the latest observation of each presample series occurs simultaneously.
- All predictor series (i.e. columns) in `X` are associated with each error series in `Z` to produce `numPaths` response series `Y`.

Output Arguments

`Y`

Simulated responses, returned as a `numobs`-by-`numPaths` matrix.

`E`

Simulated, mean 0 innovations of the ARIMA error model, returned as a `numobs`-by-`numPaths` matrix.

`U`

Simulated unconditional disturbances, returned as a `numobs`-by-`numPaths` matrix.

Examples

Compare Responses from `filter` and `simulate`

Simulate responses using `filter` and `simulate`. Then compare the simulated responses.

Both `filter` and `simulate` filter a series of errors to produce output responses `y`, innovations `e`, and unconditional disturbances `u`. The difference is that `simulate` generates errors from `Mdl.Distribution`, whereas `filter` accepts a random array of errors that you generate from any distribution.

Specify the following regression model with ARMA(2,1) errors:

$$y_t = X_t \begin{bmatrix} 0.1 \\ -0.2 \end{bmatrix} + u_t$$
$$u_t = 0.5u_{t-1} - 0.8u_{t-2} + \varepsilon_t - 0.5\varepsilon_{t-1},$$

where ε_t is Gaussian with variance 0.1.

```
Mdl = regARIMA('Intercept',0,'AR',{0.5 -0.8}, ...
    'MA',-0.5,'Beta',[0.1 -0.2],'Variance',0.1);
```

Simulate data for the predictors and from `Mdl` using Monte Carlo simulation.

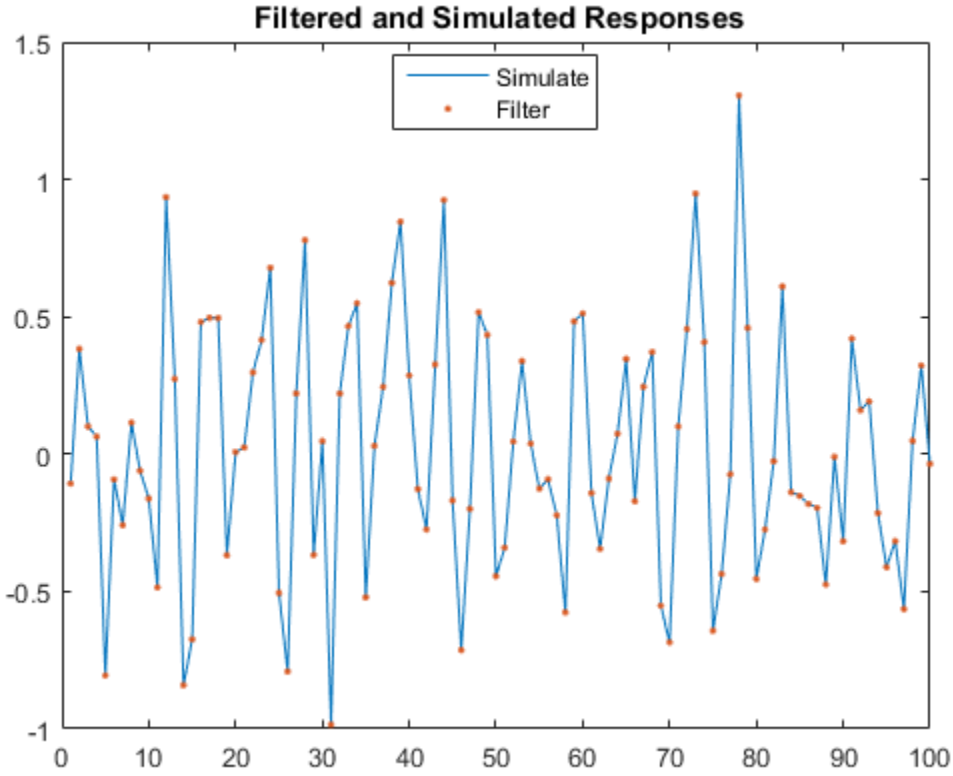
```
rng(1);           % For reproducibility
X = randn(100,2); % Simulate predictor data
[ySim,eSim,uSim] = simulate(Mdl,100,'X',X);
```

Standardize the simulated innovations and filter them.

```
z1 = eSim./sqrt(Mdl.Variance);
[yFlt1,eFlt1,uFlt1] = filter(Mdl,z1,'X',X);
```

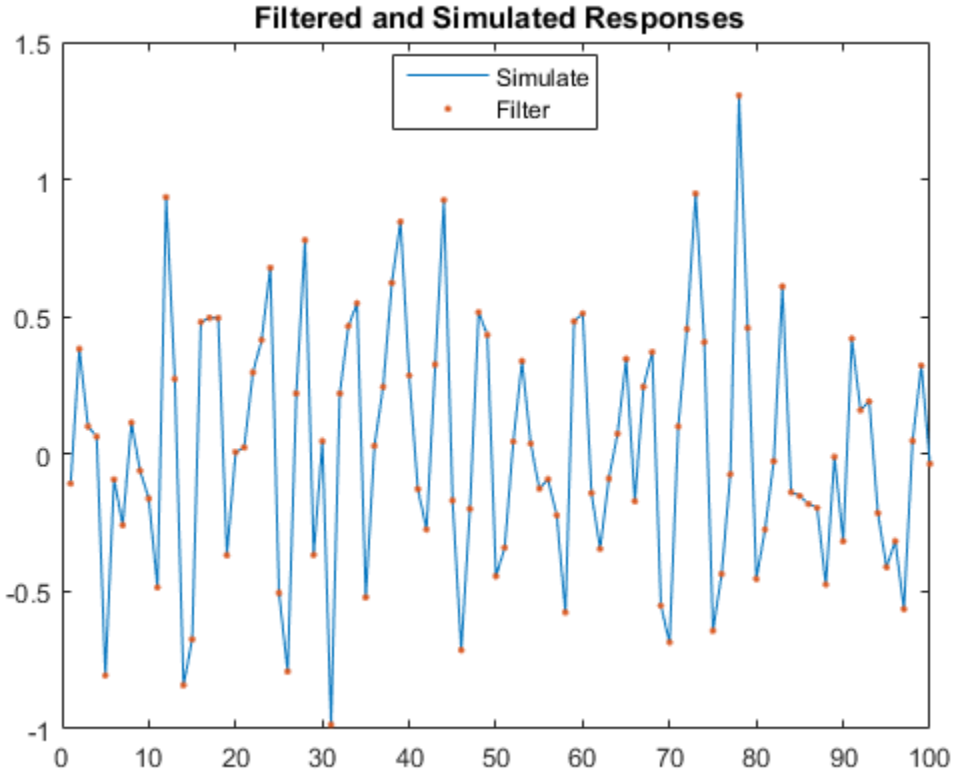
Confirm that the simulated responses from `simulate` and `filter` are identical using a plot.

```
figure
h1 = plot(ySim);
hold on
h2 = plot(yFlt1,'.');
title '{\bf Filtered and Simulated Responses}';
legend([h1, h2],'Simulate','Filter','Location','Best')
hold off
```



Alternatively, simulate responses by randomly generating your own errors and passing them into `filter`.

```
rng(1);
X = randn(100,2);
z2 = randn(100,1);
yFlt2 = filter(Mdl,z2,'X',X);
figure
h1 = plot(ySim);
hold on
h2 = plot(yFlt2,'.');
title '{\bf Filtered and Simulated Responses}';
legend([h1, h2],'Simulate','Filter','Location','Best')
hold off
```



This plot is the same as the previous plot, confirming that both simulation methods are equivalent.

`filter` multiplies the error, `Z`, by `sqrt(Mdl.Variance)` before filtering `Z` through the model. Therefore, if you want to specify your own distribution, set `Mdl.Variance` to 1, and then generate your own errors using, for example, `random('unif',a,b)` for the Uniform(a, b) distribution.

Simulate an Impulse Response Function

Simulate the impulse response of an innovation shock to the regression model with ARMA(2,1) errors.

The impulse response assesses the dynamic behavior of a system to a one-time shock. Typically, the magnitude of the shock is 1. Alternatively, it might be more meaningful to examine an impulse response of an innovation shock with a magnitude of one standard deviation.

In regression models with ARIMA errors,

- The impulse response function is invariant to the behavior of the predictors and the intercept.
- The impulse response of the model is defined as the impulse response of the unconditional disturbances as governed by the ARIMA error component.

Specify the following regression model with ARMA(2,1) errors:

$$\begin{aligned} y_t &= u_t \\ u_t &= 0.5u_{t-1} - 0.8u_{t-2} + \varepsilon_t - 0.5\varepsilon_{t-1}, \end{aligned}$$

where ε_t is Gaussian with variance 0.1.

```
Mdl = regARIMA('Intercept', 0, 'AR', {0.5 -0.8}, ...
    'MA', -0.5,'Variance',0.1);
```

When you construct an impulse response function for a regression model with ARIMA errors, you must set `Intercept` to 0.

Simulate the first 30 responses of the impulse response function by generating a error series with a one-time impulse with magnitude equal to one standard deviation, and then filter it. Also, use `impulse` to compute the impulse response function.

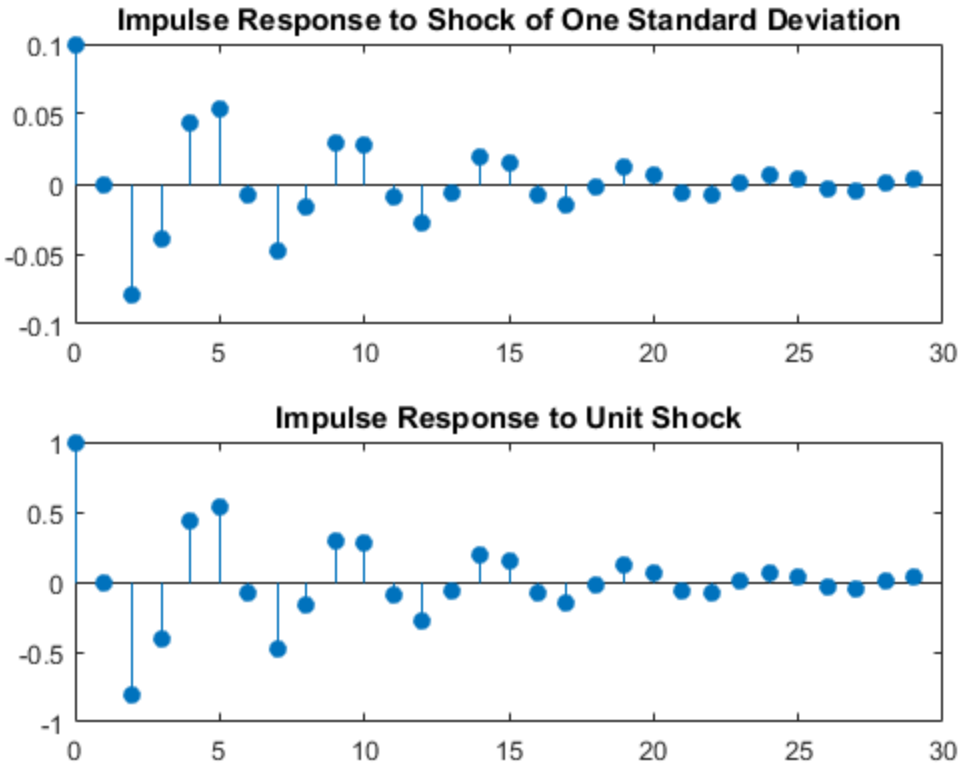
```
z = [sqrt(Mdl.Variance);zeros(29,1)]; % Shock of 1 std
yFltr  = filter(Mdl,z);
yImpls = impulse(Mdl,30);
```

When you construct an impulse response function for a regression model with ARIMA errors containing a regression component, do not specify the predictor matrix, `X`, in `filter`.

Plot the impulse response functions.

```
figure
subplot(2,1,1)
```

```
stem((0:numel(yFltr)-1)',yFltr,'filled')
title...
    ('Impulse Response to Shock of One Standard Deviation');
subplot(2,1,2)
stem((0:numel(yImpls)-1)',yImpls,'filled')
title 'Impulse Response to Unit Shock';
```



The impulse response function given a shock of one standard deviation is a scaled version of the impulse response returned by `impulse`.

Simulate a Step Response

Simulate the step response function of a regression model with ARMA(2,1) errors.

The step response assesses the dynamic behavior of a system to a persistent shock. Typically, the magnitude of the shock is 1. Alternatively, it might be more meaningful to examine a step response of a persistent innovation shock with a magnitude of one standard deviation. This example plots the step response of a persistent innovations shock in a model without an intercept and predictor matrix for regression. However, note that `filter` is flexible in that it accepts a persistent innovations or predictor shock that you construct using any magnitude, then filters it through the model.

Specify the following regression model with ARMA(2,1) errors:

$$\begin{aligned} y_t &= u_t \\ u_t &= 0.5u_{t-1} - 0.8u_{t-2} + \varepsilon_t - 0.5\varepsilon_{t-1}, \end{aligned}$$

where ε_t is Gaussian with variance 0.1.

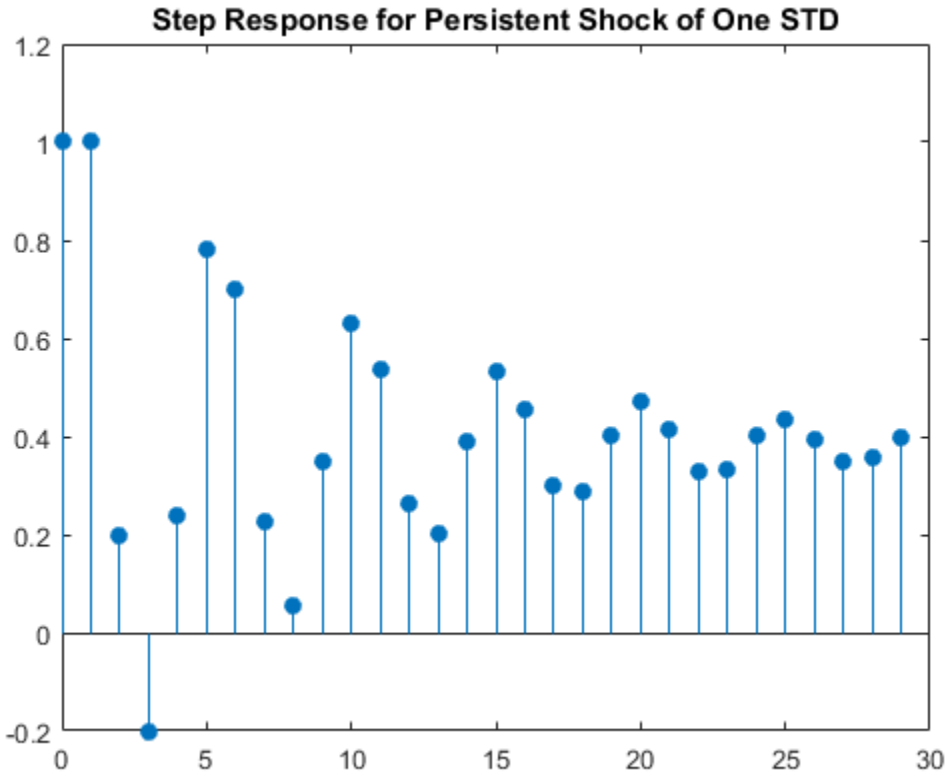
```
Mdl = regARIMA('Intercept', 0, 'AR', {0.5 -0.8}, ...
    'MA', -0.5,'Variance',0.1);
```

Simulate the first 30 responses to a sequence of unit errors by generating an error series of one standard deviation, and then filtering it.

```
z = sqrt(Mdl.Variance)*ones(30,1);...
    % Persistant shock of one std
y = filter(Mdl,z);
y = y/y(1);  % Normalize relative to y(1)
```

Plot the step response function.

```
figure
stem((0:numel(y)-1)',y,'filled')
title('Step Response for Persistent Shock of One STD')
```



The step response settles around 0.4.

References

[1] Box, G. E. P., G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis: Forecasting and Control*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

[2] Davidson, R., and J. G. MacKinnon. *Econometric Theory and Methods*. Oxford, UK: Oxford University Press, 2004.

[3] Enders, W. *Applied Econometric Time Series*. Hoboken, NJ: John Wiley & Sons, Inc., 1995.

[4] Hamilton, J. D. *Time Series Analysis*. Princeton, NJ: Princeton University Press, 1994.

[5] Pankratz, A. *Forecasting with Dynamic Regression Models*. John Wiley & Sons, Inc., 1991.

[6] Tsay, R. S. *Analysis of Financial Time Series*. 2nd ed. Hoboken, NJ: John Wiley & Sons, Inc., 2005.

Alternatives

- `filter` generalizes simulate. Both filter a series of errors to produce responses (`Y`), innovations (`E`), and unconditional disturbances (`U`). However, `simulate` autogenerates a series of mean zero, unit variance, independent and identically distributed (iid) errors according to the distribution in `Mdl`. In contrast, `filter` requires that you specify your own errors, which can come from any distribution.

See Also

regARIMA | simulate

More About

- "Monte Carlo Simulation of Conditional Mean Models" on page 5-146
- "Presample Data for Conditional Mean Model Estimation" on page 5-103

filter

Class: dssm

Forward recursion of diffuse state-space models

Syntax

```
X = filter(Mdl,Y)
X = filter(Mdl,Y,Name,Value)
[X,logL,Output] = filter( ___ )
```

Description

`X = filter(Mdl,Y)` returns filtered states (`X`) by performing forward recursion of the fully specified diffuse state-space model `Mdl`. That is, `filter` applies the diffuse Kalman filter using `Mdl` and the observed responses `Y`.

`X = filter(Mdl,Y,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments. For example, specify the regression coefficients and predictor data to deflate the observations, or specify to use the univariate treatment of a multivariate model.

If `Mdl` is not fully specified, then you must specify the unknown parameters as known scalars using the `'Params' Name,Value` pair argument.

`[X,logL,Output] = filter( ___ )` additionally returns the loglikelihood value (`logL`) and an output structure array (`Output`) using any of the input arguments in the previous syntaxes. `Output` contains:

- Filtered and forecasted states
- Estimated covariance matrices of the filtered and forecasted states
- Loglikelihood value
- Forecasted observations and its estimated covariance matrix
- Adjusted Kalman gain

- Vector indicating which data the software used to filter

Tips

- `Mdl` does not store the response data, predictor data, and the regression coefficients. Supply the data wherever necessary using the appropriate input or name-value pair arguments.
- It is a best practice to allow `dssm.filter` to determine the value of `SwitchTime`. However, in rare cases, you might experience numerical issues during estimation, filtering, or smoothing diffuse state-space models. For such cases, try experimenting with various `SwitchTime` specifications, or consider a different model structure (e.g., simplify or reverify the model). For example, convert the diffuse state-space model to a standard state-space model using ssm.
- To accelerate estimation for low-dimensional, time-invariant models, set `'Univariate',true`. Using this specification, the software sequentially updates rather then updating all at once during the filtering process.

Input Arguments

`Mdl` — Diffuse state-space model

`dssm` model object

Diffuse state-space model, specified as an `dssm` model object returned by dssm or estimate.

If `Mdl` is not fully specified (that is, `Mdl` contains unknown parameters), then specify values for the unknown parameters using the `'Params'` name-value pair argument. Otherwise, the software issues an error. `estimate` returns fully-specified state-space models.

`Mdl` does not store observed responses or predictor data. Supply the data wherever necessary using the appropriate input or name-value pair arguments.

`Y` — Observed response data

numeric matrix | cell vector of numeric vectors

Observed response data to which `Mdl` is fit, specified as a numeric matrix or a cell vector of numeric vectors.

- If `Mdl` is time invariant with respect to the observation equation, then `Y` is a T-by-n matrix, where each row corresponds to a period and each column corresponds to a particular observation in the model. T is the sample size and m is the number of observations per period. The last row of `Y` contains the latest observations.
- If `Mdl` is time varying with respect to the observation equation, then `Y` is a T-by-1 cell vector. Each element of the cell vector corresponds to a period and contains an n_t-dimensional vector of observations for that period. The corresponding dimensions of the coefficient matrices in `Mdl.C{t}` and `Mdl.D{t}` must be consistent with the matrix in `Y{t}` for all periods. The last cell of `Y` contains the latest observations.

`NaN` elements indicate missing observations. For details on how the Kalman filter accommodates missing observations, see "Algorithms" on page 9-450.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'Beta'` — Regression coefficients

[ ] (default) | numeric matrix

Regression coefficients corresponding to predictor variables, specified as the comma-separated pair consisting of `'Beta'` and a d-by-n numeric matrix. d is the number of predictor variables (see `Predictors`) and n is the number of observed response series (see `Y`).

If `Mdl` is an estimated state-space model, then specify the estimated regression coefficients stored in `estParams`.

`'Params'` — Values for unknown parameters

numeric vector

Values for unknown parameters in the state-space model, specified as the column-separated pair consisting of `'Params'` and a numeric vector.

The elements of `Params` correspond to the unknown parameters in the state-space model matrices `A`, `B`, `C`, and `D`, and, optionally, the initial state mean `Mean0` and covariance matrix `Cov0`.

- If you created `Mdl` explicitly (that is, by specifying the matrices without a parameter-to-matrix mapping function), then the software maps the elements of `Params` to `NaN`s in the state-space model matrices and initial state values. The software searches for `NaN`s column-wise following the order `A`, `B`, `C`, `D`, `Mean0`, and `Cov0`.
- If you created `Mdl` implicitly (that is, by specifying the matrices with a parameter-to-matrix mapping function), then you must set initial parameter values for the state-space model matrices, initial state values, and state types within the parameter-to-matrix mapping function.

If `Mdl` contains unknown parameters, then you must specify their values. Otherwise, the software ignores the value of `Params`.

Data Types: `double`

`'Predictors'` — Predictor variables in state-space model observation equation

`[ ]` (default) | numeric matrix

Predictor variables in the state-space model observation equation, specified as the comma-separated pair consisting of `'Predictors'` and a T-by-d numeric matrix. T is the number of periods and d is the number of predictor variables. Row t corresponds to the observed predictors at period t (Z_t). The expanded observation equation is

$$y_t - Z_t\beta = Cx_t + Du_t.$$

That is, the software deflates the observations using the regression component. β is the time-invariant vector of regression coefficients that the software estimates with all other parameters.

If there are n observations per period, then the software regresses all predictor series onto each observation.

If you specify `Predictors`, then `Mdl` must be time invariant. Otherwise, the software returns an error.

By default, the software excludes a regression component from the state-space model.

Data Types: `double`

`'SwitchTime'` — Final period for diffuse state initialization

positive integer

Final period for diffuse state initialization, specified as the comma-separated pair consisting of `'SwitchTime'` and a positive integer. That is, `estimate` uses the observations from period 1 to period `SwitchTime` as a presample to implement the *exact initial Kalman filter* (see "Diffuse Kalman Filter" on page 8-15 and [1]). After initializing the diffuse states, `estimate` applies the standard Kalman filter to the observations from periods `SwitchTime` + 1 to T.

The default value for `SwitchTime` is the last period in which the estimated smoothed state precision matrix is singular (i.e., the inverse of the covariance matrix). This specification represents the fewest number of observations required to initialize the diffuse states. Therefore, it is a best practice to use the default value.

If you set `SwitchTime` to a value greater than the default, then the effective sample size decreases. If you set `SwitchTime` to a value that is fewer than the default, then `estimate` might not have enough observations to initialize the diffuse states, which can result in an error or improper values.

In general, estimating, filtering, and smoothing state-space models with at least one diffuse state requires `SwitchTime` to be at least one. The default estimation display contains the effective sample size.

Data Types: `double`

`'Tolerance'` — Forecast uncertainty threshold
`0` (default) | nonnegative scalar

Forecast uncertainty threshold, specified as the comma-separated pair consisting of `'Tolerance'` and a nonnegative scalar.

If the forecast uncertainty for a particular observation is less than `Tolerance` during numerical estimation, then the software removes the uncertainty corresponding to the observation from the forecast covariance matrix before its inversion.

It is best practice to set `Tolerance` to a small number, for example, `1e-15`, to overcome numerical obstacles during estimation.

Example: `'Tolerance',1e-15`

Data Types: `double`

`'Univariate'` — Univariate treatment of multivariate series flag
`false` (default) | `true`

Univariate treatment of a multivariate series flag, specified as the comma-separated pair consisting of `'Univariate'` and `true` or `false`. Univariate treatment of a multivariate series is also known as *sequential filtering*.

The univariate treatment can accelerate and improve numerical stability of the Kalman filter. However, all observation innovations must be uncorrelated. That is, D_tD_t' must be diagonal, where D_t, $t = 1,...,T$, is one of the following:

- The matrix `D{t}` in a time-varying state-space model
- The matrix `D` in a time-invariant state-space model

Example: `'Univariate',true`

Data Types: `logical`

Output Arguments

`X` — Filtered states

numeric matrix | cell vector of numeric vectors

Filtered states, returned as a numeric matrix or a cell vector of numeric vectors.

If `Mdl` is time invariant, then the number of rows of `X` is the sample size, T, and the number of columns of `X` is the number of states, m. The last row of `X` contains the latest, filtered states.

If `Mdl` is time varying, then `X` is a cell vector with length equal to the sample size. Cell t of `X` contains a vector of filtered states with length equal to the number of states in period t. The last cell of `X` contains the latest, filtered states.

`filter` pads the first `SwitchTime` periods of `X` with zeros or empty cells. The zeros or empty cells represent the periods required to initialize the diffuse states.

`logL` — Loglikelihood function value

scalar

Loglikelihood function value, returned as a scalar.

Missing observations and observations before `SwitchTime` do not contribute to the loglikelihood.

`Output` — Filtering results by period

structure array

Filtering results by period, returned as a structure array.

`Output` is a T-by-1 structure, where element t corresponds to the filtering result at time t.

- If `Univariate` is `false` (it is by default), then the following table outlines the fields of `Output`.

Field	Description	Estimate of
`LogLikelihood`	Scalar loglikelihood objective function value	N/A
`FilteredStates`	m_t-by-1 vector of filtered states	$E(x_t \mid y_1,\ldots,y_t)$
`FilteredStatesCov`	m_t-by-m_t variance-covariance matrix of filtered states	$Var(x_t \mid y_1,\ldots,y_t)$
`ForecastedStates`	m_t-by-1 vector of state forecasts	$E(x_t \mid y_1,\ldots,y_{t-1})$
`ForecastedStatesCov`	m_t-by-m_t variance-covariance matrix of state forecasts	$Var(x_t \mid y_1,\ldots,y_{t-1})$
`ForecastedObs`	h_t-by-1 forecasted observation vector	$E(y_t \mid y_1,\ldots,y_{t-1})$
`ForecastedObsCov`	h_t-by-h_t variance-covariance matrix of forecasted observations	$Var(y_t \mid y_1,\ldots,t_{t-1})$
`KalmanGain`	m_t-by-n_t adjusted Kalman gain matrix	N/A
`DataUsed`	h_t-by-1 logical vector indicating whether the software filters using a particular observation. For example, if observation i at time t is a `NaN`, then	N/A

Field	Description	Estimate of
	element i in `DataUsed` at time t is `0`.	

- If `Univariate` is `true`, then the fields of `Output` are the same as in the previous table, except for the following amendments.

Field	Changes
`ForecastedObs`	Same dimensions as if `Univariate = 0`, but only the first elements are equal
`ForecastedObsCov`	n-by-1 vector of forecasted observation variances. The first element of this vector is equivalent to `ForecastedObsCov(1,1)` when `Univariate` is `false`. The rest of the elements are not necessarily equivalent to their corresponding values in `ForecastObsCov` when `Univariate`.
`KalmanGain`	Same dimensions as if `Univariate` is `false`, though `KalmanGain` might have different entries.

`filter` pads the first `SwitchTime` periods of the fields of `Output` with empty cells. These empty cells represent the periods required to initialize the diffuse states.

Examples

Filter States of Time-Invariant Diffuse State-Space Model

Suppose that a latent process is a random walk. Subsequently, the state equation is

$$x_t = x_{t-1} + u_t,$$

where u_t is Gaussian with mean 0 and standard deviation 1.

Generate a random series of 100 observations from x_t, assuming that the series starts at 1.5.

```
T = 100;
x0 = 1.5;
rng(1); % For reproducibility
u = randn(T,1);
x = cumsum([x0;u]);
x = x(2:end);
```

Suppose further that the latent process is subject to additive measurement error. Subsequently, the observation equation is

$$y_t = x_t + \varepsilon_t,$$

where ε_t is Gaussian with mean 0 and standard deviation 0.75. Together, the latent process and observation equations compose a state-space model.

Use the random latent state process (`x`) and the observation equation to generate observations.

```
y = x + 0.75*randn(T,1);
```

Specify the four coefficient matrices.

```
A = 1;
B = 1;
C = 1;
D = 0.75;
```

Create the diffuse state-space model using the coefficient matrices. Specify that the inital state distribution is diffuse.

```
Mdl = dssm(A,B,C,D,'StateType',2)
```

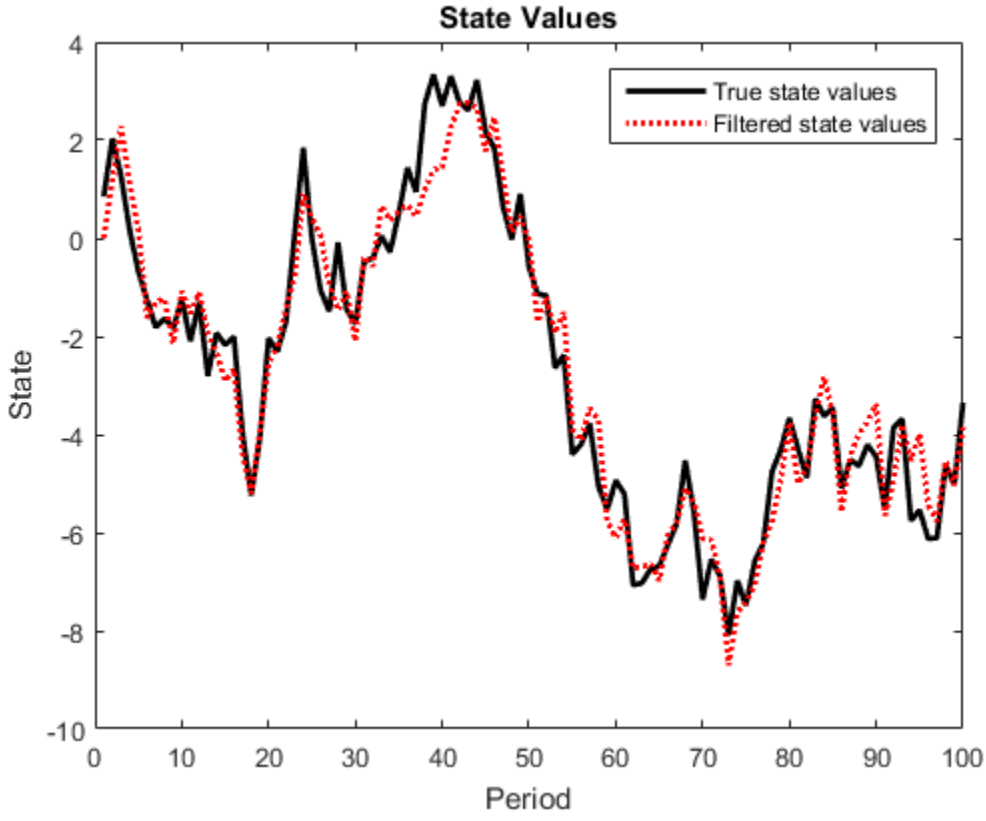
```
Mdl =


State-space model type: <a href="matlab: doc dssm">dssm</a>

State vector length: 1
Observation vector length: 1
State disturbance vector length: 1
Observation innovation vector length: 1
```

```
Sample size supported by model: Unlimited

State variables: x1, x2,...
State disturbances: u1, u2,...
Observation series: y1, y2,...
Observation innovations: e1, e2,...

State equation:
x1(t) = x1(t-1) + u1(t)

Observation equation:
y1(t) = x1(t) + (0.75)e1(t)

Initial state distribution:

Initial state means
 x1
  0

Initial state covariance matrix
     x1
 x1  Inf

State types
    x1
 Diffuse
```

`Mdl` is an `dssm` model. Verify that the model is correctly specified using the display in the Command Window.

Filter states for periods 1 through 100. Plot the true state values and the filtered state estimates.

```
filteredX = filter(Mdl,y);

figure
plot(1:T,x,'-k',1:T,filteredX,':r','LineWidth',2)
title({'State Values'})
xlabel('Period')
ylabel('State')
legend({'True state values','Filtered state values'})
```



The true values and filter estimates are approximately the same, except for the first filtered state, which is zero.

Filter States of Diffuse State-Space Model Containing Regression Component

Suppose that the linear relationship between unemployment rate and the nominal gross national product (nGNP) is of interest. Suppose further that unemployment rate is an AR(1) series. Symbolically, and in state-space form, the model is

$$x_t = \phi x_{t-1} + \sigma u_t$$
$$y_t - \beta Z_t = x_t,$$

where:

- x_t is the unemployment rate at time t.
- y_t is the observed change in the unemployment rate being deflated by the return of nGNP (Z_t).
- u_t is the Gaussian series of state disturbances having mean 0 and unknown standard deviation σ.

Load the Nelson-Plosser data set, which contains the unemployment rate and nGNP series, among other things.

```
load Data_NelsonPlosser
```

Preprocess the data by taking the natural logarithm of the nGNP series, and removing the starting `NaN` values from each series.

```
isNaN = any(ismissing(DataTable),2);       % Flag periods containing NaNs
gnpn = DataTable.GNPN(~isNaN);
y = diff(DataTable.UR(~isNaN));
T = size(gnpn,1);                          % The sample size
Z = price2ret(gnpn);
```

This example continues using the series without `NaN` values. However, using the Kalman filter framework, the software can accommodate series containing missing values.

Specify the coefficient matrices.

```
A = NaN;
B = NaN;
C = 1;
```

Create the state-space model using `dssm` by supplying the coefficient matrices and specifying that the state values come from a diffuse distribution. The diffuse specification indicates complete ignorance about the moments of the initial distribution.

```
StateType = 2;
Mdl = dssm(A,B,C,'StateType',StateType);
```

Estimate the parameters. Specify the regression component and its initial value for optimization using the `'Predictors'` and `'Beta0'` name-value pair arguments, respectively. Display the estimates and all optimization diagnostic information. Restrict the estimate of σ to all positive, real numbers.

```
params0 = [0.3 0.2]; % Initial values chosen arbitrarily
```

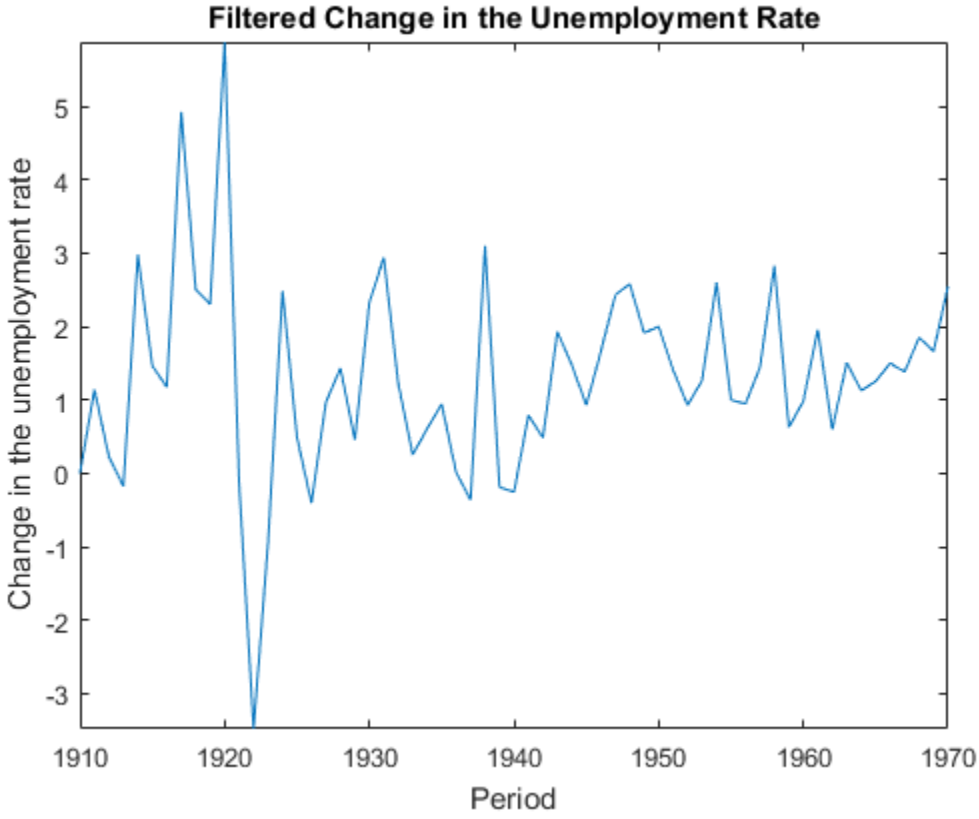
```
Beta0 = 0.1;
[EstMdl,estParams] = estimate(Mdl,y,params0,'Predictors',Z,'Beta0',Beta0,...
    'lb',[-Inf 0 -Inf]);
```

```
Method: Maximum likelihood (fmincon)
Effective Sample size:             60
Logarithmic  likelihood:     -110.477
Akaike   info criterion:      226.954
Bayesian info criterion:      233.287
           |      Coeff       Std Err    t Stat    Prob
-------------------------------------------------------
 c(1)      |   0.59436       0.09408     6.31738  0
 c(2)      |   1.52554       0.10758    14.17991  0
 y <- z(1) | -24.26161       1.55730   -15.57930  0
           |
           |    Final State   Std Dev     t Stat   Prob
 x(1)      |   2.54764        0           Inf     0
```

`EstMdl` is an `ssm` model, and you can access its properties using dot notation.

Filter the estimated diffuse state-space model. `EstMdl` does not store the data or the regression coefficients, so you must pass in them in using the name-value pair arguments `'Predictors'` and `'Beta'`, respectively. Plot the estimated, filtered states.

```
filteredX = filter(EstMdl,y,'Predictors',Z,'Beta',estParams(end));

figure
plot(dates(end-(T-1)+1:end),filteredX);
xlabel('Period')
ylabel('Change in the unemployment rate')
title('Filtered Change in the Unemployment Rate')
axis tight
```



Extract Other Estimates from `Output`

Estimate a diffuse state-space model, filter the states, and then extract other estimates from the `Output` output argument.

Consider the diffuse state-space model

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix} = \begin{bmatrix} \phi & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \end{bmatrix} + \begin{bmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{bmatrix} \begin{bmatrix} u_{1,t} \\ u_{2,t} \end{bmatrix}$$
$$y_t = \begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix}$$

The state variable $x_{1,t}$ is an AR(1) model with autoregressive coefficient ϕ. $x_{2,t}$ is a random walk. The disturbances $u_{1,t}$ and $u_{2,t}$ are independent Gaussian random variables with mean 0 and standard deviations σ_1 and σ_2, respectively. The observation y_t is the error-free sum of $x_{1,t}$ and $x_{2,t}$.

Generate data from the state-space model. To simulate the data, suppose that the sample size $T = 100$, $\phi = 0.6$, $\sigma_1 = 0.2$, $\sigma_2 = 0.1$, and $x_{1,0} = x_{2,0} = 2$.

```
rng(1); % For reproducibility
T = 100;
ARMdl = arima('AR',0.6,'Constant',0,'Variance',0.2^2);
x1 = simulate(ARMdl,T,'Y0',2);
u3 = 0.1*randn(T,1);
x3 = cumsum([2;u3]);
x3 = x3(2:end);
y = x1 + x3;
```

Specify the coefficient matrices of the state-space model. To indicate unknown parameters, use `NaN` values.

```
A = [NaN 0; 0 1];
B = [NaN 0; 0 NaN];
C = [1 1];
```

Create a diffuse state-space model that describes the model above. Specify that $x_{1,t}$ and $x_{2,t}$ have diffuse initial state distributions.

```
StateType = [2 2];
Mdl = dssm(A,B,C,'StateType',StateType);
```

Estimate the unknown parameters of `Mdl`. Choose initial parameter values for optimization. Specify that the standard deviations are constrained to be positive, but all other parameters are unconstrained using the `'lb'` name-value pair argument.

```
params0 = [0.01 0.1 0.01]; % Initial values chosen arbitrarily
EstMdl = estimate(Mdl,y,params0,'lb',[-Inf 0 0]);
```

```
Method: Maximum likelihood (fmincon)
Effective Sample size:             98
Logarithmic  likelihood:      3.44283
Akaike   info criterion:     -0.885655
```

```
Bayesian info criterion:      6.92986
      |     Coeff      Std Err   t Stat     Prob
-------------------------------------------------
 c(1) | 0.54134       0.20494    2.64145  0.00826
 c(2) | 0.18439       0.03305    5.57897   0
 c(3) | 0.11783       0.04347    2.71039  0.00672
      |
      |  Final State   Std Dev    t Stat    Prob
 x(1) | 0.24884       0.17168    1.44943  0.14722
 x(2) | 1.73762       0.17168   10.12121   0
```

The parameters are close to their true values.

Filter the states of `EstMdl`, and request all other available output.

```
[X,logL,Output] = filter(EstMdl,y);
```

`X` is a `T`-by-2 matrix of filtered states, `logL` is the final, optimized log-likelihood value, and `Output` is a structure array containing various estimates that the Kalman filter requires. List the fields of `output` using `fields`.

```
fields(Output)
```

```
ans =

    'LogLikelihood'
    'FilteredStates'
    'FilteredStatesCov'
    'ForecastedStates'
    'ForecastedStatesCov'
    'ForecastedObs'
    'ForecastedObsCov'
    'KalmanGain'
    'DataUsed'
```

Convert `Output` to a table.

```
OutputTbl = struct2table(Output);
OutputTbl(1:10,1:5) % Display first ten rows of first five variables
```

```
ans =
```

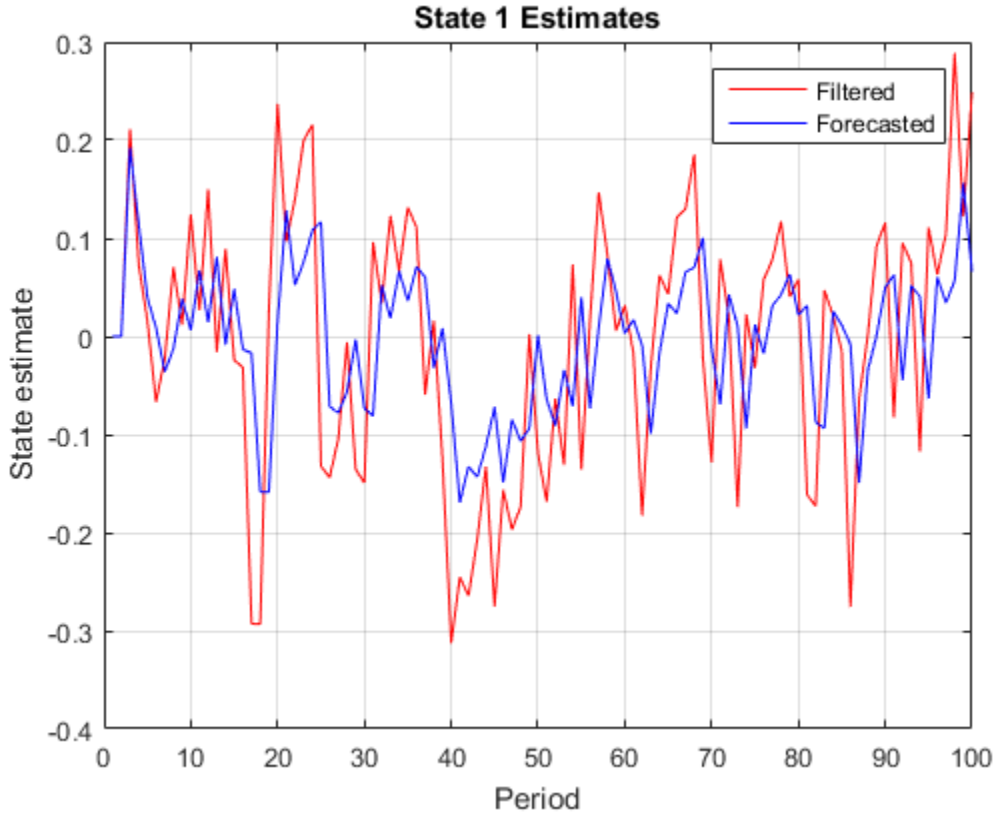
```
    LogLikelihood    FilteredStates    FilteredStatesCov    ForecastedStates    Forecas
    _____________    ______________    _________________    ________________    _______

    []               []                []                   []                  []
    []               []                []                   []                  []
    [ 0.1827]        [2x1 double]      [2x2 double]         [2x1 double]        [2x2 do
    [ 0.0972]        [2x1 double]      [2x2 double]         [2x1 double]        [2x2 do
    [ 0.4472]        [2x1 double]      [2x2 double]         [2x1 double]        [2x2 do
    [ 0.2073]        [2x1 double]      [2x2 double]         [2x1 double]        [2x2 do
    [ 0.5167]        [2x1 double]      [2x2 double]         [2x1 double]        [2x2 do
    [ 0.2389]        [2x1 double]      [2x2 double]         [2x1 double]        [2x2 do
    [ 0.5064]        [2x1 double]      [2x2 double]         [2x1 double]        [2x2 do
    [-0.0105]        [2x1 double]      [2x2 double]         [2x1 double]        [2x2 do
```

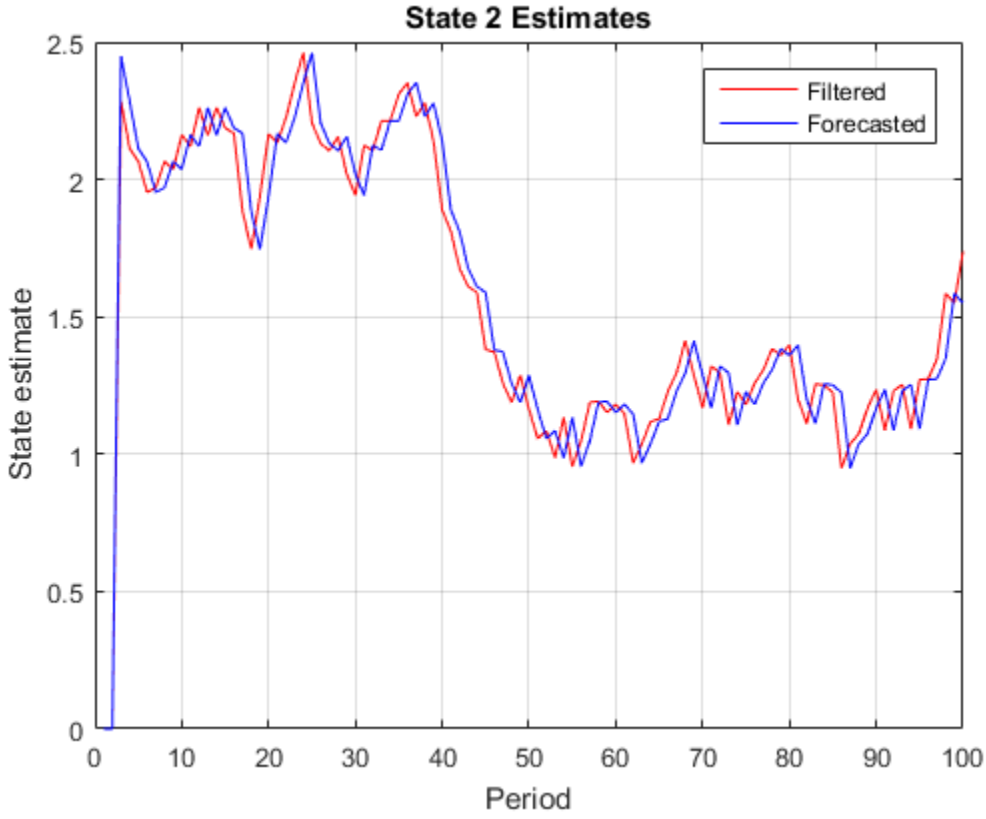
The first two rows of the table contain empty cells or zeros. These correspond to the observations required to initialize the diffuse Kalman filter. That is, `SwitchTime` is 2.

```
SwitchTime = 2;
```

Plot the filtered and forecasted states.

```
ForeX = cell2mat(OutputTbl.ForecastedStates')'; % Orient forecasted states
ForeX = [zeros(SwitchTime,2);ForeX]; % Include zeros for initialization

figure;
plot(1:T,X(:,1),'r',1:T,ForeX(:,1),'b');
xlabel('Period');
ylabel('State estimate');
title('State 1 Estimates')
legend('Filtered','Forecasted');
grid on;

figure;
plot(1:T,X(:,2),'r',1:T,ForeX(:,2),'b');
xlabel('Period');
ylabel('State estimate');
title('State 2 Estimates')
legend('Filtered','Forecasted');
grid on;
```


State 1 Estimates
Filtered
Forecasted
State estimate
Period
0.3
0.2
0.1
0
-0.1
-0.2
-0.3
-0.4
0
10
20
30
40
50
60
70
80
90
100



- "Filter Time-Varying Diffuse State-Space Model" on page 8-68
- "Smooth Time-Varying Diffuse State-Space Model" on page 8-91

Algorithms

- The Kalman filter accommodates missing data by not updating filtered state estimates corresponding to missing observations. In other words, suppose there is a missing observation at period t. Then, the state forecast for period t based on the previous $t - 1$ observations and filtered state for period t are equivalent.

- For explicitly defined state-space models, `filter` applies all predictors to each response series. However, each response series has its own set of regression coefficients.
- The diffuse Kalman filter requires presample data. If missing observations begin the time series, then the diffuse Kalman filter must gather enough nonmissing observations to initialize the diffuse states.
- For diffuse state-space models, `filter` usually switches from the diffuse Kalman filter to the standard Kalman filter when the number of cumulative observations and the number of diffuse states are equal. However, if a diffuse state-space model has identifiability issues (e.g., the model is too complex to fit to the data), then `filter` might require more observations to initialize the diffuse states. In extreme cases, `filter` requires the entire sample.

References

[1] Durbin J., and S. J. Koopman. *Time Series Analysis by State Space Methods*. 2nd ed. Oxford: Oxford University Press, 2012.

See Also

dssm | estimate | forecast | refine | smooth

More About

- "What Are State-Space Models?" on page 8-3
- "What Is the Kalman Filter?" on page 8-8

Introduced in R2015b

filter

Class: ssm

Forward recursion of state-space models

Syntax

```
X = filter(Mdl,Y)
X = filter(Mdl,Y,Name,Value)
[X,logL,Output] = filter( ___ )
```

Description

`X = filter(Mdl,Y)` returns filtered states (`X`) from performing forward recursion of the fully specified state-space model `Mdl`. That is, `filter` applies the standard Kalman filter using `Mdl` and the observed responses `Y`.

`X = filter(Mdl,Y,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments. For example, specify the regression coefficients and predictor data to deflate the observations, or specify to use the square-root filter.

If `Mdl` is not fully specified, then you must specify the unknown parameters as known scalars using the `'Params' Name,Value` pair argument.

`[X,logL,Output] = filter( ___ )` uses any of the input arguments in the previous syntaxes to additionally return the loglikelihood value (`logL`) and an output structure array (`Output`) using any of the input arguments in the previous syntaxes. `Output` contains:

- Filtered and forecasted states
- Estimated covariance matrices of the filtered and forecasted states
- Loglikelihood value
- Forecasted observations and its estimated covariance matrix
- Adjusted Kalman gain
- Vector indicating which data the software used to filter

Input Arguments

`Mdl` — Standard state-space model

`ssm` model object

Standard state-space model, specified as an `ssm` model object returned by ssm or estimate.

If `Mdl` is not fully specified (that is, `Mdl` contains unknown parameters), then specify values for the unknown parameters using the `'Params'` name-value pair argument. Otherwise, the software issues an error. `estimate` returns fully-specified state-space models.

`Mdl` does not store observed responses or predictor data. Supply the data wherever necessary using the appropriate input or name-value pair arguments.

`Y` — Observed response data

numeric matrix | cell vector of numeric vectors

Observed response data to which `Mdl` is fit, specified as a numeric matrix or a cell vector of numeric vectors.

- If `Mdl` is time invariant with respect to the observation equation, then `Y` is a T-by-n matrix, where each row corresponds to a period and each column corresponds to a particular observation in the model. T is the sample size and m is the number of observations per period. The last row of `Y` contains the latest observations.
- If `Mdl` is time varying with respect to the observation equation, then `Y` is a T-by-1 cell vector. Each element of the cell vector corresponds to a period and contains an n_t-dimensional vector of observations for that period. The corresponding dimensions of the coefficient matrices in `Mdl.C{t}` and `Mdl.D{t}` must be consistent with the matrix in `Y{t}` for all periods. The last cell of `Y` contains the latest observations.

`NaN` elements indicate missing observations. For details on how the Kalman filter accommodates missing observations, see “Algorithms” on page 9-450.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

'Beta' — Regression coefficients

[ ] (default) | numeric matrix

Regression coefficients corresponding to predictor variables, specified as the comma-separated pair consisting of 'Beta' and a d-by-n numeric matrix. d is the number of predictor variables (see Predictors) and n is the number of observed response series (see Y).

If Mdl is an estimated state-space model, then specify the estimated regression coefficients stored in estParams.

'Params' — Values for unknown parameters

numeric vector

Values for unknown parameters in the state-space model, specified as the column-separated pair consisting of 'Params' and a numeric vector.

The elements of Params correspond to the unknown parameters in the state-space model matrices A, B, C, and D, and, optionally, the initial state mean Mean0 and covariance matrix Cov0.

- If you created Mdl explicitly (that is, by specifying the matrices without a parameter-to-matrix mapping function), then the software maps the elements of Params to NaNs in the state-space model matrices and initial state values. The software searches for NaNs column-wise following the order A, B, C, D, Mean0, and Cov0.
- If you created Mdl implicitly (that is, by specifying the matrices with a parameter-to-matrix mapping function), then you must set initial parameter values for the state-space model matrices, initial state values, and state types within the parameter-to-matrix mapping function.

If Mdl contains unknown parameters, then you must specify their values. Otherwise, the software ignores the value of Params.

Data Types: double

'Predictors' — Predictor variables in state-space model observation equation

[ ] (default) | numeric matrix

Predictor variables in the state-space model observation equation, specified as the comma-separated pair consisting of 'Predictors' and a T-by-d numeric matrix. T is

the number of periods and d is the number of predictor variables. Row t corresponds to the observed predictors at period t (Z_t). The expanded observation equation is

$$y_t - Z_t\beta = Cx_t + Du_t.$$

That is, the software deflates the observations using the regression component. β is the time-invariant vector of regression coefficients that the software estimates with all other parameters.

If there are n observations per period, then the software regresses all predictor series onto each observation.

If you specify `Predictors`, then `Mdl` must be time invariant. Otherwise, the software returns an error.

By default, the software excludes a regression component from the state-space model.

Data Types: `double`

`'SquareRoot'` — Square root filter method flag

`false` (default) | `true`

Square root filter method flag, specified as the comma-separated pair consisting of `'SquareRoot'` and `true` or `false`. If `true`, then `estimate` applies the square root filter method when implementing the Kalman filter.

If you suspect that the eigenvalues of the filtered state or forecasted observation covariance matrices are close to zero, then specify `'SquareRoot',true`. The square root filter is robust to numerical issues arising from finite the precision of calculations, but requires more computational resources.

Example: `'SquareRoot',true`

Data Types: `logical`

`'Tolerance'` — Forecast uncertainty threshold

`0` (default) | nonnegative scalar

Forecast uncertainty threshold, specified as the comma-separated pair consisting of `'Tolerance'` and a nonnegative scalar.

If the forecast uncertainty for a particular observation is less than `Tolerance` during numerical estimation, then the software removes the uncertainty corresponding to the observation from the forecast covariance matrix before its inversion.

It is best practice to set `Tolerance` to a small number, for example, `1e-15`, to overcome numerical obstacles during estimation.

Example: `'Tolerance',1e-15`

Data Types: `double`

`'Univariate'` — Univariate treatment of multivariate series flag

`false` (default) | `true`

Univariate treatment of a multivariate series flag, specified as the comma-separated pair consisting of `'Univariate'` and `true` or `false`. Univariate treatment of a multivariate series is also known as *sequential filtering*.

The univariate treatment can accelerate and improve numerical stability of the Kalman filter. However, all observation innovations must be uncorrelated. That is, D_tD_t' must be diagonal, where D_t, $t = 1,...,T$, is one of the following:

- The matrix `D{t}` in a time-varying state-space model
- The matrix `D` in a time-invariant state-space model

Example: `'Univariate',true`

Data Types: `logical`

Output Arguments

`X` — Filtered states

numeric matrix | cell vector of numeric vectors

Filtered states, returned as a numeric matrix or a cell vector of numeric vectors.

If `Mdl` is time invariant, then the number of rows of `X` is the sample size, T, and the number of columns of `X` is the number of states, m. The last row of `X` contains the latest, filtered states.

If `Mdl` is time varying, then `X` is a cell vector with length equal to the sample size. Cell t of `X` contains a vector of filtered states with length equal to the number of states in period t. The last cell of `X` contains the latest, filtered states.

`logL` — Loglikelihood function value

scalar

Loglikelihood function value, returned as a scalar.

Missing observations do not contribute to the loglikelihood.

`Output` — Filtering results by period

structure array

Filtering results by period, returned as a structure array.

`Output` is a T-by-1 structure, where element t corresponds to the filtering result at time t.

- If `Univariate` is `false` (it is by default), then the following table outlines the fields of `Output`.

Field	Description	Estimate of
`LogLikelihood`	Scalar loglikelihood objective function value	N/A
`FilteredStates`	m_t-by-1 vector of filtered states	$E(x_t \mid y_1,\ldots,y_t)$
`FilteredStatesCov`	m_t-by-m_t variance-covariance matrix of filtered states	$Var(x_t \mid y_1,\ldots,y_t)$
`ForecastedStates`	m_t-by-1 vector of state forecasts	$E(x_t \mid y_1,\ldots,y_{t-1})$
`ForecastedStatesCov`	m_t-by-m_t variance-covariance matrix of state forecasts	$Var(x_t \mid y_1,\ldots,y_{t-1})$
`ForecastedObs`	h_t-by-1 forecasted observation vector	$E(y_t \mid y_1,\ldots,y_{t-1})$
`ForecastedObsCov`	h_t-by-h_t variance-covariance matrix of forecasted observations	$Var(y_t \mid y_1,\ldots,t_{t-1})$
`KalmanGain`	m_t-by-n_t adjusted Kalman gain matrix	N/A

Field	Description	Estimate of
`DataUsed`	h_t-by-1 logical vector indicating whether the software filters using a particular observation. For example, if observation i at time t is a `NaN`, then element i in `DataUsed` at time t is `0`.	N/A

- If `Univariate` is `true`, then the fields of `Output` are the same as in the previous table, except for the following amendments.

Field	Changes
`ForecastedObs`	Same dimensions as if `Univariate = 0`, but only the first elements are equal
`ForecastedObsCov`	n-by-1 vector of forecasted observation variances. The first element of this vector is equivalent to `ForecastedObsCov(1,1)` when `Univariate` is `false`. The rest of the elements are not necessarily equivalent to their corresponding values in `ForecastObsCov` when `Univariate`.
`KalmanGain`	Same dimensions as if `Univariate` is `false`, though `KalmanGain` might have different entries.

Examples

Filter States of Time-Invariant State-Space Model

Suppose that a latent process is an AR(1). Subsequently, the state equation is

$$x_t = 0.5x_{t-1} + u_t,$$

where u_t is Gaussian with mean 0 and standard deviation 1.

Generate a random series of 100 observations from x_t, assuming that the series starts at 1.5.

```
T = 100;
ARMdl = arima('AR',0.5,'Constant',0,'Variance',1);
x0 = 1.5;
rng(1); % For reproducibility
x = simulate(ARMdl,T,'Y0',x0);
```

Suppose further that the latent process is subject to additive measurement error. Subsequently, the observation equation is

$$y_t = x_t + \varepsilon_t,$$

where ε_t is Gaussian with mean 0 and standard deviation 0.75. Together, the latent process and observation equations compose a state-space model.

Use the random latent state process (`x`) and the observation equation to generate observations.

```
y = x + 0.75*randn(T,1);
```

Specify the four coefficient matrices.

```
A = 0.5;
B = 1;
C = 1;
D = 0.75;
```

Specify the state-space model using the coefficient matrices.

```
Mdl = ssm(A,B,C,D)
```

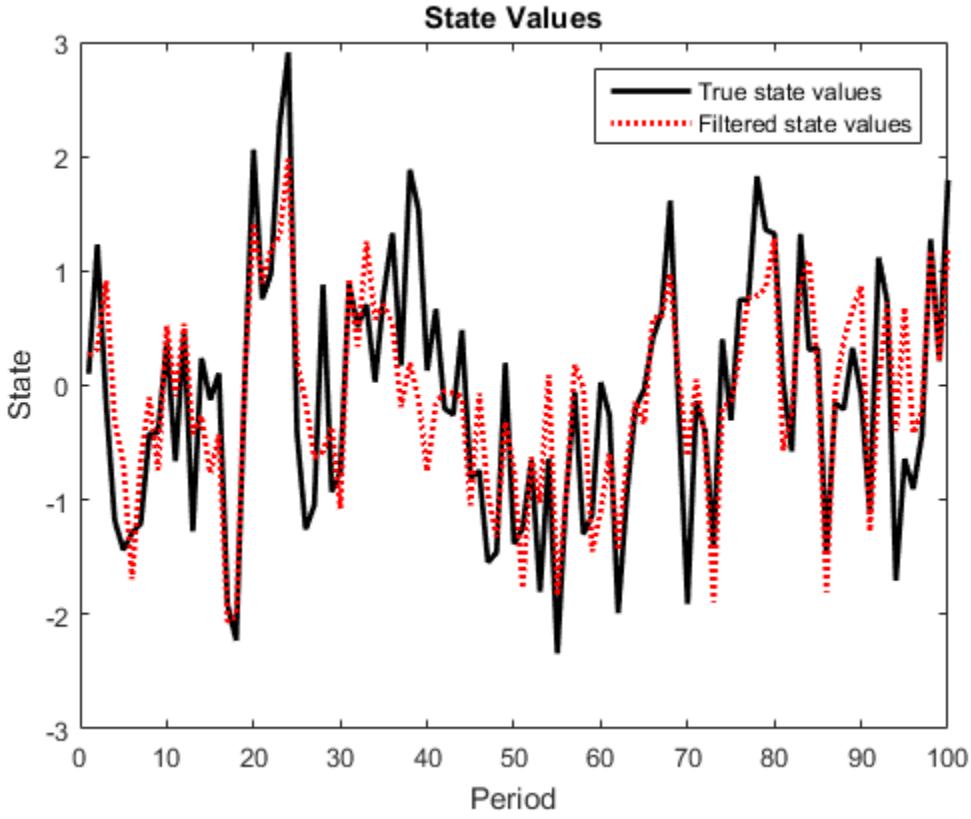
```
Mdl =


State-space model type: <a href="matlab: doc ssm">ssm</a>

State vector length: 1
Observation vector length: 1
State disturbance vector length: 1
Observation innovation vector length: 1
```

```
Sample size supported by model: Unlimited

State variables: x1, x2,...
State disturbances: u1, u2,...
Observation series: y1, y2,...
Observation innovations: e1, e2,...

State equation:
x1(t) = (0.50)x1(t-1) + u1(t)

Observation equation:
y1(t) = x1(t) + (0.75)e1(t)

Initial state distribution:

Initial state means
 x1
  0

Initial state covariance matrix
     x1
 x1  1.33

State types
     x1
 Stationary
```

`Mdl` is an `ssm` model. Verify that the model is correctly specified using the display in the Command Window. The software infers that the state process is stationary. Subsequently, the software sets the initial state mean and covariance to the mean and variance of the stationary distribution of an AR(1) model.

Filter states for periods 1 through 100. Plot the true state values and the filtered state estimates.

```
filteredX = filter(Mdl,y);

figure
plot(1:T,x,'-k',1:T,filteredX,':r','LineWidth',2)
title({'State Values'})
xlabel('Period')
ylabel('State')
legend({'True state values','Filtered state values'})
```



The true values and filter estimates are approximately the same.

Filter States of State-Space Model Containing Regression Component

Suppose that the linear relationship between the change in the unemployment rate and the nominal gross national product (nGNP) growth rate is of interest. Suppose further that the first difference of the unemployment rate is an ARMA(1,1) series. Symbolically, and in state-space form, the model is

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix} = \begin{bmatrix} \phi & \theta \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix} u_{1,t}$$
$$y_t - \beta Z_t = x_{1,t} + \sigma \varepsilon_t,$$

where:

- $x_{1,t}$ is the change in the unemployment rate at time t.
- $x_{2,t}$ is a dummy state for the MA(1) effect.
- $y_{1,t}$ is the observed change in the unemployment rate being deflated by the growth rate of nGNP (Z_t).
- $u_{1,t}$ is the Gaussian series of state disturbances having mean 0 and standard deviation 1.
- ε_t is the Gaussian series of observation innovations having mean 0 and standard deviation σ.

Load the Nelson-Plosser data set, which contains the unemployment rate and nGNP series, among other things.

```
load Data_NelsonPlosser
```

Preprocess the data by taking the natural logarithm of the nGNP series, and the first difference of each series. Also, remove the starting `NaN` values from each series.

```
isNaN = any(ismissing(DataTable),2);       % Flag periods containing NaNs
gnpn = DataTable.GNPN(~isNaN);
u = DataTable.UR(~isNaN);
T = size(gnpn,1);                          % Sample size
Z = [ones(T-1,1) diff(log(gnpn))];
y = diff(u);
```

Though this example removes missing values, the software can accommodate series containing missing values in the Kalman filter framework.

Specify the coefficient matrices.

```
A = [NaN NaN; 0 0];
B = [1; 1];
C = [1 0];
D = NaN;
```

Specify the state-space model using `ssm`.

```
Mdl = ssm(A,B,C,D);
```

Estimate the model parameters, and use a random set of initial parameter values for optimization. Specify the regression component and its initial value for optimization

using the 'Predictors' and 'Beta0' name-value pair arguments, respectively. Restrict the estimate of σ to all positive, real numbers.

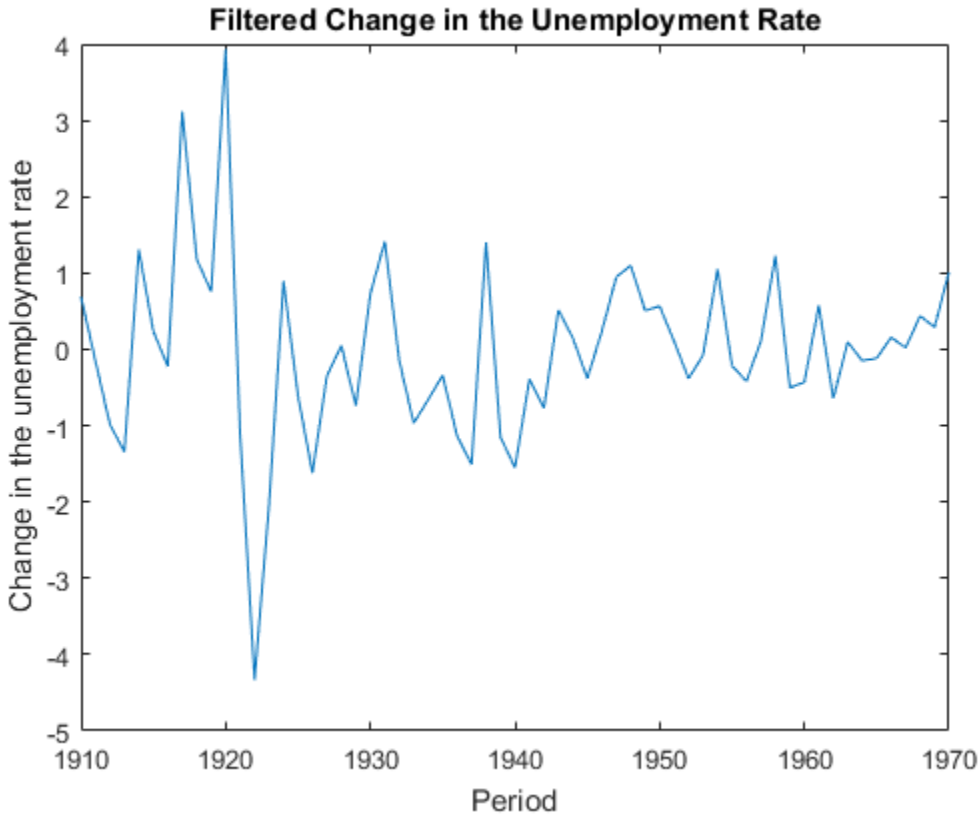
```
params0 = [0.3 0.2 0.2];
[EstMdl,estParams] = estimate(Mdl,y,params0,'Predictors',Z,...
    'Beta0',[0.1 0.2],'lb',[-Inf,-Inf,0,-Inf,-Inf]);
```

```
Method: Maximum likelihood (fmincon)
Sample size: 61
Logarithmic  likelihood:     -99.7245
Akaike   info criterion:      209.449
Bayesian info criterion:      220.003
            |      Coeff       Std Err    t Stat     Prob
-------------------------------------------------------------
 c(1)       |  -0.34098       0.29608    -1.15164  0.24948
 c(2)       |   1.05003       0.41377     2.53771  0.01116
 c(3)       |   0.48592       0.36790     1.32080  0.18657
 y <- z(1)  |   1.36121       0.22338     6.09358   0
 y <- z(2)  | -24.46711       1.60018   -15.29024   0
            |
            |    Final State   Std Dev     t Stat    Prob
 x(1)       |   1.01264       0.44690     2.26592  0.02346
 x(2)       |   0.77718       0.58917     1.31912  0.18713
```

EstMdl is an ssm model, and you can access its properties using dot notation.

Filter the estimated state-space model. EstMdl does not store the data or the regression coefficients, so you must pass in them in using the name-value pair arguments 'Predictors' and 'Beta', respectively. Plot the estimated, filtered states. Recall that the first state is the change in the unemployment rate, and the second state helps build the first.

```
filteredX = filter(EstMdl,y,'Predictors',Z,'Beta',estParams(end-1:end));

figure
plot(dates(end-(T-1)+1:end),filteredX(:,1));
xlabel('Period')
ylabel('Change in the unemployment rate')
title('Filtered Change in the Unemployment Rate')
```



- "Filter Time-Varying State-Space Model" on page 8-62

Algorithms

- The Kalman filter accommodates missing data by not updating filtered state estimates corresponding to missing observations. In other words, suppose there is a missing observation at period t. Then, the state forecast for period t based on the previous $t-1$ observations and filtered state for period t are equivalent.
- For explicitly defined state-space models, `ssm.filter` applies all predictors to each response series. However, each response series has its own set of regression coefficients.

Tips

- `Mdl` does not store the response data, predictor data, and the regression coefficients. Supply the data wherever necessary using the appropriate input or name-value pair arguments.
- To accelerate estimation for low-dimensional, time-invariant models, set `'Univariate',true`. Using this specification, the software sequentially updates rather then updating all at once during the filtering process.

References

[1] Durbin J., and S. J. Koopman. *Time Series Analysis by State Space Methods*. 2nd ed. Oxford: Oxford University Press, 2012.

See Also

estimate | forecast | refine | smooth | ssm

More About

- "What Are State-Space Models?" on page 8-3
- "What Is the Kalman Filter?" on page 8-8

forecast

Forecast conditional variances from conditional variance models

Syntax

```
V = forecast(Mdl,numPeriods)
V = forecast(Mdl,numPeriods,Name,Value)
```

Description

`V = forecast(Mdl,numPeriods)` forecasts conditional variances of the fully specified, univariate conditional variance model `Mdl` over the forecast horizon `numPeriods`. `Mdl` can be a `garch`, `egarch`, or `gjr` model.

`V = forecast(Mdl,numPeriods,Name,Value)` generates forecasts with additional options specified by one or more `Name,Value` pair arguments. For example, you can specify presample responses or conditional variances.

Examples

Forecast GARCH Model Conditional Variances

Forecast the conditional variance of simulated data over a 30-period horizon.

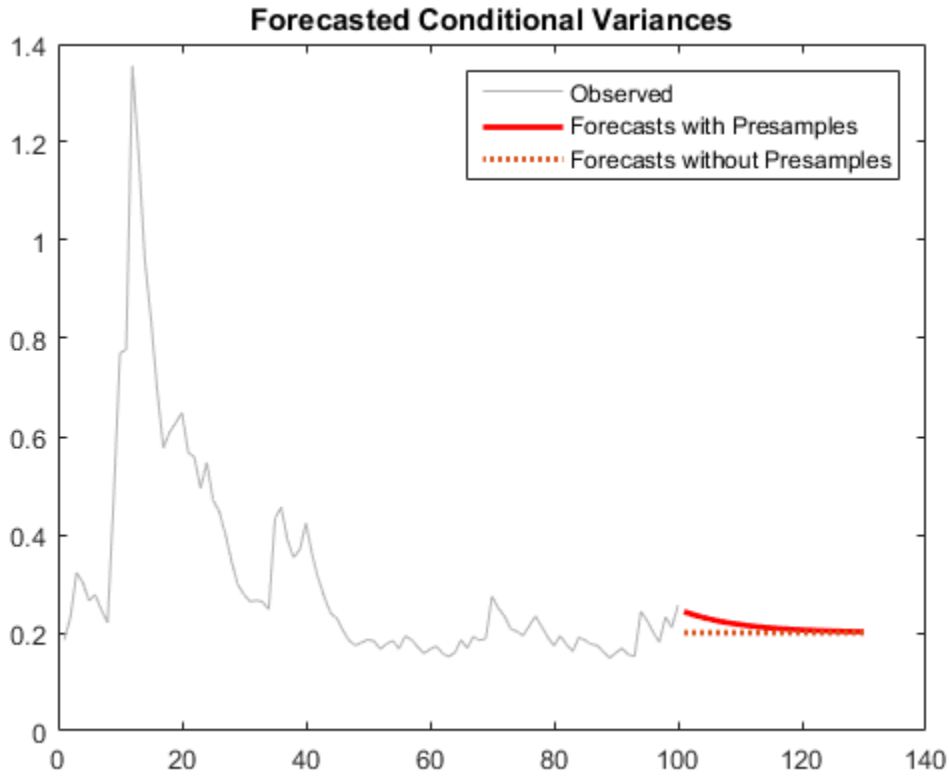
Simulate 100 observations from a GARCH(1,1) model with known parameters.

```
Mdl = garch('Constant',0.02,'GARCH',0.8,'ARCH',0.1);
rng default; % For reproducibility
[v,y] = simulate(Mdl,100);
```

Forecast the conditional variances over a 30-period horizon, with and without using the simulated data as presample innovations. Plot the forecasts.

```
vF1 = forecast(Mdl,30,'Y0',y);
vF2 = forecast(Mdl,30);

figure
plot(v,'Color',[.7,.7,.7])
hold on
```

```
plot(101:130,vF1,'r','LineWidth',2);
plot(101:130,vF2,':','LineWidth',2);
title('Forecasted Conditional Variances')
legend('Observed','Forecasts with Presamples',...
  'Forecasts without Presamples','Location','NorthEast')
hold off
```



Forecasts made without using presample innovations equal the unconditional innovation variance. Forecasts made using presample innovations converge asymptotically to the unconditional innovation variance.

Forecast EGARCH Model Conditional Variances

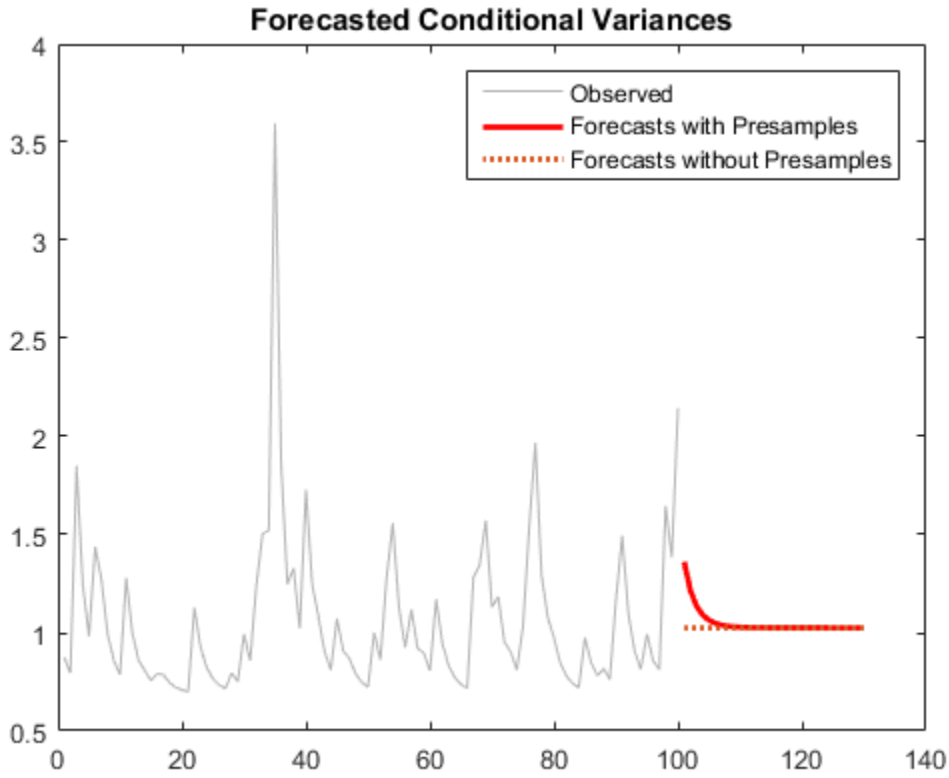
Forecast the conditional variance of simulated data over a 30-period horizon.

Simulate 100 observations from an EGARCH(1,1) model with known parameters.

```
Mdl = egarch('Constant',0.01,'GARCH',0.6,'ARCH',0.2,...
             'Leverage',-0.2);
rng default; % For reproducibility
[v,y] = simulate(Mdl,100);
```

Forecast the conditional variance over a 30-period horizon, with and without using the simulated data as presample innovations. Plot the forecasts.

```
Vf1 = forecast(Mdl,30,'Y0',y);
Vf2 = forecast(Mdl,30);

figure
plot(v,'Color',[.7,.7,.7])
hold on
plot(101:130,Vf1,'r','LineWidth',2);
plot(101:130,Vf2,':','LineWidth',2);
title('Forecasted Conditional Variances')
legend('Observed','Forecasts with Presamples',...
  'Forecasts without Presamples','Location','NorthEast')
hold off
```



Forecasts made without using presample innovations equal the unconditional innovation variance. Forecasts made using presample innovations converge asymptotically to the unconditional innovation variance.

Forecast GJR Model Conditional Variances

Forecast the conditional variance of simulated data over a 30-period horizon.

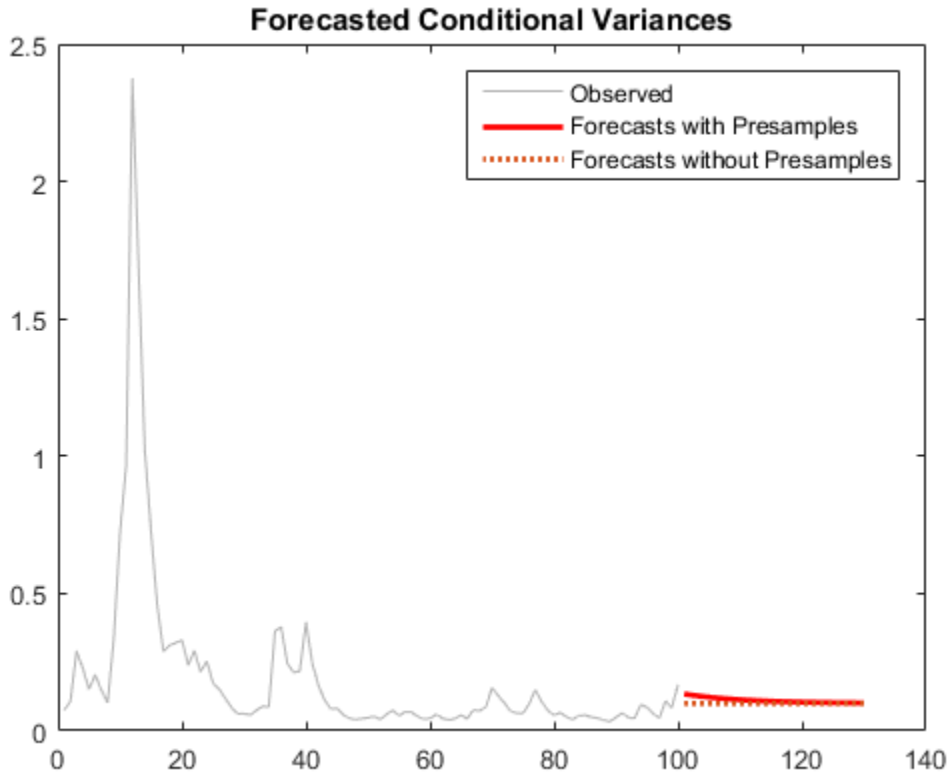
Simulate 100 observations from a GJR(1,1) model with known parameters.

```
Mdl = gjr('Constant',0.01,'GARCH',0.6,'ARCH',0.2,...
    'Leverage',0.2);
rng default; % For reproducibility
```

```
[v,y] = simulate(Mdl,100);
```

Forecast the conditional variances over a 30-period horizon, with and without using the simulated data as presample innovations. Plot the forecasts.

```
vF1 = forecast(Mdl,30,'Y0',y);
vF2 = forecast(Mdl,30);

figure
plot(v,'Color',[.7,.7,.7])
hold on
plot(101:130,vF1,'r','LineWidth',2);
plot(101:130,vF2,':','LineWidth',2);
title('Forecasted Conditional Variances')
legend('Observed','Forecasts with Presamples',...
  'Forecasts without Presamples','Location','NorthEast')
hold off
```



Forecasts made without using presample innovations equal the unconditional innovation variance. Forecasts made using presample innovations converge asymptotically to the unconditional innovation variance.

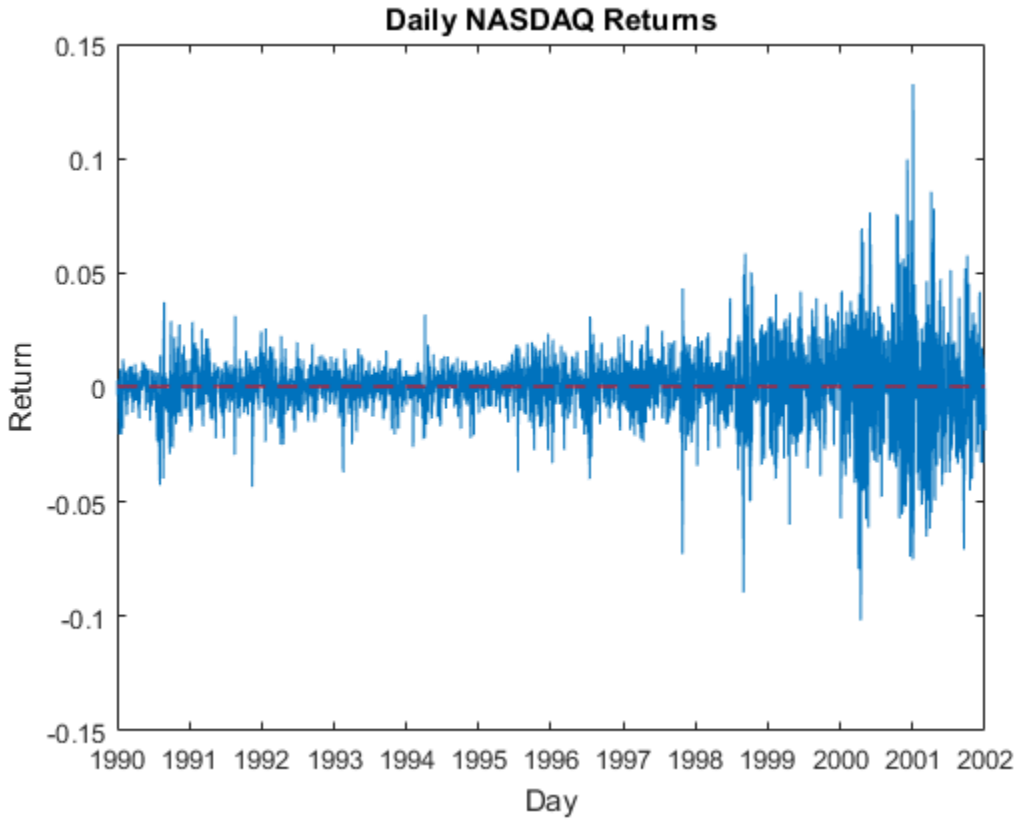
Compare Conditional Variance Forecasts of NYSE Returns

Forecast the conditional variance of the NASDAQ Composite Index returns over a 500-day horizon using GARCH(1,1), EGARCH(1,1) and GJR(1,1) models.

Load the NASDAQ data included with the toolbox. Convert the index to returns. Plot the returns.

```
load Data_EquityIdx
```

```
nasdaq = DataTable.NASDAQ;
r = price2ret(nasdaq);
T = length(r);
meanR = mean(r)

figure;
plot(dates(2:end),r,dates(2:end),meanR*ones(T,1),'--r');
datetick;
title('Daily NASDAQ Returns');
xlabel('Day');
ylabel('Return');


meanR =

   4.7771e-04
```



The variance of the series seems to change. This change is an indication of volatility clustering. The conditional mean model offset is very close to zero.

Fit GARCH(1,1), EGARCH(1,1), and GJR(1,1) models to the data. By default, the software sets the conditional mean model offset to zero.

```
MdlGARCH = garch(1,1);
MdlEGARCH = egarch(1,1);
MdlGJR = gjr(1,1);

EstMdlGARCH = estimate(MdlGARCH,r);
EstMdlEGARCH = estimate(MdlEGARCH,r);
EstMdlGJR = estimate(MdlGJR,r);
```

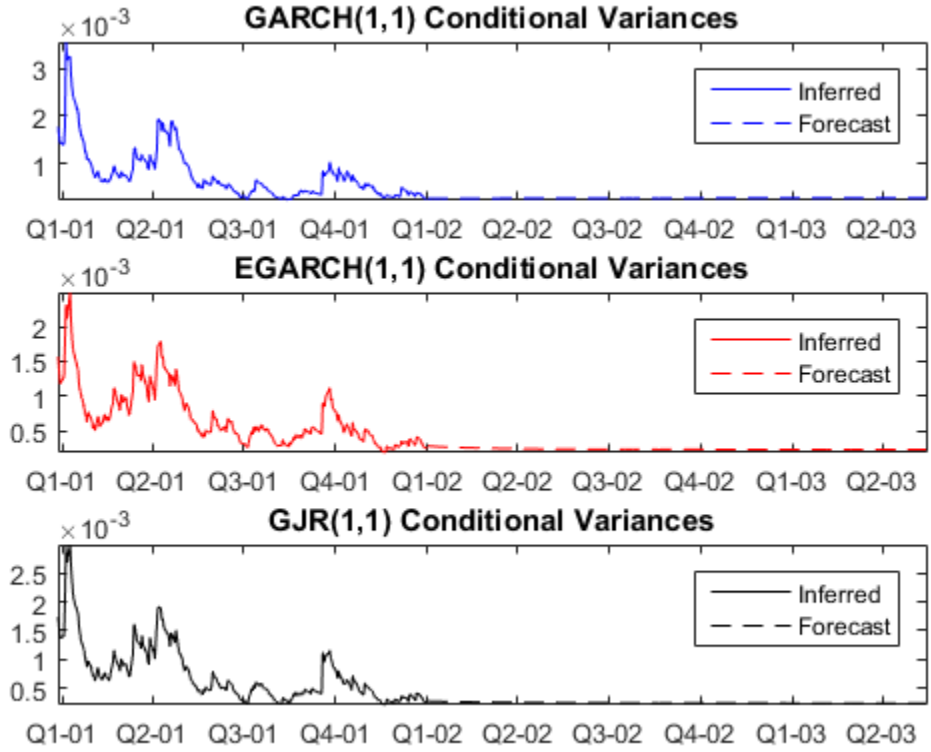
```
  GARCH(1,1) Conditional Variance Model:
  --------------------------------------
  Conditional Probability Distribution: Gaussian

                              Standard          t
   Parameter       Value        Error       Statistic
  -----------   -----------   ------------   -----------
   Constant    2.01008e-06   5.43141e-07        3.70085
   GARCH{1}       0.883294    0.00845285        104.497
    ARCH{1}       0.109193    0.00766209        14.2511

  EGARCH(1,1) Conditional Variance Model:
  ---------------------------------------
  Conditional Probability Distribution: Gaussian

                              Standard          t
   Parameter       Value        Error       Statistic
  -----------   -----------   ------------   -----------
   Constant      -0.134944     0.0220955       -6.10729
   GARCH{1}       0.983893    0.00242249        406.149
    ARCH{1}        0.19964     0.0139637        14.2971
Leverage{1}     -0.0602422    0.00564594         -10.67

  GJR(1,1) Conditional Variance Model:
  ------------------------------------
  Conditional Probability Distribution: Gaussian

                              Standard          t
   Parameter       Value        Error       Statistic
  -----------   -----------   ------------   -----------
   Constant    2.45669e-06   5.68278e-07        4.32305
   GARCH{1}       0.881439     0.0094779        92.9995
    ARCH{1}      0.0639391    0.00917697        6.96735
Leverage{1}      0.0889072    0.00990237        8.97838
```

Forecast the conditional variance for 500 days using the fitted models. Use the observed returns as presample innovations for the forecasts.

```
vFGARCH = forecast(EstMdlGARCH,500,'Y0',r);
vFEGARCH = forecast(EstMdlEGARCH,500,'Y0',r);
vFGJR= forecast(EstMdlGJR,500,'Y0',r);
```

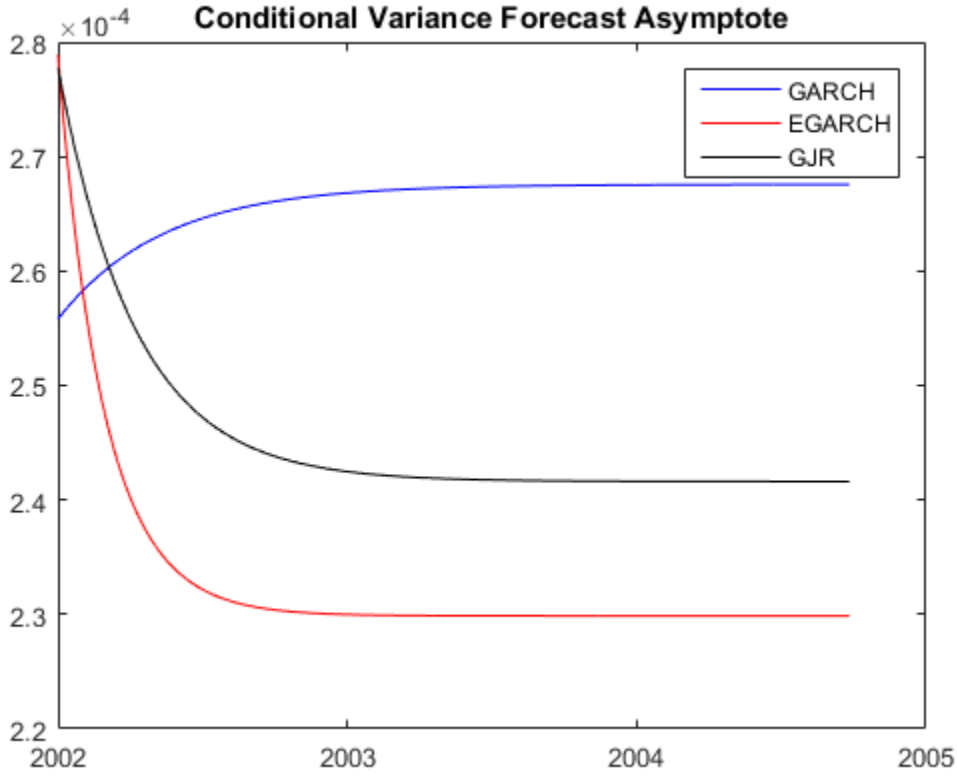
Plot the forecasts along with the conditional variances inferred from the data.

```
vGARCH = infer(EstMdlGARCH,r);
vEGARCH = infer(EstMdlEGARCH,r);
vGJR = infer(EstMdlGJR,r);
datesFH = dates(end):(dates(end)+1000); % 1000 period forecast horizon

figure;
subplot(3,1,1);
plot(dates(end-250:end),vGARCH(end-250:end),'b',...
    datesFH(2:end-500),vFGARCH,'b--');
legend('Inferred','Forecast','Location','NorthEast');
title('GARCH(1,1) Conditional Variances');
datetick;
axis tight;
subplot(3,1,2);
plot(dates(end-250:end),vEGARCH(end-250:end),'r',...
    datesFH(2:end-500),vFEGARCH,'r--');
legend('Inferred','Forecast','Location','NorthEast');
title('EGARCH(1,1) Conditional Variances');
datetick;
axis tight;

subplot(3,1,3);
plot(dates(end-250:end),vGJR(end-250:end),'k',...
    datesFH(2:end-500),vFGJR,'k--');
legend('Inferred','Forecast','Location','NorthEast');
title('GJR(1,1) Conditional Variances');
datetick;
axis tight;
```



Plot conditional variance forecasts for 1000 days.

```
vF1000GARCH = forecast(EstMdlGARCH,1000,'Y0',r);
vF1000EGARCH = forecast(EstMdlEGARCH,1000,'Y0',r);
vF1000GJR = forecast(EstMdlGJR,1000,'Y0',r);

figure;
plot(datesFH(2:end),vF1000GARCH,'b',...
    datesFH(2:end),vF1000EGARCH,'r',...
    datesFH(2:end),vF1000GJR,'k');
legend('GARCH','EGARCH','GJR','Location','NorthEast');
title('Conditional Variance Forecast Asymptote')
datetick;
```



The forecasts converge asymptotically to the unconditional variances of their repective processes.

- “Forecast a Conditional Variance Model” on page 6-126
- “Forecast GJR Models” on page 6-123

Input Arguments

`Mdl` — Conditional variance model

`garch` model object | `egarch` model object | `gjr` model object

Conditional variance model without any unknown parameters, specified as a `garch`, `egarch`, or `gjr` model object.

`Mdl` cannot contain any properties that have `NaN` value.

`numPeriods` — Forecast horizon
positive integer

Forecast horizon, specified as a positive integer.

The periods in the forecast horizon must be consistent with the periodicity of `Mdl` and the presample data.

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Y0',[1 1;0.5 0.5],'V0',[1 0.5;1 0.5]` specifies two equivalent presample paths of response data and two, different presample paths of conditional variances.

`'Y0'` — Presample responses
numeric column vector | numeric matrix

Presample responses whose conditional variance is forecasted, specified as the comma-separated pair consisting of `'Y0'` and a numeric column vector or matrix.

`Y0` usually represents a time series of presample innovations with mean 0 and variance encompassed in the input conditional variance model `Mdl`. `Y0` can also represent a time series of innovations with mean 0 plus an offset. If `Mdl` has a nonzero offset, then the software stores its value in the `Offset` property (`Mdl.Offset`).

If you specify `Y0`, then the software derives any necessary presample innovation observations (`E0`) from `Y0` by subtracting any offset. The software uses these presample innovations as initial values for the conditional variance model forecast.

`Y0` must have at least `Mdl.Q` elements or rows to initialize the variance equation. If the number of rows exceeds `Mdl.Q`, then the software uses the latest `Mdl.Q` observations to derive presample innovations (denoted `E0` in other methods).

- If `Y0` is a column vector, it represents a single path of the underlying innovation series. If `V0` is a matrix, then `forecast` applies `Y0` to each path.
- If `Y0` is a matrix, then each column represents a presample path of the underlying innovation series. If `V0` is also a matrix, then `Y0` must have the same number of columns as `V0`.

The last element or row contains the latest observation.

For GARCH(P,Q) and GJR(P,Q) models, `forecast` sets any necessary presample innovations to the unconditional standard deviation of the conditional variance model by default.

For EGARCH(P,Q) models, `filter` sets any necessary presample innovations to zero by default.

Data Types: `double`

`'V0'` — Presample conditional variances

numeric column vector with positive entries | numeric matrix with positive entries

Presample conditional variances, specified as the comma-separated pair consisting of `'V0'` and a numeric column vector or matrix with positive entries. `V0` provides initial values for the conditional variance model.

- If `V0` is a column vector, it represents a single presample path of the conditional variance series. If `Y0` is a matrix, then `forecast` applies `V0` to each path.
- If `V0` is a matrix, then each column represents a presample path of the underlying conditional variance series. If `Y0` is also a matrix, then `V0` must have the same number of columns as `Y0`.
- For GARCH(P,Q) and GJR(P,Q) models, `V0` must have at least `Mdl.P` rows to initialize the variance equation.
- For EGARCH(P,Q) models, `V0` must have at least max(P,Q) rows to initialize the variance equation.

If the number of elements or rows exceeds the necessary number, then `forecast` uses the latest observations only.

The last row contains the latest observation.

If `Y0` has at least max(P,Q) + P elements or rows, then `forecast` infers necessary presample observations from the corresponding presample response data in `Y0`. If you do not specify `Y0` or it has insufficient length, the defaults are:

- For GARCH(P,Q) and GJR(P,Q) models, `forecast` sets any necessary presample conditional variances to the unconditional variance of the conditional variance process.
- For EGARCH(P,Q) models, `forecast` sets any necessary presample conditional variances to the exponentiated, unconditional mean of the logarithm of the EGARCH(P,Q) variance process.

Data Types: `double`

Notes

- `NaN`s indicate missing values. `forecast` removes missing values. The software merges the presample data (`Y0` and `V0`), and then uses list-wise deletion to remove rows containing at least one `NaN`. Removing missing values in the data reduces the sample size. Removing missing values can also create irregular time series.
- `forecast` assumes that you synchronize presample data such that the last observation of each presample series occurs simultaneously.

Output Arguments

`V` — Minimum mean square error forecasts of conditional variances of future model innovations

numeric column vector | numeric matrix

Minimum mean square error forecasts of conditional variances of future model innovations, returned as a numeric column vector or matrix. `V` has `numPeriods` rows and the same number of columns as `Y0` and `V0`. If you do not specify `Y0` and `V0`, then `V` is a column vector.

The first row (or element) of `V` contains the conditional variance forecasts in period 1, the second row contains the conditional variance forecasts in period 2, and so on, until the last row. The last row contains the conditional variance forecasts at the forecast horizon specified by the input argument `numPeriods`.

More About

- Using garch Objects
- Using egarch Objects
- Using gjr Objects
- "MMSE Forecasting of Conditional Variance Models" on page 6-117

References

[1] Bollerslev, T. "Generalized Autoregressive Conditional Heteroskedasticity." *Journal of Econometrics*. Vol. 31, 1986, pp. 307–327.

[2] Bollerslev, T. "A Conditionally Heteroskedastic Time Series Model for Speculative Prices and Rates of Return." *The Review of Economics and Statistics*. Vol. 69, 1987, pp. 542–547.

[3] Box, G. E. P., G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis: Forecasting and Control*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

[4] Enders, W. *Applied Econometric Time Series*. Hoboken, NJ: John Wiley & Sons, 1995.

[5] Engle, R. F. "Autoregressive Conditional Heteroskedasticity with Estimates of the Variance of United Kingdom Inflation." *Econometrica*. Vol. 50, 1982, pp. 987–1007.

[6] Glosten, L. R., R. Jagannathan, and D. E. Runkle. "On the Relation between the Expected Value and the Volatility of the Nominal Excess Return on Stocks." *The Journal of Finance*. Vol. 48, No. 5, 1993, pp. 1779–1801.

[7] Hamilton, J. D. *Time Series Analysis*. Princeton, NJ: Princeton University Press, 1994.

[8] Nelson, D. B. "Conditional Heteroskedasticity in Asset Returns: A New Approach." *Econometrica*. Vol. 59, 1991, pp. 347–370.

See Also

`egarch` | `estimate` | `filter` | `garch` | `gjr` | `infer` | `print` | `simulate`

Introduced in R2012a

forecast

Class: arima

Forecast ARIMA or ARIMAX process

Syntax

```
[Y,YMSE] = forecast(Mdl,numPeriods)
[Y,YMSE,V] = forecast(Mdl,numPeriods)
[Y,YMSE,V] = forecast(Mdl,numPeriods,Name,Value)
```

Description

`[Y,YMSE] = forecast(Mdl,numPeriods)` forecasts responses for a univariate ARIMA model, and generates corresponding mean square errors, `YMSE`.

`[Y,YMSE,V] = forecast(Mdl,numPeriods)` additionally forecasts conditional variances for an ARIMA model with a conditional variance model.

`[Y,YMSE,V] = forecast(Mdl,numPeriods,Name,Value)` generates the forecasts with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

`Mdl` — ARIMA or ARIMAX model

`arima` model

ARIMA or ARIMAX model, specified as an arima model returned by `arima` or estimate.

The properties of `Mdl` cannot contain `NaN`s.

`numPeriods` — Forecast horizon

positive integer

Forecast horizon, specified as a positive integer.

The periods in the forecast horizon must be consistent with the periodicity of `Mdl` and the presample data.

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'E0'` — Presample innovations

numeric column vector | numeric matrix

Presample innovations that have mean 0 and provide initial values for the model, specified as the comma-separated pair consisting of `'E0'` and a numeric column vector or numeric matrix. `E0` must contain at least `Mdl.Q` rows. If you specify a conditional variance model, then `E0` might require more than `Mdl.Q` rows. If `E0` contains extra rows, then `forecast` only uses the latest presample innovations. The last row contains the latest presample innovation. If `E0` is a column vector, then the software applies it to each forecasted path.

By default, if `Y0` contains enough rows (at least `Mdl.P` + `Mdl.Q`), then `forecast` uses `infer` and the presample data to infer `E0`. For models with a regression component, if `forecast` infers `E0`, but `X0` does not contain enough rows (at least the number of rows of `Y0` – `Mdl.P`), then `forecast` displays an error. If the number of rows of `Y0` is insufficient, then `E0` is 0.

Data Types: `double`

`'V0'` — Presample conditional variances

numeric column vector with positive entries | numeric matrix with positive entries

Presample conditional variances providing initial values for any conditional variance model, specified as the comma-separated pair consisting of `'V0'` and a numeric column vector or matrix with positive entries. If the variance of the model is constant, then `V0` is unnecessary. `V0` is a column vector or a matrix with `numPaths` columns with enough rows to initialize the variance model. If `V0` contains extra rows, then `forecast` only uses the latest conditional variances. The last row contains the latest conditional variance. If `V0` is a column vector, then `forecast` applies it to each forecasted path.

By default, if E0 has sufficient length for the conditional variance model, then forecast infers the necessary presample conditional variances from the corresponding innovations E0. If E0 does not have sufficient length, then forecast sets V0 to the unconditional variance of the variance process.

Data Types: double

'X0' — Presample predictor data

numeric matrix

Presample predictor data that indicates the presence of a regression component in the conditional mean model, specified as the comma-separated pair of 'X0' and a numeric matrix. The columns of X0 are separate time series. X0 and XF must have the same number of columns. X0 must contain at least the number of rows of Y0 – Mdl.P. If X0 contains extra rows, then forecast only uses the latest observations. The last row indicates the latest observation of each series.

By default, forecast does not include a regression component in the conditional mean model regardless of the value of the regression coefficient Mdl.Beta.

Data Types: double

'XF' — Predictor forecasts

numeric matrix

Predictor forecasts, specified as the comma-separated pair of 'XF' and a numeric matrix. The columns of XF are separate time series. XF and X0 must have the same number of columns. XF must have at least numPeriods rows. Row i of XF contains the i period-ahead forecasts of X0. If XF exceeds numPeriods rows, then forecast only uses the first numPeriods forecasts. forecast treats XF as a fixed (nonstochastic) matrix.

By default, forecast does not include a regression component in the conditional mean model regardless of the value of the regression coefficient Mdl.Beta.

'Y0' — Presample responses

numeric column vector | numeric matrix

Presample responses that provide initial values for the model, specified as the comma-separated pair consisting of 'Y0' and a numeric column vector or numeric matrix. Y0 must contain at least Mdl.P rows. If the number of rows exceeds Mdl.P, then forecast only uses the latest Mdl.P observations. The last row contains the latest observation. If Y0 is a column vector, then it is applied to each forecasted path.

By default, if the process is stationary and `Mdl` does not contain an regression component, then `forecast` sets the necessary presample observations to the unconditional mean of the process. Otherwise, `Y0` is 0.

Data Types: `double`

Notes

- If any of `E0`, `V0`, or `Y0` contain `numPaths` > 1 columns, then each must have either `numPaths` columns or one column, otherwise an error occurs. For example, if `Y0` has five columns, then `E0` and `V0` can either have five columns or one column. If `E0` has one column, then it is applied to each path in `Y0`.
- `NaN`s indicate missing values and `forecast` removes them. The software merges the presample data sets, then uses list-wise deletion to remove any `NaN`s. Removing `NaN`s in the data reduces the sample size, and can also create irregular time series.
- `forecast` assumes that you synchronize presample data such that the latest observation of each presample series occurs simultaneously.
- Set `X0` to the same predictor matrix as `X` used in the estimation, simulation, or inference of `Mdl`. This assignment ensures correct computation of the innovations `E0`.

Output Arguments

`Y` — Minimum mean square error forecasts of response data

numeric matrix

Minimum mean square error (MMSE) forecasts of the conditional mean of the response data, returned as a numeric matrix. `Y` has `numPeriods` rows and `numPaths` columns.

`forecast` sets the number of columns of `Y` (`numPaths`) to the largest number of columns of the presample arrays `Y0`, `E0`, and `V0`. If you do not specify `Y0`, `E0`, or `V0`, then `Y` is a `numPeriods` column vector.

In all cases, row i contains the conditional mean forecasts for the ith period.

Data Types: `double`

`YMSE` — Mean square errors forecasts of conditional mean

numeric matrix

Mean square errors (MSE) forecasts of the conditional mean `Y`, returned as a numeric matrix. `YMSE` has `numPeriods` rows and `numPaths` columns.

`forecast` sets the number of columns of `YMSE` (`numPaths`) to the largest number of columns of the presample arrays `Y0`, `E0`, and `V0`. If you do not specify `Y0`, `E0`, or `V0`, then `Y` is a `numPeriods` column vector.

In all cases, row *i* contains the forecast error variances for the *i*th period.

The square roots of `YMSE` are the standard errors of the forecasts of `Y`.

The predictor data does not contribute variability to `YMSE` because `forecast` treats `XF` as a nonstochastic matrix.

Data Types: `double`

`V` — Minimum mean square error forecasts of conditional variances of future model innovations

numeric matrix

Minimum mean square error (MMSE) forecasts of the conditional variances of future model innovations, returned as a numeric matrix. `V` has `numPeriods` rows and `numPaths` columns.

`forecast` sets the number of columns of `V` (`numPaths`) to the largest number of columns of the presample arrays `Y0`, `E0`, and `V0`. If you do not specify `Y0`, `E0`, and `V0`, then `V` is a `numPeriods` column vector.

In all cases, row *i* contains the conditional variance forecasts for the *i*th period.

Data Types: `double`

Examples

Forecast the Conditional Mean Response

Forecast the conditional mean response of simulated data over a 30-period horizon.

Simulate 130 observations from a multiplicative seasonal MA model with known parameter values.

```
Mdl = arima('MA',{0.5,-0.3},'SMA',0.4,'SMALags',12,...
```

```
  'Constant',0.04,'Variance',0.2);
rng(200);
Y = simulate(Mdl,130);
```

Fit a seasonal MA model to the first 100 observations, and reserve the remaining 30 observations to evaluate forecast performance.

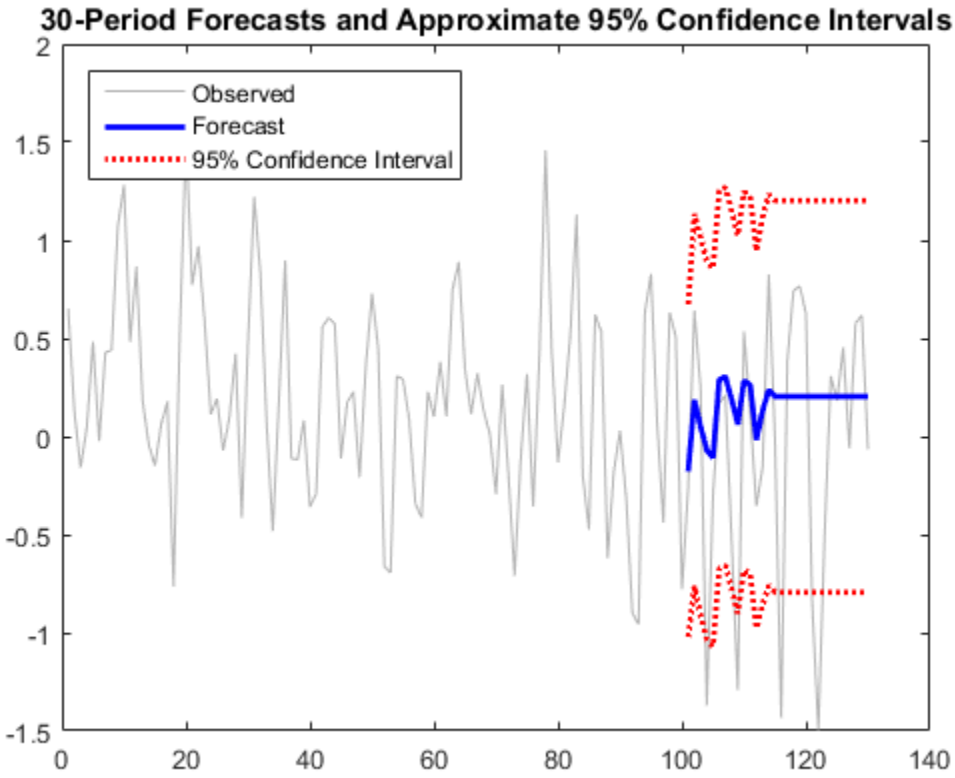
```
ToEstMdl = arima('MALags',1:2,'SMALags',12);
EstMdl = estimate(ToEstMdl,Y(1:100));
```

```
    ARIMA(0,0,2) Model with Seasonal MA(12):
    ----------------------------------------
    Conditional Probability Distribution: Gaussian

                               Standard          t
     Parameter       Value       Error       Statistic
    -----------   -----------   ------------   -----------
     Constant        0.20403     0.0690637        2.95424
        MA{1}       0.502116     0.0972984        5.16058
        MA{2}       -0.20174      0.104466       -1.93115
       SMA{12}       0.27028      0.109071        2.47803
     Variance        0.18681     0.0327319        5.70728
```

`EstMdl` is a new `arima` model with parameters estimated.

Use the fitted model to forecast a 30-period horizon, and visually compare the forecasts to the holdout data.

```
[YF YMSE] = forecast(EstMdl,30,'Y0',Y(1:100));

figure
h1 = plot(Y,'Color',[.7,.7,.7]);
hold on
h2 = plot(101:130,YF,'b','LineWidth',2);
h3 = plot(101:130,YF + 1.96*sqrt(YMSE),'r:',...
  'LineWidth',2);
plot(101:130,YF - 1.96*sqrt(YMSE),'r:','LineWidth',2);
legend([h1 h2 h3],'Observed','Forecast',...
  '95% Confidence Interval','Location','NorthWest');
title(['30-Period Forecasts and Approximate 95% '...
   'Confidence Intervals'])
hold off
```



Forecast the NASDAQ Composite Index

Forecast the daily NASDAQ Composite Index over a 500-day horizon.

Load the NASDAQ data included with the toolbox, and extract the first 1500 observations.

```
load Data_EquityIdx
nasdaq = DataTable.NASDAQ(1:1500);
```

Fit an ARIMA(1,1,1) model to the data.

```
nasdaqModel = arima(1,1,1);
nasdaqFit = estimate(nasdaqModel,nasdaq);
```

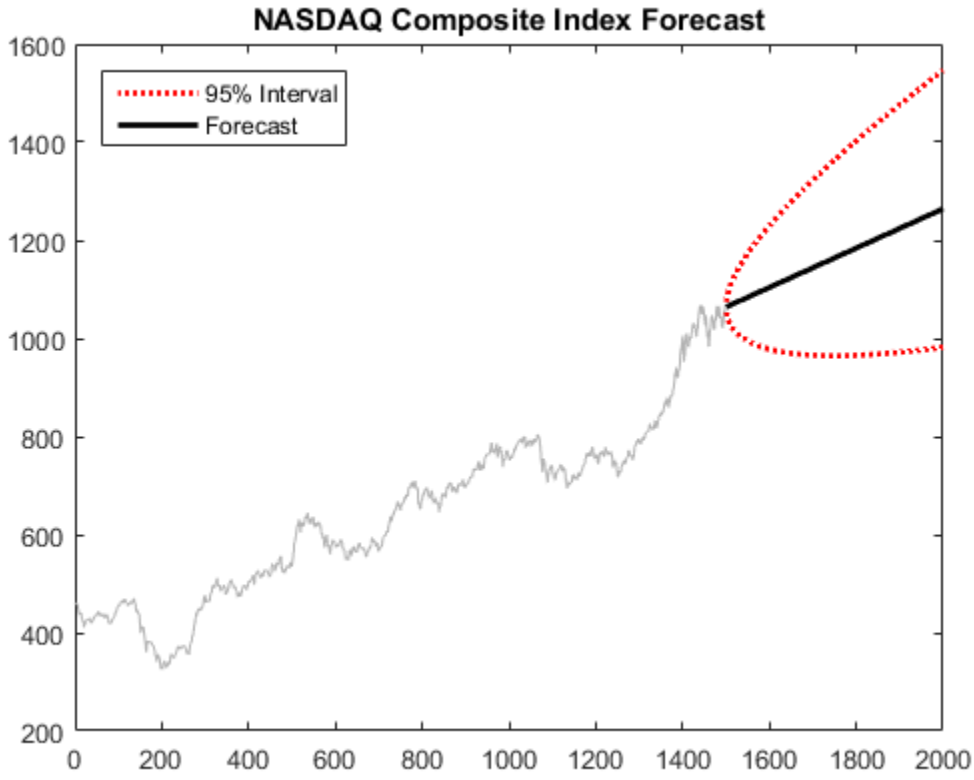
```
ARIMA(1,1,1) Model:
--------------------
Conditional Probability Distribution: Gaussian

                              Standard          t
 Parameter       Value          Error       Statistic
-----------   -----------   ------------   -----------
 Constant       0.430313      0.185554        2.31907
    AR{1}     -0.0743894      0.081985      -0.907353
    MA{1}       0.311256     0.0772657        4.02838
 Variance         27.826      0.636248        43.7346
```

Forecast the Composite Index for 500 days using the fitted model. Use the observed data as presample data.

```
[Y,YMSE] = forecast(nasdaqFit,500,'Y0',nasdaq);
```

Plot the forecasts and 95% forecast intervals.

```
lower = Y - 1.96*sqrt(YMSE);
upper = Y + 1.96*sqrt(YMSE);

figure
plot(nasdaq,'Color',[.7,.7,.7]);
hold on
h1 = plot(1501:2000,lower,'r:','LineWidth',2);
plot(1501:2000,upper,'r:','LineWidth',2)
h2 = plot(1501:2000,Y,'k','LineWidth',2);
legend([h1 h2],'95% Interval','Forecast',...
       'Location','NorthWest')
title('NASDAQ Composite Index Forecast')
hold off
```



The process is nonstationary, so the widths of the forecast intervals grow with time.

- "Forecast Multiplicative ARIMA Model" on page 5-192
- "Convergence of AR Forecasts" on page 5-186
- "Model Seasonal Lag Effects Using Indicator Variables" on page 5-117
- "Forecast Conditional Mean and Variance Model" on page 5-197
- "Forecast IGD Rate Using ARIMAX Model" on page 5-122

References

[1] Baillie, R., and T. Bollerslev. "Prediction in Dynamic Models with Time-Dependent Conditional Variances." *Journal of Econometrics*. Vol. 52, 1992, pp. 91–113.

[2] Bollerslev, T. "Generalized Autoregressive Conditional Heteroskedasticity." *Journal of Econometrics*. Vol. 31, 1996, pp. 307–327.

[3] Bollerslev, T. "A Conditionally Heteroskedastic Time Series Model for Speculative Prices and Rates of Return." *The Review Economics and Statistics*. Vol. 69, 1987, pp. 542–547.

[4] Box, G. E. P., G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis: Forecasting and Control* 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

[5] Enders, W. *Applied Econometric Time Series*. Hoboken, NJ: John Wiley & Sons, 1995.

[6] Engle, R. F. "Autoregressive Conditional Heteroskedasticity with Estimates of the Variance of United Kingdom Inflation." *Econometrica*. Vol. 50, 1982, pp. 987–1007.

[7] Hamilton, J. D. *Time Series Analysis*. Princeton, NJ: Princeton University Press, 1994.

See Also

arima | estimate | filter | impulse | infer | print | simulate

More About

- "MMSE Forecasting of Conditional Mean Models" on page 5-182
- "Monte Carlo Forecasting of Conditional Mean Models" on page 5-181

forecast

Class: regARIMA

Forecast responses of regression model with ARIMA errors

Syntax

```
[Y,YMSE] = forecast(Mdl,numPeriods)
[Y,YMSE,U] = forecast(Mdl,numPeriods)
[Y,YMSE,U] = forecast(Mdl,numPeriods,Name,Value)
```

Description

`[Y,YMSE] = forecast(Mdl,numPeriods)` forecasts responses (`Y`) for a regression model with ARIMA time series errors and generates corresponding mean square errors (`YMSE`).

`[Y,YMSE,U] = forecast(Mdl,numPeriods)` additionally forecasts unconditional disturbances for a regression model with ARIMA errors.

`[Y,YMSE,U] = forecast(Mdl,numPeriods,Name,Value)` forecasts with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

`Mdl` — Regression model with ARIMA errors

`regARIMA` model

Regression model with ARIMA errors, specified as a regARIMA model returned by `regARIMA` or estimate.

The properties of `Mdl` cannot contain `NaN`s.

`numPeriods` — Forecast horizon

positive integer

Forecast horizon, specified as a positive integer.

The periods in the forecast horizon must be consistent with the periodicity of `Mdl` and the presample data.

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'E0'` — Presample innovations

numeric column vector | numeric matrix

Presample innovations that have mean 0 and provide initial values for the ARIMA error model, specified as the comma-separated pair consisting of `'E0'` and a numeric column vector or numeric matrix.

- If `E0` is a column vector, then `forecast` applies it to each forecasted path.
- If `E0`, `Y0`, and `U0` are matrices with multiple paths, then they require the same number of columns.
- `E0` requires at least `Mdl.Q` rows. If `E0` contains extra rows, then `forecast` uses the latest presample innovations. The last row contains the latest presample innovation.

By default, if `U0` contains at least `Mdl.P` + `Mdl.Q` rows, then `forecast` infers `E0` from `U0`. If `U0` has an insufficient number of rows and `forecast` cannot infer sufficient observations of `U0` from the presample data (`Y0` and `X0`), then `E0` is 0.

Data Types: `double`

`'U0'` — Presample unconditional disturbances

numeric column vector | numeric matrix

Presample unconditional disturbances that provide initial values for the ARIMA error model, specified as the comma-separated pair consisting of `'U0'` and a numeric column vector or numeric matrix.

- If `U0` is a column vector, then `forecast` applies it to each forecasted path.

- If `U0`, `Y0`, and `E0` are matrices with multiple paths, then they require the same number of columns.
- `U0` requires at least `Mdl.P` rows. If `U0` contains extra rows, then `forecast` uses the latest presample unconditional disturbances. The last row contains the latest presample unconditional disturbance.

By default, if the presample data (`Y0` and `X0`) contains at least `Mdl.P` rows, then `forecast` infers `U0` from the presample data. If you do not specify presample data, then `U0` is 0.

Data Types: `double`

`'X0'` — Presample predictor data

matrix

Presample predictor data that provides initial values for the regression model, specified as the comma-separated pair consisting of `'X0'` and a matrix. The columns of `X0` are separate time series.

- If you do not specify `U0`, then `X0` requires at least `Mdl.P` rows to infer `U0`. If `X0` contains extra rows, then `forecast` uses the latest observations. The last row indicates the latest observation of each series.
- `X0` requires the same number of columns as the length of `Mdl.Beta`.
- If you specify `X0`, then you must also specify `XF`.
- `forecast` treats `X0` as a fixed (nonstochastic) matrix.

Data Types: `double`

`'XF'` — Predictor forecasts

numeric matrix

Predictor forecasts, specified as the comma-separated pair consisting of `'XF'` and a numeric matrix. The columns of `XF` are separate time series, each corresponding to forecasts of the series in `X0`. Row i of `XF` contains the i period-ahead forecasts of `X0`.

If you specify `X0`, then you must also specify `XF`. `XF` and `X0` require the same number of columns. `XF` requires at least `numPeriods` rows. If `XF` exceeds `numPeriods` rows, then `forecast` uses the first `numPeriods` forecasts.

`forecast` treats `XF` as a fixed (nonstochastic) matrix.

By default, `forecast` does not include a regression component in the model regardless of the presence of regression coefficients in `Mdl`.

Data Types: `double`

`'Y0'` — Presample responses

numeric column vector | numeric matrix

Presample responses that provide initial values for the regression model, specified as the comma-separated pair consisting of `'Y0'` and a numeric column vector or numeric matrix.

- If `Y0` is a column vector, then it is applied to each forecasted path.
- If `Y0`, `E0`, and `U0` are matrices with multiple paths, then they all require the same number of columns.
- If you do not specify `U0`, then `Y0` requires at least `Mdl.P` rows to infer `U0`. If `Y0` contains extra rows, then `forecast` uses the latest observations. The last row indicates the latest observation.

Data Types: `double`

Notes

- `NaN`s in `E0`, `U0`, `X0`, `XF`, and `Y0` indicate missing values and `forecast` removes them. The software merges the presample data sets (`E0`, `U0`, `X0`, and `Y0`), then uses list-wise deletion to remove any `NaN`s. `forecast` similarly removes `NaN`s from `XF`. Removing `NaN`s in the data reduces the sample size. Such removal can also create irregular time series.
- `forecast` assumes that you synchronize presample data such that the latest observation of each presample series occurs simultaneously.
- Set `X0` to the same predictor matrix as `X` used in the estimation, simulation, or inference of `Mdl`. This assignment ensures correct inference of the unconditional disturbances, `U0`.

Output Arguments

`Y` — Minimum mean square error forecasts of response data

numeric matrix

Minimum mean square error (MMSE) forecasts of the response data, returned as a numeric matrix. `Y` has `numPeriods` rows and `numPaths` columns.

- If you do not specify `Y0`, `E0`, and `U0`, then `Y` is a `numPeriods` column vector.
- If you specify `Y0`, `E0`, and `U0`, all having `numPaths` columns, then `Y` is a `numPeriods`-by-`numPaths` matrix.
- Row i of `Y` contains the forecasts for the ith period.

Data Types: `double`

`YMSE` — Mean square errors of forecasted responses

numeric matrix

Mean square errors (MSEs) of the forecasted responses, returned as a numeric matrix. `YMSE` has `numPeriods` rows and `numPaths` columns.

- If you do not specify `Y0`, `E0`, and `U0`, then `YMSE` is a `numPeriods` column vector.
- If you specify `Y0`, `E0`, and `U0`, all having `numPaths` columns, then `YMSE` is a `numPeriods`-by-`numPaths` matrix.
- Row i of `YMSE` contains the forecast error variances for the ith period.
- The predictor data does not contribute variability to `YMSE` because `forecast` treats `XF` as a nonstochastic matrix.
- The square roots of `YMSE` are the standard errors of the forecasts of `Y`.

Data Types: `double`

`U` — Minimum mean square error forecasts of future ARIMA error model unconditional disturbances

numeric matrix

Minimum mean square error (MMSE) forecasts of future ARIMA error model unconditional disturbances, returned as a numeric matrix. `U` has `numPeriods` rows and `numPaths` columns.

- If you do not specify `Y0`, `E0`, and `U0`, then `U` is a `numPeriods` column vector.
- If you specify `Y0`, `E0`, and `U0`, all having `numPaths` columns, then `U` is a `numPeriods`-by-`numPaths` matrix.
- Row i of `U` contains the forecasted unconditional disturbances for the ith period.

Data Types: `double`

Examples

Forecast Responses of a Regression Model with ARIMA Errors

Forecast responses from the following regression model with ARMA(2,1) errors over a 30-period horizon:

$$y_t = X_t \begin{bmatrix} 0.1 \\ -0.2 \end{bmatrix} + u_t$$
$$u_t = 0.5u_{t-1} - 0.8u_{t-2} + \varepsilon_t - 0.5\varepsilon_{t-1},$$

where ε_t is Gaussian with variance 0.1.

Specify the model. Simulate responses from the model and two predictor series.

```
Mdl = regARIMA('Intercept',0,'AR',{0.5 -0.8},...
    'MA',-0.5,'Beta',[0.1 -0.2],'Variance',0.1);
rng(1); % For reproducibility
X =  randn(130,2);
y = simulate(Mdl,130,'X',X);
```

Fit the model to the first 100 observations, and reserve the remaining 30 observations to evaluate forecast performance.

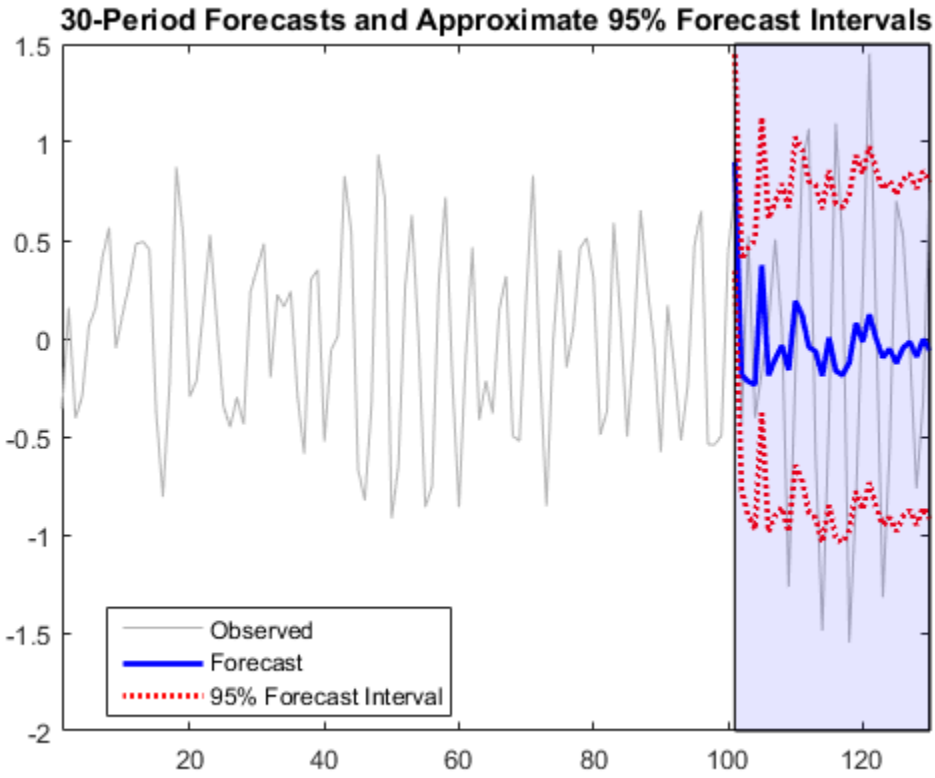
```
ToEstMdl = regARIMA('ARLags',1:2);
EstMdl = estimate(ToEstMdl,y(1:100),'X',X(1:100,:));
[yF,yMSE] = forecast(EstMdl,30,'Y0',y(1:100),...
    'X0',X(1:100,:),'XF',X(101:end,:));
```

```
    Regression with ARIMA(2,0,0) Error Model:
    ----------------------------------------
    Conditional Probability Distribution: Gaussian

                               Standard          t
     Parameter       Value       Error       Statistic
    -----------   -----------   ------------   -----------
    Intercept     0.00435796     0.0213144        0.20446
        AR{1}       0.368332      0.067103        5.48906
        AR{2}      -0.750627     0.0908646       -8.26094
        Beta1      0.0763979     0.0230081        3.32048
        Beta2      -0.139598     0.0232979       -5.99189
     Variance      0.0798765     0.0134196        5.95222
```

`EstMdl` is a new `regARIMA` model containing the estimates. The estimates are close to their true values.

Use `EstMdl` to forecast a 30-period horizon. Visually compare the forecasts to the holdout data using a plot.

```
[yF,yMSE] = forecast(EstMdl,30,'Y0',y(1:100),...
    'X0',X(1:100,:),'XF',X(101:end,:));

figure
plot(y,'Color',[.7,.7,.7]);
hold on
plot(101:130,yF,'b','LineWidth',2);
plot(101:130,yF+1.96*sqrt(yMSE),'r:',...
  'LineWidth',2);
plot(101:130,yF-1.96*sqrt(yMSE),'r:','LineWidth',2);
h = gca;
ph = patch([repmat(101,1,2) repmat(130,1,2)],...
        [h.YLim fliplr(h.YLim)],...
        [0 0 0 0],'b');
ph.FaceAlpha = 0.1;
legend('Observed','Forecast',...
  '95% Forecast Interval','Location','Best');
title(['30-Period Forecasts and Approximate 95% '...
   'Forecast Intervals'])
axis tight
hold off
```



Many observations in the holdout sample fall beyond the 95% forecast intervals. Two reasons for this are:

- The predictors are randomly generated in this example. `estimate` treats the predictors as fixed. Subsequently, the 95% forecast intervals based on the estimates from `estimate` do not account for the variability in the predictors.
- By shear chance, the estimation period seems less volatile than the forecast period. `estimate` uses the less volatile estimation period data to estimate the parameters.

Therefore, forecast intervals based on the estimates should not cover observations that have an underlying innovations process with larger variability.

Forecast the GDP Using a Regression Model with ARMA Errors

Forecast stationary, log GDP using a regression model with ARMA(1,1) errors, including CPI as a predictor.

Load the U.S. macroeconomic data set and preprocess the data.

```
load Data_USEconModel;
logGDP = log(DataTable.GDP);
dlogGDP = diff(logGDP);        % For stationarity
dCPI = diff(DataTable.CPIAUCSL); % For stationarity
numObs = length(dlogGDP);
gdp = dlogGDP(1:end-15);   % Estimation sample
cpi = dCPI(1:end-15);
T = length(gdp);        % Effective sample size
frstHzn =  T+1:numObs;  % Forecast horizon
hoCPI = dCPI(frstHzn);  % Holdout sample
dts = dates(2:end);     % Date nummbers
```

Fit a regression model with ARMA(1,1) errors.

```
ToEstMdl = regARIMA('ARLags',1,'MALags',1);
EstMdl = estimate(ToEstMdl,gdp,'X',cpi);
```

```
    Regression with ARIMA(1,0,1) Error Model:
    ----------------------------------------
    Conditional Probability Distribution: Gaussian

                               Standard          t
     Parameter       Value       Error       Statistic
    -----------   -----------   ------------   -----------
    Intercept      0.0147934    0.00162892       9.08176
        AR{1}       0.576012      0.100093       5.75477
        MA{1}      -0.152584      0.119784      -1.27382
        Beta1     0.00289724    0.00139893       2.07104
     Variance    9.57339e-05   6.55617e-06       14.6021
```

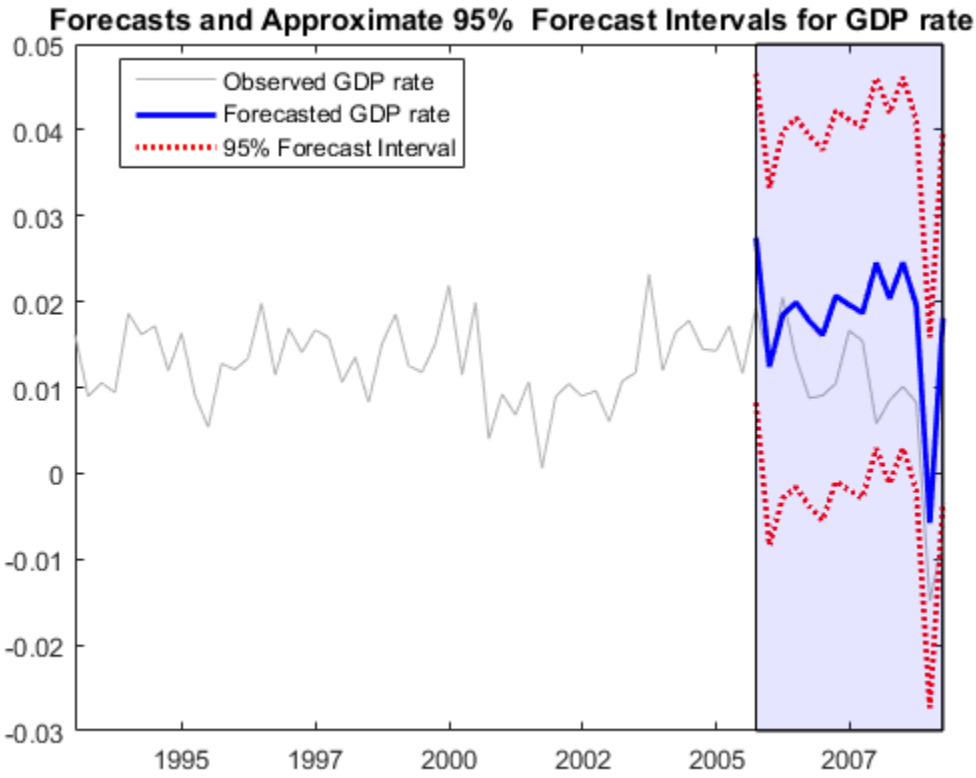
Forecast the GDP rate over a 15-quarter horizon. Use the estimation sample as a presample for the forecast.

```
[gdpF,gdpMSE] = forecast(EstMdl,15,'Y0',gdp,...
```

```
    'XO',cpi,'XF',hoCPI);
```

Plot the forecasts and 95% forecast intervals.

```
figure
h1 = plot(dts(end-65:end),dlogGDP(end-65:end),...
    'Color',[.7,.7,.7]);
datetick
hold on
h2 = plot(dts(frstHzn),gdpF,'b','LineWidth',2);
h3 = plot(dts(frstHzn),gdpF+1.96*sqrt(gdpMSE),'r:',...
  'LineWidth',2);
plot(dts(frstHzn),gdpF-1.96*sqrt(gdpMSE),'r:','LineWidth',2);
ha = gca;
legend([h1 h2 h3],'Observed GDP rate',...
    'Forecasted GDP rate ',...
    '95% Forecast Interval','Location','Best');
title(['{\bf Forecasts and Approximate 95% }'...
    '{\bf Forecast Intervals for GDP rate}']);

ph = patch([repmat(dts(frstHzn(1)),1,2) repmat(dts(frstHzn(end)),1,2)],...
    [ha.YLim fliplr(ha.YLim)],...
    [0 0 0 0],'b');
ph.FaceAlpha = 0.1;
axis tight
hold off
```



Forecast Using a Regression Model with ARIMA Errors and a Known Intercept

Forecast unit root nonstationary, log GDP using a regression model with ARIMA(1,1,1) errors, including CPI as a predictor and a known intercept.

Load the U.S. Macroeconomic data set and preprocess the data.

```
load Data_USEconModel;
numObs = length(DataTable.GDP);
logGDP = log(DataTable.GDP(1:end-15));
cpi = DataTable.CPIAUCSL(1:end-15);
T = length(logGDP);                % Effective sample size
frstHzn =  T+1:numObs;             % Forecast horizon
hoCPI = DataTable.CPIAUCSL(frstHzn); % Holdout sample
```

Specify the model for the estimation period.

```
ToEstMdl = regARIMA('ARLags',1,'MALags',1,'D',1);
```

The intercept is not identifiable in a model with integrated errors, so fix its value before estimation. One way to do this is to estimate the intercept using simple linear regression.

```
Reg4Int = [ones(T,1), cpi]\logGDP;
intercept = Reg4Int(1);
```

Consider performing a sensitivity analysis by using a grid of intercepts.

Set the intercept and fit the regression model with ARIMA(1,1,1) errors.

```
ToEstMdl.Intercept = intercept;
EstMdl = estimate(ToEstMdl,logGDP,'X',cpi,...
    'Display','off')
```

```
EstMdl =

    Regression with ARIMA(1,1,1) Error Model:
    ----------------------------------------
    Distribution: Name = 'Gaussian'
       Intercept: 5.80142
            Beta: [0.00396694]
               P: 2
               D: 1
               Q: 1
              AR: {0.922708} at Lags [1]
             SAR: {}
              MA: {-0.387844} at Lags [1]
             SMA: {}
        Variance: 0.000108942
```

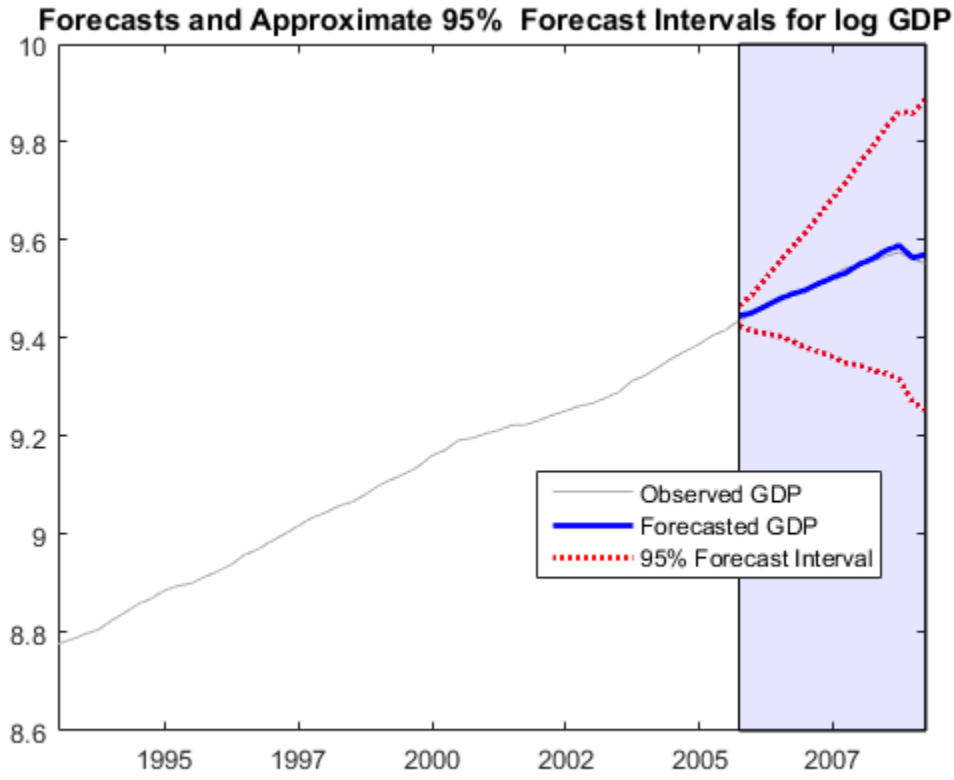
Forecast GDP over a 15-quarter horizon. Use the estimation sample as a presample for the forecast.

```
[gdpF,gdpMSE] = forecast(EstMdl,15,'Y0',logGDP,...
    'X0',cpi,'XF',hoCPI);
```

Plot the forecasts and 95% forecast intervals.

```
figure
h1 = plot(dates(end-65:end),log(DataTable.GDP(end-65:end)),...
    'Color',[.7,.7,.7]);
datetick
```

```
hold on
h2 = plot(dates(frstHzn),gdpF,'b','LineWidth',2);
h3 = plot(dates(frstHzn),gdpF+1.96*sqrt(gdpMSE),'r:',...
  'LineWidth',2);
plot(dates(frstHzn),gdpF-1.96*sqrt(gdpMSE),'r:',...
    'LineWidth',2);
ha = gca;
legend([h1 h2 h3],'Observed GDP','Forecasted GDP',...
  '95% Forecast Interval','Location','Best');
title(['{\bf Forecasts and Approximate 95% }'...
   '{\bf Forecast Intervals for log GDP}']);
ph = patch([repmat(dates(frstHzn(1)),1,2) repmat(dates(frstHzn(end)),1,2)],...
        [ha.YLim fliplr(ha.YLim)],...
        [0 0 0 0],'b');
ph.FaceAlpha = 0.1;
axis tight
hold off
```



The unconditional disturbances, u_t, are nonstationary, therefore the widths of the forecast intervals grow with time.

Algorithms

`forecast` computes the forecasted response MSEs, `YMSE`, by treating the predictor data matrices (`X0` and `XF`) as nonstochastic and statistically independent of the model innovations. Therefore, `YMSE` reflects the variance associated with the unconditional disturbances of the ARIMA error model alone.

References

[1] Box, G. E. P., G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis: Forecasting and Control*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

[2] Davidson, R., and J. G. MacKinnon. *Econometric Theory and Methods*. Oxford, UK: Oxford University Press, 2004.

[3] Enders, W. *Applied Econometric Time Series*. Hoboken, NJ: John Wiley & Sons, Inc., 1995.

[4] Hamilton, J. D. *Time Series Analysis*. Princeton, NJ: Princeton University Press, 1994.

[5] Pankratz, A. *Forecasting with Dynamic Regression Models*. John Wiley & Sons, Inc., 1991.

[6] Tsay, R. S. *Analysis of Financial Time Series*. 2nd ed. Hoboken, NJ: John Wiley & Sons, Inc., 2005.

See Also

regARIMA | estimate | infer | simulate

More About

- "MMSE Forecasting of Conditional Mean Models" on page 5-182
- "Monte Carlo Forecasting of Conditional Mean Models" on page 5-181

forecast

Class: dssm

Forecast states and observations of diffuse state-space models

Syntax

```
[Y,YMSE] = forecast(Mdl,numPeriods,Y0)
[Y,YMSE] = forecast(Mdl,numPeriods,Y0,Name,Value)
[Y,YMSE,X,XMSE] = forecast( ___ )
```

Description

`[Y,YMSE] = forecast(Mdl,numPeriods,Y0)` returns forecasted observations (`Y`) and their corresponding variances (`YMSE`) from forecasting the diffuse state-space model `Mdl` using a `numPeriods` forecast horizon and in-sample observations `Y0`.

`[Y,YMSE] = forecast(Mdl,numPeriods,Y0,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments. For example, for state-space models that include a linear regression component in the observation model, include in-sample predictor data, predictor data for the forecast horizon, and the regression coefficient.

`[Y,YMSE,X,XMSE] = forecast( ___ )` uses any of the input arguments in the previous syntaxes to additionally return state forecasts (`X`) and their corresponding variances (`XMSE`).

Tip

`Mdl` does not store the response data, predictor data, and the regression coefficients. Supply them whenever necessary using the appropriate input or name-value pair arguments.

Input Arguments

`Mdl` — Diffuse state-space model

`dssm` model object

Diffuse state-space model, specified as an `dssm` model object returned by dssm or estimate.

If `Mdl` is not fully specified (that is, `Mdl` contains unknown parameters), then specify values for the unknown parameters using the `'Params'` name-value pair argument. Otherwise, the software issues an error. `estimate` returns fully-specified state-space models.

`Mdl` does not store observed responses or predictor data. Supply the data wherever necessary using the appropriate input or name-value pair arguments.

`numPeriods` — Forecast horizon

positive integer

Forecast horizon, specified as a positive integer. That is, the software returns 1,..,`numPeriods` forecasts.

Data Types: `double`

`Y0` — In-sample, observed responses

cell vector of numeric vectors | numeric matrix

In-sample, observed responses, specified as a cell vector of numeric vectors or a matrix.

- If `Mdl` is time invariant, then `Y0` is a T-by-n numeric matrix, where each row corresponds to a period and each column corresponds to a particular observation in the model. Therefore, T is the sample size and m is the number of observations per period. The last row of `Y` contains the latest observations.
- If `Mdl` is time varying with respect to the observation equation, then `Y` is a T-by-1 cell vector. Each element of the cell vector corresponds to a period and contains an n_t-dimensional vector of observations for that period. The corresponding dimensions of the coefficient matrices in `Mdl.C{t}` and `Mdl.D{t}` must be consistent with the matrix in `Y{t}` for all periods. The last cell of `Y` contains the latest observations.

If `Mdl` is an estimated state-space model (that is, returned by estimate), then it is best practice to set `Y0` to the same data set that you used to fit `Mdl`.

`NaN` elements indicate missing observations. For details on how the Kalman filter accommodates missing observations, see "Algorithms" on page 9-510.

Data Types: `double | cell`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'A'` — Forecast-horizon, state-transition, coefficient matrices

cell vector of numeric matrices

Forecast-horizon, state-transition, coefficient matrices, specified as the comma-separated pair consisting of `'A'` and a cell vector of numeric matrices.

`A` must contain at least `numPeriods` cells. Each cell must contain a matrix specifying how the states transition in the forecast horizon. If the length of `A` is greater than `numPeriods`, then the software uses the first `numPeriods` cells. The last cell indicates the latest period in the forecast horizon.

The matrices in `A` cannot contain `NaN` values.

If `Mdl` is time invariant with respect to the states, then each cell of `A` must contain an m-by-m matrix, where m is the number of the in-sample states per period. By default, the software uses `Mdl.A` throughout the forecast horizon.

If `Mdl` is time varying with respect to the states, then the dimensions of the matrices in the cells of `A` may vary, but the dimensions of each matrix must be consistent with the matrices in `B` and `C` in the corresponding periods. By default, the software uses `Mdl.A{end}` throughout the forecast horizon.

Data Types: `cell`

`'B'` — Forecast-horizon, state-disturbance-loading, coefficient matrices

cell vector of matrices

Forecast-horizon, state-disturbance-loading, coefficient matrices, specified as the comma-separated pair consisting of `'B'` and a cell vector of matrices.

B must contain at least numPeriods cells. Each cell must contain a matrix specifying how the states transition in the forecast horizon. If the length of B is greater than numPeriods, then the software uses the first numPeriods cells. The last cell indicates the latest period in the forecast horizon.

The matrices in B cannot contain NaN values.

If Mdl is time invariant with respect to the states and state disturbances, then each cell of B must contain an m-by-k matrix, where m is the number of the in-sample states per period, and k is the number of in-sample, state disturbances per period. By default, the software uses Mdl.B throughout the forecast horizon.

If Mdl is time varying, then the dimensions of the matrices in the cells of B may vary, but the dimensions of each matrix must be consistent with the matrices in A in the corresponding periods. By default, the software uses Mdl.B{end} throughout the forecast horizon.

Data Types: cell

'C' — Forecast-horizon, measurement-sensitivity, coefficient matrices
cell vector of matrices

Forecast-horizon, measurement-sensitivity, coefficient matrices, specified as the comma-separated pair consisting of 'C' and a cell vector of matrices.

C must contain at least numPeriods cells. Each cell must contain a matrix specifying how the states transition in the forecast horizon. If the length of C is greater than numPeriods, then the software uses the first numPeriods cells. The last cell indicates the latest period in the forecast horizon.

The matrices in C cannot contain NaN values.

If Mdl is time invariant with respect to the states and the observations, then each cell of C must contain an n-by-m matrix, where n is the number of the in-sample observations per period, and m is the number of in-sample states per period. By default, the software uses Mdl.C throughout the forecast horizon.

If Mdl is time varying with respect to the states or the observations, then the dimensions of the matrices in the cells of C may vary, but the dimensions of each matrix must be consistent with the matrices in A and D in the corresponding periods. By default, the software uses Mdl.C{end} throughout the forecast horizon.

Data Types: cell

'D' — Forecast-horizon, observation-innovation, coefficient matrices

cell vector of matrices

Forecast-horizon, observation-innovation, coefficient matrices, specified as the comma-separated pair consisting of 'D' and a cell vector of matrices.

D must contain at least numPeriods cells. Each cell must contain a matrix specifying how the states transition in the forecast horizon. If the length of D is greater than numPeriods, then the software uses the first numPeriods cells. The last cell indicates the latest period in the forecast horizon.

The matrices in D cannot contain NaN values.

If Mdl is time invariant with respect to the observations and the observation innovations, then each cell of D must contain an n-by-h matrix, where n is the number of the in-sample observations per period, and h is the number of in-sample, observation innovations per period. By default, the software uses Mdl.D throughout the forecast horizon.

If Mdl is time varying with respect to the observations or the observation innovations, then the dimensions of the matrices in the cells of D may vary, but the dimensions of each matrix must be consistent with the matrices in C in the corresponding periods. By default, the software uses Mdl.D{end} throughout the forecast horizon.

Data Types: cell

'Beta' — Regression coefficients

[ ] (default) | numeric matrix

Regression coefficients corresponding to predictor variables, specified as the comma-separated pair consisting of 'Beta' and a d-by-n numeric matrix. d is the number of predictor variables (see PredictorsO and PredictorsF) and n is the number of observed response series (see YO).

If you specify Beta, then you must also specify PredictorsO and PredictorsF.

If Mdl is an estimated state-space model, then specify the estimated regression coefficients stored in Mdl.estParams.

By default, the software excludes a regression component from the state-space model.

'PredictorsO' — In-sample, predictor variables in state-space model observation equation

[ ] (default) | matrix

In-sample, predictor variables in the state-space model observation equation, specified as the comma-separated pair consisting of `'PredictorsO'` and a matrix. The columns of `PredictorsO` correspond to individual predictor variables. `PredictorsO` must have T rows, where row t corresponds to the observed predictors at period t (Z_t). The expanded observation equation is

$$y_t - Z_t\beta = Cx_t + Du_t.$$

that is, the software deflates the observations using the regression component. β is the time-invariant vector of regression coefficients that the software estimates with all other parameters.

If there are n observations per period, then the software regresses all predictor series onto each observation.

If you specify `PredictorsO`, then `Mdl` must be time invariant. Otherwise, the software returns an error.

If you specify `PredictorsO`, then you must also specify `Beta` and `PredictorsF`.

If `Mdl` is an estimated state-space model (that is, returned by estimate), then it is best practice to set `PredictorsO` to the same predictor data set that you used to fit `Mdl`.

By default, the software excludes a regression component from the state-space model.

Data Types: `double`

`'PredictorsF'` — Forecast-horizon, predictor variables in state-space model observation equation

[ ] (default) | numeric matrix

In-sample, predictor variables in the state-space model observation equation, specified as the comma-separated pair consisting of `'PredictorsO'` and a T-by-d numeric matrix. T is the number of in-sample periods and d is the number of predictor variables. Row t corresponds to the observed predictors at period t (Z_t). The expanded observation equation is

$$y_t - Z_t\beta = Cx_t + Du_t.$$

That is, the software deflates the observations using the regression component. β is the time-invariant vector of regression coefficients that the software estimates with all other parameters.

If there are n observations per period, then the software regresses all predictor series onto each observation.

If you specify `PredictorsO`, then `Mdl` must be time invariant. Otherwise, the software returns an error.

If you specify `PredictorsO`, then you must also specify `Beta` and `PredictorsF`.

If `Mdl` is an estimated state-space model (that is, returned by `estimate`), then it is best practice to set `PredictorsO` to the same predictor data set that you used to fit `Mdl`.

By default, the software excludes a regression component from the state-space model.

Data Types: `double`

Output Arguments

`Y` — Forecasted observations

matrix | cell vector of numeric vectors

Forecasted observations, returned as a matrix or a cell vector of numeric vectors.

If `Mdl` is a time-invariant, state-space model with respect to the observations, then `Y` is a `numPeriods`-by-n matrix.

If `Mdl` is a time-varying, state-space model with respect to the observations, then `Y` is a `numPeriods`-by-1 cell vector of numeric vectors. Cell t of `Y` contains an n_t-by-1 numeric vector of forecasted observations for period t.

`YMSE` — Error variances of forecasted observations

matrix | cell vector of numeric vectors

Error variances of forecasted observations, returned as a matrix or a cell vector of numeric vectors.

If `Mdl` is a time-invariant, state-space model with respect to the observations, then `YMSE` is a `numPeriods`-by-n matrix.

If `Mdl` is a time-varying, state-space model with respect to the observations, then `YMSE` is a `numPeriods`-by-1 cell vector of numeric vectors. Cell t of `YMSE` contains an n_t-by-1 numeric vector of error variances for the corresponding forecasted observations for period t.

X — State forecasts

matrix | cell vector of numeric vectors

State forecasts, returned as a matrix or a cell vector of numeric vectors.

If `Mdl` is a time-invariant, state-space model with respect to the states, then `X` is a `numPeriods`-by-m matrix.

If `Mdl` is a time-varying, state-space model with respect to the states, then `X` is a `numPeriods`-by-1 cell vector of numeric vectors. Cell t of `X` contains an m_t-by-1 numeric vector of forecasted observations for period t.

XMSE — Error variances of state forecasts

matrix | cell vector of numeric vectors

Error variances of state forecasts, returned as a matrix or a cell vector of numeric vectors.

If `Mdl` is a time-invariant, state-space model with respect to the states, then `XMSE` is a `numPeriods`-by-m matrix.

If `Mdl` is a time-varying, state-space model with respect to the states, then `XMSE` is a `numPeriods`-by-1 cell vector of numeric vectors. Cell t of `XMSE` contains an m_t-by-1 numeric vector of error variances for the corresponding forecasted observations for period t.

Examples

Forecast Observations of Time-Invariant Diffuse State-Space Model

Suppose that a latent process is a random walk. Subsequently, the state equation is

$$x_t = x_{t-1} + u_t,$$

where u_t is Gaussian with mean 0 and standard deviation 1.

Generate a random series of 100 observations from x_t, assuming that the series starts at 1.5.

```
T = 100;
x0 = 1.5;
rng(1); % For reproducibility
u = randn(T,1);
```

```
x = cumsum([x0;u]);
x = x(2:end);
```

Suppose further that the latent process is subject to additive measurement error. Subsequently, the observation equation is

$$y_t = x_t + \varepsilon_t,$$

where ε_t is Gaussian with mean 0 and standard deviation 0.75. Together, the latent process and observation equations compose a state-space model.

Use the random latent state process (`x`) and the observation equation to generate observations.

```
y = x + 0.75*randn(T,1);
```

Specify the four coefficient matrices.

```
A = 1;
B = 1;
C = 1;
D = 0.75;
```

Create the diffuse state-space model using the coefficient matrices. Specify that the inital state distribution is diffuse.

```
Mdl = dssm(A,B,C,D,'StateType',2)
```

```
Mdl =


State-space model type: <a href="matlab: doc dssm">dssm</a>

State vector length: 1
Observation vector length: 1
State disturbance vector length: 1
Observation innovation vector length: 1
Sample size supported by model: Unlimited

State variables: x1, x2,...
State disturbances: u1, u2,...
Observation series: y1, y2,...
Observation innovations: e1, e2,...

State equation:
```

```
x1(t) = x1(t-1) + u1(t)

Observation equation:
y1(t) = x1(t) + (0.75)e1(t)

Initial state distribution:

Initial state means
 x1
  0

Initial state covariance matrix
     x1
 x1  Inf

State types
    x1
 Diffuse
```

`Mdl` is an `dssm` model. Verify that the model is correctly specified using the display in the Command Window.

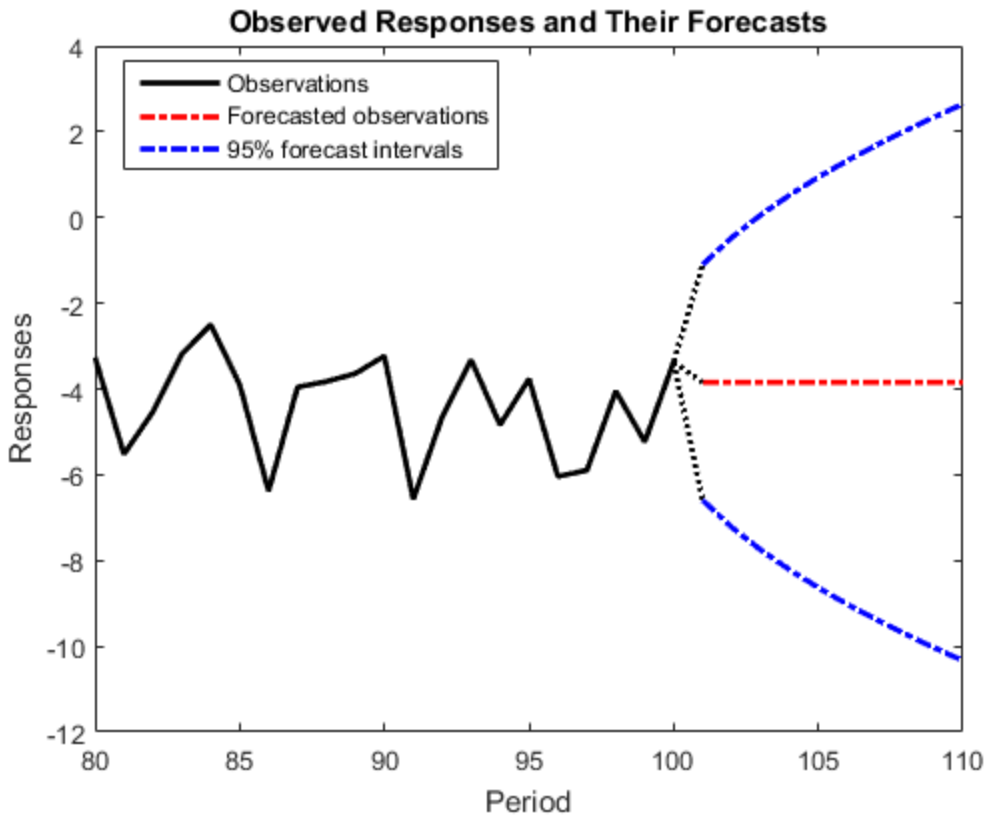
Forecast observations 10 periods into the future, and estimate the mean squared errors of the forecasts.

```
numPeriods = 10;
[ForecastedY,YMSE] = forecast(Mdl,numPeriods,y);
```

Plot the forecasts with the in-sample responses, and 95% Wald-type forecast intervals.

```
ForecastIntervals(:,1) = ForecastedY - 1.96*sqrt(YMSE);
ForecastIntervals(:,2) = ForecastedY + 1.96*sqrt(YMSE);

figure
plot(T-20:T,y(T-20:T),'-k',T+1:T+numPeriods,ForecastedY,'-.r',...
    T+1:T+numPeriods,ForecastIntervals,'-.b',...
    T:T+1,[y(end)*ones(3,1),[ForecastedY(1);ForecastIntervals(1,:)']],':k',...
    'LineWidth',2)
hold on
title({'Observed Responses and Their Forecasts'})
xlabel('Period')
ylabel('Responses')
legend({'Observations','Forecasted observations','95% forecast intervals'},...
    'Location','Best')
```

```
hold off
```



The forecast intervals flare out because the process is nonstationary.

Forecast Observations of Diffuse State-Space Model Containing Regression Component

Suppose that the linear relationship between unemployment rate and the nominal gross national product (nGNP) is of interest. Suppose further that unemployment rate is an AR(1) series. Symbolically, and in state-space form, the model is

$$x_t = \phi x_{t-1} + \sigma u_t$$
$$y_t - \beta Z_t = x_t,$$

where:

- x_t is the unemployment rate at time t.
- y_t is the observed change in the unemployment rate being deflated by the return of nGNP (Z_t).
- u_t is the Gaussian series of state disturbances having mean 0 and unknown standard deviation σ.

Load the Nelson-Plosser data set, which contains the unemployment rate and nGNP series, among other things.

```
load Data_NelsonPlosser
```

Preprocess the data by taking the natural logarithm of the nGNP series, and removing the starting `NaN` values from each series.

```
isNaN = any(ismissing(DataTable),2);       % Flag periods containing NaNs
gnpn = DataTable.GNPN(~isNaN);
y = diff(DataTable.UR(~isNaN));
T = size(gnpn,1);                          % The sample size
Z = price2ret(gnpn);
```

This example continues using the series without `NaN` values. However, using the Kalman filter framework, the software can accommodate series containing missing values.

To determine how well the model forecasts observations, remove the last 10 observations for comparison.

```
numPeriods = 10;                    % Forecast horizon
isY = y(1:end-numPeriods);          % In-sample observations
oosY = y(end-numPeriods+1:end);     % Out-of-sample observations
ISZ = Z(1:end-numPeriods);          % In-sample predictors
OOSZ = Z(end-numPeriods+1:end);  % Out-of-sample predictors
```

Specify the coefficient matrices.

```
A = NaN;
B = NaN;
C = 1;
```

Create the state-space model using `dssm` by supplying the coefficient matrices and specifying that the state values come from a diffuse distribution. The diffuse specification indicates complete ignorance about the moments of the initial distribution.

```
StateType = 2;
Mdl = dssm(A,B,C,'StateType',StateType);
```

Estimate the parameters. Specify the regression component and its initial value for optimization using the `'Predictors'` and `'Beta0'` name-value pair arguments, respectively. Display the estimates and all optimization diagnostic information. Restrict the estimate of σ to all positive, real numbers.

```
params0 = [0.3 0.2]; % Initial values chosen arbitrarily
Beta0 = 0.1;
[EstMdl,estParams] = estimate(Mdl,y,params0,'Predictors',Z,'Beta0',Beta0,...
    'lb',[-Inf 0 -Inf]);
```

```
Method: Maximum likelihood (fmincon)
Effective Sample size:             60
Logarithmic  likelihood:     -110.477
Akaike   info criterion:      226.954
Bayesian info criterion:      233.287
           |      Coeff       Std Err    t Stat    Prob
-------------------------------------------------------
 c(1)      |   0.59436       0.09408     6.31738  0
 c(2)      |   1.52554       0.10758    14.17991  0
 y <- z(1) | -24.26161       1.55730   -15.57930  0
           |
           |    Final State   Std Dev     t Stat   Prob
 x(1)      |   2.54764        0           Inf     0
```

`EstMdl` is a `dssm` model, and you can access its properties using dot notation.

Forecast observations over the forecast horizon. `EstMdl` does not store the data set, so you must pass it in appropriate name-value pair arguments.

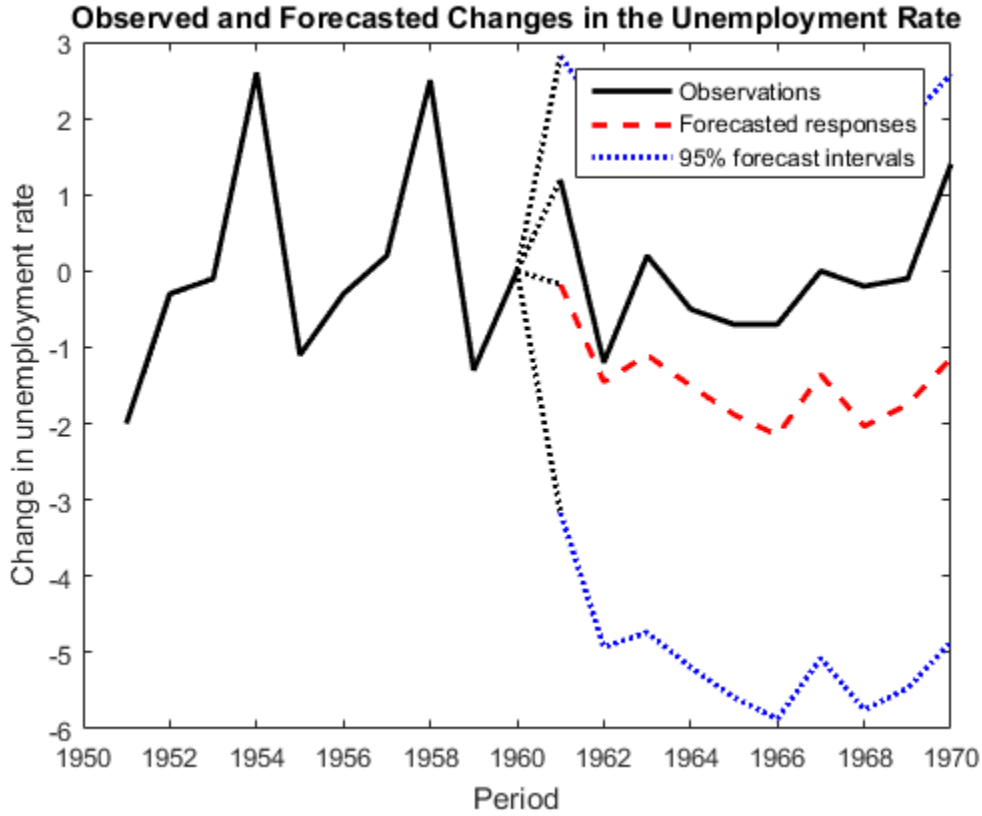
```
[fY,yMSE] = forecast(EstMdl,numPeriods,isY,'PredictorsO',ISZ,...
    'PredictorsF',OOSZ,'Beta',estParams(end));
```

`fY` is a 10-by-1 vector containing the forecasted observations, and `yMSE` is a 10-by-1 vector containing the variances of the forecasted observations.

Obtain 95% Wald-type forecast intervals. Plot the forecasted observations with their true values and the forecast intervals.

```
ForecastIntervals(:,1) = fY - 1.96*sqrt(yMSE);
ForecastIntervals(:,2) = fY + 1.96*sqrt(yMSE);

figure
h = plot(dates(end-numPeriods-9:end-numPeriods),isY(end-9:end),'-k',...
    dates(end-numPeriods+1:end),oosY,'-k',...
```

```
    dates(end-numPeriods+1:end),fY,'--r',...
    dates(end-numPeriods+1:end),ForecastIntervals,':b',...
    dates(end-numPeriods:end-numPeriods+1),...
    [isY(end)*ones(4,1),[oosY(1);ForecastIntervals(1,:)';fY(1)]],':k',...
    'LineWidth',2);
xlabel('Period')
ylabel('Change in unemployment rate')
legend(h([1,3,4]),{'Observations','Forecasted responses',...
    '95% forecast intervals'})
title('Observed and Forecasted Changes in the Unemployment Rate')
```



Forecast States of Diffuse State-Space Model

Suppose that a latent process is a random walk. Subsequently, the state equation is

$$x_t = x_{t-1} + u_t,$$

where u_t is Gaussian with mean 0 and standard deviation 1.

Generate a random series of 100 observations from x_t, assuming that the series starts at 1.5.

```
T = 100;
x0 = 1.5;
rng(1); % For reproducibility
u = randn(T,1);
x = cumsum([x0;u]);
x = x(2:end);
```

Suppose further that the latent process is subject to additive measurement error. Subsequently, the observation equation is

$$y_t = x_t + \varepsilon_t,$$

where ε_t is Gaussian with mean 0 and standard deviation 0.75. Together, the latent process and observation equations compose a state-space model.

Use the random latent state process (`x`) and the observation equation to generate observations.

```
y = x + 0.75*randn(T,1);
```

Specify the four coefficient matrices.

```
A = 1;
B = 1;
C = 1;
D = 0.75;
```

Create the diffuse state-space model using the coefficient matrices. Specify that the inital state distribution is diffuse.

```
Mdl = dssm(A,B,C,D,'StateType',2)
```

```
Mdl =


State-space model type: <a href="matlab: doc dssm">dssm</a>
```

```
State vector length: 1
Observation vector length: 1
State disturbance vector length: 1
Observation innovation vector length: 1
Sample size supported by model: Unlimited

State variables: x1, x2,...
State disturbances: u1, u2,...
Observation series: y1, y2,...
Observation innovations: e1, e2,...

State equation:
x1(t) = x1(t-1) + u1(t)

Observation equation:
y1(t) = x1(t) + (0.75)e1(t)

Initial state distribution:

Initial state means
 x1
  0

Initial state covariance matrix
     x1
 x1  Inf

State types
    x1
 Diffuse
```

`Mdl` is an `dssm` model. Verify that the model is correctly specified using the display in the Command Window.

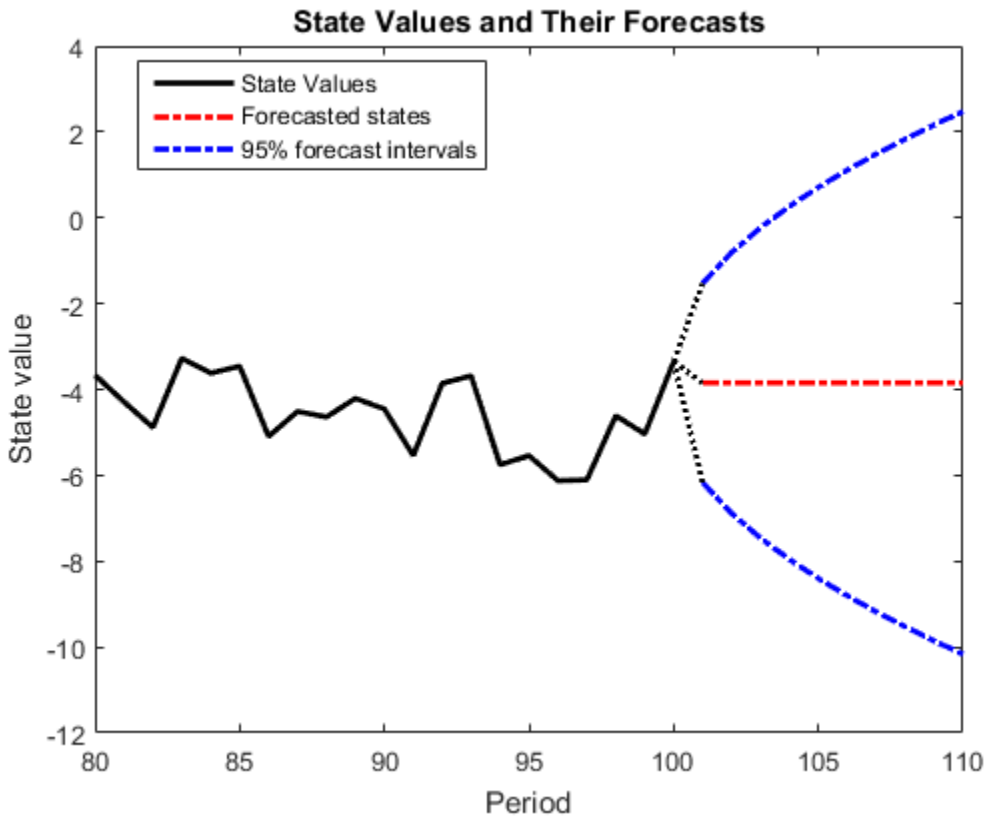
Forecast states 10 periods into the future, and estimate the mean squared errors of the forecasts.

```
numPeriods = 10;
[~,~,ForecastedX,XMSE] = forecast(Mdl,numPeriods,y);
```

Plot the forecasts with the in-sample states, and 95% Wald-type forecast intervals.

```
ForecastIntervals(:,1) = ForecastedX - 1.96*sqrt(XMSE);
```

```
ForecastIntervals(:,2) = ForecastedX + 1.96*sqrt(XMSE);

figure
plot(T-20:T,x(T-20:T),'-k',T+1:T+numPeriods,ForecastedX,'-.r',...
    T+1:T+numPeriods,ForecastIntervals,'-.b',...
    T:T+1,[x(end)*ones(3,1),[ForecastedX(1);ForecastIntervals(1,:)']],':k',...
    'LineWidth',2)
hold on
title({'State Values and Their Forecasts'})
xlabel('Period')
ylabel('State value')
legend({'State Values','Forecasted states','95% forecast intervals'},...
    'Location','Best')
hold off
```



The forecast intervals flare out because the process is nonstationary.

- "Forecast Time-Varying Diffuse State-Space Model" on page 8-156
- "Choose State-Space Model Specification Using Backtesting" on page 8-181

Algorithms

The Kalman filter accommodates missing data by not updating filtered state estimates corresponding to missing observations. In other words, suppose there is a missing observation at period t. Then, the state forecast for period t based on the previous $t-1$ observations and filtered state for period t are equivalent.

References

[1] Durbin J., and S. J. Koopman. *Time Series Analysis by State Space Methods*. 2nd ed. Oxford: Oxford University Press, 2012.

See Also

dssm | estimate | filter | smooth

More About

- "What Are State-Space Models?" on page 8-3
- "What Is the Kalman Filter?" on page 8-8
- "Rolling Window Analysis for Predictive Performance" on page 8-169

Introduced in R2015b

forecast

Class: ssm

Forecast states and observations of state-space models

Syntax

```
[Y,YMSE] = forecast(Mdl,numPeriods,Y0)
[Y,YMSE] = forecast(Mdl,numPeriods,Y0,Name,Value)
[Y,YMSE,X,XMSE] = forecast( ___ )
```

Description

`[Y,YMSE] = forecast(Mdl,numPeriods,Y0)` returns forecasted observations (`Y`) and their corresponding variances (`YMSE`) from forecasting the state-space model `Mdl` using a `numPeriods` forecast horizon and in-sample observations `Y0`.

`[Y,YMSE] = forecast(Mdl,numPeriods,Y0,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments. For example, for state-space models that include a linear regression component in the observation model, include in-sample predictor data, predictor data for the forecast horizon, and the regression coefficient.

`[Y,YMSE,X,XMSE] = forecast( ___ )` uses any of the input arguments in the previous syntaxes to additionally return state forecasts (`X`) and their corresponding variances (`XMSE`).

Input Arguments

`Mdl` — Standard state-space model

`ssm` model object

Standard state-space model, specified as an `ssm` model object returned by ssm or estimate.

If `Mdl` is not fully specified (that is, `Mdl` contains unknown parameters), then specify values for the unknown parameters using the `'Params'` name-value pair argument.

Otherwise, the software issues an error. `estimate` returns fully-specified state-space models.

`Mdl` does not store observed responses or predictor data. Supply the data wherever necessary using the appropriate input or name-value pair arguments.

`numPeriods` — Forecast horizon
positive integer

Forecast horizon, specified as a positive integer. That is, the software returns 1,..,`numPeriods` forecasts.

Data Types: `double`

`Y0` — In-sample, observed responses
cell vector of numeric vectors | numeric matrix

In-sample, observed responses, specified as a cell vector of numeric vectors or a matrix.

- If `Mdl` is time invariant, then `Y0` is a T-by-n numeric matrix, where each row corresponds to a period and each column corresponds to a particular observation in the model. Therefore, T is the sample size and m is the number of observations per period. The last row of `Y` contains the latest observations.
- If `Mdl` is time varying with respect to the observation equation, then `Y` is a T-by-1 cell vector. Each element of the cell vector corresponds to a period and contains an n_t-dimensional vector of observations for that period. The corresponding dimensions of the coefficient matrices in `Mdl.C{t}` and `Mdl.D{t}` must be consistent with the matrix in `Y{t}` for all periods. The last cell of `Y` contains the latest observations.

If `Mdl` is an estimated state-space model (that is, returned by estimate), then it is best practice to set `Y0` to the same data set that you used to fit `Mdl`.

`NaN` elements indicate missing observations. For details on how the Kalman filter accommodates missing observations, see "Algorithms" on page 9-450.

Data Types: `double | cell`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'A'` — Forecast-horizon, state-transition, coefficient matrices

cell vector of numeric matrices

Forecast-horizon, state-transition, coefficient matrices, specified as the comma-separated pair consisting of `'A'` and a cell vector of numeric matrices.

`A` must contain at least `numPeriods` cells. Each cell must contain a matrix specifying how the states transition in the forecast horizon. If the length of `A` is greater than `numPeriods`, then the software uses the first `numPeriods` cells. The last cell indicates the latest period in the forecast horizon.

The matrices in `A` cannot contain `NaN` values.

If `Mdl` is time invariant with respect to the states, then each cell of `A` must contain an m-by-m matrix, where m is the number of the in-sample states per period. By default, the software uses `Mdl.A` throughout the forecast horizon.

If `Mdl` is time varying with respect to the states, then the dimensions of the matrices in the cells of `A` may vary, but the dimensions of each matrix must be consistent with the matrices in `B` and `C` in the corresponding periods. By default, the software uses `Mdl.A{end}` throughout the forecast horizon.

Data Types: `cell`

`'B'` — Forecast-horizon, state-disturbance-loading, coefficient matrices

cell vector of matrices

Forecast-horizon, state-disturbance-loading, coefficient matrices, specified as the comma-separated pair consisting of `'B'` and a cell vector of matrices.

`B` must contain at least `numPeriods` cells. Each cell must contain a matrix specifying how the states transition in the forecast horizon. If the length of `B` is greater than `numPeriods`, then the software uses the first `numPeriods` cells. The last cell indicates the latest period in the forecast horizon.

The matrices in `B` cannot contain `NaN` values.

If `Mdl` is time invariant with respect to the states and state disturbances, then each cell of `B` must contain an m-by-k matrix, where m is the number of the in-sample states per

period, and k is the number of in-sample, state disturbances per period. By default, the software uses `Mdl.B` throughout the forecast horizon.

If `Mdl` is time varying, then the dimensions of the matrices in the cells of `B` may vary, but the dimensions of each matrix must be consistent with the matrices in `A` in the corresponding periods. By default, the software uses `Mdl.B{end}` throughout the forecast horizon.

Data Types: `cell`

`'C'` — Forecast-horizon, measurement-sensitivity, coefficient matrices

cell vector of matrices

Forecast-horizon, measurement-sensitivity, coefficient matrices, specified as the comma-separated pair consisting of `'C'` and a cell vector of matrices.

`C` must contain at least `numPeriods` cells. Each cell must contain a matrix specifying how the states transition in the forecast horizon. If the length of `C` is greater than `numPeriods`, then the software uses the first `numPeriods` cells. The last cell indicates the latest period in the forecast horizon.

The matrices in `C` cannot contain `NaN` values.

If `Mdl` is time invariant with respect to the states and the observations, then each cell of `C` must contain an n-by-m matrix, where n is the number of the in-sample observations per period, and m is the number of in-sample states per period. By default, the software uses `Mdl.C` throughout the forecast horizon.

If `Mdl` is time varying with respect to the states or the observations, then the dimensions of the matrices in the cells of `C` may vary, but the dimensions of each matrix must be consistent with the matrices in `A` and `D` in the corresponding periods. By default, the software uses `Mdl.C{end}` throughout the forecast horizon.

Data Types: `cell`

`'D'` — Forecast-horizon, observation-innovation, coefficient matrices

cell vector of matrices

Forecast-horizon, observation-innovation, coefficient matrices, specified as the comma-separated pair consisting of `'D'` and a cell vector of matrices.

`D` must contain at least `numPeriods` cells. Each cell must contain a matrix specifying how the states transition in the forecast horizon. If the length of `D` is greater than

numPeriods, then the software uses the first numPeriods cells. The last cell indicates the latest period in the forecast horizon.

The matrices in D cannot contain NaN values.

If Mdl is time invariant with respect to the observations and the observation innovations, then each cell of D must contain an n-by-h matrix, where n is the number of the in-sample observations per period, and h is the number of in-sample, observation innovations per period. By default, the software uses Mdl.D throughout the forecast horizon.

If Mdl is time varying with respect to the observations or the observation innovations, then the dimensions of the matrices in the cells of D may vary, but the dimensions of each matrix must be consistent with the matrices in C in the corresponding periods. By default, the software uses Mdl.D{end} throughout the forecast horizon.

Data Types: cell

'Beta' — Regression coefficients

[ ] (default) | numeric matrix

Regression coefficients corresponding to predictor variables, specified as the comma-separated pair consisting of 'Beta' and a d-by-n numeric matrix. d is the number of predictor variables (see PredictorsO and PredictorsF) and n is the number of observed response series (see YO).

If you specify Beta, then you must also specify PredictorsO and PredictorsF.

If Mdl is an estimated state-space model, then specify the estimated regression coefficients stored in Mdl.estParams.

By default, the software excludes a regression component from the state-space model.

'PredictorsO' — In-sample, predictor variables in state-space model observation equation

[ ] (default) | matrix

In-sample, predictor variables in the state-space model observation equation, specified as the comma-separated pair consisting of 'PredictorsO' and a matrix. The columns of PredictorsO correspond to individual predictor variables. PredictorsO must have T rows, where row t corresponds to the observed predictors at period t (Z_t). The expanded observation equation is

$$y_t - Z_t\beta = Cx_t + Du_t.$$

that is, the software deflates the observations using the regression component. β is the time-invariant vector of regression coefficients that the software estimates with all other parameters.

If there are n observations per period, then the software regresses all predictor series onto each observation.

If you specify `PredictorsO`, then `Mdl` must be time invariant. Otherwise, the software returns an error.

If you specify `PredictorsO`, then you must also specify `Beta` and `PredictorsF`.

If `Mdl` is an estimated state-space model (that is, returned by estimate), then it is best practice to set `PredictorsO` to the same predictor data set that you used to fit `Mdl`.

By default, the software excludes a regression component from the state-space model.

Data Types: `double`

`'PredictorsF'` — Forecast-horizon, predictor variables in state-space model observation equation

[ ] (default) | numeric matrix

In-sample, predictor variables in the state-space model observation equation, specified as the comma-separated pair consisting of `'PredictorsO'` and a T-by-d numeric matrix. T is the number of in-sample periods and d is the number of predictor variables. Row t corresponds to the observed predictors at period t (Z_t). The expanded observation equation is

$$y_t - Z_t\beta = Cx_t + Du_t.$$

That is, the software deflates the observations using the regression component. β is the time-invariant vector of regression coefficients that the software estimates with all other parameters.

If there are n observations per period, then the software regresses all predictor series onto each observation.

If you specify `Predictors0`, then `Mdl` must be time invariant. Otherwise, the software returns an error.

If you specify `Predictors0`, then you must also specify `Beta` and `PredictorsF`.

If `Mdl` is an estimated state-space model (that is, returned by `estimate`), then it is best practice to set `Predictors0` to the same predictor data set that you used to fit `Mdl`.

By default, the software excludes a regression component from the state-space model.

Data Types: `double`

Output Arguments

`Y` — Forecasted observations

matrix | cell vector of numeric vectors

Forecasted observations, returned as a matrix or a cell vector of numeric vectors.

If `Mdl` is a time-invariant, state-space model with respect to the observations, then `Y` is a `numPeriods`-by-n matrix.

If `Mdl` is a time-varying, state-space model with respect to the observations, then `Y` is a `numPeriods`-by-1 cell vector of numeric vectors. Cell t of `Y` contains an n_t-by-1 numeric vector of forecasted observations for period t.

`YMSE` — Error variances of forecasted observations

matrix | cell vector of numeric vectors

Error variances of forecasted observations, returned as a matrix or a cell vector of numeric vectors.

If `Mdl` is a time-invariant, state-space model with respect to the observations, then `YMSE` is a `numPeriods`-by-n matrix.

If `Mdl` is a time-varying, state-space model with respect to the observations, then `YMSE` is a `numPeriods`-by-1 cell vector of numeric vectors. Cell t of `YMSE` contains an n_t-by-1 numeric vector of error variances for the corresponding forecasted observations for period t.

X — State forecasts

matrix | cell vector of numeric vectors

State forecasts, returned as a matrix or a cell vector of numeric vectors.

If `Mdl` is a time-invariant, state-space model with respect to the states, then `X` is a `numPeriods`-by-m matrix.

If `Mdl` is a time-varying, state-space model with respect to the states, then `X` is a `numPeriods`-by-1 cell vector of numeric vectors. Cell t of `X` contains an m_t-by-1 numeric vector of forecasted observations for period t.

XMSE — Error variances of state forecasts

matrix | cell vector of numeric vectors

Error variances of state forecasts, returned as a matrix or a cell vector of numeric vectors.

If `Mdl` is a time-invariant, state-space model with respect to the states, then `XMSE` is a `numPeriods`-by-m matrix.

If `Mdl` is a time-varying, state-space model with respect to the states, then `XMSE` is a `numPeriods`-by-1 cell vector of numeric vectors. Cell t of `XMSE` contains an m_t-by-1 numeric vector of error variances for the corresponding forecasted observations for period t.

Examples

Forecast Observations of Time-Invariant State-Space Model

Suppose that a latent process is an AR(1). Subsequently, the state equation is

$$x_t = 0.5x_{t-1} + u_t,$$

where u_t is Gaussian with mean 0 and standard deviation 1.

Generate a random series of 100 observations from x_t, assuming that the series starts at 1.5.

```
T = 100;
ARMdl = arima('AR',0.5,'Constant',0,'Variance',1);
x0 = 1.5;
rng(1); % For reproducibility
x = simulate(ARMdl,T,'Y0',x0);
```

Suppose further that the latent process is subject to additive measurement error. Subsequently, the observation equation is

$$y_t = x_t + \varepsilon_t,$$

where ε_t is Gaussian with mean 0 and standard deviation 0.75. Together, the latent process and observation equations compose a state-space model.

Use the random latent state process (`x`) and the observation equation to generate observations.

```
y = x + 0.75*randn(T,1);
```

Specify the four coefficient matrices.

```
A = 0.5;
B = 1;
C = 1;
D = 0.75;
```

Specify the state-space model using the coefficient matrices.

```
Mdl = ssm(A,B,C,D)
```

```
Mdl =


State-space model type: <a href="matlab: doc ssm">ssm</a>

State vector length: 1
Observation vector length: 1
State disturbance vector length: 1
Observation innovation vector length: 1
Sample size supported by model: Unlimited

State variables: x1, x2,...
```

```
State disturbances: u1, u2,...
Observation series: y1, y2,...
Observation innovations: e1, e2,...

State equation:
x1(t) = (0.50)x1(t-1) + u1(t)

Observation equation:
y1(t) = x1(t) + (0.75)e1(t)

Initial state distribution:

Initial state means
 x1
  0

Initial state covariance matrix
     x1
 x1  1.33

State types
     x1
 Stationary
```

`Mdl` is an `ssm` model. Verify that the model is correctly specified using the display in the Command Window. The software infers that the state process is stationary. Subsequently, the software sets the initial state mean and covariance to the mean and variance of the stationary distribution of an AR(1) model.

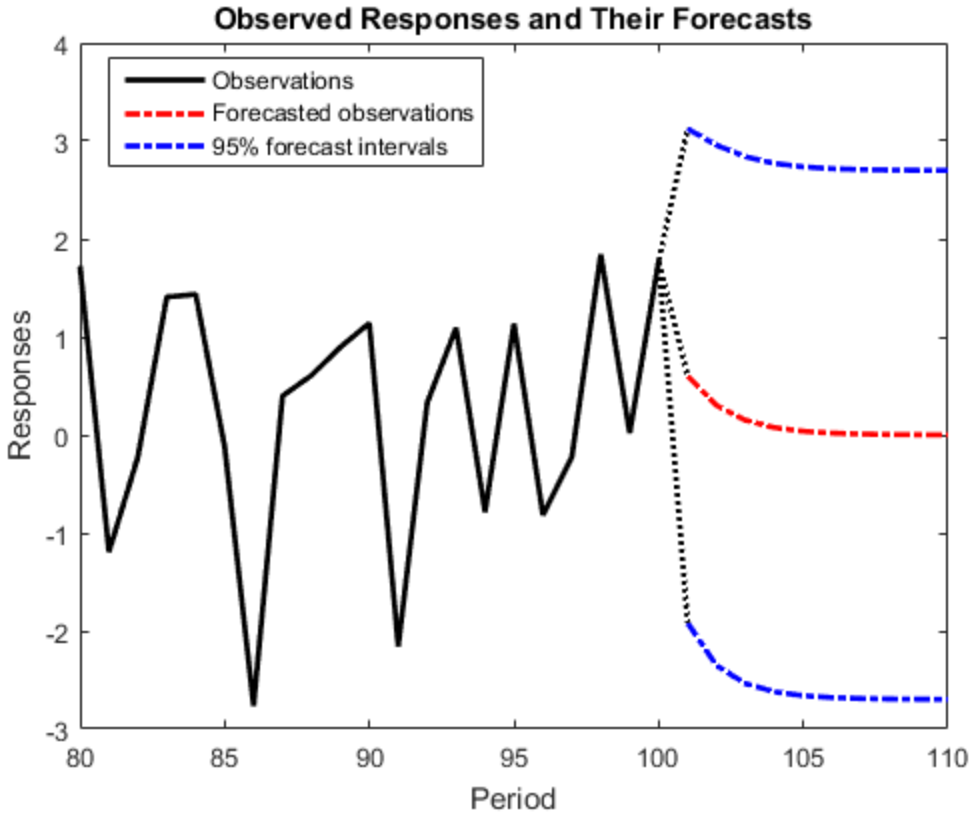
Forecast the observations 10 periods into the future, and estimate their variances.

```
numPeriods = 10;
[ForecastedY,YMSE] = forecast(Mdl,numPeriods,y);
```

Plot the forecasts with the in-sample responses, and 95% Wald-type forecast intervals.

```
ForecastIntervals(:,1) = ForecastedY - 1.96*sqrt(YMSE);
ForecastIntervals(:,2) = ForecastedY + 1.96*sqrt(YMSE);

figure
plot(T-20:T,y(T-20:T),'-k',T+1:T+numPeriods,ForecastedY,'-.r',...
    T+1:T+numPeriods,ForecastIntervals,'-.b',...
    T:T+1,[y(end)*ones(3,1),[ForecastedY(1);ForecastIntervals(1,:)']],':k',...
```

```
    'LineWidth',2)
hold on
title({'Observed Responses and Their Forecasts'})
xlabel('Period')
ylabel('Responses')
legend({'Observations','Forecasted observations','95% forecast intervals'},...
    'Location','Best')
hold off
```



Forecast Observations of State-Space Model Containing Regression Component

Suppose that the linear relationship between the change in the unemployment rate and the nominal gross national product (nGNP) growth rate is of interest. Suppose further

that the first difference of the unemployment rate is an ARMA(1,1) series. Symbolically, and in state-space form, the model is

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix} = \begin{bmatrix} \phi & \theta \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix} u_{1,t}$$
$$y_t - \beta Z_t = x_{1,t} + \sigma \varepsilon_t,$$

where:

- $x_{1,t}$ is the change in the unemployment rate at time t.
- $x_{2,t}$ is a dummy state for the MA(1) effect.
- $y_{1,t}$ is the observed change in the unemployment rate being deflated by the growth rate of nGNP (Z_t).
- $u_{1,t}$ is the Gaussian series of state disturbances having mean 0 and standard deviation 1.
- ε_t is the Gaussian series of observation innovations having mean 0 and standard deviation σ.

Load the Nelson-Plosser data set, which contains the unemployment rate and nGNP series, among other things.

```
load Data_NelsonPlosser
```

Preprocess the data by taking the natural logarithm of the nGNP series, and the first difference of each series. Also, remove the starting `NaN` values from each series.

```
isNaN = any(ismissing(DataTable),2);       % Flag periods containing NaNs
gnpn = DataTable.GNPN(~isNaN);
u = DataTable.UR(~isNaN);
T = size(gnpn,1);                          % Sample size
Z = [ones(T-1,1) diff(log(gnpn))];
y = diff(u);
```

Though this example removes missing values, the software can accommodate series containing missing values in the Kalman filter framework.

To determine how well the model forecasts observations, remove the last 10 observations for comparison.

```
numPeriods = 10;                    % Forecast horizon
```

```
isY = y(1:end-numPeriods);          % In-sample observations
oosY = y(end-numPeriods+1:end);     % Out-of-sample observations
ISZ = Z(1:end-numPeriods,:);        % In-sample predictors
OOSZ = Z(end-numPeriods+1:end,:);  % Out-of-sample predictors
```

Specify the coefficient matrices.

```
A = [NaN NaN; 0 0];
B = [1; 1];
C = [1 0];
D = NaN;
```

Specify the state-space model using `ssm`.

```
Mdl = ssm(A,B,C,D);
```

Estimate the model parameters, and use a random set of initial parameter values for optimization. Specify the regression component and its initial value for optimization using the `'Predictors'` and `'Beta0'` name-value pair arguments, respectively. Restrict the estimate of σ to all positive, real numbers. For numerical stability, specify the Hessian when the software computes the parameter covariance matrix, using the `'CovMethod'` name-value pair argument.

```
params0 = [0.3 0.2 0.1]; % Chosen arbitrarily
[EstMdl,estParams] = estimate(Mdl,isY,params0,'Predictors',ISZ,...
    'Beta0',[0.1 0.2],'lb',[-Inf,-Inf,0,-Inf,-Inf],'CovMethod','hessian');
```

```
Method: Maximum likelihood (fmincon)
Sample size: 51
Logarithmic  likelihood:     -87.2409
Akaike   info criterion:      184.482
Bayesian info criterion:      194.141
           |      Coeff       Std Err    t Stat     Prob
-------------------------------------------------------------
 c(1)      |  -0.31780       0.19429    -1.63572  0.10190
 c(2)      |   1.21242       0.48882     2.48031  0.01313
 c(3)      |   0.45583       0.63930     0.71301  0.47584
 y <- z(1) |   1.32407       0.26313     5.03201   0
 y <- z(2) | -24.48733       1.90115   -12.88024   0
           |
           |    Final State   Std Dev     t Stat    Prob
 x(1)      |  -0.38117       0.42842    -0.88971  0.37363
 x(2)      |   0.23402       0.66222     0.35339  0.72380
```

`EstMdl` is an `ssm` model, and you can access its properties using dot notation.

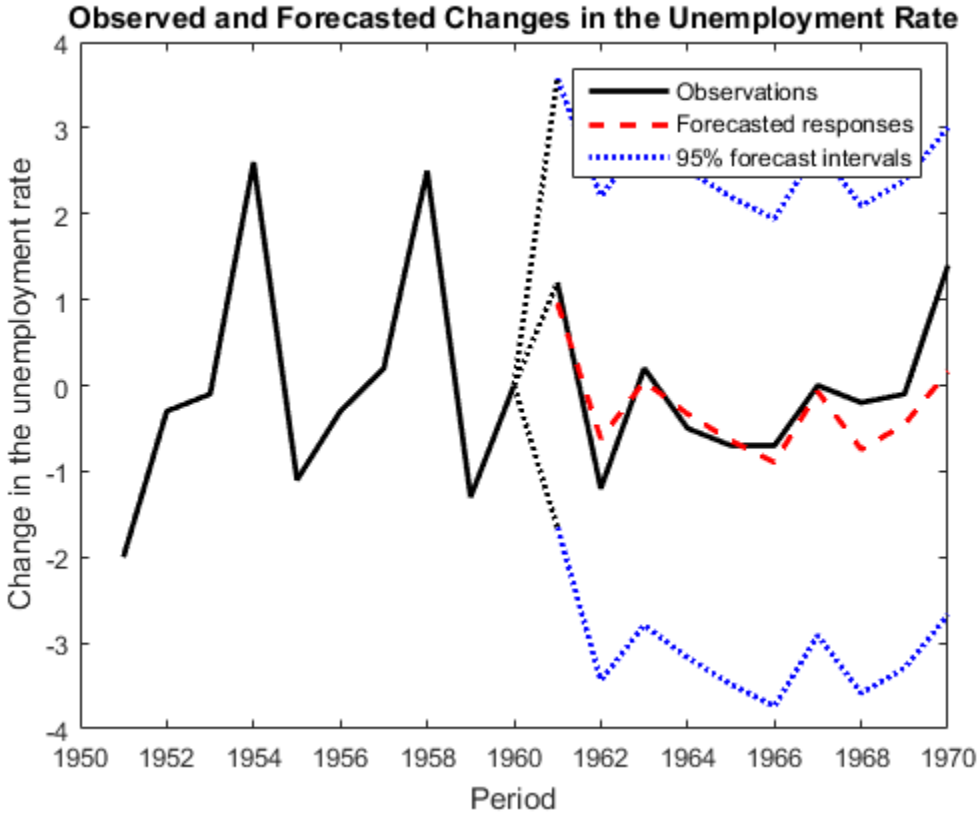
Forecast observations over the forecast horizon. `EstMdl` does not store the data set, so you must pass it in appropriate name-value pair arguments.

```
[fY,yMSE] = forecast(EstMdl,numPeriods,isY,'PredictorsO',ISZ,...
    'PredictorsF',OOSZ,'Beta',estParams(end-1:end));
```

`fY` is a 10-by-1 vector containing the forecasted observations, and `yMSE` is a 10-by-1 vector containing the variances of the forecasted observations.

Obtain 95% Wald-type forecast intervals. Plot the forecasted observations with their true values and the forecast intervals.

```
ForecastIntervals(:,1) = fY - 1.96*sqrt(yMSE);
ForecastIntervals(:,2) = fY + 1.96*sqrt(yMSE);

figure
h = plot(dates(end-numPeriods-9:end-numPeriods),isY(end-9:end),'-k',...
    dates(end-numPeriods+1:end),oosY,'-k',...
    dates(end-numPeriods+1:end),fY,'--r',...
    dates(end-numPeriods+1:end),ForecastIntervals,':b',...
    dates(end-numPeriods:end-numPeriods+1),...
    [isY(end)*ones(3,1),[oosY(1);ForecastIntervals(1,:)']],':k',...
    'LineWidth',2);
xlabel('Period')
ylabel('Change in the unemployment rate')
legend(h([1,3,4]),{'Observations','Forecasted responses',...
    '95% forecast intervals'})
title('Observed and Forecasted Changes in the Unemployment Rate')
```



This model seems to forecast the changes in the unemployment rate well.

Forecast States of State-Space Model

Suppose that a latent process is an AR(1). Subsequently, the state equation is

$$x_t = 0.5x_{t-1} + u_t,$$

where u_t is Gaussian with mean 0 and standard deviation 1.

Generate a random series of 100 observations from x_t, assuming that the series starts at 1.5.

```
T = 100;
```

```
ARMdl = arima('AR',0.5,'Constant',0,'Variance',1);
x0 = 1.5;
rng(1); % For reproducibility
x = simulate(ARMdl,T,'Y0',x0);
```

Suppose further that the latent process is subject to additive measurement error. Subsequently, the observation equation is

$$y_t = x_t + \varepsilon_t,$$

where ε_t is Gaussian with mean 0 and standard deviation 0.75. Together, the latent process and observation equations compose a state-space model.

Use the random latent state process (`x`) and the observation equation to generate observations.

```
y = x + 0.75*randn(T,1);
```

Specify the four coefficient matrices.

```
A = 0.5;
B = 1;
C = 1;
D = 0.75;
```

Specify the state-space model using the coefficient matrices.

```
Mdl = ssm(A,B,C,D);
```

`Mdl` is an `ssm` model.

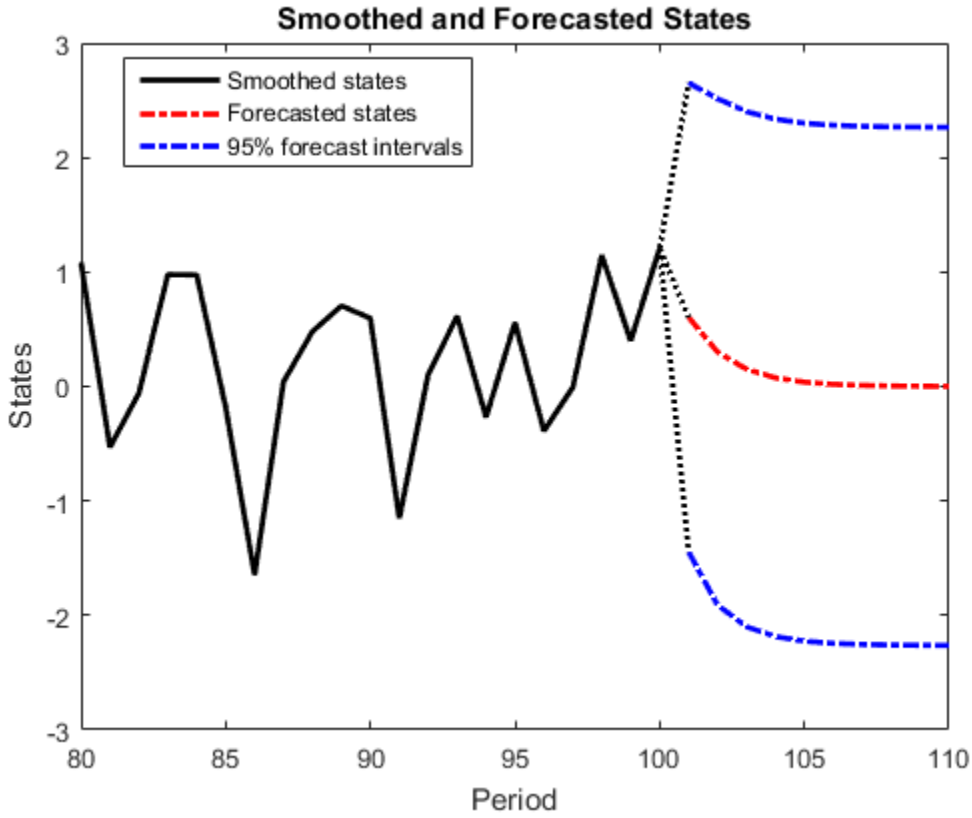
Forecast the states 10 periods into the future, and estimate their variances.

```
numPeriods = 10;
[~,~,ForecastedX,XMSE] = forecast(Mdl,numPeriods,y);
```

Plot the forecasts with the smoothed states, and 95% Wald-type forecast intervals.

```
smoothX = smooth(Mdl,y);
ForecastIntervals(:,1) = ForecastedX - 1.96*sqrt(XMSE);
ForecastIntervals(:,2) = ForecastedX + 1.96*sqrt(XMSE);

figure
plot(T-20:T,smoothX(T-20:T),'-k',T+1:T+numPeriods,ForecastedX,'-.r',...
    T+1:T+numPeriods,ForecastIntervals,'-.b',...
```

```
    T:T+1,[smoothX(end)*ones(3,1),[ForecastedX(1);ForecastIntervals(1,:)']],...
    ':k','LineWidth',2)
hold on
title({'Smoothed and Forecasted States'})
xlabel('Period')
ylabel('States')
legend({'Smoothed states','Forecasted states','95% forecast intervals'},...
    'Location','Best')
hold off
```



- “Forecast Time-Varying State-Space Model” on page 8-143
- “Forecast State-Space Model Using Monte-Carlo Methods” on page 8-125
- “Choose State-Space Model Specification Using Backtesting” on page 8-181

Algorithms

The Kalman filter accommodates missing data by not updating filtered state estimates corresponding to missing observations. In other words, suppose there is a missing observation at period t. Then, the state forecast for period t based on the previous $t-1$ observations and filtered state for period t are equivalent.

Tip

`Mdl` does not store the response data, predictor data, and the regression coefficients. Supply them whenever necessary using the appropriate input or name-value pair arguments.

References

[1] Durbin J., and S. J. Koopman. *Time Series Analysis by State Space Methods*. 2nd ed. Oxford: Oxford University Press, 2012.

See Also

estimate | filter | smooth | ssm

More About

- “What Are State-Space Models?” on page 8-3
- “What Is the Kalman Filter?” on page 8-8
- “Rolling Window Analysis for Predictive Performance” on page 8-169

garch

Create GARCH conditional variance model object

Create a `garch` model object to represent a generalized autoregressive conditional heteroscedastic (GARCH) model. The GARCH(P,Q) conditional variance model includes P past conditional variances composing the GARCH polynomial, and Q past squared innovations composing the ARCH polynomial.

Use `garch` to create a model with known or unknown coefficients, and then estimate any unknown coefficients from data using `estimate`. You can also simulate or forecast conditional variances from fully specified models using `simulate` or `forecast`, respectively.

For more information about `garch` model objects, see Using garch Objects.

Syntax

```
Mdl = garch
Mdl = garch(P,Q)
Mdl = garch(Name,Value)
```

Description

`Mdl = garch` creates a zero-degree conditional variance GARCH model object.

`Mdl = garch(P,Q)` creates a GARCH model with GARCH polynomial degree `P` and ARCH polynomial degree `Q`.

`Mdl = garch(Name,Value)` creates a GARCH model with additional options specified by one or more `Name,Value` pair arguments. For example, you can specify a conditional variance model constant, the number of ARCH polynomial lags, and the innovation distribution.

Examples

Create Default GARCH Model

Create a default `garch` model object and specify its parameter values using dot notation.

Create a GARCH(0,0) model.

```
Mdl = garch
```

```
Mdl =

    GARCH(0,0) Conditional Variance Model:
    --------------------------------------
    Distribution: Name = 'Gaussian'
               P: 0
               Q: 0
        Constant: NaN
           GARCH: {}
            ARCH: {}
```

`Mdl` is a `garch` model. It contains an unknown constant, its offset is `0`, and the innovation distribution is `'Gaussian'`. The model does not have a GARCH or ARCH polynomial.

Specify two unknown ARCH coefficients for lags one and two using dot notation.

```
Mdl.ARCH = {NaN NaN}
```

```
Mdl =

    GARCH(0,2) Conditional Variance Model:
    --------------------------------------
    Distribution: Name = 'Gaussian'
               P: 0
               Q: 2
        Constant: NaN
           GARCH: {}
            ARCH: {NaN NaN} at Lags [1 2]
```

The Q and ARCH properties are updated to 2 and {NaN NaN}. The two ARCH coefficients are associated with lags 1 and 2.

Create GARCH Model Using Shorthand Syntax

Create a garch model using the shorthand notation garch(P,Q), where P is the degree of the GARCH polynomial and Q is the degree of the ARCH polynomial.

Create a GARCH(3,2) model.

```
Mdl = garch(3,2)
```

```
Mdl =

    GARCH(3,2) Conditional Variance Model:
    --------------------------------------
    Distribution: Name = 'Gaussian'
               P: 3
               Q: 2
        Constant: NaN
           GARCH: {NaN NaN NaN} at Lags [1 2 3]
            ARCH: {NaN NaN} at Lags [1 2]
```

Mdl is a garch model object. All properties of Mdl, except P, Q, and Distribution, are NaN values. By default, the software:

- Includes a conditional variance model constant
- Excludes a conditional mean model offset (i.e., the offset is 0)
- Includes all lag terms in the ARCH and GARCH lag-operator polynomials up to lags Q and P, respectively

Mdl specifies only the functional form of a GARCH model. Because it contains unknown parameter values, you can pass Mdl and the time-series data to estimate to estimate the parameters.

Create GARCH Model

Create a garch model using name-value pair arguments.

Specify a GARCH(1,1) model. By default, the conditional mean model offset is zero. Specify that the offset is NaN.

```
Mdl = garch('GARCHLags',1,'ARCHLags',1,'Offset',NaN)
```

```
Mdl =

    GARCH(1,1) Conditional Variance Model with Offset:
    --------------------------------------------------
    Distribution: Name = 'Gaussian'
               P: 1
               Q: 1
        Constant: NaN
           GARCH: {NaN} at Lags [1]
            ARCH: {NaN} at Lags [1]
          Offset: NaN
```

`Mdl` is a `garch` model object. The software sets all parameters (the properties of the model object) to `NaN`, except `P`, `Q`, and `Distribution`.

Since `Mdl` contains `NaN` values, `Mdl` is only appropriate for estimation only. Pass `Mdl` and time-series data to `estimate`. For a continuation of this example, see "Estimate GARCH Model".

Create GARCH Model with Known Coefficients

Create a GARCH(1,1) model with mean offset,

$$y_t = 0.5 + \varepsilon_t,$$

where $\varepsilon_t = \sigma_t z_t,$

$$\sigma_t^2 = 0.0001 + 0.75\sigma_{t-1}^2 + 0.1\varepsilon_{t-1}^2,$$

and z_t is an independent and identically distributed standard Gaussian process.

```
Mdl = garch('Constant',0.0001,'GARCH',0.75,...
    'ARCH',0.1,'Offset',0.5)


Mdl =

    GARCH(1,1) Conditional Variance Model with Offset:
    --------------------------------------------------
    Distribution: Name = 'Gaussian'
               P: 1
               Q: 1
        Constant: 0.0001
```

```
 GARCH: {0.75} at Lags [1]
  ARCH: {0.1} at Lags [1]
Offset: 0.5
```

`garch` assigns default values to any properties you do not specify with name-value pair arguments.

- "Specify GARCH Models Using garch" on page 6-8
- "Modify Properties of Conditional Variance Models" on page 6-42
- "Specify the Conditional Variance Model Innovation Distribution" on page 6-48
- "Specify Conditional Mean and Variance Models" on page 5-79
- "Specify Conditional Variance Model For Exchange Rates" on page 6-53

Input Arguments

`P` — Number of past consecutive conditional variance terms

nonnegative integer

Number of past consecutive conditional variance terms to include in the GARCH polynomial, specified as a nonnegative integer. That is, `P` is the degree of the GARCH polynomial, where the polynomial includes each lag term from $t - 1$ to $t - P$. `P` also specifies the minimum number of presample conditional variances the software requires to initiate the model.

You can specify `P` using the `garch(P,Q)` shorthand syntax only. You cannot specify `P` in conjunction with `Name,Value` pair arguments.

If `P` > 0, then you must specify `Q` as a positive integer.

Example: `garch(3,2)`

Data Types: `double`

`Q` — Number of past consecutive squared innovation terms

nonnegative integer

Number of past consecutive squared innovation terms to include in the ARCH polynomial, specified as a nonnegative integer. That is, `Q` is the degree of the ARCH polynomial, where the polynomial includes each lag term from $t - 1$ to $t - Q$. `Q` also specifies the minimum number of presample innovations the software requires to initiate the model.

You can specify this property using the `garch(P,Q)` shorthand syntax only. You cannot specify `Q` in conjunction with `Name,Value` pair arguments.

If `P` > 0, then you must specify `Q` as a positive integer.

Example: `garch(3,2)`

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Constant',0.5,'ARCHLags',2,'Distribution',struct('Name','t','DoF',5)` specifies a conditional variance model constant of 0.5, two squared innovation terms at lags 1 and 2 of the ARCH polynomial, and a t distribution with 5 degrees of freedom for the innovations.

`'Constant'` — Conditional variance model constant

`NaN` (default) | positive scalar

Conditional variance model constant, specified as the comma-separated pair consisting of `'Constant'` and a positive scalar.

Example: `'Constant',0.5`

Data Types: `double`

`'GARCH'` — Coefficients corresponding to past conditional variance terms

cell vector of `NaN`s (default) | cell vector of nonnegative scalars

Coefficients corresponding to the past conditional variance terms that compose the GARCH polynomial, specified as the comma-separated pair consisting of `'GARCH'` and a cell vector of nonnegative scalars.

If you specify `GARCHLags`, then `GARCH` is an equivalent-length cell vector of coefficients associated with the lags in `GARCHLags`. Otherwise, `GARCH` is a *P*-element cell vector of coefficients corresponding to lags 1, 2,..., *P*.

The coefficients must compose a stationary model. For details, see "GARCH Model" on page 9-537.

By default, GARCH is a cell vector of NaNs of length P (the degree of the GARCH polynomial) or numel(GARCHLags).

Example: 'GARCH',{0.1 0 0 0.02}

Data Types: cell

'ARCH' — Coefficients corresponding to past squared innovation terms

cell vector of NaNs (default) | cell vector of nonnegative scalars

Coefficients corresponding to the past squared innovation terms that compose the ARCH polynomial, specified as the comma-separated pair consisting of 'ARCH' and a cell vector of nonnegative scalars.

If you specify ARCHLags, then ARCH is an equivalent-length cell vector of coefficients associated with the lags in ARCHLags. Otherwise, ARCH is a Q-element cell vector of coefficients corresponding to lags 1, 2,..., Q.

The coefficients must compose a stationary model. For details, see "GARCH Model" on page 9-537.

By default, ARCH is a cell vector of NaNs of length Q (the degree of the ARCH polynomial) or numel(ARCHLags).

Example: 'ARCH',{0.5 0 0.2}

Data Types: cell

'Offset' — Innovation mean model offset

0 (default) | scalar

Innovation mean model offset or additive constant, specified as the comma-separated pair consisting of 'Offset' and a scalar.

Example: 'Offset',0.1

Data Types: double

'GARCHLags' — Lags associated with GARCH polynomial coefficients

vector of positive integers

Lags associated with the GARCH polynomial coefficients, specified as the comma-separated pair consisting of `'GARCHLags'` and a vector of positive integers. The maximum value of `GARCHLags` determines P, the GARCH polynomial degree.

If you specify `GARCH`, then `GARCHLags` is an equivalent-length vector of positive integers specifying the lags of the corresponding coefficients in `GARCH`. Otherwise, `GARCHLags` indicates the lags of unknown coefficients in the GARCH polynomial.

By default, `GARCHLags` is a vector containing the integers 1 through P.

Example: `'GARCHLags',[1 2 4 3]`

Data Types: `double`

`'ARCHLags'` — Lags associated with the ARCH polynomial coefficients
vector of positive integers

Lags associated with the ARCH polynomial coefficients, specified as the comma-separated pair consisting of `'ARCHLags'` and a vector of positive integers. The maximum value of `ARCHLags` determines Q, the ARCH polynomial degree.

If you specify `ARCH`, then `ARCHLags` is an equivalent-length vector of positive integers specifying the lags of the corresponding coefficients in `ARCH`. Otherwise, `ARCHLags` indicates the lags of unknown coefficients in the ARCH polynomial.

By default, `ARCHLags` is a vector containing the integers 1 through Q.

Example: `'ARCHLags',[3 1 2]`

Data Types: `double`

`'Distribution'` — Conditional probability distribution of innovation process
`'Gaussian'` (default) | string | structure array

Conditional probability distribution of the innovation process, specified as the comma-separated pair consisting of `'Distribution'` and a string or a structure array.

This table contains the available distributions.

Distribution	String	Structure Array
Gaussian	`'Gaussian'`	`struct('Name','Gaussian')`
t	`'t'` By default, `DoF` is `NaN`.	`struct('Name','t','DoF',DoF)` `DoF` > 2 or `DoF` = `NaN`

Example: `'Distribution',struct('Name','t','DoF',10)`

Data Types: `char` | `struct`

Note: All GARCH and ARCH coefficients are subject to a near-zero tolerance exclusion test. That is, the software:

1. Creates lag operator polynomials for each of the GARCH and ARCH components.
2. Compares each coefficient to the default lag operator zero tolerance, `1e-12`.
3. Includes a coefficient in the model if its magnitude is greater than `1e-12`, and excludes the coefficient otherwise. In other words, the software considers excluded coefficients to be sufficiently close to zero.

For details, see LagOp.

Output Arguments

`Mdl` — GARCH model

`garch` model object

GARCH model, returned as a `garch` model object.

For the property descriptions of `Mdl`, see Conditional Variance Model Properties.

If `Mdl` contains unknown parameters (indicated by `NaN`s), then you can specify them using dot notation. Alternatively, you can pass `Mdl` and time series data to `estimate` to obtain estimates.

If `Mdl` is fully specified, then you can simulate or forecast conditional variances using `simulate` or `forecast`, respectively.

More About

GARCH Model

A *GARCH model* is an innovations process that addresses conditional heteroscedasticity. Specifically, the model posits that the current conditional variance is the sum of these linear processes, with coefficients for each term:

- Past conditional variances (the GARCH component or polynomial)
- Past squared innovations (the ARCH component or polynomial)

Consider the time series

$$y_t = \mu + \varepsilon_t,$$

where $\varepsilon_t = \sigma_t z_t$. The GARCH($P$,$Q$) conditional variance process, σ_t^2, has the form

$$\sigma_t^2 = \kappa + \gamma_1\sigma_{t-1}^2 + \ldots + \gamma_P\sigma_{t-P}^2 + \alpha_1\varepsilon_{t-1}^2 + \ldots + \alpha_Q\varepsilon_{t-Q}^2.$$

In lag operator notation, the model is

$$\left(1 - \gamma_1 L - \ldots - \gamma_P L^P\right)\sigma_t^2 = \kappa + \left(\alpha_1 L + \ldots + \alpha_Q L^Q\right)\varepsilon_t^2.$$

The table shows how the variables correspond to the properties of the `garch` model object.

Variable	Description	Property
μ	Innovation mean model constant offset	`'Offset'`
$\kappa > 0$	Conditional variance model constant	`'Constant'`
$\gamma_i \geq 0$	GARCH component coefficients	`'GARCH'`
$\alpha_j \geq 0$	ARCH component coefficients	`'ARCH'`
z_t	Series of independent random variables with mean 0 and variance 1	`'Distribution'`

For stationarity and positivity, GARCH models use these constraints:

- $\kappa > 0$
- $\gamma_i \geq 0,\ \alpha_j \geq 0$

- $\sum_{i=1}^{P}\gamma_i + \sum_{j=1}^{Q}\alpha_j < 1$

Engle's original ARCH(Q) model is equivalent to a GARCH(0,Q) specification.

GARCH models are appropriate when positive and negative shocks of equal magnitude contribute equally to volatility [1].

- "GARCH Model" on page 6-3

References

[1] Tsay, R. S. *Analysis of Financial Time Series*. 3rd ed. Hoboken, NJ: John Wiley & Sons, Inc., 2010.

See Also

`estimate` | `filter` | `forecast` | `infer` | `print` | `simulate`

Introduced in R2012a

Using garch Objects

GARCH conditional variance time series model

Description

A `garch` model object specifies the functional form and stores the parameter values of a generalized autoregressive conditional heteroscedastic (GARCH) model. "GARCH Model" on page 9-537 attempt to address volatility clustering in an innovations process. Volatility clustering occurs when an innovations process does not exhibit significant autocorrelation, but the variance of the process changes with time. GARCH models are appropriate when positive and negative shocks of equal magnitude contribute equally to volatility [1].

The GARCH(P,Q) conditional variance model includes:

- P past conditional variances that compose the GARCH component polynomial
- Q past squared innovations that compose the ARCH component polynomial

To create a `garch` model object, use `garch`. Specify only the GARCH and ARCH polynomial degrees P and Q, respectively, using the shorthand syntax `garch(P,Q)`. Then, pass the model and time series data to `estimate` to fit the model to the data. Or, specify the values of some parameters, and then estimate others.

Use a completely specified model (i.e., all parameter values of the model are known) to:

- Simulate conditional variances or responses using `simulate`
- Forecast conditional variances using `forecast`

Examples

Create GARCH Model

Create a `garch` model using name-value pair arguments.

Specify a GARCH(1,1) model. By default, the conditional mean model offset is zero. Specify that the offset is `NaN`.

```
Mdl = garch('GARCHLags',1,'ARCHLags',1,'Offset',NaN)
```

```
Mdl =

    GARCH(1,1) Conditional Variance Model with Offset:
    --------------------------------------------------
    Distribution: Name = 'Gaussian'
               P: 1
               Q: 1
        Constant: NaN
           GARCH: {NaN} at Lags [1]
            ARCH: {NaN} at Lags [1]
          Offset: NaN
```

`Mdl` is a `garch` model object. The software sets all parameters (the properties of the model object) to `NaN`, except `P`, `Q`, and `Distribution`.

Since `Mdl` contains `NaN` values, `Mdl` is only appropriate for estimation only. Pass `Mdl` and time-series data to `estimate`. For a continuation of this example, see "Estimate GARCH Model".

Create GARCH Model Using Shorthand Syntax

Create a `garch` model using the shorthand notation `garch(P,Q)`, where `P` is the degree of the GARCH polynomial and `Q` is the degree of the ARCH polynomial.

Create a GARCH(3,2) model.

```
Mdl = garch(3,2)
```

```
Mdl =

    GARCH(3,2) Conditional Variance Model:
    --------------------------------------
    Distribution: Name = 'Gaussian'
               P: 3
               Q: 2
        Constant: NaN
           GARCH: {NaN NaN NaN} at Lags [1 2 3]
            ARCH: {NaN NaN} at Lags [1 2]
```

`Mdl` is a `garch` model object. All properties of `Mdl`, except `P`, `Q`, and `Distribution`, are `NaN` values. By default, the software:

- Includes a conditional variance model constant
- Excludes a conditional mean model offset (i.e., the offset is `0`)
- Includes all lag terms in the ARCH and GARCH lag-operator polynomials up to lags `Q` and `P`, respectively

`Mdl` specifies only the functional form of a GARCH model. Because it contains unknown parameter values, you can pass `Mdl` and the time-series data to `estimate` to estimate the parameters.

Access GARCH Model Properties

Access the properties of a `garch` model object using dot notation.

Create a `garch` model object.

```
Mdl = garch(3,2)
```

```
Mdl =

    GARCH(3,2) Conditional Variance Model:
    --------------------------------------
    Distribution: Name = 'Gaussian'
               P: 3
               Q: 2
        Constant: NaN
           GARCH: {NaN NaN NaN} at Lags [1 2 3]
            ARCH: {NaN NaN} at Lags [1 2]
```

Remove the second GARCH term from the model. That is, specify that the GARCH coefficient of the second lagged conditional variance is `0`.

```
Mdl.GARCH{2} = 0
```

```
Mdl =

    GARCH(3,2) Conditional Variance Model:
    --------------------------------------
    Distribution: Name = 'Gaussian'
               P: 3
               Q: 2
```

```
        Constant: NaN
           GARCH: {NaN NaN} at Lags [1 3]
            ARCH: {NaN NaN} at Lags [1 2]
```

The GARCH polynomial has two unknown parameters corresponding to lags 1 and 3.

Display the distribution of the disturbances.

```
Mdl.Distribution
```

```
ans =

    Name: 'Gaussian'
```

The disturbances are Gaussian with mean 0 and variance 1.

Specify that the underlying I.I.D. disturbances have a t distribution with five degrees of freedom.

```
Mdl.Distribution = struct('Name','t','DoF',5)
```

```
Mdl =

    GARCH(3,2) Conditional Variance Model:
    --------------------------------------
    Distribution: Name = 't', DoF = 5
               P: 3
               Q: 2
        Constant: NaN
           GARCH: {NaN NaN} at Lags [1 3]
            ARCH: {NaN NaN} at Lags [1 2]
```

Specify that the ARCH coefficients are 0.2 for the first lag and 0.1 for the second lag.

```
Mdl.ARCH = {0.2 0.1}
```

```
Mdl =

    GARCH(3,2) Conditional Variance Model:
    --------------------------------------
```

```
    Distribution: Name = 't', DoF = 5
               P: 3
               Q: 2
        Constant: NaN
           GARCH: {NaN NaN} at Lags [1 3]
            ARCH: {0.2 0.1} at Lags [1 2]
```

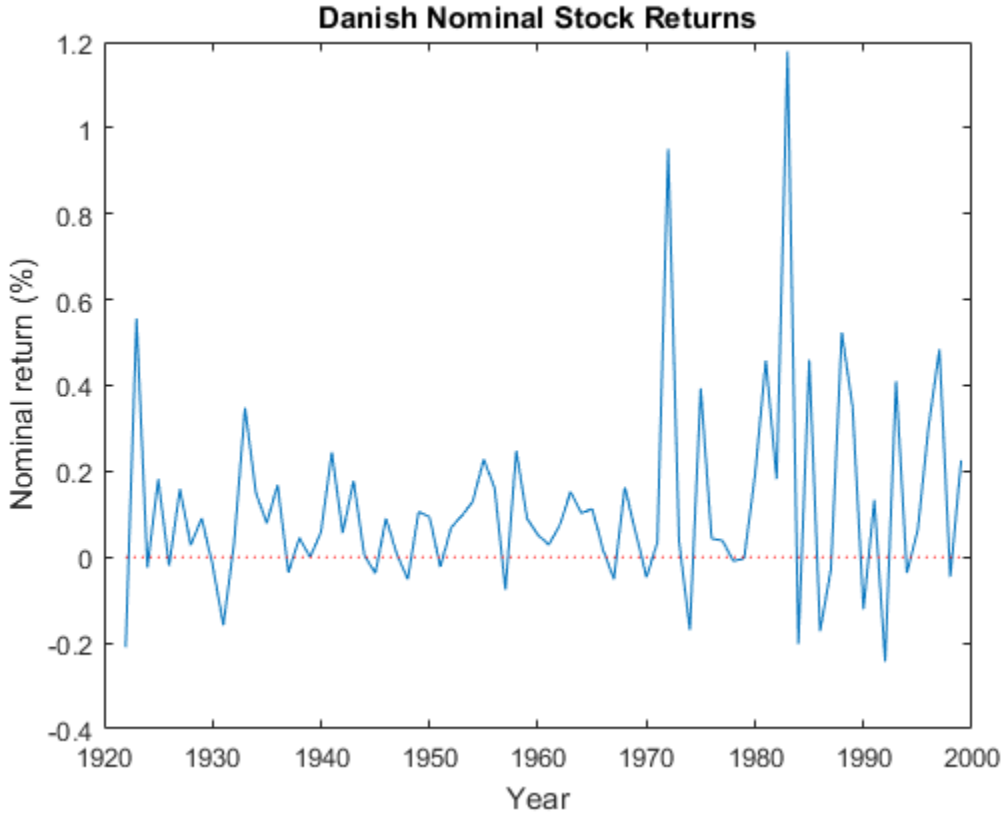
To estimate the remaining parameters, you can pass `Mdl` and your data to `estimate` and use the specified parameters as equality constraints. Or, you can specify the rest of the parameter values, and then simulate or forecast conditional variances from the GARCH model by passing the fully specified model to `simulate` or `forecast`, respectively.

Estimate GARCH Model

Fit a GARCH model to an annual time series of Danish nominal stock returns from 1922-1999. The example follows from “Create GARCH Model”.

Load the `Data_Danish` data set. Plot the nominal returns (`nr`).

```
load Data_Danish;
nr = DataTable.RN;

figure;
plot(dates,nr);
hold on;
plot([dates(1) dates(end)],[0 0],'r:'); % Plot y = 0
hold off;
title('Danish Nominal Stock Returns');
ylabel('Nominal return (%)');
xlabel('Year');
```



The nominal return series seems to have a nonzero conditional mean offset and seems to exhibit volatility clustering. That is, the variability is smaller for earlier years than it is for later years. For this example, assume that a GARCH(1,1) model is appropriate for this series.

Create a GARCH(1,1) model. The conditional mean offset is zero by default. To estimate the offset, specify that it is `NaN`.

```
Mdl = garch('GARCHLags',1,'ARCHLags',1,'Offset',NaN);
```

Fit the GARCH(1,1) model to the data.

```
EstMdl = estimate(Mdl,nr);
```

```
GARCH(1,1) Conditional Variance Model:
----------------------------------------
Conditional Probability Distribution: Gaussian

                           Standard          t
 Parameter       Value       Error       Statistic
-----------   -----------   ------------   -----------
 Constant     0.00444761    0.00781404       0.569182
 GARCH{1}       0.849317      0.264946        3.20563
  ARCH{1}      0.0732495      0.149532       0.489857
   Offset       0.112266     0.0392138        2.86293
```

`EstMdl` is a fully specified `garch` model object. That is, it does not contain `NaN` values. You can assess the adequacy of the model by generating residuals using `infer`, and then analyzing them.

To simulate conditional variances or responses, pass `EstMdl` to `simulate`. See "Simulate GARCH Model Observations and Conditional Variances".

To forecast innovations, pass `EstMdl` to `forecast`. See "Forecast GARCH Model Conditional Variances".

Simulate GARCH Model Observations and Conditional Variances

Simulate conditional variance or response paths from a fully specified `garch` model object. That is, simulate from an estimated `garch` model or a known `garch` model in which you specify all parameter values. This example follows from "Estimate GARCH Model".

Load the `Data_Danish` data set.

```
load Data_Danish;
nr = DataTable.RN;
```

Create a GARCH(1,1) model with an unknown conditional mean offset. Fit the model to the annual nominal return series.

```
Mdl = garch('GARCHLags',1,'ARCHLags',1,'Offset',NaN);
EstMdl = estimate(Mdl,nr);
```

```
GARCH(1,1) Conditional Variance Model:
```

```
    ----------------------------------------
    Conditional Probability Distribution: Gaussian

                               Standard          t
     Parameter       Value       Error       Statistic
    -----------   -----------   ------------   -----------
     Constant     0.00444761    0.00781404       0.569182
     GARCH{1}       0.849317      0.264946        3.20563
      ARCH{1}      0.0732495      0.149532       0.489857
       Offset       0.112266     0.0392138        2.86293
```

Simulate 100 paths of conditional variances and responses for each period from the estimated GARCH model.

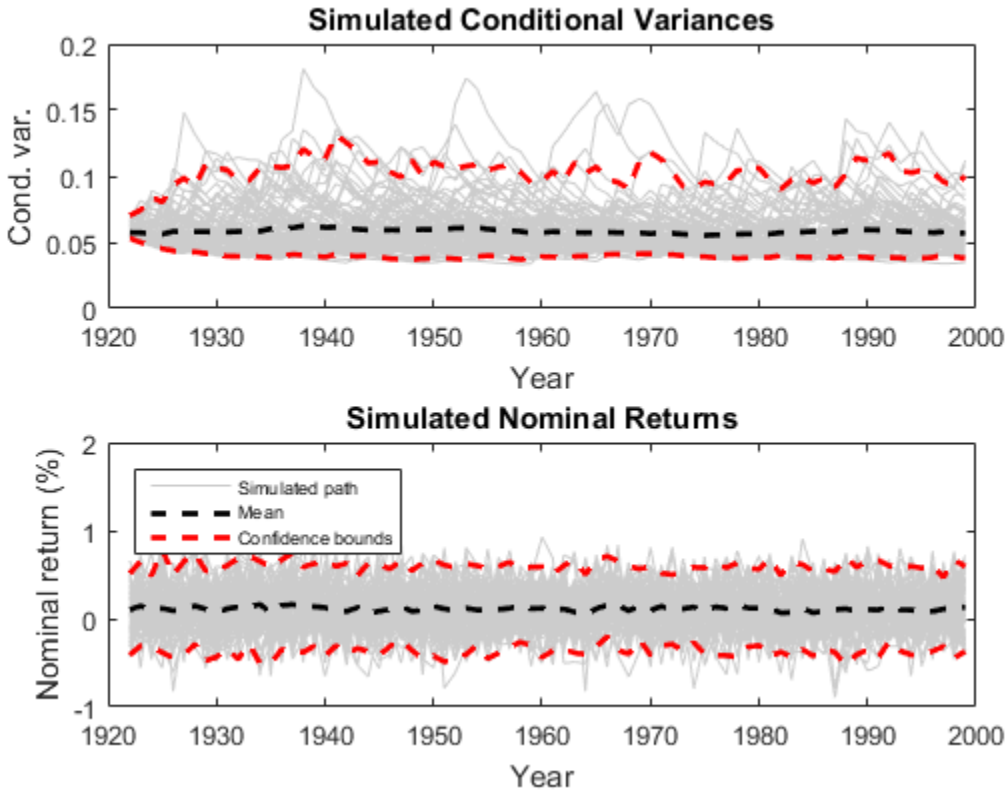
```
numObs = numel(nr); % Sample size (T)
numPaths = 100;     % Number of paths to simulate
rng(1);             % For reproducibility
[VSim,YSim] = simulate(EstMdl,numObs,'NumPaths',numPaths);
```

`VSim` and `YSim` are `T`-by-`numPaths` matrices. Rows correspond to a sample period, and columns correspond to a simulated path.

Plot the average and the 97.5% and 2.5% percentiles of the simulated paths. Compare the simulation statistics to the original data.

```
VSimBar = mean(VSim,2);
VSimCI = quantile(VSim,[0.025 0.975],2);
YSimBar = mean(YSim,2);
YSimCI = quantile(YSim,[0.025 0.975],2);

figure;
subplot(2,1,1);
h1 = plot(dates,VSim,'Color',0.8*ones(1,3));
hold on;
h2 = plot(dates,VSimBar,'k--','LineWidth',2);
h3 = plot(dates,VSimCI,'r--','LineWidth',2);
hold off;
title('Simulated Conditional Variances');
ylabel('Cond. var.');
xlabel('Year');

subplot(2,1,2);
h1 = plot(dates,YSim,'Color',0.8*ones(1,3));
hold on;
```

```
h2 = plot(dates,YSimBar,'k--','LineWidth',2);
h3 = plot(dates,YSimCI,'r--','LineWidth',2);
hold off;
title('Simulated Nominal Returns');
ylabel('Nominal return (%)');
xlabel('Year');
legend([h1(1) h2 h3(1)],{'Simulated path' 'Mean' 'Confidence bounds'},...
    'FontSize',7,'Location','NorthWest');
```



Forecast GARCH Model Conditional Variances

Forecast conditional variances from a fully specified `garch` model object. That is, forecast from an estimated `garch` model or a known `garch` model in which you specify all parameter values. The example follows from “Estimate GARCH Model”.

Load the `Data_Danish` data set.

```
load Data_Danish;
nr = DataTable.RN;
```

Create a GARCH(1,1) model with an unknown conditional mean offset, and fit the model to the annual, nominal return series.

```
Mdl = garch('GARCHLags',1,'ARCHLags',1,'Offset',NaN);
EstMdl = estimate(Mdl,nr);
```

```
    GARCH(1,1) Conditional Variance Model:
    ----------------------------------------
    Conditional Probability Distribution: Gaussian

                               Standard          t
     Parameter       Value       Error       Statistic
    -----------   -----------   ------------   -----------
     Constant     0.00444761    0.00781404       0.569182
     GARCH{1}       0.849317      0.264946        3.20563
      ARCH{1}      0.0732495      0.149532       0.489857
       Offset       0.112266     0.0392138        2.86293
```

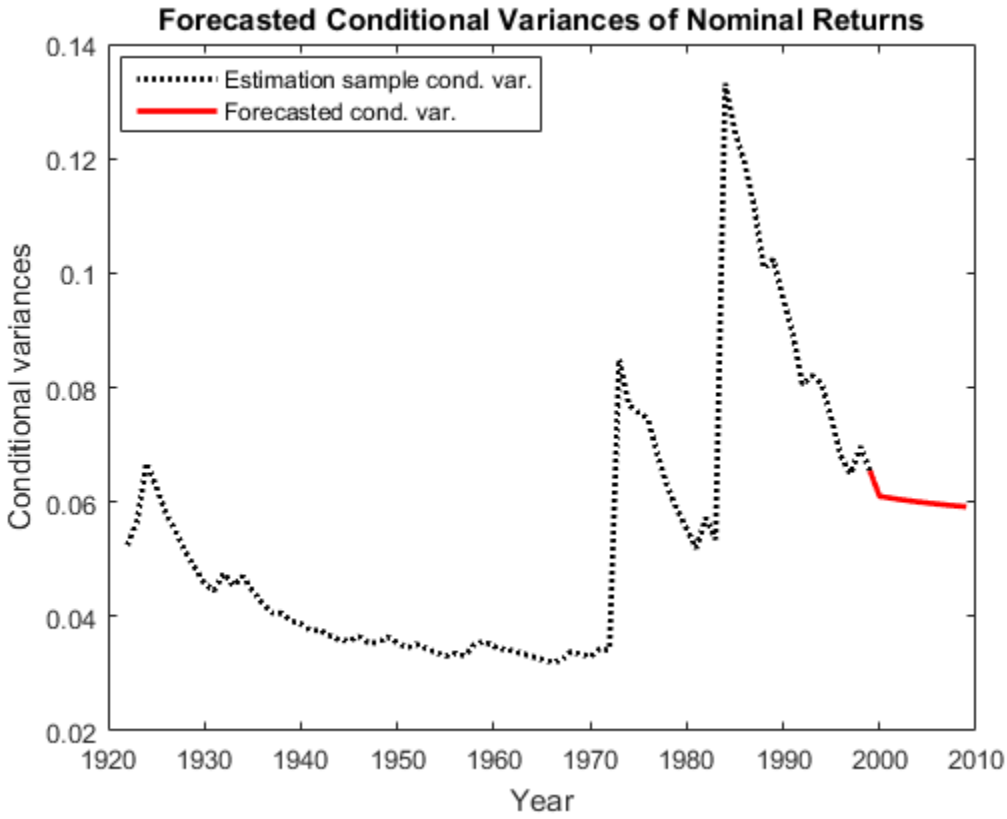
Forecast the conditional variance of the nominal return series 10 years into the future using the estimated GARCH model. Specify the entire returns series as presample observations. The software infers presample conditional variances using the presample observations and the model.

```
numPeriods = 10;
vF = forecast(EstMdl,numPeriods,'Y0',nr);
```

Plot the forecasted conditional variances of the nominal returns. Compare the forecasts to the observed conditional variances.

```
v = infer(EstMdl,nr);

figure;
plot(dates,v,'k:','LineWidth',2);
hold on;
plot(dates(end):dates(end) + 10,[v(end);vF],'r','LineWidth',2);
title('Forecasted Conditional Variances of Nominal Returns');
ylabel('Conditional variances');
xlabel('Year');
```

```
legend({'Estimation sample cond. var.','Forecasted cond. var.'},...
    'Location','Best');
```



- “Specify GARCH Models Using garch” on page 6-8
- “Modify Properties of Conditional Variance Models” on page 6-42
- “Specify Conditional Mean and Variance Models” on page 5-79
- “Infer Conditional Variances and Residuals” on page 6-77
- “Compare Conditional Variance Models Using Information Criteria” on page 6-87
- “Simulate GARCH Models” on page 6-97
- “Forecast a Conditional Variance Model” on page 6-126

Properties

Conditional Variance Model Properties	Specify conditional variance model functional form and parameter values

Object Functions

estimate	Fit conditional variance model to data
filter	Filter disturbances through conditional variance model
forecast	Forecast conditional variances from conditional variance models
infer	Infer conditional variances of conditional variance models
print	Display parameter estimation results for conditional variance models
simulate	Monte Carlo simulation of conditional variance models

Create Object

Create `garch` models using `garch`.

You can specify a `garch` model as part of a composition of conditional mean and variance models. For details, see arima.

See Also

arima | `egarch` | `gjr`

More About

- "Conditional Variance Models" on page 6-2
- "GARCH Model" on page 6-3

Introduced in R2012a

garchar

Convert ARMA model to AR model

Compatibility

`garchar` has been removed. Use `arma2ar` instead.

Syntax

```
InfiniteAR = garchar(AR,MA,NumLags)
```

Description

`InfiniteAR = garchar(AR,MA,NumLags)` computes the coefficients of an infinite-order AR model, using the coefficients of the equivalent univariate, stationary, invertible, finite-order ARMA(R,M) model as input. `garchar` truncates the infinite-order AR coefficients to accommodate a user-specified number of lagged AR coefficients.

Input Arguments

`AR`	*R*-element vector of autoregressive coefficients associated with the lagged observations of a univariate return series modeled as a finite-order, stationary, invertible ARMA(R,M) model.
`MA`	*M*-element vector of moving-average coefficients associated with the lagged innovations of a finite-order, stationary, invertible univariate ARMA(R,M) model.
`NumLags`	(optional) Number of lagged AR coefficients that `garchar` includes in the approximation of the infinite-order AR representation. `NumLags` is an integer scalar and determines the length of the infinite-order AR output vector. If `NumLags = []` or is unspecified, the default is `10`.

Output Arguments

`InfiniteAR`	Vector of coefficients of the infinite-order AR representation associated with the finite-order ARMA model specified by the `AR` and `MA` input vectors. `InfiniteAR` is a vector of length `NumLags`. The *j*th element of `InfiniteAR` is the coefficient of the *j*th lag of the input series in an infinite-order AR representation. Box, Jenkins, and Reinsel refer to the infinite-order AR coefficients as "π weights."

In the following ARMA(R,M) model, $\{y_t\}$ is the return series of interest and $\{\varepsilon_t\}$ the innovations noise process.

$$y_t = \sum_{i=1}^{R} \phi_i y_{t-1} + \varepsilon_t \sum_{j=1}^{M} \theta_j \varepsilon_{j-1}$$

If you write this model equation as

$$y_t = \phi_1 y_{t-1} + \ldots + \phi_R y_{t-R} + \varepsilon_t + \theta_1 \varepsilon_{t-1} + \ldots + \theta_M \varepsilon_{t-M}$$

you can specify the `garchar` input coefficient vectors, `AR` and `MA`, as you read them from the model. In general, the *j*th elements of `AR` and `MA` are the coefficients of the *j*th lag of the return series and innovations processes y_{t-j} and ε_{t-j}, respectively. `garchar` assumes that the current-time-index coefficients of y_t and ε_t are `1` and are *not* part of `AR` and `MA`.

In theory, you can use the π weights returned in `InfiniteAR` to approximatey_t as a pure AR process.

$$y_t = \sum_{i=1}^{\infty} \pi_i y_{t-i} + \varepsilon_t$$

In this equation, the *j*th element of the truncated infinite-order autoregressive output vector,π_j or `InfiniteAR(j)`, is consistently the coefficient of the *j*th lag of the observed return series, y_{t-j}. See Box, Jenkins, and Reinsel [15], Section 4.2.3, pages 106-109.

References

[1] Box, G. E. P., G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis: Forecasting and Control*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

See Also

`garchma`

Introduced before R2006a

garchma

Convert ARMA model to MA model

Compatibility

`garchma` has been removed. Use `arma2ma` instead.

Syntax

```
InfiniteMA = garchma(AR,MA,NumLags)
```

Description

`InfiniteMA = garchma(AR,MA,NumLags)` computes the coefficients of an infinite-order MA model, using the coefficients of the equivalent univariate, stationary, invertible, finite-order ARMA(R,M) model as input. `garchma` truncates the infinite-order MA coefficients to accommodate the number of lagged MA coefficients you specify in `NumLags`.

This function is useful for calculating the standard errors of minimum mean square error forecasts of univariate ARMA models.

Arguments

`AR`	*R*-element vector of autoregressive coefficients associated with the lagged observations of a univariate return series modeled as a finite-order, stationary, invertible ARMA(R,M) model.
`MA`	*M*-element vector of moving-average coefficients associated with the lagged innovations of a finite-order, stationary, invertible, univariate ARMA(R,M) model.
`NumLags`	(optional) Number of lagged MA coefficients that `garchma` includes in the approximation of the infinite-order MA representation. `NumLags` is an integer scalar and determines the length of the infinite-order MA output vector. If `NumLags = []` or is unspecified, the default is `10`.

Output Arguments

`InfiniteMA`	Vector of coefficients of the infinite-order MA representation associated with the finite-order ARMA model specified by `AR` and `MA`. `InfiniteMA` is a vector of length `NumLags`. The *j*th element of `InfiniteMA` is the coefficient of the *j*th lag of the innovations noise sequence in an infinite-order MA representation. Box, Jenkins, and Reinsel refer to the infinite-order MA coefficients as the "ψ weights."

In the following ARMA(R,M) model,$\{y_t\}$ is the return series of interest and $\{\varepsilon_t\}$ the innovations noise process.

$$y_t = \sum_{i=1}^{R} \phi_i y_{t-1} + \varepsilon_t \sum_{j=1}^{M} \theta_j \varepsilon_{j-1}$$

If you write this model equation as

$$y_t = \phi_1 y_{t-1} + \ldots + \phi_R y_{t-R} + \varepsilon_t + \theta_1 \varepsilon_{t-1} + \ldots + \theta_M \varepsilon_{t-M}$$

you can specify the `garchma` input coefficient vectors, `AR` and `MA`, as you read them from the model. In general, the *j*th elements of `AR` and `MA` are the coefficients of the *j*th lag of the return series and innovations processes y_{t-j} and ε_{t-j}, respectively. `garchma` assumes that the current-time-index coefficients of y_t and ε_t are `1` and are not part of `AR` and `MA`.

In theory, you can use the ψ weights returned in `InfiniteMA` to approximate y_t as a pure MA process.

$$y_t = \varepsilon_t + \sum_{i=1}^{\infty} \psi_i \varepsilon_{t-i}$$

The *j*th element of the truncated infinite-order moving-average output vector, ψ_j or `InfiniteMA(j)`, is consistently the coefficient of the *j*th lag of the innovations process, ε_{t-j}, in this equation. See Box, Jenkins, and Reinsel [15], Section 5.2.2, pages 139-141.

References

[1] Box, G. E. P., G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis: Forecasting and Control*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

See Also

`garchar`

Introduced before R2006a

gjr

Create GJR conditional variance model object

Create a `gjr` model object to represent a Glosten, Jagannathan, and Runkle (GJR) model. The GJR(P,Q) conditional variance model includes P past conditional variances composing the GARCH polynomial, and Q past squared innovations composing the ARCH and leverage polynomials.

Use `gjr` to create a model with known or unknown coefficients, and then estimate any unknown coefficients from data using `estimate`. You can also simulate or forecast conditional variances from fully specified models using `simulate` or `forecast`, respectively.

For more information about `gjr` model objects, see Using gjr Objects.

Syntax

```
Mdl = gjr
Mdl = gjr(P,Q)
Mdl = gjr(Name,Value)
```

Description

`Mdl = gjr` creates a zero-degree conditional variance GJR model object.

`Mdl = gjr(P,Q)` creates a GJR model with GARCH polynomial degree `P`, and ARCH and leverage polynomials having degree `Q`.

`Mdl = gjr(Name,Value)` creates a GJR model with additional options specified by one or more `Name,Value` pair arguments. For example, you can specify a conditional variance model constant, the number of ARCH polynomial lags, and the innovation distribution.

Examples

Create Default GJR Model

Create a default `gjr` model object and specify its parameter values using dot notation.

Create a GJR(0,0) model.

```
Mdl = gjr
```

```
Mdl =

    GJR(0,0) Conditional Variance Model:
    ------------------------------------
    Distribution: Name = 'Gaussian'
               P: 0
               Q: 0
        Constant: NaN
           GARCH: {}
            ARCH: {}
        Leverage: {}
```

`Mdl` is a `gjr` model object. It contains an unknown constant, its offset is `0`, and the innovation distribution is `'Gaussian'`. The model does not have GARCH, ARCH, or leverage polynomials.

Specify two unknown ARCH and leverage coefficients for lags one and two using dot notation.

```
Mdl.ARCH = {NaN NaN};
Mdl.Leverage = {NaN NaN};
Mdl
```

```
Mdl =

    GJR(0,2) Conditional Variance Model:
    ------------------------------------
    Distribution: Name = 'Gaussian'
               P: 0
               Q: 2
        Constant: NaN
           GARCH: {}
```

```
           ARCH: {NaN NaN} at Lags [1 2]
      Leverage: {NaN NaN} at Lags [1 2]
```

The `Q`, `ARCH`, and `Leverage` properties update to `2`, `{NaN NaN}`, and `{NaN NaN}`, respectively. The two ARCH and leverage coefficients are associated with lags 1 and 2.

Create GJR Model Using Shorthand Syntax

Create a `gjr` model object using the shorthand notation `gjr(P,Q)`, where `P` is the degree of the GARCH polynomial and `Q` is the degree of the ARCH and leverage polynomials.

Create an GJR(3,2) model.

```
Mdl = gjr(3,2)
```

```
Mdl =

    GJR(3,2) Conditional Variance Model:
    ------------------------------------
    Distribution: Name = 'Gaussian'
               P: 3
               Q: 2
        Constant: NaN
           GARCH: {NaN NaN NaN} at Lags [1 2 3]
            ARCH: {NaN NaN} at Lags [1 2]
        Leverage: {NaN NaN} at Lags [1 2]
```

`Mdl` is a `gjr` model object. All properties of `Mdl`, except `P`, `Q`, and `Distribution`, are `NaN` values. By default, the software:

- Includes a conditional variance model constant
- Excludes a conditional mean model offset (i.e., the offset is `0`)
- Includes all lag terms in the GARCH polynomial up to lags `P`
- Includes all lag terms in the ARCH and leverage polynomials up to lag `Q`

`Mdl` specifies only the functional form of a GJR model. Because it contains unknown parameter values, you can pass `Mdl` and time-series data to `estimate` to estimate the parameters.

Create GJR Model

Create a `gjr` model using name-value pair arguments.

Specify a GJR(1,1) model.

```
Mdl = gjr('GARCHLags',1,'ARCHLags',1,'LeverageLags',1)
```

```
Mdl =

    GJR(1,1) Conditional Variance Model:
    ------------------------------------
    Distribution: Name = 'Gaussian'
               P: 1
               Q: 1
        Constant: NaN
           GARCH: {NaN} at Lags [1]
            ARCH: {NaN} at Lags [1]
        Leverage: {NaN} at Lags [1]
```

`Mdl` is a `gjr` model object. The software sets all parameters to `NaN`, except `P`, `Q`, `Distribution`, and `Offset` (which is `0` by default).

Since `Mdl` contains `NaN` values, `Mdl` is only appropriate for estimation only. Pass `Mdl` and time-series data to `estimate`. For a continuation of this example, see "Estimate GJR Model".

Create GJR Model with Known Coefficients

Create a GJR(1,1) model with mean offset

$$y_t = 0.5 + \varepsilon_t,$$

where $\varepsilon_t = \sigma_t z_t$,

$$\sigma_t^2 = 0.0001 + 0.35\sigma_{t-1}^2 + 0.1\varepsilon_{t-1}^2 + 0.03\varepsilon_{t-1}^2 I(\varepsilon_{t-1} < 0) + 0.01\varepsilon_{t-3}^2 I(\varepsilon_{t-3} < 0),$$

and z_t is an independent and identically distributed standard Gaussian process.

```
Mdl = gjr('Constant',0.0001,'GARCH',0.35,...
    'ARCH',0.1,'Offset',0.5,'Leverage',{0.03 0 0.01})
```

```
Mdl =

    GJR(1,3) Conditional Variance Model with Offset:
```

```
    ----------------------------------------------
    Distribution: Name = 'Gaussian'
               P: 1
               Q: 3
        Constant: 0.0001
           GARCH: {0.35} at Lags [1]
            ARCH: {0.1} at Lags [1]
        Leverage: {0.03 0.01} at Lags [1 3]
          Offset: 0.5
```

`gjr` assigns default values to any properties you do not specify with name-value pair arguments. An alternative way to specify the leverage component is `'Leverage',{0.03 0.01},'LeverageLags',[1 3]`.

- "Specify GJR Models Using gjr" on page 6-31
- "Modify Properties of Conditional Variance Models" on page 6-42
- "Specify the Conditional Variance Model Innovation Distribution" on page 6-48
- "Specify Conditional Mean and Variance Models" on page 5-79
- "Specify Conditional Variance Model For Exchange Rates" on page 6-53

Input Arguments

`P` — Number of past consecutive conditional variance terms

nonnegative integer

Number of past consecutive conditional variance terms to include in the GARCH polynomial, specified as a nonnegative integer. That is, `P` is the degree of the GARCH polynomial, where the polynomial includes each lag term from $t - 1$ to $t - P$. `P` also specifies the minimum number of presample conditional variances the software requires to initiate the model.

You can specify `P` using the `gjr(P,Q)` shorthand syntax only. You cannot specify `P` in conjunction with `Name,Value` pair arguments.

If $P > 0$, then you must specify `Q` as a positive integer.

Example: `gjr(3,2)`

Data Types: `double`

Q — Number of past consecutive squared innovation terms

nonnegative integer

Number of past consecutive squared innovation terms to include in the ARCH and leverage polynomials, specified as a nonnegative integer. That is, Q is the degrees of the ARCH and leverage polynomials, where each polynomial includes each lag term from $t - 1$ to $t - Q$. Q also specifies the minimum number of presample innovations the software requires to initiate the model.

You can specify this property when using the `gjr(P,Q)` shorthand syntax only. You cannot specify Q in conjunction with `Name,Value` pair arguments.

If $P > 0$, then you must specify Q as a positive integer.

Example: `gjr(3,2)`

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example:
`'Constant',0.5,'ARCHLags',2,'Distribution',struct('Name','t','DoF',5)`
specifies a conditional variance model constant of 0.5, two squared innovation terms at lags 1 and 2 of the ARCH polynomial, and a t distribution with 5 degrees of freedom for the innovations.

'Constant' — Conditional variance model constant

`NaN` (default) | positive scalar

Conditional variance model constant, specified as the comma-separated pair consisting of `'Constant'` and a positive scalar.

Example: `'Constant',0.5`

Data Types: `double`

'GARCH' — Coefficients corresponding to past conditional variance terms

cell vector of `NaN`s (default) | cell vector of nonnegative scalars

Coefficients corresponding to the past conditional variance terms that compose the GARCH polynomial, specified as the comma-separated pair consisting of `'GARCH'` and a cell vector of nonnegative scalars.

If you specify `GARCHLags`, then `GARCH` is an equivalent-length cell vector of coefficients associated with the lags in `GARCHLags`. Otherwise, `GARCH` is a P-element cell vector of coefficients corresponding to lags 1, 2,..., P.

The coefficients must compose a stationary model. For details, see "GJR Model" on page 9-568.

By default, `GARCH` is a cell vector of `NaN`s of length P (the degree of the GARCH polynomial) or `numel(GARCHLags)`.

Example: `'GARCH',{0.1 0 0 0.02}`

Data Types: `cell`

`'ARCH'` — Coefficients corresponding to past squared innovation terms

cell vector of `NaN`s (default) | cell vector of nonnegative scalars

Coefficients corresponding to the past squared innovation terms that compose the ARCH polynomial, specified as the comma-separated pair consisting of `'ARCH'` and a cell vector of nonnegative scalars.

If you specify `ARCHLags`, then `ARCH` is an equivalent-length cell vector of coefficients associated with the lags in `ARCH`. Otherwise, `ARCH` is a Q-element cell vector of coefficients corresponding to lags 1 through the number of elements in `ARCH`.

The coefficients must compose a stationary model. For details, see "GJR Model" on page 9-568.

By default, `ARCH` is a cell vector of `NaN`s of length Q (the degree of the ARCH polynomial) or `numel(ARCHLags)`.

Example: `'ARCH',{0.5 0 0.2}`

Data Types: `cell`

`'Leverage'` — Coefficients corresponding to past sign-weighted, squared innovation terms

cell vector of `NaN`s (default) | cell vector of scalars

Coefficients corresponding to the past sign-weighted, squared innovation terms that compose the leverage polynomial, specified as the comma-separated pair consisting of `'Leverage'` and a cell vector of scalars.

If you specify `LeverageLags`, then `Leverage` is an equivalent-length cell vector of coefficients associated with the lags in `ARCHLags`. Otherwise, `Leverage` is a cell vector of coefficients corresponding to lags 1 through the number of elements in `Leverage`.

The coefficients must compose a stationary model. For details, see "GJR Model" on page 9-568.

By default, `Leverage` is a cell vector of `NaN`s with the same length as the leverage polynomial degree or `numel(LeverageLags)`.

Example: `'Leverage',{-0.1 0 0 0.03}`

`'Offset'` — Innovation mean model offset

`0` (default) | scalar

Innovation mean model offset or additive constant, specified as the comma-separated pair consisting of `'Offset'` and a scalar.

Example: `'Offset',0.1`

Data Types: `double`

`'GARCHLags'` — Lags associated with GARCH polynomial coefficients

vector of positive integers

Lags associated with the GARCH polynomial coefficients, specified as the comma-separated pair consisting of `'GARCHLags'` and a vector of positive integers. The maximum value of `GARCHLags` determines *P*, the GARCH polynomial degree.

If you specify `GARCH`, then `GARCHLags` is an equivalent-length vector of positive integers specifying the lags of the corresponding coefficients in `GARCH`. Otherwise, `GARCHLags` indicates the lags of unknown coefficients in the GARCH polynomial.

By default, `GARCHLags` is a vector containing the integers 1 through *P*.

Example: `'GARCHLags',[1 2 4 3]`

Data Types: `double`

`'ARCHLags'` — Lags associated with ARCH polynomial coefficients

vector of positive integers

Lags associated with the ARCH polynomial coefficients, specified as the comma-separated pair consisting of `'ARCHLags'` and a vector of positive integers. The maximum value of `ARCHLags` determines the ARCH polynomial degree.

If you specify ARCH, then ARCHLags is an equivalent-length vector of positive integers specifying the lags of the corresponding coefficients in ARCH. Otherwise, ARCHLags indicates the lags of unknown coefficients in the ARCH polynomial.

By default, ARCHLags is a vector containing the integers 1 through the ARCH polynomial degree.

Example: 'ARCHLags',[3 1 2]

Data Types: double

'LeverageLags' — Lags associated with leverage polynomial coefficients
vector of positive integers

Lags associated with the leverage polynomial coefficients, specified as the comma-separated pair consisting of 'LeverageLags' and a vector of positive integers. The maximum value of LeverageLags determines the leverage polynomial degree.

If you specify Leverage, then LeverageLags is an equivalent-length vector of positive integers specifying the lags of the corresponding coefficients in LeverageLags. Otherwise, LeverageLags indicates the lags of unknown coefficients in the leverage polynomial.

By default, LeverageLags is a vector containing the integers 1 through the leverage polynomial degree.

Example: 'LeverageLags',1:4

Data Types: double

'Distribution' — Conditional probability distribution of innovation process
'Gaussian' (default) | string | structure array

Conditional probability distribution of the innovation process, specified as the comma-separated pair consisting of 'Distribution' and a string or a structure array.

This table contains the available distributions.

Distribution	String	Structure Array
Gaussian	'Gaussian'	struct('Name','Gaussian')
t	't' By default, DoF is NaN.	struct('Name','t','DoF',DoF) DoF > 2 or DoF = NaN

Example: `'Distribution',struct('Name','t','DoF',10)`

Data Types: `char` | `struct`

Notes:

- All `GARCH`, `ARCH` and `Leverage` coefficients are subject to a near-zero tolerance exclusion test. That is, the software:

  1. Creates lag operator polynomials for each of the `GARCH`, `ARCH` and `Leverage` components.
  2. Compares each coefficient to the default lag operator zero tolerance, `1e-12`.
  3. Includes a coefficient in the model if its magnitude is greater than `1e-12`, and excludes the coefficient otherwise. In other words, the software considers excluded coefficients to be sufficiently close to zero.

  For details, see LagOp.
- The lengths of `ARCH` and `Leverage` might differ. The difference can occur because the software defines the property `Q` as the largest lag associated with nonzero `ARCH` and `Leverage` coefficients, or `max(ARCHLags,LeverageLags)`. Typically, the number and corresponding lags of nonzero `ARCH` and `Leverage` coefficients are equivalent, but this is not a requirement.

Output Arguments

`Mdl` — GJR model

`gjr` model object

GJR model, returned as a `gjr` model object.

For the property descriptions of `Mdl`, see Conditional Variance Model Properties.

If `Mdl` contains unknown parameters (indicated by `NaN`s), then you can specify them using dot notation. Alternatively, you can pass `Mdl` and time series data to `estimate` to obtain estimates.

If `Mdl` is fully specified, then you can simulate or forecast conditional variances using `simulate` or `forecast`, respectively.

More About

GJR Model

The *Glosten, Jagannathan, and Runkle (GJR) model* is a generalization of the GARCH model that is appropriate for modelling asymmetric volatility clustering [1]. Specifically, the model posits that the current conditional variance is the sum of these linear processes, with coefficients:

- Past conditional variances (the GARCH component or polynomial).
- Past squared innovations (the ARCH component or polynomial).
- Past squared, negative innovations (the leverage component or polynomial).

Consider the time series

$$y_t = \mu + \varepsilon_t,$$

where $\varepsilon_t = \sigma_t z_t$. The GJR($P$,$Q$) conditional variance process, σ_t^2, has the form

$$\sigma_t^2 = \kappa + \sum_{i=1}^{P} \gamma_i \sigma_{t-i}^2 + \sum_{j=1}^{Q} \alpha_j \varepsilon_{t-j}^2 + \sum_{j=1}^{Q} \xi_j I\left[\varepsilon_{t-j} < 0\right] \varepsilon_{t-j}^2.$$

The table shows how the variables correspond to the properties of the `garch` model object. In the table, $I[x < 0] = 1$, and 0 otherwise.

Variable	Description	Property
μ	Innovation mean model constant offset	`'Offset'`
$\kappa > 0$	Conditional variance model constant	`'Constant'`
γ_j	GARCH component coefficients	`'GARCH'`
α_j	ARCH component coefficients	`'ARCH'`
ξ_j	Leverage component coefficients	`'Leverage'`

Variable	Description	Property
z_t	Series of independent random variables with mean 0 and variance 1	`'Distribution'`

For stationarity and positivity, GJR models use these constraints:

- $\kappa > 0$
- $\gamma_i \geq 0,\ \alpha_j \geq 0$
- $\alpha_j + \xi_j \geq 0$
- $\sum_{i=1}^{P} \gamma_i + \sum_{j=1}^{Q} \alpha_j + \frac{1}{2}\sum_{j=1}^{Q} \xi_j < 1$

GJR models are appropriate when negative shocks of contribute more to volatility than positive shocks [2].

If all leverage coefficients are zero, then the GJR model reduces to the GARCH model. Because the GARCH model is nested in the GJR model, you can use likelihood ratio tests to compare a GARCH model fit against a GJR model fit.

- "Conditional Variance Models" on page 6-2
- "GJR Model" on page 6-6

References

[1] Glosten, L. R., R. Jagannathan, and D. E. Runkle. "On the Relation between the Expected Value and the Volatility of the Nominal Excess Return on Stocks." *The Journal of Finance*. Vol. 48, No. 5, 1993, pp. 1779–1801.

[2] Tsay, R. S. *Analysis of Financial Time Series*. 3rd ed. Hoboken, NJ: John Wiley & Sons, Inc., 2010.

See Also

`estimate` | `filter` | `forecast` | `infer` | `print` | `simulate`

Introduced in R2012a

Using gjr Objects

GJR conditional variance time series model

Description

A `gjr` model object specifies the functional form and stores the parameter values of a Glosten, Jagannathan, and Runkle (GJR) model [1], which is a generalized autoregressive conditional heteroscedastic (GARCH) model generalization. "GJR Model" on page 9-568 attempt to address volatility clustering in an innovations process. Volatility clustering occurs when an innovations process does not exhibit significant autocorrelation, but the variance of the process changes with time. GJR models are appropriate when negative shocks contribute more to volatility than positive shocks [2].

The GJR(P,Q) conditional variance model includes:

- P past conditional variances that compose the GARCH component polynomial
- Q past squared innovations that compose the ARCH and leverage component polynomials

To create a `gjr` model object, use `gjr`. Specify only the GARCH and ARCH (and leverage) polynomial degrees P and Q, respectively, using the shorthand syntax `gjr(P,Q)`. Then, pass the model and time series data to `estimate` to fit the model to the data. Or, specify the values of some parameters, and then estimate others.

Use a completely specified model (i.e., all parameter values of the model are known) to:

- Simulate conditional variances or responses using `simulate`
- Forecast conditional variances using `forecast`

Examples

Create GJR Model

Create a `gjr` model using name-value pair arguments.

Specify a GJR(1,1) model.

```
Mdl = gjr('GARCHLags',1,'ARCHLags',1,'LeverageLags',1)
```

```
Mdl =

    GJR(1,1) Conditional Variance Model:
    ------------------------------------
    Distribution: Name = 'Gaussian'
               P: 1
               Q: 1
        Constant: NaN
           GARCH: {NaN} at Lags [1]
            ARCH: {NaN} at Lags [1]
        Leverage: {NaN} at Lags [1]
```

`Mdl` is a `gjr` model object. The software sets all parameters to `NaN`, except `P`, `Q`, `Distribution`, and `Offset` (which is `0` by default).

Since `Mdl` contains `NaN` values, `Mdl` is only appropriate for estimation only. Pass `Mdl` and time-series data to `estimate`. For a continuation of this example, see "Estimate GJR Model".

Create GJR Model Using Shorthand Syntax

Create a `gjr` model object using the shorthand notation `gjr(P,Q)`, where `P` is the degree of the GARCH polynomial and `Q` is the degree of the ARCH and leverage polynomials.

Create an GJR(3,2) model.

```
Mdl = gjr(3,2)
```

```
Mdl =

    GJR(3,2) Conditional Variance Model:
    ------------------------------------
    Distribution: Name = 'Gaussian'
               P: 3
               Q: 2
        Constant: NaN
           GARCH: {NaN NaN NaN} at Lags [1 2 3]
            ARCH: {NaN NaN} at Lags [1 2]
        Leverage: {NaN NaN} at Lags [1 2]
```

`Mdl` is a `gjr` model object. All properties of `Mdl`, except `P`, `Q`, and `Distribution`, are `NaN` values. By default, the software:

- Includes a conditional variance model constant
- Excludes a conditional mean model offset (i.e., the offset is `0`)
- Includes all lag terms in the GARCH polynomial up to lags `P`
- Includes all lag terms in the ARCH and leverage polynomials up to lag `Q`

`Mdl` specifies only the functional form of a GJR model. Because it contains unknown parameter values, you can pass `Mdl` and time-series data to `estimate` to estimate the parameters.

Access GJR Model Properties

Access the properties of a `gjr` model object using dot notation.

Create a `gjr` model object.

```
Mdl = gjr(3,2)
```

```
Mdl =

    GJR(3,2) Conditional Variance Model:
    ------------------------------------
    Distribution: Name = 'Gaussian'
               P: 3
               Q: 2
        Constant: NaN
           GARCH: {NaN NaN NaN} at Lags [1 2 3]
            ARCH: {NaN NaN} at Lags [1 2]
        Leverage: {NaN NaN} at Lags [1 2]
```

Remove the second GARCH term from the model. That is, specify that the GARCH coefficient of the second lagged conditional variance is `0`.

```
Mdl.GARCH{2} = 0
```

```
Mdl =

    GJR(3,2) Conditional Variance Model:
    ------------------------------------
    Distribution: Name = 'Gaussian'
               P: 3
```

```
               Q: 2
        Constant: NaN
           GARCH: {NaN NaN} at Lags [1 3]
            ARCH: {NaN NaN} at Lags [1 2]
        Leverage: {NaN NaN} at Lags [1 2]
```

The GARCH polynomial has two unknown parameters corresponding to lags 1 and 3.

Display the distribution of the disturbances.

```
Mdl.Distribution
```

```
ans =

    Name: 'Gaussian'
```

The disturbances are Gaussian with mean 0 and variance 1.

Specify that the underlying disturbances have a t distribution with five degrees of freedom.

```
Mdl.Distribution = struct('Name','t','DoF',5)
```

```
Mdl =

    GJR(3,2) Conditional Variance Model:
    ------------------------------------
    Distribution: Name = 't', DoF = 5
               P: 3
               Q: 2
        Constant: NaN
           GARCH: {NaN NaN} at Lags [1 3]
            ARCH: {NaN NaN} at Lags [1 2]
        Leverage: {NaN NaN} at Lags [1 2]
```

Specify that the ARCH coefficients are 0.2 for the first lag and 0.1 for the second lag.

```
Mdl.ARCH = {0.2 0.1}
```

```
Mdl =
```

```
GJR(3,2) Conditional Variance Model:
--------------------------------------
Distribution: Name = 't', DoF = 5
           P: 3
           Q: 2
    Constant: NaN
       GARCH: {NaN NaN} at Lags [1 3]
        ARCH: {0.2 0.1} at Lags [1 2]
    Leverage: {NaN NaN} at Lags [1 2]
```

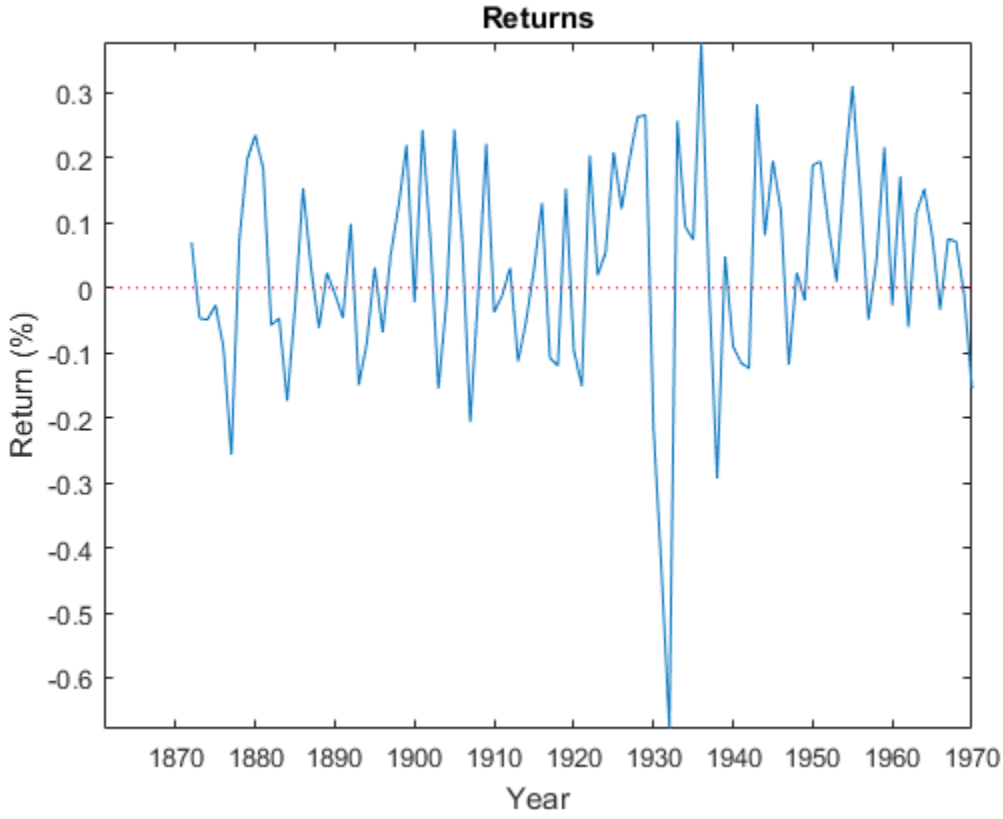
To estimate the remaining parameters, you can pass `Mdl` and your data to estimate and use the specified parameters as equality constraints. Or, you can specify the rest of the parameter values, and then simulate or forecast conditional variances from the GARCH model by passing the fully specified model to `simulate` or `forecast`, respectively.

Estimate GJR Model

Fit a GJR model to an annual time series of stock price index returns from 1861-1970. The example follows from “Create GJR Model”.

Load the Nelson-Plosser data set. Convert the yearly stock price indices (`SP`) to returns. Plot the returns.

```
load Data_NelsonPlosser;
sp = price2ret(DataTable.SP);

figure;
plot(dates(2:end),sp);
hold on;
plot([dates(2) dates(end)],[0 0],'r:'); % Plot y = 0
hold off;
title('Returns');
ylabel('Return (%)');
xlabel('Year');
axis tight;
```



The return series does not seem to have a conditional mean offset, and seems to exhibit volatility clustering. That is, the variability is smaller for earlier years than it is for later years. For this example, assume that an GJR(1,1) model is appropriate for this series.

Create a GJR(1,1) model. The conditional mean offset is zero by default. The software includes a conditional variance model constant by default.

```
Mdl = gjr('GARCHLags',1,'ARCHLags',1,'LeverageLags',1);
```

Fit the GJR(1,1) model to the data.

```
EstMdl = estimate(Mdl,sp);
```

```
    GJR(1,1) Conditional Variance Model:
    ------------------------------------
    Conditional Probability Distribution: Gaussian

                               Standard          t
     Parameter       Value       Error       Statistic
    -----------   -----------   ------------   -----------
     Constant     0.00457276    0.00441991        1.03458
     GARCH{1}       0.558076      0.240004        2.32528
      ARCH{1}       0.204606      0.178856        1.14397
  Leverage{1}       0.180658      0.268015       0.674059
```

`EstMdl` is a fully specified `gjr` model object. That is, it does not contain `NaN` values. You can assess the adequacy of the model by generating residuals using `infer`, and then analyzing them.

To simulate conditional variances or responses, pass `EstMdl` to `simulate`. See "Simulate GJR Model Observations and Conditional Variances".

To forecast innovations, pass `EstMdl` to `forecast`. See "Forecast GJR Model Conditional Variances".

Simulate GJR Model Observations and Conditional Variances

Simulate conditional variance or response paths from a fully specified `gjr` model object. That is, simulate from an estimated `gjr` model or a known `gjr` model in which you specify all parameter values. This example follows from "Estimate GJR Model".

Load the Nelson-Plosser data set. Convert the yearly stock price indices to returns.

```
load Data_NelsonPlosser;
sp = price2ret(DataTable.SP);
```

Create a GJR(1,1) model. Fit the model to the return series.

```
Mdl = gjr(1,1);
EstMdl = estimate(Mdl,sp);
```

```
    GJR(1,1) Conditional Variance Model:
    ------------------------------------
    Conditional Probability Distribution: Gaussian

                               Standard          t
```

```
     Parameter       Value          Error       Statistic
    -----------   -----------   ------------   -----------
     Constant     0.00457276    0.00441991        1.03458
     GARCH{1}       0.558076      0.240004        2.32528
      ARCH{1}       0.204606      0.178856        1.14397
  Leverage{1}       0.180658      0.268015       0.674059
```

Simulate 100 paths of conditional variances and responses from the estimated GJR model.

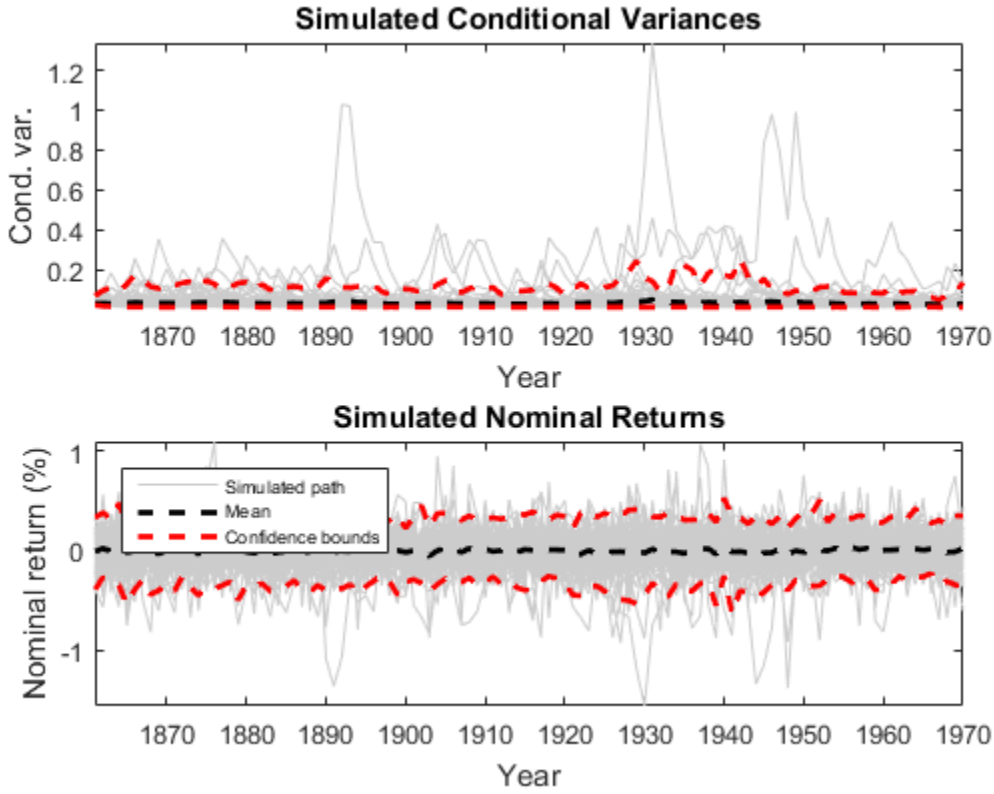
```
numObs = numel(sp); % Sample size (T)
numPaths = 100;     % Number of paths to simulate
rng(1);             % For reproducibility
[VSim,YSim] = simulate(EstMdl,numObs,'NumPaths',numPaths);
```

`VSim` and `YSim` are `T`-by-`numPaths` matrices. Rows correspond to a sample period, and columns correspond to a simulated path.

Plot the average and the 97.5% and 2.5% percentiles of the simulated paths. Compare the simulation statistics to the original data.

```
dates = dates(2:end);
VSimBar = mean(VSim,2);
VSimCI = quantile(VSim,[0.025 0.975],2);
YSimBar = mean(YSim,2);
YSimCI = quantile(YSim,[0.025 0.975],2);

figure;
subplot(2,1,1);
h1 = plot(dates,VSim,'Color',0.8*ones(1,3));
hold on;
h2 = plot(dates,VSimBar,'k--','LineWidth',2);
h3 = plot(dates,VSimCI,'r--','LineWidth',2);
hold off;
title('Simulated Conditional Variances');
ylabel('Cond. var.');
xlabel('Year');
axis tight;

subplot(2,1,2);
h1 = plot(dates,YSim,'Color',0.8*ones(1,3));
hold on;
h2 = plot(dates,YSimBar,'k--','LineWidth',2);
h3 = plot(dates,YSimCI,'r--','LineWidth',2);
hold off;
```

```
title('Simulated Nominal Returns');
ylabel('Nominal return (%)');
xlabel('Year');
axis tight;
legend([h1(1) h2 h3(1)],{'Simulated path' 'Mean' 'Confidence bounds'},...
    'FontSize',7,'Location','NorthWest');
```



Forecast GJR Model Conditional Variances

Forecast conditional variances from a fully specified `gjr` model object. That is, forecast from an estimated `gjr` model or a known `gjr` model in which you specify all parameter values. This example follows from “Estimate GJR Model”.

Load the Nelson-Plosser data set. Convert the yearly stock price indices (`SP`) to returns.

```
load Data_NelsonPlosser;
sp = price2ret(DataTable.SP);
```

Create a GJR(1,1) model and fit it to the return series.

```
Mdl = gjr('GARCHLags',1,'ARCHLags',1,'LeverageLags',1);
EstMdl = estimate(Mdl,sp);
```

```
    GJR(1,1) Conditional Variance Model:
    ------------------------------------
    Conditional Probability Distribution: Gaussian

                               Standard          t
     Parameter       Value       Error       Statistic
    -----------   -----------   ------------   -----------
     Constant     0.00457276    0.00441991        1.03458
     GARCH{1}       0.558076      0.240004        2.32528
      ARCH{1}       0.204606      0.178856        1.14397
  Leverage{1}       0.180658      0.268015       0.674059
```

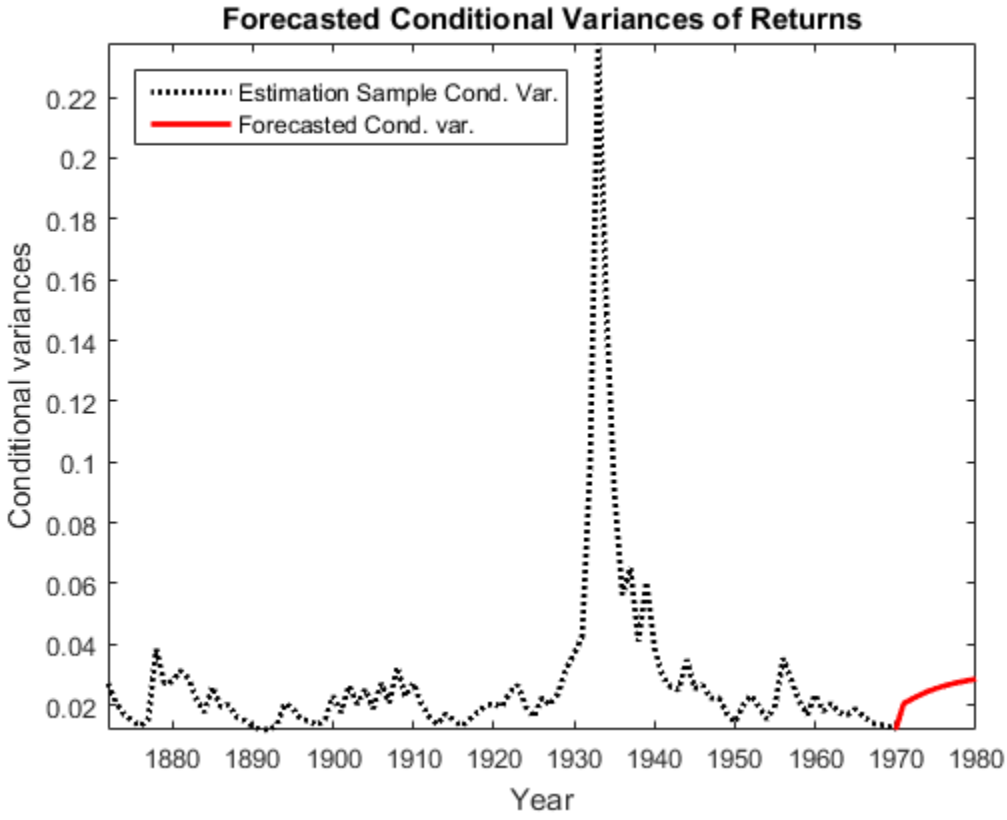
Forecast the conditional variance of the nominal return series 10 years into the future using the estimated GJR model. Specify the entire return series as presample observations. The software infers presample conditional variances using the presample observations and the model.

```
numPeriods = 10;
vF = forecast(EstMdl,numPeriods,'Y0',sp);
```

Plot the forecasted conditional variances of the nominal returns. Compare the forecasts to the observed conditional variances.

```
v = infer(EstMdl,sp);
nV = size(v,1);
dates = dates((end - nV + 1):end);

figure;
plot(dates,v,'k:','LineWidth',2);
hold on;
plot(dates(end):dates(end) + 10,[v(end);vF],'r','LineWidth',2);
title('Forecasted Conditional Variances of Returns');
ylabel('Conditional variances');
xlabel('Year');
axis tight;
```

```
legend({'Estimation Sample Cond. Var.','Forecasted Cond. var.'},...
    'Location','NorthWest');
```



- “Specify GJR Models Using gjr” on page 6-31
- “Modify Properties of Conditional Variance Models” on page 6-42
- “Specify Conditional Mean and Variance Models” on page 5-79
- “Infer Conditional Variances and Residuals” on page 6-77
- “Compare Conditional Variance Models Using Information Criteria” on page 6-87
- “Simulate GARCH Models” on page 6-97
- “Forecast GJR Models” on page 6-123

Properties

Conditional Variance Model Properties	Specify conditional variance model functional form and parameter values

Object Functions

estimate	Fit conditional variance model to data
filter	Filter disturbances through conditional variance model
forecast	Forecast conditional variances from conditional variance models
infer	Infer conditional variances of conditional variance models
print	Display parameter estimation results for conditional variance models
simulate	Monte Carlo simulation of conditional variance models

Create Object

Create `gjr` models using `gjr`.

You can specify a `gjr` model as part of a composition of conditional mean and variance models. For details, see arima.

See Also

arima | `egarch` | `garch`

More About

- "Conditional Variance Models" on page 6-2
- "GJR Model" on page 6-6

Introduced in R2012a

hac

Heteroscedasticity and autocorrelation consistent covariance estimators

Syntax

```
EstCov = hac(X,y)

EstCov = hac(Tbl)

EstCov = hac(Mdl)

EstCov = hac(___,Name,Value)

[EstCov,se,coeff] = hac(___)
```

Description

`EstCov = hac(X,y)` returns robust covariance estimates for ordinary least squares (OLS) coefficient estimates of multiple linear regression models $y = X\beta + \varepsilon$ under general forms of heteroscedasticity and autocorrelation in the innovations process ε.

`NaN`s in the data indicate missing values, which `hac` removes using list-wise deletion. `hac` sets `Data = [X y]`, then it removes any row in `Data` containing at least one `NaN`. This reduces the effective sample size, and changes the time base of the series.

`EstCov = hac(Tbl)` returns robust covariance estimates for OLS coefficient estimates of multiple linear regression models, with predictor data, `X`, in the first `numPreds` columns of the tabular array, `Tbl`, and response data, `y`, in the last column.

`hac` removes all missing values in `Tbl`, indicated by `NaN`s, using list-wise deletion. In other words, `hac` removes all rows in `Tbl` containing at least one `NaN`. This reduces the effective sample size, and changes the time base of the series.

`EstCov = hac(Mdl)` returns robust covariance estimates for OLS coefficient estimates from a fitted multiple linear regression model, `Mdl`, as returned by `fitlm`.

`EstCov = hac(___,Name,Value)` uses any of the input arguments in the previous syntaxes and additional options that you specify by one or more `Name,Value` pair arguments.

For example, use `Name,Value` pair arguments to choose weights for HAC or HC estimators, set a bandwidth for a HAC estimator, or prewhiten the residuals.

`[EstCov,se,coeff] = hac(___)` additionally returns a vector of corrected coefficient standard errors, `se` = `sqrt(diag(EstCov))`, and a vector of OLS coefficient estimates, `coeff`.

Examples

Estimate White's Robust Covariance for OLS Coefficient Estimates

Model an automobile's price with its curb weight, engine size, and cylinder bore diameter using the linear model:

$$\mathtt{price}_i = \beta_0 + \beta_1 \mathtt{curbWeight}_i + \beta_2 \mathtt{engineSize}_i + \beta_3 \mathtt{bore}_i + \varepsilon_i.$$

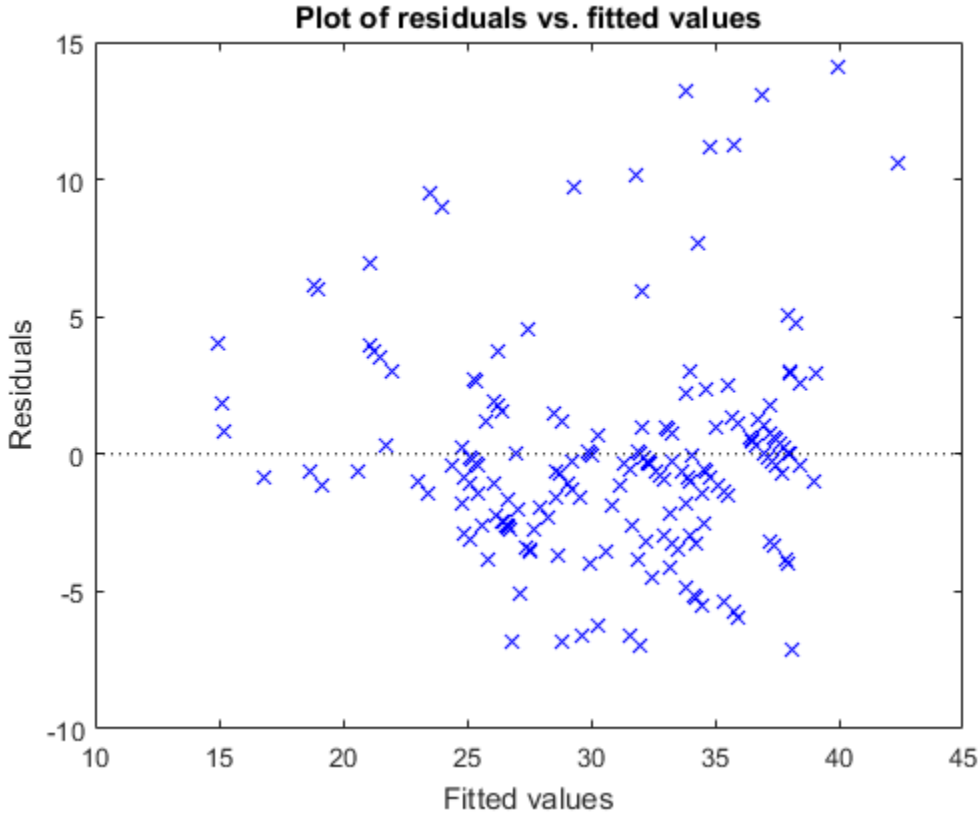
Estimate model coefficients and White's robust covariance.

Load the 1985 automobile imports data set (Frank and Asuncion, 2012). Extract the columns that correspond to the predictor and response variables.

```
load imports-85
Tbl = table(X(:,7),X(:,8),X(:,9),X(:,15),...
   'Variablenames',{'curbWeight','engineSize',...
   'bore','price'});
```

Fit the linear model to the data and plot the residuals versus the fitted values.

```
Mdl = fitlm(Tbl);
plotResiduals(Mdl,'fitted')
```



The residuals seem to flare out, which indicates heteroscedasticity.

Compare the coefficient covariance estimate from OLS and from using `hac` to calculate White's heteroscedasticity robust estimate.

```
[LSCov,LSSe,coeff] = hac(Mdl,'type','HC','weights',...
   'CLM','display','off');
    %Usual OLS estimates, also found in
    %Mdl.CoefficientCovariance
LSCov
[WhiteCov,WhiteSe,coeff] = hac(Mdl,'type','HC','weights',...
   'HCO','display','off'); % White's estimates
WhiteCov
```

```
LSCov =

   13.7124    0.0000    0.0120   -4.5609
    0.0000    0.0000   -0.0000   -0.0005
    0.0120   -0.0000    0.0002   -0.0017
   -4.5609   -0.0005   -0.0017    1.8195


WhiteCov =

   15.5122   -0.0008    0.0137   -4.4461
   -0.0008    0.0000   -0.0000   -0.0003
    0.0137   -0.0000    0.0001   -0.0010
   -4.4461   -0.0003   -0.0010    1.5707
```

The OLS coefficient covariance estimate is not equal to White's robust estimate because the latter accounts for the heteroscedasticity in the residuals.

Estimate the Newey-West OLS Coefficient Covariance Matrix

Model nominal GNP (`GNPN`) with consumer price index (`CPI`), real wages (`WR`), and the money stock (`MS`) using the linear model:

$$\mathtt{GNPN}_i = \beta_0 + \beta_1 \mathtt{CPI}_i + \beta_2 \mathtt{WR}_i + \beta_3 \mathtt{MS}_i + \varepsilon_i.$$

Estimate the model coefficients and the Newey-West OLS coefficient covariance matrix.

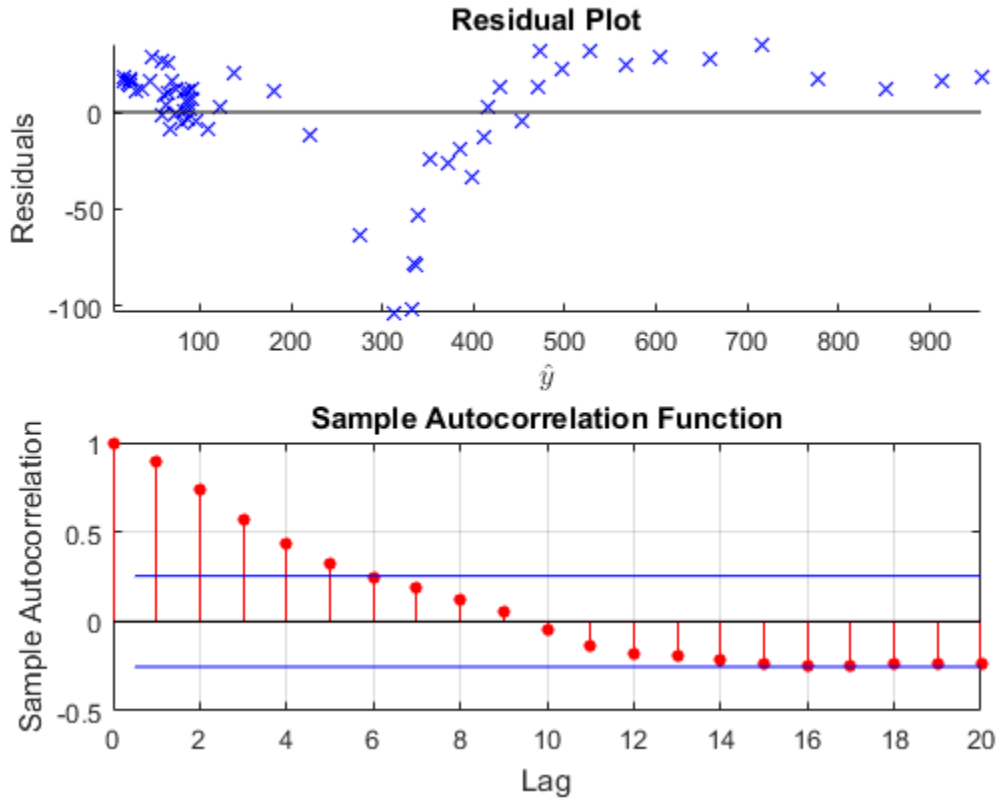
Load the Nelson Plosser data set.

```
load Data_NelsonPlosser
Tbl = DataTable(:,[8,10,11,2]);  % Tabular array containing the variables
T = sum(~any(ismissing(Tbl),2)); % Remove NaNs to obtain sample size

y = Tbl{:,4};   % Numeric response
X = Tbl{:,1:3}; % Numeric matrix of predictors
```

Fit the linear model. Remove the beginning block of `NaN` values in the residual vector for `autocorr`.

```
Mdl = fitlm(X,y);
resid = Mdl.Residuals.Raw(~isnan(Mdl.Residuals.Raw));

figure
```

```
subplot(2,1,1)
hold on
plotResiduals(Mdl,'fitted')
axis tight
plot([min(Mdl.Fitted) max(Mdl.Fitted)],[0 0],'k-')
title('Residual Plot')
xlabel('$\hat y$','Interpreter','latex')
ylabel('Residuals')
axis tight
subplot(2,1,2)
autocorr(resid)
```



The residual plot exhibits signs of heteroscedasticity, autocorrelation, and possibly model misspecification. The sample autocorrelation function clearly exhibits autocorrelation.

Calculate the lag selection parameter for the standard Newey-West HAC estimate (Andrews and Monohan, 1992).

```
maxLag = floor(4*(T/100)^(2/9));
```

Estimate the standard Newey-West OLS coefficient covariance using `hac` by setting the bandwidth to `maxLag` + 1. Display the OLS coefficient estimates, their standard errors, and the covariance matrix.

```
EstCov = hac(X,y,'bandwidth',maxLag+1,'display','full');
```

```
Estimator type: HAC
Estimation method: BT
Bandwidth: 4.0000
Whitening order: 0
Effective sample size: 62
Small sample correction: on

Coefficient Estimates:

       |  Coeff      SE
-------------------------
 Const | 20.2317  35.0767
 x1    | -0.1000   0.7965
 x2    | -1.5602   1.1546
 x3    |  2.6329   0.2043

Coefficient Covariances:

       |   Const       x1        x2        x3
-----------------------------------------------
 Const | 1230.3727  -15.3285  -24.2677   6.7855
 x1    |  -15.3285    0.6343   -0.2960  -0.0957
 x2    |  -24.2677   -0.2960    1.3331  -0.1285
 x3    |    6.7855   -0.0957   -0.1285   0.0418
```

The first column in the output contains the OLS estimates ($\hat{\beta}_0, \ldots, \hat{\beta}_3$, respectively), and the second column contains their standard errors. The last four columns contained in the table represent the estimated coefficient covariance matrix. For example, $Cov(\hat{\beta}_1, \hat{\beta}_2) = -0.2960$.

Alternatively, pass in a tabular array to `hac`.

```
EstCov = hac(Tbl,'bandwidth',maxLag+1,'display','off');
```

The advantage of passing in a tabular array is that the top and left margins of the covariance table use the variable names.

Plot Kernel Densities

Plot the kernel density functions available in `hac`.

Set domain, `x`, and range `w`.

```
x = (0:0.001:3.2)';
w = zeros(size(x));
```

Compute the truncated kernel density.

```
cTR = 2; % Renormalization constant
TR = (abs(x) <= 1);
TRRn = (abs(cTR*x) <= 1);
wTR = w;
wTR(TR) = 1;
wTRRn = w;
wTRRn(TRRn) = 1;
```

Compute the Bartlett kernel density.

```
cBT = 2/3; % Renormalization constant
BT = (abs(x) <= 1);
BTRn = (abs(cBT*x) <= 1);
wBT = w;
wBT(BT) = 1-abs(x(BT));
wBTRn = w;
wBTRn(BTRn) = 1-abs(cBT*x(BTRn));
```

Compute the Parzen kernel density.

```
cPZ = 0.539285; % Renormalization constant
PZ1 = (abs(x) >= 0) & (abs(x) <= 1/2);
PZ2 = (abs(x) >= 1/2) & (abs(x) <= 1);
PZ1Rn = (abs(cPZ*x) >= 0) & (abs(cPZ*x) <= 1/2);
PZ2Rn = (abs(cPZ*x) >= 1/2) & (abs(cPZ*x) <= 1);
wPZ = w;
wPZ(PZ1) = 1-6*x(PZ1).^2+6*abs(x(PZ1)).^3;
```

```
wPZ(PZ2) = 2*(1-abs(x(PZ2))).^3;
wPZRn = w;
wPZRn(PZ1Rn) = 1-6*(cPZ*x(PZ1Rn)).^2 ...
    + 6*abs(cPZ*x(PZ1Rn)).^3;
wPZRn(PZ2Rn) = 2*(1-abs(cPZ*x(PZ2Rn))).^3;
```

Compute the Tukey-Hanning kernel density.

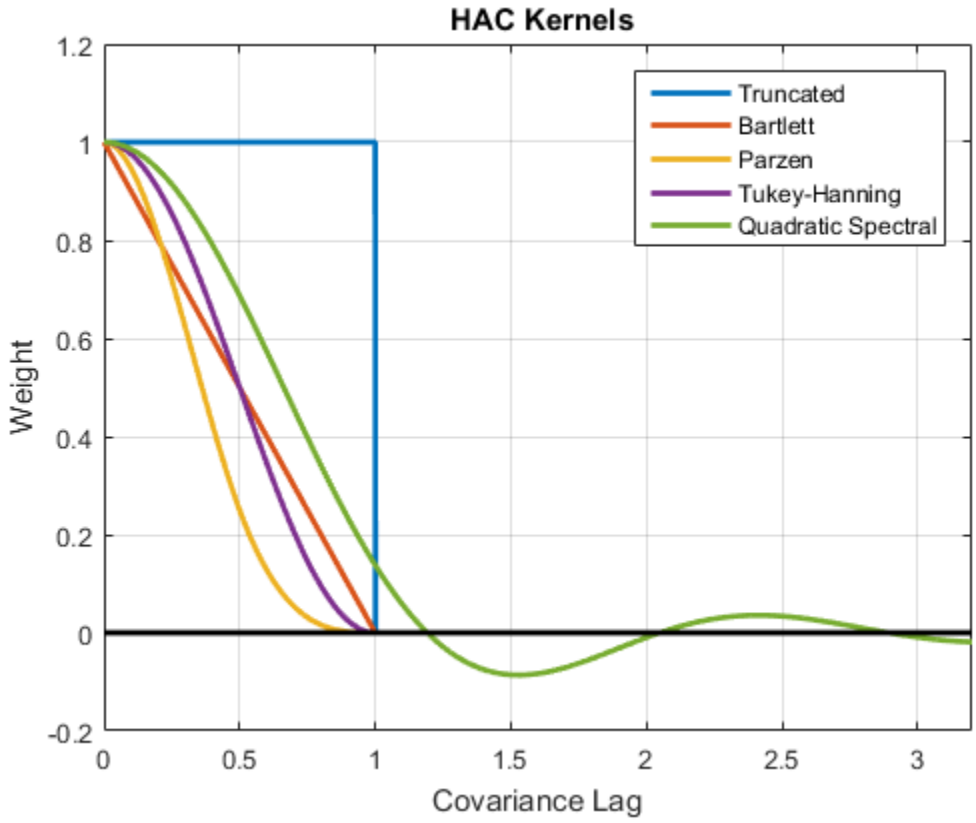
```
cTH = 3/4; % Renormalization constant
TH = (abs(x) <= 1);
THRn = (abs(cTH*x) <= 1);
wTH = w;
wTH(TH) = (1+cos(pi*x(TH)))/2;
wTHRn = w;
wTHRn(THRn) = (1+cos(pi*cTH*x(THRn)))/2;
```

Compute the quadratic spectral kernel density.

```
argQS = 6*pi*x/5;
w1 = 3./(argQS.^2);
w2 = (sin(argQS)./argQS)-cos(argQS);
wQS = w1.*w2;
wQS(x == 0) = 1;
wQSRn = wQS; % Renormalization constant = 1
```

Plot the kernel densities.

```
figure
plot(x,[wTR,wBT,wPZ,wTH,wQS],'LineWidth',2)
hold on
plot(x,w,'k','LineWidth',2)
axis([0 3.2 -0.2 1.2])
grid on
title('{\bf HAC Kernels}')
legend({'Truncated','Bartlett','Parzen','Tukey-Hanning',...
    'Quadratic Spectral'})
xlabel('Covariance Lag')
ylabel('Weight')
```

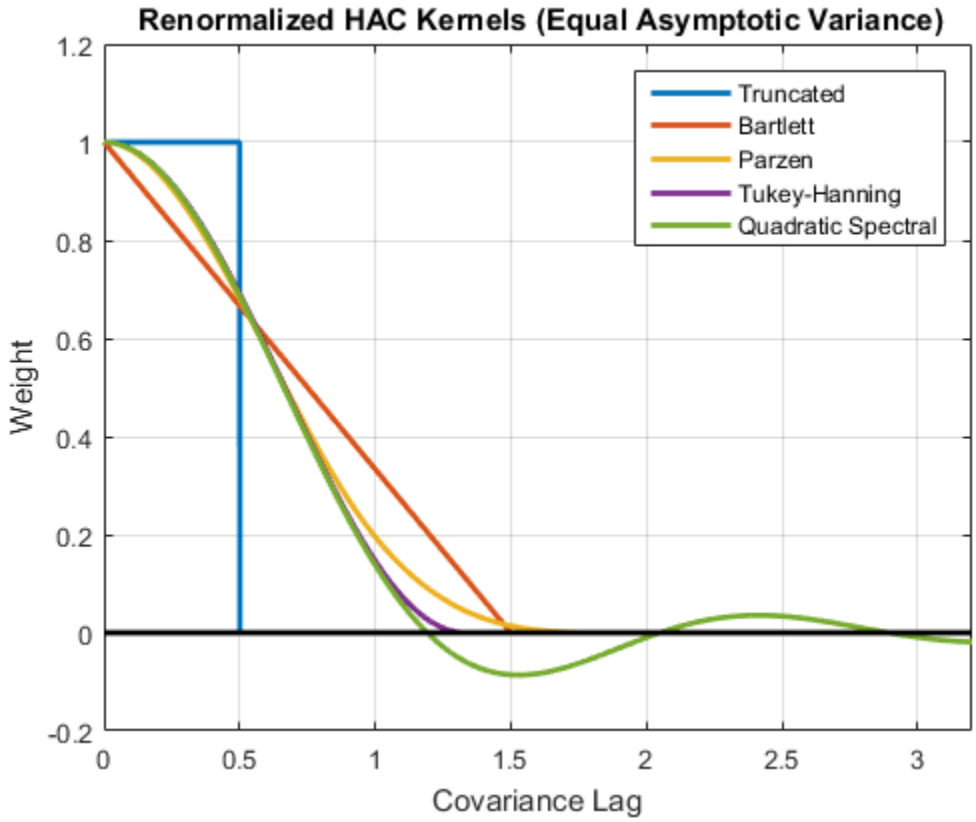


All graphs are truncated at `Covariance Lag` = 1, except for the quadratic spectral. The quadratic spectral density approaches 0 as `Covariance Lag` gets large, but does not get truncated.

Plot renormalized kernels. Unlike the densities in the previous plot, these have the same asymptotic variance (Andrews, 1991).

```
figure
plot(x,[wTRRn,wBTRn,wPZRn,wTHRn,wQSRn],'LineWidth',2)
hold on
plot(x,w,'k','LineWidth',2)
axis([0 3.2 -0.2 1.2])
grid on
title('{\bf Renormalized HAC Kernels} (Equal Asymptotic Variance)')
```

```
legend({'Truncated','Bartlett','Parzen','Tukey-Hanning',...
    'Quadratic Spectral'})
xlabel('Covariance Lag')
ylabel('Weight')
```



Examine the effects of changing the bandwidth parameter on the quadratic spectral density.

Assign several bandwidth values to `b`. Assign the domain to `l`. Calculate `x = l/|b|`.

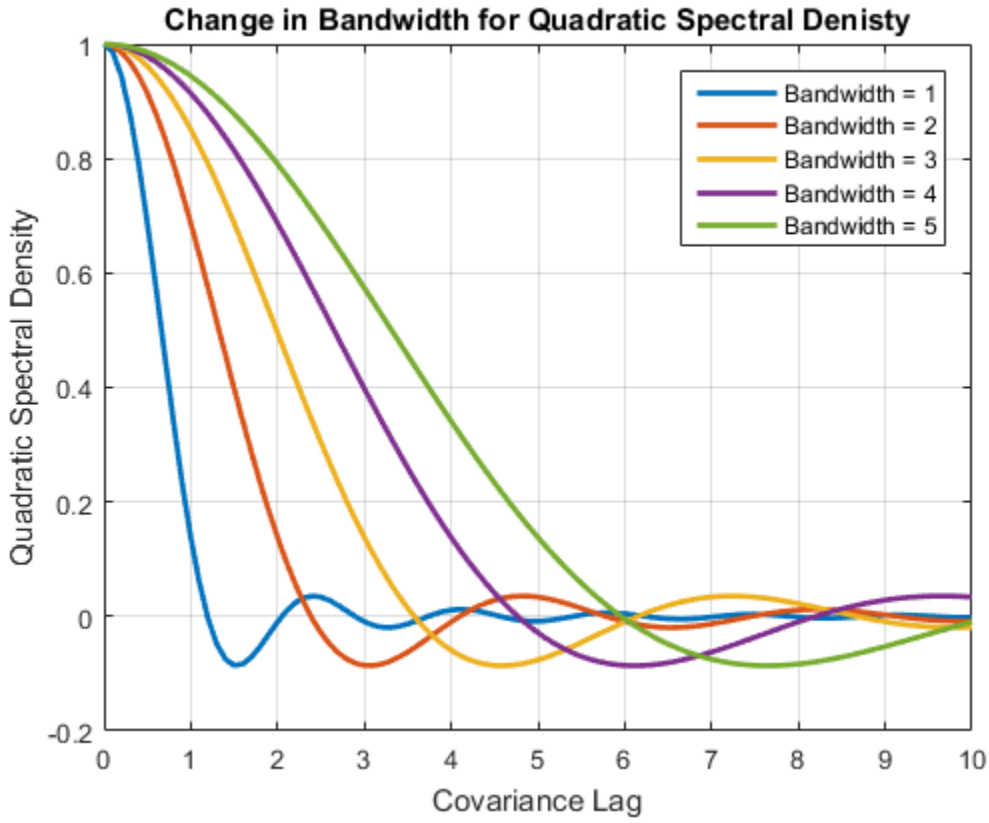
```
b = (1:5)';
l = (0:0.1:10);
x = bsxfun(@rdivide,repmat(l,[size(b),1]),b)';
```

Calculate the quadratic spectral density under the domain for each bandwidth value.

```
argQS = 6*pi*x/5;
w1 = 3./(argQS.^2);
w2 = (sin(argQS)./argQS)-cos(argQS);
wQS = w1.*w2;
wQS(x == 0) = 1;
```

Plot the quadratic spectral densities.

```
figure;
plot(l,wQS,'LineWidth',2);
grid on;
xlabel('Covariance Lag');
ylabel('Quadratic Spectral Density');
title('Change in Bandwidth for Quadratic Spectral Denisty');
legend('Bandwidth = 1','Bandwidth = 2','Bandwidth = 3',...
    'Bandwidth = 4','Bandwidth = 5');
```



As the bandwidth increases, the kernel imparts more weight to larger lags.

- "Classical Model Misspecification Tests"
- "Time Series Regression I: Linear Models"
- "Time Series Regression VI: Residual Diagnostics"
- "Time Series Regression X: Generalized Least Squares and HAC Estimators"
- "Plot a Confidence Band Using HAC Estimates" on page 3-95
- "Change the Bandwidth of a HAC Estimator" on page 3-105

Input Arguments

`X` — Predictor data

numeric matrix

Predictor data for the multiple linear regression model, specified as a `numObs`-by-`numPreds` numeric matrix.

`numObs` is the number of observations and `numPreds` is the number of predictor variables.

Data Types: `double`

`y` — Response data

vector

Response data for the multiple linear regression model, specified as a `numObs`-by-1 vector with numeric or logical entries.

Data Types: `double | logical`

`Tbl` — Predictor and response data

tabular array

Predictor and response data for the multiple linear regression model, specified as a `numObs`-by-`numPreds` + 1 tabular array.

The first `numPreds` variables of `Tbl` are the predictor data, and the last variable is the response data.

The predictor data must be numeric, and the response data must be numeric or logical.

Data Types: `table`

`Mdl` — Fitted linear model

`LinearModel` model

Fitted linear model, specified as a model returned by `fitlm`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example:
`'type','HAC','bandwidth',floor(4*(T/100)^(2/9))+1,'weights','BT'`
specifies the standard Newey-West OLS coefficient covariance estimate.

`'varNames'` — Variable names

cell vector of strings

Variable names used in displays and plots of the results, specified as the comma-separated pair consisting of `'varNames'` and a cell vector of strings. `varNames` must have length `numPreds`, and each cell corresponds to a variable name. The software truncates all variable names to the first five characters.

`varNames` must include variable names for all variables in the model, such as an intercept term (e.g., `'Const'`) or higher-order terms (e.g., `'x1^2'` or `'x1:x2'`).

The default variable names for:

- The matrix `X` is the cell vector of strings `{'x1','x2',...}`
- The tabular array `Tbl` is the property `Tbl.Properties.VariableNames`
- The linear model `Mdl` is the property `Mdl.CoefficientNames`

Example: `'varNames',{'Const','AGE','BBD'}`

Data Types: `cell`

`'intercept'` — Indicate whether to include model intercept

`true` (default) | `false`

Indicate whether to include model intercept when `hac` fits the model, specified as the comma-separated pair consisting of `'intercept'` and a logical value.

Value	Description
`true`	Include an intercept in the model.
`false`	Exclude an intercept from the model.

If you specify `Mdl`, then `hac` ignores `intercept` and uses the intercept in `Mdl`.

Example: `'intercept',false`

Data Types: `logical`

`'type'` — Coefficient covariance estimator type
`'HAC'` (default) | `'HC'`

Coefficient covariance estimator type, specified as the comma-separated pair consisting of `'type'` and a string.

Value	Covariance Estimate	Usage
`'HAC'`	Return heteroscedasticity-and-autocorrelation-consistent (HAC) estimate as described in [1], [2], [6], and [10].	When residuals exhibit both heteroscedasticity and autocorrelation
`'HC'`	Return heteroscedasticity-consistent (HC) estimate as described in [3], [9], and [12].	When residuals exhibit only heteroscedasticity

Example: `'type','HC'`

`'weights'` — Coefficient covariance estimator weighting scheme
`'CLM'` | `'HC0'` | `'HC1'` | `'HC2'` | `'HC3'` | `'HC4'` | `'TR'` | `'BT'` | `'PZ'` | `'TH'` | `'QS'` | vector

Coefficient covariance estimator weighting scheme, specified as the comma-specified pair consisting of `'weights'` and a string or length `numObs` numeric vector.

Set `'weights'` to specify the structure of the innovations covariance Ω. `hac` uses this specification to compute $\hat{\Phi} = X'\hat{\Omega}X$ (see "Sandwich Estimators" on page 9-601).

- If `type` is `HC`, then $\hat{\Omega} = \mathrm{diag}(\omega)$, where ω_i estimates the ith innovation variance, $i = 1,\ldots,T$, and $T =$ `numObs`. `hac` estimates ω_i using the ith residual, ε_i, its *leverage* $h_i = x_i(X'X)^{-1}x_i'$, $d_i = \min\left(4, \frac{h_i}{\bar{h}}\right)$, and the degrees of freedom, dfe.

  Use the following table to choose `'weights'`.

Value	Weight	Reference
`'CLM'`	$\omega_i = \frac{1}{dfe}\sum_{i=1}^{T}\varepsilon_i^2$	[7]
`'HCO'` (default when `'type','HC'`)	$\omega_i = \varepsilon_i^2$	[12]
`'HC1'`	$\omega_i = \frac{T}{dfe}\varepsilon_i^2$	[9]
`'HC2'`	$\omega_i = \frac{\varepsilon_i^2}{1-h_i}$	[9]
`'HC3'`	$\omega_i = \frac{\varepsilon_i^2}{(1-h_i)^2}$	[9]
`'HC4'`	$\omega_i = \frac{\varepsilon_i^2}{(1-h_i)^{d_i}}$	[3]

- If `type` is `HAC`, then `hac` weights the component products that form $\hat{\Phi}$, $x_i'\varepsilon_i\varepsilon_j x_j$, using a measure of autocorrelation strength, $\omega(l)$, at each lag, $l = |i-j|$. $\omega(l) = k(l/b)$, where k is a kernel density estimator and b is a bandwidth specified by `'bandwidth'`.

  Use the following table to choose `'weights'`.

Value	Kernel Density	Kernel Density Function	Reference
`'TR'`	Truncated	$k(z) = \begin{cases} 1 \text{ for } \lvert z\rvert \le 1 \\ 0 \text{ otherwise} \end{cases}$	[13]
`'BT'` (default when `'type','HAC'`)	Bartlett	$k(z) = \begin{cases} 1-\lvert z\rvert \text{ for } \lvert z\rvert \le 1 \\ 0 \text{ otherwise} \end{cases}$	[10]

Value	Kernel Density	Kernel Density Function	Reference
`'PZ'`	Parzen	$k(z)=\begin{cases}1-6x^2+6\lvert z\rvert^3 & \text{for } 0\le z<0.5\\ 2(1-\lvert z\rvert)^3 & \text{for } 0.5\le z\le 1\\ 0 & \text{otherwise}\end{cases}$	[6]
`'TH'`	Tukey-Hanning	$k(z)=\begin{cases}\frac{1+\cos(\pi z)}{2} & \text{for } \lvert z\rvert\le 1\\ 0 & \text{otherwise}\end{cases}$	[1]
`'QS'`	Quadratic spectral	$k(z)=\frac{25}{12\pi^2 z^2}\left(\frac{\sin(6\pi z/5)}{6\pi z/5}-\cos(6\pi z/5)\right)$	[1]

For a visual description of these kernel densities, see “Plot Kernel Densities” on page 9-588.

- For either `type`, you can set `'weights'` to any length `numObs` numeric vector without containing `NaN`s. However, a user-defined `weights` vector might not produce positive definite matrices.

  If you set `weights` to a numeric vector, then `hac` sets `Data = [X y weights]` = `[DS weights]` and removes any row in `Data` containing at least one `NaN`.

Example: `'weights','QS'`

Data Types: `single` | `double`

`'bandwidth'` — Bandwidth value or method

`'AR1'` `'AR1MLE'` (default) | `'AR1OLS'` | `'ARMA11'` | positive scalar

Bandwidth value or method indicating how `hac` estimates the data-driven bandwidth parameter, specified as the comma-separated pair consisting of `'bandwidth'` and either a scalar or a string.

- If `type` is `HC`, then `hac` ignores `bandwidth`.
- If `type` is `HAC`, then provide a nonzero scalar for the bandwidth, or use a string listed in the following table to indicate which model and method `hac` uses to estimate the data-driven bandwidth. For details, see [1].

Value	Model	Method
'AR1'	AR(1)	Maximum Likelihood
'AR1MLE'	AR(1)	Maximum Likelihood
'AR1OLS'	AR(1)	OLS
'ARMA11'	ARMA(1,1)	Maximum Likelihood

Example: 'bandwidth',floor(4*(T/100)^(2/9))+1

Data Types: single | double

'smallT' — Indicate whether to apply small sample correction

true | false

Indicate whether to apply the small sample correction to the estimated covariance matrix, specified as the comma-separated pair consisting of 'smallT' and a logical value.

The small sample correction factor is $\frac{T}{dfe}$, where T is the sample size and dfe is the residual degrees of freedom. For details, see [1].

Value	Description
true	Apply the small sample correction.
false	Do not apply the small sample correction.

- If type is HC, then smallT is false.
- If type is HAC, then smallT is true.

Example: 'smallT',false

Data Types: logical

'whiten' — Lag order for VAR filter

0 (default) | nonnegative integer

Lag order for the VAR model prewhitening filter, specified as the comma-separated pair consisting of 'whiten' and a nonnegative integer.

For details on prewhitening filters, see [2].

- If `type` is `HC`, then `hac` ignores `'whiten'`.
- If `'whiten'` is `0`, then `hac` does not apply a prewhitening filter.

Example: `'whiten',1`

Data Types: `single | double`

`'display'` — Display results in Command Window

`'cov'` (default) | `'full'` | `'off'`

Display results in the Command Window in tabular form, specified as the comma-separated pair consisting of `'display'` and a string in the following table.

Value	Description
`'cov'`	Display a table of the estimated covariances of the OLS coefficients.
`'full'`	Display a table of coefficient estimates, their standard errors, and their estimated covariances.
`'off'`	Do not display an estimates table to the Command Window.

Example: `'display','off'`

Output Arguments

`EstCov` — Coefficient covariance estimate

array

Coefficient covariance estimate, returned as a `numPreds`-by-`numPreds` array.

`EstCov` is organized according to the order of the predictor matrix columns, or as specified by `Mdl`. For example, in a model with an intercept, the estimated covariance of $\hat{\beta}_1$ (corresponding to the predictor x_1) and $\hat{\beta}_2$ (corresponding to the predictor x_2) are in positions (2,3) and (3,2) of `EstCov`, respectively.

`se` — Coefficient standard error estimates

vector

Coefficient standard error estimates, returned as a length `numPreds` vector whose elements are `sqrt(diag(EstCov))`.

`se` is organized according to the order of the predictor matrix columns, or as specified by `Mdl`. For example, in a model with an intercept, the estimated standard error of $\hat{\beta}_1$ (corresponding to the predictor x_1) is in position 2 of `se`, and is the square root of the value in position (2,2) of `EstCov`.

`coeff` — OLS coefficient estimates

vector

OLS coefficient estimates, returned as a `numPreds` vector.

`coeff` is organized according to the order of the predictor matrix columns, or as specified by `Mdl`. For example, in a model with an intercept, the value of $\hat{\beta}_1$ (corresponding to the predictor x_1) is in position 2 of `coeff`.

More About

Sandwich Estimators

This estimator has the form $A^{-1}BA^{-1}$.

The estimated covariance matrix that `hac` returns is called a *sandwich* estimator because of its form:

$$c(X'X)^{-1}\Phi(X'X)^{-1},$$

where $(X'X)^{-1}$ is the *bread*, $\hat{\Phi} = X'\hat{\Omega}X$ is the *meat*, and c is an optional small sample correction.

Lag-Truncation Parameter

This parameter directs a kernel density to assign no weight to all lags above its value.

For kernel densities with unit-interval support, the bandwidth parameter, b, is often called the *lag-truncation parameter* since $w(l) = k(l/b) = 0$ for lags $l > b$.

Tips

[2] recommends prewhitening for HAC estimators to reduce bias. The procedure tends to increase estimator variance and mean-squared error, but can improve confidence interval coverage probabilities and reduce the over-rejection of t statistics.

Algorithms

- The original White HC estimator, specified by `'type','HC','weights','HC0'`, is justified asymptotically. The other `weights` values, `HC1`, `HC2`, `HC3`, and `HC4`, are meant to improve small-sample performance. [6] and [3] recommend using `HC3` and `HC4`, respectively, in the presence of influential observations.
- HAC estimators formed using the truncated kernel might not be positive semidefinite in finite samples. [10] proposes using the Bartlett kernel as a remedy, but the resulting estimator is suboptimal in terms of its rate of consistency. The quadratic spectral kernel achieves an optimal rate of consistency.
- The default estimation method for HAC bandwidth selection is `AR1MLE`. It is generally more accurate, but slower, than the AR(1) alternative, `AR1OLS`. If you specify `'bandwidth','ARMA11'`, then `hac` estimates the model using maximum likelihood.
- Bandwidth selection models might exhibit sensitivity to the relative scale of the predictors in `X`.

- “Autocorrelation and Partial Autocorrelation” on page 3-13
- “Engle’s ARCH Test” on page 3-25
- “Nonspherical Models” on page 3-94

References

[1] Andrews, D. W. K. “Heteroskedasticity and Autocorrelation Consistent Covariance Matrix Estimation.” *Econometrica*. Vol. 59, 1991, pp. 817–858.

[2] Andrews, D. W. K., and J. C. Monohan. “An Improved Heteroskedasticity and Autocorrelation Consistent Covariance Matrix Estimator.” *Econometrica*. Vol. 60, 1992, pp. 953–966.

[3] Cribari-Neto, F. "Asymptotic Inference Under Heteroskedasticity of Unknown Form." *Computational Statistics & Data Analysis*. Vol. 45, 2004, pp. 215–233.

[4] den Haan, W. J., and A. Levin. "A Practitioner's Guide to Robust Covariance Matrix Estimation." In *Handbook of Statistics*. Edited by G. S. Maddala and C. R. Rao. Amsterdam: Elsevier, 1997.

[5] Frank, A., and A. Asuncion. UCI Machine Learning Repository. Irvine, CA: University of California, School of Information and Computer Science. http://archive.ics.uci.edu/ml, 2012.

[6] Gallant, A. R. *Nonlinear Statistical Models*. Hoboken, NJ: John Wiley & Sons, Inc., 1987.

[7] Kutner, M. H., C. J. Nachtsheim, J. Neter, and W. Li. *Applied Linear Statistical Models*. 5th ed. New York: McGraw-Hill/Irwin, 2005.

[8] Long, J. S., and L. H. Ervin. "Using Heteroscedasticity-Consistent Standard Errors in the Linear Regression Model." *The American Statistician*. Vol. 54, 2000, pp. 217–224.

[9] MacKinnon, J. G., and H. White. "Some Heteroskedasticity-Consistent Covariance Matrix Estimators with Improved Finite Sample Properties." *Journal of Econometrics*. Vol. 29, 1985, pp. 305–325.

[10] Newey, W. K., and K. D. West. "A Simple, Positive-Definite, Heteroskedasticity and Autocorrelation Consistent Covariance Matrix." *Econometrica*. Vol. 55, 1987, pp. 703–708.

[11] Newey, W. K, and K. D. West. “Automatic Lag Selection in Covariance Matrix Estimation.” *The Review of Economic Studies*. Vol. 61 No. 4, 1994, pp. 631–653.

[12] White, H. "A Heteroskedasticity-Consistent Covariance Matrix and a Direct Test for Heteroskedasticity." *Econometrica*. Vol. 48, 1980, pp. 817–838.

[13] White, H. *Asymptotic Theory for Econometricians*. New York: Academic Press, 1984.

See Also

`fitlm` | `lscov`

Introduced in R2013a

hpfilter

Hodrick-Prescott filter for trend and cyclical components

Syntax

```
hpfilter(S)
hpfilter(S,smoothing)
T = hpfilter(...)
[T,C] = hpfilter(...)
```

Description

- `hpfilter(S)` uses a Hodrick-Prescott filter and a default smoothing parameter of 1600 to separate the columns of `S` into trend and cyclical components. `S` is an m-by-n matrix with m samples from n time series. A plot displays each time series together with its trend (the time series with the cyclic component removed).
- `hpfilter(S,smoothing)` applies the smoothing parameter `smoothing` to the columns of `S`. If `smoothing` is a scalar, `hpfilter` applies it to all columns. If `S` has n columns and `smoothing` is a conformable vector (n-by-1 or 1-by-n), `hpfilter` applies the vector components of `smoothing` to the corresponding columns of `S`.

  If the smoothing parameter is `0`, no smoothing takes place. As the smoothing parameter increases in value, the smoothed series becomes more linear. A smoothing parameter of `Inf` produces a linear trend component.

  Appropriate values of the smoothing parameter depend upon the periodicity of the data. The following reference suggests the following values:

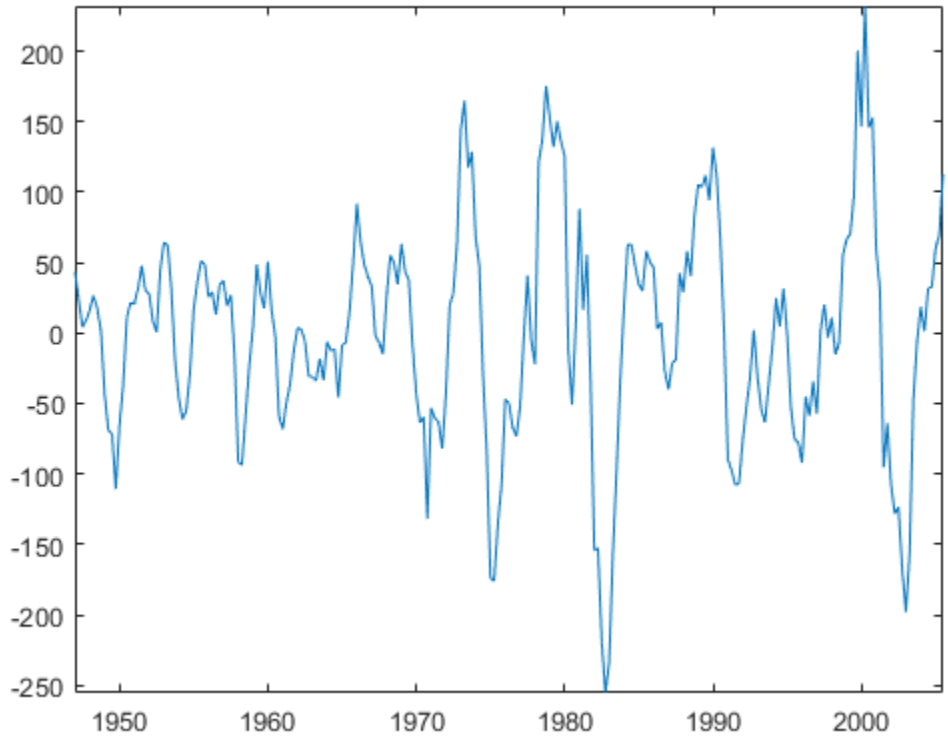
  - Yearly — 100
  - Quarterly — 1600
  - Monthly — 14400
- `T = hpfilter(...)` returns the trend components of the columns of `S` in `T`, without plotting.
- `[T,C] = hpfilter(...)` returns the cyclical components of the columns of `S` in `C`, without plotting.

Examples

Apply the HP Filter to Time Series Data

Plot the cyclical component of the U.S. post-WWII seasonally-adjusted real GNP. In `hpfilter`, specify that `smoothing` is 1600, which is appropriate for quarterly data.

```
load Data_GNP
gnpDate = dates;
realgnp = DataTable.GNPR;
[~,c] = hpfilter(realgnp,1600);

plot(gnpDate,c)
axis tight
```



More About

Algorithms

The Hodrick-Prescott filter separates a time series y_t into a trend component T_t and a cyclical component C_t such that $y_t = T_t + C_t$. It is equivalent to a cubic spline smoother, with the smoothed portion in T_t.

The objective function for the filter has the form

$$\sum_{t=1}^{m} C_t^2 + \lambda \sum_{t=2}^{m-1} ((T_{t+1} - T_t) - (T_t - T_{t-1}))^2$$

where m is the number of samples and λ is the smoothing parameter. The programming problem is to minimize the objective over all T_1, ..., T_m. The first sum minimizes the difference between the time series and its trend component (which is its cyclical component). The second sum minimizes the second-order difference of the trend component (which is analogous to minimization of the second derivative of the trend component).

References

[1] Hodrick, Robert J, and Edward C. Prescott. "Postwar U.S. Business Cycles: An Empirical Investigation." *Journal of Money, Credit, and Banking*. Vol. 29, No. 1, February 1997, pp. 1–16.

Introduced in R2006b

i10test

Paired integration and stationarity tests

Syntax

```
i10test(X)
i10test(X,Name,Value)

H = i10test(___)
[H,PValue] = i10test(___)
```

Description

`i10test(X)` displays the results of paired integration and stationarity tests on the variables in `X`.

`i10test(X,Name,Value)` uses additional options specified by one or more `Name,Value` pairs. If you specify the `numDiffs` option, the paired integration and stationarity tests are conducted on the variables in `X` and their specified differences.

`H = i10test(___)` returns logical values with the rejection decisions for the tests. You can use any of the previous input arguments.

`[H,PValue] = i10test(___)` additionally returns the p-values for the test statistics.

Examples

Conduct the Default Integration and Stationarity Tests

Conduct paired integration and stationarity tests on two time series using the default tests and settings.

Load the Nelson-Plosser data, and extract the series of real GNP, `GNPR`, and consumer price index, `CPI`.

```
load Data_NelsonPlosser
```

```
X = DataTable{:,{'GNPR','CPI'}};
```

`X` is a matrix containing the data for the variables `GNPR` and `CPI`.

Conduct the default integration (`adftest`) stationarity (`kpsstest`) tests on the two time series.

```
i10test(X)
```

```
Warning: Test statistic #1 above tabulated critical values:
maximum p-value = 0.999 reported.
Warning: Test statistic #1 above tabulated critical values:
minimum p-value = 0.010 reported.
Warning: Test statistic #1 above tabulated critical values:
maximum p-value = 0.999 reported.
Warning: Test statistic #1 above tabulated critical values:
minimum p-value = 0.010 reported.

Test Results

        I(1)    I(0)
======================
var1    0       1
        0.9990  0.0100
----------------------
var2    0       1
        0.9990  0.0100
----------------------
```

The warnings indicate that the p-values are very large for `adftest` and very small for `kpsstest` (that is, they are outside the Monte Carlo simulated tables). For both series, a unit root is not rejected (`H = 0` for `I(1)`), and stationarity is rejected (`H = 1` for `I(0)`).

Test for the Degree of Integration

Conduct paired integration and stationarity tests on two time series and their differences.

Load the Nelson-Plosser data, and extract the series of real GNP, `GNPR`, and consumer price index, `CPI`.

```
load Data_NelsonPlosser
X = DataTable(:,{'GNPR','CPI'});
```

X is a tabular array containing the variables GNPR and CPI.

Set the integration and stationarity test parameters.

```
I.names = {'lags','model'};
I.vals = {1,'TS'};

S.names = {'trend'};
S.vals = {true};
```

The integration test is the default (adftest), augmented with one lagged difference term and a trend-stationary alternative. The stationarity test is the default (kpsstest) with a trend.

Conduct the integration and stationarity tests on the variables and their first differences, specified using numDiffs.

```
i1Otest(X,'numDiffs',1,'itest','adf','iparams',I,...
        'stest','kpss','sparams',S)
```

```
Warning: Test statistic #1 above tabulated critical values:
minimum p-value = 0.010 reported.
Warning: Test statistic #1 below tabulated critical values:
maximum p-value = 0.100 reported.
Warning: Test statistic #1 above tabulated critical values:
minimum p-value = 0.010 reported.
Warning: Test statistic #1 below tabulated critical values:
minimum p-value = 0.001 reported.

Test Results

        I(1)    I(0)
======================
GNPR    0       1
        0.8760  0.0100

D1GNPR  1       0
        0.0054  0.1000
----------------------
CPI     0       1
        0.9799  0.0100

D1CPI   1       0
        0.0010  0.0568
----------------------
```

The warnings indicate that the p-values are very large or small for some of the tests (that is, they are outside the Monte Carlo simulated tables). For each original series, a unit root is not rejected (`H = 0` for `I(1)`), and stationarity is rejected (`H = 1` for `I(0)`). For the differenced series, a unit root is rejected and stationarity is not rejected.

At the given parameter settings, the tests suggest that both series have one degree of integration.

Return Test Results Without Display

Conduct paired integration and stationarity tests on two time series and their differences. Turn the results display off, and return the test decisions and p-values.

Load the Nelson-Plosser data, and extract the series of real GNP, `GNPR`, and consumer price index, `CPI`.

```
load Data_NelsonPlosser
X = DataTable(:,{'GNPR','CPI'});
```

`X` is a tabular array containing the variables `GNPR` and `CPI`.

Set the integration and stationarity test parameters.

```
I.names = {'lags','model'};
I.vals = {1,'TS'};

S.names = {'trend'};
S.vals = {true};
```

Conduct the integration and stationarity tests on the variables and their first differences, specified using `numDiffs`.

```
[H,PValue] = i10test(X,'numDiffs',1,'itest','adf',...
                     'iparams',I,'stest','kpss',...
                     'sparams',S,'display','off')
```

```
Warning: Test statistic #1 above tabulated critical values:
minimum p-value = 0.010 reported.
Warning: Test statistic #1 below tabulated critical values:
maximum p-value = 0.100 reported.
Warning: Test statistic #1 above tabulated critical values:
minimum p-value = 0.010 reported.
Warning: Test statistic #1 below tabulated critical values:
```

```
minimum p-value = 0.001 reported.

H =

     0     1
     1     0
     0     1
     1     0


PValue =

    0.8760    0.0100
    0.0054    0.1000
    0.9799    0.0100
    0.0010    0.0568
```

The warnings indicate that the p-values are very large or small for some of the tests (that is, they are outside the Monte Carlo simulated tables). The test decisions and p-values are stored in `H` and `PValue`, respectively.

For each original series, a unit root is not rejected (`H = 0`), and stationarity is rejected (`H = 1`), as indicated in the first and third rows of the output `H`. For each differenced series, a unit root is rejected (`H = 1`), and stationarity is not rejected (`H = 0`), as indicated in the second and fourth rows of the output `H`.

At the given parameter settings, the tests suggest that both series have one degree of integration.

- "Unit Root Tests" on page 3-44

Input Arguments

`X` — Input variables

numeric matrix | tabular array

Input variables on which to perform the stationary and integration tests, specified as a `numObs`-by-`numVars` numeric matrix or tabular array. `X` consists of `numObs` observations made on `numVars` variables.

If `X` is a tabular array, then the variables must be numeric.

Data Types: `double` | `table`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'itest','pp','numDiffs',1` specifies a Phillips-Perron integration test (and default stationarity test) on all variables and their first differences

`'varNames'` — Variable names

cell array of strings

Variable names to be used in the display, specified as the comma-separated pair consisting of `'varNames'` and a cell array of strings with `numVars` names. All variable names are truncated to the first five characters.

- If `X` is a matrix, then the default variable names are `{'var1','var2',...}`.
- If `X` is a tabular array, then the default variable names are `X.Properties.VariableNames`.

Example: `'varNames',{'CPF','AGE','BBD'}`

`'numDiffs'` — Number of differences

`0` (default) | scalar

Number of differences of each variable in `X` to test, specified as the comma-separated pair consisting of `'numDiffs'` and a scalar number.

Example: `'numDiffs',2`

`'itest'` — Integration test

`'adf'` (default) | `'pp'`

Integration test to conduct, specified as the comma-separated pair consisting of `'itest'` and one of the following:

`'adf'`	Augmented Dickey-Fuller test
`'pp'`	Phillips-Perron test

Example: `'itest','pp'`

`'iparams'` — Integration test parameters

structure

Integration test parameters, specified as the comma-separated pair consisting of `'iparams'` and a structure, `I`, with two fields, `I.names` and `I.vals`.

- `I.names` is a cell array of strings listing valid parameter names for the integration test specified in `itest`.
- `I.vals` is a cell array the same length as `I.names` containing corresponding parameter values for the parameter names in `I.names`.

If any parameters for the integration test are unspecified, then `i1Otest` uses default values. The default value for `I` is an empty structure, meaning `i1Otest` uses test defaults.

`'stest'` — Stationarity test

`'kpss'` (default) | `'lmc'`

Stationarity test to conduct, specified as the comma-separated pair consisting of `'stest'` and one of the following:

`'kpss'`	KPSS test
`'lmc'`	Leybourne-McCabe test

Example: `'stest','lmc'`

`'sparams'` — Stationarity test parameters

structure

Stationarity test parameters, specified as the comma-separated pair consisting of `'sparams'` and a structure, `S`, with two fields, `S.names` and `S.vals`.

- `S.names` is a cell array of strings listing valid parameter names for the integration test specified in `stest`.
- `S.vals` is a cell array the same length as `S.names` containing corresponding parameter values for the parameter names in `S.names`.

If any parameters for the stationarity test are unspecified, then `i1Otest` uses default values. The default value for `S` is an empty structure, meaning `i1Otest` uses test defaults.

`'display'` — Results table flag

`'on'` (default) | `'off'`

Results table flag for whether to display a results table in the Command Window, specified as the comma-separated pair consisting of `'display'` and one of `'on'` or `'off'`.

If you specify the value `'on'`, then the outputs are displayed to the Command Window in a table showing test results, `H`, and corresponding p-values, `PValue`. Rows are labeled by variable names and their differences. Columns are labeled as `I(1)` (for integration) and `I(0)` (for stationarity), respectively, indicating the null hypothesis of the tests.

Example: `'display','off'`

Output Arguments

`H` — Test decisions

matrix of logical values

Test decisions, returned as a `numVars*numDiffs+1`-by-`2` matrix of logical values. `H` equal to `1` indicates rejection of the null hypothesis in favor of the alternative. `H` equal to `0` indicates failure to reject the null hypothesis.

- Rows of `H` correspond, in order, to $x_1, \Delta x_1, \Delta^2 x_1, \ldots, \Delta^D x_1, x_2, \Delta x_2, \Delta^2 x_2, \ldots, \Delta^D x_2, \ldots$, where Δ is the differencing operator and D is the specified number of differences.
- Columns of `H` correspond to the null hypothesis of integration, `I(1)`, and the null hypothesis of stationarity, `I(0)`, respectively.

`PValue` — P-values

matrix

P-values for the tests, returned as a `numVars*numDiffs+1`-by-`2` matrix.

- Rows of `PValue` correspond, in order, to $x_1, \Delta x_1, \Delta^2 x_1, \ldots, \Delta^D x_1, x_2, \Delta x_2, \Delta^2 x_2, \ldots, \Delta^D x_2, \cdots$, where Δ is the differencing operator and D is the specified number of differences.
- Columns of `PValue` correspond to the null hypothesis of integration, `I(1)`, and the null hypothesis of stationarity, `I(0)`, respectively.

More About

Tips

- Paired integration and stationarity tests have been suggested as a method for mutual confirmation of individual test results (for example, Kwiatkowski, Phillips, Schmidt, and Shin [1]). However, on the same set of data, different integration tests might disagree, different stationarity tests might disagree, and stationarity tests might fail to confirm integration tests. Still, Monte Carlo studies (for example, Amano and van Norden [2], Burke[3]) suggest that paired testing is generally more reliable than using either type of test alone.

- "Unit Root Nonstationarity" on page 3-34

References

[1] Kwiatkowski, D., P. C. B. Phillips, P. Schmidt, and Y. Shin. "Testing the Null Hypothesis of Stationarity Against the Alternative of a Unit Root." *Journal of Econometrics*. Vol. 54, 1992, pp. 159–178.

[2] Amano, R. A., and S. van Norden. "Unit Root Tests and the Burden of Proof." Bank of Canada. Working paper 92–7, 1992.

[3] Burke, S. P. "Confirmatory Data Analysis: The Joint Application of Stationarity and Unit Root Tests." University of Reading, UK. Discussion paper 20, 1994.

See Also

`adftest` | `kpsstest` | `lmctest` | `pptest`

Introduced in R2012a

infer

Infer conditional variances of conditional variance models

Syntax

```
V = infer(Mdl,Y)
[V,logL] = infer(Mdl,Y)
[V,logL] = infer(Mdl,Y,Name,Value)
```

Description

`V = infer(Mdl,Y)` infers the conditional variances of the fully specified, univariate conditional variance model `Mdl` fit to the response data `Y`. `Mdl` can be a `garch`, `egarch`, or `gjr` model.

`[V,logL] = infer(Mdl,Y)` additionally returns the loglikelihood objective function values.

`[V,logL] = infer(Mdl,Y,Name,Value)` infers the conditional variances of `Mdl` with additional options specified by one or more `Name,Value` pair arguments. For example, you can specify presample innovations or conditional variances.

Examples

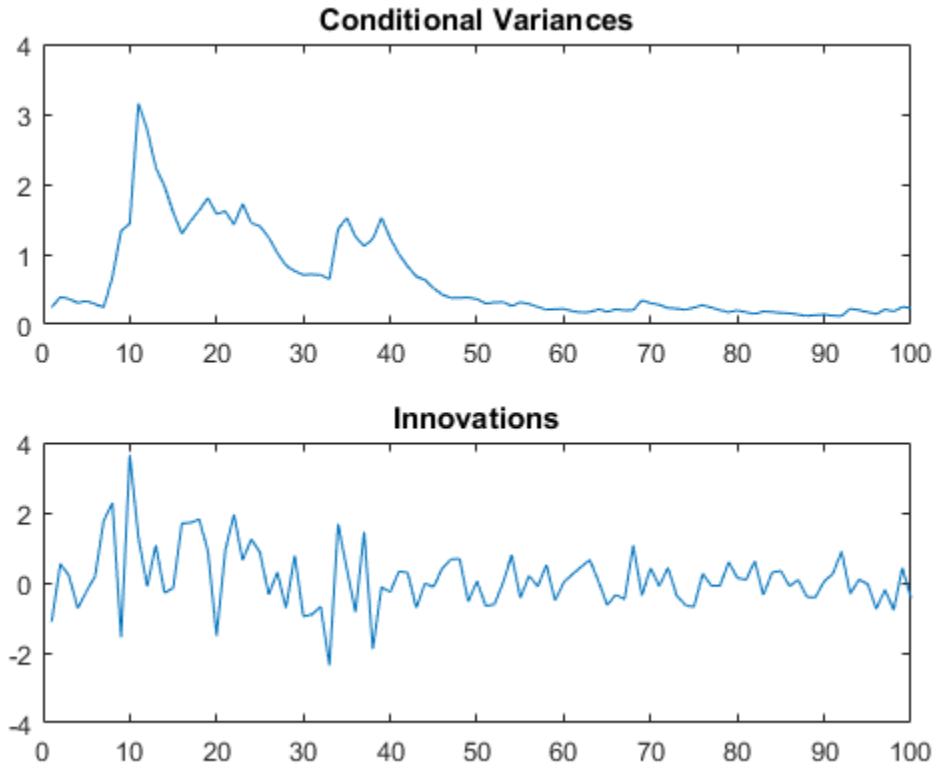
Infer GARCH Model Conditional Variances

Infer conditional variances from a GARCH(1,1) model with known coefficients. When you use, and then do not use presample data, compare the results from `infer`.

Specify a GARCH(1,1) model with known parameters. Simulate 101 conditional variances and responses (innovations) from the model. Set aside the first observation from each series to use as presample data.

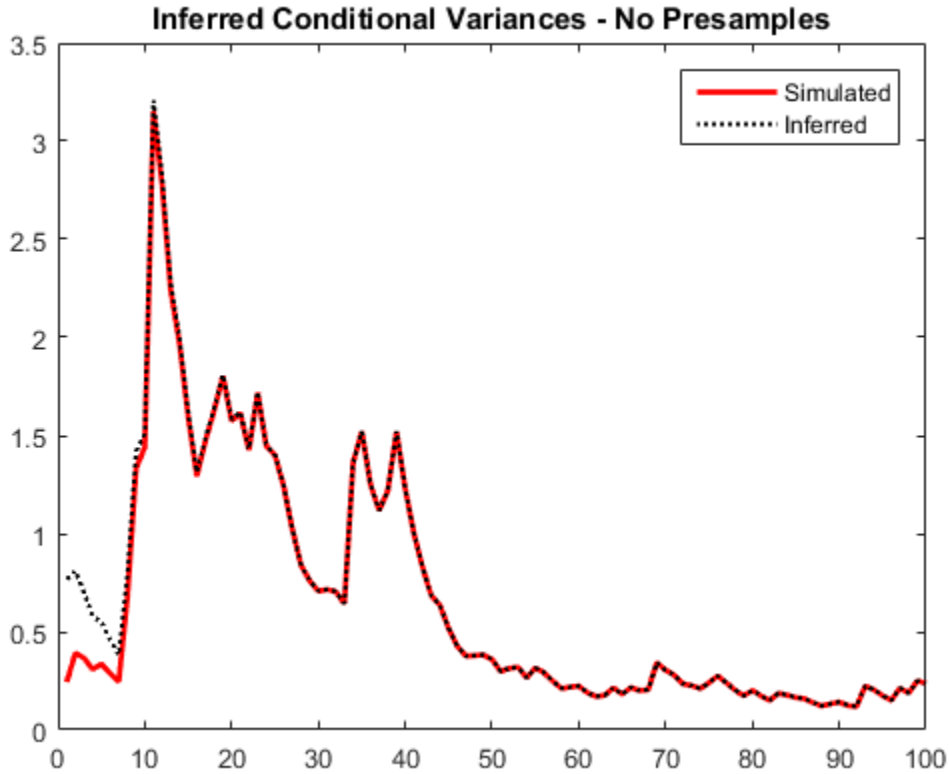
```
Mdl = garch('Constant',0.01,'GARCH',0.8,'ARCH',0.15);

rng default; % For reproducibility
[vS,yS] = simulate(Mdl,101);
y0 = yS(1);
```

```
v0 = vS(1);
y = yS(2:end);
v = vS(2:end);

figure
subplot(2,1,1)
plot(v)
title('Conditional Variances')
subplot(2,1,2)
plot(y)
title('Innovations')
```



Infer the conditional variances of `y` without using presample data. Compare them to the known (simulated) conditional variances.

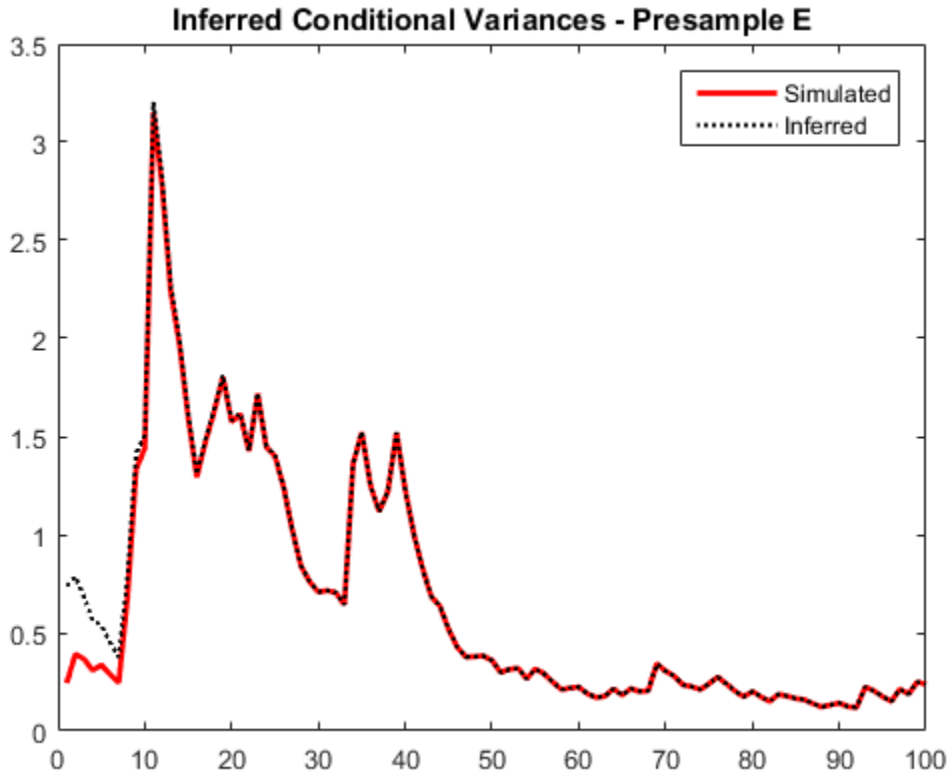
```
vI = infer(Mdl,y);

figure
plot(1:100,v,'r','LineWidth',2)
hold on
plot(1:100,vI,'k:','LineWidth',1.5)
legend('Simulated','Inferred','Location','NorthEast')
title('Inferred Conditional Variances - No Presamples')
hold off
```



Notice the transient response (discrepancy) in the early time periods due to the absence of presample data.

Infer conditional variances using the set-aside presample innovation, `y0`. Compare them to the known (simulated) conditional variances.

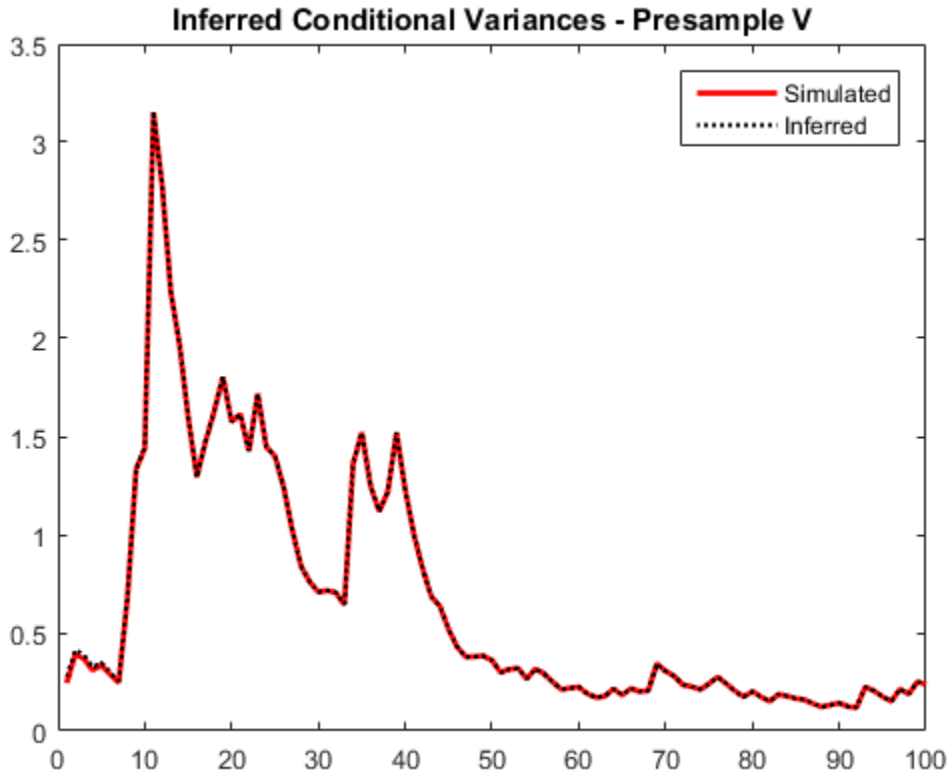
```
vE = infer(Mdl,y,'E0',y0);

figure
plot(1:100,v,'r','LineWidth',2)
hold on
plot(1:100,vE,'k:','LineWidth',1.5)
legend('Simulated','Inferred','Location','NorthEast')
title('Inferred Conditional Variances - Presample E')
hold off
```



There is a slightly reduced transient response in the early time periods.

Infer conditional variances using the set-aside presample conditional variance, `v0`. Compare them to the known (simulated) conditional variances.

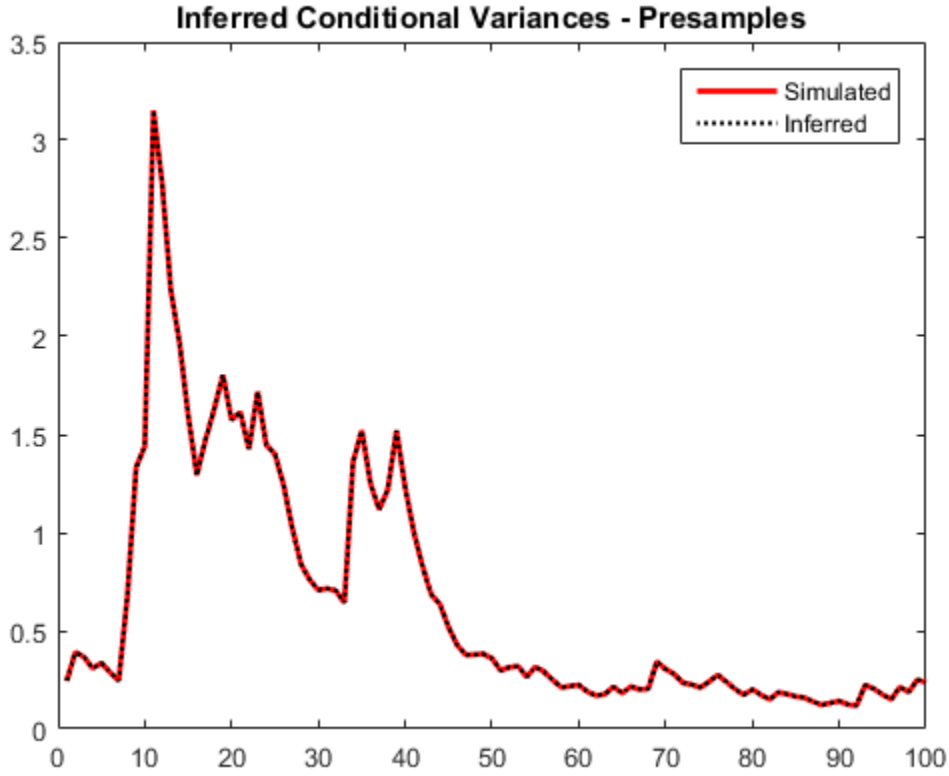
```
v0 = infer(Mdl,y,'V0',v0);

figure
plot(v)
plot(1:100,v,'r','LineWidth',2)
hold on
plot(1:100,v0,'k:','LineWidth',1.5)
legend('Simulated','Inferred','Location','NorthEast')
title('Inferred Conditional Variances - Presample V')
hold off
```



There is a much smaller transient response in the early time periods.

Infer conditional variances using both the presample innovation and conditional variance. Compare them to the known (simulated) conditional variances.

```
vE0 = infer(Mdl,y,'E0',y0,'V0',v0);

figure
plot(v)
plot(1:100,v,'r','LineWidth',2)
hold on
plot(1:100,vE0,'k:','LineWidth',1.5)
legend('Simulated','Inferred','Location','NorthEast')
title('Inferred Conditional Variances - Presamples')
hold off
```



When you use sufficient presample innovations and conditional variances, the inferred conditional variances are exact (there is no transient response).

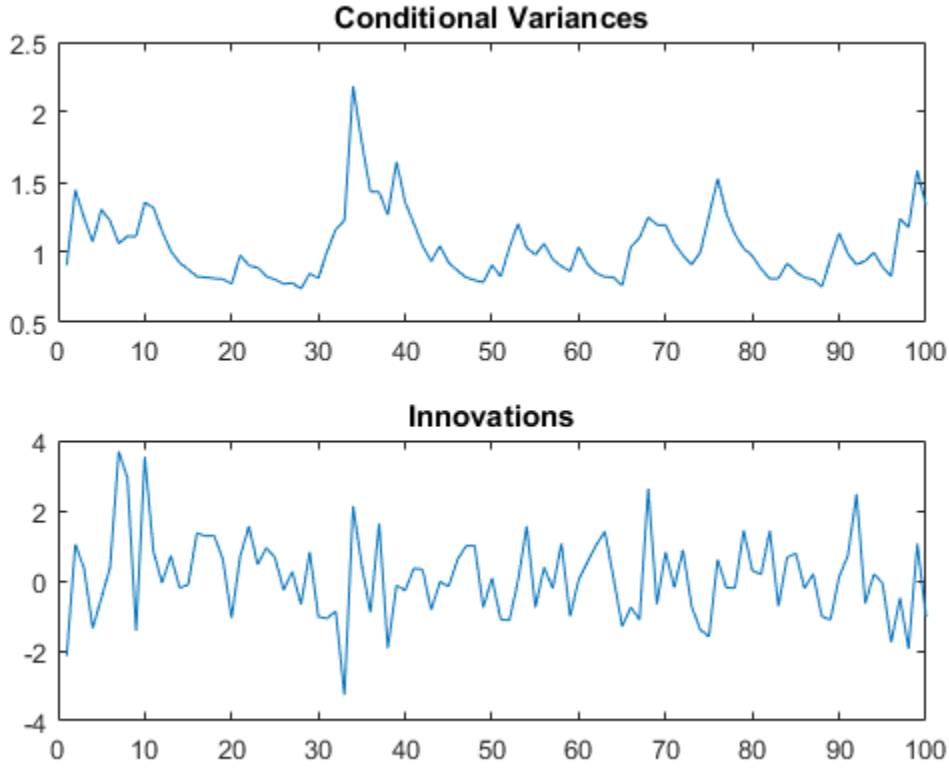
Infer EGARCH Model Conditional Variances

Infer conditional variances from an EGARCH(1,1) model with known coefficients. When you use, and then do not use presample data, compare the results from `infer`.

Specify an EGARCH(1,1) model with known parameters. Simulate 101 conditional variances and responses (innovations) from the model. Set aside the first observation from each series to use as presample data.

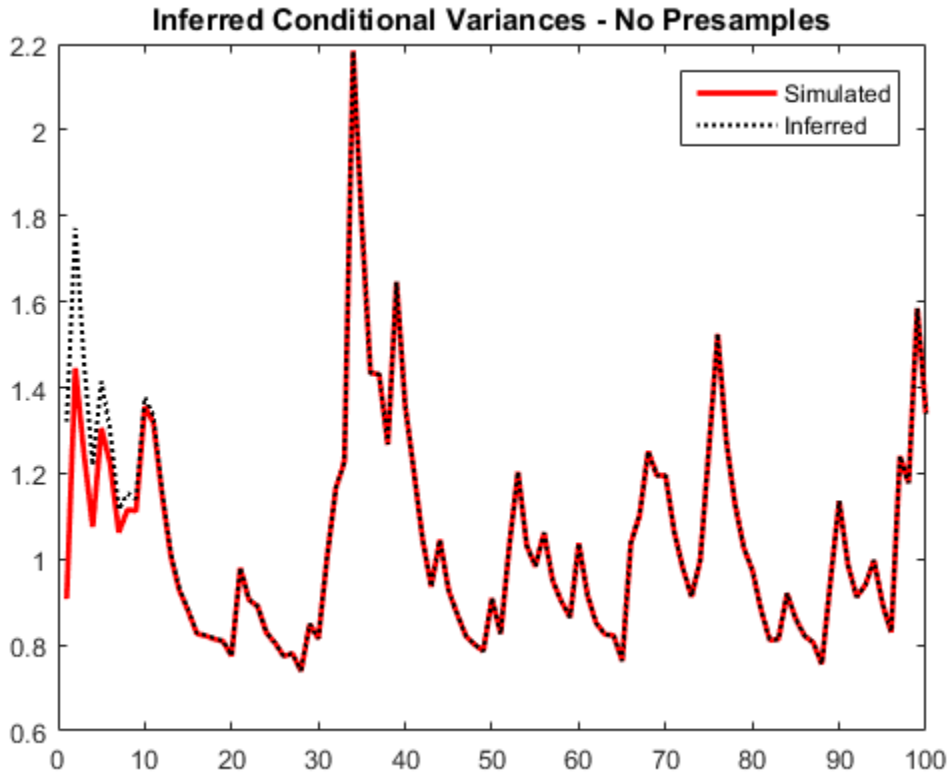
```
Mdl = egarch('Constant',0.001,'GARCH',0.8,...
               'ARCH',0.15,'Leverage',-0.1);
```

```
rng default; % For reproducibility
[vS,yS] = simulate(Mdl,101);
y0 = yS(1);
v0 = vS(1);
y = yS(2:end);
v = vS(2:end);

figure
subplot(2,1,1)
plot(v)
title('Conditional Variances')
subplot(2,1,2)
plot(y)
title('Innovations')
```



Infer the conditional variances of `y` without using any presample data. Compare them to the known (simulated) conditional variances.

```
vI = infer(Mdl,y);

figure
plot(1:100,v,'r','LineWidth',2)
hold on
plot(1:100,vI,'k:','LineWidth',1.5)
legend('Simulated','Inferred','Location','NorthEast')
title('Inferred Conditional Variances - No Presamples')
hold off
```

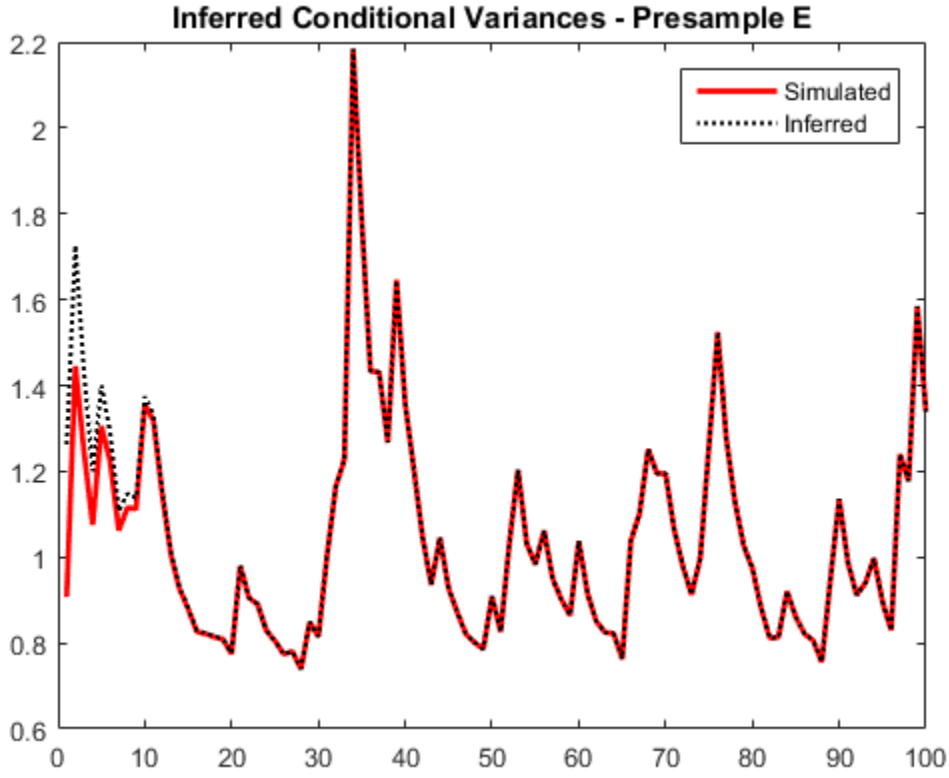


Notice the transient response (discrepancy) in the early time periods due to the absence of presample data.

Infer conditional variances using the set-aside presample innovation, `y0`. Compare them to the known (simulated) conditional variances.

```
vE = infer(Mdl,y,'E0',y0);

figure
plot(1:100,v,'r','LineWidth',2)
hold on
plot(1:100,vE,'k:','LineWidth',1.5)
legend('Simulated','Inferred','Location','NorthEast')
title('Inferred Conditional Variances - Presample E')
```

```
hold off
```

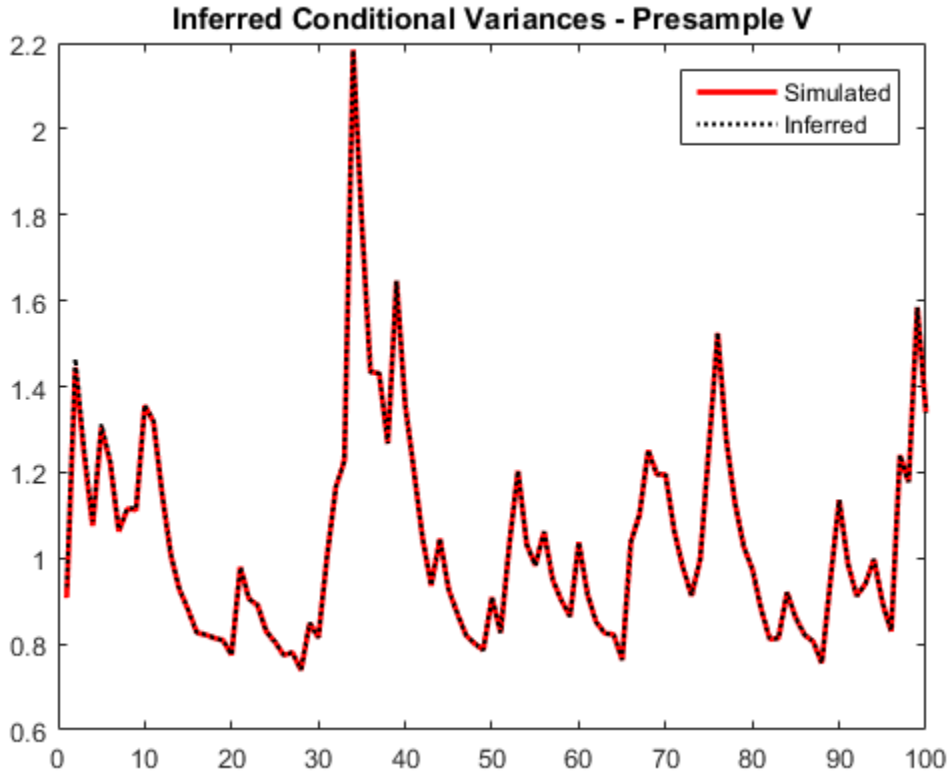


There is a slightly reduced transient response in the early time periods.

Infer conditional variances using the set-aside presample variance, `v0`. Compare them to the known (simulated) conditional variances.

```
v0 = infer(Mdl,y,'V0',v0);

figure
plot(v)
plot(1:100,v,'r','LineWidth',2)
hold on
plot(1:100,v0,'k:','LineWidth',1.5)
```

```
legend('Simulated','Inferred','Location','NorthEast')
title('Inferred Conditional Variances - Presample V')
hold off
```

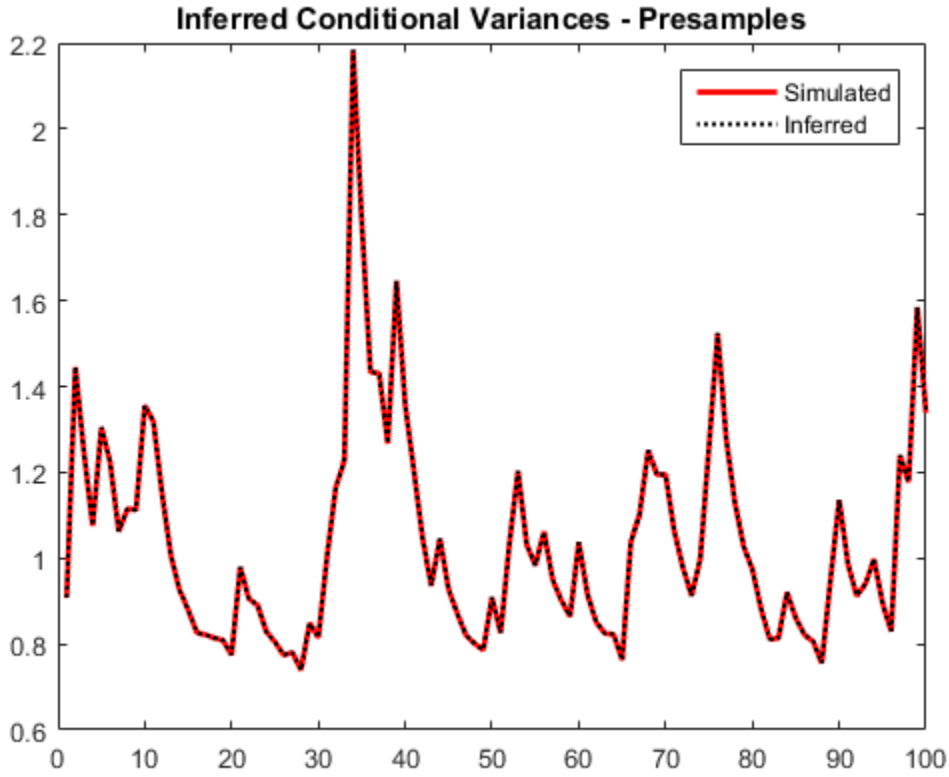


The transient response is almost eliminated.

Infer conditional variances using both the presample innovation and conditional variance. Compare them to the known (simulated) conditional variances.

```
vEO = infer(Mdl,y,'EO',y0,'V0',v0);

figure
plot(v)
plot(1:100,v,'r','LineWidth',2)
```

```
hold on
plot(1:100,vEO,'k:','LineWidth',1.5)
legend('Simulated','Inferred','Location','NorthEast')
title('Inferred Conditional Variances - Presamples')
hold off
```



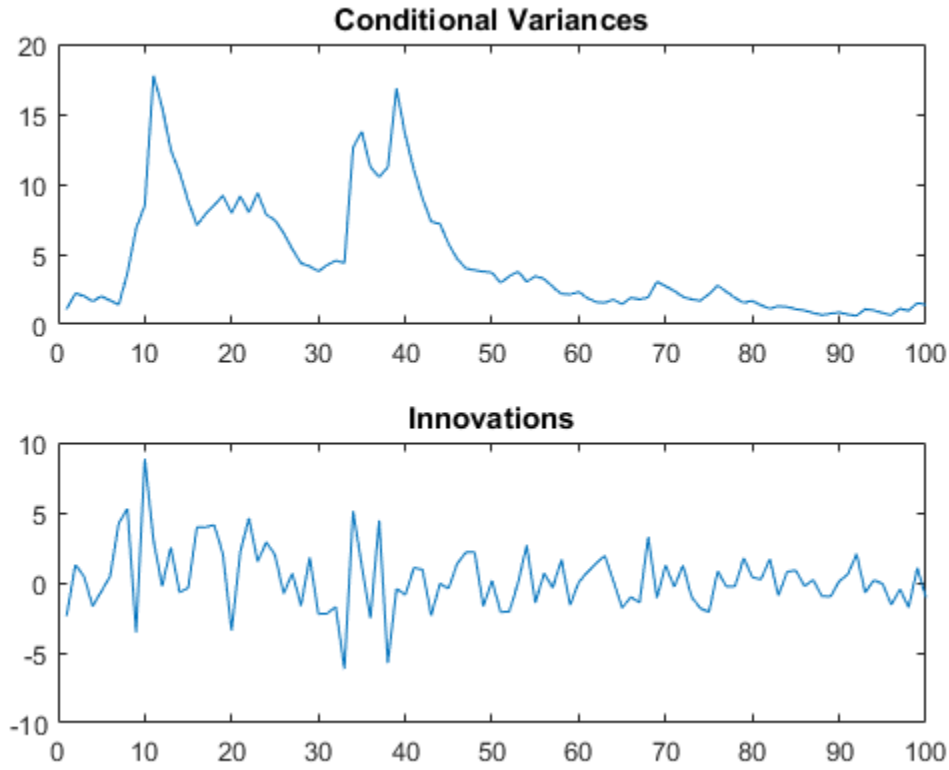
When you use sufficient presample innovations and conditional variances, the inferred conditional variances are exact (there is no transient response).

Infer GJR Model Conditional Variances

Infer conditional variances from a GJR(1,1) model with known coefficients. When you use, and then do not use presample data, compare the results from `infer`.

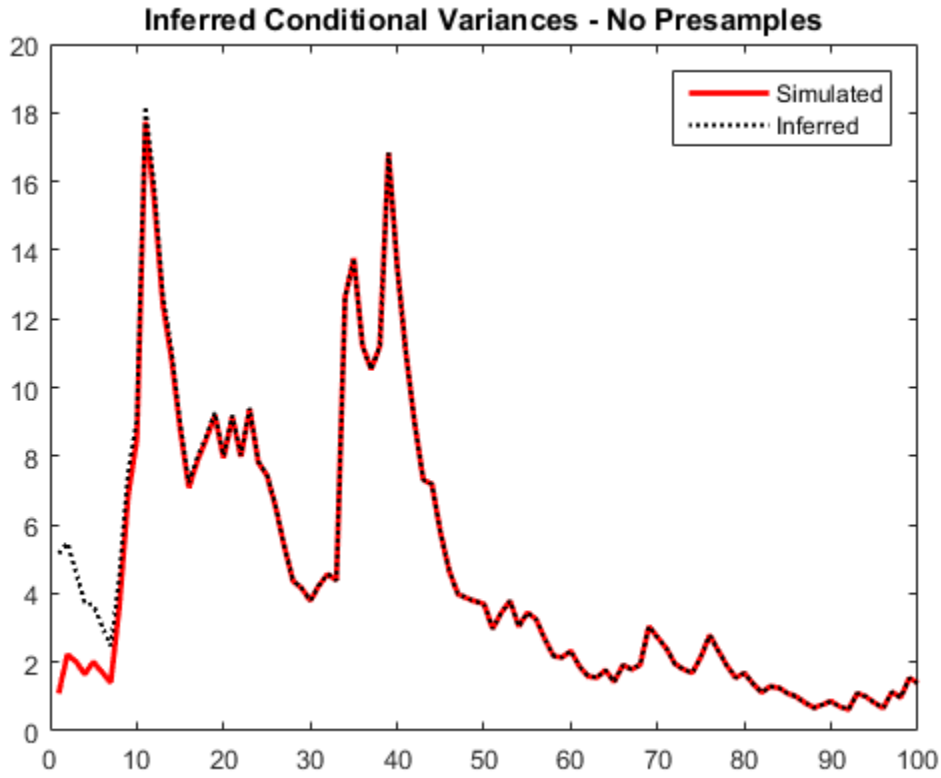
Specify a GJR(1,1) model with known parameters. Simulate 101 conditional variances and responses (innovations) from the model. Set aside the first observation from each series to use as presample data.

```
Mdl = gjr('Constant',0.01,'GARCH',0.8,'ARCH',0.14,...
            'Leverage',0.1);

rng default; % For reproducibility
[vS,yS] = simulate(Mdl,101);
y0 = yS(1);
v0 = vS(1);
y = yS(2:end);
v = vS(2:end);

figure
subplot(2,1,1)
plot(v)
title('Conditional Variances')
subplot(2,1,2)
plot(y)
title('Innovations')
```



Infer the conditional variances of `y` without using any presample data. Compare them to the known (simulated) conditional variances.

```
vI = infer(Mdl,y);

figure
plot(1:100,v,'r','LineWidth',2)
hold on
plot(1:100,vI,'k:','LineWidth',1.5)
legend('Simulated','Inferred','Location','NorthEast')
title('Inferred Conditional Variances - No Presamples')
hold off
```

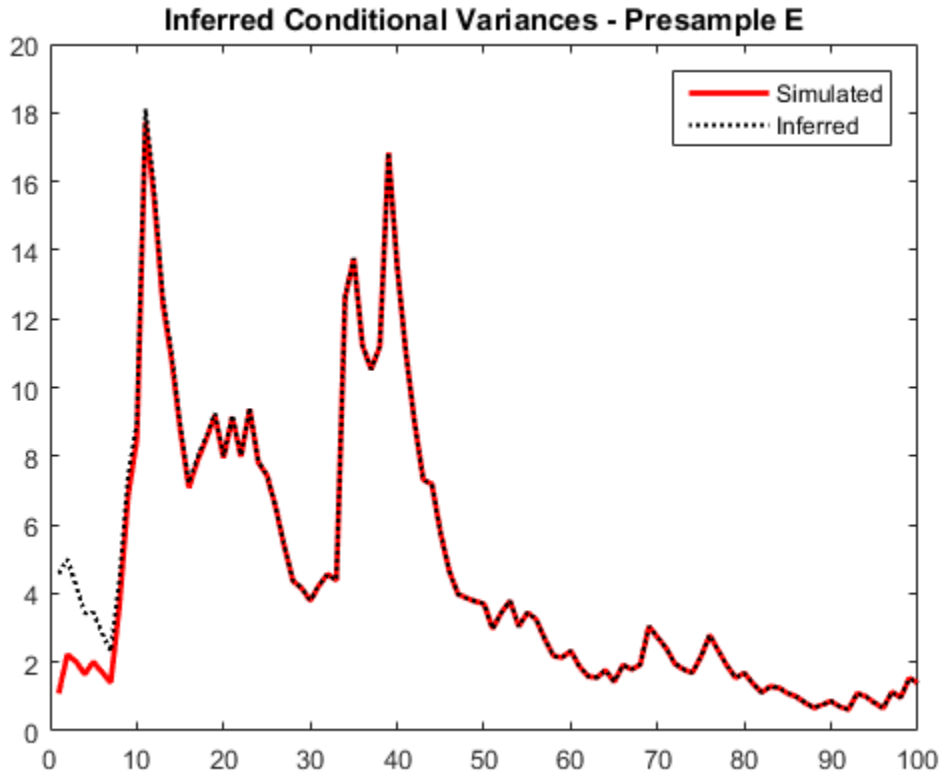


Notice the transient response (discrepancy) in the early time periods due to the absence of presample data.

Infer conditional variances using the set-aside presample innovation, `y0`. Compare them to the known (simulated) conditional variances.

```
vE = infer(Mdl,y,'E0',y0);

figure
plot(1:100,v,'r','LineWidth',2)
hold on
plot(1:100,vE,'k:','LineWidth',1.5)
legend('Simulated','Inferred','Location','NorthEast')
title('Inferred Conditional Variances - Presample E')
```

```
hold off
```

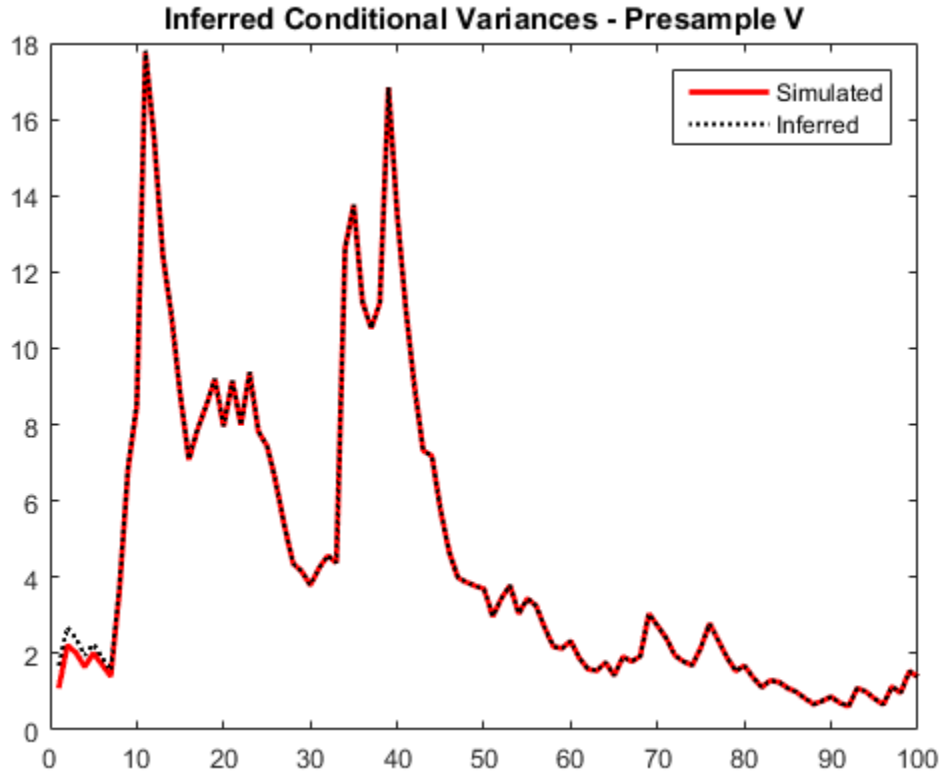


There is a slightly reduced transient response in the early time periods.

Infer conditional variances using the set-aside presample conditional variance, `v0`. Compare them to the known (simulated) conditional variances.

```
v0 = infer(Mdl,y,'V0',v0);

figure
plot(v)
plot(1:100,v,'r','LineWidth',2)
hold on
plot(1:100,v0,'k:','LineWidth',1.5)
```

```
legend('Simulated','Inferred','Location','NorthEast')
title('Inferred Conditional Variances - Presample V')
hold off
```

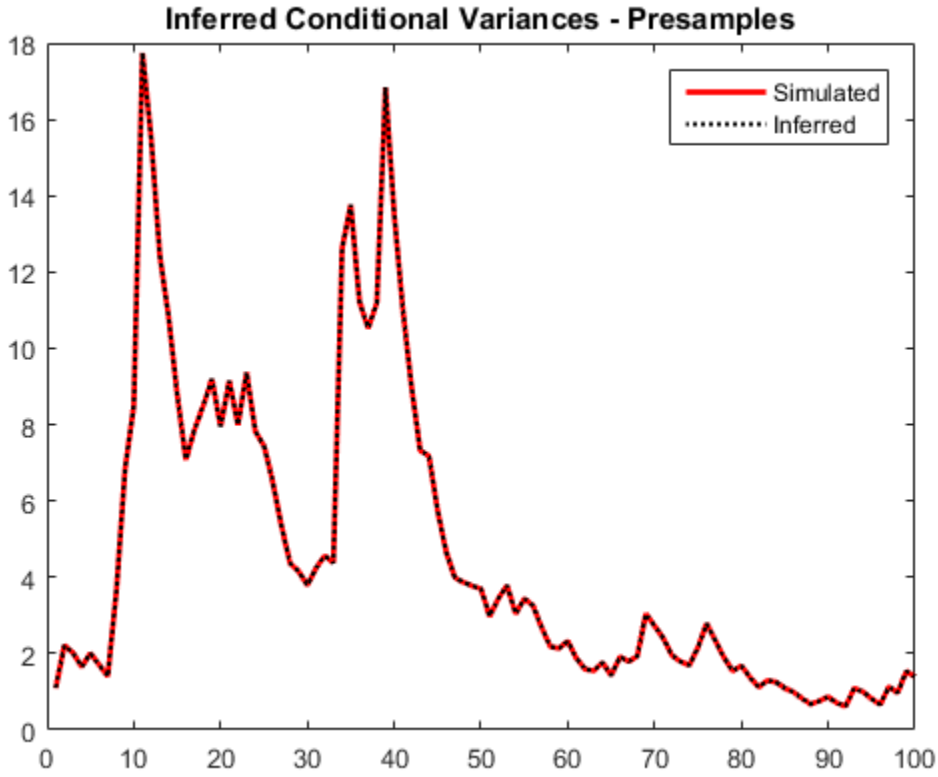


There is a much smaller transient response in the early time periods.

Infer conditional variances using both the presample innovation and conditional variance. Compare them to the known (simulated) conditional variances.

```
vEO = infer(Mdl,y,'EO',y0,'V0',v0);

figure
plot(v)
plot(1:100,v,'r','LineWidth',2)
```

```
hold on
plot(1:100,vEO,'k:','LineWidth',1.5)
legend('Simulated','Inferred','Location','NorthEast')
title('Inferred Conditional Variances - Presamples')
hold off
```



When you use sufficient presample innovations and conditional variances, the inferred conditional variances are exact (there is no transient response).

Conduct Likelihood Ratio Test for EGARCH Fit Comparison

Infer the loglikelihood objective function values for an EGARCH(1,1) and EGARCH(2,1) model fit to NASDAQ Composite Index returns. To identify which model is the more parsimonious, adequate fit, conduct a likelihood ratio test.

Load the NASDAQ data included with the toolbox, and convert the index to returns. Set aside the first two observations to use as presample data.

```
load Data_EquityIdx
nasdaq = DataTable.NASDAQ;
r = price2ret(nasdaq);
r0 = r(1:2);
rn = r(3:end);
```

Fit an EGARCH(1,1) model to the returns, and infer the loglikelihood objective function value.

```
Mdl1 = egarch(1,1);
EstMdl1 = estimate(Mdl1,rn,'E0',r0);
[~,logL1] = infer(EstMdl1,rn,'E0',r0);
```

```
    EGARCH(1,1) Conditional Variance Model:
    --------------------------------------
    Conditional Probability Distribution: Gaussian

                               Standard          t
     Parameter       Value       Error       Statistic
    -----------   -----------   ------------   -----------
     Constant      -0.135179     0.0221338       -6.10736
     GARCH{1}       0.983864    0.00242682        405.413
      ARCH{1}       0.199966     0.0139933        14.2902
  Leverage{1}     -0.0602428    0.00565582       -10.6515
```

Fit an EGARCH(2,1) model to the returns, and infer the loglikelihood objective function value.

```
Mdl2 = egarch(2,1);
EstMdl2 = estimate(Mdl2,rn,'E0',r0);
[~,logL2] = infer(EstMdl2,rn,'E0',r0);
```

```
    EGARCH(2,1) Conditional Variance Model:
    --------------------------------------
    Conditional Probability Distribution: Gaussian

                               Standard          t
     Parameter       Value       Error       Statistic
    -----------   -----------   ------------   -----------
     Constant      -0.145595     0.0284356       -5.12017
```

```
     GARCH{1}       0.853073       0.140183        6.08542
     GARCH{2}       0.129516       0.138377        0.93596
      ARCH{1}       0.219686      0.0294646        7.45596
  Leverage{1}      -0.0679354      0.0108795       -6.24435
```

Conduct a likelihood ratio test, with the more parsimonious EGARCH(1,1) model as the null model, and the EGARCH(2,1) model as the alternative. The degree of freedom for the test is 1, because the EGARCH(2,1) model has one more parameter than the EGARCH(1,1) model (an additional GARCH term).

```
[h,p] = lratiotest(logL2,logL1,1)
```

```
h =

     0


p =

    0.2256
```

The null hypothesis is not rejected (`h = 0`). At the 0.05 significance level, the EGARCH(1,1) model is not rejected in favor of the EGARCH(2,1) model.

Conduct Likelihood Ratio Test for GARCH and GJR Fit Comparison

A GARCH(*P*, *Q*) model is nested within a GJR(*P*, *Q*) model. Therefore, you can perform a likelihood ratio test to compare GARCH(*P*, *Q*) and GJR(*P*, *Q*) model fits.

Infer the loglikelihood objective function values for a GARCH(1,1) and GJR(1,1) model fit to NASDAQ Composite Index returns. Conduct a likelihood ratio test to identify which model is the more parsimonious, adequate fit.

Load the NASDAQ data included with the toolbox, and convert the index to returns. Set aside the first two observations to use as presample data.

```
load Data_EquityIdx
nasdaq = DataTable.NASDAQ;
r = price2ret(nasdaq);
r0 = r(1:2);
rn = r(3:end);
```

Fit a GARCH(1,1) model to the returns, and infer the loglikelihood objective function value.

```
Mdl1 = garch(1,1);
EstMdl1 = estimate(Mdl1,rn,'E0',r0);
[~,logL1] = infer(EstMdl1,rn,'E0',r0);
```

```
    GARCH(1,1) Conditional Variance Model:
    ----------------------------------------
    Conditional Probability Distribution: Gaussian

                              Standard          t
     Parameter       Value      Error       Statistic
    -----------   -----------   ------------   -----------
     Constant    2.00502e-06   5.42982e-07        3.69262
     GARCH{1}       0.883327    0.00845366         104.49
      ARCH{1}       0.109239    0.00766663        14.2486
```

Fit a GJR(1,1) model to the returns, and infer the loglikelihood objective function value.

```
Mdl2 = gjr(1,1);
EstMdl2 = estimate(Mdl2,rn,'E0',r0);
[~,logL2] = infer(EstMdl2,rn,'E0',r0);
```

```
    GJR(1,1) Conditional Variance Model:
    --------------------------------------
    Conditional Probability Distribution: Gaussian

                              Standard          t
     Parameter       Value      Error       Statistic
    -----------   -----------   ------------   -----------
     Constant    2.47525e-06   5.69837e-07        4.34379
     GARCH{1}       0.881015    0.00951049        92.6361
      ARCH{1}      0.0640148    0.00918489        6.96958
  Leverage{1}      0.0892972    0.00992115        9.00069
```

Conduct a likelihood ratio test, with the more parsimonious GARCH(1,1) model as the null model, and the GJR(1,1) model as the alternative. The degree of freedom for the test is 1, because the GJR(1,1) model has one more parameter than the GARCH(1,1) model (a leverage term).

```
[h,p] = lratiotest(logL2,logL1,1)
```

```
h =

     1


p =

   4.5816e-10
```

The null hypothesis is rejected (`h = 1`). At the 0.05 significance level, the GARCH(1,1) model is rejected in favor of the GJR(1,1) model.

- "Infer Conditional Variances and Residuals" on page 6-77
- "Compare Conditional Variance Models Using Information Criteria" on page 6-87

Input Arguments

`Mdl` — Conditional variance model

`garch` model object | `egarch` model object | `gjr` model object

Conditional variance model without any unknown parameters, specified as a `garch`, `egarch`, or `gjr` model object.

`Mdl` cannot contain any properties that have `NaN` value.

`Y` — Response data

numeric column vector | numeric matrix

Response data, specified as a numeric column vector or matrix.

As a column vector, `Y` represents a single path of the underlying series.

As a matrix, the rows of `Y` correspond to periods and the columns correspond to separate paths. The observations across any row occur simultaneously.

`infer` infers the conditional variances of `Y`. `Y` usually represents an innovation series with mean 0 and variances characterized by `Mdl`. It is the continuation of the presample innovation series `E0`. `Y` can also represent a time series of innovations with mean 0 plus

an offset. If `Mdl` has a nonzero offset, then the software stores its value in the `Offset` property (`Mdl.Offset`).

`infer` assumes that observations across any row occur simultaneously.

The last observation of any series is the latest observation.

Note: `NaN`s indicate missing values. `infer` removes missing values. `infer` uses list-wise deletion to remove any `NaN`s. Removing `NaN`s in the data reduces the sample size. Removing missing values, can also create irregular time series.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'E0',[1 1;0.5 0.5],'V0',[1 0.5;1 0.5]` specifies two equivalent presample paths of innovations and two, different presample paths of conditional variances.

`'E0'` — Presample innovations

numeric column vector | numeric matrix

Presample innovations, specified as the comma-separated pair consisting of `'E0'` and a numeric column vector or matrix. The presample innovations provide initial values for the innovations process of the conditional variance model `Mdl`, and derive from a distribution with mean 0.

`E0` must contain at least `Mdl.Q` elements or rows. If `E0` contains extra rows, then `infer` uses the latest `Mdl.Q` only.

The last element or row contains the latest presample innovation.

- If `E0` is a column vector, it represents a single path of the underlying innovation series. `infer` applies `E0` to each inferred path.
- If `E0` is a matrix, then each column represents a presample path of the underlying innovation series. `E0` must have at least as many columns as `Y`. If `E0` has more columns than necessary, `infer` uses the first `size(Y,2)` columns only.

The defaults are:

- For GARCH(P,Q) and GJR(P,Q) models, `infer` sets any necessary presample innovations to the square root of the average squared value of the offset-adjusted response series `Y`.

For EGARCH(P,Q) models, `infer` sets any necessary presample innovations to zero.

Example: `'E0',[1 1;0.5 0.5]`

Data Types: `double`

`'V0'` — Presample conditional variances

numeric column vector with positive entries | numeric matrix with positive entries

Presample conditional variances, specified as the comma-separated pair consisting of `'V0'` and a numeric column vector or matrix with positive entries. `V0` provides initial values for the conditional variances in the model.

- If `V0` is a column vector, then `infer` applies it to each output path.
- If `V0` is a matrix, then each column represents a presample path of conditional variances. `V0` must have at least as many columns as `Y`. If `V0` has more columns than required, `infer` uses the first `size(Y,2)` columns only.
- For GARCH(P,Q) and GJR(P,Q) models, `V0` must have at least `Mdl.P` rows (or elements) to initialize the variance equation.
- For EGARCH(P,Q) models, `V0` must have at least `max(Mdl.P,Mdl.Q)` rows to initialize the variance equation.

If the number of rows in `V0` exceeds the necessary number, then `infer` uses the latest, required number of observations only.

The last element row contains the latest observation.

By default, `infer` sets any necessary observations to the average squared value of the offset-adjusted response series `Y`.

Example: `'V0',[1 0.5;1 0.5]`

Data Types: `double`

Notes:

- NaNs indicate missing values. `infer` removes missing values. The software merges the presample data (`E0` and `V0`) separately from the input response data (`Y`), and then

  uses list-wise deletion to remove any rows containing at least one `NaN`. Removing `NaN`s in the data reduces the sample size. Removing missing values can also create irregular time series.
- `infer` assumes that you synchronize presample data such that the latest observation of each presample series occurs simultaneously.
- If you do not specify `E0` and `V0`, then `infer` derives the necessary presample observations from the unconditional, or long-run, variance of the offset-adjusted response process.
  - For all conditional variance models, `V0` is the sample average of the squared disturbances of the offset-adjusted response data `Y`.
  - For GARCH(P,Q) and GJR(P,Q) models, `E0` is the square root of the average squared value of the offset-adjusted response series `Y`.
  - For EGARCH(P,Q) models, `E0` is `0`.

  These specifications minimize initial transient effects.

Output Arguments

`V` — Conditional variances

numeric column vector | numeric matrix

Conditional variances inferred from the response data `Y`, returned as a numeric column vector or matrix.

The dimensions of `V` and `Y` are equivalent. If `Y` is a matrix, then the columns of `V` are the inferred conditional variance paths corresponding to the columns of `Y`.

Rows of `V` are periods corresponding to the periodicity of `Y`.

`logL` — Loglikelihood objective function values

scalar | numeric vector

Loglikelihood objective function values associated with the model `Mdl`, returned as a scalar or numeric vector.

If `Y` is a vector, then `logL` is a scalar. Otherwise, `logL` is vector of length `size(Y,2)`, and each element is the loglikelihood of the corresponding column (or path) in `Y`.

Data Types: `double`

More About

- Using garch Objects
- Using egarch Objects
- Using gjr Objects

References

[1] Bollerslev, T. "Generalized Autoregressive Conditional Heteroskedasticity." *Journal of Econometrics*. Vol. 31, 1986, pp. 307–327.

[2] Bollerslev, T. "A Conditionally Heteroskedastic Time Series Model for Speculative Prices and Rates of Return." *The Review of Economics and Statistics*. Vol. 69, 1987, pp. 542–547.

[3] Box, G. E. P., G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis: Forecasting and Control*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

[4] Enders, W. *Applied Econometric Time Series*. Hoboken, NJ: John Wiley & Sons, 1995.

[5] Engle, R. F. "Autoregressive Conditional Heteroskedasticity with Estimates of the Variance of United Kingdom Inflation." *Econometrica*. Vol. 50, 1982, pp. 987–1007.

[6] Glosten, L. R., R. Jagannathan, and D. E. Runkle. "On the Relation between the Expected Value and the Volatility of the Nominal Excess Return on Stocks." *The Journal of Finance*. Vol. 48, No. 5, 1993, pp. 1779–1801.

[7] Hamilton, J. D. *Time Series Analysis*. Princeton, NJ: Princeton University Press, 1994.

See Also

`egarch` | `estimate` | `filter` | `forecast` | `garch` | `gjr` | `print` | `simulate`

Introduced in R2012a

impulse

Class: arima

Impulse response function

Syntax

```
impulse(Mdl)
impulse(Mdl,numObs)
Y = impulse( ___ )
```

Description

`impulse(Mdl)` plots a discrete stem plot of the impulse response function for the univariate ARIMA model, `Mdl`, in the current figure window.

`impulse(Mdl,numObs)` plots the impulse response function for `numObs` periods.

`Y = impulse( ___ )` returns the impulse response in a column vector for any of the previous input arguments.

Tips

- To improve performance of the filtering algorithm, specify the number of observations to include in the impulse response, `numObs`. When you do not specify `numObs`, `impulse` computes the impulse response by converting the input model to a truncated, infinite-degree, moving average representation using the relatively slow lag operator polynomial division algorithm. This results in an impulse response of generally unknown length.

Input Arguments

`Mdl`

ARIMA model, as created by arima or estimate.

numObs

Positive integer indicating the number of observations to include in the impulse response (the number of periods for which `impulse` computes the impulse response).

When you specify `numObs`, `impulse` computes the impulse response by filtering a unit impulse followed by a vector of zeros of appropriate length. The filtering algorithm is very fast and results in an impulse response of known length.

If you do not specify `numObs`, `impulse` determines the number of observation using the polynomial division algorithm of the underlying lag operator polynomials, mldivide.

Output Arguments

Y

Column vector of impulse responses. If you specify `numObs`, then `Y` is `numObs`-by-1. If you do not specify `numObs`, the underlying lag operator polynomial division algorithm returns an impulse response of generally unknown length.

Definitions

Impulse Response Function

The *impulse response function* for a univariate ARIMA process is the dynamic response of the system to a single impulse, or innovation shock, of unit size. The specific impulse response calculated by `impulse` is the dynamic multiplier, defined as the partial derivative of the output response with respect to an innovation shock at time zero.

For a univariate ARIMA process, y_t, and innovation series ε_t, the impulse response at time j, Ψ_j, is given by

$$\psi_j = \frac{\partial y_j}{\partial \varepsilon_0}.$$

Expressed as a function of time, the sequence of dynamic multipliers, Ψ_1, Ψ_2,..., measures the sensitivity of the process to a purely transitory change in the innovation process. `impulse` computes the impulse response function by shocking the system with a unit impulse $\varepsilon_0 = 1$, with all past observations of y_t and all future shocks of ε_t set to zero. Because the impulse response function is the partial derivative of the ARIMA process with respect to an innovation shock at time 0, the presence of a constant in the model has no effect on the output.

This impulse response is sometimes called the *forecast error impulse response*, because the innovations, ε_t, can be interpreted as the one-step-ahead forecast errors.

Examples

Plot an Impulse Response Function

Specify the AR(2) model,

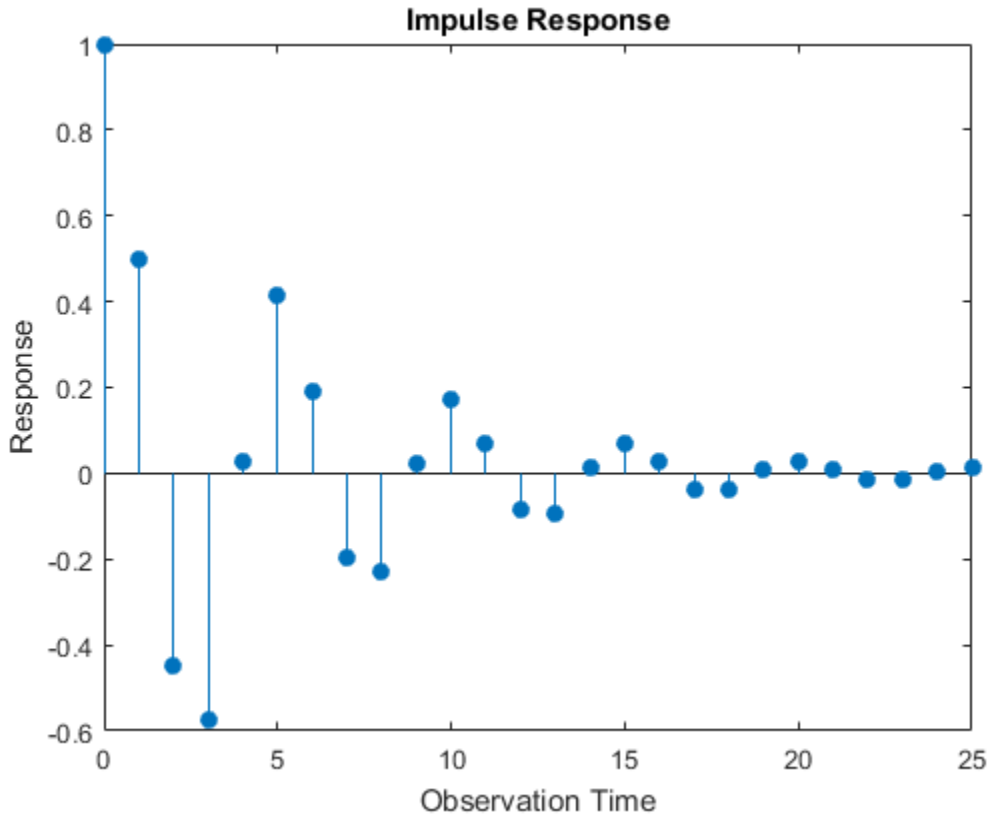
$$y_t = 0.5y_{t-1} - 0.7y_{t-2} + \varepsilon_t.$$

```
Mdl = arima('AR',{0.5,-0.7},'Constant',0)
```

```
Mdl =

    ARIMA(2,0,0) Model:
    --------------------
    Distribution: Name = 'Gaussian'
               P: 2
               D: 0
               Q: 0
        Constant: 0
              AR: {0.5 -0.7} at Lags [1 2]
             SAR: {}
              MA: {}
             SMA: {}
        Variance: NaN
```

Plot the impulse response function without specifying the number of observations.

```
impulse(Mdl)
```



The model is stationary; the impulse response function decays in a sinusoidal pattern.

Store an Impulse Response Function

Specify the ARMA(1,1) model,

$$y_t = 0.7y_{t-1} + \varepsilon_t + 0.2\varepsilon_{t-1}.$$

```
Mdl = arima('AR',0.7,'MA',0.2,'Constant',0)
```

```
Mdl =
```

```
ARIMA(1,0,1) Model:
--------------------
Distribution: Name = 'Gaussian'
           P: 1
           D: 0
           Q: 1
    Constant: 0
          AR: {0.7} at Lags [1]
         SAR: {}
          MA: {0.2} at Lags [1]
         SMA: {}
    Variance: NaN
```

Store the impulse response function for 15 periods.

```
Y = impulse(Mdl,15)
```

```
Y =

    1.0000
    0.9000
    0.6300
    0.4410
    0.3087
    0.2161
    0.1513
    0.1059
    0.0741
    0.0519
    0.0363
    0.0254
    0.0178
    0.0125
    0.0087
```

When you specify the number of observations, you know the length of the output impulse response series.

- "Plot the Impulse Response Function" on page 5-88

References

[1] Box, G. E. P., G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis: Forecasting and Control* 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

[2] Enders, W. *Applied Econometric Time Series*. Hoboken, NJ: John Wiley & Sons, 1995.

[3] Hamilton, J. D. *Time Series Analysis*. Princeton, NJ: Princeton University Press, 1994.

[4] Lütkepohl, H. *New Introduction to Multiple Time Series Analayis*. New York, NY: Springer-Verlag, 2007.

See Also

arima | estimate | filter | forecast | infer | print | simulate

More About

- "Impulse Response Function" on page 5-86

impulse

Class: regARIMA

Impulse response of regression model with ARIMA errors

Syntax

```
impulse(Mdl)
impulse(Mdl,numObs)
Y = impulse( ___ )
```

Description

`impulse(Mdl)` plots a discrete stem plot of the impulse response function for the regression model with ARIMA time series errors, `Mdl`, in the current figure window.

`impulse(Mdl,numObs)` plots the impulse response function for `numObs` periods.

`Y = impulse( ___ )` returns the impulse response in a column vector for any of the previous input arguments.

Tips

- To improve performance of the filtering algorithm, specify the number of observations, `numObs`, to include in the impulse response.

Input Arguments

`Mdl`

Regression model with ARIMA errors, as created by regARIMA or estimate.

`numObs`

Number of observations to include in the impulse response, specified as a positive integer. `numObs` is the number of periods for which `impulse` computes the impulse response.

Default: `impulse` determines the number of observations using the polynomial division algorithm of the underlying lag operator polynomials, mldivide.

Output Arguments

`Y`

Impulse responses of the model `Mdl`, specified as a column vector.

- If you specify `numObs`, then `Y` is `numObs`-by-1.
- If you do not specify `numObs`, the underlying lag operator polynomial division algorithm returns an impulse response of generally unknown length.

Definitions

Impulse Response Function

The *impulse response function* for regression models with ARIMA errors is the dynamic response of the system to a single impulse, or innovation shock, of unit size. The specific impulse response calculated by `impulse` is the *dynamic multiplier*, defined as the partial derivative of the output response with respect to an innovation shock at time 0.

For a regression model with ARIMA errors, y_t, unconditional disturbances u_t, and innovation series ε_t, the impulse response at time j, Ψ_j, is given by

$$\psi_j = \frac{\partial y_j}{\partial \varepsilon_0} = \frac{\partial u_j}{\partial \varepsilon_0}.$$

Expressed as a function of time, the sequence of dynamic multipliers, Ψ_1, Ψ_2,..., measures the sensitivity of the process to a purely transitory change in the innovation process. `impulse` computes the impulse response function by shocking the system with a unit impulse $\varepsilon_0 = 1$, with all past observations of y_t and all future shocks of ε_t set to 0. The impulse response function is the partial derivative of the ARIMA process with respect to an innovation shock at time 0. Because of this, the presence of an intercept or a regression component corresponding to predictors in the model has no effect on the output.

This impulse response is sometimes called the *forecast error impulse response*, because the innovations, ε_t, can be interpreted as the one-step-ahead forecast errors.

Examples

Plot an Impulse Response Function

Specify the following regression model with ARMA(2,1) errors:

$$\begin{aligned} y_t &= X_t \begin{bmatrix} 0.1 \\ -0.2 \end{bmatrix} + u_t \\ u_t &= 0.5u_{t-1} - 0.8u_{t-2} + \varepsilon_t - 0.5\varepsilon_{t-1}, \end{aligned}$$

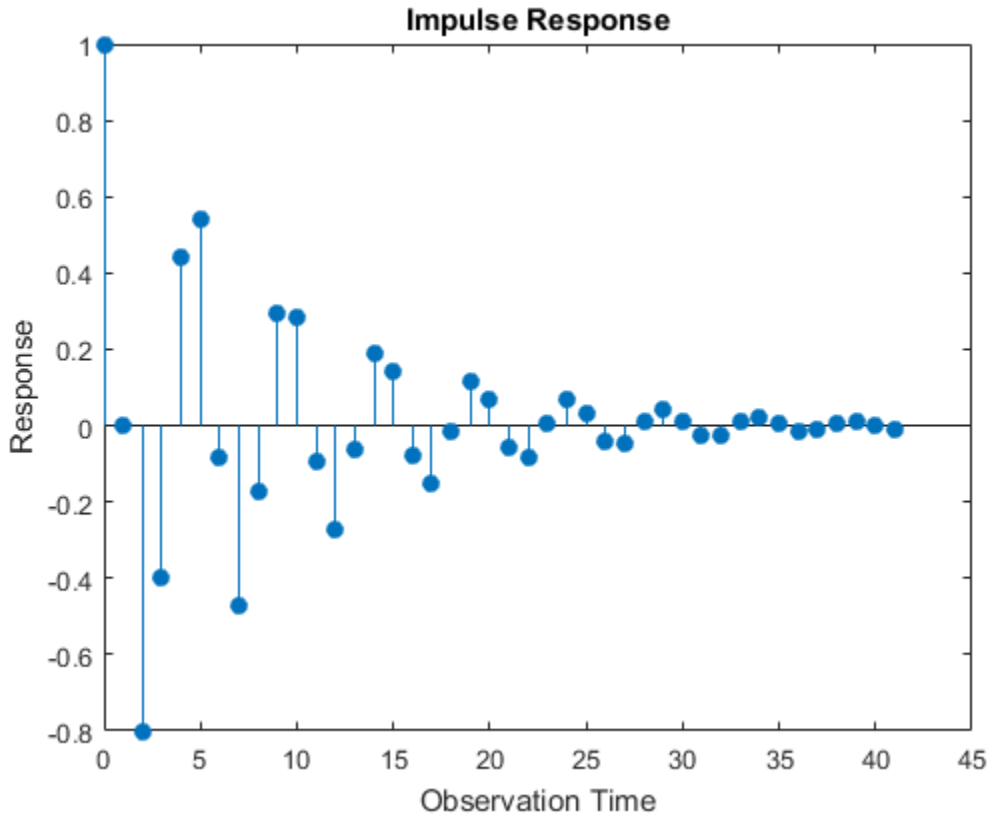
where ε_t is Gaussian with variance 0.1.

```
Mdl = regARIMA('Intercept',0, 'AR', {0.5 -0.8}, ...
    'MA',-0.5,'Beta',[0.1 -0.2], 'Variance',0.1);
```

Time the calculation of and plot the impulse response function without specifying the number of observations.

```
tic
impulse(Mdl)
toc
```

```
Elapsed time is 0.164070 seconds.
```

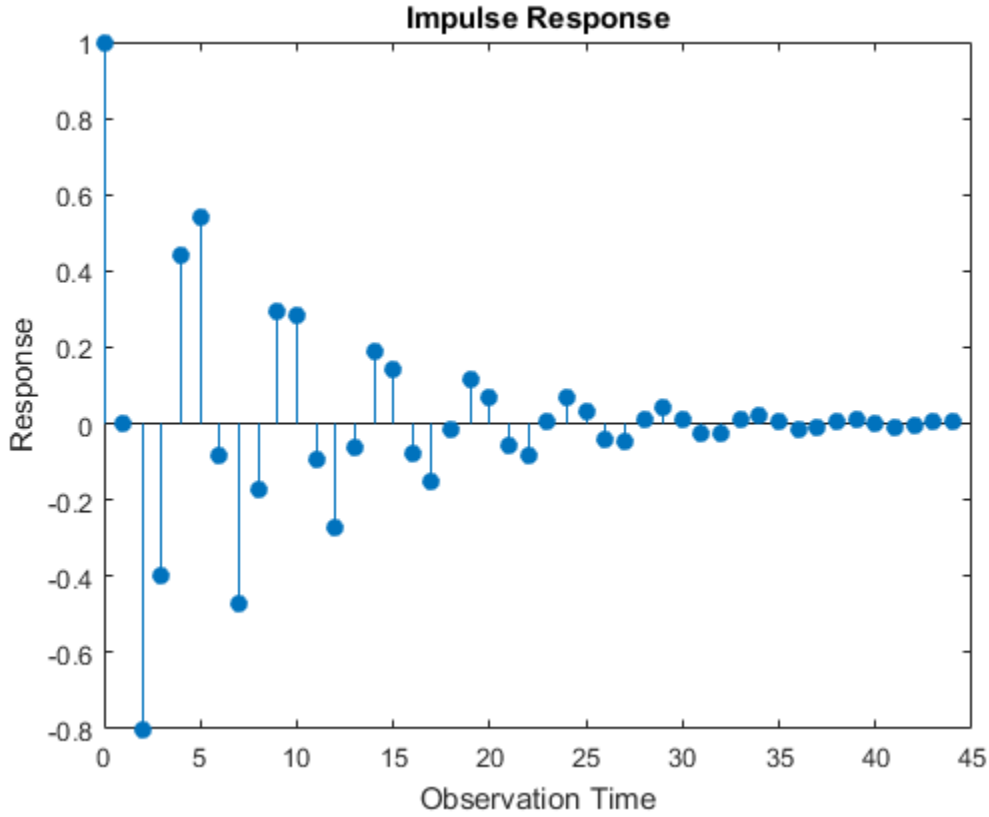


The model is stationary; the impulse response function decays in a sinusoidal pattern.

Time the calculation of and plot the impulse response function using 45 observations.

```
tic
impulse(Mdl,45)
toc
```

```
Elapsed time is 0.089573 seconds.
```



There are more observations represented in this plot than the one generated in the previous step. However, the impulse response function and the plot took less time to generate in this step than the previous. This is because the software did not calculate the impulse response function using an infinite-degree moving average as in the previous step.

Store an Impulse Response Function

Specify the following regression model with ARMA(2,1) errors:

$$y_t = X_t \begin{bmatrix} 0.1 \\ -0.2 \end{bmatrix} + u_t$$
$$u_t = 0.5u_{t-1} - 0.8u_{t-2} + \varepsilon_t - 0.5\varepsilon_{t-1},$$

where ε_t is Gaussian with variance 0.1.

```
Mdl = regARIMA('Intercept',0, 'AR', {0.5 -0.8}, ...
    'MA',-0.5,'Beta',[0.1 -0.2], 'Variance',0.1);
```

Store the impulse response function for 15 periods.

```
Y = impulse(Mdl,15)
```

```
Y =

    1.0000
         0
   -0.8000
   -0.4000
    0.4400
    0.5400
   -0.0820
   -0.4730
   -0.1709
    0.2930
    0.2832
   -0.0928
   -0.2729
   -0.0623
    0.1872
```

The length of the output impulse response series is `numObs`.

- "Plot the Impulse Response of regARIMA Models" on page 4-77

Algorithms

- If you specify the number of observations, `numObs`, `impulse` computes the impulse response by filtering a unit shock followed by an appropriate length vector of 0s. The filtering algorithm is very fast and results in an impulse response of known (`numObs`) length.
- If you do not specify `numObs`, then `impulse` converts the error model to a truncated, infinite-degree moving average using the relatively slow lag operator polynomial division algorithm. This produces an impulse response of generally unknown length.

References

[1] Box, G. E. P., G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis: Forecasting and Control* 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

[2] Enders, W. *Applied Econometric Time Series*. Hoboken, NJ: John Wiley & Sons, 1995.

[3] Hamilton, J. D. *Time Series Analysis*. Princeton, NJ: Princeton University Press, 1994.

[4] Lütkepohl, H. *New Introduction to Multiple Time Series Analayis*. New York, NY: Springer-Verlag, 2007.

See Also

filter | regARIMA | simulate

More About

- "Impulse Response for Regression Models with ARIMA Errors" on page 4-75

infer

Class: arima

Infer ARIMA or ARIMAX model residuals or conditional variances

Syntax

```
[E,V] = infer(Mdl,Y)
[E,V,logL] = infer(Mdl,Y)
[E,V,logL] = infer(Mdl,Y,Name,Value)
```

Description

`[E,V] = infer(Mdl,Y)` infers residuals and conditional variances of a univariate ARIMA model fit to data `Y`.

`[E,V,logL] = infer(Mdl,Y)` additionally returns the loglikelihood objective function values.

`[E,V,logL] = infer(Mdl,Y,Name,Value)` infers the ARIMA or ARIMAX model residuals and conditional variances, and returns the loglikelihood objective function values, with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

`Mdl` — ARIMA or ARIMAX model

`arima` model

ARIMA or ARIMAX model, specified as an arima model returned by `arima` or estimate.

The properties of `Mdl` cannot contain `NaN`s.

`Y` — Response data

numeric column vector | numeric matrix

Response data, specified as a numeric column vector or numeric matrix. If `Y` is a matrix, then it has `numObs` observations and `numPaths` rows.

`infer` infers the residuals and variances of `Y`. `Y` represents the time series characterized by `Mdl`, and it is the continuation of the presample series `Y0`.

- If `Y` is a column vector, then it represents one path of the underlying series.
- If `Y` is a matrix, then it represents `numObs` observations of `numPaths` paths of an underlying time series.

`infer` assumes that observations across any row occur simultaneously. The last observation of any series is the latest.

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'E0'` — Presample innovations

`0` (default) | numeric column vector | numeric matrix

Presample innovations that have mean 0 and provide initial values for the model, specified as the comma-separated pair consisting of `'E0'` and a numeric column vector or numeric matrix.

`E0` must contain at least `numPaths` columns and enough rows to initialize the ARIMA model and any conditional variance model. That is, `E0` must contain at least `Mdl.Q` innovations, but can be greater if you use a conditional variance model. If the number of rows in `E0` exceeds the number necessary, then `infer` only uses the latest observations. The last row contains the latest observation.

If the number of columns exceeds `numPaths`, then `infer` only uses the first `numPaths` columns. If `E0` is a column vector, then `infer` applies it to each inferred path.

Data Types: `double`

`'V0'` — Presample conditional variances

numeric column vector | numeric matrix

Presample conditional variances providing initial values for any conditional variance model, specified as the comma-separated pair consisting of `'V0'` and a numeric column vector or matrix with positive entries.

V0 must contain at least numPaths columns and enough rows to initialize the variance model. If the number of rows in V0 exceeds the number necessary, then infer only uses the latest observations. The last row contains the latest observation.

If the number of columns exceeds numPaths, then infer only uses the first numPaths columns. If V0 is a column vector, then infer applies it to each inferred path.

By default, infer sets the necessary observations to the unconditional variance of the conditional variance process.

Data Types: double

'X' — Exogenous predictors

numeric matrix

Exogenous predictors in the regression model, specified as the comma-separated pair consisting of 'X' and a matrix.

The columns of X are separate, synchronized time series, with the last row containing the latest observations.

If you do not specify Y0, then the number of rows of X must be at least size(Y,2) + Mdl.P. Otherwise, the number of rows of X should be at least numel(Y,2). In either case, if the number of rows of X exceeds the number necessary, then infer only uses the latest observations.

By default, the conditional mean model does not have a regression coefficient.

Data Types: double

'Y0' — Presample response data

numeric column vector | numeric matrix

Presample response data that provides initial values for the model, specified as the comma-separated pair consisting of 'Y' and a numeric column vector or numeric matrix. Y0 must contain at least Mdl.P rows and numPaths columns. If the number of rows in Y0 exceeds Mdl.P, then infer only uses the latest Mdl.P observations. The last row contains the latest observation. If the number of columns exceeds numPaths, then infer only uses the first numPaths columns. If Y0 is a column vector, then infer applies it to each inferred path.

By default, infer backcasts to obtain the necessary observations.

Data Types: `double`

Notes

- `NaN`s indicate missing values and `infer` removes them. The software merges the presample data and main data sets separately, then uses list-wise deletion to remove any `NaN`s. That is, `infer` sets `PreSample` = `[Y0 E0 V0]` and `Data` = `[Y X]`, then it removes any row in `PreSample` or `Data` that contains at least one `NaN`.
- The removal of `NaN`s in the main data reduces the effective sample size. Such removal can also create irregular time series.
- `infer` assumes that you synchronize the response and predictor series such that the latest observation of each occurs simultaneously. The software also assumes that you synchronize the presample series similarly.
- The software applies all exogenous series in `X` to each response series in `Y`.

Output Arguments

`E` — Inferred residuals

numeric matrix

Inferred residuals, returned as a numeric matrix. `E` has `numObs` rows and `numPaths` columns.

`V` — Inferred conditional variances

numeric matrix

Inferred conditional variances, returned as a numeric matrix. `V` has `numObs` rows and `numPaths` columns.

`logL` — Loglikelihood objective function values

numeric vector

Loglikelihood objective function values associated with the model `Mdl`, returned as a numeric vector. `logL` has `numPaths` elements associated with the corresponding path in `Y`.

Data Types: `double`

Examples

Infer Residuals

Infer residuals from an AR model.

Specify an AR(2) model using known parameters.

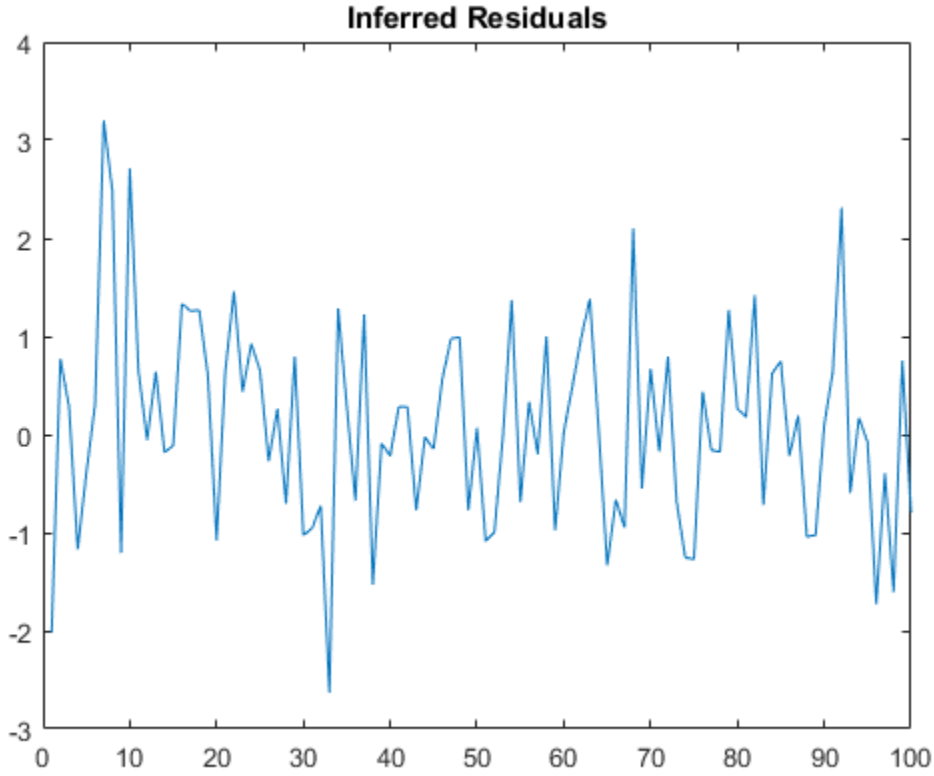
```
Mdl = arima('AR',{0.5,-0.8},'Constant',0.002,...
 'Variance',0.8);
```

Simulate response data with 102 observations.

```
rng 'default';
Y = simulate(Mdl,102);
```

Use the first two responses as presample data, and infer residuals for the remaining 100 observations.

```
E = infer(Mdl,Y(3:end),'Y0',Y(1:2));
figure;
plot(E);
title 'Inferred Residuals';
```



Infer Conditional Variances

Infer the conditional variances from an AR(1) and GARCH(1,1) composite model.

Specify an AR(1) model using known parameters. Set the variance equal to a `garch` model.

```
Mdl = arima('AR',{0.8,-0.3},'Constant',0);
MdlVar = garch('Constant',0.0002,'GARCH',0.6,...
 'ARCH',0.2);
Mdl.Variance = MdlVar;
```

Simulate response data with 102 observations.

```
rng 'default';
```

```
Y = simulate(Mdl,102);
```

Infer conditional variances for the last 100 observations without using presample data.

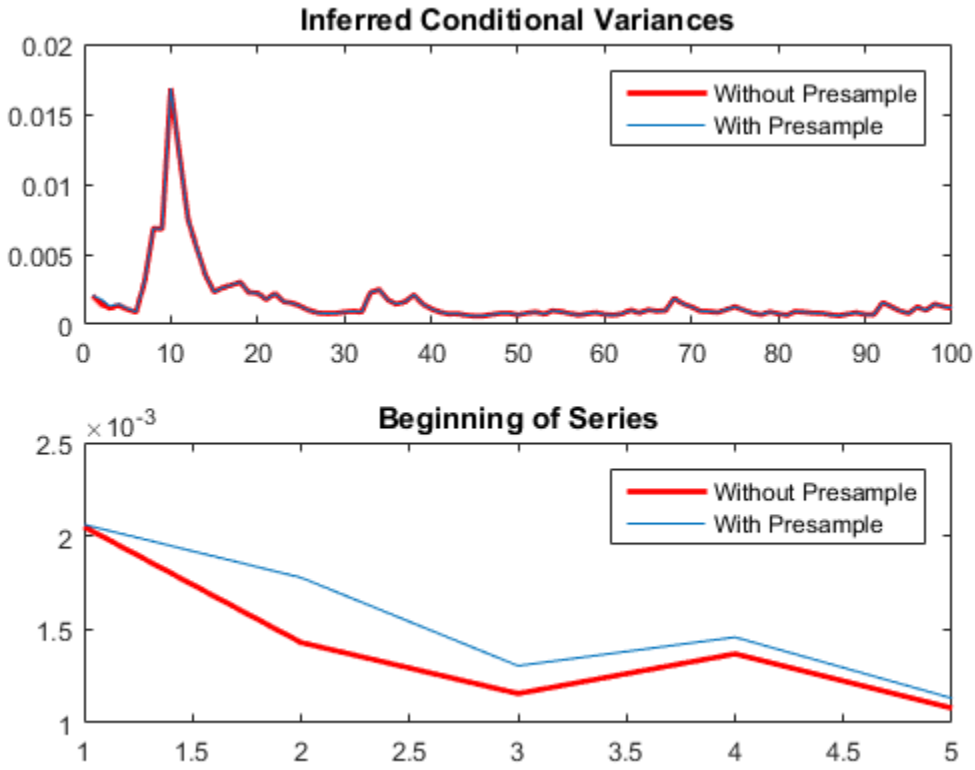
```
[Ew,Vw] = infer(Mdl,Y(3:end));
```

Infer conditional variances for the last 100 observations using the first two observations as presample data.

```
[E,V] = infer(Mdl,Y(3:end),'Y0',Y(1:2));
```

Plot the two sets of conditional variances for comparison. Examine the first few observations to see the slight difference between the series at the beginning.

```
figure;
subplot(2,1,1);
plot(Vw,'r','LineWidth',2);
hold on;
plot(V);
legend('Without Presample','With Presample');
title 'Inferred Conditional Variances';
hold off

subplot(2,1,2);
plot(Vw(1:5),'r','LineWidth',2);
hold on;
plot(V(1:5));
legend('Without Presample','With Presample');
title 'Beginning of Series';
hold off
```



Infer Residuals Using Predictor Data

Infer residuals from an ARMAX model.

Specify an ARMA(1,2) model using known parameters for the response (`MdlY`) and an AR(1) model for the predictor data (`MdlX`).

```
MdlY = arima('AR',0.2,'MA',{-0.1,0.6},'Constant',...
    1,'Variance',2,'Beta',3);
MdlX = arima('AR',0.3,'Constant',0,'Variance',1);
```

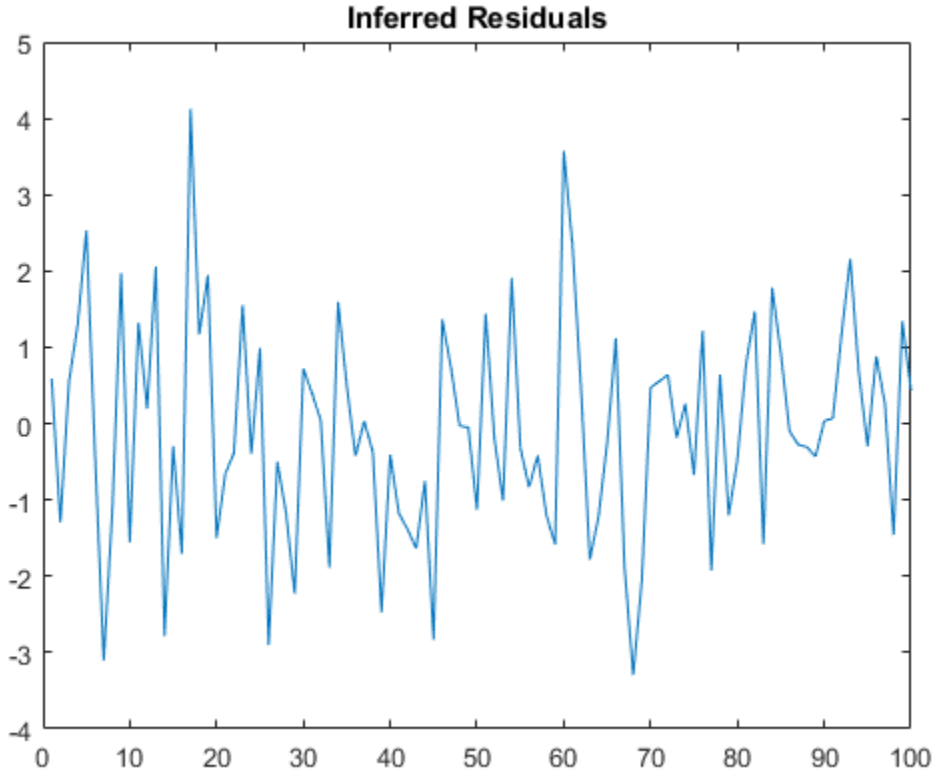
Simulate response and predictor data with 102 observations.

```
rng 'default'; % random number seed to duplicate data
```

```
X = simulate(MdlX,102);
Y = simulate(MdlY,102,'X',X);
```

Use the first two responses as presample data, and infer residuals for the remaining 100 observations.

```
E = infer(MdlY,Y(3:end),'Y0',Y(1:2),'X',X);
figure;
plot(E);
title 'Inferred Residuals';
```



- “Infer Residuals for Diagnostic Checking” on page 5-140

References

[1] Box, G. E. P., G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis: Forecasting and Control* 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

[2] Enders, W. *Applied Econometric Time Series*. Hoboken, NJ: John Wiley & Sons, 1995.

[3] Hamilton, J. D. *Time Series Analysis*. Princeton, NJ: Princeton University Press, 1994.

See Also

arima | estimate | filter | forecast | impulse | print | simulate

More About

- "Residual Diagnostics" on page 3-90

infer

Class: regARIMA

Infer innovations of regression models with ARIMA errors

Syntax

```
E = infer(Mdl,Y)
[E,U,V,logL] = infer(Mdl,Y)
[E,U,V,logL] = infer(Mdl,Y,Name,Value)
```

Description

`E = infer(Mdl,Y)` infers residuals of a univariate regression model with ARIMA time series errors fit to response data `Y`.

`[E,U,V,logL] = infer(Mdl,Y)` additionally returns the unconditional disturbances `U`, the innovation variances `V`, and the loglikelihood objective function values `logL`.

`[E,U,V,logL] = infer(Mdl,Y,Name,Value)` returns the output arguments using additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

`Mdl` — Regression model with ARIMA errors

`regARIMA` model

Regression model with ARIMA errors, specified as a regARIMA model returned by `regARIMA` or estimate.

The properties of `Mdl` cannot contain `NaN`s.

`Y` — Response data

numeric column vector | numeric matrix

Response data, specified as a numeric column vector or numeric matrix. If `Y` is a matrix, then it has `numObs` observations and `numPaths` rows.

`infer` infers the residuals (estimated innovations) and unconditional disturbances of `Y`. `Y` represents the time series characterized by `Mdl`, and it is the continuation of the presample series `Y0`.

- If `Y` is a column vector, then it represents one path of the underlying series.
- If `Y` is a matrix, then it represents `numObs` observations of `numPaths` paths of an underlying time series.

`infer` assumes that observations across any row occur simultaneously. The last observation of any series is the latest.

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'E0'` — Presample innovations

numeric column vector | numeric matrix

Presample innovations that have mean 0 and provide initial values for the ARIMA error model, specified as the comma-separated pair consisting of `'E0'` and a numeric column vector or numeric matrix.

- If `E0` is a column vector, then it is applied to each inferred path.
- If `E0` is a matrix, then it requires at least `numPaths` columns. If `E0` contains more columns than required, then `infer` uses the first `numPaths` columns.
- `E0` must contain at least `Mdl.Q` rows. If `E0` contains extra rows, then `infer` uses the latest presample innovations. The last row contains the latest presample innovation.

By default, `infer` sets the necessary observations to 0.

Data Types: `double`

`'U0'` — Presample unconditional disturbances

numeric column vector | numeric matrix

Presample unconditional disturbances that provide initial values for the ARIMA error model, specified as the comma-separated pair consisting of `'U0'` and a numeric column vector or numeric matrix.

- If `U0` is a column vector, then it is applied to each inferred path.
- If `U0` is a matrix, then it requires at least `numPaths` columns. If `U0` contains more columns than required, then `infer` uses the first `numPaths` columns.
- `U0` must contain at least `Mdl.P` rows. If `U0` contains extra rows, then `infer` uses the latest presample unconditional disturbances. The last row contains the latest presample unconditional disturbance.

By default, `infer` backcasts for the necessary presample unconditional disturbances.

Data Types: `double`

`'X'` — Predictor data

numeric matrix

Predictor data in the regression model, specified as the comma-separated pair consisting of `'X'` and a numeric matrix.

The columns of `X` are separate, synchronized time series, with the last row containing the latest observations. The number of rows of `X` should be at least the length of `Y`. If the number of rows of `X` exceeds the number required, then `infer` uses the latest observations.

By default, `infer` does not include a regression component in the model regardless of the presence of regression coefficients in `Mdl`.

Data Types: `double`

Notes

- `NaN`s in `Y`, `X`, `E0`, and `U0` indicate missing values and `infer` removes them. The software merges the presample data sets (`E0` and `U0`), then uses list-wise deletion to remove any `NaN`s. `infer` similarly removes `NaN`s from the effective sample data (`X` and `Y`). Removing `NaN`s in the data reduces the sample size, and can also create irregular time series.
- `infer` assumes that you synchronize presample data such that the latest observation of each presample series occurs simultaneously.

- All predictors (that is, columns in `X`) are associated with each response path in `Y`.
- `V` is equal to the variance in `Mdl`.

Output Arguments

`E` — Inferred residuals

numeric matrix

Inferred residuals (estimated innovations of the unconditional disturbances), returned as a numeric matrix. `E` has `numObs` rows and `numPaths` columns.

Data Types: `double`

`U` — Inferred unconditional disturbances

numeric matrix

Inferred unconditional disturbances, returned as a numeric matrix. `U` has `numObs` rows and `numPaths` columns.

Data Types: `double`

`V` — Inferred variances

numeric matrix

Inferred variances, returned as a numeric matrix. `V` has `numObs` rows and `numPaths` columns.

Data Types: `double`

`logL` — Loglikelihood objective function values

numeric vector

Loglikelihood objective function values associated with the model `Mdl`, returned as a numeric vector. `logL` has `numPaths` elements associated with the corresponding path in `Y`.

Data Types: `double`

Examples

Infer Residuals from a Regression Model with ARIMA Errors

Forecast responses from the following regression model with ARMA(2,1) errors over a 30-period horizon:

$$y_t = X_t \begin{bmatrix} 0.1 \\ -0.2 \end{bmatrix} + u_t$$
$$u_t = 0.5u_{t-1} - 0.8u_{t-2} + \varepsilon_t - 0.5\varepsilon_{t-1},$$

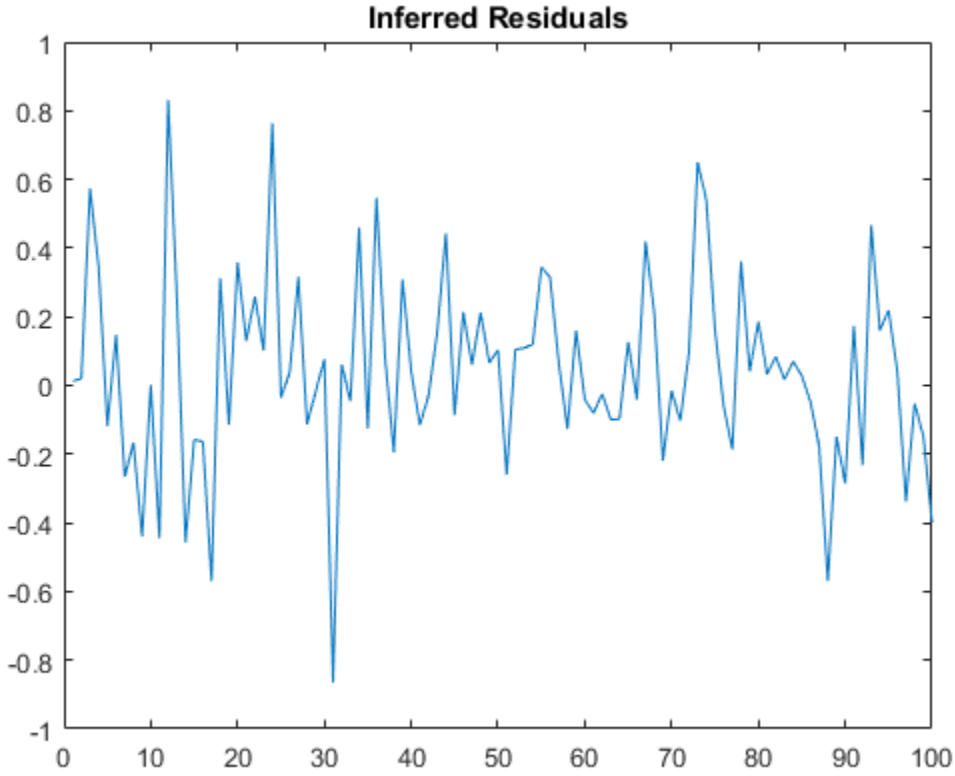
where ε_t is Gaussian with variance 0.1.

Specify the regression model with ARIMA errors. Simulate responses from the model and two predictor series.

```
Mdl = regARIMA('Intercept', 0, 'AR', {0.5 -0.8}, ...
    'MA',-0.5,'Beta',[0.1 -0.2], 'Variance',0.1);
rng(1); % For reproducibility
X =  randn(100,2);
y = simulate(Mdl,100,'X',X);
```

Infer, and then plot residuals. By default, `infer` backcasts for the necessary presample unconditional disturbances.

```
e = infer(Mdl,y,'X',X);

figure
plot(e)
title('Inferred Residuals')
```



Regress the GDP onto the CPI and Examine Residuals

Regress the log GDP onto the CPI using a regression model with ARMA(1,1) errors, and then examine the residuals.

Load the U.S. Macroeconomic data set and preprocess the data.

```
load Data_USEconModel;
logGDP = log(DataTable.GDP);
dlogGDP = diff(logGDP);         % For stationarity
dCPI = diff(DataTable.CPIAUCSL); % For stationarity
T = length(dlogGDP); % Effective sample size
```

Fit a regression model with ARMA(1,1) errors.

```
ToEstMdl = regARIMA(1,0,1);
EstMdl = estimate(ToEstMdl,dlogGDP,'X',dCPI);
```

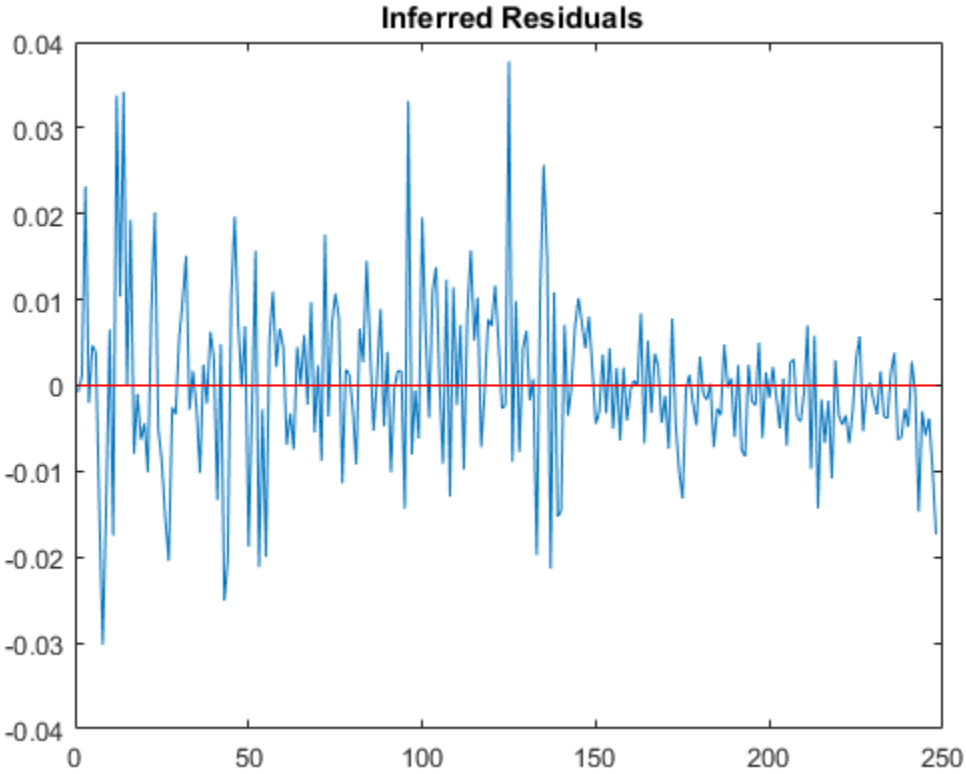
```
    Regression with ARIMA(1,0,1) Error Model:
    ----------------------------------------
    Conditional Probability Distribution: Gaussian

                               Standard          t
     Parameter       Value       Error       Statistic
    -----------   -----------   ------------   -----------
    Intercept       0.014776    0.00146271        10.1018
        AR{1}       0.605274     0.0892902        6.77872
        MA{1}      -0.161651       0.10956       -1.47546
        Beta1     0.00204403   0.000706163        2.89456
     Variance    9.35782e-05   6.03135e-06        15.5153
```

Infer the residuals over all observations. By default, `infer` backcasts for the necessary unconditional disturbances.

```
e = infer(EstMdl,dlogGDP,'X',dCPI);
```

Plot the inferred residuals.

```
figure
plot(1:T,e,[1 T],[0 0],'r')
title('{\bf Inferred Residuals}')
```



The residuals are centered around 0, but show signs of heteroscedasticity.

- "Infer Residuals for Diagnostic Checking" on page 5-140
- "Forecast a Regression Model with Multiplicative Seasonal ARIMA Errors" on page 4-206

References

[1] Box, G. E. P., G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis: Forecasting and Control*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

[2] Davidson, R., and J. G. MacKinnon. *Econometric Theory and Methods*. Oxford, UK: Oxford University Press, 2004.

[3] Enders, W. *Applied Econometric Time Series*. Hoboken, NJ: John Wiley & Sons, Inc., 1995.

[4] Hamilton, J. D. *Time Series Analysis*. Princeton, NJ: Princeton University Press, 1994.

[5] Pankratz, A. *Forecasting with Dynamic Regression Models*. John Wiley & Sons, Inc., 1991.

[6] Tsay, R. S. *Analysis of Financial Time Series*. 2nd ed. Hoboken, NJ: John Wiley & Sons, Inc., 2005.

See Also

regARIMA | estimate | forecast | simulate

More About

- "Residual Diagnostics" on page 3-90
- "Select a Regression Model with ARIMA Errors" on page 4-123
- "Intercept Identifiability Illustration" on page 4-132

isEqLagOp

Class: LagOp

Determine if two LagOp objects are same mathematical polynomial

Syntax

```
indicator = isEqLagOp(A,B)
indicator = isEqLagOp(A,B,Name,Value)
```

Description

`indicator = isEqLagOp(A,B)` determines if two lag operator polynomials *A* and *B* are the same. *indicator* is a boolean indicator for the equality test. TRUE indicates the two polynomials are identical to within tolerance; FALSE indicates the two polynomials are not identical to within tolerance.

`indicator = isEqLagOp(A,B,Name,Value)` determines if two lag operator polynomials are the same with additional options specified by one or more `Name,Value` pair arguments.

If at least one of *A* or *B* is a lag operator polynomial object, the other can be a cell array of matrices (initial lag operator coefficients), or a single matrix (zero-degree lag operator).

Input Arguments

A

Lag operator polynomial object, as created by LagOp, against which the equality of B is tested.

B

Lag operator polynomial object, as created by LagOp, against which the equality of A is tested.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'Tolerance'`

Nonnegative scalar tolerance used for testing equality. The default is `1e-12`. Specifying a tolerance greater than the default relaxes the comparison criterion. Two polynomials are deemed sufficiently close to indicate equality if the differences in magnitude of all elements of all coefficient matrices at all lags are less than or equal to the specified tolerance.

Default: `1e-12`

Output Arguments

`indicator`

Boolean indicator for the equality test. `true` indicates the two polynomials are identical to within tolerance; `false` indicates the two polynomials are not identical to within tolerance.

Examples

Determine the Equivalence of Two Lag Polynomials

Create a lag operator polynomial and convert it to a cell array:

```
A = LagOp({1 0.8 0.3 0.2});
B = toCellArray(A);
isEqLagOp(A,B)
```

```
ans =

     1
```

The converted cell array is equivalent to the LagOp polynomial object.

See Also

toCellArray

How To

- "Specify Lag Operator Polynomials" on page 2-11

isNonZero

Class: LagOp

Find lags associated with nonzero coefficients of `LagOp` objects

Syntax

```
indicator = isNonZero(A,testLags)
```

Description

Given a vector of candidate lags to test, `indicator = isNonZero(A,testLags)`, determines which lags are associated with nonzero coefficients of a lag operator polynomial *A(L)*.

Examples

Determine Which Lag Has a Nonzero Coefficient

Create a Lag Operator polynomial object and add a term with the `Coefficients` property:

```
A = LagOp({1 0.8 0.3 0.2});
A.Coefficients(7)={0.5};
isNonZero(A,7)
```

```
ans =

     1
```

isStable

Class: LagOp

Determine stability of lag operator polynomial

Syntax

```
[indicator,eigenvalues] = isStable(A)
```

Description

`[indicator,eigenvalues] = isStable(A)` takes a lag operator polynomial object `A` and checks if it is stable. The stability condition requires that the magnitudes of all roots of the characteristic polynomial are less than 1 to within a small numerical tolerance.

Tips

- Zero-degree polynomials are always stable.
- For polynomials of degree greater than zero, the presence of NaN-valued coefficients returns a `false` stability indicator and vector of `NaN`s in *`eigenvalues`*.
- When testing for stability, the comparison incorporates a small numerical tolerance. The indicator is `true` when the magnitudes of all eigenvalues are less than `1-10*eps`, where `eps` is machine precision. Users who wish to incorporate their own tolerance (including `0`) may simply ignore *`indicator`* and determine stability as follows:

  ```
  [~,eigenvalues] = isStable(A);
  indicator = all(abs(eigenvalues) < (1-tol));
  ```

  for some small, nonnegative tolerance `tol`.

Input Arguments

A

Lag operator polynomial object, as produced by LagOp.

Output Arguments

`indicator`

Boolean value for the stability test. `true` indicates that *A(L)* is stable and that the magnitude of all eigenvalues of its characteristic polynomial are less than one; `false` indicates that *A(L)* is unstable and that the magnitude of at least one of the eigenvalues of its characteristic polynomial is greater than or equal to one.

`eigenvalues`

Eigenvalues of the characteristic polynomial associated with *A(L)*. The length of *`eigenvalues`* is the product of the degree and dimension of *A(L)*.

Examples

Check a Lag Operator Polynomial for Stability

Divide two Lag Operator polynomial objects and check if the resulting polynomial is stable:

```
A = LagOp({1 -0.6 0.08});
B = LagOp({1 -0.5});
[indicator,eigenvalues]=isStable(A\B)
```

```
indicator =

     1


eigenvalues =

   0.3531 + 0.0000i
```

```
-0.0723 + 0.3003i
-0.0723 - 0.3003i
-0.3086 + 0.0000i
```

References

[1] Hamilton, J. D. *Time Series Analysis*. Princeton, NJ: Princeton University Press, 1994.

See Also

mldivide | mrdivide

How To

- "Specify Lag Operator Polynomials" on page 2-11
- "Plot the Impulse Response Function" on page 5-88

jcitest

Johansen cointegration test

Syntax

```
[h,pValue,stat,cValue,mles] = jcitest(Y)
[h,pValue,stat,cValue,mles] = jcitest(Y,Name,Value)
```

Description

Johansen tests assess the null hypothesis $H(r)$ of cointegration rank less than or equal to r among the `numDims`-dimensional time series in `Y` against alternatives H(`numDims`) (`trace` test) or $H(r+1)$ (`maxeig` test). The tests also produce maximum likelihood estimates of the parameters in a vector error-correction (VEC) model of the cointegrated series.

`[h,pValue,stat,cValue,mles] = jcitest(Y)` performs the Johansen cointegration test on a data matrix `Y`.

`[h,pValue,stat,cValue,mles] = jcitest(Y,Name,Value)` performs the Johansen cointegration test on a data matrix `Y` with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

`Y`

`numObs`-by-`numDims` matrix representing `numObs` observations of a `numDims`-dimensional time series y_t, with the last observation the most recent. Y cannot have more than 12 columns. Observations containing `NaN` values are removed. Initial values for lagged variables in VEC model estimation are taken from the beginning of the data.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'model'`

String or cell vector of strings specifying the form of the deterministic components of the VEC(q) model of y_t:

$$\Delta \boldsymbol{y}_t = C\boldsymbol{y}_{t-1} + B_1\Delta \boldsymbol{y}_{t-1} + \ldots + B_q\Delta \boldsymbol{y}_{t-q} + DX + \varepsilon_t$$

If $r <$ `numDims` is the cointegration rank, then $C = AB'$, where A is a `numDims`-by-r matrix of error-correction speeds and B is a `numDims`-by-r matrix of basis vectors for the space of cointegrating relations. X contains any exogenous terms representing deterministic trends in the data. For maximum likelihood estimation, it is assumed that $\varepsilon_t \sim \text{NID}(0,Q)$, where Q is the innovations covariance matrix.

Values of `model` are those considered by Johansen [2].

Value	Form of $Cy_{t-1} + DX$
`H2`	$AB'y_{t-1}$. There are no intercepts or trends in the cointegrating relations and there are no trends in the data. This model is only appropriate if all series have zero mean.
`H1*`	$A(B'y_{t-1}+c_0)$. There are intercepts in the cointegrating relations and there are no trends in the data. This model is appropriate for nontrending data with nonzero mean.
`H1`	$A(B'y_{t-1}+c_0)+c_1$. There are intercepts in the cointegrating relations and there are linear trends in the data. This is a model of *deterministic cointegration*, where the relations eliminate both stochastic and deterministic trends in the data. This is the default value.
`H*`	$A(B'y_{t-1}+c_0+d_0t)+c_1$. There are intercepts and linear trends in the cointegrating relations and there are linear trends in the data. This is a model of *stochastic cointegration*, where the relations eliminate stochastic but not deterministic trends in the data.
`H`	$A(B'y_{t-1}+c_0+d_0t)+c_1+d_1t$. There are intercepts and linear trends in the cointegrating relations and there are quadratic trends in the data. Unless quadratic trends are actually

Value	Form of $Cy_{t-1} + DX$
	present in the data, this model might produce good in-sample fits but poor out-of-sample forecasts.

Deterministic terms outside of the cointegrating relations, c_1 and d_1, are identified by projecting constant and linear regression coefficients, respectively, onto the orthogonal complement of A.

`'lags'`

Scalar or vector of nonnegative integers indicating the number q of lagged differences in the VEC(q) model of y_t.

Lagging and differencing a time series reduce the sample size. Absent any presample values, if y_t is defined for $t = 1{:}N$, then the lagged series y_{t-k} is defined for $t = k + 1{:}N$. Differencing reduces the time base to $k{+}2{:}N$. With q lagged differences, the common time base is $q{+}2{:}N$ and the effective sample size is $T = N{-}(q{+}1)$.

Default: 0

`'test'`

String or cell vector of strings indicating the type of test to be performed. Values are `'trace'` or `'maxeig'`. The default value is `'trace'`. Both tests assess the null hypothesis $H(r)$ of cointegration rank less than or equal to r. Statistics are computed using the effective sample size T and ordered estimates of the eigenvalues of $C = AB'$, $\lambda_1 > \ldots > \lambda_d$, where $d =$ `numDims`.

- When the value is `'trace'`, the alternative hypothesis is $H($`numDims`$)$. Statistics are:

  $$-T\left[\log(1-\lambda_{r+1}) + \ldots + \log(1-\lambda_{numDims})\right]$$

- When the value is `'maxeig'`, the alternative hypothesis is $H(r{+}1)$. Statistics are:

  $$-T\log(1-\lambda_{r+1})$$

`'alpha'`

Scalar or vector of nominal significance levels for the tests. Values must be between 0.001 and 0.999.

Default: 0.05

'display'

String or cell vector of strings indicating whether or not to display a summary of test results and parameter estimates in the Command Window.

Value	Display
`off`	No display to the command window. This is the default if `jcitest` is called with only one output argument (`h`).
`summary`	Display a summary of test results. Null ranks $r = 0$:`numDims` − 1 are displayed in the first column of each summary. Multiple tests are displayed in separate summaries. This is the default if `jcitest` is called with more than one output argument (that is, if `pValue` is computed), and is unavailable if `jcitest` is called with only one output argument (`h`).
`params`	Display maximum likelihood estimates of the parameter values associated with the reduced-rank VEC(q) model of y_t. This display is only available if `jcitest` is called with five output arguments (that is, if `mles` is computed). Displayed parameter values are returned in `mles.rn(m).paramVals` for null rank r = `n` and test `m`.
`full`	Display both `summary` and `params`.

Scalar or single string values are expanded to the length of any vector value (the number of tests). Vector values must have equal length.

Output Arguments

h

`numTests`-by-`numDims` tabular array of Boolean decisions for the tests.

Rows of `h` correspond to tests specified by the input arguments, and the software labels the rows `t1`,`t2`,...,`tu`, where u = `numTests`. Variables of `h` correspond to different, maintained cointegration ranks r = 0,...,`numDims` − 1, and the software labels the variables `r0`,`r1`,...,`rR`, where R = `numDims` − 1. To access results stored in `h`, for example, the result for test m of null rank n, use `h.rn(m)`.

Values of `h` equal to `1` (`true`) indicate rejection of the null of cointegration rank r in favor of the alternative. Values of `h` equal to `0` (`false`) indicate a failure to reject the null.

pValue

numTests-by-numDims tabular array of right-tail probabilities of the test statistics.

Rows of pValue correspond to tests specified by the input arguments, and the software labels the rows t1,t2,...,t*u*, where *u* = numTests. Variables of pValue correspond to different, maintained cointegration ranks *r* = 0,...,numDims – 1, and the software labels the variables r0,r1,...,r*R*, where *R* = numDims – 1. To access results stored in pValue, for example, the result for test *m* of null rank *n*, use pValue.r*n*(*m*).

stat

numTests-by-numDims tabular array of test statistics, determined by the test name-value pair argument.

Rows of stat correspond to tests specified by the input arguments, and the software labels the rows t1,t2,...,t*u*, where *u* = numTests. Variables of stat correspond to different, maintained cointegration ranks *r* = 0,...,numDims – 1, and the software labels the variables r0,r1,...,r*R*, where *R* = numDims – 1. To access results stored in stat, for example, the result for test *m* of null rank *n*, use stat.r*n*(*m*).

cValue

numTests-by-numDims tabular array of critical values for right-tail probabilities, determined by the alpha name-value pair argument. jcitest loads tables of critical values from the file Data_JCITest.mat, then linearly interpolates test-critical values from the tables. Tabulated values were computed using methods described in [3].

Rows of cValue correspond to tests specified by the input arguments, and the software labels the rows t1,t2,...,t*u*, where *u* = numTests. Variables of cValue correspond to different, maintained cointegration ranks *r* = 0,...,numDims – 1, and the software labels the variables r0,r1,...,r*R*, where *R* = numDims – 1. To access results stored in cValue, for example, the result for test *m* of null rank *n*, use cValue.r*n*(*m*).

mles

numTests-by-numDims tabular array of structures of maximum likelihood estimates associated with the VEC(q) model of y_t. Each structure contains these fields.

Field	Description
paramNames	Cell vector of parameter names, of the form:

Field	Description
	{A, B, B1,...,Bq, c0, d0, c1, d1} Elements depend on the values of `lags` and `model`.
`paramVals`	Structure of parameter estimates with field names corresponding to the parameter names in `paramNames`.
`res`	T-by-`numDims` matrix of residuals, where T is the effective sample size, obtained by fitting the VEC(q) model of y_t to the input data.
`EstCov`	Estimated covariance Q of the innovations process ε_t.
`eigVal`	Eigenvalue associated with $H(r)$.
`eigVec`	Eigenvector associated with the eigenvalue in `eigVal`. Eigenvectors v are normalized so that $v'S_{11}v = 1$, where S_{11} is defined as in [2].
`rLL`	Restricted loglikelihood of `Y` under the null.
`uLL`	Unrestricted loglikelihood of `Y` under the alternative.

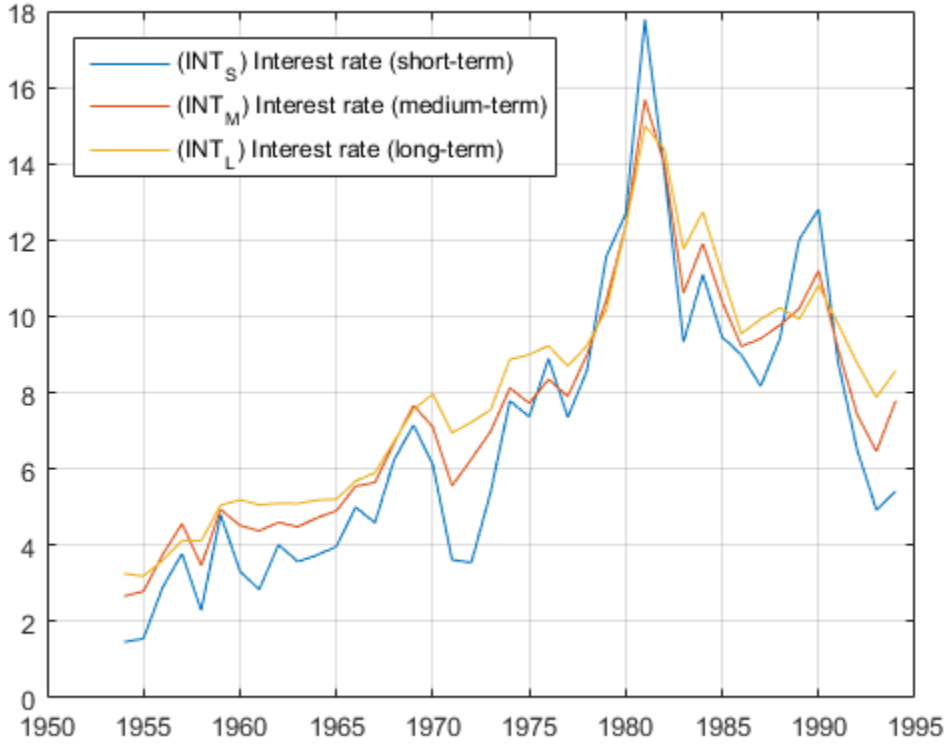
Rows of `mles` correspond to tests specified by the input arguments, and the software labels the rows `t1`,`t2`,...,`tu`, where u = `numTests`. Variables of `mles` correspond to different, maintained cointegration ranks $r = 0,...,$`numDims` $- 1$, and the software labels the variables `r0`,`r1`,...,`rR`, where R = `numDims` $- 1$. To access results stored in `mles`, for example, the result for test m of null rank n, use `mles.rn(m)`. You can further access the fields of the structure using dot notation, for example, enter `mles.rn(m).paramNames` for the parameter names.

Examples

Test Multiple Series for Cointegration Using jcitest

Load data on term structure of interest rates in Canada:

```
load Data_Canada
Y = Data(:,3:end);
names = series(3:end);
plot(dates,Y)
legend(names,'location','NW')
grid on
```



Test for cointegration:

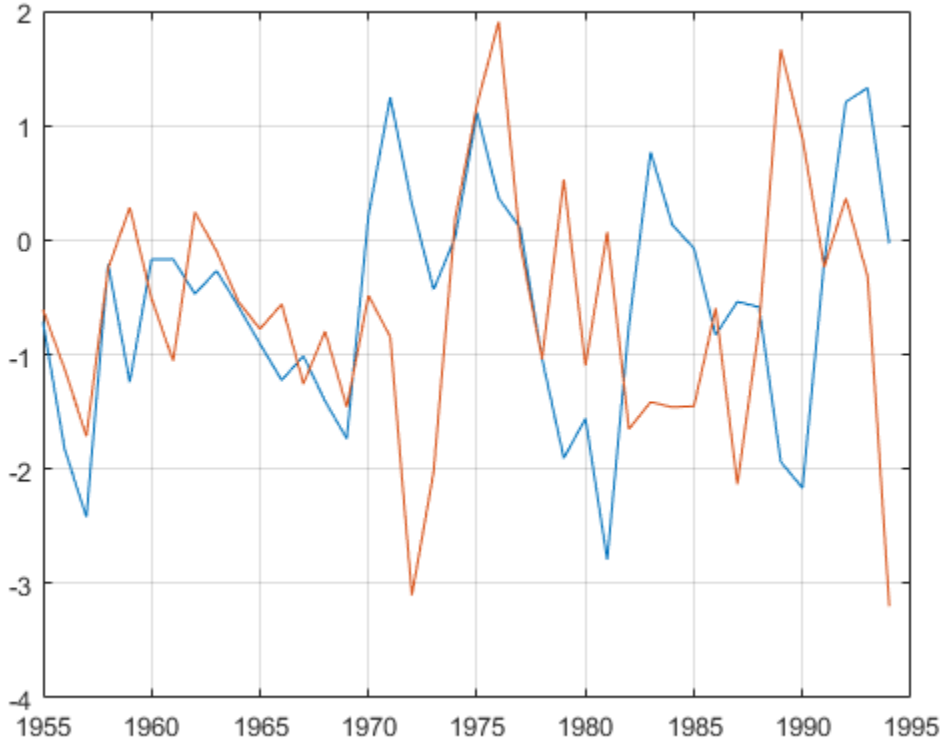
```
[h,pValue,stat,cValue,mles] = jcitest(Y,'model','H1');
h,pValue
```

```
************************
Results Summary (Test 1)

Data: Y
Effective sample size: 40
Model: H1
Lags: 0
Statistic: trace
```

```
Significance level: 0.05

r  h  stat      cValue   pValue   eigVal
-----------------------------------------
0  1  37.6886   29.7976  0.0050   0.4101
1  1  16.5770   15.4948  0.0343   0.2842
2  0  3.2003    3.8415   0.0737   0.0769

h =

           r0       r1       r2
          _____    _____    _____

    t1    true     true     false


pValue =

             r0           r1           r2
          _________    ________    ________

    t1    0.0050497    0.034294    0.073661
```

Plot estimated cointegrating relations $B'y_{t-1} + c_0$:

```
YLag = Y(2:end,:);
T = size(YLag,1);
B = mles.r2.paramVals.B;
c0 = mles.r2.paramVals.c0;
plot(dates(2:end),YLag*B+repmat(c0',T,1))
grid on
```



More About

Algorithms

Time series in `Y` might be stationary in levels or first differences (i.e., $I(0)$ or $I(1)$). Rather than pretesting series for unit roots (using, e.g., `adftest`, `pptest`, `kpsstest`, or `lmctest`), the Johansen procedure formulates the question within the model. An $I(0)$ series is associated with a standard unit vector in the space of cointegrating relations, and its presence can be tested using `jcontest`.

If `jcitest` fails to reject the null of cointegration rank $r = 0$, the inference is that the error-correction coefficient C is zero, and the VEC(q) model reduces to a standard VAR(q)

model in first differences. If `jcitest` rejects all cointegration ranks r less than `numDims`, the inference is that C has full rank, and y_t is stationary in levels.

The parameters A and B in the reduced-rank VEC(q) model are not uniquely identified, though their product $C = AB'$ is. `jcitest` constructs `B = V(:,1:r)` using the orthonormal eigenvectors `V` returned by `eig`, then renormalizes so that `V'*S11*V = I`, as in [2].

To test linear constraints on the error-correction speeds A and the space of cointegrating relations spanned by B, use `jcontest`.

To convert VEC(q) model parameters in the `mles` output to VAR(q+1) model parameters, use `vec2var`.

- "Cointegration and Error Correction Analysis" on page 7-108

References

[1] Hamilton, J. D. *Time Series Analysis*. Princeton, NJ: Princeton University Press, 1994.

[2] Johansen, S. *Likelihood-Based Inference in Cointegrated Vector Autoregressive Models*. Oxford: Oxford University Press, 1995.

[3] MacKinnon, J. G., A. A. Haug, and L. Michelis. "Numerical Distribution Functions of Likelihood Ratio Tests for Cointegration." *Journal of Applied Econometrics*. v. 14, 1999, pp. 563–577.

[4] Turner, P. M. "Testing for Cointegration Using the Johansen Approach: Are We Using the Correct Critical Values?" *Journal of Applied Econometrics*. v. 24, 2009, pp. 825–831.

See Also

`egcitest` | `jcontest` | `vec2var`

Introduced in R2011a

jcontest

Johansen constraint test

Syntax

```
[h,pValue,stat,cValue,mles] = jcontest(Y,r,test,Cons)
[h,pValue,stat,cValue,mles] = jcontest(Y,r,test,Cons,Name,Value)
```

Description

`jcontest` tests linear constraints on either the error-correction speeds A or the cointegration space spanned by B in the reduced-rank VEC(q) model of y_t:

$$\Delta \boldsymbol{y}_t = AB'\boldsymbol{y}_{t-1} + B_1\Delta \boldsymbol{y}_{t-1} + \ldots + B_q\Delta \boldsymbol{y}_{t-q} + DX + \varepsilon_t.$$

Null hypotheses specifying constraints on A or B are tested against the alternative $H(r)$ of cointegration rank less than or equal to r, without the constraints. The tests also produce maximum likelihood estimates of the parameters in the VEC(q) model, subject to the constraints.

`[h,pValue,stat,cValue,mles] = jcontest(Y,r,test,Cons)` performs the Johansen constraint test on a data matrix `Y`.

`[h,pValue,stat,cValue,mles] = jcontest(Y,r,test,Cons,Name,Value)` performs the Johansen constraint test on a data matrix `Y` with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

Y

`numObs`-by-`numDims` matrix representing `numObs` observations of a `numDims`-dimensional time series y_t, with the last observation the most recent. Observations containing `NaN` values are removed. Initial values for lagged variables in VEC model estimation are taken from the beginning of the data.

`r`

Scalar or vector of integers between 1 and `numDims`−1, inclusive, specifying the common rank of *A* and *B*, as inferred by `jcitest`.

`test`

String or cell vector of strings specifying the type of tests to be performed. Values are:

`ACon`	Test linear constraints on *A*.
`AVec`	Test specific vectors in *A*.
`BCon`	Test linear constraints on *B*.
`BVec`	Test specific vectors in *B*.

`Cons`

Matrix or cell vector of matrices specifying test constraints. For constraints on *B*, the number of rows in each matrix, `numDims1`, is the number of dimensions in the data, `numDims`, unless `model` is `H*`or `H1*`, in which case `numDims1` = `numDims` + 1 and constraints include the restricted deterministic term in the model.

Test	Cons
`ACon`	`numDims`-by-`numCons` matrix `R` specifying `numCons` constraints on *A* given by `R'*A = 0`. `numCons` must not exceed `numDims` − *r*.
`AVec`	`numDims`-by-`numCons` matrix specifying `numCons` of the error-correction speed vectors in *A*. `numCons` must not exceed *r*.
`BCon`	`numDims1`-by-`numCons` matrix `R` specifying `numCons` constraints on *B* given by `R'*B = 0`. `numCons` must not exceed `numDims` − *r*.
`BVec`	`numDims1`-by-`numCons` matrix specifying `numCons` of the cointegrating vectors in *B*. `numCons` must not exceed *r*.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

'model'

String or cell vector of strings specifying the form of the deterministic components of the VEC(q) model of y_t. Values of `model` are those considered by Johansen [3]:

Value	Form of $AB'y_{t-1} + DX$
`H2`	$AB'y_{t-1}$. There are no intercepts or trends in the cointegrating relations and there are no trends in the data. This model is only appropriate if all series have zero mean.
`H1*`	$A(B'y_{t-1} + c_0)$. There are intercepts in the cointegrating relations and there are no trends in the data. This model is appropriate for nontrending data with nonzero mean.
`H1`	$A(B'y_{t-1} + c_0) + c_1$. There are intercepts in the cointegrating relations and there are linear trends in the data. This is a model of *deterministic cointegration*, where the relations eliminate both stochastic and deterministic trends in the data. This is the default value.
`H*`	$A(B'y_{t-1} + c_0 + d_0t) + c_1$. There are intercepts and linear trends in the cointegrating relations and there are linear trends in the data. This is a model of *stochastic cointegration*, where the relations eliminate stochastic but not deterministic trends in the data.
`H`	$A(B'y_{t-1} + c_0 + d_0t) + c_1 + d_1t$. There are intercepts and linear trends in the cointegrating relations and there are quadratic trends in the data. Unless quadratic trends are actually present in the data, this model may produce good in-sample fits but poor out-of-sample forecasts.

Deterministic terms outside of the cointegrating relations, c_1 and d_1, are identified by projecting constant and linear regression coefficients, respectively, onto the orthogonal complement of A.

'lags'

Scalar or vector of nonnegative integers indicating the number q of lagged differences in the VEC(q) model of y_t.

Lagging and differencing a time series reduce the sample size. Absent any presample values, if y_t is defined for $t = 1{:}N$, then the lagged series y_{t-k} is defined for $t = k{+}1{:}N$. Differencing reduces the time base to $k{+}2{:}N$. With q lagged differences, the common time base is $q{+}2$:N and the effective sample size is $T = N - (q{+}1)$.

Default: 0

`'alpha'`

Scalar or vector of nominal significance levels for the tests. Values must be greater than zero and less than one. The default value is `0.05`.

Single-element values for inputs are expanded to the length of any vector value (the number of tests). Vector values must have equal length. If any value is a row vector, all outputs are row vectors.

Output Arguments

`h`

Vector of Boolean decisions for the tests, with length equal to the number of tests. Values of `h` equal to `1` (`true`) indicate rejection of the null that the constraints hold in favor of the alternative that they do not. Values of `h` equal to `0` (`false`) indicate a failure to reject the null.

`pValue`

Vector of right-tail probabilities of the test statistics, with length equal to the number of tests.

`stat`

Vector of test statistics, with length equal to the number of tests. Statistics are likelihood ratios determined by the test.

`cValue`

Critical values for right-tail probabilities, with length equal to the number of tests. The asymptotic distributions of the test statistics are chi-square, with the degree-of-freedom parameter determined by the test.

`mles`

Structure of maximum likelihood estimates associated with the VEC(q) model of y_t, subject to the constraints. Each structure has the following fields:

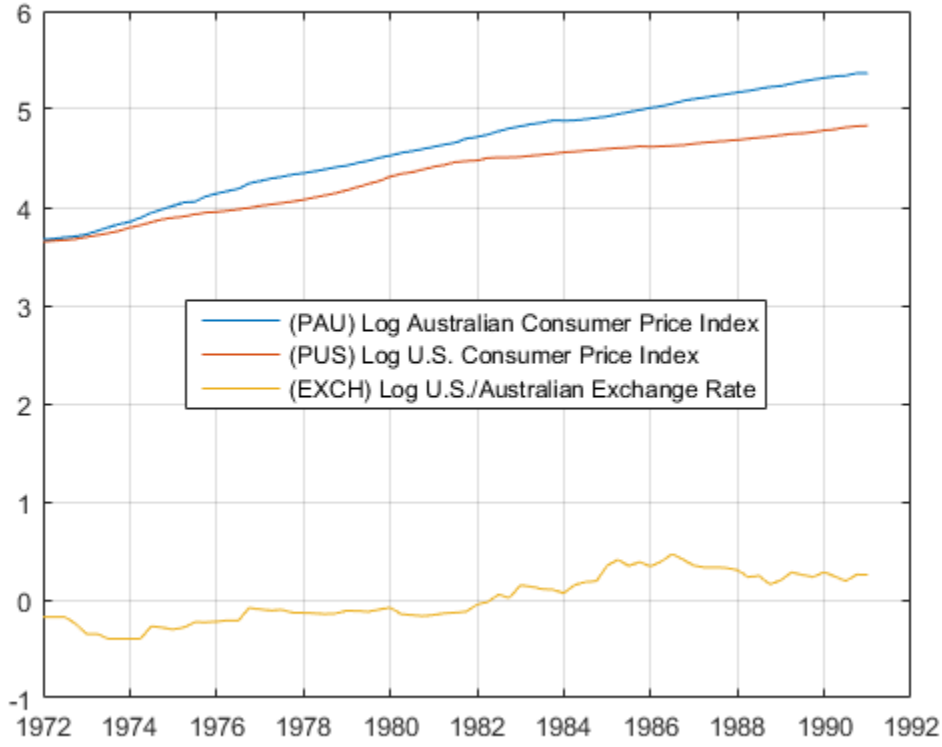
`paramNames`	Cell vector of parameter names, of the form: {A, B, B1,...,Bq, c0, d0, c1, d1} Elements depend on the values of `lags` and `model`.
`paramVals`	Structure of parameter estimates with field names corresponding to the parameter names in `paramNames`.
`res`	T-by-`numDims` matrix of residuals, where T is the effective sample size, obtained by fitting the VEC(q) model of $y(t)$ to the input data.
`EstCov`	Estimated covariance Q of the innovations process ε_t.
`rLL`	Restricted loglikelihood of `Y` under the null.
`uLL`	Unrestricted loglikelihood of `Y` under the alternative.
`dof`	Degrees of freedom of the asymptotic chi-square distribution of the test statistic.

Examples

Test Purchasing Power Parity Using jcontest

Load data on Australian and U.S. prices:

```
load Data_JAustralian
p1 = DataTable.PAU; % Log Australian Consumer Price Index
p2 = DataTable.PUS; % Log U.S. Consumer Price Index
s12 = DataTable.EXCH; % Log AUD/USD Exchange Rate
Y = [p1 p2 s12];
plot(dates,Y)
datetick('x','yyyy')
legend(series(1:3),'Location','Best')
grid on
```



Pretest the individual series for stationarity:

```
[h0,pValue0] = jcontest(Y,1,'BVec',{[1 0 0]',[0 1 0]',[0 0 1]'})
```

```
h0 =

     1     1     0

pValue0 =

    0.0000    0.0000    0.0657
```

Test for cointegration:

```
[h1,pValue1] = jcitest(Y)
```

```
Warning: Test statistic #1 above tabulated critical values:
minimum p-value = 0.001 reported.

************************
Results Summary (Test 1)

Data: Y
Effective sample size: 76
Model: H1
Lags: 0
Statistic: trace
Significance level: 0.05


r  h  stat      cValue   pValue   eigVal
----------------------------------------
0  1  60.3393   29.7976  0.0010   0.4687
1  0  12.2749   15.4948  0.1446   0.1157
2  0  2.9315    3.8415   0.0869   0.0378

h1 =

           r0       r1       r2
          _____    _____    _____

    t1    true     false    false


pValue1 =

           r0         r1           r2
          _____    _______    ________

    t1    0.001    0.14455    0.086906
```

Test for purchasing power parity ($p1 = p2 + s12$):

```
[h2,pValue2] = jcontest(Y,1,'BCon',[1 -1 -1]')
```

```
h2 =

     0


pValue2 =

    0.0540
```

More About

Algorithms

- The parameters A and B in the reduced-rank VEC(q) model are not uniquely identified. `jcontest` identifies B using the methods in [3], depending on the test.
- When constructing constraints, interpret the rows and columns of the `numDims`-by-r matrices A and B as follows:
  - Row i of A contains the adjustment speeds of variable y_i to disequilibrium in each of the r cointegrating relations.
  - Column j of A contains the adjustment speeds of each of the `numDims` variables to disequilibrium in the jth cointegrating relation.
  - Row i of B contains the coefficients of variable y_i in each of the r cointegrating relations.
  - Column j of B contains the coefficients of each `numDims` variable in the jth cointegrating relation.
- Tests on B answer questions about the space of cointegrating relations. Tests on A answer questions about common driving forces in the system. For example, an all-zero row in A indicates a variable that is weakly exogenous with respect to the coefficients in B. Such a variable might affect other variables, but it does not adjust to disequilibrium in the cointegrating relations. Similarly, a standard unit vector column in A indicates a variable that is exclusively adjusting to disequilibrium in a particular cointegrating relation.
- Constraints matrices `R` satisfying $R'A = 0$ or $R'B = 0$ are equivalent to $A = H\varphi$ or $B = H\varphi$, where H is the orthogonal complement of R (`null(R')`) and φ is a vector of free parameters.

- `jcontest` compares finite-sample statistics to asymptotic critical values, and tests can show significant size distortions for small samples. See [2]. Larger samples lead to more reliable inferences.
- To convert VEC(q) model parameters in the `mles` output to vector autoregressive (VAR) model parameters, use the utility `vec2var`.
- "Cointegration and Error Correction Analysis" on page 7-108

References

[1] Hamilton, J. D. *Time Series Analysis*. Princeton, NJ: Princeton University Press, 1994.

[2] Haug, A. "Testing Linear Restrictions on Cointegrating Vectors: Sizes and Powers of Wald Tests in Finite Samples." *Econometric Theory*. v. 18, 2002, pp. 505–524.

[3] Johansen, S. *Likelihood-Based Inference in Cointegrated Vector Autoregressive Models*. Oxford: Oxford University Press, 1995.

[4] Juselius, K. *The Cointegrated VAR Model*. Oxford: Oxford University Press, 2006.

[5] Morin, N. "Likelihood Ratio Tests on Cointegrating Vectors, Disequilibrium Adjustment Vectors, and their Orthogonal Complements." *European Journal of Pure and Applied Mathematics*. v. 3, 2010, pp. 541–571.

See Also

`jcitest` | `vec2var`

Introduced in R2011a

kpsstest

KPSS test for stationarity

Syntax

```
h = kpsstest(y)
h = kpsstest(y,Name,Value)
[h,pValue] = kpsstest( ___ )
[h,pValue,stat,cValue,reg] = kpsstest( ___ )
```

Description

`h = kpsstest(y)` returns the logical value (`h`) with the rejection decision from conducting the Kwiatkowski, Phillips, Schmidt, and Shin (KPSS) test for a unit root in the univariate time series `y`.

`h = kpsstest(y,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

- If any `Name,Value` pair argument is a vector, then all `Name,Value` pair arguments specified must be vectors of equal length or length one. `kpsstest(y,Name,Value)` treats each element of a vector input as a separate test, and returns a vector of rejection decisions.
- If any `Name,Value` pair argument is a row vector, then `kpsstest(y,Name,Value)` returns a row vector.

`[h,pValue] = kpsstest( ___ )` returns the rejection decision and p-value for the hypothesis test, using any of the input arguments in the previous syntaxes.

`[h,pValue,stat,cValue,reg] = kpsstest( ___ )` additionally returns the test statistic, critical value, and a structure of regression statistics for the hypothesis test.

Examples

Assess Trend Stationarity of a Series

Reproduce the first row of the second half of Table 5 in Kwiatkowski et al., 1992.

Load the Nelson-Plosser Macroeconomic series data set.

```
load Data_NelsonPlosser
```

Linearize the real gross national product series (RGNP).

```
logGNPR = log(DataTable.GNPR);
```

Assess the null hypothesis that the series is trend stationary over a range of lags.

```
lags = (0:8)';
[~,pValue,stats] = kpsstest(logGNPR,'Lags',lags,'Trend',true);
results = [lags pValue stats]
```

```
Warning: Test statistic #1 above tabulated critical values:
minimum p-value = 0.010 reported.
Warning: Test statistic #2 above tabulated critical values:
minimum p-value = 0.010 reported.
Warning: Test statistic #3 above tabulated critical values:
minimum p-value = 0.010 reported.

results =

         0    0.0100    0.6299
    1.0000    0.0100    0.3367
    2.0000    0.0100    0.2421
    3.0000    0.0169    0.1976
    4.0000    0.0276    0.1729
    5.0000    0.0401    0.1578
    6.0000    0.0484    0.1479
    7.0000    0.0589    0.1412
    8.0000    0.0668    0.1370
```

Warnings appear because the tests using $0 \le$ `lags` ≤ 2 produce p-values that are less than 0.01. For `lags` < 7, the tests indicate sufficient evidence to suggest that log rGNP is unit root nonstationary (i.e., not trend stationary) at the default 5% level.

Test Trend Stationarity by Specifying Lags

Test whether the wage series in the manufacturing sector (1900-1970) has a unit root.

Load the Nelson-Plosser Macroeconomic data set.

```
load Data_NelsonPlosser
```

```
wages = DataTable.WN;
T = sum(isfinite(wages)); % Sample size without NaNs
sqrtT = sqrt(T) % See Kwiatkowski et al., 1992
```

```
sqrtT =

    8.4261
```

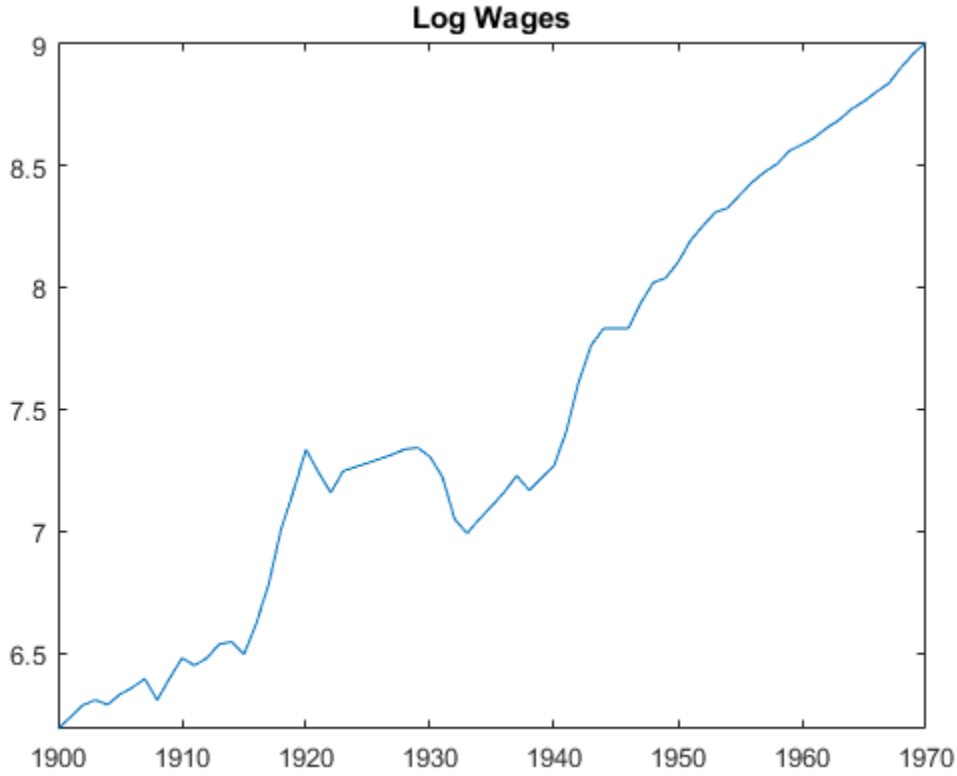
Plot the wages series.

```
plot(dates,wages)
title('Wages')
axis tight
```

The plot suggests that the wages series grows exponentially.

Linearize the wages series.

```
logWages = log(wages);
plot(dates,logWages)
title('Log Wages')
axis tight
```



The plot suggests that the log wages series has a linear trend.

Test the hypothesis that the log wages series is a unit root process with a trend (i.e., difference stationary), against the alternative that there is no unit root (i.e., trend

stationary). Conduct the test by setting a range of lags around $\sqrt{T}$, as suggested in Kwiatkowski et al., 1992.

```
[h,pValue] = kpsstest(logWages,'lags',[7:10])
```

```
Warning: Test statistic #1 below tabulated critical values:
maximum p-value = 0.100 reported.
Warning: Test statistic #2 below tabulated critical values:
maximum p-value = 0.100 reported.
Warning: Test statistic #3 below tabulated critical values:
maximum p-value = 0.100 reported.
Warning: Test statistic #4 below tabulated critical values:
maximum p-value = 0.100 reported.

h =

     0     0     0     0


pValue =

    0.1000    0.1000    0.1000    0.1000
```

All tests fail to reject the null hypothesis that the log wages series is trend stationary.

The warning messages do not indicate a problem. Rather, they indicate that the p-values are larger than 0.1. The software compares the test statistic to critical values and computes p-values that it interpolates from tables in Kwiatkowski et al., 1992.

- "Unit Root Nonstationarity" on page 3-34

Input Arguments

y — Univariate time series

vector

Univariate time series, specified as a vector. The last element is the most recent observation.

`NaN`s indicate missing observations, and `kpsstest` removes them from `y`. Removing `NaN`s decreases the effective sample size and can cause an irregular time series.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'alpha',0.1,'lags',0:2` specifies three tests that include 0, 1, and 2 autocovariance lags in the Newey-West estimator of the long-run variance, each conducted at 0.1 level of significance.

`'lags'` — Number of autocovariance lags

0 (default) | nonnegative integer | vector of nonnegative integers

Number of autocovariance lags to include in the Newey-West estimator of the long-run variance, specified as the comma-separated pair consisting of `'lags'` and a nonnegative integer or vector of nonnegative integers. Use a vector to conduct multiple tests.

Example: `'lags',0:2`

Data Types: `double`

`'trend'` — Indicate whether to include deterministic trend

`true` (default) | `false` | vector of logical values

Indicate whether to include the deterministic trend term δt in the model, specified as the comma-separated pair consisting of `'trend'` and a logical value or a vector of logical values. Use a vector to conduct multiple tests.

Example: `'trend',false`

Data Types: `logical`

`'alpha'` — Significance levels

0.05 (default) | scalar | vector

Significance levels for the hypothesis tests, specified as the comma-separated pair consisting of `'alpha'` and a scalar or vector. All values of `alpha` must be between 0.01 and 0.10. Use a vector to conduct multiple tests.

Example: `'alpha',0.01`

Data Types: `double`

Output Arguments

`h` — Test rejection decisions

logical | vector of logical values

Test rejection decisions, returned as a logical value or vector of logical values with a length equal to the number of tests that the software conducts.

- `h = 1` indicates rejection of the trend-stationary null in favor of the unit root alternative.
- `h = 0` indicates failure to reject the trend-stationary null.

`pValue` — Test statistic p-values

scalar | vector

Test statistic p-values, returned as a scalar or vector with a length equal to the number of tests that the software conducts. The p-values are right-tail probabilities.

`stat` — Test statistics

scalar | vector

Test statistics, returned as a scalar or vector with a length equal to the number of tests that the software conducts.

`kpsstest` computes test statistics using an ordinary least squares (OLS) regression.

- If you set `'trend',false`, then the software regresses `y` on an intercept.
- Otherwise, the software regresses `y` on an intercept and trend term.

`cValue` — Critical values

scalar | vector

Critical values, returned as a scalar or vector with a length equal to the number of tests that the software conducts. Critical values are for right-tail probabilities.

`reg` — Regression statistics

data structure | data structure array

Regression statistics for ordinary least squares (OLS) estimation of coefficients in the alternative model, returned as a data structure or data structure array with a length equal to the number of tests that the software conducts.

Each data structure has the following fields.

Field	Description
`num`	Length of input series with `NaN`s removed
`size`	Effective sample size, adjusted for lags
`names`	Regression coefficient names
`coeff`	Estimated coefficient values
`se`	Estimated coefficient standard errors
`Cov`	Estimated coefficient covariance matrix
`tStats`	*t* statistics of coefficients and p-values
`FStat`	*F* statistic and p-value
`yMu`	Mean of the lag-adjusted input series
`ySigma`	Standard deviation of the lag-adjusted input series
`yHat`	Fitted values of the lag-adjusted input series
`res`	Regression residuals
`DWStat`	Durbin-Watson statistic
`SSR`	Regression sum of squares
`SSE`	Error sum of squares
`SST`	Total sum of squares
`MSE`	Mean square error
`RMSE`	Standard error of the regression
`RSq`	R^2 statistic
`aRSq`	Adjusted R^2 statistic
`LL`	Loglikelihood of data under Gaussian innovations
`AIC`	Akaike information criterion
`BIC`	Bayesian (Schwarz) information criterion
`HQC`	Hannan-Quinn information criterion

More About

Kwiatkowski, Phillips, Schmidt, and Shin (KPSS) Test

Assesses the null hypothesis that a univariate time series is trend stationary against the alternative that it is a nonstationary unit root process.

The test uses the structural model:

$$y_t = c_t + \delta t + u_{1t}$$
$$c_t = c_{t-1} + u_{2t},$$

where

- δ is the trend coefficient.
- u_{1t} is a stationary process.
- u_{2t} is an independent and identically distributed process with mean 0 and variance σ^2.

The null hypothesis is that $\sigma^2 = 0$, which implies that the random walk term (c_t) is constant and acts as the model intercept. The alternative hypothesis is that $\sigma^2 > 0$, which introduces the unit root in the random walk.

The test statistic is

$$\frac{\sum_{t=1}^{T} S_t^2}{s^2 T^2},$$

where

- T is the sample size.
- s^2 is the Newey-West estimate of the long-run variance.
- $S_t = e_1 + e_2 + \ldots + e_t$.

Tips

- In order to draw valid inferences from the KPSS test, you should determine a suitable value for `'lags'`. These two methods determine a suitable number of lags:

  - Begin with a small number of lags and then evaluate the sensitivity of the results by adding more lags.
  - Kwiatkowski et al. [2] suggest that a number of lags on the order of $\sqrt{T}$, where T is the sample size, is often satisfactory under both the null and the alternative.

  For consistency of the Newey-West estimator, the number of lags must approach infinity as the sample size increases.

- You should determine the value of `'trend'` by the growth characteristics of the time series. Determine its value with a specific testing strategy in mind.
  - If a series is growing, then include a trend term to provide a reasonable comparison of a trend stationary null and a unit root process with drift. `kpsstest` sets `'trend',true` by default.
  - If a series does not exhibit long-term growth characteristics, then don't include a trend term (i.e., set `'trend',false`).

Algorithms

- `kpsstest` performs a regression to find the ordinary least squares (OLS) fit between the data and the null model.
- Test statistics follow nonstandard distributions under the null, even asymptotically. Kwiatkowski et al. [2] use Monte Carlo simulations, for models with and without a trend, to tabulate asymptotic critical values for a standard set of significance levels between 0.01 and 0.1. `kpsstest` interpolates critical values and p-values from these tables.

- "Unit Root Nonstationarity" on page 3-34

References

[1] Hamilton, J. D. *Time Series Analysis*. Princeton, NJ: Princeton University Press, 1994.

[2] Kwiatkowski, D., P. C. B. Phillips, P. Schmidt, and Y. Shin. "Testing the Null Hypothesis of Stationarity against the Alternative of a Unit Root." *Journal of Econometrics*. Vol. 54, 1992, pp. 159–178.

[3] Newey, W. K., and K. D. West. “A Simple, Positive Semidefinite, Heteroskedasticity and Autocorrelation Consistent Covariance Matrix.” *Econometrica*. Vol. 55, 1987, pp. 703–708.

See Also

`adftest` | `lmctest` | `pptest` | `vratiotest`

Introduced in R2009b

lagmatrix

Create matrix of lagged time series

Syntax

```
XLAG = lagmatrix(X,Lags)
```

Description

`XLAG = lagmatrix(X,Lags)` creates a lagged (shifted) version of a time series matrix. The `lagmatrix` function is useful for creating a regression matrix of explanatory variables for fitting the conditional mean of a return series.

Input Arguments

`X`	Time series of explanatory data. `X` can be a column vector or a matrix. As a column vector, `X` represents a univariate time series whose first element contains the oldest observation and whose last element contains the most recent observation. As a matrix, `X` represents a multivariate time series whose rows correspond to time indices. The first row contains the oldest observations and the last row contains the most recent observations. `lagmatrix` assumes that observations across any given row occur at the same time. Each column is an individual time series.
`Lags`	Vector of integer lags. `lagmatrix` applies the first lag to every series in `X`, then applies the second lag to every series in `X`, and so forth. To include a time series as is, include a `0` lag. Positive lags correspond to delays, and shift a series back in time. Negative lags correspond to leads, and shift a series forward in time.

Output Arguments

`XLAG`	Lagged transform of the time series `X`. To create `XLAG`, `lagmatrix` shifts each time series in `X` by the first lag, then shifts each time series in `X` by the second lag, and so forth. Since `XLAG` represents an explanatory

	regression matrix, each column is an individual time series. XLAG has the same number of rows as there are observations in X. Its column dimension is equal to the product of the number of columns in X and the length of Lags. lagmatrix uses a NaN (Not-a-Number) to indicate an undefined observation.

Examples

Create a Lag Matrix

Create a bivariate time series matrix X with five observations each:

```
X = [1 -1; 2 -2 ;3 -3 ;4 -4 ;5 -5]  % Create a simple
                                     % bivariate series.
```

```
X =

     1    -1
     2    -2
     3    -3
     4    -4
     5    -5
```

Create a lagged matrix XLAG, composed of X and the first two lags of X:

```
XLAG = lagmatrix(X,[0 1 2]) % Create the lagged matrix.
```

```
XLAG =

     1    -1   NaN   NaN   NaN   NaN
     2    -2     1    -1   NaN   NaN
     3    -3     2    -2     1    -1
     4    -4     3    -3     2    -2
     5    -5     4    -4     3    -3
```

The result, XLAG, is a 5-by-6 matrix.

See Also

filter | isnan | nan

Introduced before R2006a

LagOp class

Create lag operator polynomial (LagOp) object

Description

Create a lag operator polynomial *A(L)*, by specifying the coefficients and, optionally, the corresponding lags.

Construction

`A = LagOp(coefficients)`

`A = LagOp(coefficients,Name,Value)` creates a lag operator polynomial with additional options specified by one or more `Name,Value` pair arguments. `Name` can also be a property name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Input Arguments

`coefficients`

The coefficients of the lag operator polynomial. Generally, *`coefficients`* is a cell array of square matrices. For convenience, coefficients may also be specified in other ways:

- As a vector, representing a univariate time series polynomial with multiple lags.
- As a matrix, representing a multivariate time series polynomial with a single lag.
- As an existing `LagOp` object, to be updated according to the optional inputs.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'Lags'`

Vector of integer lags associated with the polynomial coefficients. If specified, the number of lags must be the same as the number of coefficients.

Default: Coefficients are associated with lags 0, 1,..., *`numCoefficients`*–1.

`'Tolerance'`

Nonnegative scalar tolerance used to determine which lags are included in the object. Specifying a tolerance greater than the default (`1e–12`) excludes lags with near-zero coefficients. A lag is excluded if the magnitudes of all elements of the coefficient matrix are less than or equal to the specified tolerance.

Default: `1e–12`

Output Arguments

`A`

Lag operator polynomial (`LagOp`) object.

Properties

`Coefficients`

Lag indexed cell array of nonzero polynomial coefficients

`Degree`

Polynomial degree (the highest lag associated with a nonzero coefficient)

`Dimension`

Polynomial dimension (the number of time series to which it may be applied)

`Lags`

Polynomial lags associated with nonzero coefficient

Methods

filter	Apply lag operator polynomial to filter time series
isEqLagOp	Determine if two `LagOp` objects are same mathematical polynomial
isNonZero	Find lags associated with nonzero coefficients of `LagOp` objects
isStable	Determine stability of lag operator polynomial
minus	Lag operator polynomial subtraction
mldivide	Lag operator polynomial left division
mrdivide	Lag operator polynomial right division
mtimes	Lag operator polynomial multiplication
plus	Lag operator polynomial addition
reflect	Reflect lag operator polynomial coefficients around lag zero
toCellArray	Convert lag operator polynomial object to cell array

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Indexing

The coefficients of lag operator polynomials are accessible by lag-based indexing; that is, by specifying nonnegative integer lags associated with the coefficients of interest.

Examples

Specify a Lag Operator Polynomial

Create a `LagOp` polynomial object:

```
A = LagOp({1 -0.6 0.08});
```

Return the coefficient at lag `L = 2`:

```
a2 = A.Coefficients{2};
```

Assign a nonzero coefficient to the 3rd lag:

```
A.Coefficients{3} = 0.5;
```

- “Specify Lag Operator Polynomials” on page 2-11

More About

- Class Attributes
- Property Attributes

lbqtest

Ljung-Box Q-test for residual autocorrelation

Syntax

```
h = lbqtest(res)
h = lbqtest(res,Name,Value)
[h,pValue] = lbqtest( ___ )
[h,pValue,stat,cValue] = lbqtest( ___ )
```

Description

`h = lbqtest(res)` returns a logical value (`h`) with the rejection decision from conducting a "Ljung-Box Q-Test" on page 9-730 for autocorrelation in the residual series `res`.

`h = lbqtest(res,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

- If any `Name,Value` pair argument is a vector, then all `Name,Value` pair arguments specified must be vectors of equal length or length one. `lbqtest(res,Name,Value)` treats each element of a vector input as a separate test, and returns a vector of rejection decisions.
- If any `Name,Value` pair argument is a row vector, then `lbqtest(res,Name,Value)` returns a row vector.

`[h,pValue] = lbqtest( ___ )` returns the rejection decision and *p*-value for the hypothesis test, using any of the input arguments in the previous syntaxes.

`[h,pValue,stat,cValue] = lbqtest( ___ )` additionally returns the test statistic (`stat`) and critical value (`cValue`) for the hypothesis test.

Examples

Test a Time Series for Autocorrelation and ARCH Effects

Load the Deutschmark/British pound foreign-exchange rate data set.

```
load Data_MarkPound
```

Convert the prices to returns.

```
returns = price2ret(Data);
```

Compute the deviations of the return series.

```
res = returns - mean(returns);
```

Test the hypothesis that the residual series is not autocorrelated, using the default number of lags.

```
h1 = lbqtest(res)
```

```
h1 =

     0
```

`h1 = 0` indicates that there is not enough evidence to reject the null hypothesis that the residuals of the returns are not autocorrelated.

Test the hypothesis that there are significant ARCH effects, using the default number of lags [3].

```
h2 = lbqtest(res.^2)
```

```
h2 =

     1
```

`h2 = 1` indicates that there are signifcant ARCH effects in the residuals of the returns.

Test for residual heteroscedasticity using `archtest` and the default number of lags.

```
h3 = archtest(res)
```

```
h3 =
```

```
     1
```

`h3 = 1` indicates that the null hypothesis of no residual heteroscedasticity should be rejected in favor of an ARCH(1) model. This result is consistent with `h2`.

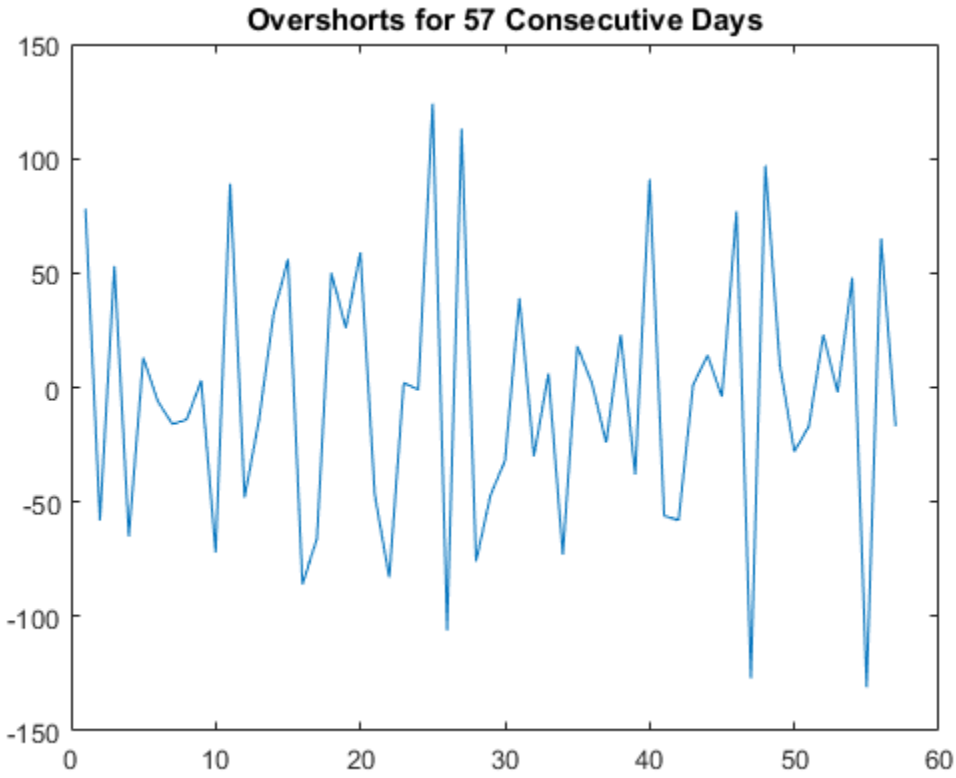
Conduct Ljung-Box Q-Test over Various Lags

Conduct multiple Ljung-Box Q-tests for autocorrelation by including various lags in the test statistic. The data set is a time series of 57 consecutive days of *overshorts* from an underground gasoline tank in Colorado [2]. That is, the current overshort (y_t) represents the accuracy in measuring the amount of fuel:

- In the tank at the end of day t
- In the tank at the end of day $t-1$
- Delivered to the tank on day t
- Sold on day t.

Load the data set.

```
load(fullfile(matlabroot,'examples','econ','Data_Overshort'))
y = Data;
T = length(y); % Sample size

figure
plot(y)
title('Overshorts for 57 Consecutive Days')
```



`lbqtest` is appropriate for a series with a constant mean. Since the series appears to fluctuate around a constant mean, you do not need to transform the data.

Compute the residuals.

```
res = y - mean(y);
```

Assess whether the residuals are autocorrelated. Include 5, 10, and 15 lags in the test statistic computation.

```
[h,pValue] = lbqtest(res,'lags',[5,10,15])
```

```
h =
```

```
     1     1     1

pValue =

    0.0016    0.0007    0.0013
```

`h` and `pValue` are vectors containing three elements corresponding to tests at each of the three lags. The first element of each output corresponds to the test at lag 5, the second element corresponds to the test at lag 10, and the third element corresponds to the test at lag 15.

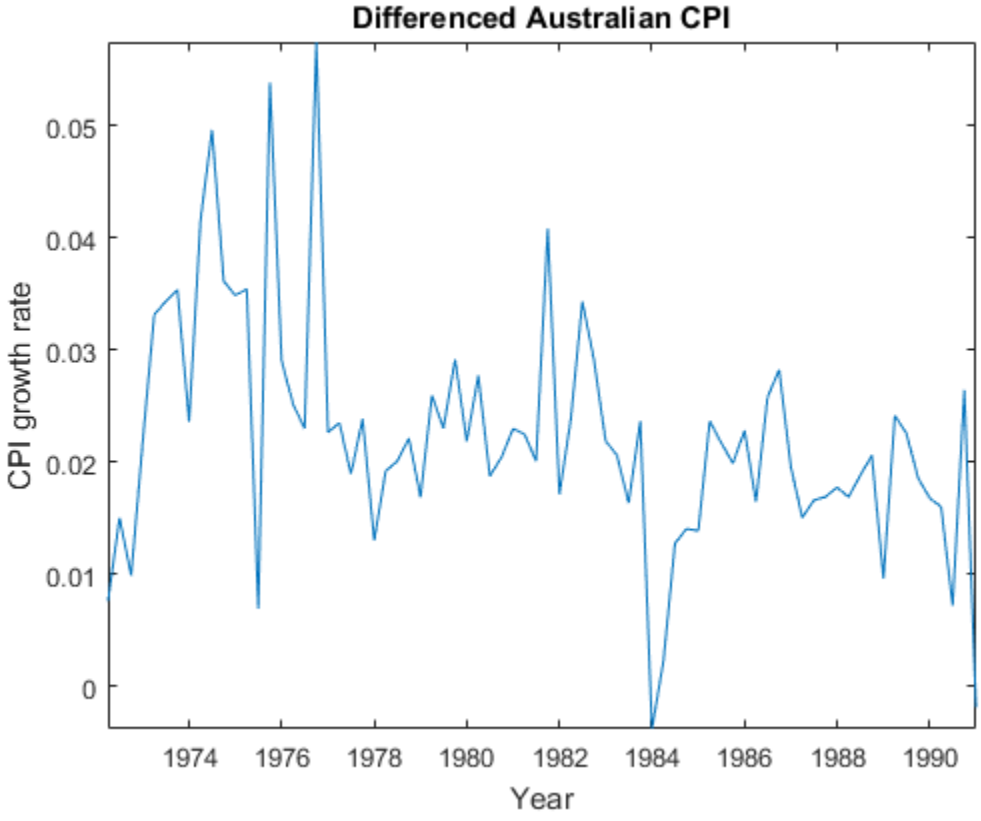
`h = 1` indicates the rejection of the null hypothesis that the residuals are not autocorrelated. `pValue` indicates the strength at which the test rejects the null hypothesis. Since all three $p-\text{values}$ are less than 0.01, there is strong evidence to reject the null hypothesis that the residuals are not autocorrelated.

Assess Autocorrelation in Inferred Residuals

Infer residuals from an estimated ARIMA model, and assess whether the residuals exhibit autocorrelation using `lbqtest`.

Load the Australian Consumer Price Index (CPI) data set. The time series (`cpi`) is the log quarterly CPI from 1972 to 1991. Remove the trend in the series by taking the first difference.

```
load Data_JAustralian
cpi = DataTable.PAU;
T = length(cpi);
dCPI = diff(cpi);

figure
plot(dates(2:T),dCPI)
title('Differenced Australian CPI')
xlabel('Year')
ylabel('CPI growth rate')
datetick
axis tight
```



The differenced series appears stationary.

Fit an AR(1) model to the series, and then infer residuals from the estimated model.

```
Mdl = arima(1,0,0);
EstMdl = estimate(Mdl,dCPI);
res = infer(EstMdl,dCPI);
stdRes = res/sqrt(EstMdl.Variance); % Standardized residuals
```

```
    ARIMA(1,0,0) Model:
    --------------------
    Conditional Probability Distribution: Gaussian
```

```
                              Standard          t
    Parameter       Value       Error       Statistic
    -----------   -----------   ------------   -----------
    Constant      0.0155643    0.00287662        5.41062
       AR{1}       0.296464       0.11048        2.68341
    Variance    0.000103804   1.19323e-05        8.69941
```

Assess whether the residuals are autocorrelated by conducting a Ljung-Box Q-test. The standardized residuals originate from the estimated model (`EstMdl`) containing parameters. When using such residuals, it is best practice to do the following:

- Adjust the degrees of freedom (`dof`) of the test statistic distribution to account for the estimated parameters.
- Set the number of lags to include in the test statistic.
- When you count the estimated parameters, skip the constant and variance parameters.

```
lags = 10;
dof  = lags - 1; % One autoregressive parameter

[h,pValue] = lbqtest(stdRes,'Lags',lags,'DOF',dof)
```

```
h =

     1


pValue =

    0.0119
```

`pValue = 0.0130` suggests that there is significant autocorrelation in the residuals at the 5% level.

- “Time Series Regression VI: Residual Diagnostics”
- “Detect Autocorrelation” on page 3-18
- “Check Fit of Multiplicative ARIMA Model” on page 3-81
- “Specify Conditional Mean and Variance Models” on page 5-79

Input Arguments

`res` — Residual series

vector

Residual series for which the software computes the test statistic, specified as a vector. The last element corresponds to the latest observation.

Typically, you fit a model to an observed time series, and `res` contains the standardized residuals from the fitted model.

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'lags',1:4,'alpha',0.1` specifies four tests with 1, 2, 3, and 4 lagged terms conducted at the 0.1 significance level.

`'lags'` — Number of lagged terms

`min(20,T-1)` (default) | positive integer | vector of positive integers

Number of lagged terms to include in the test statistic calculation, specified as the comma-separated pair consisting of `'lags'` and a positive integer or vector of positive integers.

Use a vector to conduct multiple tests.

Each element of `lags` must be less than `length(res) – 1`.

Example: `'lags',1:4`

Data Types: `double`

`'alpha'` — Significance levels

0.05 (default) | scalar | vector

Significance levels for the hypothesis tests, specified as the comma-separated pair consisting of `'alpha'` and a scalar or vector.

Use a vector to conduct multiple tests.

Each element of `alpha` must be greater than 0 and less than 1.

Example: `'alpha',0.01`

Data Types: `double`

`'dof'` — Degrees of freedom

`lags` (default) | positive integer | vector of positive integers

Degrees of freedom for the asymptotic, chi-square distribution of the test statistics, specified as the comma-separated pair consisting of `'dof'` and a positive integer or vector of positive integers.

Use a vector to conduct multiple tests.

If `dof` is an integer, then it must be less than `lags`. Otherwise, each element of `dof` must be less than the corresponding element of `lags`.

Example: `'dof',15`

Data Types: `double`

Output Arguments

`h` — Test rejection decisions

logical | vector of logicals

Test rejection decisions, returned as a logical value or vector of logical values with a length equal to the number of tests that the software conducts.

- `h = 1` indicates rejection of the no residual autocorrelation null hypothesis in favor of the alternative.
- `h = 0` indicates failure to reject the no residual autocorrelation null hypothesis.

`pValue` — Test statistic *p*-values

scalar | vector

Test statistic *p*-values, returned as a scalar or vector with a length equal to the number of tests that the software conducts.

`stat` — Test statistics

scalar | vector

Test statistics, returned as a scalar or vector with a length equal to the number of tests that the software conducts.

`cValue` — Critical values

scalar | vector

Critical values determined by `alpha`, returned as a scalar or vector with a length equal to the number of tests that the software conducts.

More About

Ljung-Box Q-Test

The Ljung-Box Q-test is a "portmanteau" test that assesses the null hypothesis that a series of residuals exhibits no autocorrelation for a fixed number of lags L, against the alternative that some autocorrelation coefficient $\rho(k)$, $k = 1, ..., L$, is nonzero.

The test statistic is

$$Q = T(T+2)\sum_{k=1}^{L}\left(\frac{\rho(k)^2}{(T-k)}\right),$$

where T is the sample size, L is the number of autocorrelation lags, and $\rho(k)$ is the sample autocorrelation at lag k. Under the null hypothesis, the asymptotic distribution of Q is chi-square with L degrees of freedom.

Tips

If you obtain `res` by fitting a model to data, then you should reduce the degrees of freedom (the argument `dof`) by the number of estimated coefficients, excluding constants. For example, if you obtain `res` by fitting an `ARMA`(p,q) model, set `dof` to $L-p-q$, where L is `lags`.

Algorithms

- The `lags` argument affects the power of the test.

  - If L is too small, then the test does not detect high-order autocorrelations.
  - If L is too large, then the test loses power when a significant correlation at one lag is washed out by insignificant correlations at other lags.
  - Box, Jenkins, and Reinsel suggest setting `min[20,T-1]` as the default value for `lags` [1].
  - Tsay cites simulation evidence that setting `lags` to a value approximating $\log(T)$ provides better power performance [5].
- `lbqtest` does not directly test for serial dependencies other than autocorrelation. However, you can use it to identify conditional heteroscedasticity (ARCH effects) by testing squared residuals [4].

  Engle's test assesses the significance of ARCH effects directly. For details, see `archtest`.

- "Ljung-Box Q-Test" on page 3-16

References

[1] Box, G. E. P., G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis: Forecasting and Control.* 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

[2] Brockwell, P. J. and R. A. Davis. *Introduction to Time Series and Forecasting*. 2nd ed. New York, NY: Springer, 2002.

[3] Gourieroux, C. *ARCH Models and Financial Applications.* New York: Springer-Verlag, 1997.

[4] McLeod, A. I. and W. K. Li. "Diagnostic Checking ARMA Time Series Models Using Squared-Residual Autocorrelations." Journal of Time Series Analysis. Vol. 4, 1983, pp. 269–273.

[5] Tsay, R. S. *Analysis of Financial Time Series.* 2nd Ed. Hoboken, NJ: John Wiley & Sons, Inc., 2005.

See Also

`archtest` | `autocorr`

Introduced before R2006a

lmctest

Leybourne-McCabe stationarity test

Syntax

```
h = lmctest(y)
h = lmctest(y,'ParameterName',ParameterValue)
[h,pValue] = lmctest(...)
[h,pValue,stat] = lmctest(...)
[h,pValue,stat,cValue] = lmctest(...)
[h,pValue,stat,cValue,reg1] = lmctest(...)
[h,pValue,stat,cValue,reg1,reg2] = lmctest(...)
```

Description

`h = lmctest(y)` assesses the null hypothesis that a univariate time series `y` is a trend stationary AR(p) process, against the alternative that it is a nonstationary ARIMA(p,1,1) process.

`h = lmctest(y,'ParameterName',ParameterValue)` accepts one or more comma-separated parameter name/value pairs. Specify `ParameterName` inside single quotes. Perform multiple tests by passing a vector value for any parameter. Multiple tests yield vector results.

`[h,pValue] = lmctest(...)` returns p-values of the test statistics.

`[h,pValue,stat] = lmctest(...)` returns the test statistics.

`[h,pValue,stat,cValue] = lmctest(...)` returns critical values for the tests.

`[h,pValue,stat,cValue,reg1] = lmctest(...)` returns a structure of regression statistics from the maximum likelihood estimation of the reduced-form model.

`[h,pValue,stat,cValue,reg1,reg2] = lmctest(...)` returns a structure of regression statistics from the OLS estimation of the filtered data on a linear trend.

Input Arguments

`y`

Vector of time-series data. The last element is the most recent observation. The test ignores NaN values, which indicate missing entries.

Parameter Name/Value Pairs

`'alpha'`

Scalar or vector of nominal significance levels for the tests. Set values between 0.01 and 0.1.

Default: `0.05`

`'Lags'`

Scalar or vector of nonnegative integers indicating the number `p` of lagged values of `y` to include in the structural model (equal to the number `p` of lagged changes of `y` in the reduced-form model).

For best results, give a suitable value for `'lags'`. For information on selecting `'lags'`, see “Determine Appropriate Lags” on page 3-41.

Default: `0`

`'trend'`

Scalar or vector of Boolean values indicating whether or not to include the deterministic trend term `d*t` in the structural model (equivalent to including the drift term `d` in the reduced-form model).

Determine the value of `trend` by the growth characteristics of the time series `y`. Choose `trend` with a specific testing strategy in mind. If `y` is growing, set `trend` to `true` to provide a reasonable comparison of a trend-stationary null and a unit-root process with drift. If `y` does not exhibit long-term growth characteristics, set `trend` to `false`.

Default: `true`

`'test'`

String or cell vector of strings indicating which estimate of the variance σ_1^2 to use in computing the test statistic. Values are `'var1'` or `'var2'`.

Default: `'var2'`

Output Arguments

`h`

Vector of Boolean decisions for the tests, with length equal to the number of tests. Values of *h* equal to `1` indicate rejection of the AR(p) null in favor of the ARIMA(p,1,1) alternative. Values of *h* equal to `0` indicate a failure to reject the AR(p) null.

`pValue`

Vector of *p*-values of the test statistics, with length equal to the number of tests. Values are right-tail probabilities.

`stat`

Vector of test statistics, with length equal to the number of tests. For details, see "Test Statistics" on page 9-737.

`cValue`

Vector of critical values for the tests, with length equal to the number of tests. Values are for right-tail probabilities.

`reg1`

Structure of regression statistics from the maximum likelihood estimation of the reduced-form model. The structure is described in "Regression Statistics Structure" on page 9-738.

`reg2`

Structure of regression statistics The structure is described in "Regression Statistics Structure" on page 9-738.

Examples

Assess Whether a Series Is Trend Stationary and AR(p)

Test the growth of the U.S. unemployment rate using the data in Schwert, 1987.

Load Schwert's macroeconomic data set.

```
load Data_SchwertMacro
```

Focus on the unemployment rate growth over the dates condsidered in Leybourne and McCabe, 1999.

```
UN = DataTableMth.UN;
t1 = find(datesMth == datenum([1948 01 01]));
t2 = find(datesMth == datenum([1985 12 01]));
dUN = diff(UN(t1:t2)); % Unemployment rate growth
```

Assess the null hypothesis that the unemployment rate growth is a trend stationary, AR(1) process using the estimated variance from OLS regression.

```
[h1,~,stat1,cValue] = lmctest(dUN,'lags',1,'test','var1')
```

```
Warning: Test statistic #1 below tabulated critical values:
maximum p-value = 0.100 reported.

h1 =

     0


stat1 =

    0.0992


cValue =

    0.1460
```

The warning indicates that the pvalue is below 0.1. `h1` = 0 indicates that there is not enough evidence to reject that the unemployment rate growth is a trend stationary, AR(1) process.

Assess the null hypothesis that the unemployment rate growth is a trend stationary, AR(1) process using the estimated variance from the maximum liklihood of the reduced-form regression model.

```
[h2,~,stat2,cValue] = lmctest(dUN,'lags',1,'test','var2')
```

```
h2 =

     1


stat2 =

    0.1874


cValue =

    0.1460
```

`h2` = 1 indicates that the there is enough evidence to asuggest that the unemployment rate growth is nonstationary.

Leybourne and McCabe, 1999 report that the original LMC statistic fails to reject stationarity, while the modified LMC statistic does reject it.

More About

Model Equations

`lmctest` uses the structural model

$$y(t) = c(t) + \delta t + b_1 y(t-1) + \cdots + b_p y(t-p) + u_1(t)$$
$$c(t) = c(t-1) + u_2(t),$$

where

$$u_1(t) \sim \text{i.i.d.}\left(0, \sigma_1^2\right)$$
$$u_2(t) \sim \text{i.i.d.}\left(0, \sigma_2^2\right),$$

and u_1 and u_2 are independent of each other.

The model is second-order equivalent in moments to the reduced-form ARIMA(p,1,1) model
$(1 - L)y(t) = \delta + b_1(1 - L)y(t - 1) + ... + b_p(1 - L)y(t - p) + (1 - aL)v(t)$,

where L is the lag operator $Ly(t) = y(t–1)$, and $v(t) \sim$ i.i.d$(0,\sigma^2)$.

The null hypothesis is that $\sigma^2 = 0$ in the structural model, which is equivalent to $a = 1$ in the reduced-form model. The alternative is that $\sigma^2 > 0$ or $a < 1$. Under the null, the structural model is AR(p) with intercept $c(0)$ and trend δt; the reduced-form model is an over-differenced ARIMA(p,1,1) representation of the same process.

Test Statistics

`lmctest` computes test statistics using a two-stage method that first finds maximum likelihood estimates (MLEs) of coefficients in the reduced-form model. It then regresses the filtered data
$z(t) = y(t) - b_1y(t–1) - ... - b_py(t–p)$

on an intercept and, if `'trend'` is `true`, on a trend. It forms the `stat` test statistic using the residuals e from the first regression as follows:

$$\text{stat} = \frac{e^T V e}{s^2 T^2},$$

where $V(i,j) = \min(i,j)$, s^2 is an estimate of σ_1^2 that depends on the value of `test` (estimate of the variance), and T is the effective sample size.

Test Choices

You can choose between test values of `'var1'` and `'var2'`. These distinguish between the algorithm for estimating the variance σ_1^2.

- `'var1'` — The estimate is `(e'*e)/T`, where `e` is the residual vector from the OLS regression `reg2` and `T` is the effective sample size. This is the original Leybourne-McCabe test described in [3], with a rate of consistency O(`T`).

- `'var1'` — The estimate is `a*`σ^2, where `a` and σ^2 are MLEs from the estimation `reg1` of the reduced-form model. This is the modified Leybourne-McCabe test described in [4], with a rate of consistency $O(T^2)$.

Regression Statistics Structure

Lagging and differencing a time series reduces the sample size. Absent any presample values, if $y(t)$ is defined for t = 1:N, then the lagged series $y(t–k)$ is defined for t = k+1:N. Differencing reduces the time base to k+2:N. With p lagged differences, the common time base is p+2:N and the effective sample size is N – (p+1).

The maximum likelihood estimation of `reg1` regresses $Y = (1–L)y(t)$, with num = N–1, on p lagged changes of y, so that size = N – (p+1).

The OLS estimation of `reg2` regresses $Y = z(t)$, with num = N–p, on an intercept and, if `trend` is `true`, a trend, so that size = num.

The regression statistics structures have the following form:

`num`	Length of input series with `NaN`s removed
`size`	Effective sample size, adjusted for lags and difference
`names`	Regression coefficient names
`coeff`	Estimated coefficient values
`se`	Estimated coefficient standard errors
`Cov`	Estimated coefficient covariance matrix
`tStats`	t statistics of coefficients and p-values
`FStat`	F statistic and p-value
`yMu`	Mean of the lag-adjusted input series
`ySigma`	Standard deviation of the lag-adjusted input series
`yHat`	Fitted values of the lag-adjusted input series
`res`	Regression residuals
`DWStat`	Durbin-Watson statistic
`SSR`	Regression sum of squares
`SSE`	Error sum of squares
`SST`	Total sum of squares

`MSE`	Mean square error
`RMSE`	Standard error of the regression
`RSq`	R^2 statistic
`aRSq`	Adjusted R^2 statistic
`LL`	Loglikelihood of data under Gaussian innovations
`AIC`	Akaike information criterion
`BIC`	Bayesian (Schwarz) information criterion
`HQC`	Hannan-Quinn information criterion

Algorithms

Test statistics follow nonstandard distributions under the null, even asymptotically. Asymptotic critical values for a standard set of significance levels between 0.01 and 0.1, for models with and without a trend, have been tabulated in [2] using Monte Carlo simulations. Critical values and *p*-values reported by `lmctest` are interpolated from the tables. Tables are identical to those for `kpsstest`.

[1] shows that bootstrapped critical values, used by tests with a unit root null (such as `adftest` and `pptest`), are not possible for `lmctest`. As a result, size distortions for small samples may be significant, especially for highly persistent processes.

[3] shows that the test is robust when p takes values greater than the value in the data-generating process. [3] also notes simulation evidence that, under the null, the marginal distribution of the MLE of b_p is asymptotically normal, and so may be subject to a standard t-test for significance. Estimated standard errors, however, are unreliable in cases where the MA(1) coefficient a is near 1. As a result, [4] proposes another test for model order, valid under both the null and the alternative, that relies only on the MLEs of b_p and a, and not on their standard errors.

- "Unit Root Nonstationarity" on page 3-34

References

[1] Caner, M., and L. Kilian. "Size Distortions of Tests of the Null Hypothesis of Stationarity: Evidence and Implications for the PPP Debate." *Journal of International Money and Finance*. Vol. 20, 2001, pp. 639–657.

[2] Kwiatkowski, D., P. C. B. Phillips, P. Schmidt and Y. Shin. "Testing the Null Hypothesis of Stationarity against the Alternative of a Unit Root." *Journal of Econometrics*. Vol. 54, 1992, pp. 159–178.

[3] Leybourne, S. J., and B. P. M. McCabe. "A Consistent Test for a Unit Root." *Journal of Business and Economic Statistics*. Vol. 12, 1994, pp. 157–166.

[4] Leybourne, S. J., and B. P. M. McCabe. "Modified Stationarity Tests with Data-Dependent Model-Selection Rules." *Journal of Business and Economic Statistics*. Vol. 17, 1999, pp. 264–270.

[5] Schwert, G. W. "Effects of Model Specification on Tests for Unit Roots in Macroeconomic Data." *Journal of Monetary Economics*. Vol. 20, 1987, pp. 73–103.

See Also

`pptest` | `adftest` | `vratiotest` | `kpsstest`

Introduced in R2010a

lmtest

Lagrange multiplier test of model specification

Syntax

```
h = lmtest(score,ParamCov,dof)
h = lmtest(score,ParamCov,dof,alpha)
[h,pValue] = lmtest( ___ )
[h,pValue,stat,cValue] = lmtest( ___ )
```

Description

`h = lmtest(score,ParamCov,dof)` returns a logical value (`h`) with the rejection decision from conducting a Lagrange multiplier test of model specification at the 5% significance level. `lmtest` constructs the test statistic using the score function (`score`), the estimated parameter covariance (`ParamCov`), and the degrees of freedom (`dof`).

`h = lmtest(score,ParamCov,dof,alpha)` returns the rejection decision of the Lagrange multipler test conducted at significance level `alpha`.

- If `score` and `ParamCov` are length k cell arrays, then all other arguments must be length k vectors or scalars. `lmtest` treats each cell as a separate test, and returns a vector of rejection decisions.
- If `score` is a row cell array, then `lmtest` returns a row vector.

`[h,pValue] = lmtest( ___ )` returns the rejection decision and p-value (`pValue`) for the hypothesis test, using any of the input arguments in the previous syntaxes.

`[h,pValue,stat,cValue] = lmtest( ___ )` additionally returns the test statistic (`stat`) and critical value (`cValue`) for the hypothesis test.

Examples

Choose the Best AR Model Specification

Compare AR model specifications for a simulated response series using `lmtest`.

Consider the AR(3) model:

$$y_t = 1 + 0.9y_{t-1} - 0.5y_{t-2} + 0.4y_{t-3} + \varepsilon_t,$$

where ε_t is Gaussian with mean 0 and variance 1. Specify this model using `arima`.

```
Mdl = arima('Constant',1,'Variance',1,'AR',{0.9,-0.5,0.4});
```

`Mdl` is a fully specified, AR(3) model.

Simulate presample and effective sample responses from `Mdl`.

```
T = 100;
rng(1);                % For reproducibility
n = max(Mdl.P,Mdl.Q); % Number of presample observations
y = simulate(Mdl,T + n);
```

`y` is a a random path from `Mdl` that includes presample observations.

Specify the restricted model:

$$y_t = c + \phi_1 y_{t-1} + \phi_2 y_{t-2} + \varepsilon_t,$$

where ε_t is Gaussian with mean 0 and variance σ^2.

```
Mdl0 = arima(3,0,0);
Mdl0.AR{3} = 0;
```

The structure of `Mdl0` is the same as `Mdl`. However, every parameter is unknown, except that $\phi_3 = 0$. This is an equality constraint during estimation.

Estimate the restricted model using the simulated data (`y`).

```
[EstMdl0,EstParamCov] = estimate(Mdl0,y((n+1):end),...
    'Y0',y(1:n),'display','off');
phi10 = EstMdl0.AR{1};
phi20 = EstMdl0.AR{2};
phi30 = 0;
c0 = EstMdl0.Constant;
phi0 = [c0;phi10;phi20;phi30];
v0 = EstMdl0.Variance;
```

`EstMdl0` contains the parameter estimates of the restricted model.

`lmtest` requires the unrestricted model score evaluated at the restricted model estimates. The unrestricted model gradient is

$$\frac{\partial l(\phi_1,\phi_2,\phi_3,c,\sigma^2;y_t,...,y_{t-3})}{\partial c} = \frac{1}{\sigma^2}(y_t - c - \phi_1 y_{t-1} - \phi_2 y_{t-2} - \phi_3 y_{t-3})$$

$$\frac{\partial l(\phi_1,\phi_2,\phi_3,c,\sigma^2;y_t,...,y_{t-3})}{\partial \phi_j} = \frac{1}{\sigma^2}(y_t - c - \phi_1 y_{t-1} - \phi_2 y_{t-2} - \phi_3 y_{t-3})y_{t-j}$$

$$\frac{\partial l(\phi_1,\phi_2,\phi_3,c,\sigma^2;y_t,...,y_{t-3})}{\partial \sigma^2} = -\frac{1}{2\sigma^2} + \frac{1}{2\sigma^4}(y_t - c - \phi_1 y_{t-1} - \phi_2 y_{t-2} - \phi_3 y_{t-3})^2.$$

```
MatY = lagmatrix(y,1:3);
LagY = MatY(all(~isnan(MatY),2),:);
cGrad = (y((n+1):end)-[ones(T,1),LagY]*phi0)/v0;
phi1Grad = ((y((n+1):end)-[ones(T,1),LagY]*phi0).*LagY(:,1))/v0;
phi2Grad = ((y((n+1):end)-[ones(T,1),LagY]*phi0).*LagY(:,2))/v0;
phi3Grad = ((y((n+1):end)-[ones(T,1),LagY]*phi0).*LagY(:,3))/v0;
vGrad = -1/(2*v0)+((y((n+1):end)-[ones(T,1),LagY]*phi0).^2)/(2*v0^2);
Grad = [cGrad,phi1Grad,phi2Grad,phi3Grad,vGrad]; % Gradient matrix

score = sum(Grad)'; % Score under the restricted model
```

Evaluate the unrestricted parameter covariance estimator using the restricted MLEs and the outer product of gradients (OPG) method.

```
EstParamCov0 = inv(Grad'*Grad);
dof = 1; % Number of model restrictions
```

Test the null hypothesis that $\phi_3 = 0$ at a 1% significance level using `lmtest`.

```
[h,pValue] = lmtest(score,EstParamCov0,dof,0.1)
```

```
h =

     1


pValue =
```

```
    2.2524e-09
```

`pValue` is close to 0, which suggests that there is strong evidence to reject the restricted, AR(2) model in favor of the unrestriced, AR(3) model.

Assess Model Specifications Using the Lagrange Multiplier Test

Compare two model specifications for simulated education and income data. The unrestricted model has the following loglikelihood:

$$l(\beta,\rho) = -n\log\Gamma(\rho) + \rho\sum_{k=1}^{n}\log\beta_k + (\rho-1)\sum_{k=1}^{n}\log(y_k) - \sum_{k=1}^{n}y_k\beta_k,$$

where

- $\beta_k = \frac{1}{\beta+x_k}$.
- x_k is the number of grades that person k completed.
- y_k is the income (in thousands of USD) of person k.

That is, the income of person k given the number of grades that person k completed is Gamma distributed with shape ρ and rate β_i. The restricted model sets $\rho = 1$, which implies that the income of person k given the number of grades person k completed is exponentially distributed with mean $\beta + x_i$.

The restricted model is $H_0 : \rho = 1$. In order to compare this model to the unrestricted model, you require:

- The gradient vector of the unrestricted model
- The maximum likelihood estimate (MLE) under the restriced model
- The parameter covariance estimator evaluated under the MLEs of the restricted model

Load the data.

```
load Data_Income1
x = DataTable.EDU;
y = DataTable.INC;
```

Estimate the restricted model parameters by maximizing $l(\rho,\beta)$ with respect to β subject to the restriction $\rho = 1$. The gradient of $l(\rho,\beta)$ is

$$\frac{\partial l(\rho,\beta)}{\partial \beta} = \sum_{i=1}^{T}(y_i\beta_i^2 - \rho\beta_i)$$

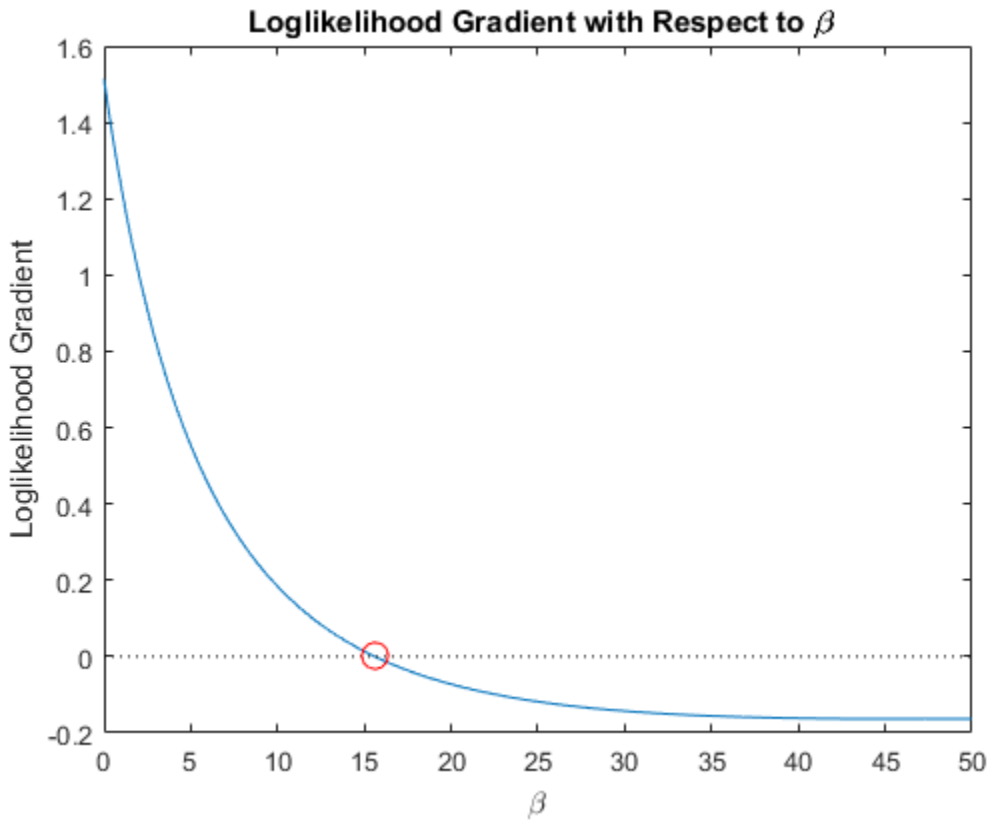
$$\frac{\partial l(\rho,\beta)}{\partial \rho} = -T\Psi(\rho) + \sum_{i=1}^{T}(\log \beta_i y_i),$$

where $\Psi(\rho)$ is the digamma function.

```
rho0 = 1; % Restricted rho
dof = 1;  % Number of restrictions
dLBeta = @(beta) sum(y./((beta + x).^2) - rho0./(beta + x));...
    % Anonymous gradient function

[betaHat0,fVal,exitFlag] = fzero(dLBeta,0)

beta = [0:0.1:50];
plot(beta,arrayfun(dLBeta,beta))
hold on
plot([beta(1);beta(end)],zeros(2,1),'k:')
plot(betaHat0,fVal,'ro','MarkerSize',10)
xlabel('{\beta}')
ylabel('Loglikelihood Gradient')
title('{\bf Loglikelihood Gradient with Respect to \beta}')
hold off
```

```
betaHat0 =

   15.6027


fVal =

   2.7756e-17


exitFlag =
```

1



The gradient with respect to β (`dLBeta`) is decreasing, which suggests that there is a local maximum at its root. Therefore, `betaHat0` is the MLE for the restricted model. `fVal` indicates that the value of the gradient is very close to 0 at `betaHat0`. The exit flag (`exitFlag`) is 1, which indicates that `fzero` found a root of the gradient without a problem.

Estimate the parameter covariance under the restricted model using the outer product of gradients (OPG).

```
rGradient = [-rho0./(betaHat0+x)+y.*(betaHat0+x).^(-2),...
```

```
      log(y./(betaHat0+x))-psi(rho0)];    % Gradient per unit
rScore = sum(rGradient)';                 % Score function
rEstParamCov = inv(rGradient'*rGradient); % Parameter covariance estimate
```

Test the unrestricted model against the restricted model using the Lagrange multipler test.

```
[h,pValue] = lmtest(rScore,rEstParamCov,dof)
```

```
h =

     1


pValue =

   7.4744e-05
```

`pValue` is close to 0, which indicates that there is strong evidence to suggest that the unrestricted model fits the data better than the restricted model.

Assess Conditional Heteroscedasticity Using the Lagrange Multiplier Test

Test whether there are significant ARCH effects in a simulated response series using `lmtest`. The parameter values in this example are arbitrary.

Specify the AR(1) model with an ARCH(1) variance:

$$y_t = 0.9y_{t-1} + \varepsilon_t,$$

where

- $\varepsilon_t = w_t\sqrt{h_t}$.
- $h_t = 1 + 0.5\varepsilon_{t-1}^2$.
- w_t is Gaussian with mean 0 and variance 1.

```
VarMdl = garch('ARCH',0.5,'Constant',1);
Mdl = arima('Constant',0,'Variance',VarMdl,'AR',0.9);
```

`Mdl` is a fully specified, AR(1) model with an ARCH(1) variance.

Simulate presample and effective sample responses from `Mdl`.

```
T = 100;
rng(1);  % For reproducibility
n = 2;   % Number of presample observations required for the gradient
[y,ep,v] = simulate(Mdl,T + n);
```

`ep` is the random path of innovations from `VarMdl`. The software filters `ep` through `Mdl` to yield the random response path `y`.

Specify the restricted model and assume that the AR model constant is 0:

$$y_t = c + \phi_1 y_{t-1} + \varepsilon_t,$$

where $h_t = \alpha_0 + \alpha_1 \varepsilon_{t-1}^2$.

```
VarMdl0 = garch(0,1);
VarMdl0.ARCH{1} = 0;
Mdl0 = arima('ARLags',1,'Constant',0,'Variance',VarMdl0);
```

The structure of `Mdl0` is the same as `Mdl`. However, every parameter is unknown, except for the restriction $\alpha_1 = 0$. These are equality constraints during estimation. You can interpret `Mdl0` as an AR(1) model with the Gaussian innovations that have mean 0 and constant variance.

Estimate the restricted model using the simulated data (`y`).

```
psI = 1:n;              % Presample indeces
esI = (n + 1):(T + n); % Estimation sample indeces

[EstMdl0,EstParamCov] = estimate(Mdl0,y(esI),...
    'Y0',y(psI),'E0',ep(psI),'V0',v(psI),'display','off');
phi10 = EstMdl0.AR{1};
alpha00 = EstMdl0.Variance.Constant;
```

`EstMdl0` contains the parameter estimates of the restricted model.

`lmtest` requires the unrestricted model score evaluated at the restricted model estimates. The unrestricted model loglikelihood function is

$$l(\phi_1, \alpha_0, \alpha_1) = \sum_{t=2}^{T} \left(-0.5 \log(2\pi) - 0.5 \log h_t - \frac{\varepsilon_t^2}{2h_t} \right),$$

where $\varepsilon_t = y_t - \phi_1 y_{t-1}$. The unrestricted gradient is

$$\frac{\partial l(\phi_1, \alpha_0, \alpha_1)}{\partial \alpha} = \sum_{t=2}^{T} \frac{1}{2h_t} z_t f_t,$$

where $z_t = [1, \varepsilon_{t-1}^2]$ and $f_t = \frac{\varepsilon_t^2}{h_t} - 1$. The information matrix is

$$I = \frac{1}{2h_t^2} \sum_{t=2}^{T} z_t' z_t.$$

Under the null, restricted model, $h_t = h_0 = \hat{\alpha}_0$ for all t, where $\hat{\alpha}_0$ is the estimate from the restricted model analysis.

Evaluate the gradient and information matrix under the restricted model. Estimate the parameter covariance by inverting the information matrix.

```
e = y - phi1O*lagmatrix(y,1);
eLag1Sq = lagmatrix(e,1).^2;
hO = alpha0O;
ft = (e(esI).^2/hO - 1);
zt = [ones(T,1),eLag1Sq(esI)]';

scoreO = 1/(2*hO)*zt*ft;        % Score function
InfoMatO = (1/(2*hO^2))*(zt*zt');
EstParamCovO = inv(InfoMatO);   % Estimated parameter covariance
dof = 1;                        % Number of model restrictions
```

Test the null hypothesis that $\alpha_1 = 0$ at the 5% significance level using `lmtest`.

```
[h,pValue] = lmtest(scoreO,EstParamCovO,dof)
```

```
h =

     1


pValue =
```

```
4.0443e-06
```

`pValue` is close to 0, which suggests that there is evidence to reject the restricted AR(1) model in favor of the unrestriced AR(1) model with an ARCH(1) variance.

- "Classical Model Misspecification Tests"
- "Conduct a Lagrange Multiplier Test" on page 3-70

Input Arguments

`score` — Unrestricted model loglikelihood gradients

vector | cell array of vectors

Unrestricted model loglikelihood gradients evaluated at the restricted model parameter estimates, specified as a vector or cell vector.

- For a single test, `score` can be a p-vector or a singleton cell array containing a p-by-1 vector. p is the number of parameters in the unrestricted model.
- For conducting $k > 1$ tests, `score` must be a length k cell array. Cell j must contain one p_j-by-1 vector that corresponds to one independent test. p_j is the number of parameters in the unrestricted model of test j.

Data Types: `double` | `cell`

`ParamCov` — Parameter covariance estimate

matrix | cell array of matrices

Parameter covariance estimate, specified as a symmetric matrix of cell array of symmetric matrices. `ParamCov` is the unrestricted model parameter covariance estimator evaluated at the restricted model parameter estimates.

- For a single test, `ParamCov` can be a p-by-p matrix or singleton cell array containing a p-by-p matrix. p is the number of parameters in the unrestricted model.
- For conducting $k > 1$ tests, `ParamCov` must be a length k cell array. Cell j must contain one p_j-by-p_j matrix that corresponds to one independent test. p_j is the number of parameters in the unrestricted model of test j.

Data Types: `double` | `cell`

`dof` — Degrees of freedom

positive integer | vector of positive integers

Degrees of freedom for the asymptotic, chi-square distribution of the test statistics, specified as a positive integer or vector of positive integers.

For each corresponding test, the elements of `dof`:

- Are the number of model restrictions
- Should be less than the number of parameters in the unrestricted model

When conducting $k > 1$ tests,

- If `dof` is a scalar, then the software expands it to a k-by-1 vector.
- If `dof` is a vector, then it must have length k.

`alpha` — Nominal significance levels

0.05 (default) | scalar | vector

Nominal significance levels for the hypothesis tests, specified as a scalar or vector.

Each element of `alpha` must be greater than 0 and less than 1.

When conducting $k > 1$ tests,

- If `alpha` is a scalar, then the software expands it to a k-by-1 vector.
- If `alpha` is a vector, then it must have length k.

Data Types: `double`

Output Arguments

`h` — Test rejection decisions

logical | vector of logicals

Test rejection decisions, returned as a logical value or vector of logical values with a length equal to the number of tests that the software conducts.

- `h = 1` indicates rejection of the null, restricted model in favor of the alternative, unrestricted model.

- `h = 0` indicates failure to reject the null, restricted model.

`pValue` — Test statistic *p*-values

scalar | vector

Test statistic *p*-values, returned as a scalar or vector with a length equal to the number of tests that the software conducts.

`stat` — Test statistics

scalar | vector

Test statistics, returned as a scalar or vector with a length equal to the number of tests that the software conducts.

`cValue` — Critical values

scalar | vector

Critical values determined by `alpha`, returned as a scalar or vector with a length equal to the number of tests that the software conducts.

More About

Lagrange Multiplier Test

This test compares specifications of nested models by assessing the significance of restrictions to an extended model with unrestricted parameters. The test statistic (*LM*) is

$$LM = S'VS,$$

where

- S is the gradient of the unrestricted loglikelihood function, evaluated at the restricted parameter estimates (`score`), i.e.,

$$S = \frac{\partial l(\theta)}{\partial \theta}\bigg|_{\theta=\theta_{0,MLE}}.$$

- V is the covariance estimator for the unrestricted model parameters, evaluated at the restricted parameter estimates.

If *LM* exceeds a critical value in its asymptotic distribution, then the test rejects the null, restricted (nested) model in favor of the alternative, unrestricted model.

The asymptotic distribution of *LM* is chi-square. Its degrees of freedom (`dof`) is the number of restrictions in the corresponding model comparison. The nominal significance level of the test (`alpha`) determines the critical value (`cValue`).

Tips

- `lmtest` requires the unrestricted model score and parameter covariance estimator evaluated at parameter estimates for the restricted model. For example, to compare competing, nested `arima` models:

  1. Analytically compute the score and parameter covariance estimator based on the innovation distribution.
  2. Use `estimate` to estimate the restricted model parameters.
  3. Evaluate the score and covariance estimator at the restricted model estimates.
  4. Pass the evaluated score, restricted covariance estimate, and the number of restrictions (i.e., the degrees of freedom) into `lmtest`.

- If you find estimating parameters in the unrestricted model difficult, then use `lmtest`. By comparison:

  - `waldtest` only requires unrestricted parameter estimates.
  - `lratiotest` requires both unrestricted and restricted parameter estimates.

Algorithms

- `lmtest` performs multiple, independent tests when inputs are cell arrays.

  - If the gradients and covariance estimates are the same for all tests, but the restricted parameter estimates vary, then `lmtest` "tests down" against multiple restricted models.
  - If the gradients and covariance estimates vary, but the restricted parameter estimates do not, then `lmtest` "tests up" against multiple unrestricted models.
  - Otherwise, `lmtest` compares model specifications pair-wise.

- `alpha` is nominal in that it specifies a rejection probability in the asymptotic distribution. The actual rejection probability can differ from the nominal significance. Lagrange multiplier tests tend to under-reject for small values of `alpha`, and over-reject for large values of `alpha`.

Lagrange multiplier tests typically yield lower rejection errors than likelihood ratio and Wald tests.

- "Model Comparison Tests" on page 3-65

References

[1] Davidson, R. and J. G. MacKinnon. *Econometric Theory and Methods*. Oxford, UK: Oxford University Press, 2004.

[2] Godfrey, L. G. *Misspecification Tests in Econometrics*. Cambridge, UK: Cambridge University Press, 1997.

[3] Greene, W. H. *Econometric Analysis*. 6th ed. Upper Saddle River, NJ: Pearson Prentice Hall, 2008.

[4] Hamilton, J. D. *Time Series Analysis*. Princeton, NJ: Princeton University Press, 1994.

See Also

arima | estimate | `lratiotest` | `vgxvarx` | `waldtest`

Introduced in R2009a

lratiotest

Likelihood ratio test of model specification

Syntax

```
h = lratiotest(uLogL,rLogL,dof)
h = lratiotest(uLogL,rLogL,dof,alpha)
[h,pValue] = lratiotest( ___ )
[h,pValue,stat,cValue] = lratiotest( ___ )
```

Description

`h = lratiotest(uLogL,rLogL,dof)` returns a logical value (`h`) with the rejection decision from conducting a likelihood ratio test of model specification.

`lratiotest` constructs the test statistic using the loglikelihood objective function evaluated at the unrestricted model parameter estimates (`uLogL`) and the restricted model parameter estimates (`rLogL`). The test statistic distribution has `dof` degrees of freedom.

- If `uLogL` or `rLogL` is a vector, then the other must be a scalar or vector of equal length. `lratiotest(uLogL,rLogL,dof)` treats each element of a vector input as a separate test, and returns a vector of rejection decisions.
- If `uLogL` or `rLogL` is a row vector, then `lratiotest(uLogL,rLogL,dof)` returns a row vector.

`h = lratiotest(uLogL,rLogL,dof,alpha)` returns the rejection decision of the likelihood ratio test conducted at significance level `alpha`.

`[h,pValue] = lratiotest( ___ )` returns the rejection decision and *p*-value (`pValue`) for the hypothesis test, using any of the input arguments in the previous syntaxes.

`[h,pValue,stat,cValue] = lratiotest( ___ )` additionally returns the test statistic (`stat`) and critical value (`cValue`) for the hypothesis test.

Examples

Assess Model Specifications Using the Likelihood Ratio Test

Compare two model specifications for simulated education and income data. The unrestricted model has the following loglikelihood:

$$l(\beta,\rho) = -n\log\Gamma(\rho) + \rho\sum_{k=1}^{n}\log\beta_k + (\rho-1)\sum_{k=1}^{n}\log(y_k) - \sum_{k=1}^{n}y_k\beta_k,$$

where

- $\beta_k = \frac{1}{\beta+x_k}$.
- x_k is the number of grades that person k completed.
- y_k is the income (in thousands of USD) of person k.

That is, the income of person k given the number of grades that person k completed is Gamma distributed with shape ρ and rate β_k. The restricted model sets $\rho = 1$, which implies that the income of person k given the number of grades person k completed is exponentially distributed with mean $\beta + x_k$.

The restricted model is $H_0 : \rho = 1$. Comparing this model to the unrestricted model using `lratiotest` requires the following:

- The loglikelihood function
- The maximum likelihood estimate (MLE) under the unrestricted model
- The MLE under the restricted model

Load the data.

```
load Data_Income1
x = DataTable.EDU;
y = DataTable.INC;
```

To estimate the unrestricted model parameters, maximize $l(\rho,\beta)$ with respect to ρ and β. The gradient of $l(\rho,\beta)$ is

$$\frac{\partial l(\rho,\beta)}{\partial \rho} = -n\psi(\rho) + \sum_{k=1}^{n} \log(y_k\beta_k)$$

$$\frac{\partial l(\rho,\beta)}{\partial \beta} = \sum_{k=1}^{n} \beta_k(\beta_k y_k - \rho),$$

where $\psi(\rho)$ is the digamma function.

```
nLogLGradFun = @(theta) deal(-sum(-gammaln(theta(1)) - ...
    theta(1)*log(theta(2) + x) + (theta(1)-1)*log(y) - ...
    y./(theta(2)+x)),...
    -[sum(-psi(theta(1))+log(y./(theta(2)+x)));...
    sum(1./(theta(2)+x).*(y./(theta(2)+x)-theta(1)))]);
```

`nLogLGradFun` is an anonymous function that returns the negative loglikelihood and the gradient given the input `theta`, which holds the parametes ρ and β, respectively.

Numerically optimize the negative loglikelihood function using `fmincon`, which minimizes an objective function subject to constraints.

```
theta0 = randn(2,1); % Initial value for optimization
uLB = [0 -min(x)];   % Unrestricted model lower bound
uUB = [Inf Inf];     % Unrestricted model upper bound
options = optimoptions('fmincon','Algorithm','interior-point',...
    'TolFun',1e-10,'Display','off','GradObj','on');...
    % Optimization options

[uMLE,uLogL] = fmincon(nLogLGradFun,theta0,[],[],[],[],uLB,uUB,[],options);
uLogL = -uLogL;
```

`uMLE` is the unrestricted maximum likelihood estimate, and `uLogL` is the loglikelihood maximum.

Impose the restriction to the loglikelihood by setting the corresponding lower and upper bound constraints of ρ to 1. Minimize the negative, restricted loglikelihood.

```
dof = 1;           % Number of restrictions
rLB = [1 -min(x)]; % Restricted model lower bound
rUB = [1 Inf];     % Restricted model upper bound
[rMLE,rLogL] = fmincon(nLogLGradFun,theta0,[],[],[],[],rLB,rUB,[],options);
```

```
rLogL = -rLogL;
```

`rMLE` is the unrestricted maximum likelihood estimate, and `rLogL` is the loglikelihood maximum.

Use the likelihood ratio test to assess whether the data provide enough evidence to favor the unrestricted model over the restricted model.

```
[h,pValue,stat] = lratiotest(uLogL,rLogL,dof)
```

```
h =

     1


pValue =

   8.9146e-04


stat =

   11.0404
```

`pValue` is close to 0, which indicates that there is strong evidence suggesting that the unrestricted model fits the data better than the restricted model.

Test Among Multiple Nested Model Specifications

Assess model specifications by testing down among multiple restricted models using simulated data. The true model is the ARMA(2,1)

$$y_t = 3 + 0.9y_{t-1} - 0.5y_{t-2} + \varepsilon_t + 0.7\varepsilon_{t-1},$$

where ε_t is Gaussian with mean 0 and variance 1.

Specify the true ARMA(2,1) model, and simulate 100 response values.

```
TrueMdl = arima('AR',{0.9,-0.5},'MA',0.7,...
    'Constant',3,'Variance',1);
T = 100;
```

```
rng(1); % For reproducibility
y = simulate(TrueMdl,T);
```

Specify the unrestriced model and the candidate models for testing down.

```
Mdl = {arima(2,0,2),arima(2,0,1),arima(2,0,0),arima(1,0,2),arima(1,0,1),...
    arima(1,0,0),arima(0,0,2),arima(0,0,1)};
rMdlNames = {'ARMA(2,1)','AR(2)','ARMA(1,2)','ARMA(1,1)',...
    'AR(1)','MA(2)','MA(1)'};
```

`Mdl` is a 1-by-7 cell array. `Mdl{1}` is the unrestricted model, and all other cells contain a candidate model.

Fit the candidate models to the simulated data.

```
logL = zeros(size(Mdl,1),1); % Preallocate loglikelihoods
dof = logL;                  % Preallocate degress of freedom
for k = 1:size(Mdl,2)
    [EstMdl,~,logL(k)] = estimate(Mdl{k},y,'Display','off');
    dof(k) = 4 - (EstMdl.P + EstMdl.Q); % Number of restricted parameters
end
uLogL = logL(1);
rLogL = logL(2:end);
dof = dof(2:end);
```

`uLogL` and `rLogL` are the values of the unrestricted loglikelihood evaluated at the unrestricted and restricted model parameter estimates, respectively.

Apply the likelihood ratio test at a 1% significance level to find the appropriate, restricted model specification(s).

```
alpha = .01;
h = lratiotest(uLogL,rLogL,dof,alpha);
RestrictedModels = rMdlNames(~h)
```

```
RestrictedModels =

    'ARMA(2,1)'    'ARMA(1,2)'    'ARMA(1,1)'    'MA(2)'
```

The most appropriate restricted models are ARMA(2,1), ARMA(1,2), ARMA(1,1), or MA(2).

You can test down again, but use ARMA(2,1) as the unrestricted model. In this case, you must remove MA(2) from the possible restricted models.

Assess Conditional Heteroscedasticity Using the Likelihood Ratio Test

Test whether there are significant ARCH effects in a simulated response series using `lratiotest`. The parameter values in this example are arbitrary.

Specify the AR(1) model with an ARCH(1) variance:

$$y_t = 0.9y_{t-1} + \varepsilon_t,$$

where

- $\varepsilon_t = w_t\sqrt{h_t}$.
- $h_t = 1 + 0.5\varepsilon_{t-1}^2$.
- w_t is Gaussian with mean 0 and variance 1.

```
VarMdl = garch('ARCH',0.5,'Constant',1);
Mdl = arima('Constant',0,'Variance',VarMdl,'AR',0.9);
```

`Mdl` is a fully specified AR(1) model with an ARCH(1) variance.

Simulate presample and effective sample responses from `Mdl`.

```
T = 100;
rng(1);  % For reproducibility
n = 2;   % Number of presample observations required for the gradient
[y,epsilon,condVariance] = simulate(Mdl,T + n);

psI = 1:n;             % Presample indices
esI = (n + 1):(T + n); % Estimation sample indices
```

`epsilon` is the random path of innovations from `VarMdl`. The software filters `epsilon` through `Mdl` to yield the random response path `y`.

Specify the unrestricted model assuming that the conditional mean model constant is 0:

$$y_t = \phi_1 y_{t-1} + \varepsilon_t,$$

where $h_t = \alpha_0 + \alpha_1\varepsilon_{t-1}^2$. Fit the simulated data (`y`) to the unrestricted model using the presample observations.

```
UVarMdl = garch(0,1);
UMdl = arima('ARLags',1,'Constant',0,'Variance',UVarMdl);
[~,~,uLogL] = estimate(UMdl,y(esI),'Y0',y(psI),'E0',epsilon(psI),...
    'V0',condVariance(psI),'Display','off');
```

`uLogL` is the maximimum value of the unrestricted loglikelihood function.

Specify the restricted model assuming that the conditional mean model constant is 0:

$$y_t = \phi_1 y_{t-1} + \varepsilon_t,$$

where $h_t = \alpha_0$. Fit the simulated data (`y`) to the restricted model using the presample observations.

```
RVarMdl = garch(0,1);
RVarMdl.ARCH{1} = 0;
RMdl = arima('ARLags',1,'Constant',0,'Variance',RVarMdl);
[~,~,rLogL] = estimate(RMdl,y(esI),'Y0',y(psI),'E0',epsilon(psI),...
    'V0',condVariance(psI),'Display','off');
```

The structure of `RMdl` is the same as `UMdl`. However, every parameter is unknown, except for the restriction. These are equality constraints during estimation. You can interpret `RMdl` as an AR(1) model with the Gaussian innovations that have mean 0 and constant variance.

Test the null hypothesis that $\alpha_1 = 0$ at the default 5% significance level using `lratoitest`.

```
dof = (UMdl.P + UMdl.Q + UVarMdl.P + UVarMdl.Q) ...
    - (RMdl.P + RMdl.Q + RVarMdl.P + RVarMdl.Q);
[h,pValue,stat,cValue] = lratiotest(uLogL,rLogL,dof)
```

```
h =

     1


pValue =

   6.7505e-04


stat =
```

```
   11.5567

cValue =

    3.8415
```

`h = 1` indicates that the null, restricted model should be rejected in favor of the alternative, unrestricted model. `pValue` is close to 0, suggesting that there is strong evidence for the rejection. `stat` is the value of the chi-square test statistic, and `cValue` is the critical value for the test.

- "Compare GARCH Models Using Likelihood Ratio Test" on page 3-77
- "Classical Model Misspecification Tests"

Input Arguments

`uLogL` — Unrestricted model loglikelihood maxima

scalar | vector

Unrestricted model loglikelihood maxima, specified as a scalar or vector. If `uLogL` is a scalar, then the software expands it to the same length as `rLogL`.

Data Types: `double`

`rLogL` — Restricted model loglikelihood maxima

scalar | vector

Restricted model loglikelihood maxima, specified as a scalar or vector. If `rLogL` is a scalar, then the software expands it to the same length as `uLogL`. Elements of `rLogL` should not exceed the corresponding elements of `uLogL`.

Data Types: `double`

`dof` — Degrees of freedom

positive integer | vector of positive integers

Degrees of freedom for the asymptotic, chi-square distribution of the test statistics, specified as a positive integer or vector of positive integers.

For each corresponding test, the elements of `dof`:

- Are the number of model restrictions
- Should be less than the number of parameters in the unrestricted model.

When conducting $k > 1$ tests,

- If `dof` is a scalar, then the software expands it to a k-by-1 vector.
- If `dof` is a vector, then it must have length k.

Data Types: `double`

`alpha` — Nominal significance levels

0.05 (default) | scalar | vector

Nominal significance levels for the hypothesis tests, specified as a scalar or vector.

Each element of `alpha` must be greater than 0 and less than 1.

When conducting $k > 1$ tests,

- If `alpha` is a scalar, then the software expands it to a k-by-1 vector.
- If `alpha` is a vector, then it must have length k.

Data Types: `double`

Output Arguments

`h` — Test rejection decisions

logical | vector of logicals

Test rejection decisions, returned as a logical value or vector of logical values with a length equal to the number of tests that the software conducts.

- `h = 1` indicates rejection of the null, restricted model in favor of the alternative, unrestricted model.
- `h = 0` indicates failure to reject the null, restricted model.

`pValue` — Test statistic *p*-values

scalar | vector

Test statistic *p*-values, returned as a scalar or vector with a length equal to the number of tests that the software conducts.

`stat` — Test statistics
scalar | vector

Test statistics, returned as a scalar or vector with a length equal to the number of tests that the software conducts.

`cValue` — Critical values
scalar | vector

Critical values determined by `alpha`, returned as a scalar or vector with a length equal to the number of tests that the software conducts.

More About

Likelihood Ratio Test

The *likelihood ratio test* compares specifications of nested models by assessing the significance of restrictions to an extended model with unrestricted parameters.

The test uses the following algorithm:

1. Maximize the loglikelihood function [$l(\theta)$] under the restricted and unrestricted model assumptions. Denote the MLEs for the restricted and unrestricted models $\hat{\theta}_0$ and $\hat{\theta}$, respectively.
2. Evaluate the loglikelihood objective function at the restricted and unrestricted MLEs, i.e., $\hat{l}_0 = l\left(\hat{\theta}_0\right)$ and $\hat{l} = l\left(\hat{\theta}\right)$.
3. Compute the likelihood ratio test statistic, $LR = 2\left(\hat{l} - \hat{l}_0\right)$.
4. If LR exceeds a critical value (C_a) relative to its asymptotic distribution, then reject the null, restricted model in favor of the alternative, unrestricted model.

   - Under the null hypothesis, LR is χ_d^2 distributed with d degrees of freedom.
   - The degrees of freedom for the test (d) is the number of restricted parameters.
   - The significance level of the test (α) determines the critical value (C_a).

Tips

- Estimate unrestricted and restricted univariate linear time series models, such as `arima` or `garch`, or time series regression models (`regARIMA`) using `estimate`. Estimate unrestricted and restricted multivariate linear time series models using `vgxvarx`.

  `estimate` and `vgxvarx` return loglikelihood maxima, which you can use as inputs to `lratiotest`.
- If you can easily compute both restricted and unrestricted parameter estimates, then use `lratiotest`. By comparison:
  - `waldtest` only requires unrestricted parameter estimates.
  - `lmtest` requires restricted parameter estimates.

Algorithms

- `lratiotest` performs multiple, independent tests when the unrestricted or restricted model loglikelihood maxima (`uLogL` and `rLogL`, respectively) is a vector.
  - If `rLogL` is a vector and `uLogL` is a scalar, then `lratiotest` "tests down" against multiple restricted models.
  - If `uLogL` is a vector and `rLogL` is a scalar, then `lratiotest` "tests up" against multiple unrestricted models.
  - Otherwise, `lratiotest` compares model specifications pair-wise.
- `alpha` is nominal in that it specifies a rejection probability in the asymptotic distribution. The actual rejection probability is generally greater than the nominal significance.

- Using garch Objects
- "Model Comparison Tests" on page 3-65

References

[1] Davidson, R. and J. G. MacKinnon. *Econometric Theory and Methods*. Oxford, UK: Oxford University Press, 2004.

[2] Godfrey, L. G. *Misspecification Tests in Econometrics*. Cambridge, UK: Cambridge University Press, 1997.

[3] Greene, W. H. *Econometric Analysis*. 6th ed. Upper Saddle River, NJ: Pearson Prentice Hall, 2008.

[4] Hamilton, J. D. *Time Series Analysis*. Princeton, NJ: Princeton University Press, 1994.

See Also

arima | estimate | `estimate` | estimate | `garch` | `lmtest` | regARIMA | `vgxvarx` | `waldtest`

Introduced before R2006a

minus

Class: LagOp

Lag operator polynomial subtraction

Syntax

```
C = minus(A, B, 'Tolerance', tolerance)
C = A -B
```

Description

Given two lag operator polynomials *A(L)* and *B(L)*, `C = minus(A, B, 'Tolerance', tolerance)` performs a polynomial subtraction *C(L) = A(L) – B(L)* with tolerance `tolerance`. `'Tolerance'` is the nonnegative scalar tolerance used to determine which coefficients are included in the result. The default tolerance is `1e–12`. Specifying a tolerance greater than `0` allows the user to exclude polynomial lags with near-zero coefficients. A coefficient matrix of a given lag is excluded only if the magnitudes of all elements of the matrix are less than or equal to the specified tolerance.

`C = A -B` performs a polynomial subtraction.

If at least one of `A` or `B` is a lag operator polynomial object, the other can be a cell array of matrices (initial lag operator coefficients), or a single matrix (zero-degree lag operator).

Examples

Subtract Two Lag Operator Polynomials

Create two `LagOp` polynomials and subtract one from the other:

```
A = LagOp({1 -0.6 0.08});
B = LagOp({1 -0.5});
A-B
```

```
ans =

    1-D Lag Operator Polynomial:
    ----------------------------
        Coefficients: [-0.1 0.08]
                Lags: [1 2]
              Degree: 2
           Dimension: 1
```

Algorithms

The subtraction operator (–) invokes `minus`, but the optional coefficient *`tolerance`* is available only by calling `minus` directly.

See Also

plus

mldivide

Class: LagOp

Lag operator polynomial left division

Syntax

```
B = A\C
B = mldivide(A, C'PropertyName',PropertyValue)
```

Description

Given two lag operator polynomials, *A(L)* and *C(L)*`B = A\C` perform a left division so that C(L) = *A*(*L*)**B*(*L*), or *B*(*L*) = *A*(*L*)*C*(*L*). Left division requires invertibility of the coefficient matrix associated with lag 0 of the denominator polynomial *A*(*L*).

`B = mldivide(A, C'PropertyName',PropertyValue)` accepts one or more comma-separated property name/value pairs.

Tips

The right division operator (\) invokes `mldivide`, but the optional inputs are available only by calling `mldivide` directly.

To right-invert a stable *B(L)*, set *C(L)* = `eye(B.Dimension)`.

Input Arguments

`A`

Denominator (divisor) lag operator polynomial object, as produced by `LagOp`, in the quotient *A(L)**C(L)*.

`C`

Numerator (dividend) lag operator polynomial object, as produced by `LagOp`, in the quotient *A(L)**C(L)*).

If at least one of *A* or *C* is a lag operator polynomial object, the other can be a cell array of matrices (initial lag operator coefficients), or a single matrix (zero-degree lag operator).

`'AbsTol'`

Nonnegative scalar absolute tolerance used as part of the termination criterion of the calculation of the quotient coefficients and, subsequently, to determine which coefficients to include in the quotient. Specifying an absolute tolerance allows for customization of the termination criterion. Once the algorithm has terminated, `'AbsTol'` is used to exclude polynomial lags with near-zero coefficients. A coefficient matrix for a given lag is excluded if the magnitudes of all elements of the matrix are less than or equal to the absolute tolerance.

Default: `1e-12`

`'RelTol'`

Nonnegative scalar relative tolerance used as part of the termination criterion of the calculation of the quotient coefficients. At each lag, a coefficient matrix is calculated and its 2-norm compared to the largest coefficient 2-norm. If the ratio of the current norm to the largest norm is less than or equal to `'RelTol'`, then the relative termination criterion is satisfied.

Default: `0.01`

`'Window'`

Positive integer indicating the size of the window used to check termination tolerances. `Window` represents the number of consecutive lags for which coefficients must satisfy a tolerance-based termination criterion in order to terminate the calculation of the quotient coefficients. If coefficients remain below tolerance for the length of the specified tolerance window, they are assumed to have died out sufficiently to terminate the algorithm (see notes below).

Default: `20`

`'Degree'`

Nonnegative integer indicating the maximum degree of the quotient polynomial. For stable denominators, the default is the power to which the magnitude of the largest eigenvalue of the denominator must be raised to equal the relative termination tolerance `'RelTol'`; for unstable denominators, the default is the power to which the magnitude of

the largest eigenvalue must be raised to equal the largest positive floating point number (see `realmax`). The default is `1000`, regardless of the stability of the denominator.

Default: `1000`

Output Arguments

`B`

Quotient lag operator polynomial object, such that *B(L) = A(L)\C(L)*.

Examples

Divide Lag Operator Polynomials

Create two `LagOp` polynomial objects:

```
A = LagOp({1 -0.6 0.08});
B = LagOp({1 -0.5});
```

The ratios `A/B` and `B\A` are equal:

```
isEqLagOp(A/B,B\A)
```

```
ans =

     1
```

Algorithms

Lag operator polynomial division generally results in infinite-degree polynomials. `mldivide` imposes a termination criterion to truncate the degree of the quotient polynomial.

If `'Degree'` is unspecified, the maximum degree of the quotient is determined by the stability of the denominator. Stable denominator polynomials usually result in quotients whose coefficients exhibit geometric decay in absolute value. (When coefficients change

sign, it is the coefficient envelope which decays geometrically.) Unstable denominators usually result in quotients whose coefficients exhibit geometric growth in absolute value. In either case, maximum degree will not exceed the value of `'Degree'`.

To control truncation error by terminating the coefficient sequence too early, the termination criterion involves three steps:

1. At each lag in the quotient polynomial, a coefficient matrix is calculated and tested against both a relative and an absolute tolerance (see `'RelTol'` and `'AbsTol'` inputs).
2. If the current coefficient matrix is below either tolerance, then a tolerance window is opened to ensure that all subsequent coefficients remain below tolerance for a number of lags determined by `'Window'`.
3. If any subsequent coefficient matrix within the window is above both tolerances, then the tolerance window is closed and additional coefficients are calculated, repeating steps (1) and (2) until a subsequent coefficient matrix is again below either tolerance, and a new window is opened.

Steps (1)-(3) are repeated until a coefficient is below tolerance and subsequent coefficients remains below tolerance for 'Window' lags, or until the maximum `'Degree'` is encountered, or until a coefficient becomes numerically unstable (`NaN` or `+/-Inf`).

References

[1] Box, G.E.P., G.M. Jenkins, and G.C. Reinsel. *Time Series Analysis: Forecasting and Control*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

[2] Hayashi, F. *Econometrics*. Princeton, NJ: Princeton University Press, 2000.

[3] Hamilton, J. D. *Time Series Analysis*. Princeton, NJ: Princeton University Press, 1994.

See Also

mrdivide

How To

- “Specify Lag Operator Polynomials” on page 2-11

- “Plot the Impulse Response Function” on page 5-88

mrdivide

Class: LagOp

Lag operator polynomial right division

Syntax

```
A = C/B
A = mrdivide(C, B,'PropertyName', PropertyValue)
```

Description

`A = C/B` returns the quotient lag operator polynomial (*A*), which is the result of *C*(*L*)/*B*(*L*).

`A = mrdivide(C, B,'PropertyName', PropertyValue)` accepts one or more optional comma-separated property name/value pairs.

Tips

The right division operator (/) invokes `mrdivide`, but the optional inputs are available only by calling `mrdivide` directly.

To right-invert a stable *B(L)*, set *C(L)* = eye(B.Dimension).

Input Arguments

`C`

Numerator (dividend) lag operator polynomial object, as produced by `LagOp`, in the quotient *C(L)/B(L)*.

`B`

Denominator (divisor) lag operator polynomial object, as produced by `LagOp`, in the quotient *C(L)/B(L)*.

If at least one of `C` or `B` is a lag operator polynomial object, the other can be a cell array of matrices (initial lag operator coefficients), or a single matrix (zero-degree lag operator).

`'AbsTol'`

Nonnegative scalar absolute tolerance used as part of the termination criterion of the calculation of the quotient coefficients and, subsequently, to determine which coefficients to include in the quotient. Specifying an absolute tolerance allows for customization of the termination criterion. Once the algorithm has terminated, `'AbsTol'` is used to exclude polynomial lags with near-zero coefficients. A coefficient matrix for a given lag is excluded if the magnitudes of all elements of the matrix are less than or equal to the absolute tolerance.

Default: `1e-12`

`'RelTol'`

Nonnegative scalar relative tolerance used as part of the termination criterion of the calculation of the quotient coefficients. At each lag, a coefficient matrix is calculated and its 2-norm compared to the largest coefficient 2-norm. If the ratio of the current norm to the largest norm is less than or equal to `'RelTol'`, then the relative termination criterion is satisfied.

Default: `0.01`

`'Window'`

Positive integer indicating the size of the window used to check termination tolerances. `Window` represents the number of consecutive lags for which coefficients must satisfy a tolerance-based termination criterion in order to terminate the calculation of the quotient coefficients. If coefficients remain below tolerance for the length of the specified tolerance window, they are assumed to have died out sufficiently to terminate the algorithm (see notes below).

Default: `20`

`'Degree'`

Nonnegative integer indicating the maximum degree of the quotient polynomial. For stable denominators, the default is the power to which the magnitude of the largest eigenvalue of the denominator must be raised to equal the relative termination tolerance `'RelTol'`; for unstable denominators, the default is the power to which the magnitude of

the largest eigenvalue must be raised to equal the largest positive floating point number (see `realmax`). The default is 1000, regardless of the stability of the denominator.

Default: 1000

Output Arguments

A

Quotient lag operator polynomial object, with $A(L) = C(L)/B(L)$.

Examples

Invert a Lag Operator Polynomial

Create a `LagOp` polynomial object with a sequence of scalar coefficients specified as a cell array:

```
A = LagOp({1 -0.5});
```

Invert the polynomial by using the short-hand slash ("/") operator:

```
a = 1 / A
```

```
a =

    1-D Lag Operator Polynomial:
    ----------------------------
        Coefficients: [1 0.5 0.25 0.125 0.0625 0.03125 0.015625]
                Lags: [0 1 2 3 4 5 6]
              Degree: 6
           Dimension: 1
```

Algorithms

Lag operator polynomial division generally results in infinite-degree polynomials. `mrdivide` imposes a termination criterion to truncate the degree of the quotient polynomial.

If `'Degree'` is unspecified, the maximum degree of the quotient is determined by the stability of the denominator. Stable denominator polynomials usually result in quotients whose coefficients exhibit geometric decay in absolute value. (When coefficients change sign, it is the coefficient envelope which decays geometrically.) Unstable denominators usually result in quotients whose coefficients exhibit geometric growth in absolute value. In either case, maximum degree will not exceed the value of `'Degree'`.

To control truncation error by terminating the coefficient sequence too early, the termination criterion involves three steps:

1. At each lag in the quotient polynomial, a coefficient matrix is calculated and tested against both a relative and an absolute tolerance (see `'RelTol'` and `'AbsTol'` inputs).
2. If the current coefficient matrix is below either tolerance, then a tolerance window is opened to ensure that all subsequent coefficients remain below tolerance for a number of lags determined by `'Window'`.
3. If any subsequent coefficient matrix within the window is above both tolerances, then the tolerance window is closed and additional coefficients are calculated, repeating steps (1) and (2) until a subsequent coefficient matrix is again below either tolerance, and a new window is opened.

The algorithm repeats steps 1–3 until a coefficient is below tolerance and subsequent coefficients remains below tolerance for `'Window'` lags, or until the maximum `'Degree'` is encountered, or until a coefficient becomes numerically unstable (`NaN` or `+/-Inf`).

References

[1] Box, G.E.P., G.M. Jenkins, and G.C. Reinsel. *Time Series Analysis: Forecasting and Control*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

[2] Hayashi, F. *Econometrics*. Princeton, NJ: Princeton University Press, 2000.

[3] Hamilton, J. D. *Time Series Analysis*. Princeton, NJ: Princeton University Press, 1994.

See Also

mldivide

mtimes

Class: LagOp

Lag operator polynomial multiplication

Syntax

```
C = mtimes(A, B, 'Tolerance',tolerance)
C = A * B
```

Description

Given two lag operator polynomials *A(L)* and *B(L)*,`C = mtimes(A, B, 'Tolerance',tolerance)` performs a polynomial multiplication *C(L) = A(L) * B(L)*. If at least one of `A` or `B` is a lag operator polynomial object, the other can be a cell array of matrices (initial lag operator coefficients), or a single matrix (zero-degree lag operator). '`Tolerance`' is the nonnegative scalar tolerance used to determine which coefficients are included in the result. The default tolerance is `1e-12`. Specifying a tolerance greater than `0` allows the user to exclude polynomial lags with near-zero coefficients. A coefficient matrix of a given lag is excluded only if the magnitudes of all elements of the matrix are less than or equal to the specified tolerance.

`C = A * B` performs a polynomial multiplication *C(L) = A(L) * B(L)*.

Tips

The multiplication operator (*) invokes `mtimes`, but the optional coefficient tolerance is available only by calling `mtimes` directly.

Examples

Multiply Two Lag Operator Polynomials

Create two `LagOp` polynomials and multiply them together:

```
A = LagOp({1 -0.6 0.08});
B = LagOp({1 -0.5});
mtimes(A,B)


ans =

    1-D Lag Operator Polynomial:
    ----------------------------
        Coefficients: [1 -1.1 0.38 -0.04]
                Lags: [0 1 2 3]
              Degree: 3
           Dimension: 1
```

See Also

mrdivide | mldivide

parcorr

Sample partial autocorrelation

Syntax

```
parcorr(y)
parcorr(y,numLags)
parcorr(y,numLags,numAR,numSTD)

pacf = parcorr(y)
pacf = parcorr(y,numLags)
pacf = parcorr(y,numLags,numAR,numSTD)
[pacf,lags,bounds] = parcorr(___)
```

Description

`parcorr(y)` plots the sample partial autocorrelation function (PACF) of the univariate, stochastic time series `y` with confidence bounds.

`parcorr(y,numLags)` plots the PACF, where `numLags` indicates the number of lags in the sample PACF.

`parcorr(y,numLags,numAR,numSTD)` plots the PACF, where `numAR` specifies the number of lags beyond which the theoretical PACF is effectively 0, and `numSTD` specifies the number of standard deviations of the sample PACF estimation error.

`pacf = parcorr(y)` returns the sample partial autocorrelation function (PACF) of the univariate, stochastic time series `y`.

`pacf = parcorr(y,numLags)` returns the PACF, where `numLags` specifies the number of lags in the sample PACF.

`pacf = parcorr(y,numLags,numAR,numSTD)` returns the PACF, where `numAR` specifies the number of lags beyond which the theoretical PACF is effectively 0, and `numSTD` specifies the number of standard deviations of the sample PACF estimation error.

`[pacf,lags,bounds] = parcorr( ___ )` additionally returns the lags (`lags`) corresponding to the PACF and the approximate upper and lower confidence bounds (`bounds`), using any of the input arguments in the previous syntaxes.

Examples

Plot the Partial Autocorrelation Function of a Time Series

Specify the AR(2) model:

$$y_t = 0.6y_{t-1} - 0.5y_{t-2} + \varepsilon_t,$$

where ε_t is Gaussian with mean 0 and variance 1.

```
rng(1); % For reproducibility
Mdl = arima('AR',{0.6 -0.5},'Constant',0,'Variance',1)
```

```
Mdl =

    ARIMA(2,0,0) Model:
    --------------------
    Distribution: Name = 'Gaussian'
               P: 2
               D: 0
               Q: 0
        Constant: 0
              AR: {0.6 -0.5} at Lags [1 2]
             SAR: {}
              MA: {}
             SMA: {}
        Variance: 1
```

Simulate 1000 observarions from `Mdl`.

```
y = simulate(Mdl,1000);
```

Compute the PACF.

```
[partialACF,lags,bounds] = parcorr(y,[],2);
```

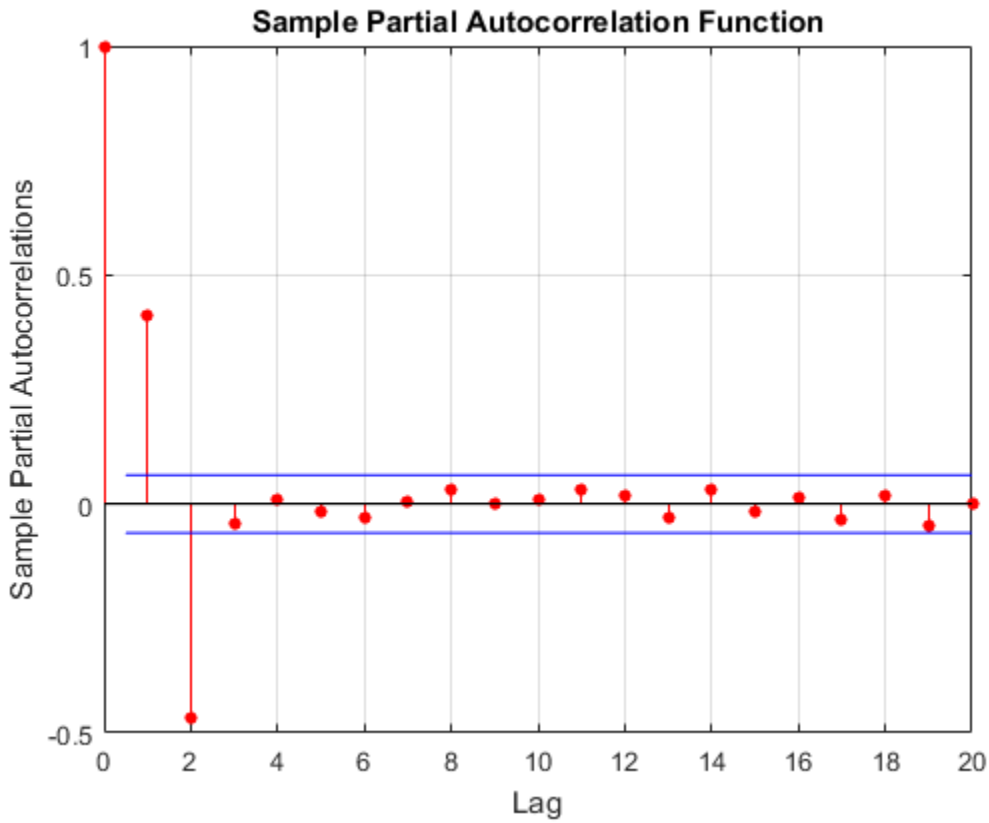
```
bounds
```

```
bounds =

    0.0633
   -0.0633
```

`bounds` displays (-0.0633, 0.0633), which are the upper and lower confidence bounds.

Plot the PACF.

```
parcorr(y)
```



The PACF cuts off after the second lag. This behavior indicates an AR(2) process.

Specify More Lags for the PACF Plot

Specify the multiplicative seasonal ARMA $(2,0,1) \times (3,0,0)_{12}$ model:

$$(1 - 0.75L - 0.15L^2)(1 - 0.9L^{12} + 0.75L^{24} - 0.5L^{36})y_t = 2 + \varepsilon_t - 0.5\varepsilon_{t-1},$$

where ε_t is Gaussian with mean 0 and variance 1.

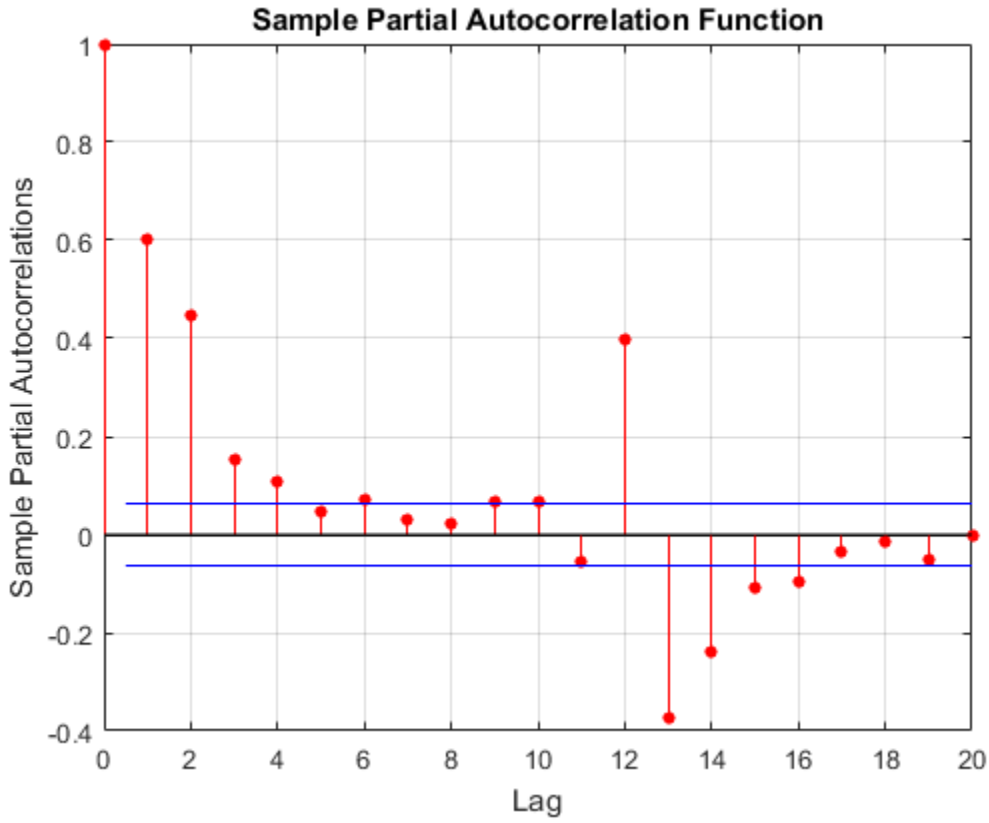
```
Mdl = arima('AR',{0.75,0.15},'SAR',{0.9,-0.75,0.5},...
    'SARLags',[12,24,36],'MA',-0.5,'Constant',2,...
    'Variance',1);
```

Simulate data from `Mdl`.

```
rng(1);
y = simulate(Mdl,1000);
```

Plot the default partial autocorrelation function (PACF).

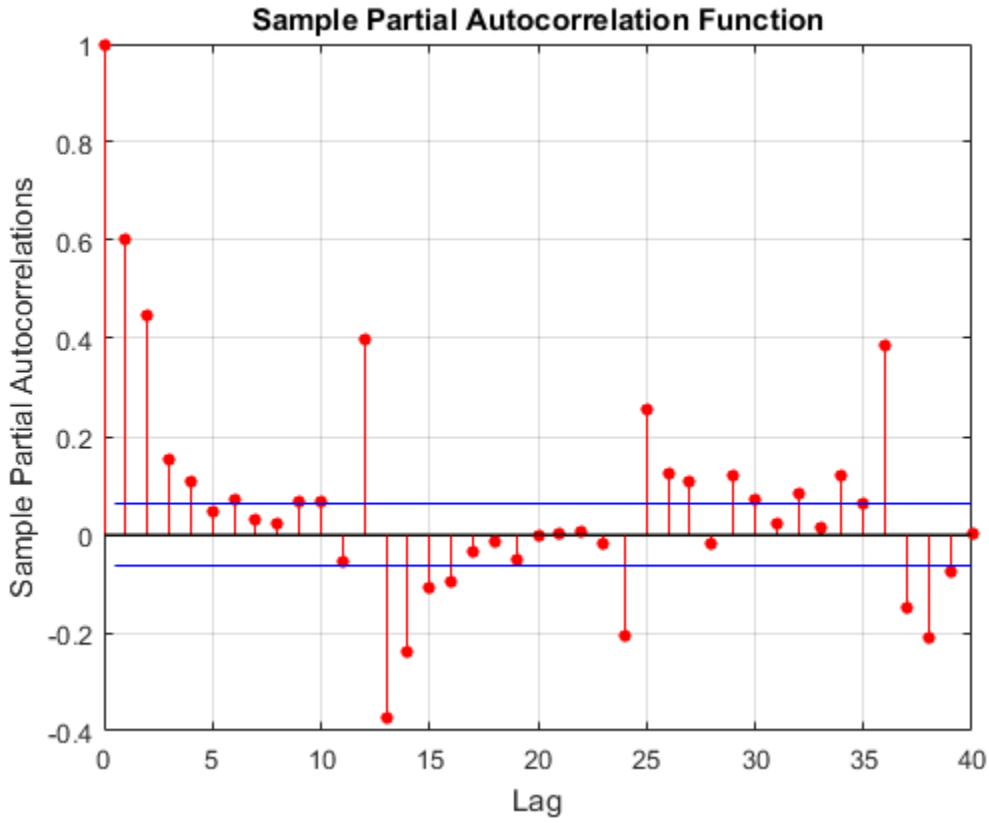
```
figure
parcorr(y)
```



The default correlogram does not display the dependence structure for higher lags.

Plot the PACF for 40 lags.

```
figure
parcorr(y,40)
```



The correlogram shows the larger correlations at lags 12, 24, and 36.

- "Box-Jenkins Model Selection" on page 3-4
- "Detect Autocorrelation" on page 3-18

Input Arguments

y — Observed univariate time series

vector

Observed univariate time series for which the software computes or plots the PACF, specified as a vector. The last element of `y` contains the most recent observation.

Data Types: `double`

`numLags` — Number of lags

`min(20,length(y)-1)` (default) | positive integer

Number of lags of the PACF that the software returns or plots, specified as a positive integer.

For example, `parcorr(y,10)` plots the PACF for lags 0 through 10.

Data Types: `double`

`numAR` — AR order

0 (default) | nonnegative integer

AR order that specifies the number of lags beyond which the theoretical PACF is effectively 0, specified as a nonnegative integer.

- `numAR` must be less than `numLags`.
- Specify `numAR` to assess whether the PACF is effectively 0 beyond lag `numAR`. Specifically, if `y` is an AR(`numAR`) process, then:
  - The PACF coefficient estimates at lags greater than `numAR` are approximately mean 0, independently distributed Gaussian variates.
  - The standard errors of the estimated PACF coefficients for lags greater than `numAR` of a length T series are $1/\sqrt{T}$ [1].

Example: `[~,~,bounds] = parcorr(y,[],5)`

Data Types: `double`

`numSTD` — Number of standard deviations

2 (default) | positive scalar

Number of standard deviations for the sample PACF estimation error assuming that `y` is an AR(`numAR`), specified as a positive scalar. For example, `parcorr(y,[],[],1.5)` plots the PACF with estimation error bounds 1.5 standard deviations away from 0.

If the software estimates the PACF coefficient at of lag `numAR` using `T` observations, then the confidence bounds are:

$$\pm\frac{\texttt{numSTD}}{\sqrt{T}}.$$

The default (`numSTD` = 2) corresponds to approximate 95% confidence bounds.

Data Types: `double`

Output Arguments

`pacf` — Sample PACF

vector

Sample PACF of the univariate time series `y`, returned as a vector of length `numLags + 1`.

The elements of `pacf` correspond to lags 0, 1, 2,... `numLags`.

The first element, which corresponds to lag 0, is unity (i.e., `pacf(1) = 1`). This corresponds to the coefficient of `y` regressed onto itself.

`lags` — Sample PACF lags

vector

Sample PACF lags, returned as a vector. Specifically, `lags = 0:numLags`.

`bounds` — Approximate confidence bounds

vector

Approximate confidence bounds of the PACF assuming `y` is an AR(`numAR`) process, returned as a two-element vector. `bounds` is approximate for `lags` > `numAR`.

More About

Partial Autocorrelation Function

Measures the correlation between y_t and y_{t+k} after adjusting for the linear effects of $y_{t+1},\ldots,y_{t+k-1}$.

The estimation of the PACF involves solving the Yule-Walker equations with respect to the autocorrelations. However, the software estimates the PACF by fitting successive autoregressive models of orders 1, 2,... using ordinary least squares. For details, see [1], Chapter 3.

Tips

To plot the ACF without confidence bounds, set `numSTD` to 0.

- "Box-Jenkins Methodology" on page 3-2
- "Autocorrelation and Partial Autocorrelation" on page 3-13

References

[1] Box, G. E. P., G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis: Forecasting and Control*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

[2] Hamilton, J. D. *Time Series Analysis*. Princeton, NJ: Princeton University Press, 1994.

See Also

arima | `crosscorr` | `filter`

Introduced before R2006a

plus

Class: LagOp

Lag operator polynomial addition

Syntax

```
C = plus(A, B, 'Tolerance', tolerance)
C = A + B
```

Description

Given two lag operator polynomials *A(L)* and *B(L)*, `C = plus(A, B, 'Tolerance', tolerance)` performs a polynomial addition *C(L) = A(L) + B(L)*with tolerance `tolerance`.`'Tolerance'` is the nonnegative scalar tolerance used to determine which coefficients are included in the result. The default tolerance is `1e-12`. Specifying a tolerance greater than `0` allows the user to exclude polynomial lags with near-zero coefficients. A coefficient matrix of a given lag is excluded only if the magnitudes of all elements of the matrix are less than or equal to the specified tolerance.

`C = A + B` performs a polynomial addition.

If at least one of `A` or `B` is a lag operator polynomial object, the other can be a cell array of matrices (initial lag operator coefficients), or a single matrix (zero-degree lag operator).

Algorithms

The addition operator (+) invokes plus, but the optional coefficient `tolerance` is available only by calling `plus` directly.

Examples

Add Two Lag Operator Polynomials

Create two `LagOp` polynomials and add them:

```
A = LagOp({1 -0.6 0.08});
B = LagOp({1 -0.5});
plus(A,B)


ans =

    1-D Lag Operator Polynomial:
    ----------------------------
        Coefficients: [2 -1.1 0.08]
                Lags: [0 1 2]
              Degree: 2
           Dimension: 1
```

See Also

minus

pptest

Phillips-Perron test for one unit root

Syntax

```
[h,pValue,stat,cValue,reg] = pptest(y)

[h,pValue,stat,cValue,reg] = pptest(y,'ParameterName',ParameterValue,...)
```

Description

Phillips-Perron tests assess the null hypothesis of a unit root in a univariate time series `y`. All tests use the model:
$y_t = c + \delta t + a\, y_{t-1} + e(t)$.

The null hypothesis restricts $a = 1$. Variants of the test, appropriate for series with different growth characteristics, restrict the drift and deterministic trend coefficients, c and δ, respectively, to be 0. The tests use modified Dickey-Fuller statistics (see `adftest`) to account for serial correlations in the innovations process $e(t)$.

Input Arguments

y

Vector of time-series data. The last element is the most recent observation. `NaN`s indicating missing values are removed.

Name-Value Pair Arguments

'lags'

Scalar or vector of nonnegative integers indicating the number of autocovariance lags to include in the Newey-West estimator of the long-run variance.

For best results, give a suitable value for `lags`. For information on selecting `lags`, see "Determining an Appropriate Number of Lags" on page 7-19.

Default: `0`

`'model'`

String or cell vector of strings indicating the model variant. Values are:

- `'AR'` (autoregressive)

  `pptest` tests the null model
  $y_t = y_{t-1} + e(t)$.

  against the alternative model
  $y_t = a\, y_{t-1} + e(t)$.

  with AR(1) coefficient $a < 1$.
- `'ARD'` (autoregressive with drift)

  `pptest` tests the `'AR'` null model against the alternative model
  $y_t = c + a\, y_{t-1} + e(t)$.

  with drift coefficient c and AR(1) coefficient $a < 1$.
- `'TS'` (trend stationary)

  `pptest` tests the null model
  $y_t = c + y_{t-1} + e(t)$.

  against the alternative model
  $y_t = c + \delta\, t + a\, y_{t-1} + e(t)$.

  with drift coefficient c, deterministic trend coefficient δ, and AR(1) coefficient $a < 1$.

Default: `'AR'`

`'test'`

String or cell vector of strings indicating the test statistic. Values are:

- `'t1'`

`pptest` computes a modification of the standard t statistic
$t_1 = (a - 1)/\text{se}$

from OLS estimates of the AR(1) coefficient and its standard error (se) in the alternative model. The test assesses the significance of the restriction $a - 1 = 0$.

- `'t2'`

  `pptest` computes a modification of the "unstudentized" t statistic
  $t_2 = T(a - 1)$

  from an OLS estimate of the AR(1) coefficient a and the stationary coefficients in the alternative model. T is the effective sample size, adjusted for lag and missing values. The test assesses the significance of the restriction $a - 1 = 0$.

Default: `'t1'`

`'alpha'`

Scalar or vector of nominal significance levels for the tests. Set values between `0.001` and `0.999`.

Default: `0.05`

Output Arguments

`h`

Vector of Boolean decisions for the tests, with length equal to the number of tests. Values of `h` equal to `1` indicate rejection of the unit-root null in favor of the alternative model. Values of `h` equal to `0` indicate a failure to reject the unit-root null.

`pValue`

Vector of p-values of the test statistics, with length equal to the number of tests. p-values are left-tail probabilities.

`stat`

Vector of test statistics, with length equal to the number of tests. Statistics are computed using OLS estimates of the coefficients in the alternative model.

cValue

Vector of critical values for the tests, with length equal to the number of tests. Values are for left-tail probabilities.

reg

Structure of regression statistics for the OLS estimation of coefficients in the alternative model. The number of records equals the number of tests. Each record has the following fields:

`num`	Length of input series with `NaN`s removed
`size`	Effective sample size, adjusted for lags
`names`	Regression coefficient names
`coeff`	Estimated coefficient values
`se`	Estimated coefficient standard errors
`Cov`	Estimated coefficient covariance matrix
`tStats`	t statistics of coefficients and *p*-values
`FStat`	F statistic and *p*-value
`yMu`	Mean of the lag-adjusted input series
`ySigma`	Standard deviation of the lag-adjusted input series
`yHat`	Fitted values of the lag-adjusted input series
`res`	Regression residuals
`autoCov`	Estimated residual autocovariances
`NWEst`	Newey-West estimator
`DWStat`	Durbin-Watson statistic
`SSR`	Regression sum of squares
`SSE`	Error sum of squares
`SST`	Total sum of squares
`MSE`	Mean square error
`RMSE`	Standard error of the regression
`RSq`	R^2 statistic
`aRSq`	Adjusted R^2 statistic

LL	Loglikelihood of data under Gaussian innovations
AIC	Akaike information criterion
BIC	Bayesian (Schwarz) information criterion
HQC	Hannan-Quinn information criterion

Definitions

The Phillips-Perron model is
$y_t = c + \delta t + a\, y_{t-1} + e(t)$.

where $e(t)$ is the innovations process.

The test assesses the null hypothesis under the model variant appropriate for series with different growth characteristics ($c = 0$ or $\delta = 0$).

Examples

Assess Stationarity Using the Phillips-Perron Test

Test GDP data for a unit root using a trend-stationary alternative with 0, 1, and 2 lags for the Newey-West estimator.

Load the GDP data set.

```
load Data_GDP
logGDP = log(Data);
```

Perform the Phillips-Perron test including 0, 1, and 2 autocovariance lags in the Newey-West robust covariance estimator.

```
h = pptest(logGDP,'model','TS','lags',0:2)
```

```
h =

     0     0     0
```

Each test returns `h = 0`, which means the test fails to reject the unit-root null hypothesis for each set of lags. Therefore, there is not enough evidence to suggest that log GDP is trend stationary.

More About

Algorithms

`pptest` performs a least-squares regression to estimate coefficients in the null model.

The tests use modified Dickey-Fuller statistics (see `adftest`) to account for serial correlations in the innovations process $e(t)$. Phillips-Perron statistics follow nonstandard distributions under the null, even asymptotically. Critical values for a range of sample sizes and significance levels have been tabulated using Monte Carlo simulations of the null model with Gaussian innovations and five million replications per sample size. `pptest` interpolates critical values and p-values from the tables. Tables for tests of type `'t1'` and `'t2'` are identical to those for `adftest`.

- "Unit Root Nonstationarity" on page 3-34

References

[1] Davidson, R., and J. G. MacKinnon. *Econometric Theory and Methods*. Oxford, UK: Oxford University Press, 2004.

[2] Elder, J., and P. E. Kennedy. "Testing for Unit Roots: What Should Students Be Taught?" *Journal of Economic Education*. Vol. 32, 2001, pp. 137–146.

[3] Hamilton, J. D. *Time Series Analysis*. Princeton, NJ: Princeton University Press, 1994.

[4] Newey, W. K., and K. D. West. "A Simple Positive Semidefinite, Heteroskedasticity and Autocorrelation Consistent Covariance Matrix." *Econometrica*. Vol. 55, 1987, pp. 703–708.

[5] Perron, P. "Trends and Random Walks in Macroeconomic Time Series: Further Evidence from a New Approach." *Journal of Economic Dynamics and Control*. Vol. 12, 1988, pp. 297–332.

[6] Phillips, P. “Time Series Regression with a Unit Root.” *Econometrica*. Vol. 55, 1987, pp. 277–301.

[7] Phillips, P., and P. Perron. “Testing for a Unit Root in Time Series Regression." *Biometrika*. Vol. 75, 1988, pp. 335–346.

[8] Schwert, W. “Tests for Unit Roots: A Monte Carlo Investigation.” *Journal of Business and Economic Statistics*. Vol. 7, 1989, pp. 147–159.

[9] White, H., and I. Domowitz. “Nonlinear Regression with Dependent Observations.” *Econometrica*. Vol. 52, 1984, pp. 143–162.

See Also

`adftest` | `kpsstest` | `vratiotest` | `lmctest`

Introduced in R2009b

price2ret

Convert prices to returns

Syntax

```
[RetSeries,RetIntervals] = ...
price2ret(TickSeries,TickTimes,Method)
```

Description

`[RetSeries,RetIntervals] = ...`
`price2ret(TickSeries,TickTimes,Method)` computes asset returns for NUMOBS price observations of NUMASSETS assets.

Input Arguments

`TickSeries`	Time series of price data. `TickSeries` can be a column vector or a matrix: • As a vector, `TickSeries` represents a univariate price series. The length of the vector is the number of observations (NUMOBS). The first element contains the oldest observation, and the last element the most recent. • As a matrix, `TickSeries` represents a NUMOBS-by-number of assets (NUMASSETS) matrix of asset prices. Rows correspond to time indices. The first row contains the oldest observations and the last row the most recent. `price2ret` assumes that the observations across a given row occur at the same time for all columns, where each column is a price series of an individual asset.
`TickTimes`	A NUMOBS element vector of monotonically increasing observation times. Times are numeric and taken either as serial date numbers (day units), or as decimal numbers in arbitrary units (for example, yearly). If `TickTimes` is `[]` or unspecified, then `price2ret` assumes sequential observation times from 1, 2, ..., NUMOBS.

`Method`	Character string indicating the compounding method to compute asset returns. If `Method` is `'Continuous'`, `[]`, or unspecified, then `price2ret` computes continuously compounded returns. If `Method = 'Periodic'`, then `price2ret` assumes simple periodic returns. `Method` is case insensitive.

Output Arguments

`RetSeries`	Array of asset returns: • When `TickSeries` is a `NUMOBS` element column vector, `RetSeries` is a `NUMOBS-1` column vector. • When `TickSeries` is a `NUMOBS`-by-`NUMASSETS` matrix, `RetSeries` is a `(NUMOBS-1)`-by-`NUMASSETS` matrix. `price2ret` quotes the *i*th return of an asset for the period `TickTimes(i)` to `TickTimes(i+1)`. It then normalizes it by the time interval between successive price observations. Assuming that `RetIntervals`(*i*) = `TickTimes`(*i*+1) – `TickTimes`(*i*) then if `Method` is `'Continuous'`, `[]`, or is unspecified, `price2ret` computes the continuously compounded returns as `RetSeries`(*i*) = log [`TickSeries`(*i*+1)/`TickSeries`(*i*)]/`RetIntervals`(*i*) If `Method` is `'Periodic'`, then `price2ret` computes the simple returns as `RetSeries`(*i*) = [`TickSeries`(*i*+1)/`TickSeries`(*i*)] – 1 /`RetIntervals`(*i*)
`RetIntervals`	`NUMOBS-1` element vector of times between observations. If `TickTimes` is `[]` or is unspecified, `price2ret` assumes that all intervals are `1`.

Examples

Convert a Stock Price Series to a Return Series

Create a stock price process continuously compounded at 10 percent:

```
S = 100*exp(0.10 * [0:19]');
    % Create the stock price series
```

Convert the price series to a 10 percent return series:

```
R = price2ret(S);   % Convert the price series to a
                    % 10 percent return series
[S [R;NaN]]  % Pad the return series so vectors are of
   % same length. price2ret computes the ith return from
   % the ith and i+1th prices.
```

```
ans =

  100.0000    0.1000
  110.5171    0.1000
  122.1403    0.1000
  134.9859    0.1000
  149.1825    0.1000
  164.8721    0.1000
  182.2119    0.1000
  201.3753    0.1000
  222.5541    0.1000
  245.9603    0.1000
  271.8282    0.1000
  300.4166    0.1000
  332.0117    0.1000
  366.9297    0.1000
  405.5200    0.1000
  448.1689    0.1000
  495.3032    0.1000
  547.3947    0.1000
  604.9647    0.1000
  668.5894       NaN
```

See Also

`ret2price` | `tick2ret`

Introduced before R2006a

print

Display parameter estimation results for conditional variance models

Syntax

```
print(Mdl,EstParamCov)
```

Description

`print(Mdl,EstParamCov)` displays parameter estimates, standard errors, and t statistics for the fitted conditional variance model `Mdl`, with estimated parameter variance-covariance matrix `EstParamCov`. `Mdl` can be a `garch`, `egarch`, or `gjr` model.

Examples

Print GARCH Estimation Results

Print the results from estimating a GARCH model using simulated data.

Simulate data from an GARCH(1,1) model with known parameter values.

```
modSim = garch('Constant',0.01,'GARCH',0.8,'ARCH',0.14)
rng 'default';
[V,Y] = simulate(modSim,100);
```

```
modSim =

    GARCH(1,1) Conditional Variance Model:
    --------------------------------------
    Distribution: Name = 'Gaussian'
               P: 1
               Q: 1
        Constant: 0.01
           GARCH: {0.8} at Lags [1]
            ARCH: {0.14} at Lags [1]
```

Fit a GARCH(1,1) model to the simulated data, turning off the print display.

```
model = garch(1,1);
[fit,VarCov] = estimate(model,Y,'print',false);
```

Print the estimation results.

```
print(fit,VarCov)
```

```
    GARCH(1,1) Conditional Variance Model:
    ----------------------------------------
    Conditional Probability Distribution: Gaussian

                                Standard          t
     Parameter       Value        Error       Statistic
    -----------   -----------   ------------   -----------
     Constant      0.0167004     0.0165077        1.01167
     GARCH{1}        0.77263     0.0776905        9.94498
      ARCH{1}       0.191686     0.0750675        2.55351
```

Print EGARCH Estimation Results

Print the results from estimating an EGARCH model using simulated data.

Simulate data from an EGARCH(1,1) model with known parameter values.

```
modSim = egarch('Constant',0.01,'GARCH',0.8,'ARCH',0.14,...
            'Leverage',-0.1);
rng 'default';
[V,Y] = simulate(modSim,100);
```

Fit an EGARCH(1,1) model to the simulated data, turning off the print display.

```
model = egarch(1,1);
[fit,VarCov] = estimate(model,Y,'print',false);
```

Print the estimation results.

```
print(fit,VarCov)
```

```
    EGARCH(1,1) Conditional Variance Model:
    ----------------------------------------
    Conditional Probability Distribution: Gaussian

                                Standard          t
     Parameter       Value        Error       Statistic
```

```
    -----------   -----------   ------------   -----------
     Constant      0.0654887     0.0746315       0.877494
     GARCH{1}        0.85807      0.154361        5.55886
      ARCH{1}        0.27702      0.171036        1.61966
  Leverage{1}      -0.179034      0.125057       -1.43162
```

Print GJR Estimation Results

Print the results from estimating a GJR model using simulated data.

Simulate data from a GJR(1,1) model with known parameter values.

```
modSim = gjr('Constant',0.01,'GARCH',0.8,'ARCH',0.14,...
             'Leverage',0.1);
rng 'default';
[V,Y] = simulate(modSim,100);
```

Fit a GJR(1,1) model to the simulated data, turning off the print display.

```
model = gjr(1,1);
[fit,VarCov] = estimate(model,Y,'print',false);
```

Print the estimation results.

```
print(fit,VarCov)
```

```
    GJR(1,1) Conditional Variance Model:
    ----------------------------------------
    Conditional Probability Distribution: Gaussian

                               Standard          t
     Parameter       Value       Error       Statistic
    -----------   -----------   ------------   -----------
     Constant       0.194785      0.254199       0.766271
     GARCH{1}        0.69954       0.11266        6.20928
      ARCH{1}       0.192966     0.0931335        2.07192
  Leverage{1}       0.214988      0.223923       0.960099
```

Input Arguments

`Mdl` — Conditional variance model

`garch` model object | `egarch` model object | `gjr` model object

Conditional variance model without any unknown parameters, specified as a `garch`, `egarch`, or `gjr` model object.

`Mdl` is usually the estimated conditional variance model returned by `estimate`.

`EstParamCov` — Estimated parameter variance-covariance matrix
numeric matrix

Estimated parameter variance-covariance matrix, returned as a numeric matrix.

`EstParamCov` is usually the estimated conditional variance model returned by `estimate`.

The rows and columns associated with any parameters contain the covariances. The standard errors of the parameter estimates are the square root of the entries along the main diagonal.

The rows and columns associated with any parameters held fixed as equality constraints during estimation contain `0`s.

The order of the parameters in `EstParamCov` must be:

- Constant
- Nonzero GARCH coefficients at positive lags
- Nonzero ARCH coefficients at positive lags
- For EGARCH and GJR models, nonzero leverage coefficients at positive lags
- Degrees of freedom (t innovation distribution only)
- Offset (models with nonzero offset only)

Data Types: `double`

More About

- Using garch Objects
- Using egarch Objects
- Using gjr Objects

See Also

`egarch` | `estimate` | `filter` | `forecast` | `garch` | `gjr` | `infer` | `simulate`

Introduced in R2012a

print

Class: arima

Display parameter estimation results for ARIMA or ARIMAX models

Syntax

```
print(EstMdl,EstParamCov)
```

Description

`print(EstMdl,EstParamCov)` displays parameter estimates, standard errors, and *t* statistics for a fitted ARIMA or ARIMAX model.

Input Arguments

`EstMdl`

`arima` model estimated using estimate.

`EstParamCov`

Estimation error variance-covariance matrix, as output by `estimate`. `EstParamCov` is a square matrix with a row and column for each parameter known to the optimizer when `Mdl` was fit by `estimate`. Known parameters include all parameters `estimate` estimated. If you specified a parameter as fixed during estimation, then it is also a known parameter and the rows and columns associated with it contain `0`s.

The parameters in `EstParamCov` are ordered as follows:

- Constant
- Nonzero AR coefficients at positive lags
- Nonzero SAR coefficients at positive lags
- Nonzero MA coefficients at positive lags
- Nonzero SMA coefficients at positive lags

- Regression coefficients (when `EstMdl` contains them)
- Variance parameters (scalar for constant-variance models, or a vector of parameters for a conditional variance model)
- Degrees of freedom (t innovation distribution only)

Examples

Print ARIMA Estimation Results

Print the results from estimating an ARIMA model using simulated data.

Simulate data from an ARMA(1,1) model using known parameter values.

```
MdlSim = arima('Constant',0.01,'AR',0.8,'MA',0.14,...
    'Variance',0.1);
rng 'default';
Y = simulate(MdlSim,100);
```

Fit an ARMA(1,1) model to the simulated data, turning off the print display.

```
Mdl = arima(1,0,1);
[EstMdl,EstParamCov] = estimate(Mdl,Y,'print',false);
```

Print the estimation results.

```
print(EstMdl,EstParamCov)
```

```
    ARIMA(1,0,1) Model:
    --------------------
    Conditional Probability Distribution: Gaussian

                               Standard          t
     Parameter       Value       Error       Statistic
    -----------   -----------   -----------   -----------
     Constant      0.0445373     0.0460376      0.967412
        AR{1}       0.822892     0.0711631       11.5635
        MA{1}        0.12032      0.101817       1.18173
     Variance       0.133727     0.0178793        7.4794
```

Print ARIMAX Estimation Results

Print the results of estimating an ARIMAX model.

Load the Credit Defaults data set, assign the response IGD to `Y` and the predictors AGE, CPF, and SPR to the matrix `X`, and obtain the sample size `T`. To avoid distraction from the purpose of this example, assume that all predictor series are stationary.

```
load Data_CreditDefaults
X = Data(:,[1 3:4]);
T = size(X,1);
y = Data(:,5);
```

Separate the initial values from the main response and predictor series.

```
y0 = y(1);
yEst = y(2:T);
XEst = X(2:end,:);
```

Set the ARIMAX(1,0,0) model $y_t = c + \phi_1 y_{t-1} + \varepsilon_t$ to `MdlY` to fit to the data.

```
MdlY = arima(1,0,0);
```

Fit the model to the data and specify the initial values.

```
[EstMdl,EstParamCov] = estimate(MdlY,yEst,'X',XEst,...
'Y0',y0,'print',false);
```

Print the estimation results.

```
print(EstMdl,EstParamCov)
```

```
    ARIMAX(1,0,0) Model:
    --------------------
    Conditional Probability Distribution: Gaussian

                                Standard          t
     Parameter       Value        Error       Statistic
    -----------   -----------   ------------   -----------
     Constant      -0.204768      0.266078      -0.769578
        AR{1}      -0.017309      0.565618      -0.030602
        Beta1       0.0239329     0.0218417       1.09574
        Beta2      -0.0124602    0.00749917      -1.66154
        Beta3       0.0680871     0.0745041       0.91387
     Variance      0.00539463    0.00224393        2.4041
```

See Also

arima | estimate | filter | forecast | impulse | infer | simulate

print

Class: regARIMA

Display estimation results for regression models with ARIMA errors

Syntax

```
print(Mdl,ParamCov)
```

Description

`print(Mdl,ParamCov)` displays parameter estimates, standard errors, and t statistics for the fitted regression model with ARIMA time series errors `Mdl`.

Input Arguments

`Mdl` — Regression model with ARIMA errors

`regARIMA` model

Regression model with ARIMA errors, specified as a regARIMA model returned by `regARIMA` or estimate.

`ParamCov` — Estimation error variance-covariance

numeric matrix

Estimation error variance-covariance, specified as a numeric matrix.

`ParamCov` is a square matrix with a row and column for each parameter known to the optimizer that `estimate` uses to fit `Mdl`. Known parameters include all parameters `estimate` estimates. If you specify a parameter as fixed during estimation, then it is also a known parameter and the rows and columns associated with it contain `0`s.

`print` omits coefficients of lag operator polynomials at lags excluded from `Mdl`.

`print` orders the parameters in `ParamCov` as follows:

- Intercept
- Nonzero AR coefficients at positive lags
- Nonzero SAR coefficients at positive lags
- Nonzero MA coefficients at positive lags
- Nonzero SMA coefficients at positive lags
- Regression coefficients (when `Mdl` contains them)
- Variance
- Degrees of freedom for the *t*-distribution

Data Types: `double`

Examples

Print Estimation Results of a Regression Model with ARIMA Errors Fit

Regress GDP onto CPI using a regression model with ARMA(1,1) errors, and print the results.

Load the US Macroeconomic data set and preprocess the data.

```
load Data_USEconModel;
logGDP = log(DataTable.GDP);
dlogGDP = diff(logGDP);
dCPI = diff(DataTable.CPIAUCSL);
```

Fit the model to the data.

```
ToEstMdl = regARIMA('ARLags',1,'MALags',1);
[EstMdl,EstParamCov] = estimate(ToEstMdl,dlogGDP,'X',...
   dCPI,'Display','off');
```

Print the estimates.

```
print(EstMdl,EstParamCov)
```

```
    Regression with ARIMA(1,0,1) Error Model:
    ----------------------------------------
    Conditional Probability Distribution: Gaussian
```

```
                          Standard          t
 Parameter       Value      Error       Statistic
-----------   -----------  ------------  -----------
Intercept       0.014776   0.00146271       10.1018
    AR{1}       0.605274    0.0892902       6.77872
    MA{1}      -0.161651      0.10956      -1.47546
    Beta1     0.00204403  0.000706163       2.89456
 Variance    9.35782e-05  6.03135e-06       15.5153
```

See Also

regARIMA | estimate

recessionplot

Overlay recession bands on a time series plot

Syntax

```
recessionplot
recessionplot(Name,Value)

hBands = recessionplot( ___ )
```

Description

`recessionplot` overlays shaded recession bands on a time series plot.

`recessionplot(Name,Value)` uses additional options specified by one or more `Name,Value` pairs.

`hBands = recessionplot( ___ )` returns a vector of handles to the recession bands, using any of the previous input arguments.

Examples

Overlay Recession Bands

Overlay recession bands on a plot of multiple time series.

Load data on credit defaults, and extract the predictor variables in the first four columns.

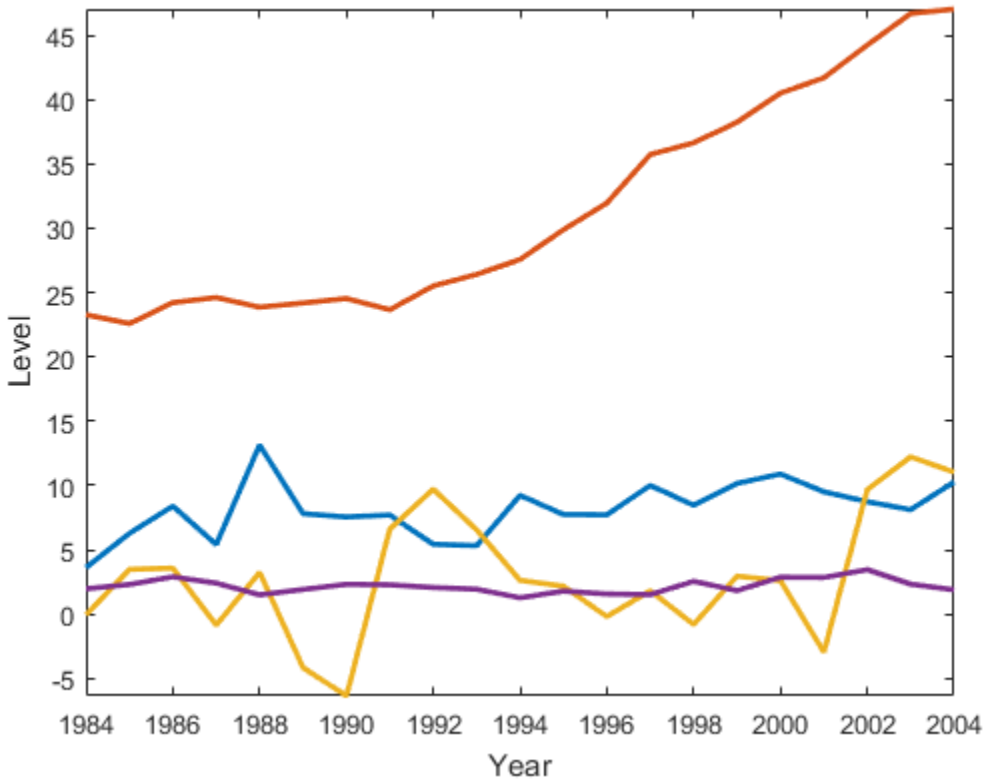
```
load Data_CreditDefaults
X0 = Data(:,1:4);
T0 = size(X0,1);
```

Convert the dates to serial date numbers, as required by `recessionplot`.

```
dates = datenum([dates,ones(T0,2)]);
```

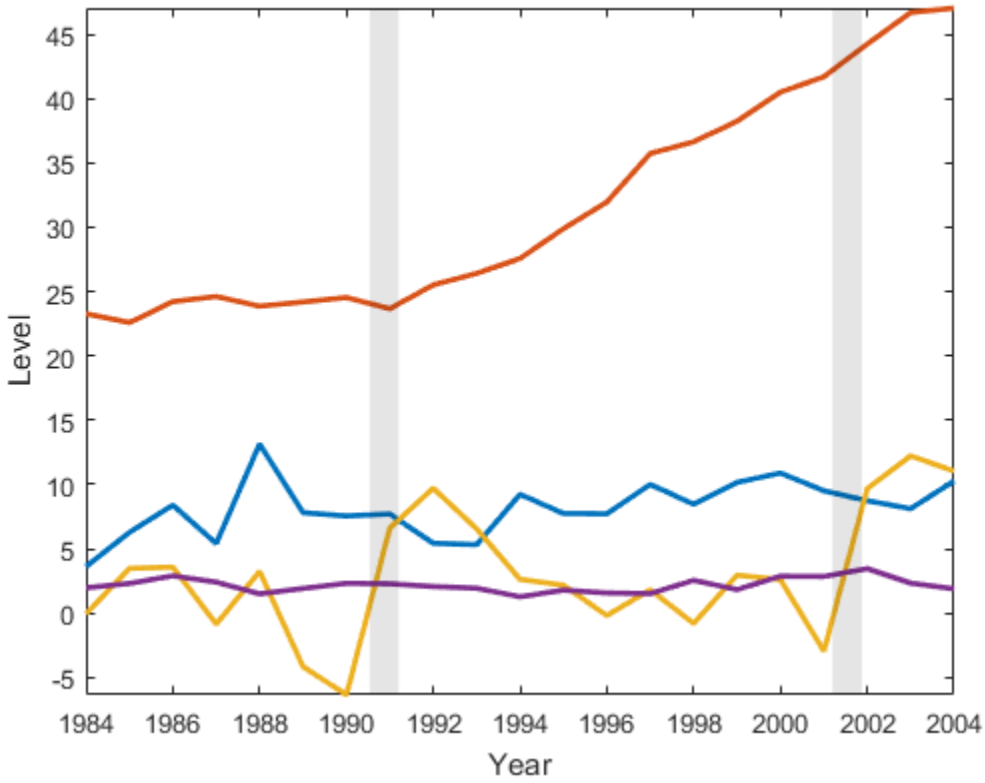
Create a time series plot of the four credit default predictors.

```
figure;
plot(dates,X0,'LineWidth',2);
h = gca;
h.XTick = dates(1:2:end);
datetick('x','yyyy','keepticks')
xlabel 'Year';
ylabel 'Level';
axis tight;
```



Overlay recession bands corresponding to U.S. recessions reported by the National Bureau of Economic Research.

```
recessionplot
```



The plots shows that two recessions occurred within the range of the time series.

Change Color and Transparency of Recession Bands

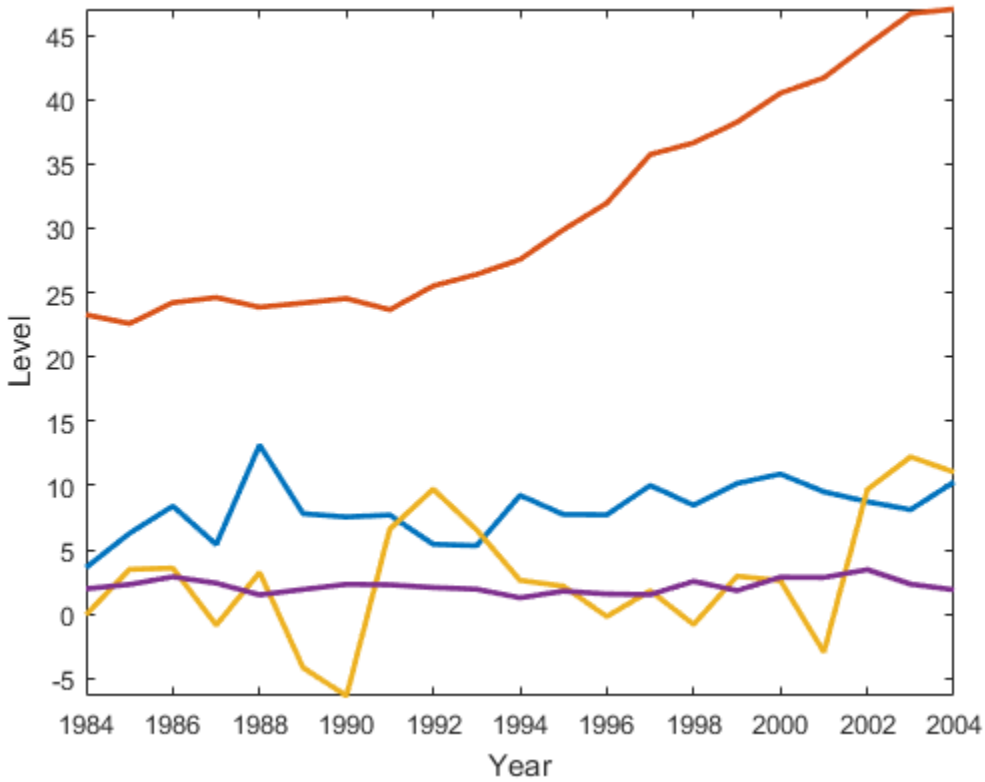
Overlay recession bands on a plot of multiple time series. Return the handles of the recession bands so you can change their color and transparency.

Load data on credit defaults, and extract the predictor variables in the first four columns.

```
load Data_CreditDefaults
X0 = Data(:,1:4);
T0 = size(X0,1);
```

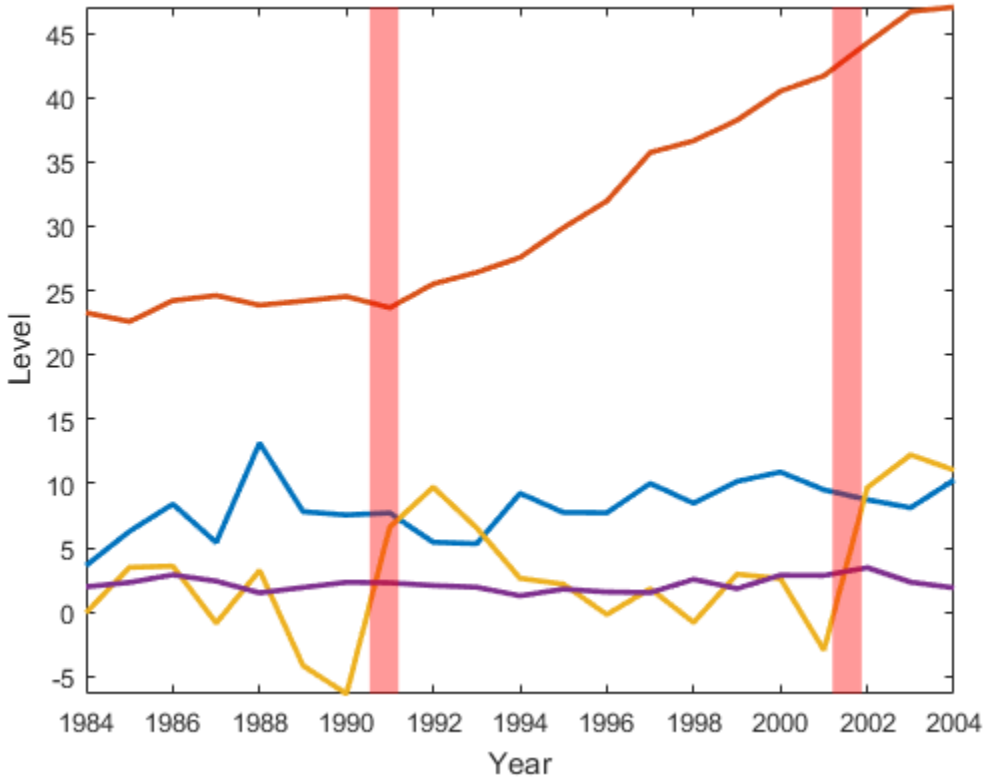
Convert dates to serial date numbers, and then plot the four time series.

```
dates = datenum([dates,ones(T0,2)]);

figure;
plot(dates,X0,'LineWidth',2);
h = gca;
h.XTick = dates(1:2:end);
datetick('x','yyyy','keepticks')
xlabel 'Year';
ylabel 'Level';
axis tight
```



Overlay recession bands, returning the handles to the bands. Change the band color to red and increase the transparency.

```
hBands = recessionplot;
set(hBands,'FaceColor','r','FaceAlpha',0.4)
```



Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: 'axes',h1 overlays recession bands on the axes identified by the handle h1

'axes' — Handle to axes

gca (default) | handle

Handle to axes displaying a time series plot, specified as the comma-separated pair consisting of 'axes' and an axes handle. The time series plot must have serial date numbers on the horizontal axis

Example: 'axes',h1

'recessions' — Recession data

Data_Recessions.mat (default) | matrix

Recession data indicating the beginning and end of historical recessions, specified as the comma-separated pair consisting of 'recessions' and a numRecessions-by-2 matrix of serial date numbers. The first column indicates the beginning of the recession, and the second column indicates the end of the recession. The default recession data is the U.S. recession data in Data_Recessions.mat, reported by the National Bureau of Economic Research.

Output Arguments

hBands — Handles

vector

Handles to the recession bands, returned as a vector of handles.

More About

Tips

- recessionplot requires that you express dates on the horizontal axis of a time series plot as serial date numbers. To convert other date information to this format before plotting, use datenum.
- Use the output handles to change the color and transparency of the recession bands by setting their FaceColor and FaceAlpha properties. This might be necessary to achieve satisfactory displays when working with certain monitors and projectors.

See Also

`datenum`

Introduced in R2012a

refine

Class: dssm

Refine initial parameters to aid diffuse state-space model estimation

Syntax

```
refine(Mdl,Y,params0)
refine(Mdl,Y,params0,Name,Value)
Output = refine( ___ )
```

Description

`refine(Mdl,Y,params0)` finds a set of initial parameter values to use when fitting the state-space model `Mdl` to the response data `Y` ,using the crude set of initial parameter values `params0`. The software uses several routines, and displays the resulting loglikelihood and initial parameter values for each routine.

`refine(Mdl,Y,params0,Name,Value)` displays results of the routines with additional options specified by one or more `Name,Value` pair arguments. For example, you can include a linear regression component composed of predictors and an initial value for the coefficients.

`Output = refine( ___ )` returns a structure array (`Output`) containing a vector of refined, initial parameter values, the loglikelihood corresponding the initial parameter values, and the computation method yielding the values. You can use any of the input arguments in the previous syntaxes.

Tips

- Likelihood surfaces of state-space models can be complicated, for example, they can contain multiple local maxima. If estimate fails to converge, or converges to an unsatisfactory solution, then `refine` can find a better set of initial parameter values to pass to `estimate`.

- The refined initial parameter values returned by `refine` can appear similar to each other and to `params0`. Choose a set yielding estimates that make economic sense and correspond to relatively large loglikelihood values.
- If a refinement attempt fails, then the software displays errors and sets the corresponding loglikelihood to `-Inf`. It also sets its initial parameter values to `[]`.

Input Arguments

`Mdl` — Diffuse state-space model

`dssm` model object

Diffuse state-space model containing unknown parameters, specified as a `dssm` model object returned by dssm.

`Mdl` does not store observed responses or predictor data. Supply the data wherever necessary using, the appropriate input and name-value pair arguments.

`Y` — Observed response data

numeric matrix | cell vector of numeric vectors

Observed response data to which `Mdl` is fit, specified as a numeric matrix or a cell vector of numeric vectors.

- If `Mdl` is time invariant with respect to the observation equation, then `Y` is a T-by-n matrix. Each row of the matrix corresponds to a period and each column corresponds to a particular observation in the model. Therefore, T is the sample size and n is the number of observations per period. The last row of `Y` contains the latest observations.
- If `Mdl` is time varying with respect to the observation equation, then `Y` is a T-by-1 cell vector. `Y{t}` contains an n_t-dimensional vector of observations for period t, where $t = 1,...,T$. The corresponding dimensions of the coefficient matrices in `Mdl.C{t}` and `Mdl.D{t}` must be consistent with the matrix in `Y{t}` for all periods. The last cell of `Y` contains the latest observations.
- Suppose that you created `Mdl` implicitly by specifying a parameter-to-matrix mapping function, and the function has input arguments for the observed responses or predictors. The mapping function establishes a link to observed responses and the predictor data in the MATLAB workspace, which overrides the value of `Y`.

`NaN` elements indicate missing observations. For details on how the Kalman filter accommodates missing observations, see “Algorithms” on page 9-829.

Data Types: `double | cell`

`params0` — Initial values of unknown parameters

numeric vector

Initial values of unknown parameters for numeric maximum likelihood estimation, specified as a numeric vector.

The elements of `params0` correspond to the unknown parameters in the state-space model matrices `A`, `B`, `C`, and `D`, and, optionally, the initial state mean `Mean0` and covariance matrix `Cov0`.

- If you created `Mdl` explicitly (that is, by specifying the matrices without a parameter-to-matrix mapping function), then the software maps the elements of `params` to `NaN`s in the state-space model matrices and initial state values. The software searches for `NaN`s column-wise, following the order `A`, `B`, `C`, `D`, `Mean0`, `Cov0`.
- If you created `Mdl` implicitly (that is, by specifying the matrices with a parameter-to-matrix mapping function), then set initial parameter values for the state-space model matrices, initial state values, and state types within the parameter-to-matrix mapping function.

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'Beta0'` — Initial values of regression coefficients

numeric matrix

Initial values of regression coefficients, specified as the comma-separated pair consisting of `'Beta0'` and a d-by-n numeric matrix. d is the number of predictor variables (see `Predictors`) and n is the number of observed response series (see `Y`).

By default, `Beta0` is the ordinary least-squares estimate of `Y` onto `Predictors`.

Data Types: `double`

`'Predictors'` — Predictor data

[ ] (default) | numeric matrix

Predictor data used to deflate the observations in a time-invariant state-space model, specified as the comma-separated pair consisting of `'Predictors'` and a T-by-d numeric matrix. T is the number of periods and d is the number of predictor variables. Row t corresponds to the observed predictors at period t (Z_t) in the expanded observation equation

$$y_t - Z_t\beta = Cx_t + Du_t.$$

That is, the software deflates the observations using the regression component. β is the time-invariant vector of regression coefficients that the software estimates with all other parameters. `Predictors` and `Y` must have the same number of rows.

For n observations per period, the software regresses all predictor series onto each observation. Then, the software returns a d-by-n matrix of fitted regression coefficient vectors for each observation series.

If you specify `Predictors`, then `Mdl` must be time invariant. Otherwise, the software returns an error.

By default, the software excludes a regression component from the state-space model.

Data Types: `double`

Output Arguments

`Output` — Information about initial parameter values

structure array

Information about the initial parameter values, returned as a 1-by-5 structure array. The software uses five algorithms to find initial parameter values, and each element of `Output` corresponds to an algorithm.

This table describes the fields of `Output`.

Field	Description
`Description`	Refinement algorithm.

Field	Description

	Each element of `Output` corresponds to one of these algorithms:
	
	`'Loose bound interior point'`
	`'Nelder-Mead algorithm'`
	`'Quasi-Newton'`
	`'Starting value perturbation'`
	`'Starting value shrinkage'`

`Loglikelihood`	Loglikelihood corresponding to the initial parameter values.
`Parameters`	Vector of refined initial parameter values. The order of the parameters is the same as the order in `params0`. If you pass these initial values to `estimate`, then the estimation results can improve.

Examples

Refine Parameters When Fitting Time-Invariant Diffuse State-Space Model

Suppose that a latent process is a random walk. Subsequently, the state equation is

$$x_t = x_{t-1} + u_t,$$

where u_t is Gaussian with mean 0 and standard deviation 1.

Generate a random series of 100 observations from x_t, assuming that the series starts at 1.5.

```
T = 100;
rng(1); % For reproducibility
u = randn(T,1);
x = cumsum([1.5;u]);
x = x(2:end);
```

Suppose further that the latent process is subject to additive measurement error. Subsequently, the observation equation is

$$y_t = x_t + \varepsilon_t,$$

where ε_t is Gaussian with mean 0 and standard deviation 1.

Use the random latent state process (`x`) and the observation equation to generate observations.

```
y = x + randn(T,1);
```

Together, the latent process and observation equations compose a state-space model. Assume that the state is a stationary AR(1) process. Then the state-space model to estimate is

$$\begin{aligned} x_t &= \phi x_{t-1} + \sigma_1 u_t \\ y_t &= x_t + \sigma_2 \varepsilon_t. \end{aligned}$$

Specify the coefficient matrices. Use `NaN` values for unknown parameters.

```
A = NaN;
B = NaN;
C = 1;
D = NaN;
```

Create the diffuse state-space model by passing the coefficient matrices to `dssm` and specifying that the state type is diffuse.

```
StateType = 2;
Mdl = dssm(A,B,C,D,'StateType',StateType);
```

`Mdl` is a `dssm` model object. The software sets values for the initial state mean and variance to 0 and `Inf`. Verify that the model is specified correctly using the display in the Command Window.

Pass the observations to `estimate` to estimate the parameters. For the `params0` parameters that are unlikely to correspond to their true values. Also, specify lower bound constraints of `0` for the standard deviations.

```
params0 = [-1e7 1e-6 2000];
EstMdl = estimate(Mdl,y,params0,'lb',[-Inf,0,0]);
```

```
Warning: Covariance matrix of estimators cannot be computed precisely due to
inversion difficulty. Check parameter identifiability. Also try different
starting values and other options to compute the covariance matrix.

Method: Maximum likelihood (fmincon)
Effective Sample size:             99
```

```
Logarithmic  likelihood:     -1874.82
Akaike   info criterion:      3755.64
Bayesian info criterion:      3763.46
      |       Coeff         Std Err       t Stat      Prob
-------------------------------------------------------------
 c(1) |    -9.99970e+06   1.00501e+06     -9.94989   0
 c(2) | 89578.46690       1.09581e+09      0.00008  0.99993
 c(3) |     3.65719       0.00005      75597.89470   0
      |
      |      Final State   Std Dev          t Stat    Prob
 x(1) |    -3.37649       3.67423         -0.91896  0.35812
```

`estimate` failed to converge, and so the results are undesirable.

Refine `params0` using `refine`.

```
Output = refine(Mdl,y,params0);
logL = cell2mat({Output.LogLikelihood})';
[~,maxLogLIndx] = max(logL)
refinedParams0 = Output(maxLogLIndx).Parameters
Description = Output(maxLogLIndx).Description
```

```
maxLogLIndx =

     2


refinedParams0 =

    0.9781    0.8965    0.9336


Description =

Nelder-Mead simplex
```

The algorithm that yields the highest loglikelihood value is `Loose bound interior point`, which is the third `struct` in the structure array `Output`.

Estimate `Mdl` using `refinedParams0`, which is the vector of refined initial parameter values.

```
EstMdl = estimate(Mdl,y,refinedParams0,'lb',[-Inf,0,0]);
```

```
Method: Maximum likelihood (fmincon)
Effective Sample size:              99
Logarithmic  likelihood:     -179.018
Akaike   info criterion:      364.036
Bayesian info criterion:      371.851
      |     Coeff       Std Err   t Stat     Prob
-------------------------------------------------
 c(1) |  0.97805       0.02947   33.18393   0
 c(2) |  0.89651       0.18465    4.85529  0.00000
 c(3) |  0.93355       0.15187    6.14707   0
      |
      |   Final State   Std Dev    t Stat    Prob
 x(1) | -3.95108       0.72269   -5.46719   0
```

`estimate` converged, making the parameter estimates much more desirable. The AR model coefficient is within two standard errors of 1, which suggests that the state processes is a random walk.

Refine Diffuse State-Space Model Estimation Including Regression Component

Suppose that the relationship between the unemployment rate and the nominal gross national product (nGNP) is linear. Suppose further that the unemployment rate is an AR(1) series. Symbolically, and in state-space form, the model is

$$\begin{aligned} x_t &= \phi x_{t-1} + \sigma u_t \\ y_t - \beta Z_t &= x_t, \end{aligned}$$

where:

- x_t is the unemployment rate at time t.
- y_t is the observed unemployment rate being deflated by the log of nGNP (Z_t).
- u_t is the Gaussian series of state disturbances having mean 0 and unknown standard deviation σ.

Load the Nelson-Plosser data set, which contains the unemployment rate and nGNP series data.

```
load Data_NelsonPlosser
```

Preprocess the data by taking the first difference of the unemployment rate and converting nGNP to a series of returns. Remove the observations corresponding to the string of `NaN` values at the beginning of the unemployment rate series.

```
isNaN = any(ismissing(DataTable),2);       % Flag periods containing NaNs
gnpn = DataTable.GNPN(~isNaN);
y = DataTable.UR(~isNaN);
y = diff(y);
T = size(y,1);
Z = [ones(T,1) price2ret(gnpn)];
```

This example continues using the series without `NaN` values. However, using the Kalman filter framework, the software can accommodate series containing missing values.

Specify the coefficient matrices.

```
A = NaN;
B = NaN;
C = 1;
```

Create the state-space model using `dssm` by supplying the coefficient matrices and specifying that the state values come from a diffuse distribution. The diffuse specification indicates complete ignorance about the moments of the initial distribution.

```
StateType = 2;
Mdl = dssm(A,B,C,'StateType',StateType);
```

`Mdl` is a `dssm` model object.

Find a good set of starting parameters to use for estimation.

```
params0 = [150 1000]; % Initial values chosen arbitrarily
Beta0 = [1 -100];
Output = refine(Mdl,y,params0,'Predictors',Z,'Beta0',Beta0);
```

`Output` is a 1-by-5 structure array containing the recommended initial parameter values.

Choose the initial parameter values corresponding to the largest loglikelihood.

```
logL = cell2mat({Output.LogLikelihood})';
[~,maxLogLIndx] = max(logL)
refinedParams0 = Output(maxLogLIndx).Parameters
Description = Output(maxLogLIndx).Description


maxLogLIndx =
```

```
     5


refinedParams0 =

    0.2070   -1.3229    1.3610  -24.8848


Description =

Starting value shrinkage
```

The algorithm that yields the highest loglikelihood value is `Nelder-Mead simplex`, which is the second struct in the structure array `Output`.

Estimate `Mdl` using the refined initial parameter values `refinedParams0`.

```
EstMdl = estimate(Mdl,y,refinedParams0(1:(end - 2)),'Predictors',Z,...
    'Beta0',refinedParams0((end - 1):end));
```

```
Method: Maximum likelihood (fminunc)
Effective Sample size:             60
Logarithmic  likelihood:     -101.924
Akaike   info criterion:      211.849
Bayesian info criterion:      220.292
           |      Coeff       Std Err    t Stat     Prob
-------------------------------------------------------------
 c(1)      |   0.20700       0.12330     1.67891  0.09317
 c(2)      |  -1.32287       0.08415   -15.71964   0
 y <- z(1) |   1.36101       0.23736     5.73388   0
 y <- z(2) | -24.88484       1.78021   -13.97861   0
           |
           |    Final State   Std Dev     t Stat    Prob
 x(1)      |   1.21611        0           Inf      0
```

Algorithms

The Kalman filter accommodates missing data by not updating filtered state estimates corresponding to missing observations. In other words, suppose that your data has a missing observation at period t. Then, the state forecast for period t, based on the previous $t - 1$ observations, is equivalent to the filtered state for period t.

See Also

dssm | estimate

More About

- “What Are State-Space Models?” on page 8-3

Introduced in R2015b

refine

Class: ssm

Refine initial parameters to aid state-space model estimation

Syntax

```
refine(Mdl,Y,params0)
refine(Mdl,Y,params0,Name,Value)
Output = refine( ___ )
```

Description

`refine(Mdl,Y,params0)` finds a set of initial parameter values to use when fitting the state-space model `Mdl` to the response data `Y`, using the crude set of initial parameter values `params0`. The software uses several routines, and displays the resulting loglikelihood and initial parameter values for each routine.

`refine(Mdl,Y,params0,Name,Value)` displays results of the routines with additional options specified by one or more `Name,Value` pair arguments. For example, you can include a linear regression component composed of predictors and an initial value for the coefficients.

`Output = refine( ___ )` returns a structure array (`Output`) containing a vector of refined, initial parameter values, the loglikelihood corresponding the initial parameter values, and the method the software used to obtain the values. You can use any of the input arguments in the previous syntaxes.

Tips

- Likelihood surfaces of state-space models can be complicated, for example, they might contain multiple local maxima. If estimate fails to converge, or converges to an unsatisfactory solution, then `refine` might find a better set of initial parameter values to pass to `estimate`.

- The refined initial parameter values returned by `refine` might appear similar to each other and to `params0`. Choose a set yielding estimates that make economic sense and correspond to relatively large loglikelihood values.
- If a refinement attempt fails, then the software displays errors and sets the corresponding loglikelihood to `-Inf`. It also sets its initial parameter values to `[]`.

Input Arguments

`Mdl` — Standard state-space model

`ssm` model object

Standard state-space model containing unknown parameters, specified as an `ssm` model object returned by ssm.

`Mdl` does not store observed responses or predictor data. Supply the data wherever necessary, using the appropriate input and name-value pair arguments.

`Y` — Observed response data

numeric matrix | cell vector of numeric vectors

Observed response data to which `Mdl` is fit, specified as a numeric matrix or a cell vector of numeric vectors.

- If `Mdl` is time invariant with respect to the observation equation, then `Y` is a T-by-n matrix. Each row of the matrix corresponds to a period and each column corresponds to a particular observation in the model. Therefore, T is the sample size and n is the number of observations per period. The last row of `Y` contains the latest observations.
- If `Mdl` is time varying with respect to the observation equation, then `Y` is a T-by-1 cell vector. `Y{t}` contains an n_t-dimensional vector of observations for period t, where $t = 1,...,T$. The corresponding dimensions of the coefficient matrices in `Mdl.C{t}` and `Mdl.D{t}` must be consistent with the matrix in `Y{t}` for all periods. The last cell of `Y` contains the latest observations.
- Suppose that you created `Mdl` implicitly by specifying a parameter-to-matrix mapping function, and the function has input arguments for the observed responses or predictors. The mapping function establishes a link to observed responses and the predictor data in the MATLAB workspace, which overrides the value of `Y`.

`NaN` elements indicate missing observations. For details on how the Kalman filter accommodates missing observations, see “Algorithms” on page 9-829.

Data Types: `double | cell`

`params0` — Initial values of unknown parameters

numeric vector

Initial values of unknown parameters for numeric maximum likelihood estimation, specified as a numeric vector.

The elements of `params0` correspond to the unknown parameters in the state-space model matrices `A`, `B`, `C`, and `D`, and, optionally, the initial state mean `Mean0` and covariance matrix `Cov0`.

- If you created `Mdl` explicitly (that is, by specifying the matrices without a parameter-to-matrix mapping function), then the software maps the elements of `params` to `NaN`s in the state-space model matrices and initial state values. The software searches for `NaN`s column-wise, following the order `A`, `B`, `C`, `D`, `Mean0`, `Cov0`.
- If you created `Mdl` implicitly (that is, by specifying the matrices with a parameter-to-matrix mapping function), then set initial parameter values for the state-space model matrices, initial state values, and state types within the parameter-to-matrix mapping function.

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'Beta0'` — Initial values of regression coefficients

numeric matrix

Initial values of regression coefficients, specified as the comma-separated pair consisting of `'Beta0'` and a d-by-n numeric matrix. d is the number of predictor variables (see `Predictors`) and n is the number of observed response series (see `Y`).

By default, `Beta0` is the ordinary least-squares estimate of `Y` onto `Predictors`.

Data Types: `double`

`'Predictors'` — Predictor data

[ ] (default) | numeric matrix

Predictor data used to deflate the observations in a time-invariant state-space model, specified as the comma-separated pair consisting of `'Predictors'` and a T-by-d numeric matrix. T is the number of periods and d is the number of predictor variables. Row t corresponds to the observed predictors at period t (Z_t) in the expanded observation equation

$$y_t - Z_t\beta = Cx_t + Du_t.$$

That is, the software deflates the observations using the regression component. β is the time-invariant vector of regression coefficients that the software estimates with all other parameters. `Predictors` and `Y` must have the same number of rows.

For n observations per period, the software regresses all predictor series onto each observation. Then, the software returns a d-by-n matrix of fitted regression coefficient vectors for each observation series.

If you specify `Predictors`, then `Mdl` must be time invariant. Otherwise, the software returns an error.

By default, the software excludes a regression component from the state-space model.

Data Types: `double`

Output Arguments

`Output` — Information about initial parameter values

structure array

Information about the initial parameter values, returned as a 1-by-5 structure array. The software uses five algorithms to find initial parameter values, and each element of `Output` corresponds to an algorithm.

This table describes the fields of `Output`.

Field	Description
`Description`	Refinement algorithm.

Field	Description

	Each element of `Output` corresponds to one of these algorithms:
	
	`'Loose bound interior point'`
	`'Nelder-Mead algorithm'`
	`'Quasi-Newton'`
	`'Starting value perturbation'`
	`'Starting value shrinkage'`

`Loglikelihood`	Loglikelihood corresponding to the initial parameter values.
`Parameters`	Vector of refined initial parameter values. The order of the parameters is the same as the order in `params0`. If you pass these initial values to `estimate`, then the estimation results can improve.

Examples

Refine Parameters When Fitting Time-Invariant State-Space Model

Suppose that a latent process is a random walk. Subsequently, the state equation is

$$x_t = x_{t-1} + u_t,$$

where u_t is Gaussian with mean 0 and standard deviation 1.

Generate a random series of 100 observations from x_t, assuming that the series starts at 1.5.

```
T = 100;
rng(1); % For reproducibility
u = randn(T,1);
x = cumsum([1.5;u]);
x = x(2:end);
```

Suppose further that the latent process is subject to additive measurement error. Subsequently, the observation equation is

$$y_t = x_t + \varepsilon_t,$$

where ε_t is Gaussian with mean 0 and standard deviation 1.

Use the random latent state process (`x`) and the observation equation to generate observations.

```
y = x + randn(T,1);
```

Together, the latent process and observation equations compose a state-space model. Assume that the state is a stationary AR(1) process. Then the state-space model to estimate is

$$\begin{aligned} x_t &= \phi x_{t-1} + \sigma_1 u_t \\ y_t &= x_t + \sigma_2 \varepsilon_t. \end{aligned}$$

Specify the coefficient matrices. Use `NaN` values for unknown parameters.

```
A = NaN;
B = NaN;
C = 1;
D = NaN;
```

Specify the state-space model using the coefficient matrices. Specify that the initial state distribution is stationary using the `StateType` name-value pair argument.

```
StateType = 0;
Mdl = ssm(A,B,C,D,'StateType',StateType);
```

`Mdl` is an `ssm` model. The software sets values for the initial state mean and variance. Verify that the model is specified correctly using the display in the Command Window.

Pass the observations to `estimate` to estimate the parameters. For the `params0` parameters that are unlikely to correspond to their true values. Also, specify lower bound constraints of `0` for the standard deviations.

```
params0 = [-1e7 1e-6 2000];
EstMdl = estimate(Mdl,y,params0,'lb',[-Inf,0,0]);
```

```
Warning: Covariance matrix of estimators cannot be computed precisely due to
inversion difficulty. Check parameter identifiability. Also try different
starting values and other options to compute the covariance matrix.

Method: Maximum likelihood (fmincon)
Sample size: 100
```

```
Logarithmic  likelihood:     -2464.23
Akaike   info criterion:      4934.46
Bayesian info criterion:      4942.27
      |      Coeff           Std Err       t Stat     Prob
-------------------------------------------------------------
 c(1) |   -9.99977e+06      9.99977e+05  -10.00000   0
 c(2) |    1.23086e+05      1.91567e+13    0.00000  1.00000
 c(3) | 2006.86501          3.12341e+11    0.00000  1.00000
      |
      |     Final State      Std Dev        t Stat    Prob
 x(1) |   -3.37649       1999.42392       -0.00169  0.99865
```

`estimate` failed to converge, and so the results are undesirable.

Refine `params0` using `refine`.

```
Output = refine(Mdl,y,params0);
logL = cell2mat({Output.LogLikelihood})';
[~,maxLogLIndx] = max(logL)
refinedParams0 = Output(maxLogLIndx).Parameters
Description = Output(maxLogLIndx).Description
```

```
maxLogLIndx =

     2


refinedParams0 =

    0.9705   -0.8934    0.9330


Description =

Nelder-Mead simplex
```

The algorithm that yields the highest loglikelihood value is `Loose bound interior point`, which is the third `struct` in the structure array `Output`.

Estimate `Mdl` using `refinedParams0`, which is the vector of refined initial parameter values.

```
EstMdl = estimate(Mdl,y,refinedParams0,'lb',[-Inf,0,0]);
```

```
Method: Maximum likelihood (fmincon)
Sample size: 100
Logarithmic  likelihood:     -181.379
Akaike   info criterion:      368.758
Bayesian info criterion:      376.574
      |     Coeff       Std Err   t Stat     Prob
-------------------------------------------------
 c(1) |  0.97050       0.02863   33.90367   0
 c(2) |  0.89343       0.18521    4.82401  0.00000
 c(3) |  0.93303       0.15176    6.14806   0
      |
      |   Final State   Std Dev    t Stat    Prob
 x(1) | -3.93007       0.72066   -5.45343   0
```

`estimate` converged, making the parameter estimates much more desirable. The AR model coefficient is within two standard errors of 1, which suggests that the state processes is a random walk.

Refine Estimation of State-Space Model Containing Regression Component

Suppose that the relationship between the change in the unemployment rate and the nominal gross national product (nGNP) growth rate is of interest. Suppose further that the first difference of the unemployment rate is an ARMA(1,1) series. Symbolically, and in state-space form, the model is

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix} = \begin{bmatrix} \phi & \theta \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix} u_{1,t}$$
$$y_t - \beta Z_t = x_{1,t} + \sigma \varepsilon_t,$$

where:

- $x_{1,t}$ is the change in the unemployment rate at time t.
- $x_{2,t}$ is a dummy state for the MA(1) effect.
- y_t is the observed change in the unemployment rate being deflated by the growth rate of nGNP (Z_t).
- $u_{1,t}$ is the Gaussian series of state disturbances having mean 0 and standard deviation 1.
- ε_t is the Gaussian series of observation innovations having mean 0 and standard deviation σ.

Load the Nelson-Plosser data set, which contains the unemployment rate and nGNP series data.

```
load Data_NelsonPlosser
```

Preprocess the data by taking the natural logarithm of the nGNP series, and the first difference of each. Also, remove the starting `NaN` values from each series.

```
isNaN = any(ismissing(DataTable),2);       % Flag periods containing NaNs
gnpn = DataTable.GNPN(~isNaN);
u = DataTable.UR(~isNaN);
T = size(gnpn,1);                          % The sample size
Z = [ones(T-1,1) diff(log(gnpn))];
y = diff(u);
```

This example continues using the series without `NaN` values. However, using the Kalman filter framework, the software can accommodate series containing missing values.

Specify the coefficient matrices.

```
A = [NaN NaN; 0 0];
B = [1; 1];
C = [1 0];
D = NaN;
```

Specify the state-space model using `ssm`.

```
Mdl = ssm(A,B,C,D);
```

Estimate the model parameters. Specify the regression component and its initial value for optimization using the `'Predictors'` and `'Beta0'` name-value pair arguments, respectively. Restrict the estimate of σ to all positive, real numbers.

```
params0 = [0 0 1e-11];
Beta0 = [0 0];
EstMdl = estimate(Mdl,y,params0,'Predictors',Z,...
    'Beta0',Beta0,'lb',[-Inf,-Inf,0,-Inf,-Inf]);
```

```
Warning: Covariance matrix of estimators cannot be computed precisely due to
inversion difficulty. Check parameter identifiability. Also try different
starting values and other options to compute the covariance matrix.

Method: Maximum likelihood (fmincon)
Sample size: 61
```

```
Logarithmic  likelihood:     -109.709
Akaike   info criterion:      229.417
Bayesian info criterion:      239.972
           |      Coeff        Std Err      t Stat     Prob
-------------------------------------------------------------
 c(1)      |  -0.25172       0.27386       -0.91917  0.35801
 c(2)      |   0.50899       0.23603        2.15650  0.03105
 c(3)      |   0.00000       1.86328e+07    0.00000   1
 y <- z(1) |   1.85671       0.11960       15.52383   0
 y <- z(2) | -27.50973       1.01219      -27.17849   0
           |
           |    Final State   Std Dev        t Stat    Prob
 x(1)      |   0.84457        0              Inf      0
 x(2)      |   0.88637        0              Inf      0
```

The software could not estimate the parameter covariance matrix.

Refine the initial parameter values.

```
Output = refine(Mdl,y,params0,'Predictors',Z,'Beta0',Beta0);
```

`Output` is a 1-by-5 structure array containing the recommended initial parameter values.

Choose the initial parameter values corresponding to the largest loglikelihood.

```
logL = cell2mat({Output.LogLikelihood})';
[~,maxLogLIndx] = max(logL)
refinedParams0 = Output(maxLogLIndx).Parameters
Description = Output(maxLogLIndx).Description
```

```
maxLogLIndx =

     2


refinedParams0 =

   -0.3410    1.0500   -0.4859    1.3612  -24.4671


Description =

Nelder-Mead simplex
```

The algorithm that yields the highest loglikelihood value is `Quasi-Newton`, which is the first struct in the structure array `Output`.

Estimate `Mdl` using the refined initial parameter values `refinedParams0`.

```
pBeta = numel(Beta0);
EstMdl = estimate(Mdl,y,refinedParams0(1:(end - pBeta)),'Predictors',Z,...
    'Beta0',refinedParams0((end - pBeta + 1):end),...
    'lb',[-Inf,-Inf,0,-Inf,-Inf]);
```

```
Method: Maximum likelihood (fmincon)
Sample size: 61
Logarithmic  likelihood:     -99.7245
Akaike   info criterion:      209.449
Bayesian info criterion:      220.003
           |      Coeff       Std Err    t Stat     Prob
-------------------------------------------------------------
 c(1)      |  -0.34098       0.29608    -1.15164  0.24948
 c(2)      |   1.05003       0.41377     2.53771  0.01116
 c(3)      |   0.48592       0.36790     1.32079  0.18657
 y <- z(1) |   1.36121       0.22338     6.09358   0
 y <- z(2) | -24.46711       1.60018   -15.29024   0
           |
           |    Final State   Std Dev     t Stat    Prob
 x(1)      |   1.01264       0.44690     2.26592  0.02346
 x(2)      |   0.77718       0.58917     1.31912  0.18713
```

`estimate` returns reasonable parameter estimates and their corresponding standard errors.

Algorithms

The Kalman filter accommodates missing data by not updating filtered state estimates corresponding to missing observations. In other words, suppose that your data has a missing observation at period t. Then, the state forecast for period t, based on the previous $t - 1$ observations, is equivalent to the filtered state for period t.

See Also

estimate | filter | forecast | simulate | smooth | ssm

More About

- "What Are State-Space Models?" on page 8-3

regARIMA class

Create regression model with ARIMA time series errors

Description

`regARIMA` creates a regression model with ARIMA time series errors to maintain the sensitivity interpretation of regression coefficients.

By default, the time series errors (also called unconditional disturbances) are independent, identically distributed, mean 0 Gaussian random variables. If the errors have an autocorrelation structure, then you can specify models for them. The models include:

- moving average (MA)
- autoregressive (AR)
- mixed autoregressive and moving average (ARMA)
- integrated (ARIMA)
- multiplicative seasonal (SARIMA)

Specify error models containing known coefficients to:

- Simulate responses using simulate.
- Explore impulse responses using impulse.
- Forecast future observations using forecast.
- Estimate unknown coefficients with data using estimate.

Construction

`Mdl = regARIMA` creates a regression model with degree 0 ARIMA errors and no regression coefficient.

`Mdl = regARIMA(p,D,q)` creates a regression model with errors modeled by a nonseasonal, linear time series with autoregressive degree `p`, differencing degree `D`, and moving average degree `q`.

`Mdl = regARIMA(Name,Value)` creates a regression model with ARIMA errors using additional options specified by one or more `Name,Value` pair arguments. `Name` can also be a property name and `Value` is the corresponding value. `Name` must appear inside single quotes (`''`). You can specify several `Name,Value` pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Input Arguments

Note: For regression models with nonseasonal ARIMA errors, use `p`, `D`, and `q`. For regression models with seasonal ARIMA errors, use `Name,Value` pair arguments.

`p`

Nonseasonal, autoregressive polynomial degree for the error model, specified as a positive integer.

`D`

Nonseasonal integration degree for the error model, specified as a nonnegative integer.

`q`

Nonseasonal, moving average polynomial degree for the error model, specified as a positive integer.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'Intercept'`

Regression model intercept, specified as the comma-separated pair consisting of `'Intercept'` and a scalar.

Default: `NaN`

'Beta'

Regression model coefficients associated with the predictor data, specified as the comma-separated pair consisting of 'Beta' and a vector.

Default: [] (no regression coefficients corresponding to predictor data)

'AR'

Nonseasonal, autoregressive coefficients for the error model, specified as the comma-separated pair consisting of 'AR' and a cell vector. The coefficients must yield a stable polynomial.

- If you specify ARLags, then AR is an equivalent-length cell vector of coefficients associated with the lags in ARLags. For example, if ARLags = [1, 4] and AR = {0.2, 0.1}, then, ignoring all other specifications, the error model is $u_t = 0.2u_{t-1} + 0.1u_{t-4} + \varepsilon_t$.
- If you do not specify ARLags, then AR is a cell vector of coefficients at lags 1,2,...,p, which is the nonseasonal, autoregressive polynomial degree. For example, if AR = {0.2, 0.1} and you do not specify ARLags, then, ignoring all other specifications, the error model is $u_t = 0.2u_{t-1} + 0.1u_{t-2} + \varepsilon_t$.

Default: Cell vector of NaNs with the same length as ARLags.

'MA'

Nonseasonal, moving average coefficients for the error model, specified as the comma-separated pair consisting of 'MA' and a cell vector. The coefficients must yield an invertible polynomial.

- If you specify MALags, then MA is an equivalent-length cell vector of coefficients associated with the lags in MALags. For example, if MALags = [1, 4] and MA = {0.2, 0.1}, then, ignoring all other specifications, the error model is $u_t = \varepsilon_t + 0.2\varepsilon_{t-1} + 0.1\varepsilon_{t-4}$.
- If you do not specify MALags, then MA is a cell vector of coefficients at lags 1,2,...,q, which is the nonseasonal, moving average polynomial degree. For example, if MA = {0.2, 0.1} and you do not specify MALags, then, ignoring all other specifications, the error model is $u_t = \varepsilon_t + 0.2\varepsilon_{t-1} + 0.1\varepsilon_{t-2}$.

Default: Cell vector of NaNs with the same length as MALags.

'ARLags'

Lags associated with the AR coefficients in the error model, specified as the comma-separated pair consisting of 'ARLags' and a vector of positive integers.

Default: Vector of integers 1,2,...,*p*, the nonseasonal, autoregressive polynomial degree.

'MALags'

Lags associated with the MA coefficients in the error model, specified as the comma-separated pair consisting of 'MALags' and a vector of positive integers.

Default: Vector of integers 1,2,...,*q*, the nonseasonal moving average polynomial degree.

'SAR'

Seasonal, autoregressive coefficients for the error model, specified as the comma-separated pair consisting of 'SAR' and a cell vector. The coefficient must yield a stable polynomial.

- If you specify SARLags, then SAR is an equivalent-length cell vector of coefficients associated with the lags in SARLags. For example, if SARLags = [1, 4], SAR = {0.2, 0.1}, and Seasonality = 4, then, ignoring all other specifications, the error model is

  $$\left(1-0.2L-0.1L^4\right)\left(1-L^4\right)u_t=\varepsilon_t.$$

- If you do not specify SARLags, then SAR is a cell vector of coefficients at lags 1,2,...,p_s, which is the seasonal, autoregressive polynomial degree. For example, if SAR = {0.2, 0.1} and Seasonality = 4, and you do not specify SARLags, then, ignoring all other specifications, the error model is

  $$\left(1-0.2L-0.1L^2\right)\left(1-L^4\right)u_t=\varepsilon_t.$$

Default: Cell vector of NaNs with the same length as SARLags.

'SMA'

Seasonal, moving average coefficients for the error model, specified as the comma-separated pair consisting of 'SMA' and a cell vector. The coefficient must yield an invertible polynomial.

- If you specify SMALags, then SMA is an equivalent-length cell vector of coefficients associated with the lags in SMALags. For example, if SMALags = [1, 4], SMA = {0.2, 0.1}, and Seasonality = 4, then, ignoring all other specifications, the error model is $\left(1-L^4\right)u_t=\left(1+0.2L+0.1L^4\right)\varepsilon_t$.
- If you do not specify SMALags, then SMA is a cell vector of coefficients at lags 1,2,...,q_s, the seasonal, moving average polynomial degree. For example, if SMA = {0.2, 0.1} and Seasonality = 4, and you do not specify SMALags, then, ignoring all other specifications, the error model is $\left(1-L^4\right)u_t=\left(1+0.2L+0.1L^2\right)\varepsilon_t$.

Default: Cell vector of NaNs with the same length as SMALags.

'SARLags'

Lags associated with the SAR coefficients in the error model, specified as the comma-separated pair consisting of 'SARLags' and a vector of positive integers.

Default: Vector of integers 1,2,...,p_s, the seasonal, autoregressive polynomial degree.

'SMALags'

Lags associated with the SMA coefficients in the error model, specified as the comma-separated pair consisting of 'SMALags' and a vector of positive integers.

Default: Vector of integers 1,2,...,q_s, the seasonal moving average polynomial degree.

'D'

Nonseasonal differencing polynomial degree (i.e., nonseasonal integration degree) for the error model, specified as the comma-separated pair consisting of 'D' and a nonnegative integer.

Default: 0 (no nonseasonal integration)

'Seasonality'

Seasonal differencing polynomial degree for the error model, specified as the comma-separated pair consisting of 'Seasonality' and a nonnegative integer.

Default: 0 (no seasonal integration)

`'Variance'`

Variance of the model innovations ε_t, specified as the comma-separated pair consisting of `'Variance'` and a positive scalar.

Default: `NaN`

`'Distribution'`

Conditional probability distribution of the innovation process, specified as the comma-separated pair consisting of `'Distribution'` and a string or a structure array.

Distribution	String	Structure array
Gaussian	`'Gaussian'`	`struct('Name','Gaussian')`
Student's t	`'t'` By default, `DoF` is `NaN`.	`struct('Name','t','DoF',DoF)` `DoF` > 2 or `DoF = NaN`

Default: `'Gaussian'`

Notes

- Each AR, SAR, MA, and SMA coefficient is associated with an underlying lag operator polynomial and is subject to a near-zero tolerance exclusion test. That is, the software compares each coefficient to the default lag operator zero tolerance, `1e-12`. If the magnitude of a coefficient is greater than `1e-12`, then the software includes it in the model. Otherwise, the software considers the coefficient sufficiently close to 0, and excludes it from the model. For additional details, see LagOp.
- Specify the lags associated with the seasonal polynomials SAR and SMA in the periodicity of the observed data, and not as multiples of the `Seasonality` parameter. This convention does not conform to standard Box and Jenkins [1] notation, but it is a more flexible approach for incorporating multiplicative seasonality.

Properties

AR

Cell vector of nonseasonal, autoregressive coefficients corresponding to a stable polynomial of the error model. Associated lags are 1,2,...,p, which is the nonseasonal, autoregressive polynomial degree, or as specified in `ARLags`.

Beta

Real vector of regression coefficients corresponding to the columns of the predictor data matrix.

D

Nonnegative integer indicating the nonseasonal integration degree of the error model.

Distribution

Data structure for the conditional probability distribution of the innovation process. The field `Name` stores the distribution name `'Gaussian'` or `'t'`. If the distribution is `'t'`, then the structure also has the field `DoF` that stores the degrees of freedom.

Intercept

Scalar intercept in the error model.

MA

Cell vector of nonseasonal moving average coefficients corresponding to an invertible polynomial of the error model. Associated lags are 1,2,...,q to the degree of the nonseasonal moving average polynomial, or as specified in `MALags`.

P

Scalar, compound autoregressive polynomial degree of the error model.

`P` is the total number of lagged observations necessary to initialize the autoregressive component of the error model. `P` includes the effects of nonseasonal and seasonal integration captured by the properties `D` and `Seasonality`, respectively, and the nonseasonal and seasonal autoregressive polynomials `AR` and `SAR`, respectively.

`P` does not necessarily conform to standard Box and Jenkins notation [1]. If `D = 0`, `Seasonality = 0`, and `SAR = {}`, then `P` conforms to the standard notation.

Q

Scalar, compound moving average polynomial degree of the error model.

`Q` is the total number of lagged innovations necessary to initialize the moving average component of the model. `Q` includes the effects of nonseasonal and seasonal moving average polynomials `MA` and `SMA`, respectively.

`Q` does not necessarily conform to standard Box and Jenkins notation [1]. If `SMA = {}`, then `Q` conforms to the standard notation.

`SAR`

Cell vector of seasonal autoregressive coefficients corresponding to a stable polynomial of the error model. Associated lags are 1,2,...,p_s, which is the seasonal autoregressive polynomial degree, or as specified in `SARLags`.

`SMA`

Cell vector of seasonal moving average coefficients corresponding to an invertible polynomial of the error model. Associated lags are 1,2,...,q_s, which is the seasonal moving average polynomial degree, or as specified in `SMALags`.

`Seasonality`

Nonnegative integer indicating the seasonal differencing polynomial degree for the error model.

`Variance`

Positive scalar variance of the model innovations.

Methods

arima	Convert regression model with ARIMA errors to ARIMAX model
estimate	Estimate parameters of regression models with ARIMA errors
filter	Filter disturbances through regression model with ARIMA errors
forecast	Forecast responses of regression model with ARIMA errors
impulse	Impulse response of regression model with ARIMA errors
infer	Infer innovations of regression models with ARIMA errors

print	Display estimation results for regression models with ARIMA errors
simulate	Monte Carlo simulation of regression model with ARIMA errors

Definitions

Regression Model with ARIMA Time Series Errors

A model that explains the behavior of a response using a linear regression model with predictor data, though the errors have autocorrelation indicative of an ARIMA process.

The model has the following form (in lag operator notation):

$$y_t = c + X_t\beta + u_t$$
$$a(L)A(L)(1-L)^D\left(1-L^s\right)u_t = b(L)B(L)\varepsilon_t,$$

where

- $t = 1,...,T$.
- y_t is the response series.
- X_t is row t of X, which is the matrix of concatenated predictor data vectors. That is, X_t is observation t of each predictor series.
- c is the regression model intercept.
- β is the regression coefficient.
- u_t is the disturbance series.
- ε_t is the innovations series.
- $L^j y_t = y_{t-j}$.
- $a(L) = \left(1 - a_1L - ... - a_pL^p\right)$, which is the degree p, nonseasonal autoregressive polynomial.
- $A(L) = \left(1 - A_1L - ... - A_{p_s}L^{p_s}\right)$, which is the degree p_s, seasonal autoregressive polynomial.

- $(1-L)^D$, which is the degree D, nonseasonal integration polynomial.
- $\left(1-L^s\right)$, which is the degree s, seasonal integration polynomial.
- $b(L)=\left(1+b_1L+\ldots+b_qL^q\right)$, which is the degree q, nonseasonal moving average polynomial.
- $B(L)=\left(1+B_1L+\ldots+B_{q_s}L^{q_s}\right)$, which is the degree q_s, seasonal moving average polynomial.

Regression models with ARIMA errors contain a hierarchy of error series. The unconditional disturbance, u_t, or structural disturbance, is based on the structural regression component. The conditional error (one-step-ahead forecast or prediction error), ε_t is the innovation of u_t.

Note: The degrees of the lag operators in the seasonal polynomials $A(L)$ and $B(L)$ do not conform to those defined by Box and Jenkins [1]. In other words, Econometrics Toolbox does not treat $p_1 = s$, $p_2 = 2s,\ldots,p_s = c_ps$ nor $q_1 = s$, $q_2 = 2s,\ldots,q_s = c_qs$ where c_p and c_q are positive integers. The software is flexible as it lets you specify the lag operator degrees. See "Multiplicative ARIMA Model Specifications" on page 5-48.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Specify a Regression Model with Nonseasonal ARIMA Errors

Specify the following regression model with ARIMA(2,1,3) errors:

$$\begin{aligned} y_t &= u_t \\ (1-\phi_1L-\phi_2L^2)(1-L)u_t &= (1+\theta_1L+\theta_2L^2+\theta_3L^3)\varepsilon_t. \end{aligned}$$

```
Mdl = regARIMA(2,1,3)


Mdl =

    ARIMA(2,1,3) Error Model:
    -------------------------
    Distribution: Name = 'Gaussian'
       Intercept: NaN
               P: 3
               D: 1
               Q: 3
              AR: {NaN NaN} at Lags [1 2]
             SAR: {}
              MA: {NaN NaN NaN} at Lags [1 2 3]
             SMA: {}
        Variance: NaN
```

The output displays the values of the properties `P`, `D`, and `Q` of `Mdl`. The corresponding autoregressive and moving average coefficients (contained in `AR` and `MA`) are cell arrays containing the correct number of `NaN` values. Note that `P` = `p` + `D` = 3, indicating that you need three presample observations to initialize the model for estimation.

Modify a Regression Model with ARIMA Errors

Define the regression model with ARIMA errors:

$$\begin{aligned} y_t &= 2 + X_t \begin{bmatrix} 1.5 \\ 0.2 \end{bmatrix} + u_t \\ (1 - 0.2L - 0.3L^2)u_t &= (1 + 0.1L)\varepsilon_t, \end{aligned}$$

where ε_t is Gaussian with variance 0.5.

```
Mdl = regARIMA('Intercept',2,'AR',{0.2 0.3},'MA',{0.1},...
    'Variance',0.5,'Beta',[1.5 0.2])


Mdl =

    Regression with ARIMA(2,0,1) Error Model:
    -----------------------------------------
    Distribution: Name = 'Gaussian'
       Intercept: 2
```

```
            Beta: [1.5 0.2]
               P: 2
               D: 0
               Q: 1
              AR: {0.2 0.3} at Lags [1 2]
             SAR: {}
              MA: {0.1} at Lags [1]
             SMA: {}
        Variance: 0.5
```

`Mdl` is fully specified to, for example, simulate a series of responses given the predictor data matrix, X_t.

Modify the model to estimate the regression coefficient, the AR terms, and the variance of the innovations.

```
Mdl.Beta = [NaN NaN];
Mdl.AR   = {NaN NaN};
Mdl.Variance = NaN;
```

Change the innovations distribution to a t distribution with 15 degrees of freedom.

```
Mdl.Distribution = struct('Name','t','DoF',15)
```

```
Mdl =

    Regression with ARIMA(2,0,1) Error Model:
    ----------------------------------------
    Distribution: Name = 't', DoF = 15
       Intercept: 2
            Beta: [NaN NaN]
               P: 2
               D: 0
               Q: 1
              AR: {NaN NaN} at Lags [1 2]
             SAR: {}
              MA: {0.1} at Lags [1]
             SMA: {}
        Variance: NaN
```

Specify a Regression Model with SARIMA Errors

Specify the following model:

$$y_t = 1 + 6X_t + u_t$$
$$(1 - 0.2L)(1 - L)(1 - 0.5L^4 - 0.2L^8)(1 - L^4)u_t = (1 + 0.1L)(1 + 0.05L^4 + 0.01L^8)\varepsilon_t,$$

where ε_t is Gaussian with variance 1.

```
Mdl = regARIMA('Intercept',1,'Beta',6,'AR',0.2,...
    'MA',0.1,'SAR',{0.5,0.2},'SARLags',[4, 8],...
    'SMA',{0.05,0.01},'SMALags',[4 8],'D',1,...
    'Seasonality',4,'Variance',1)


Mdl =

    Regression with ARIMA(1,1,1) Error Model Seasonally Integrated with Seasonal AR(8)
    ----------------------------------------------------------------------------------
    Distribution: Name = 'Gaussian'
       Intercept: 1
            Beta: [6]
               P: 14
               D: 1
               Q: 9
              AR: {0.2} at Lags [1]
             SAR: {0.5 0.2} at Lags [4 8]
              MA: {0.1} at Lags [1]
             SMA: {0.05 0.01} at Lags [4 8]
     Seasonality: 4
        Variance: 1
```

If you do not specify `SARLags` or `SMALags`, then the coefficients in `SAR` and `SMA` correspond to lags 1 and 2 by default.

```
Mdl = regARIMA('Intercept',1,'Beta',6,'AR',0.2,...
    'MA',0.1,'SAR',{0.5,0.2},'SMA',{0.05,0.01},...
    'D',1,'Seasonality',4,'Variance',1)


Mdl =

    Regression with ARIMA(1,1,1) Error Model Seasonally Integrated with Seasonal AR(2)
    ----------------------------------------------------------------------------------
    Distribution: Name = 'Gaussian'
       Intercept: 1
            Beta: [6]
               P: 8
```

```
              D: 1
              Q: 3
             AR: {0.2} at Lags [1]
            SAR: {0.5 0.2} at Lags [1 2]
             MA: {0.1} at Lags [1]
            SMA: {0.05 0.01} at Lags [1 2]
    Seasonality: 4
       Variance: 1
```

- "Specify Regression Models with ARIMA Errors Using regARIMA" on page 4-10
- "Specify the Default Regression Model with ARIMA Errors" on page 4-20
- "Modify regARIMA Model Properties" on page 4-22
- "Multiplicative ARIMA Model Specifications" on page 5-48

References

[1] Box, G. E. P., G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis: Forecasting and Control*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

See Also

arima | estimate | filter | forecast | impulse | infer | print | simulate

More About

- "Time Series Regression Models" on page 4-3
- "Nonspherical Models" on page 3-94
- "ARIMA Model Including Exogenous Covariates" on page 5-58

reflect

Class: LagOp

Reflect lag operator polynomial coefficients around lag zero

Syntax

```
B = reflect(A)
```

Description

Given a lag operator polynomial object *A(L)*,`B = reflect(A)` negates all coefficient matrices except the coefficient matrix at lag 0. For example, given a polynomial of degree *p*,

$$A(L) = A_0 + A_1L + A_2L^2 + \ldots + A_PL^p$$

the reflected polynomial B(L) is

$$B(L) = A_0 - A_1L - A_2L^2 - \ldots - A_PL^p$$

with the same degree and dimension as *A(L)*.

Examples

Reflect a Lag Operator Polynomial

Create a `LagOp` polynomial and its reflection:

```
A = LagOp({0.8 1 0 .6});
B = reflect(A)
```

```
B =
```

```
1-D Lag Operator Polynomial:
----------------------------
    Coefficients: [0.8 -1 -0.6]
            Lags: [0 1 3]
          Degree: 3
       Dimension: 1
```

ret2price

Convert returns to prices

Syntax

```
[TickSeries,TickTimes] = ...
ret2price(RetSeries,StartPrice,RetIntervals,StartTime,Method)
```

Description

`[TickSeries,TickTimes] = ...`
`ret2price(RetSeries,StartPrice,RetIntervals,StartTime,Method)`
generates price series for the specified assets, given the asset starting prices and the return observations for each asset.

Input Arguments

`RetSeries`	Time series array of returns. `RetSeries` can be a column vector or a matrix: • As a vector, `RetSeries` represents a univariate series of returns of a single asset. The length of the vector is the number of observations (`NUMOBS`). The first element contains the oldest observation, and the last element the most recent. • As a matrix, `RetSeries` represents a `NUMOBS`-by-number of assets (`NUMASSETS`) matrix of asset returns. Rows correspond to time indices. The first row contains the oldest observations and the last row the most recent. `ret2price` assumes that the observations across a given row occur at the same time for all columns, and each column is a return series of an individual asset.
`StartPrice`	A `NUMASSETS` element vector of initial prices for each asset, or a single scalar initial price applied to all assets. If `StartPrice = []` or is unspecified, all asset prices start at `1`.

`RetIntervals`	A `NUMOBS` element vector of time intervals between return observations, or a single scalar interval applied to all observations. If `RetIntervals` is `[]` or is unspecified, `ret2price` assumes that all intervals have length `1`.
`StartTime`	(optional) Scalar starting time for the first observation, applied to the price series of all assets. The default is `0`.
`Method`	Character string indicating the compounding method used to compute asset returns. If `Method` is `'Continuous'`, `[]`, or unspecified, then `ret2price` computes continuously compounded returns. If `Method` is `'Periodic'` then `ret2price` computes simple periodic returns. `Method` is case insensitive.

Output Arguments

`TickSeries`	Array of asset prices: • When `RetSeries` is a `NUMOBS` element column vector, `TickSeries` is a `NUMOBS+1` column vector. The first element contains the starting price of the asset, and the last element the most recent price. • When `RetSeries` is a `NUMOBS`-by-`NUMASSETS` matrix, then `RetSeries` is a (`NUMOBS+1`)-by-`NUMASSETS` matrix. The first row contains the starting price of the assets, and the last row contains the most recent prices.
`TickTimes`	A `NUMOBS`+1 element vector of price observation times. The initial time is zero unless specified in `StartTime`.

Examples

Convert Between Stock Prices and Returns

Create a stock price process continuously compounded at 10 percent

```
S = 100*exp(0.10*[0:19]');
```

```
    % Create the stock price series
```

Compute 10 percent returns for reference

```
R = price2ret(S); % Convert the price series to a
                  % 10 percent return series
```

Convert the resulting return series to the original price series, and compare results:

```
P = ret2price(R, 100); % Convert to the original price
                       % series
[S P]                  % Compare the original and
                       % computed price series
```

```
ans =

  100.0000  100.0000
  110.5171  110.5171
  122.1403  122.1403
  134.9859  134.9859
  149.1825  149.1825
  164.8721  164.8721
  182.2119  182.2119
  201.3753  201.3753
  222.5541  222.5541
  245.9603  245.9603
  271.8282  271.8282
  300.4166  300.4166
  332.0117  332.0117
  366.9297  366.9297
  405.5200  405.5200
  448.1689  448.1689
  495.3032  495.3032
  547.3947  547.3947
  604.9647  604.9647
  668.5894  668.5894
```

Compare the Relative Price Performance of Stocks

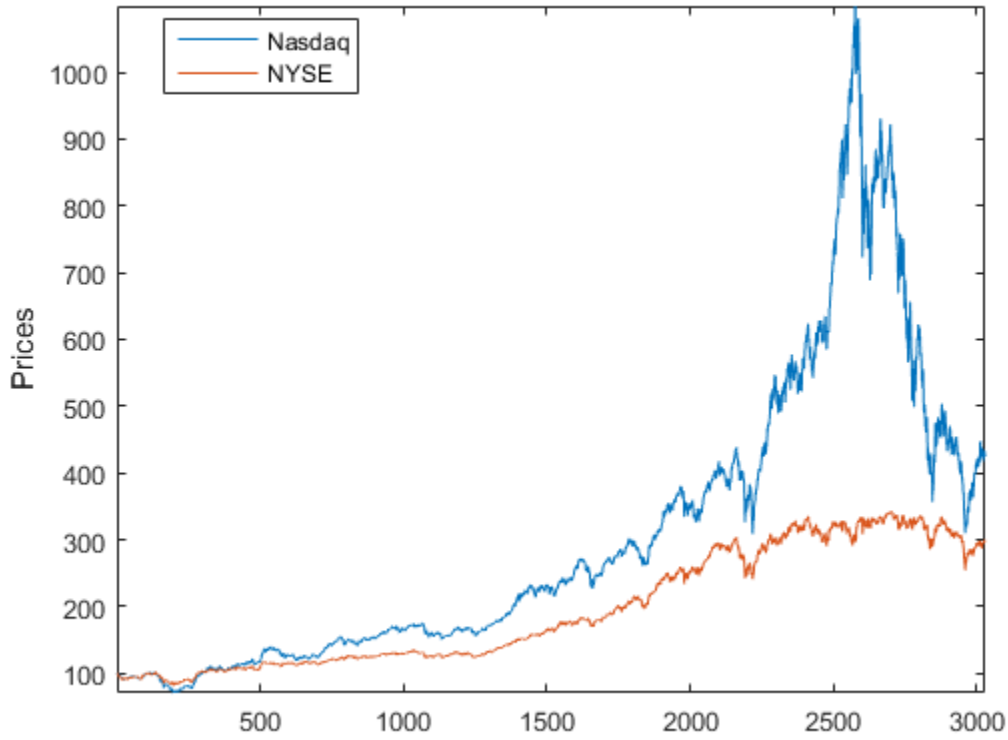
This example compares the relative price performance of the NASDAQ and the NYSE indexes.

Load the equity index data.

```
load Data_EquityIdx
```

Convert the returns back to prices, specifying the same starting price, 100, for each series, and plot the results.

```
figure;
plot(ret2price(price2ret([DataTable.NASDAQ DataTable.NYSE]), 100))
ylabel('Prices')
legend('Nasdaq', 'NYSE','Location','Best')
axis tight
```



The blue (upper) plot shows the NASDAQ price series. The green (lower) plot shows the NYSE price series.

See Also

price2ret | ret2tick

Introduced before R2006a

simsmooth

Class: ssm

State-space model simulation smoother

Syntax

```
X = simsmooth(Mdl,Y)
X = simsmooth(Mdl,Y,Name,Value)
```

Description

`X = simsmooth(Mdl,Y)` returns simulated states (`X`) by applying a simulation smoother to the time-invariant or time-varying state-space model (`Mdl`) and responses (`Y`). That is, the software uses forward filtering and back sampling to obtain one random path from the posterior distribution of the states.

`X = simsmooth(Mdl,Y,Name,Value)` returns simulated states with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

`Mdl` — Standard state-space model

`ssm` model object

Standard state-space model, specified as an`ssm` model object returned by ssm or estimate. A standard state-space model has finite initial state covariance matrix elements. That is, `Mdl` cannot be a dssm model object.

If `Mdl` is not fully specified (that is, `Mdl` contains unknown parameters), then specify values for the unknown parameters using the `'Params' Name,Value` pair argument. Otherwise, the software throws an error.

`Y` — Observed response data

numeric matrix | cell vector of numeric vectors

Observed response data to which `Mdl` is fit, specified as a numeric matrix or a cell vector of numeric vectors.

- If `Mdl` is time invariant with respect to the observation equation, then `Y` is a T-by-n matrix, where each row corresponds to a period and each column corresponds to a particular observation in the model. T is the sample size and m is the number of observations per period. The last row of `Y` contains the latest observations.
- If `Mdl` is time varying with respect to the observation equation, then `Y` is a T-by-1 cell vector. Each element of the cell vector corresponds to a period and contains an n_t-dimensional vector of observations for that period. The corresponding dimensions of the coefficient matrices in `Mdl.C{t}` and `Mdl.D{t}` must be consistent with the matrix in `Y{t}` for all periods. The last cell of `Y` contains the latest observations.

`NaN` elements indicate missing observations. For details on how the Kalman filter accommodates missing observations, see "Algorithms" on page 9-450.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'NumOut'` — Number of output arguments of parameter-to-matrix mapping function
positive integer

Number of output arguments of the parameter-to-matrix mapping function for implicitly defined state-space models, specified as the comma-separated pair consisting of `'NumOut'` and a positive integer.

If you implicitly define a state-space model and you do not supply `NumOut`, then the software automatically detects the number of output arguments of the parameter-to-matrix mapping function. Such detection consumes extra resources, and might slow the simulation smoother.

For explicitly defined models, the software ignores `NumOut` and displays a warning message.

`'NumPaths'` — Number of sample paths to generate variants
1 (default) | positive integer

Number of sample paths to generate variants, specified as the comma-separated pair consisting of `'NumPaths'` and a positive integer.

Example: `'NumPaths',1000`

Data Types: `double`

`'Params'` — Values for unknown parameters

numeric vector

Values for unknown parameters in the state-space model, specified as the column-separated pair consisting of `'Params'` and a numeric vector.

The elements of `Params` correspond to the unknown parameters in the state-space model matrices `A`, `B`, `C`, and `D`, and, optionally, the initial state mean `Mean0` and covariance matrix `Cov0`.

- If you created `Mdl` explicitly (that is, by specifying the matrices without a parameter-to-matrix mapping function), then the software maps the elements of `Params` to `NaN`s in the state-space model matrices and initial state values. The software searches for `NaN`s column-wise following the order `A`, `B`, `C`, `D`, `Mean0`, and `Cov0`.
- If you created `Mdl` implicitly (that is, by specifying the matrices with a parameter-to-matrix mapping function), then you must set initial parameter values for the state-space model matrices, initial state values, and state types within the parameter-to-matrix mapping function.

If `Mdl` contains unknown parameters, then you must specify their values. Otherwise, the software ignores the value of `Params`.

Data Types: `double`

`'Tolerance'` — Forecast uncertainty threshold

`0` (default) | nonnegative scalar

Forecast uncertainty threshold, specified as the comma-separated pair consisting of `'Tolerance'` and a nonnegative scalar.

If the forecast uncertainty for a particular observation is less than `Tolerance` during numerical estimation, then the software removes the uncertainty corresponding to the observation from the forecast covariance matrix before its inversion.

It is best practice to set `Tolerance` to a small number, for example, `1e-15`, to overcome numerical obstacles during estimation.

Example: `'Tolerance',1e-15`

Data Types: `double`

Output Arguments

X — Simulated states

numeric matrix | cell matrix of numeric vectors

Simulated states, returned as a numeric matrix or cell matrix of vectors.

If `Mdl` is a time-invariant model with respect to the states, then `X` is a `numObs`-by-m-by-`numPaths` array. That is, each row corresponds to a period, each column corresponds to a state in the model, and each page corresponds to a sample path. The last row corresponds to the latest simulated states.

If `Mdl` is a time-varying model with respect to the states, then `X` is a `numObs`-by-`numPaths` cell matrix of vectors. `X{t,j}` contains a vector of length m_t of simulated states for period t of sample path j. The last row of `X` contains the latest set of simulated states.

Definitions

Simulation Smoother

The *simulation smoother* is an algorithm for drawing samples from the conditional, joint, posterior distribution of the states given the complete observed response series. You can use these random draws to conduct a simulation study of the estimators.

For a univariate, time-invariant state-space model, the simulation smoother algorithm follows these steps.

1. Obtains the smoothed states ($\hat{x}_t;\ t = 1,..,T$) using the Kalman filter.
2. Chooses initial state mean and variance values. Draw the initial random state x_0^* from the Gaussian distribution with the initial state mean and variance.

3 Randomly generates T state disturbances and observation innovations from the standard normal distribution. Denote the random variants for period t ε_t^* and u_t^*, respectively.

4 Creates random observations and states by plugging ε_t^* and u_t^* into the state-space model

$$\begin{aligned} x_t^* &= Ax_{t-1}^* + B\varepsilon_t^* \\ y_t^* &= Cx_t^* + Du_t^* \end{aligned}.$$

5 Obtains smoothed states ($\hat{x}_t^*$) by applying the Kalman filter to the state-space model using the observation series y_t^*.

6 Obtains the random path of smoothed states from the posterior distribution using

$$\tilde{x}_t = \hat{x}_t - \hat{x}_t^* + x_t^*.$$

For more details, see [1].

Examples

Simulate States of Time-Invariant State-Space Models Using Simulation Smoother

Suppose that a latent process is an AR(1) model. Subsequently, the state equation is

$$x_t = 0.5x_{t-1} + u_t,$$

where u_t is Gaussian with mean 0 and standard deviation 1.

Generate a random series of 100 observations from x_t, assuming that the series starts at 1.5.

```
T = 100;
ARMdl = arima('AR',0.5,'Constant',0,'Variance',1);
x0 = 1.5;
rng(1); % For reproducibility
```

```
x = simulate(ARMdl,T,'Y0',x0);
```

Suppose further that the latent process is subject to additive measurement error. Subsequently, the observation equation is

$$y_t = x_t + \varepsilon_t,$$

where ε_t is Gaussian with mean 0 and standard deviation 0.75. Together, the latent process and observation equations compose a state-space model.

Use the random latent state process (`x`) and the observation equation to generate observations.

```
y = x + 0.75*randn(T,1);
```

Specify the four coefficient matrices.

```
A = 0.5;
B = 1;
C = 1;
D = 0.75;
```

Specify the state-space model using the coefficient matrices.

```
Mdl = ssm(A,B,C,D)
```

```
Mdl =


State-space model type: <a href="matlab: doc ssm">ssm</a>

State vector length: 1
Observation vector length: 1
State disturbance vector length: 1
Observation innovation vector length: 1
Sample size supported by model: Unlimited

State variables: x1, x2,...
State disturbances: u1, u2,...
Observation series: y1, y2,...
Observation innovations: e1, e2,...
```

```
State equation:
x1(t) = (0.50)x1(t-1) + u1(t)

Observation equation:
y1(t) = x1(t) + (0.75)e1(t)

Initial state distribution:

Initial state means
 x1
  0

Initial state covariance matrix
     x1
 x1  1.33

State types
     x1
 Stationary
```

`Mdl` is an `ssm` model. Verify that the model is correctly specified using the display in the Command Window. The software infers that the state process is stationary. Subsequently, the software sets the initial state mean and covariance to the mean and variance of the stationary distribution of an AR(1) model.

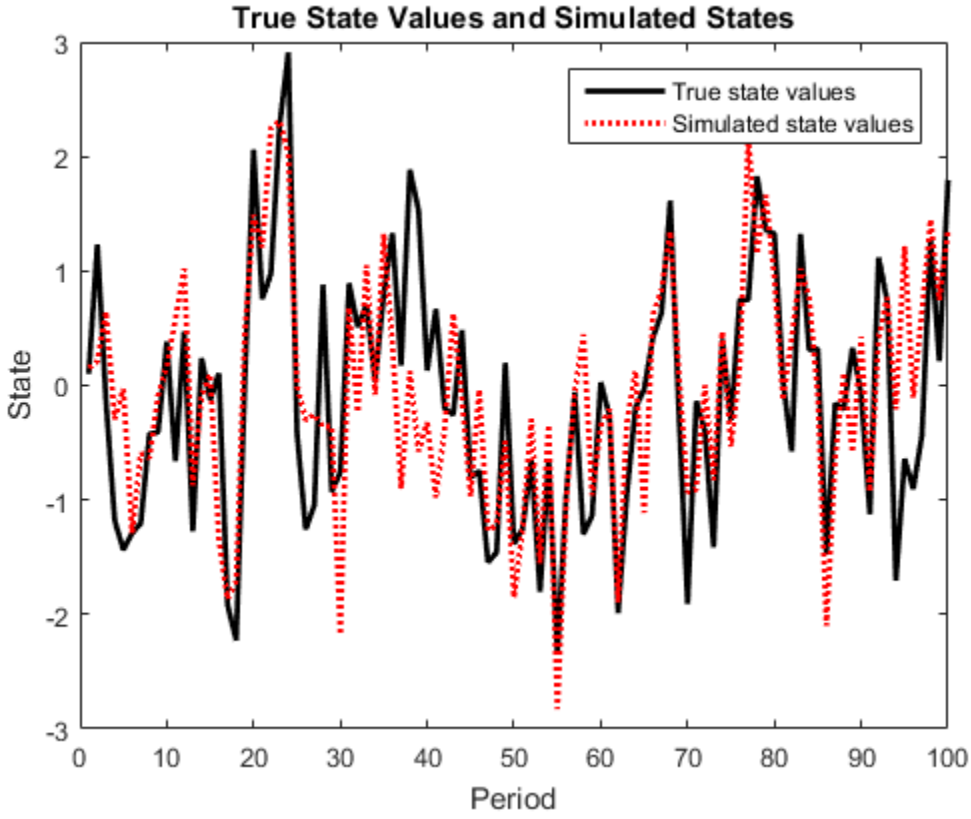
Simulate one path each of states and observations. Specify that the paths span 100 periods.

```
simX = simsmooth(Mdl,y);
```

`simX` is a 100-by-1 vector of simulated states.

Plot the true state values with the simulated states.

```
figure;
plot(1:T,x,'-k',1:T,simX,':r','LineWidth',2);
title 'True State Values and Simulated States';
xlabel 'Period';
ylabel 'State';
legend({'True state values','Simulated state values'});
```



By default, `simulate` simulates one path for each state in the state-space model. To conduct a Monte Carlo study, specify to simulate a large number of paths using the `'NumPaths'` name-value pair argument.

Estimate Posterior Distribution of States in State-Space Model

The `simsmooth` function draws random samples from the distribution of smoothed states, or the distribution of a state given all of the data and parameters. This is the definition of posterior distribution of a state. Suppose that a latent process is an AR(1). Subsequently, the state equation is

$$x_t = 0.5x_{t-1} + u_t,$$

where u_t is Gaussian with mean 0 and standard deviation 0.5.

Generate a random series of 100 observations from x_t, assuming that the series starts at 1.5.

```
T = 100;
ARMdl = arima('AR',0.5,'Constant',0,'Variance',0.5^2);
x0 = 1.5;
rng(1); % For reproducibility
x = simulate(ARMdl,T,'Y0',x0);
```

Suppose further that the latent process is subject to additive measurement error. Subsequently, the observation equation is

$$y_t = x_t + \varepsilon_t,$$

where ε_t is Gaussian with mean 0 and standard deviation 0.05. Together, the latent process and observation equations compose a state-space model.

Use the random latent state process (`x`) and the observation equation to generate observations.

```
y = x + 0.05*randn(T,1);
```

Specify the four coefficient matrices.

```
A = 0.5;
B = 1;
C = 1;
D = 0.75;
```

Specify the state-space model using the coefficient matrices.

```
Mdl = ssm(A,B,C,D);
```

Smooth the states of the state space model.

```
xsmooth = smooth(Mdl,y);
```

Draw 1000 paths from the posterior distribution of x_1.

```
N = 1000;
SimX = simsmooth(Mdl,y,'NumPaths',N);
```

`SimX` is a `100`-by- `1`-by- `1000` array. Rows correspond to periods, columns correspond to individual states, and leaves correspond to separate paths.

Because `SimX` has a singleton dimension, collapse it so that its leaves correspond to the columns using squeeze.

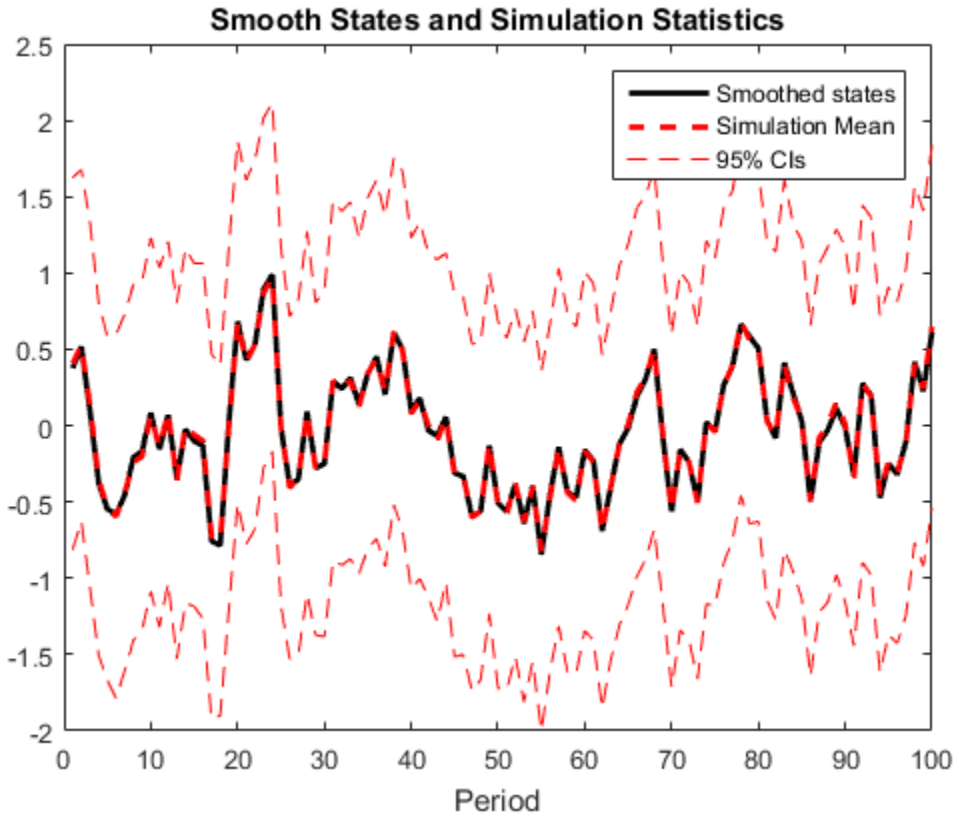
```
SimX = squeeze(SimX);
```

Compute the mean, standard deviation, and 95% confidence intervals of the state at each period.

```
xbar = mean(SimX,2);
xstd = std(SimX,[],2);
ci = [xbar - 1.96*xstd, xbar + 1.96*xstd];
```

Plot the smoothed states, and the means and 95% confidence intervals of the draws at each period.

```
figure;
plot(xsmooth,'k','LineWidth',2);
hold on;
plot(xbar,'--r','LineWidth',2);
plot(1:T,ci(:,1),'--r',1:T,ci(:,2),'--r');
legend('Smoothed states','Simulation Mean','95% CIs');
title('Smooth States and Simulation Statistics');
xlabel('Period')
```



- "Smooth States of State-Space Model" on page 8-80
- "Compare Simulation Smoother to Smoothed States" on page 8-162
- "Simulate States of Time-Varying State-Space Model Using Simulation Smoother" on page 8-112
- "Estimate Random Parameter of State-Space Model" on page 8-116

Algorithms

For increased speed in simulating states, the simulation smoother implements minimal dimensionality error checking. Therefore, for models with unknown parameter values,

you should ensure that the dimensions of the data and the dimensions of the coefficient matrices are consistent.

References

[1] Durbin J., and S. J. Koopman. "A Simple and Efficient Simulation Smoother for State Space Time Series Analysis." *Biometrika*. Vol 89., No. 3, 2002, pp. 603–615.

[2] Durbin J., and S. J. Koopman. *Time Series Analysis by State Space Methods*. 2nd ed. Oxford: Oxford University Press, 2012.

See Also

estimate | filter | simulate | smooth | ssm

More About

- "What Are State-Space Models?" on page 8-3

simulate

Monte Carlo simulation of conditional variance models

Syntax

```
V = simulate(Mdl,numObs)
V = simulate(Mdl,numObs,Name,Value)
[V,Y] = simulate( ___ )
```

Description

`V = simulate(Mdl,numObs)` simulates a `numObs`-period conditional variance path from the fully specified conditional variance model `Mdl`. `Mdl` can be a `garch`, `egarch`, or `gjr` model.

`V = simulate(Mdl,numObs,Name,Value)` simulates conditional variance paths with additional options specified by one or more `Name,Value` pair arguments. For example, you can generate multiple sample paths or specify presample innovation paths.

`[V,Y] = simulate( ___ )` additionally simulates response paths using any of the input arguments in the previous syntaxes.

Examples

Simulate GARCH Model Conditional Variances and Responses

Simulate conditional variance and response paths from a GARCH(1,1) model.

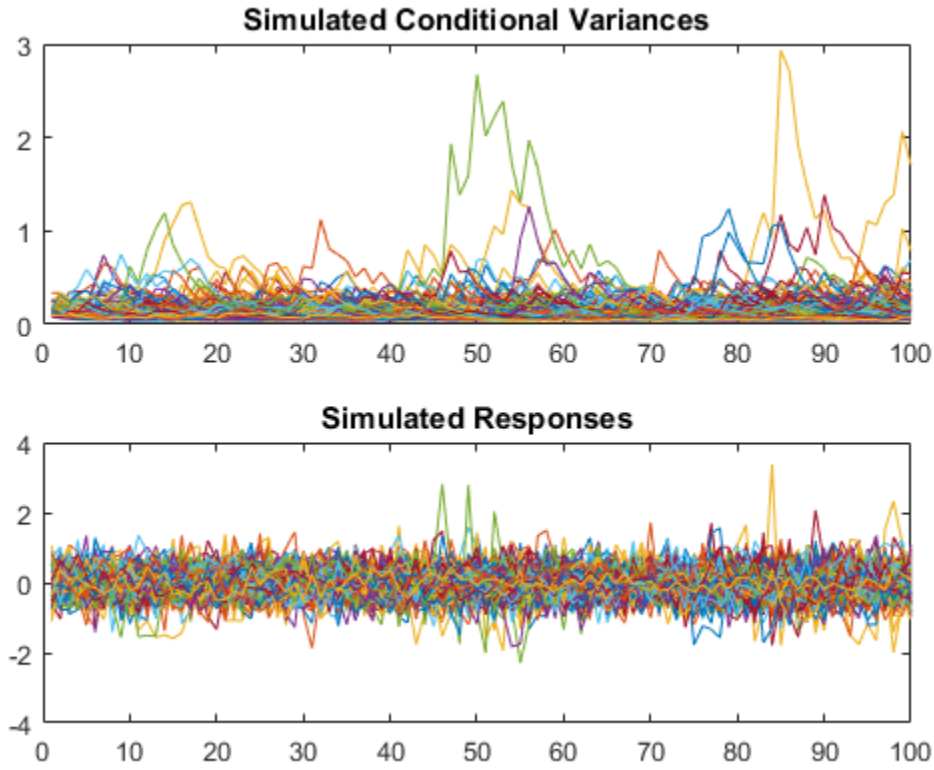
Specify a GARCH(1,1) model with known parameters.

```
Mdl = garch('Constant',0.01,'GARCH',0.7,'ARCH',0.2);
```

Simulate 500 sample paths, each with 100 observations.

```
rng default; % For reproducibility
[V,Y] = simulate(Mdl,100,'NumPaths',500);

figure
```

```
subplot(2,1,1)
plot(V)
title('Simulated Conditional Variances')

subplot(2,1,2)
plot(Y)
title('Simulated Responses')
```

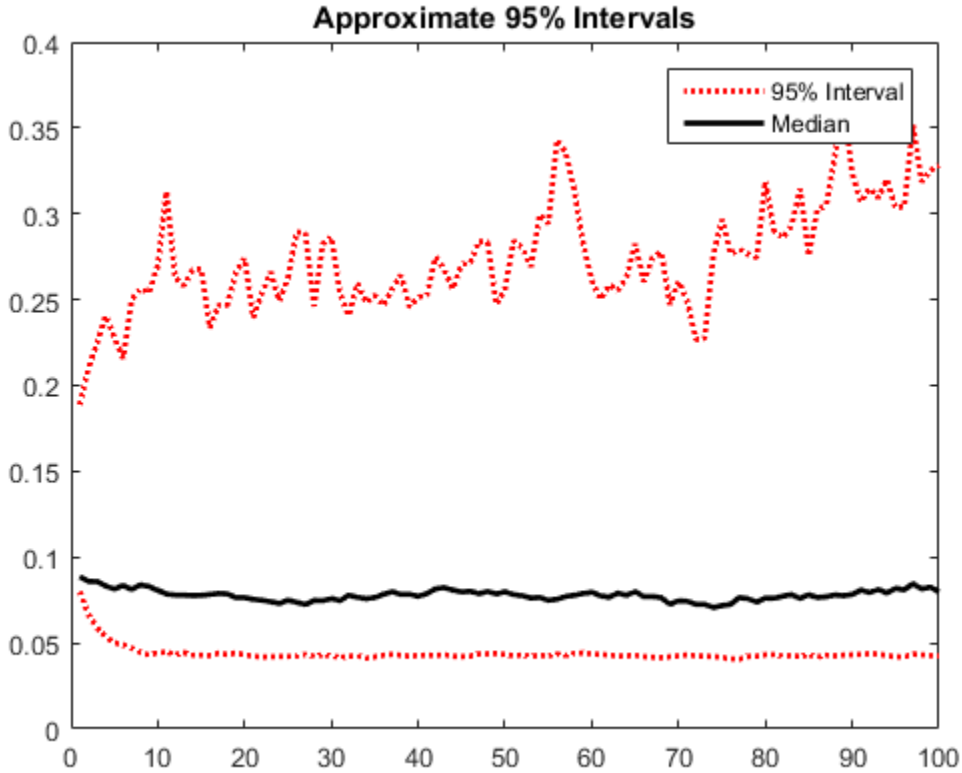


The simulated responses look like draws from a stationary stochastic process.

Plot the 2.5th, 50th (median), and 97.5th percentiles of the simulated conditional variances.

```
lower = prctile(V,2.5,2);
```

```
middle = median(V,2);
upper = prctile(V,97.5,2);

figure
plot(1:100,lower,'r:',1:100,middle,'k',...
   1:100,upper,'r:','LineWidth',2)
legend('95% Interval','Median')
title('Approximate 95% Intervals')
```



The intervals are asymmetric due to positivity constraints on the conditional variance.

Simulate EGARCH Model Conditional Variances and Responses

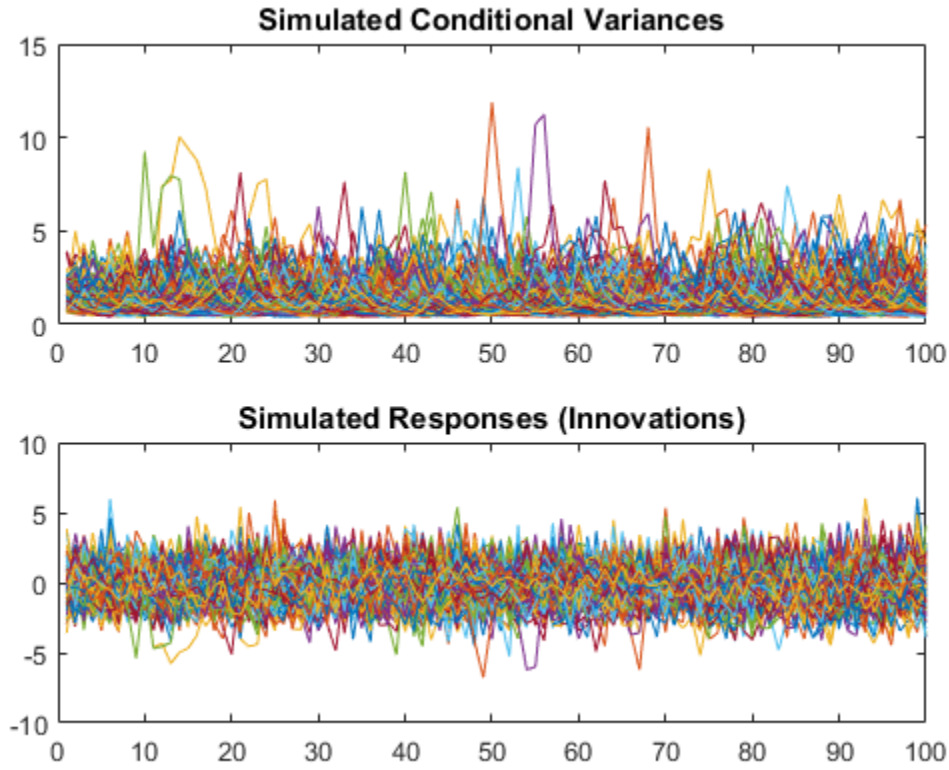
Simulate conditional variance and response paths from an EGARCH(1,1) model.

Specify an EGARCH(1,1) model with known parameters.

```
Mdl = egarch('Constant',0.001,'GARCH',0.7,'ARCH',0.2,...
             'Leverage',-0.3);
```

Simulate 500 sample paths, each with 100 observations.

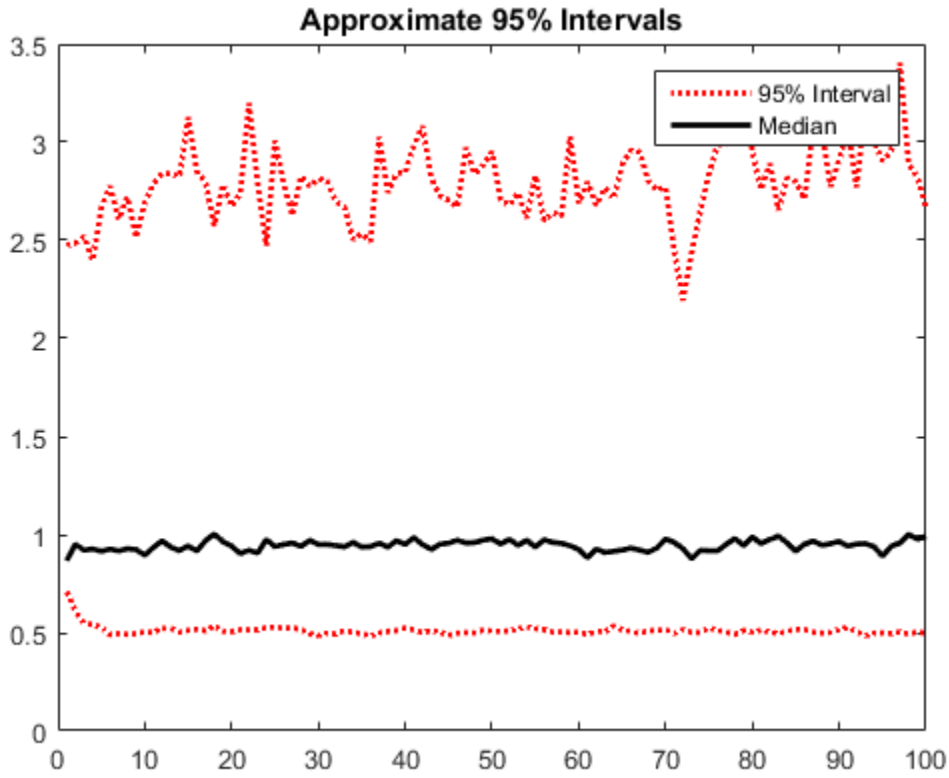
```
rng default; % For reproducibility
[V,Y] = simulate(Mdl,100,'NumPaths',500);

figure
subplot(2,1,1)
plot(V)
title('Simulated Conditional Variances')

subplot(2,1,2)
plot(Y)
title('Simulated Responses (Innovations)')
```



The simulated responses look like draws from a stationary stochastic process.

Plot the 2.5th, 50th (median), and 97.5th percentiles of the simulated conditional variances.

```
lower = prctile(V,2.5,2);
middle = median(V,2);
upper = prctile(V,97.5,2);

figure
plot(1:100,lower,'r:',1:100,middle,'k',...
     1:100, upper,'r:','LineWidth',2)
legend('95% Interval','Median')
title('Approximate 95% Intervals')
```



The intervals are asymmetric due to positivity constraints on the conditional variance.

Simulate GJR Model Conditional Variances and Responses

Simulate conditional variance and response paths from a GJR(1,1) model.

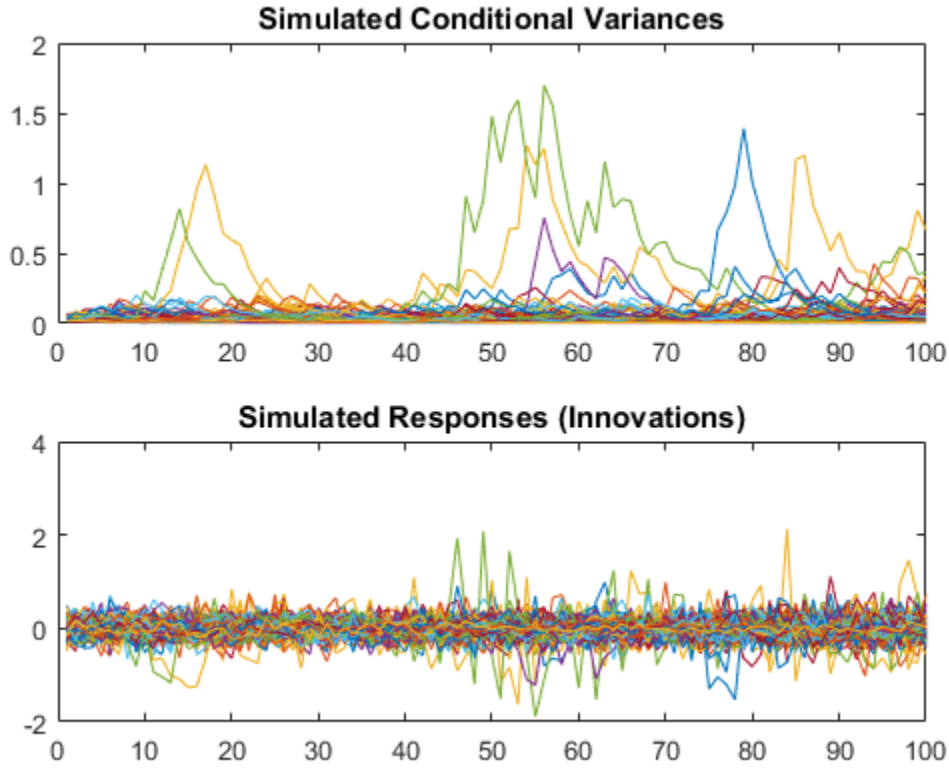
Specify a GJR(1,1) model with known parameters.

```
Mdl = gjr('Constant',0.001,'GARCH',0.7,'ARCH',0.2,...
    'Leverage',0.1);
```

Simulate 500 sample paths, each with 100 observations.

```
rng default; % For reproducibility
[V,Y] = simulate(Mdl,100,'NumPaths',500);
```

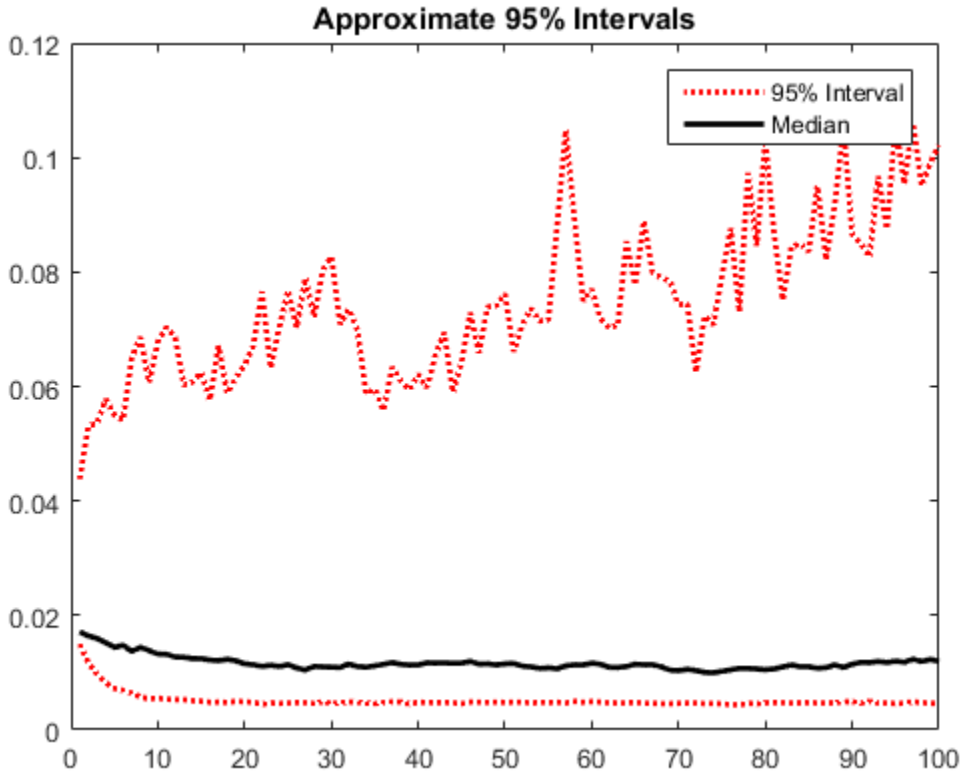
```
figure
subplot(2,1,1)
plot(V)
title('Simulated Conditional Variances')

subplot(2,1,2)
plot(Y)
title('Simulated Responses (Innovations)')
```



The simulated responses look like draws from a stationary stochastic process.

Plot the 2.5th, 50th (median), and 97.5th percentiles of the simulated conditional variances.

```
lower = prctile(V,2.5,2);
middle = median(V,2);
upper = prctile(V,97.5,2);

figure
plot(1:100,lower,'r:',1:100,middle,'k',...
     1:100, upper,'r:','LineWidth',2)
legend('95% Interval','Median')
title('Approximate 95% Intervals')
```



The intervals are asymmetric due to positivity constraints on the conditional variance.

Forecast Conditional Variances by Monte-Carlo Simulation

Simulate conditional variances of the daily NASDAQ Composite Index returns for 500 days. Use the simulations to make forecasts and approximate 95% forecast intervals. Compare the forecasts among GARCH(1,1), EGARCH(1,1), and GJR(1,1) fits.

Load the NASDAQ data included with the toolbox. Convert the index to returns.

```
load Data_EquityIdx
nasdaq = DataTable.NASDAQ;
r = price2ret(nasdaq);
T = length(r);
```

Fit GARCH(1,1), EGARCH(1,1), and GJR(1,1) models to the entire data set. Infer conditional variances to use as presample conditional variances for the forecast simulation.

```
Mdl = cell(3,1); % Preallocation
Mdl{1} = garch(1,1);
Mdl{2} = egarch(1,1);
Mdl{3} = gjr(1,1);

EstMdl = cellfun(@(x)estimate(x,r,'Display','off'),Mdl,...
    'UniformOutput',false);
v0 = cellfun(@(x)infer(x,r),EstMdl,'UniformOutput',false);
```

`EstMdl` is 3-by-1 cell vector. Each cell is a different type of estimated conditional variance model, e.g., `EstMdl{1}` is an estimated GARCH(1,1) model. `V0` is a 3-by-1 cell vector, and each cell contains the inferred conditional variances from the corresponding, estimated model.

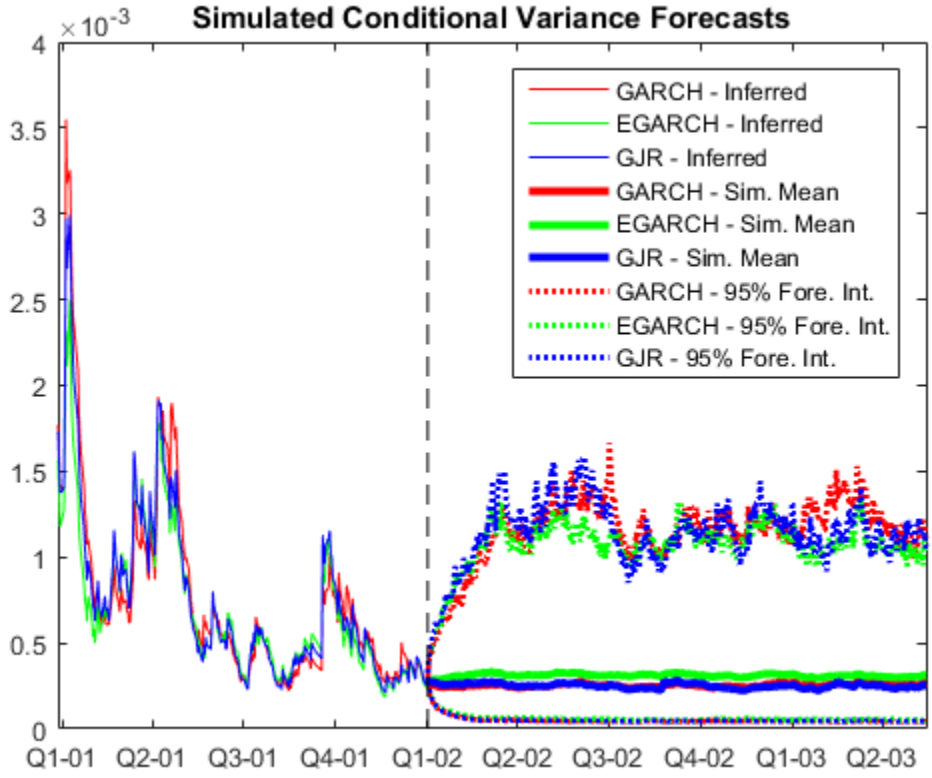
Simulate 1000 samples paths with 500 observations each. Use the observed returns and inferred conditional variances as presample data.

```
vSim = cell(3,1); % Preallocation

for j = 1:3
    rng default; % For reproducibility
    vSim{j} = simulate(EstMdl{j},500,'NumPaths',1000,'E0',r,'V0',v0{j});
end
```

`vSim` is a 3-by-1 cell vector, and each cell contains a 500-by-1000 matrix of simulated conditional variances generated from the corresponding, estimated model.

Plot the simulation mean forecasts and approximate 95% forecast intervals, along with the conditional variances inferred from the data.

```
lower = cellfun(@(x)prctile(x,2.5,2),vSim,'UniformOutput',false);
upper = cellfun(@(x)prctile(x,97.5,2),vSim,'UniformOutput',false);
mn = cellfun(@(x)mean(x,2),vSim,'UniformOutput',false);
datesPlot = dates(end - 250:end);
datesFH = dates(end) + (1:500)';

h = zeros(3,4);

figure
for j = 1:3
    col = zeros(1,3);
    col(j) = 1;
    h(j,1) = plot(datesPlot,v0{j}(end-250:end),'Color',col);
    hold on
    h(j,2) = plot(datesFH,mn{j},'Color',col,'LineWidth',3);
    h(j,3:4) = plot([datesFH datesFH],[lower{j} upper{j}],':',...
        'Color',col,'LineWidth',2);
end
hGCA = gca;
plot(datesFH(1)*[1 1],hGCA.YLim,'k--');
datetick;
axis tight;
h = h(:,1:3);
legend(h(:),'GARCH - Inferred','EGARCH - Inferred','GJR - Inferred',...
      'GARCH - Sim. Mean','EGARCH - Sim. Mean','GJR - Sim. Mean',...
      'GARCH - 95% Fore. Int.','EGARCH - 95% Fore. Int.',...
      'GJR - 95% Fore. Int.','Location','NorthEast')
title('Simulated Conditional Variance Forecasts')
hold off
```



- "Simulate GARCH Models" on page 6-97
- "Simulate Conditional Variance Model" on page 6-111
- "Assess EGARCH Forecast Bias Using Simulations" on page 6-104

Input Arguments

`Mdl` — Conditional variance model

`garch` model object | `egarch` model object | `gjr` model object

Conditional variance model without any unknown parameters, specified as a `garch`, `egarch`, or `gjr` model object.

`Mdl` cannot contain any properties that have `NaN` value.

`numObs` — Sample path length
positive integer

Sample path length, specified as a positive integer. That is, the number of random observations to generate per output path. `V` and `Y` have `numObs` rows.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'numPaths',1000,'E0',[0.5; 0.5]` specifies to generate `1000` sample paths and to use `[0.5; 0.5]` as presample innovations per path.

`'NumPaths'` — Number of sample paths to generate
`1` (default) | positive integer

Number of sample paths to generate, specified as the comma-separated pair consisting of `'NumPaths'` and a positive integer. `V` and `Y` have `NumPaths` columns.

Example: `'NumPaths',1000`

Data Types: `double`

`'E0'` — Presample innovations
numeric column vector | numeric matrix

Presample innovations, specified as the comma-separated pair consisting of `'E0'` and a numeric column vector or matrix. The presample innovations provide initial values for the innovations process of the conditional variance model `Mdl`. The presample innovations derive from a distribution with mean 0.

`E0` must contain at least `Mdl.Q` elements or rows. If `E0` contains extra rows, `simulate` uses the latest `Mdl.Q` only.

The last element or row contains the latest presample innovation.

- If `E0` is a column vector, it represents a single path of the underlying innovation series. `simulate` applies `E0` to each simulated path.

- If `E0` is a matrix, then each column represents a presample path of the underlying innovation series. `E0` must have at least `NumPaths` columns. If `E0` has more columns than necessary, `simulate` uses the first `NumPaths` columns only.

The defaults are:

- For GARCH(P,Q) and GJR(P,Q) models, `simulate` sets any necessary presample innovations to an independent sequence of disturbances with mean zero and standard deviation equal to the unconditional standard deviation of the conditional variance process.
- For EGARCH(P,Q) models, `simulate` sets any necessary presample innovations to an independent sequence of disturbances with mean zero and variance equal to the exponentiated unconditional mean of the logarithm of the EGARCH variance process.

Example: `'E0',[0.5; 0.5]`

`'V0'` — Positive presample conditional variance paths

numeric column vector | numeric matrix

Positive presample conditional variance paths, specified as a numeric vector or matrix. `V0` provides initial values for the conditional variances in the model.

- If `V0` is a column vector, then `simulate` applies it to each output path.
- If `V0` is a matrix, then it must have at least `NumPaths` columns. If `V0` has more columns than necessary, `simulate` uses the first `NumPaths` columns only.
- For GARCH(P,Q) and GJR(P,Q) models:
  - `V0` must have at least `Mdl.P` rows to initialize the variance equation.
  - By default, `simulate` sets any necessary presample variances to the unconditional variance of the conditional variance process.
- For EGARCH(P,Q) models, `simulate`:
  - `V0` must have at least `max(Mdl.P,Mdl.Q)` rows to initialize the variance equation.
  - By default, `simulate` sets any necessary presample variances to the exponentiated unconditional mean of the logarithm of the EGARCH variance process.

If the number of rows in `V0` exceeds the number necessary, then `simulate` uses the latest, required number of observations only. The last element or row contains the latest observation.

Example: `'V0',[1; 0.5]`

Data Types: `double`

Notes

- If `E0` and `V0` are column vectors, `simulate` applies them to every column of the outputs `V` and `Y`. This application allows simulated paths to share a common starting point for Monte Carlo simulation of forecasts and forecast error distributions.
- `NaN`s indicate missing values. `simulate` removes missing values. The software merges the presample data (`E0` and `V0`), and then uses list-wise deletion to remove any rows containing at least one `NaN`. Removing `NaN`s in the data reduces the sample size. Removing `NaN`s can also create irregular time series.
- `simulate` assumes that you synchronize presample data such that the latest observation of each presample series occurs simultaneously.

Output Arguments

`V` — Simulated conditional variance paths

numeric column vector | numeric matrix

Simulated conditional variance paths of the mean-zero innovations associated with `Y`, returned as a numeric column vector or matrix.

`V` is a `numObs`-by-`NumPaths` matrix, in which each column corresponds to a simulated conditional variance path. Rows of `V` are periods corresponding to the periodicity of `Mdl`.

`Y` — Simulated response paths

numeric column vector | numeric matrix

Simulated response paths, returned as a numeric column vector or matrix. `Y` usually represents a mean-zero, heteroscedastic time series of innovations with conditional variances given in `V` (a continuation of the presample innovation series `E0`).

`Y` can also represent a time series of mean-zero, heteroscedastic innovations plus an offset. If `Mdl` includes an offset, then `simulate` adds the offset to the underlying mean-zero, heteroscedastic innovations so that `Y` represents a time series of offset-adjusted innovations.

`Y` is a `numObs`-by-`NumPaths` matrix, in which each column corresponds to a simulated response path. Rows of `Y` are periods corresponding to the periodicity of `Mdl`.

More About

- Using garch Objects
- Using egarch Objects
- Using gjr Objects
- "Monte Carlo Simulation of Conditional Variance Models" on page 6-92
- "Presample Data for Conditional Variance Model Simulation" on page 6-95
- "Monte Carlo Forecasting of Conditional Variance Models" on page 6-115

References

[1] Bollerslev, T. "Generalized Autoregressive Conditional Heteroskedasticity." *Journal of Econometrics*. Vol. 31, 1986, pp. 307–327.

[2] Bollerslev, T. "A Conditionally Heteroskedastic Time Series Model for Speculative Prices and Rates of Return." *The Review of Economics and Statistics*. Vol. 69, 1987, pp. 542–547.

[3] Box, G. E. P., G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis: Forecasting and Control*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

[4] Enders, W. *Applied Econometric Time Series*. Hoboken, NJ: John Wiley & Sons, 1995.

[5] Engle, R. F. "Autoregressive Conditional Heteroskedasticity with Estimates of the Variance of United Kingdom Inflation." *Econometrica*. Vol. 50, 1982, pp. 987–1007.

[6] Glosten, L. R., R. Jagannathan, and D. E. Runkle. "On the Relation between the Expected Value and the Volatility of the Nominal Excess Return on Stocks." *The Journal of Finance*. Vol. 48, No. 5, 1993, pp. 1779–1801.

[7] Hamilton, J. D. *Time Series Analysis*. Princeton, NJ: Princeton University Press, 1994.

[8] Nelson, D. B. "Conditional Heteroskedasticity in Asset Returns: A New Approach." *Econometrica*. Vol. 59, 1991, pp. 347–370.

See Also

`egarch` | `estimate` | `filter` | `forecast` | `garch` | `gjr` | `infer` | `print`

Introduced in R2012a

simulate

Class: arima

Monte Carlo simulation of ARIMA or ARIMAX models

Syntax

```
[Y,E] = simulate(Mdl,numObs)
[Y,E,V] = simulate(Mdl,numObs)
[Y,E,V] = simulate(Mdl,numObs,Name,Value)
```

Description

`[Y,E] = simulate(Mdl,numObs)` simulates sample paths and innovations from the ARIMA model, `Mdl`. The responses can include the effects of seasonality.

`[Y,E,V] = simulate(Mdl,numObs)` additionally simulates conditional variances, `V`.

`[Y,E,V] = simulate(Mdl,numObs,Name,Value)` simulates sample paths with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

`Mdl`

ARIMA or ARIMAX model, specified as an arima model returned by `arima` or estimate.

The properties of `Mdl` cannot contain `NaN`s.

`numObs`

Positive integer that indicates the number of observations (rows) to generate for each path of the outputs `Y`, `E`, and `V`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'E0'`

Mean zero presample innovations that provide initial values for the model. `E0` is a column vector or a matrix with at least `NumPaths` columns and enough rows to initialize the model and any conditional variance model. The number of observations required is at least `Mdl.Q`, but can be more if you specify a conditional variance model. If the number of rows exceeds the number necessary, then `simulate` only uses the most recent observations. If the number of columns exceeds `NumPaths`, then `simulate` only uses the first `NumPaths` columns. If `E0` is a column vector, then it is applied to each simulated path. The last row contains the most recent presample observation.

Default: `simulate` sets the necessary presample observations to 0.

`'NumPaths'`

Positive integer that indicates the number of sample paths (columns) to generate.

Default: 1

`'V0'`

Positive presample conditional variances which provide initial values for any conditional variance model. If the variance of the model is constant, then `V0` is unnecessary. `V0` is a column vector or a matrix with at least `NumPaths` columns and enough rows to initialize the variance model. If the number of rows exceeds the number necessary, then `simulate` only uses the most recent observations. If the number of columns exceeds `NumPaths`, then `simulate` only uses the first `NumPaths` columns. If `V0` is a column vector, then `simulate` applies it to each simulated path. The last row contains the most recent observation.

Default: `simulate` sets the necessary presample observations to the unconditional variance of the conditional variance process.

`'X'`

Matrix of predictor data with length `Mdl.Beta` columns of separate series. The number of observations (rows) of `X` must equal or exceed `numObs`. If the number of observations

of X exceeds numObs, then simulate only uses the most recent observations. simulate applies the *entire* matrix X to *each* simulated response series. The last row contains the most recent observation.

Default: simulate does not use a regression component regardless of the value of Mdl.Beta.

'Y0'

Presample response data that provides initial values for the model. Y0 is a column vector or a matrix with at least Mdl.P rows and NumPaths columns. If the number of rows exceeds Mdl.P, then simulate only uses the most recent Mdl.P observations. If the number of columns exceeds NumPaths, then simulate only uses the first NumPaths columns. If Y0 is a column vector, then it is applied to each simulated path. The last row contains the most recent presample observation.

Default: simulate sets the necessary presample observations to the unconditional mean if the AR process is stable, or to 0 for unstable processes or when you specify X.

Notes

- NaNs indicate missing values, and simulate removes them. The software merges the presample data, then uses list-wise deletion to remove any NaNs in the presample data matrix or X. That is, simulate sets PreSample = [Y0 E0 V0], then it removes any row in PreSample or X that contains at least one NaN.
- The removal of NaNs in the main data reduces the effective sample size. Such removal can also create irregular time series.
- simulate assumes that you synchronize the predictor series such that the most recent observations occur simultaneously. The software also assumes that you synchronize the presample series similarly.

Output Arguments

Y

numObs-by-NumPaths matrix of simulated response data.

E

`numObs`-by-`NumPaths` matrix of simulated mean zero innovations.

V

`numObs`-by-`NumPaths` matrix of simulated conditional variances of the innovations in `E`.

Examples

Simulate Responses and Innovations

Simulate response and innovation paths from a multiplicative seasonal model.

Specify the model

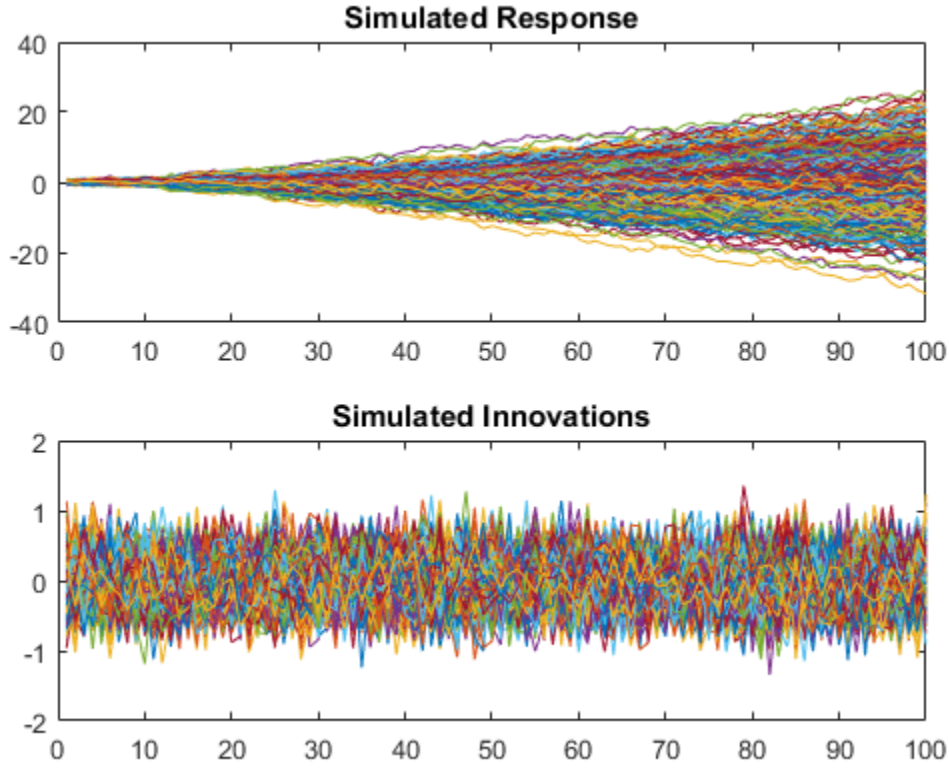
$$(1-0.2L)x_{it} = 2 + (1+0.5L-0.3L^2)\eta_{it},$$

where ε_t follows a Gaussian distribution with mean 0 and variance 0.1.

```
Mdl = arima('MA',-0.5,'SMA',0.3,...
 'SMALags',12,'D',1,'Seasonality',12,...
 'Variance',0.1,'Constant',0);
```

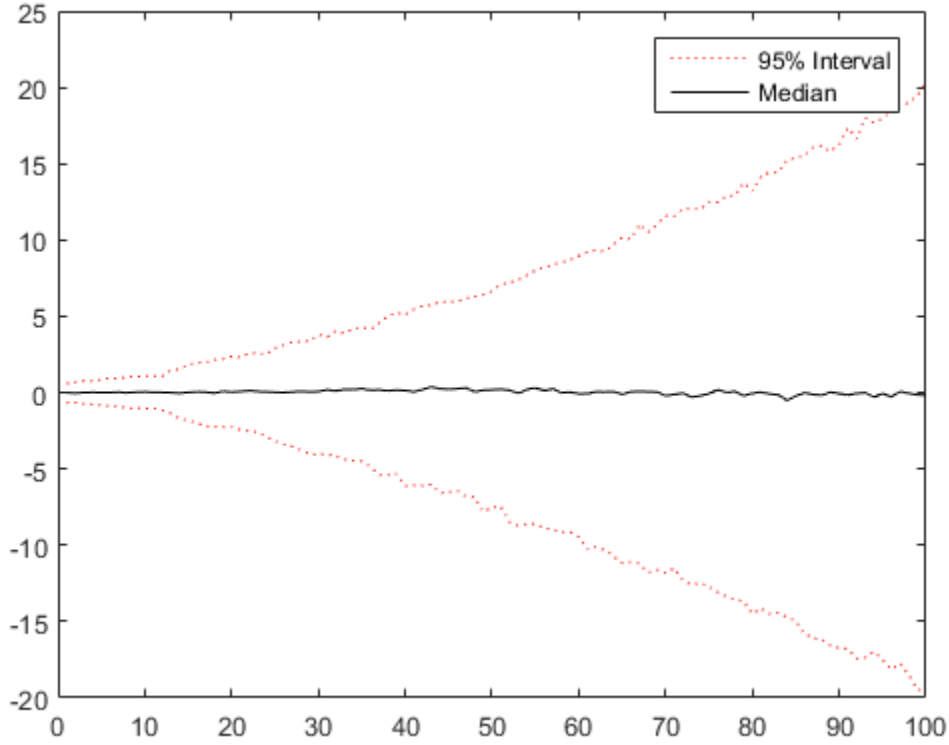
Simulate 500 paths with 100 observations each.

```
rng default % For reproducibility
[Y, E] = simulate(Mdl,100,'NumPaths',500);

figure
subplot(2,1,1);
plot(Y)
title('Simulated Response')

subplot(2,1,2);
plot(E)
title('Simulated Innovations')
```



Plot the 2.5th, 50th (median), and 97.5th percentiles of the simulated response paths.

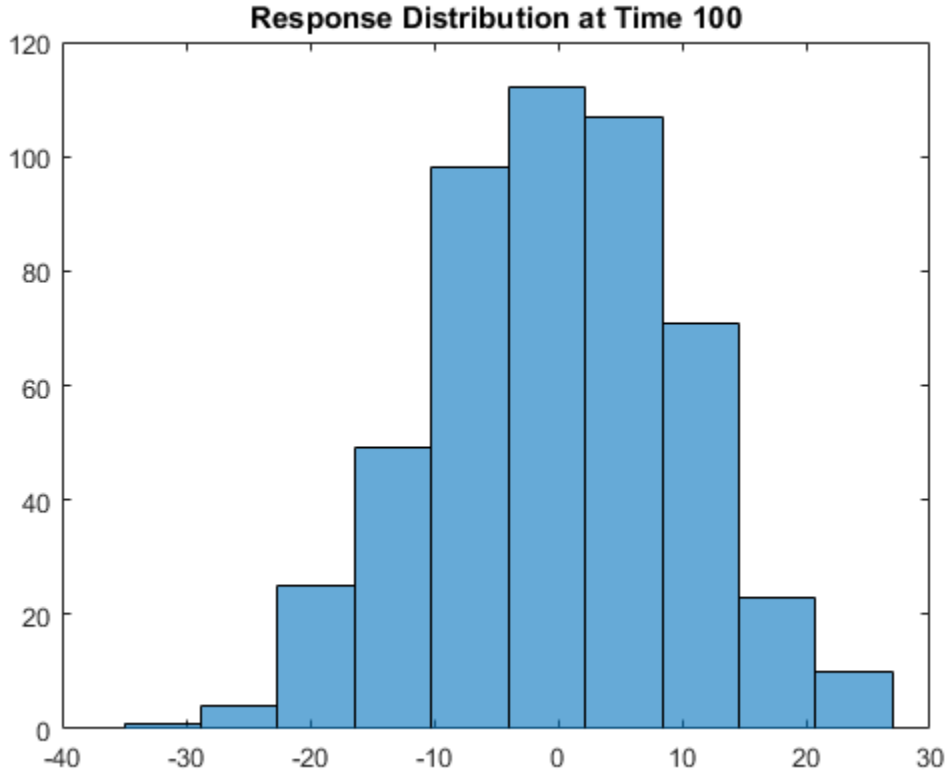
```
lower = prctile(Y,2.5,2);
middle = median(Y,2);
upper = prctile(Y,97.5,2);

figure
plot(1:100,lower,'r:',1:100,middle,'k',...
   1:100,upper,'r:')
legend('95% Interval','Median')
```



Compute statistics across the second dimension (across paths) to summarize the sample paths.

Plot a histogram of the simulated paths at time 100.

```
figure
histogram(Y(100,:),10)
title('Response Distribution at Time 100')
```



Simulate Predictors and Responses

Simulate three predictor series and a response series.

Specify and simulate a path of length 20 for each of the three predictor series modeled by

$$(1-0.2L)x_{it} = 2 + (1+0.5L-0.3L^2)\eta_{it},$$

where η_{it} follows a Gaussian distribution with mean 0 and variance 0.01, and $i = \{1,2,3\}$.

```
[MdlX1,MdlX2,MdlX3] = deal(arima('AR',0.2,'MA',...
    {0.5,-0.3},'Constant',2,'Variance',0.01));
```

```
rng(4); % For reproducibility
simX1 = simulate(MdlX1,20);
simX2 = simulate(MdlX2,20);
simX3 = simulate(MdlX3,20);
SimX = [simX1 simX2 simX3];
```

Specify and simulate a path of length 20 for the response series modeled by

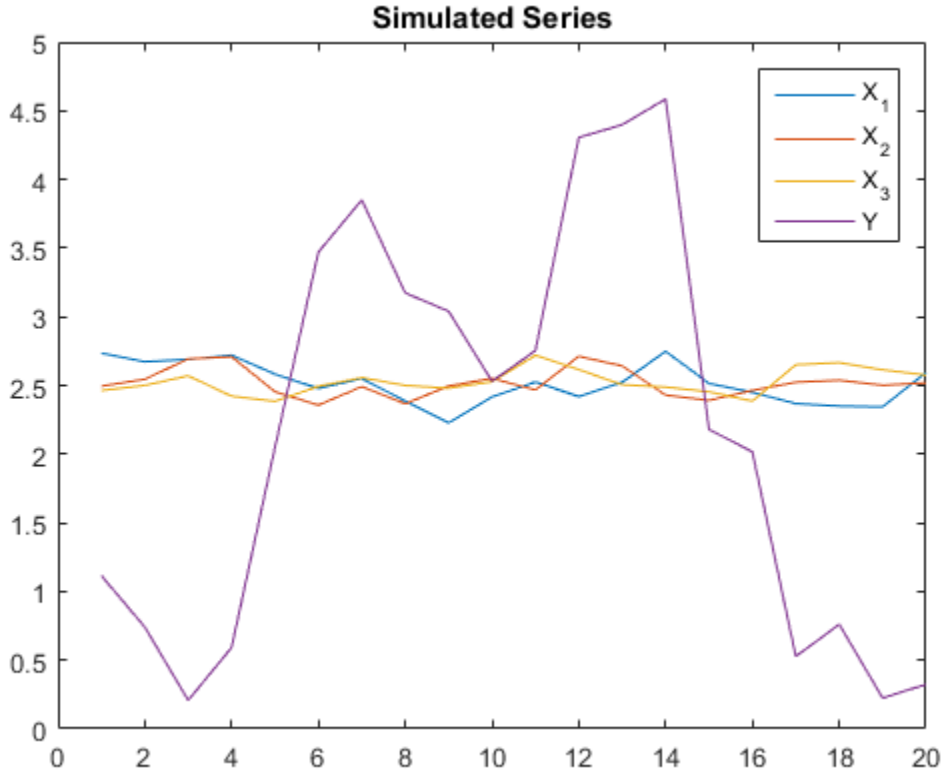
$$(1-0.05L+0.02L^2-0.01L^3)(1-L)^1 y_t = 0.05 + x_t' \begin{bmatrix} 0.5 \\ -0.03 \\ -0.7 \end{bmatrix} + (1+0.04L+0.01L^2)\varepsilon_t,$$

where ε_t follows a Gaussian distribution with mean 0 and variance 1.

```
MdlY = arima('AR',{0.05 -0.02 0.01},'MA',...
    {0.04,0.01},'D',1,'Constant',0.5,'Variance',1,...
    'Beta',[0.5 -0.03 -0.7]);
simY = simulate(MdlY,20,'X',SimX);
```

Plot the series together.

```
figure
plot([SimX simY])
title('Simulated Series')
legend('{X_1}','{X_2}','{X_3}','Y')
```



Forecast a Process Using Simulations

Forecast the daily NASDAQ Composite Index using Monte Carlo simulations.

Load the NASDAQ data included with the toolbox. Extract the first 1500 observations for fitting.

```
load Data_EquityIdx
nasdaq = DataTable.NASDAQ(1:1500);
n = length(nasdaq);
```

Specify, and then fit an ARIMA(1,1,1) model.

```
NasdaqModel = arima(1,1,1);
```

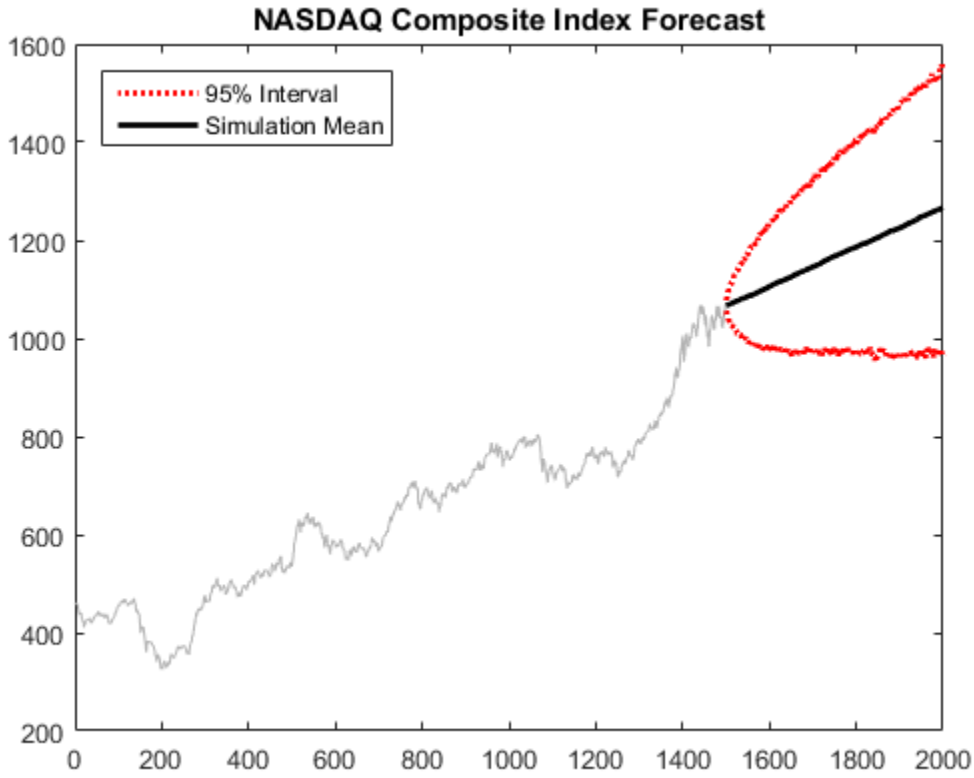
```
NasdaqFit = estimate(NasdaqModel,nasdaq);


    ARIMA(1,1,1) Model:
    --------------------
    Conditional Probability Distribution: Gaussian

                               Standard          t
     Parameter       Value       Error       Statistic
    -----------   -----------   ------------   -----------
     Constant       0.430313     0.185554        2.31907
        AR{1}     -0.0743894     0.081985      -0.907353
        MA{1}       0.311256    0.0772657        4.02838
     Variance         27.826     0.636248        43.7346
```

Simulate 1000 paths with 500 observations each. Use the observed data as presample data.

```
rng default;
Y = simulate(NasdaqFit,500,'NumPaths',1000,'Y0',nasdaq);
```

Plot the simulation mean forecast and approximate 95% forecast intervals.

```
lower = prctile(Y,2.5,2);
upper = prctile(Y,97.5,2);
mn = mean(Y,2);

figure
plot(nasdaq,'Color',[.7,.7,.7])
hold on
h1 = plot(n+1:n+500,lower,'r:','LineWidth',2);
plot(n+1:n+500,upper,'r:','LineWidth',2)
h2 = plot(n+1:n+500,mn,'k','LineWidth',2);

legend([h1 h2],'95% Interval','Simulation Mean',...
   'Location','NorthWest')
title('NASDAQ Composite Index Forecast')
hold off
```



- "Simulate Stationary Processes" on page 5-151
- "Simulate Trend-Stationary and Difference-Stationary Processes" on page 5-163
- "Simulate Multiplicative ARIMA Models" on page 5-169
- "Simulate Conditional Mean and Variance Models" on page 5-175

References

[1] Box, G. E. P., G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis: Forecasting and Control* 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

[2] Enders, W. *Applied Econometric Time Series*. Hoboken, NJ: John Wiley & Sons, 1995.

[3] Hamilton, J. D. *Time Series Analysis*. Princeton, NJ: Princeton University Press, 1994.

See Also

arima | estimate | filter | forecast | impulse | infer | print

More About

- "Monte Carlo Simulation of Conditional Mean Models" on page 5-146
- "Presample Data for Conditional Mean Model Simulation" on page 5-149
- "Transient Effects in Conditional Mean Model Simulations" on page 5-150
- "Monte Carlo Forecasting of Conditional Mean Models" on page 5-181

simulate

Class: regARIMA

Monte Carlo simulation of regression model with ARIMA errors

Syntax

```
[Y,E] = simulate(Mdl,numObs)
[Y,E,U] = simulate(Mdl,numObs)
[Y,E,U] = simulate(Mdl,numObs,Name,Value)
```

Description

`[Y,E] = simulate(Mdl,numObs)` simulates one sample path of observations (`Y`) and innovations (`E`) from the regression model with ARIMA time series errors, `Mdl`. The software simulates `numObs` observations and innovations per sample path.

`[Y,E,U] = simulate(Mdl,numObs)` additionally simulates unconditional disturbances, `U`.

`[Y,E,U] = simulate(Mdl,numObs,Name,Value)` simulates sample paths with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

`Mdl`

Regression model with ARIMA errors, specified as a regARIMA model returned by `regARIMA` or estimate.

The properties of `Mdl` cannot contain `NaN`s.

`numObs`

Number of observations (rows) to generate for each path of `Y`, `E`, and `U`, specified as a positive integer.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'E0'`

Presample innovations that have mean 0 and provide initial values for the ARIMA error model, specified as the comma-separated pair consisting of `'E0'` and a column vector or matrix.

- If `E0` is a column vector, then it is applied to each inferred path.
- If `E0` is a matrix, then it requires at least `NumPaths` columns. If `E0` contains more columns than required, then `simulate` uses the first `NumPaths` columns.
- `E0` must contain at least `Mdl.Q` rows. If `E0` contains more rows than required, then `simulate` uses the latest presample innovations. The last row contains the latest presample innovation.

Default: `simulate` sets the necessary presample innovations to 0.

`'NumPaths'`

Number of sample paths (columns) to generate for `Y`, `E`, and `U`, specified as the comma-separated pair consisting of `'NumPaths'` and a positive integer.

Default: 1

`'U0'`

Presample unconditional disturbances that provide initial values for the ARIMA error model, specified as the comma-separated pair consisting of `'U0'` and a column vector or matrix.

- If `U0` is a column vector, then it is applied to each inferred path.
- If `U0` is a matrix, then it requires at least `NumPaths` columns. If `U0` contains more columns than required, then `infer` uses the first `NumPaths` columns.
- `U0` must contain at least `Mdl.P` rows. If `U0` contains more rows than required, then `simulate` uses the latest presample unconditional disturbances. The last row contains the latest presample unconditional disturbance.

Default: `simulate` sets the necessary presample unconditional disturbances to 0.

`'X'`

Predictor data in the regression model, specified as the comma-separated pair consisting of `'X'` and a matrix.

The columns of `X` are separate, synchronized time series, with the last row containing the latest observations. `X` must have at least `numObs` rows. If the number of rows of `X` exceeds the number required, then `simulate` uses the latest observations.

Default: `simulate` does not use a regression component regardless of its presence in `Mdl`.

Notes

- `NaN`s in `E0`, `U0`, and `X` indicate missing values and `simulate` removes them. The software merges the presample data sets (`E0` and `U0`), then uses list-wise deletion to remove any `NaN`s. `simulate` similarly removes `NaN`s from `X`. Removing `NaN`s in the data reduces the sample size, and can also create irregular time series.
- `simulate` assumes that you synchronize presample data such that the latest observation of each presample series occurs simultaneously.
- All predictors (i.e., columns in `X`) are associated with each response path in `Y`.

Output Arguments

`Y`

Simulated responses, returned as a `numObs`-by-`NumPaths` matrix.

`E`

Simulated, mean 0 innovations, returned as a `numObs`-by-`NumPaths` matrix.

`U`

Simulated unconditional disturbances, returned as a `numObs`-by-`NumPaths` matrix.

Examples

Simulate Responses, Innovations, and Unconditional Disturbances

Simulate paths of responses, innovations, and unconditional disturbances from a regression model with $\mathrm{SARIMA}(2,1,1)_{12}$ errors.

Specify the model:

$$y_t = X\begin{bmatrix} 1.5 \\ -2 \end{bmatrix} + u_t$$
$$\left(1 - 0.2L - 0.1L^2\right)(1 - L)\left(1 - 0.01L^{12}\right)\left(1 - L^{12}\right)u_t = (1 + 0.5L)\left(1 + 0.02L^{12}\right)\varepsilon_t,$$

where ε_t follows a t-distribution with 15 degrees of freedom.

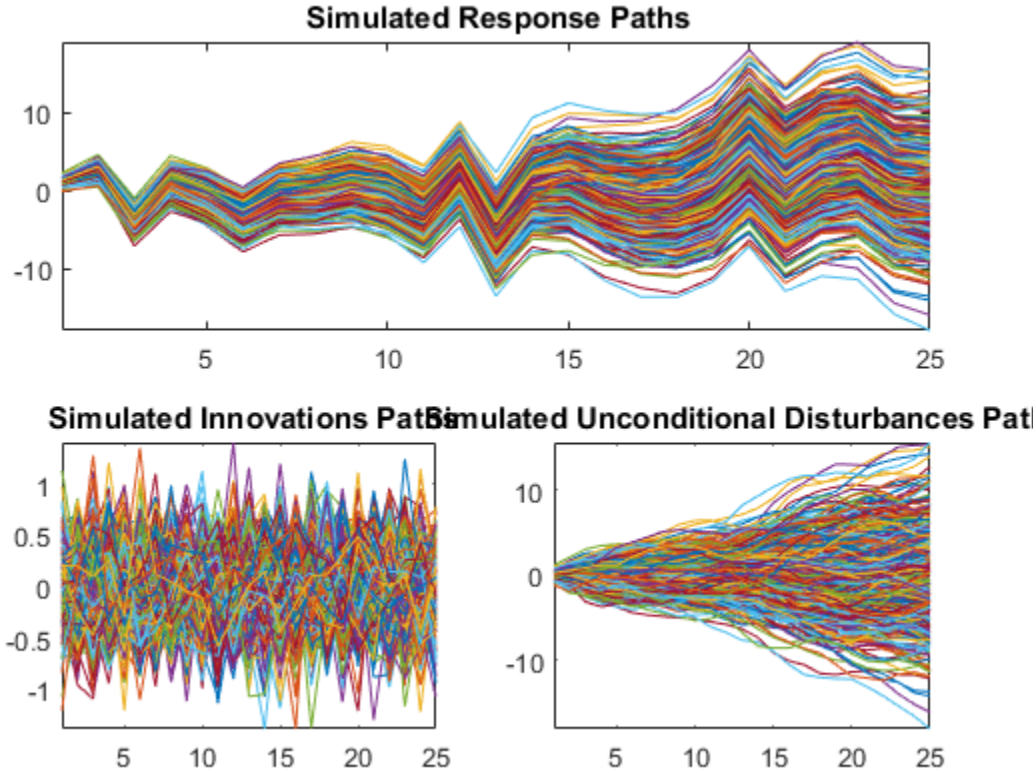
```
Distribution = struct('Name','t','DoF',15);
Mdl = regARIMA('AR',{0.2, 0.1},'MA',{0.5},'SAR',0.01,...
    'SARLags',12,'SMA',0.02,'SMALags',12,'D',1,...
    'Seasonality',12,'Beta',[1.5; -2],'Intercept',0,...
    'Variance',0.1,'Distribution',Distribution)
```

```
Mdl =

    Regression with ARIMA(2,1,1) Error Model Seasonally Integrated with Seasonal AR(12
    ----------------------------------------------------------------------------------
    Distribution: Name = 't', DoF = 15
       Intercept: 0
            Beta: [1.5 -2]
               P: 27
               D: 1
               Q: 13
              AR: {0.2 0.1} at Lags [1 2]
             SAR: {0.01} at Lags [12]
              MA: {0.5} at Lags [1]
             SMA: {0.02} at Lags [12]
     Seasonality: 12
        Variance: 0.1
```

Simulate and plot 500 paths with 25 observations each.

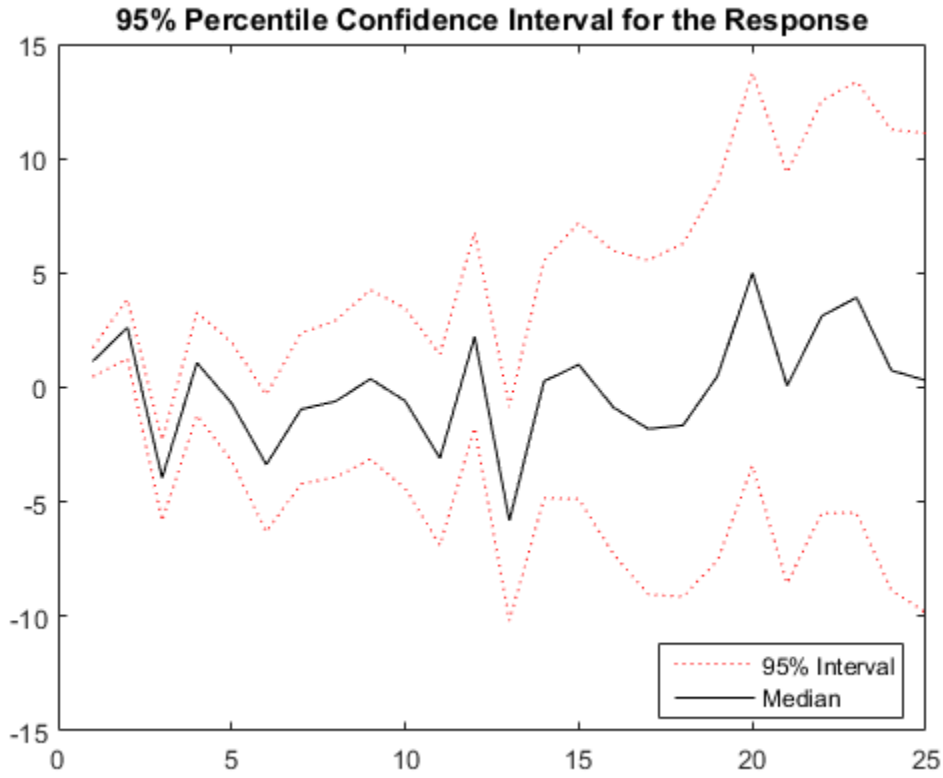
```
T = 25;
rng(1)
X = randn(T,2);
```

```
[Y,E,U] = simulate(Mdl,T,'NumPaths',500,'X',X);

figure
subplot(2,1,1);
plot(Y)
axis tight
title('{\bf Simulated Response Paths}')
subplot(2,2,3);
plot(E)
axis tight
title('{\bf Simulated Innovations Paths}')
subplot(2,2,4);
plot(U)
axis tight
title('{\bf Simulated Unconditional Disturbances Paths}')
```



Plot the 2.5th, 50th (median), and 97.5th percentiles of the simulated response paths.

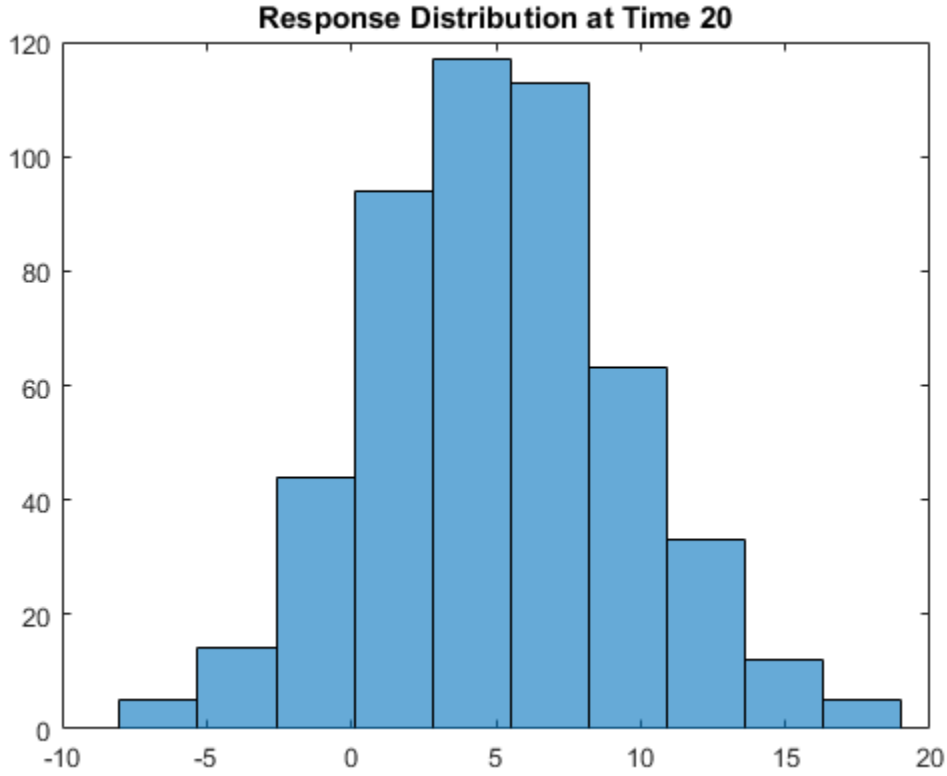
```
lower = prctile(Y,2.5,2);
middle = median(Y,2);
upper = prctile(Y,97.5,2);

figure
plot(1:25,lower,'r:',1:25,middle,'k',...
   1:25,upper,'r:')
title('\bf{95% Percentile Confidence Interval for the Response}')
legend('95% Interval','Median','Location','Best')
```



Compute statistics across the second dimension (across paths) to summarize the sample paths.

Plot a histogram of the simulated paths at time 20.

```
figure
histogram(Y(20,:),10)
title('Response Distribution at Time 20')
```



Forecast Stationary Process Using Monte Carlo Simulations

Regress the stationary, quarterly log GDP onto the CPI using a regression model with ARMA(1,1) errors, and forecast log GDP using Monte Carlo simulation.

Load the US Macroeconomic data set and preprocess the data.

```
load Data_USEconModel;
logGDP = log(DataTable.GDP);
dlogGDP = diff(logGDP);        % For stationarity
dCPI = diff(DataTable.CPIAUCSL); % For stationarity
numObs = length(dlogGDP);
gdp = dlogGDP(1:end-15);   % Estimation sample
cpi = dCPI(1:end-15);
```

```
T = length(gdp);        % Effective sample size
frstHzn =  T+1:numObs;  % Forecast horizon
hoCPI = dCPI(frstHzn);  % Holdout sample
dts = dates(2:end);     % Date nummbers
```

Fit a regression model with ARMA(1,1) errors.

```
ToEstMdl = regARIMA('ARLags',1,'MALags',1);
EstMdl = estimate(ToEstMdl,gdp,'X',cpi);
```

```
    Regression with ARIMA(1,0,1) Error Model:
    ----------------------------------------
    Conditional Probability Distribution: Gaussian

                                Standard          t
     Parameter       Value        Error       Statistic
    -----------   -----------   ------------   -----------
    Intercept      0.0147934    0.00162892        9.08176
        AR{1}       0.576012      0.100093        5.75477
        MA{1}      -0.152584      0.119784       -1.27382
        Beta1     0.00289724    0.00139893        2.07104
     Variance    9.57339e-05   6.55617e-06        14.6021
```

Infer unconditional disturbances.

```
[~,u0] = infer(EstMdl,gdp,'X',cpi);
```

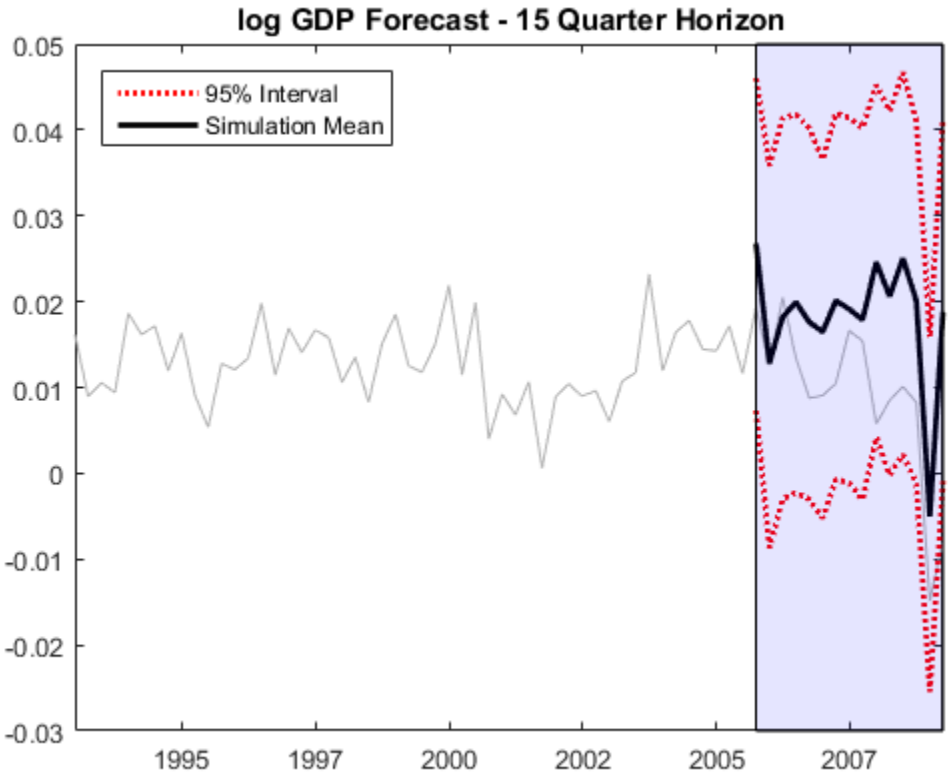
Simulate 1000 paths with 15 observations each. Use the inferred unconditional disturbances as presample data.

```
rng(1); % For reproducibility
gdpF = simulate(EstMdl,15,'NumPaths',1000,...
   'U0',u0,'X',hoCPI);
```

Plot the simulation mean forecast and approximate 95% forecast intervals.

```
lower = prctile(gdpF,2.5,2);
upper = prctile(gdpF,97.5,2);
mn = mean(gdpF,2);

figure
plot(dts(end-65:end),dlogGDP(end-65:end),'Color',[.7,.7,.7])
datetick
hold on
h1 = plot(dts(frstHzn),lower,'r:','LineWidth',2);
```

```
plot(dts(frstHzn),upper,'r:','LineWidth',2)
h2 = plot(dts(frstHzn),mn,'k','LineWidth',2);
legend([h1 h2],'95% Interval','Simulation Mean',...
 'Location','NorthWest')
h = gca;
ph = patch([repmat(dts(frstHzn(1)),1,2) repmat(dts(frstHzn(end)),1,2)],...
   [h.YLim fliplr(h.YLim)],...
   [0 0 0 0],'b');
ph.FaceAlpha = 0.1;
axis tight
title('{\bf log GDP Forecast - 15 Quarter Horizon}')
hold off
```



Forecast a Unit Root Nonstationary Process Using Monte Carlo Simulations

Regress the unit root nonstationary, quarterly log GDP onto the CPI using a regression model with ARIMA(1,1,1) errors with known intercept. Forecast log GDP using Monte Carlo simulation.

Load the US Macroeconomic data set and preprocess the data.

```
load Data_USEconModel;
numObs = length(DataTable.GDP);
logGDP = log(DataTable.GDP(1:end-15));
cpi = DataTable.CPIAUCSL(1:end-15);
T = length(logGDP);
frstHzn =  T+1:numObs;                % Forecast horizon
```

```
hoCPI = DataTable.CPIAUCSL(frstHzn);  % Holdout sample
```

Fit a regression model with ARIMA(1,1,1). The intercept is not identifiable in a model with integrated errors, so fix its value before estimation.

```
intercept = 5.867;
ToEstMdl = regARIMA('ARLags',1,'MALags',1,'D',1,...
   'Intercept',intercept);
EstMdl = estimate(ToEstMdl,logGDP,'X',cpi);
```

```
    Regression with ARIMA(1,1,1) Error Model:
    ----------------------------------------
    Conditional Probability Distribution: Gaussian

                               Standard          t
     Parameter       Value       Error       Statistic
    -----------   -----------   ------------   -----------
    Intercept         5.867         Fixed          Fixed
        AR{1}       0.92271     0.0309777        29.7863
        MA{1}      -0.38785     0.0603537       -6.42627
        Beta1    0.00396683    0.00164982         2.4044
     Variance   0.000108944   7.27203e-06        14.9813
```

Infer unconditional disturbances.

```
[~,u0] = infer(EstMdl,logGDP,'X',cpi);
```

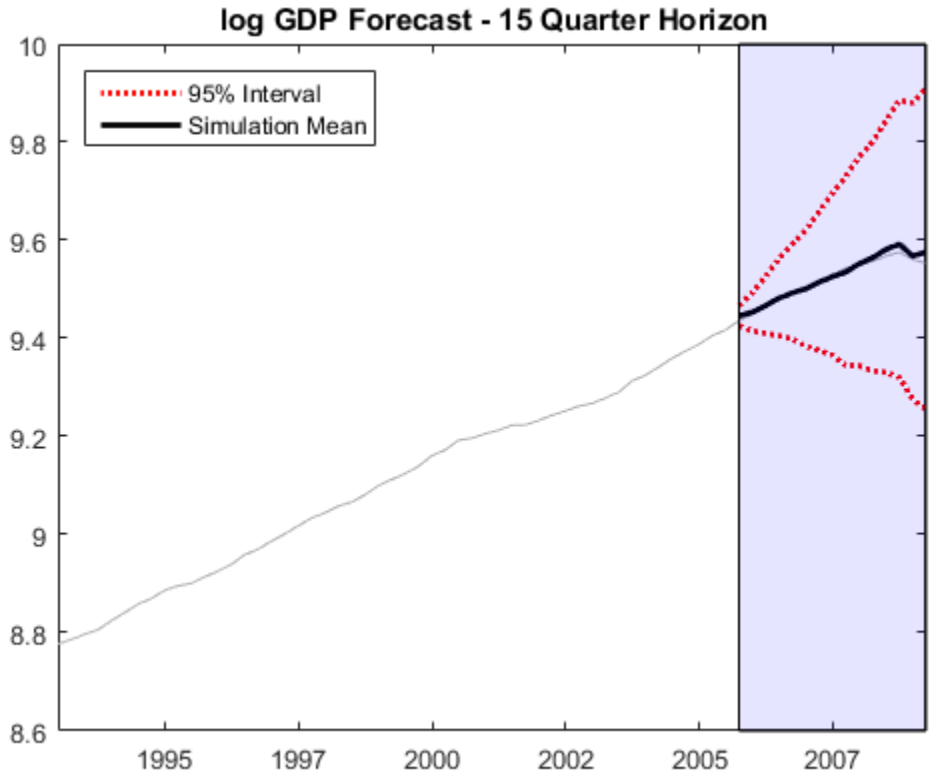
Simulate 1000 paths with 15 observations each. Use the inferred unconditional disturbances as presample data.

```
rng(1); % For reproducibility
GDPF = simulate(EstMdl,15,'NumPaths',1000,...
    'U0',u0,'X',hoCPI);
```

Plot the simulation mean forecast and approximate 95% forecast intervals.

```
lower = prctile(GDPF,2.5,2);
upper = prctile(GDPF,97.5,2);
mn = mean(GDPF,2);

figure
plot(dates(end-65:end),log(DataTable.GDP(end-65:end)),'Color',...
   [.7,.7,.7])
datetick
```

```
hold on
h1 = plot(dates(frstHzn),lower,'r:','LineWidth',2);
plot(dates(frstHzn),upper,'r:','LineWidth',2)
h2 = plot(dates(frstHzn),mn,'k','LineWidth',2);
legend([h1 h2],'95% Interval','Simulation Mean',...
   'Location','NorthWest')
h = gca;
ph = patch([repmat(dates(frstHzn(1)),1,2) repmat(dates(frstHzn(end)),1,2)],...
    [h.YLim fliplr(h.YLim)],...
    [0 0 0 0],'b');
ph.FaceAlpha = 0.1;
axis tight
title('{\bf log GDP Forecast - 15 Quarter Horizon}')
hold off
```



The unconditional disturbances, u_t, are nonstationary, therefore the widths of the forecast intervals grow with time.

- "Compare Alternative ARIMA Model Representations" on page 4-136
- "Simulate Stationary Processes" on page 5-151
- "Simulate Trend-Stationary and Difference-Stationary Processes" on page 5-163

References

[1] Box, G. E. P., G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis: Forecasting and Control*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1994.

[2] Davidson, R., and J. G. MacKinnon. *Econometric Theory and Methods*. Oxford, UK: Oxford University Press, 2004.

[3] Enders, W. *Applied Econometric Time Series*. Hoboken, NJ: John Wiley & Sons, Inc., 1995.

[4] Hamilton, J. D. *Time Series Analysis*. Princeton, NJ: Princeton University Press, 1994.

[5] Pankratz, A. *Forecasting with Dynamic Regression Models*. John Wiley & Sons, Inc., 1991.

[6] Tsay, R. S. *Analysis of Financial Time Series*. 2nd ed. Hoboken, NJ: John Wiley & Sons, Inc., 2005.

See Also

regARIMA | estimate | filter | forecast | infer

More About

- "Monte Carlo Simulation of Conditional Mean Models" on page 5-146
- "Presample Data for Conditional Mean Model Simulation" on page 5-149
- "Transient Effects in Conditional Mean Model Simulations" on page 5-150
- "Monte Carlo Forecasting of Conditional Mean Models" on page 5-181

simulate

Class: ssm

Monte Carlo simulation of state-space models

Syntax

```
[Y,X] = simulate(Mdl,numObs)
[Y,X] = simulate(Mdl,numObs,Name,Value)
[Y,X,U,E] = simulate(___)
```

Description

`[Y,X] = simulate(Mdl,numObs)` simulates one sample path of observations (`Y`) and states (`X`) from a fully specified, state-space model (`Mdl`). The software simulates `numObs` observations and states per sample path.

`[Y,X] = simulate(Mdl,numObs,Name,Value)` returns simulated responses and states with additional options specified by one or more `Name,Value` pair arguments.

For example, specify the number of paths or model parameter values.

`[Y,X,U,E] = simulate(___)` additionally simulate state disturbances (`U`) and observation innovations (`E`) using any of the input arguments in the previous syntaxes.

Tip

Simulate states from their joint conditional posterior distribution given the responses by using simsmooth.

Input Arguments

`Mdl` — Standard state-space model

`ssm` model object

Standard state-space model, specified as an`ssm` model object returned by ssm or estimate. A standard state-space model has finite initial state covariance matrix elements. That is, `Mdl` cannot be a dssm model object.

If `Mdl` is not fully specified (that is, `Mdl` contains unknown parameters), then specify values for the unknown parameters using the `'Params' Name,Value` pair argument. Otherwise, the software throws an error.

`numObs` — Number of periods per path to simulate
positive integer

Number of periods per path to generate variants, specified as a positive integer.

If `Mdl` is a time-varying model, then the length of the cell vector corresponding to the coefficient matrices must be at least `numObs`.

If `numObs` is fewer than the number of periods that `Mdl` can support, then the software only uses the matrices in the first `numObs` cells of the cell vectors corresponding to the coefficient matrices.

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'NumPaths'` — Number of sample paths to generate variants
1 (default) | positive integer

Number of sample paths to generate variants, specified as the comma-separated pair consisting of `'NumPaths'` and a positive integer.

Example: `'NumPaths',1000`

Data Types: `double`

`'Params'` — Values for unknown parameters
numeric vector

Values for unknown parameters in the state-space model, specified as the column-separated pair consisting of `'Params'` and a numeric vector.

The elements of `Params` correspond to the unknown parameters in the state-space model matrices `A`, `B`, `C`, and `D`, and, optionally, the initial state mean `Mean0` and covariance matrix `Cov0`.

- If you created `Mdl` explicitly (that is, by specifying the matrices without a parameter-to-matrix mapping function), then the software maps the elements of `Params` to `NaN`s in the state-space model matrices and initial state values. The software searches for `NaN`s column-wise following the order `A`, `B`, `C`, `D`, `Mean0`, and `Cov0`.
- If you created `Mdl` implicitly (that is, by specifying the matrices with a parameter-to-matrix mapping function), then you must set initial parameter values for the state-space model matrices, initial state values, and state types within the parameter-to-matrix mapping function.

If `Mdl` contains unknown parameters, then you must specify their values. Otherwise, the software ignores the value of `Params`.

Data Types: `double`

Output Arguments

`Y` — Simulated observations

matrix | cell matrix of numeric vectors

Simulated observations, returned as a matrix or cell matrix of numeric vectors.

If `Mdl` is a time-invariant model with respect to the observations, then `Y` is a `numObs`-by-n-by-`numPaths` array. That is, each row corresponds to a period, each column corresponds to an observation in the model, and each page corresponds to a sample path. The last row corresponds to the latest simulated observations.

If `Mdl` is a time-varying model with respect to the observations, then `Y` is a `numObs`-by-`numPaths` cell matrix of vectors. `Y{t,j}` contains a vector of length n_t of simulated observations for period t of sample path j. The last row of `Y` contains the latest set of simulated observations.

Data Types: `cell` | `double`

`X` — Simulated states

numeric matrix | cell matrix of numeric vectors

Simulated states, returned as a numeric matrix or cell matrix of vectors.

If `Mdl` is a time-invariant model with respect to the states, then `X` is a `numObs`-by-m-by-`numPaths` array. That is, each row corresponds to a period, each column corresponds to a state in the model, and each page corresponds to a sample path. The last row corresponds to the latest simulated states.

If `Mdl` is a time-varying model with respect to the states, then `X` is a `numObs`-by-`numPaths` cell matrix of vectors. `X{t,j}` contains a vector of length m_t of simulated states for period t of sample path j. The last row of `X` contains the latest set of simulated states.

`U` — Simulated state disturbances

matrix | cell matrix of numeric vectors

Simulated state disturbances, returned as a matrix or cell matrix of vectors.

If `Mdl` is a time-invariant model with respect to the state disturbances, then `U` is a `numObs`-by-h-by-`numPaths` array. That is, each row corresponds to a period, each column corresponds to a state disturbance in the model, and each page corresponds to a sample path. The last row corresponds to the latest simulated state disturbances.

If `Mdl` is a time-varying model with respect to the state disturbances, then `U` is a `numObs`-by-`numPaths` cell matrix of vectors. `U{t,j}` contains a vector of length h_t of simulated state disturbances for period t of sample path j. The last row of `U` contains the latest set of simulated state disturbances.

Data Types: `cell` | `double`

`E` — Simulated observation innovations

matrix | cell matrix of numeric vectors

Simulated observation innovations, returned as a matrix or cell matrix of numeric vectors.

If `Mdl` is a time-invariant model with respect to the observation innovations, then `E` is a `numObs`-by-h-by-`numPaths` array. That is, each row corresponds to a period, each column corresponds to an observation innovation in the model, and each page corresponds to a sample path. The last row corresponds to the latest simulated observation innovations.

If `Mdl` is a time-varying model with respect to the observation innovations, then `E` is a `numObs`-by-`numPaths` cell matrix of vectors. `E{t,j}` contains a vector of length h_t of simulated observation innovations for period t of sample path j. The last row of `E` contains the latest set of simulated observations.

Data Types: `cell` | `double`

Examples

Simulate States and Observations of Time-Invariant State-Space Model

Suppose that a latent process is an AR(1) model. Subsequently, the state equation is

$$x_t = 0.5x_{t-1} + u_t,$$

where u_t is Gaussian with mean 0 and standard deviation 1.

Generate a random series of 100 observations from x_t, assuming that the series starts at 1.5.

```
T = 100;
ARMdl = arima('AR',0.5,'Constant',0,'Variance',1);
x0 = 1.5;
rng(1); % For reproducibility
x = simulate(ARMdl,T,'Y0',x0);
```

Suppose further that the latent process is subject to additive measurement error. Subsequently, the observation equation is

$$y_t = x_t + \varepsilon_t,$$

where ε_t is Gaussian with mean 0 and standard deviation 0.75. Together, the latent process and observation equations compose a state-space model.

Use the random latent state process (`x`) and the observation equation to generate observations.

```
y = x + 0.75*randn(T,1);
```

Specify the four coefficient matrices.

```
A = 0.5;
B = 1;
C = 1;
D = 0.75;
```

Specify the state-space model using the coefficient matrices.

```
Mdl = ssm(A,B,C,D)


Mdl =


State-space model type: <a href="matlab: doc ssm">ssm</a>

State vector length: 1
Observation vector length: 1
State disturbance vector length: 1
Observation innovation vector length: 1
Sample size supported by model: Unlimited

State variables: x1, x2,...
State disturbances: u1, u2,...
Observation series: y1, y2,...
Observation innovations: e1, e2,...

State equation:
x1(t) = (0.50)x1(t-1) + u1(t)

Observation equation:
y1(t) = x1(t) + (0.75)e1(t)

Initial state distribution:

Initial state means
 x1
  0

Initial state covariance matrix
     x1
 x1  1.33

State types
     x1
 Stationary
```

`Mdl` is an `ssm` model. Verify that the model is correctly specified using the display in the Command Window. The software infers that the state process is stationary. Subsequently, the software sets the initial state mean and covariance to the mean and variance of the stationary distribution of an AR(1) model.

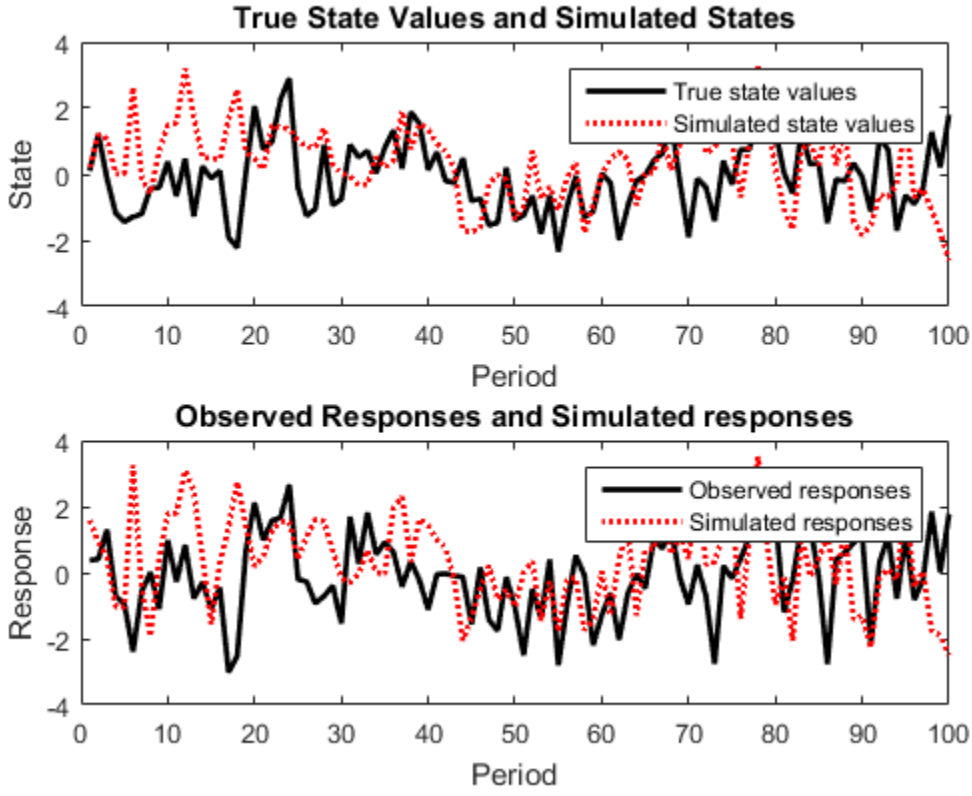
Simulate one path each of states and observations. Specify that the paths span 100 periods.

```
[simY,simX] = simulate(Mdl,100);
```

`simY` is a 100-by-1 vector of simulated responses. `simX` is a 100-by-1 vector of simulated states.

Plot the true state values with the simulated states. Also, plot the observed responses with the simulated responses.

```
figure
subplot(2,1,1)
plot(1:T,x,'-k',1:T,simX,':r','LineWidth',2)
title({'True State Values and Simulated States'})
xlabel('Period')
ylabel('State')
legend({'True state values','Simulated state values'})
subplot(2,1,2)
plot(1:T,y,'-k',1:T,simY,':r','LineWidth',2)
title({'Observed Responses and Simulated responses'})
xlabel('Period')
ylabel('Response')
legend({'Observed responses','Simulated responses'})
```



By default, `simulate` simulates one path for each state and observation in the state-space model. To conduct a Monte Carlo study, specify to simulate a large number of paths.

Simulate State-Space Models Containing Unknown Parameters

To generate variates from a state-space model, specify values for all unknown parameters.

Explicitly create the this state-space model.

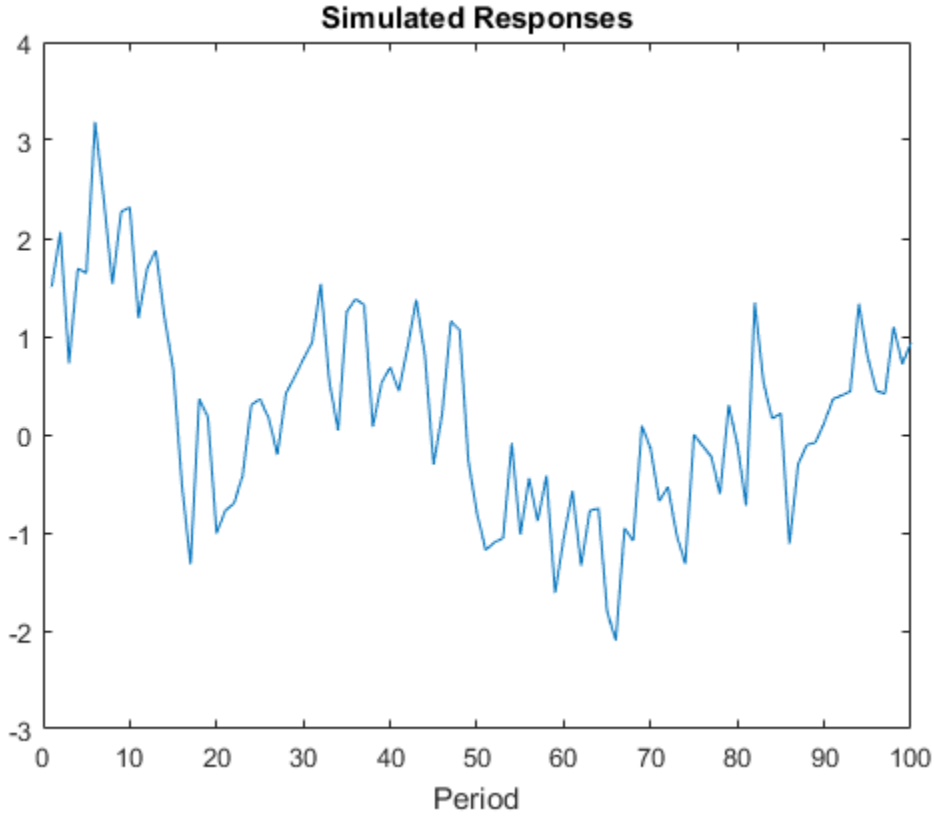
$$x_t = \phi x_{t-1} + \sigma_1 u_t$$
$$y_t = x_t + \sigma_2 \varepsilon_t$$

where u_t and ε_t are independent Gaussian random variables with mean 0 and variance 1. Suppose that the initial state mean and variance are 1, and that the state is a stationary process.

```
A = NaN;
B = NaN;
C = 1;
D = NaN;
mean0 = 1;
cov0 = 1;
stateType = 0;

Mdl = ssm(A,B,C,D,'Mean0',mean0,'Cov0',cov0,'StateType',stateType);
```

Simulate 100 responses from `Mdl`. Specify that the autoregressive coefficient is 0.75, the state disturbance standard deviation is 0.5, and the observation innovation standard deviation is 0.25.

```
params = [0.75 0.5 0.25];
y = simulate(Mdl,100,'Params',params);

figure;
plot(y);
title 'Simulated Responses';
xlabel 'Period';
```



The software searches for `NaN` values column-wise following the order A, B, C, D, Mean0, and Cov0. The order of the elements in `params` should correspond to this search.

Estimate Monte-Carlo Forecasts of State-Space Model

Suppose that the relationship between the change in the unemployment rate ($x_{1,t}$) and the nominal gross national product (nGNP) growth rate ($x_{3,t}$) can be expressed in the following, state-space model form.

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \\ x_{3,t} \\ x_{4,t} \end{bmatrix} = \begin{bmatrix} \phi_1 & \theta_1 & \gamma_1 & 0 \\ 0 & 0 & 0 & 0 \\ \gamma_2 & 0 & \phi_2 & \theta_2 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \\ x_{3,t-1} \\ x_{4,t-1} \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} u_{1,t} \\ u_{2,t} \end{bmatrix}$$

$$\begin{bmatrix} y_{1,t} \\ y_{2,t} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t} \\ x_{2,t} \\ x_{3,t} \\ x_{4,t} \end{bmatrix} + \begin{bmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{bmatrix} \begin{bmatrix} \varepsilon_{1,t} \\ \varepsilon_{2,t} \end{bmatrix},$$

where:

- $x_{1,t}$ is the change in the unemployment rate at time t.
- $x_{2,t}$ is a dummy state for the MA(1) effect on $x_{1,t}$.
- $x_{3,t}$ is the nGNP growth rate at time t.
- $x_{4,t}$ is a dummy state for the MA(1) effect on $x_{3,t}$.
- $y_{1,t}$ is the observed change in the unemployment rate.
- $y_{2,t}$ is the observed nGNP growth rate.
- $u_{1,t}$ and $u_{2,t}$ are Gaussian series of state disturbances having mean 0 and standard deviation 1.
- $\varepsilon_{1,t}$ is the Gaussian series of observation innovations having mean 0 and standard deviation σ_1.
- $\varepsilon_{2,t}$ is the Gaussian series of observation innovations having mean 0 and standard deviation σ_2.

Load the Nelson-Plosser data set, which contains the unemployment rate and nGNP series, among other things.

```
load Data_NelsonPlosser
```

Preprocess the data by taking the natural logarithm of the nGNP series, and the first difference of each. Also, remove the starting `NaN` values from each series.

```
isNaN = any(ismissing(DataTable),2);       % Flag periods containing NaNs
gnpn = DataTable.GNPN(~isNaN);
u = DataTable.UR(~isNaN);
```

```
T = size(gnpn,1);                           % Sample size
y = zeros(T-1,2);                           % Preallocate
y(:,1) = diff(u);
y(:,2) = diff(log(gnpn));
```

This example proceeds using series without `NaN` values. However, using the Kalman filter framework, the software can accommodate series containing missing values.

To determine how well the model forecasts observations, remove the last 10 observations for comparison.

```
numPeriods = 10;                    % Forecast horizon
isY = y(1:end-numPeriods,:);        % In-sample observations
oosY = y(end-numPeriods+1:end,:);  % Out-of-sample observations
```

Specify the coefficient matrices.

```
A = [NaN NaN NaN 0; 0 0 0 0; NaN 0 NaN NaN; 0 0 0 0];
B = [1 0;1 0 ; 0 1; 0 1];
C = [1 0 0 0; 0 0 1 0];
D = [NaN 0; 0 NaN];
```

Specify the state-space model using `ssm`. Verify that the model specification is consistent with the state-space model.

```
Mdl = ssm(A,B,C,D)
```

```
Mdl =


State-space model type: <a href="matlab: doc ssm">ssm</a>

State vector length: 4
Observation vector length: 2
State disturbance vector length: 2
Observation innovation vector length: 2
Sample size supported by model: Unlimited
Unknown parameters for estimation: 8

State variables: x1, x2,...
State disturbances: u1, u2,...
Observation series: y1, y2,...
Observation innovations: e1, e2,...
```

```
Unknown parameters: c1, c2,...

State equations:
x1(t) = (c1)x1(t-1) + (c3)x2(t-1) + (c4)x3(t-1) + u1(t)
x2(t) = u1(t)
x3(t) = (c2)x1(t-1) + (c5)x3(t-1) + (c6)x4(t-1) + u2(t)
x4(t) = u2(t)

Observation equations:
y1(t) = x1(t) + (c7)e1(t)
y2(t) = x3(t) + (c8)e2(t)

Initial state distribution:

Initial state means are not specified.
Initial state covariance matrix is not specified.
State types are not specified.
```

Estimate the model parameters, and use a random set of initial parameter values for optimization. Restrict the estimate of σ_1 and σ_2 to all positive, real numbers using the `'lb'` name-value pair argument. For numerical stability, specify the Hessian when the software computes the parameter covariance matrix, using the `'CovMethod'` name-value pair argument.

```
rng(1);
params0 = rand(8,1);
[EstMdl,estParams] = estimate(Mdl,isY,params0,...
    'lb',[-Inf -Inf -Inf -Inf -Inf -Inf 0 0],'CovMethod','hessian');
```

```
Method: Maximum likelihood (fmincon)
Sample size: 51
Logarithmic  likelihood:      -170.92
Akaike   info criterion:       357.84
Bayesian info criterion:      373.295
      |     Coeff       Std Err    t Stat     Prob
-------------------------------------------------------
 c(1) |  0.06750       0.16548     0.40791  0.68334
 c(2) | -0.01372       0.05887    -0.23302  0.81575
 c(3) |  2.71201       0.27039    10.03006   0
 c(4) |  0.83816       2.84586     0.29452  0.76836
 c(5) |  0.06273       2.83471     0.02213  0.98234
 c(6) |  0.05197       2.56873     0.02023  0.98386
```

```
 c(7) |  0.00272       2.40764     0.00113  0.99910
 c(8) |  0.00016       0.13942     0.00113  0.99910
      |
      |   Final State   Std Dev      t Stat    Prob
 x(1) | -0.00000       0.00272    -0.00033  0.99973
 x(2) |  0.12237       0.92954     0.13164  0.89527
 x(3) |  0.04049       0.00016   256.67560   0
 x(4) |  0.01183       0.00016    72.49713   0
```

`EstMdl` is an `ssm` model, and you can access its properties using dot notation.

Filter the estimated, state-space model, and extract the filtered states and their variances from the final period.

```
[~,~,Output] = filter(EstMdl,isY);
```

Modify the estimated, state-space model so that the initial state means and covariances are the filtered states and their covariances of the final period. This sets up simulation over the forecast horizon.

```
EstMdl1 = EstMdl;
EstMdl1.Mean0 = Output(end).FilteredStates;
EstMdl1.Cov0 = Output(end).FilteredStatesCov;
```

Simulate `5e5` paths of observations from the fitted, state-space model `EstMdl`. Specify to simulate observations for each period.

```
numPaths = 5e5;
SimY = simulate(EstMdl1,10,'NumPaths',numPaths);
```

`SimY` is a `10`-by- `2`-by- `numPaths` array containing the simulated observations. The rows of `SimY` correspond to periods, the columns correspond to an observation in the model, and the pages correspond to paths.

Estimate the forecasted observations and their 95% confidence intervals in the forecast horizon.

```
MCFY = mean(SimY,3);
CIFY = quantile(SimY,[0.025 0.975],3);
```

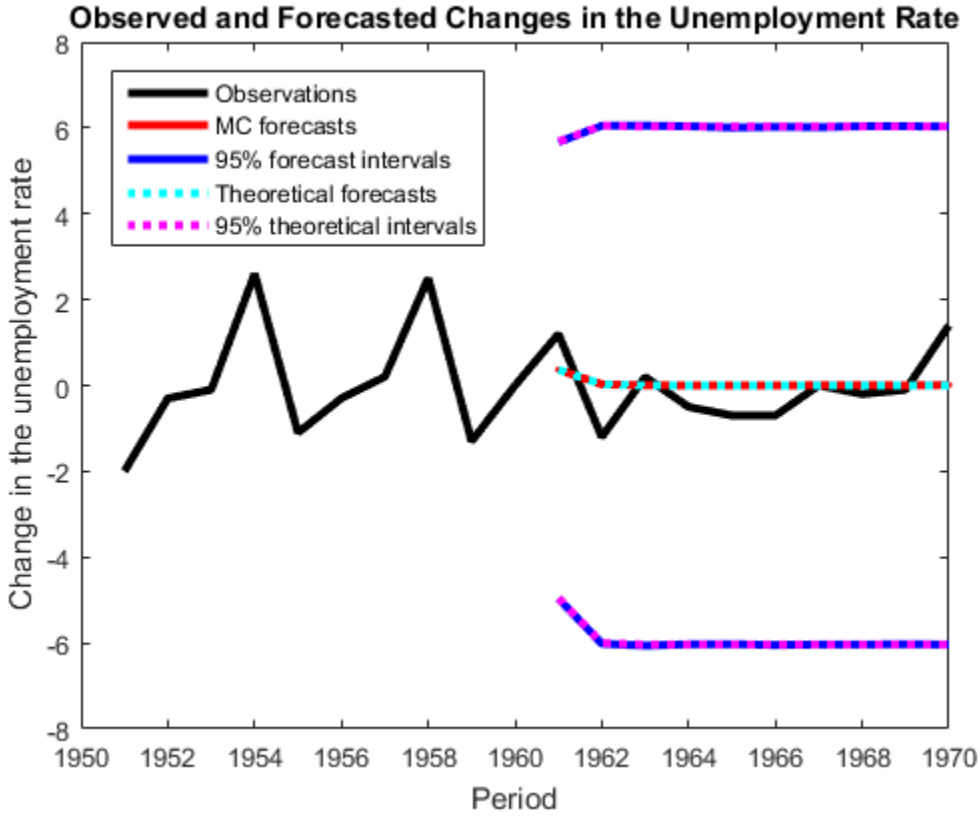
Estimate the theoretical forecast bands.

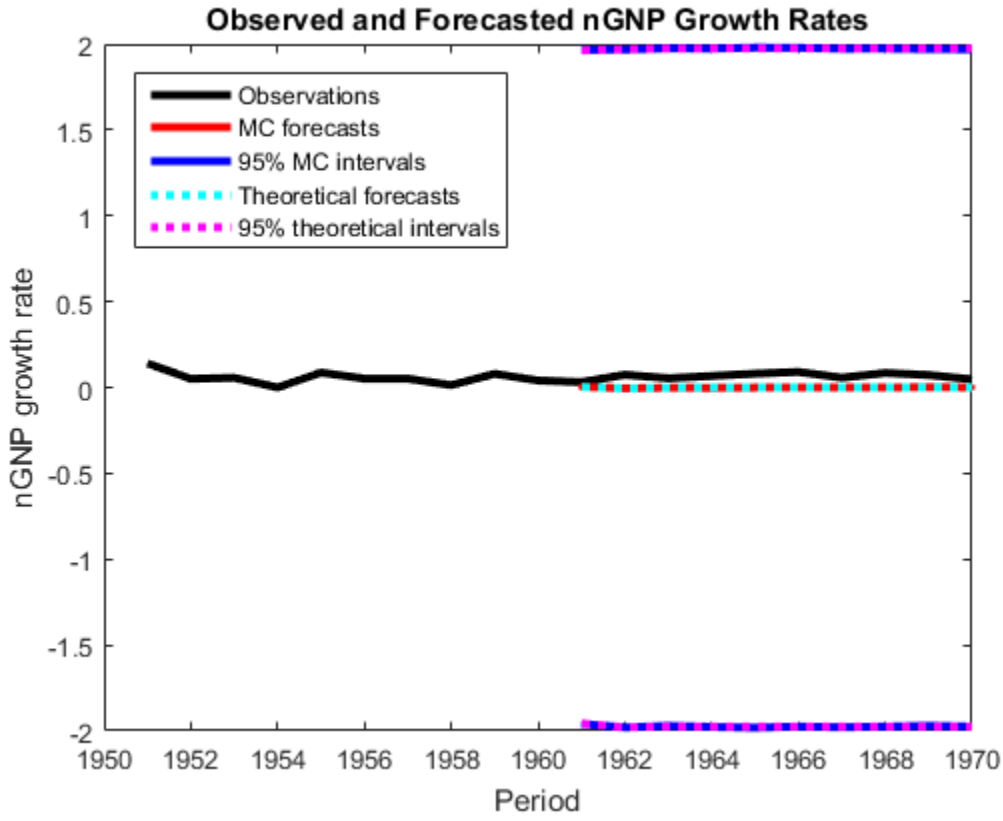
```
[Y,YMSE] = forecast(EstMdl,10,isY);
```

```
Lb = Y - sqrt(YMSE)*1.96;
Ub = Y + sqrt(YMSE)*1.96;
```

Plot the forecasted observations with their true values and the forecast intervals.

```
figure
h = plot(dates(end-numPeriods-9:end),[isY(end-9:end,1);oosY(:,1)],'-k',...
    dates(end-numPeriods+1:end),MCFY(end-numPeriods+1:end,1),'.-r',...
    dates(end-numPeriods+1:end),CIFY(end-numPeriods+1:end,1,1),'-b',...
    dates(end-numPeriods+1:end),CIFY(end-numPeriods+1:end,1,2),'-b',...
    dates(end-numPeriods+1:end),Y(:,1),':c',...
    dates(end-numPeriods+1:end),Lb(:,1),':m',...
    dates(end-numPeriods+1:end),Ub(:,1),':m',...
    'LineWidth',3);
xlabel('Period')
ylabel('Change in the unemployment rate')
legend(h([1,2,4:6]),{'Observations','MC forecasts',...
    '95% forecast intervals','Theoretical forecasts',...
    '95% theoretical intervals'},'Location','Best')
title('Observed and Forecasted Changes in the Unemployment Rate')

figure
h = plot(dates(end-numPeriods-9:end),[isY(end-9:end,2);oosY(:,2)],'-k',...
    dates(end-numPeriods+1:end),MCFY(end-numPeriods+1:end,2),'.-r',...
    dates(end-numPeriods+1:end),CIFY(end-numPeriods+1:end,2,1),'-b',...
    dates(end-numPeriods+1:end),CIFY(end-numPeriods+1:end,2,2),'-b',...
    dates(end-numPeriods+1:end),Y(:,2),':c',...
    dates(end-numPeriods+1:end),Lb(:,2),':m',...
    dates(end-numPeriods+1:end),Ub(:,2),':m',...
    'LineWidth',3);
xlabel('Period')
ylabel('nGNP growth rate')
legend(h([1,2,4:6]),{'Observations','MC forecasts',...
    '95% MC intervals','Theoretical forecasts','95% theoretical intervals'},...
    'Location','Best')
title('Observed and Forecasted nGNP Growth Rates')
```


Observed and Forecasted Changes in the Unemployment Rate
Observations
MC forecasts
95% forecast intervals
Theoretical forecasts
95% theoretical intervals
Change in the unemployment rate
Period
8
6
4
2
0
-2
-4
-6
-8
1950
1952
1954
1956
1958
1960
1962
1964
1966
1968
1970



- "Simulate States and Observations of Time-Invariant State-Space Model" on page 8-103
- "Simulate Time-Varying State-Space Model" on page 8-107
- "Forecast State-Space Model Using Monte-Carlo Methods" on page 8-125
- "Estimate Random Parameter of State-Space Model" on page 8-116

References

[1] Durbin J., and S. J. Koopman. *Time Series Analysis by State Space Methods*. 2nd ed. Oxford: Oxford University Press, 2012.

See Also

estimate | filter | forecast | simsmooth | smooth | ssm

More About

- “What Are State-Space Models?” on page 8-3

smooth

Class: dssm

Backward recursion of diffuse state-space models

Syntax

```
X = smooth(Mdl,Y)
X = smooth(Mdl,Y,Name,Value)
[X,logL,Output] = smooth( ___ )
```

Description

`X = smooth(Mdl,Y)` returns smoothed states (`X`) by performing backward recursion of the fully-specified diffuse state-space model `Mdl`. That is, `smooth` applies the diffuse Kalman filter using `Mdl` and the observed responses `Y`.

`X = smooth(Mdl,Y,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments. For example, specify the regression coefficients and predictor data to deflate the observations, or specify to use the univariate treatment of a multivariate model.

If `Mdl` is not fully specified, then you must specify the unknown parameters as known scalars using the `'Params' Name,Value` pair argument.

`[X,logL,Output] = smooth( ___ )` uses any of the input arguments in the previous syntaxes to additionally return the loglikelihood value (`logL`) and an output structure array (`Output`) using any of the input arguments in the previous syntaxes. The fields of `Output` include:

- Smoothed states and their estimated covariance matrix
- Smoothed state disturbances and their estimated covariance matrix
- Smoothed observation innovations and their estimated covariance matrix
- The loglikelihood value

- The adjusted Kalman gain
- And a vector indicating which data the software used to filter

Tips

- `Mdl` does not store the response data, predictor data, and the regression coefficients. Supply the data wherever necessary using the appropriate input or name-value pair arguments.
- It is a best practice to allow `dssm.smooth` to determine the value of `SwitchTime`. However, in rare cases, you might experience numerical issues during estimation, filtering, or smoothing diffuse state-space models. For such cases, try experimenting with various `SwitchTime` specifications, or consider a different model structure (e.g., simplify or reverify the model). For example, convert the diffuse state-space model to a standard state-space model using ssm.
- To accelerate estimation for low-dimensional, time-invariant models, set `'Univariate',true`. Using this specification, the software sequentially updates rather then updating all at once during the filtering process.

Input Arguments

`Mdl` — Diffuse state-space model

`dssm` model object

Diffuse state-space model, specified as an `dssm` model object returned by dssm or estimate.

If `Mdl` is not fully specified (that is, `Mdl` contains unknown parameters), then specify values for the unknown parameters using the `'Params'` name-value pair argument. Otherwise, the software issues an error. `estimate` returns fully-specified state-space models.

`Mdl` does not store observed responses or predictor data. Supply the data wherever necessary using the appropriate input or name-value pair arguments.

`Y` — Observed response data

numeric matrix | cell vector of numeric vectors

Observed response data to which `Mdl` is fit, specified as a numeric matrix or a cell vector of numeric vectors.

- If `Mdl` is time invariant with respect to the observation equation, then `Y` is a T-by-n matrix, where each row corresponds to a period and each column corresponds to a particular observation in the model. T is the sample size and m is the number of observations per period. The last row of `Y` contains the latest observations.
- If `Mdl` is time varying with respect to the observation equation, then `Y` is a T-by-1 cell vector. Each element of the cell vector corresponds to a period and contains an n_t-dimensional vector of observations for that period. The corresponding dimensions of the coefficient matrices in `Mdl.C{t}` and `Mdl.D{t}` must be consistent with the matrix in `Y{t}` for all periods. The last cell of `Y` contains the latest observations.

`NaN` elements indicate missing observations. For details on how the Kalman filter accommodates missing observations, see "Algorithms" on page 9-450.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'Beta'` — Regression coefficients

[ ] (default) | numeric matrix

Regression coefficients corresponding to predictor variables, specified as the comma-separated pair consisting of `'Beta'` and a d-by-n numeric matrix. d is the number of predictor variables (see `Predictors`) and n is the number of observed response series (see `Y`).

If `Mdl` is an estimated state-space model, then specify the estimated regression coefficients stored in `estParams`.

`'Params'` — Values for unknown parameters

numeric vector

Values for unknown parameters in the state-space model, specified as the column-separated pair consisting of `'Params'` and a numeric vector.

The elements of `Params` correspond to the unknown parameters in the state-space model matrices `A`, `B`, `C`, and `D`, and, optionally, the initial state mean `Mean0` and covariance matrix `Cov0`.

- If you created `Mdl` explicitly (that is, by specifying the matrices without a parameter-to-matrix mapping function), then the software maps the elements of `Params` to `NaN`s in the state-space model matrices and initial state values. The software searches for `NaN`s column-wise following the order `A`, `B`, `C`, `D`, `Mean0`, and `Cov0`.
- If you created `Mdl` implicitly (that is, by specifying the matrices with a parameter-to-matrix mapping function), then you must set initial parameter values for the state-space model matrices, initial state values, and state types within the parameter-to-matrix mapping function.

If `Mdl` contains unknown parameters, then you must specify their values. Otherwise, the software ignores the value of `Params`.

Data Types: `double`

`'Predictors'` — Predictor variables in state-space model observation equation

`[ ]` (default) | numeric matrix

Predictor variables in the state-space model observation equation, specified as the comma-separated pair consisting of `'Predictors'` and a T-by-d numeric matrix. T is the number of periods and d is the number of predictor variables. Row t corresponds to the observed predictors at period t (Z_t). The expanded observation equation is

$$y_t - Z_t\beta = Cx_t + Du_t.$$

That is, the software deflates the observations using the regression component. β is the time-invariant vector of regression coefficients that the software estimates with all other parameters.

If there are n observations per period, then the software regresses all predictor series onto each observation.

If you specify `Predictors`, then `Mdl` must be time invariant. Otherwise, the software returns an error.

By default, the software excludes a regression component from the state-space model.

Data Types: `double`

`'SwitchTime'` — Final period for diffuse state initialization

positive integer

Final period for diffuse state initialization, specified as the comma-separated pair consisting of `'SwitchTime'` and a positive integer. That is, `estimate` uses the observations from period 1 to period `SwitchTime` as a presample to implement the *exact initial Kalman filter* (see "Diffuse Kalman Filter" on page 8-15 and [1]). After initializing the diffuse states, `estimate` applies the standard Kalman filter to the observations from periods `SwitchTime` + 1 to T.

The default value for `SwitchTime` is the last period in which the estimated smoothed state precision matrix is singular (i.e., the inverse of the covariance matrix). This specification represents the fewest number of observations required to initialize the diffuse states. Therefore, it is a best practice to use the default value.

If you set `SwitchTime` to a value greater than the default, then the effective sample size decreases. If you set `SwitchTime` to a value that is fewer than the default, then `estimate` might not have enough observations to initialize the diffuse states, which can result in an error or improper values.

In general, estimating, filtering, and smoothing state-space models with at least one diffuse state requires `SwitchTime` to be at least one. The default estimation display contains the effective sample size.

Data Types: `double`

`'Tolerance'` — Forecast uncertainty threshold

`0` (default) | nonnegative scalar

Forecast uncertainty threshold, specified as the comma-separated pair consisting of `'Tolerance'` and a nonnegative scalar.

If the forecast uncertainty for a particular observation is less than `Tolerance` during numerical estimation, then the software removes the uncertainty corresponding to the observation from the forecast covariance matrix before its inversion.

It is best practice to set `Tolerance` to a small number, for example, `1e-15`, to overcome numerical obstacles during estimation.

Example: `'Tolerance',1e-15`

Data Types: `double`

`'Univariate'` — Univariate treatment of multivariate series flag

`false` (default) | `true`

Univariate treatment of a multivariate series flag, specified as the comma-separated pair consisting of `'Univariate'` and `true` or `false`. Univariate treatment of a multivariate series is also known as *sequential filtering*.

The univariate treatment can accelerate and improve numerical stability of the Kalman filter. However, all observation innovations must be uncorrelated. That is, D_tD_t' must be diagonal, where D_t, $t = 1,...,T$, is one of the following:

- The matrix `D{t}` in a time-varying state-space model
- The matrix `D` in a time-invariant state-space model

Example: `'Univariate',true`

Data Types: `logical`

Output Arguments

`X` — Smoothed states

numeric matrix | cell vector of vectors

Smoothed states, returned as a numeric matrix or a cell vector of matrices.

If `Mdl` is time invariant, then the number of rows of `X` is the sample size, and the number of columns of `X` is the number of states. The last row of `X` contains the latest, smoothed states.

If `Mdl` is time varying, then `X` is a cell vector with length equal to the sample size. Cell t of `X` contains a vector of smoothed states with length equal to the number of states in period t. The last cell of `X` contains the latest, smoothed states.

`smooth` pads the first `SwitchTime` periods of `X` with zeros or empty cells. The zeros or empty cells represent the periods required to initialize the diffuse states.

`logL` — Loglikelihood function value

scalar

Loglikelihood function value, returned as a scalar.

Missing observations and observations before `SwitchTime` do not contribute to the loglikelihood.

`Output` — Smoothing results by period

structure array

Smoothing results by period, returned as a structure array.

`Output` is a T-by-1 structure, where element t corresponds to the smoothing recursion at time t.

- If `Univariate` is `false` (it is by default), then the following table describes the fields of `Output`.

Field	Description	Estimate
`LogLikelihood`	Scalar loglikelihood objective function value	N/A
`SmoothedStates`	m_t-by-1 vector of smoothed states	$E(x_t \mid y_1,\ldots,y_T)$
`SmoothedStatesCov`	m_t-by-m_t variance-covariance matrix of the smoothed states	$Var(x_t \mid y_1,\ldots,y_T)$
`SmoothedStatesDisturb`	k_t-by-1 vector of smoothed, state disturbances	$E(u_t \mid y_1,\ldots,y_T)$
`SmoothedStateDisturbCo`	k_t-by-k_t variance-covariance matrix of the smoothed, state disturbances	$Var(u_t \mid y_1,\ldots,y_T)$
`SmoothedObsInnov`	h_t-by-1 vector of smoothed observation innovations	$E(\varepsilon_t \mid y_1,\ldots,y_T)$
`SmoothedObsInnovCov`	h_t-by-h_t variance-covariance matrix of the smoothed, observation innovations	$Var(\varepsilon_t \mid y_1,\ldots,y_T)$
`KalmanGain`	m_t-by-n_t adjusted Kalman gain matrix	N/A

Field	Description	Estimate
`DataUsed`	h_t-by-1 logical vector indicating whether the software filters using a particular observation. For example, if observation i at time t is a `NaN`, then element i in `DataUsed` at time t is `0`.	N/A

- If `Univariate` is `true`, then the fields of `Output` are the same as in the previous table, but the values in `KalmanGain` might vary.

`smooth` pads the first `SwitchTime` periods of the fields of `Output` with empty cells. These empty cells represent the periods required to initialize the diffuse states.

Data Types: `struct`

Examples

Smooth States of Time-Invariant Diffuse State-Space Model

Suppose that a latent process is a random walk. Subsequently, the state equation is

$$x_t = x_{t-1} + u_t,$$

where u_t is Gaussian with mean 0 and standard deviation 1.

Generate a random series of 100 observations from x_t, assuming that the series starts at 1.5.

```
T = 100;
x0 = 1.5;
rng(1); % For reproducibility
u = randn(T,1);
x = cumsum([x0;u]);
x = x(2:end);
```

Suppose further that the latent process is subject to additive measurement error. Subsequently, the observation equation is

$$y_t = x_t + \varepsilon_t,$$

where ε_t is Gaussian with mean 0 and standard deviation 0.75. Together, the latent process and observation equations compose a state-space model.

Use the random latent state process (`x`) and the observation equation to generate observations.

```
y = x + 0.75*randn(T,1);
```

Specify the four coefficient matrices.

```
A = 1;
B = 1;
C = 1;
D = 0.75;
```

Create the diffuse state-space model using the coefficient matrices. Specify that the inital state distribution is diffuse.

```
Mdl = dssm(A,B,C,D,'StateType',2)
```

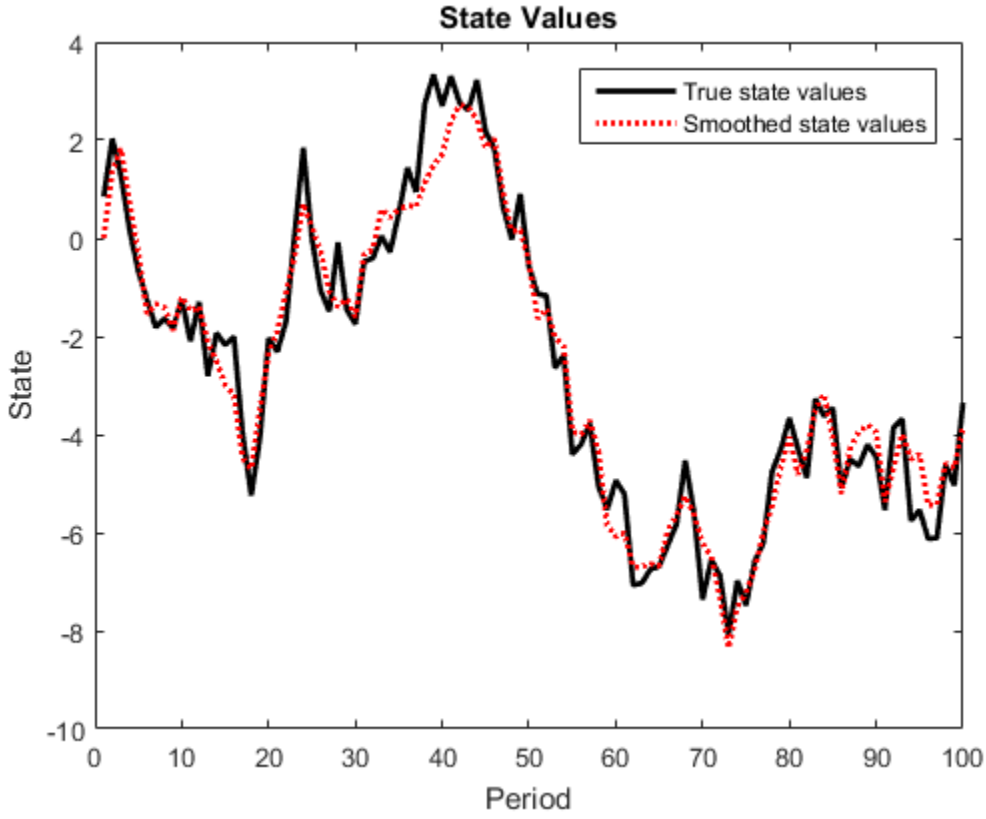
```
Mdl =


State-space model type: <a href="matlab: doc dssm">dssm</a>

State vector length: 1
Observation vector length: 1
State disturbance vector length: 1
Observation innovation vector length: 1
Sample size supported by model: Unlimited

State variables: x1, x2,...
State disturbances: u1, u2,...
Observation series: y1, y2,...
Observation innovations: e1, e2,...

State equation:
x1(t) = x1(t-1) + u1(t)

Observation equation:
y1(t) = x1(t) + (0.75)e1(t)
```

```
Initial state distribution:

Initial state means
 x1
  0

Initial state covariance matrix
     x1
 x1  Inf

State types
    x1
 Diffuse
```

`Mdl` is an `dssm` model. Verify that the model is correctly specified using the display in the Command Window.

Smooth states for periods 1 through 100. Plot the true state values and the smoothed state estimates.

```
SmoothedX = smooth(Mdl,y);

figure
plot(1:T,x,'-k',1:T,SmoothedX,':r','LineWidth',2)
title({'State Values'})
xlabel('Period')
ylabel('State')
legend({'True state values','Smoothed state values'})
```



The true values and smoothed estimates are approximately the same.

Smooth States of Diffuse State-Space Model Containing Regression Component

Suppose that the linear relationship between unemployment rate and the nominal gross national product (nGNP) is of interest. Suppose further that unemployment rate is an AR(1) series. Symbolically, and in state-space form, the model is

$$x_t = \phi x_{t-1} + \sigma u_t$$
$$y_t - \beta Z_t = x_t,$$

where:

- x_t is the unemployment rate at time t.

- y_t is the observed change in the unemployment rate being deflated by the return of nGNP (Z_t).
- u_t is the Gaussian series of state disturbances having mean 0 and unknown standard deviation σ.

Load the Nelson-Plosser data set, which contains the unemployment rate and nGNP series, among other things.

```
load Data_NelsonPlosser
```

Preprocess the data by taking the natural logarithm of the nGNP series, and removing the starting `NaN` values from each series.

```
isNaN = any(ismissing(DataTable),2);       % Flag periods containing NaNs
gnpn = DataTable.GNPN(~isNaN);
y = diff(DataTable.UR(~isNaN));
T = size(gnpn,1);                          % The sample size
Z = price2ret(gnpn);
```

This example continues using the series without `NaN` values. However, using the Kalman filter framework, the software can accommodate series containing missing values.

Specify the coefficient matrices.

```
A = NaN;
B = NaN;
C = 1;
```

Create the state-space model using `dssm` by supplying the coefficient matrices and specifying that the state values come from a diffuse distribution. The diffuse specification indicates complete ignorance about the moments of the initial distribution.

```
StateType = 2;
Mdl = dssm(A,B,C,'StateType',StateType);
```

Estimate the parameters. Specify the regression component and its initial value for optimization using the `'Predictors'` and `'Beta0'` name-value pair arguments, respectively. Display the estimates and all optimization diagnostic information. Restrict the estimate of σ to all positive, real numbers.

```
params0 = [0.3 0.2]; % Initial values chosen arbitrarily
Beta0 = 0.1;
```

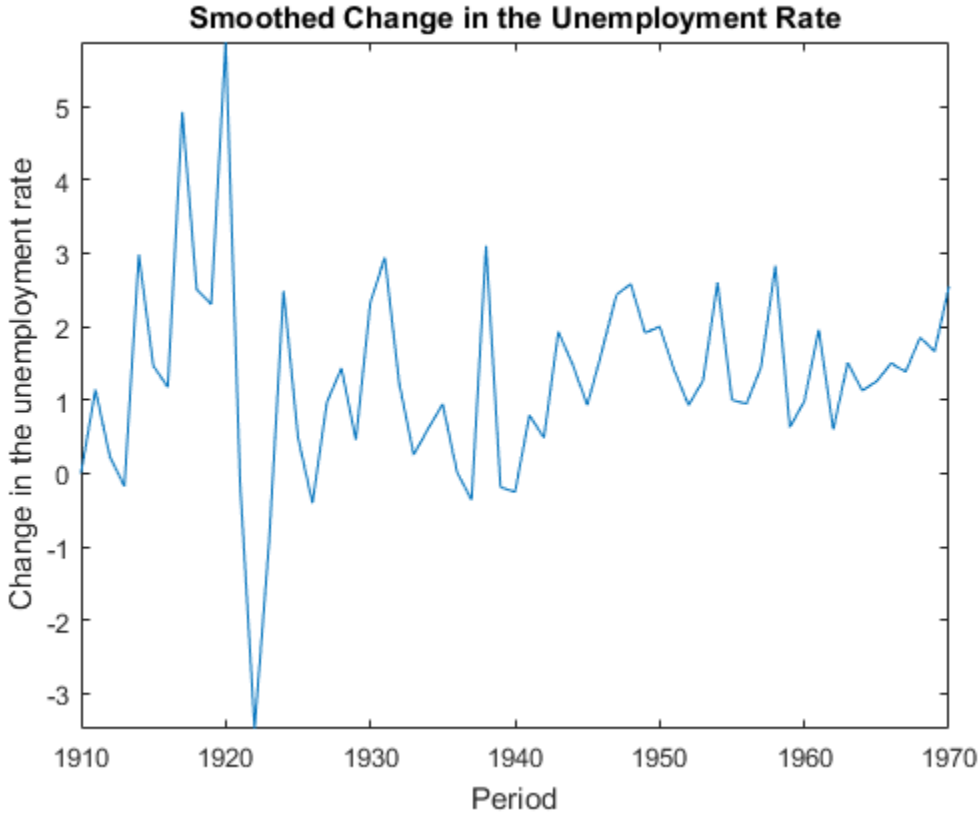
```
[EstMdl,estParams] = estimate(Mdl,y,params0,'Predictors',Z,'Beta0',Beta0,...
    'lb',[-Inf 0 -Inf]);
```

```
Method: Maximum likelihood (fmincon)
Effective Sample size:             60
Logarithmic  likelihood:     -110.477
Akaike   info criterion:      226.954
Bayesian info criterion:      233.287
           |      Coeff       Std Err    t Stat    Prob
-------------------------------------------------------
 c(1)      |   0.59436       0.09408     6.31738  0
 c(2)      |   1.52554       0.10758    14.17991  0
 y <- z(1) | -24.26161       1.55730   -15.57930  0
           |
           |    Final State   Std Dev     t Stat   Prob
 x(1)      |   2.54764        0           Inf     0
```

`EstMdl` is a `dssm` model, and you can access its properties using dot notation.

Smooth the estimated diffuse state-space model. `EstMdl` does not store the data or the regression coefficients, so you must pass in them in using the name-value pair arguments `'Predictors'` and `'Beta'`, respectively. Plot the smoothed states.

```
SmoothedX = smooth(EstMdl,y,'Predictors',Z,'Beta',estParams(end));

figure
plot(dates(end-(T-1)+1:end),SmoothedX);
xlabel('Period')
ylabel('Change in the unemployment rate')
title('Smoothed Change in the Unemployment Rate')
axis tight
```



Extract Other Estimates from `Output`

Estimate a diffuse state-space model, smooth the states, and then extract other estimates from the `Output` output argument.

Consider the diffuse state-space model

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix} = \begin{bmatrix} \phi & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \end{bmatrix} + \begin{bmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{bmatrix} \begin{bmatrix} u_{1,t} \\ u_{2,t} \end{bmatrix}$$
$$y_t = \begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix}$$

The state variable $x_{1,t}$ is an AR(1) model with autoregressive coefficient ϕ. $x_{2,t}$ is a random walk. The disturbances $u_{1,t}$ and $u_{2,t}$ are independent Gaussian random variables with mean 0 and standard deviations σ_1 and σ_2, respectively. The observation y_t is the error-free sum of $x_{1,t}$ and $x_{2,t}$.

Generate data from the state-space model. To simulate the data, suppose that the sample size $T = 100$, $\phi = 0.6$, $\sigma_1 = 0.2$, $\sigma_2 = 0.1$, and $x_{1,0} = x_{2,0} = 2$.

```
rng(1); % For reproducibility
T = 100;
ARMdl = arima('AR',0.6,'Constant',0,'Variance',0.2^2);
x1 = simulate(ARMdl,T,'Y0',2);
u3 = 0.1*randn(T,1);
x3 = cumsum([2;u3]);
x3 = x3(2:end);
y = x1 + x3;
```

Specify the coefficient matrices of the state-space model. To indicate unknown parameters, use `NaN` values.

```
A = [NaN 0; 0 1];
B = [NaN 0; 0 NaN];
C = [1 1];
```

Create a diffuse state-space model that describes the model above. Specify that $x_{1,t}$ and $x_{2,t}$ have diffuse initial state distributions.

```
StateType = [2 2];
Mdl = dssm(A,B,C,'StateType',StateType);
```

Estimate the unknown parameters of `Mdl`. Choose initial parameter values for optimization. Specify that the standard deviations are constrained to be positive, but all other parameters are unconstrained using the `'lb'` name-value pair argument.

```
params0 = [0.01 0.1 0.01]; % Initial values chosen arbitrarily
EstMdl = estimate(Mdl,y,params0,'lb',[-Inf 0 0]);
```

```
Method: Maximum likelihood (fmincon)
Effective Sample size:              98
Logarithmic  likelihood:       3.44283
Akaike   info criterion:     -0.885655
Bayesian info criterion:       6.92986
```

```
      |     Coeff      Std Err   t Stat     Prob
-------------------------------------------------
 c(1) | 0.54134       0.20494    2.64145  0.00826
 c(2) | 0.18439       0.03305    5.57897   0
 c(3) | 0.11783       0.04347    2.71039  0.00672
      |
      |  Final State   Std Dev    t Stat    Prob
 x(1) | 0.24884       0.17168    1.44943  0.14722
 x(2) | 1.73762       0.17168   10.12121   0
```

The parameters are close to their true values.

Smooth the states of `EstMdl`, and request all other available output.

```
[X,logL,Output] = smooth(EstMdl,y);
```

`X` is a `T`-by-2 matrix of smoothed states, `logL` is the final, optimized log-likelihood value, and `Output` is a structure array containing various estimates that the Kalman filter requires. List the fields of `output` using `fields`.

```
fields(Output)
```

```
ans =

    'LogLikelihood'
    'SmoothedStates'
    'SmoothedStatesCov'
    'SmoothedStateDisturb'
    'SmoothedStateDisturbCov'
    'SmoothedObsInnov'
    'SmoothedObsInnovCov'
    'KalmanGain'
    'DataUsed'
```

Convert `Output` to a table.

```
OutputTbl = struct2table(Output);
OutputTbl(1:10,1:4) % Display first ten rows of first four variables
```

```
ans =

    LogLikelihood    SmoothedStates    SmoothedStatesCov    SmoothedStateDisturb
```

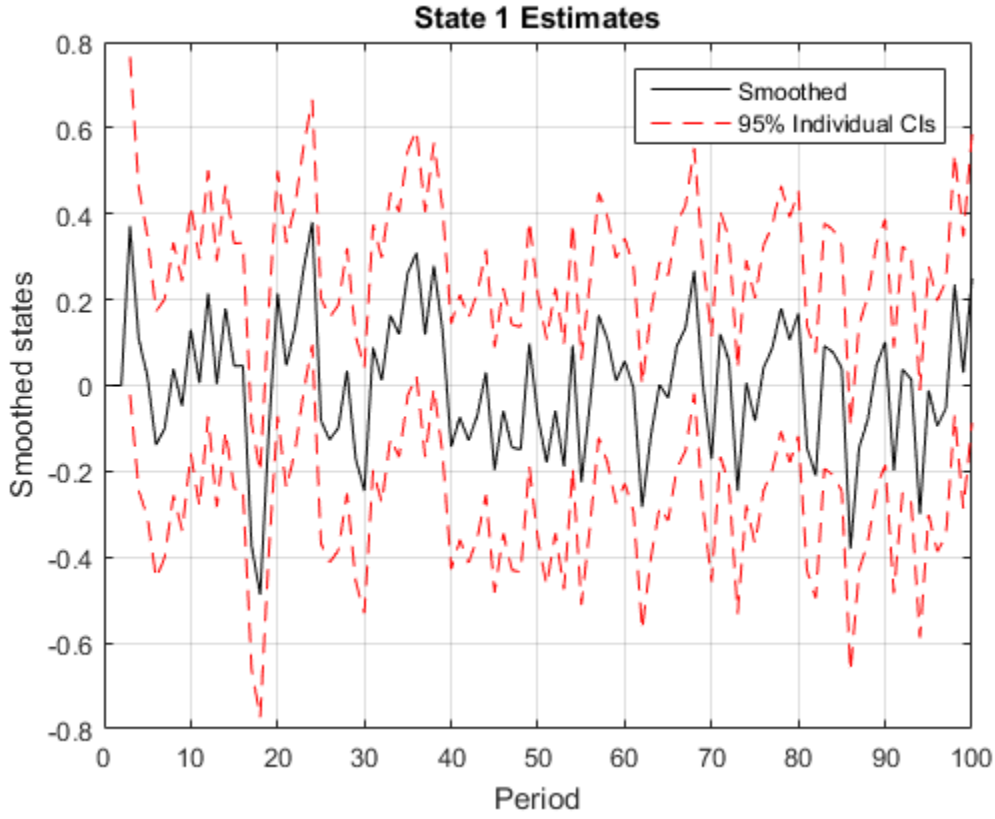
```
    ____________    _______________    _________________    ___________________

    []              []                 []                   []
    []              []                 []                   []
    [ 0.1827]       [2x1 double]       [2x2 double]         [2x1 double]
    [ 0.0972]       [2x1 double]       [2x2 double]         [2x1 double]
    [ 0.4472]       [2x1 double]       [2x2 double]         [2x1 double]
    [ 0.2073]       [2x1 double]       [2x2 double]         [2x1 double]
    [ 0.5167]       [2x1 double]       [2x2 double]         [2x1 double]
    [ 0.2389]       [2x1 double]       [2x2 double]         [2x1 double]
    [ 0.5064]       [2x1 double]       [2x2 double]         [2x1 double]
    [-0.0105]       [2x1 double]       [2x2 double]         [2x1 double]
```

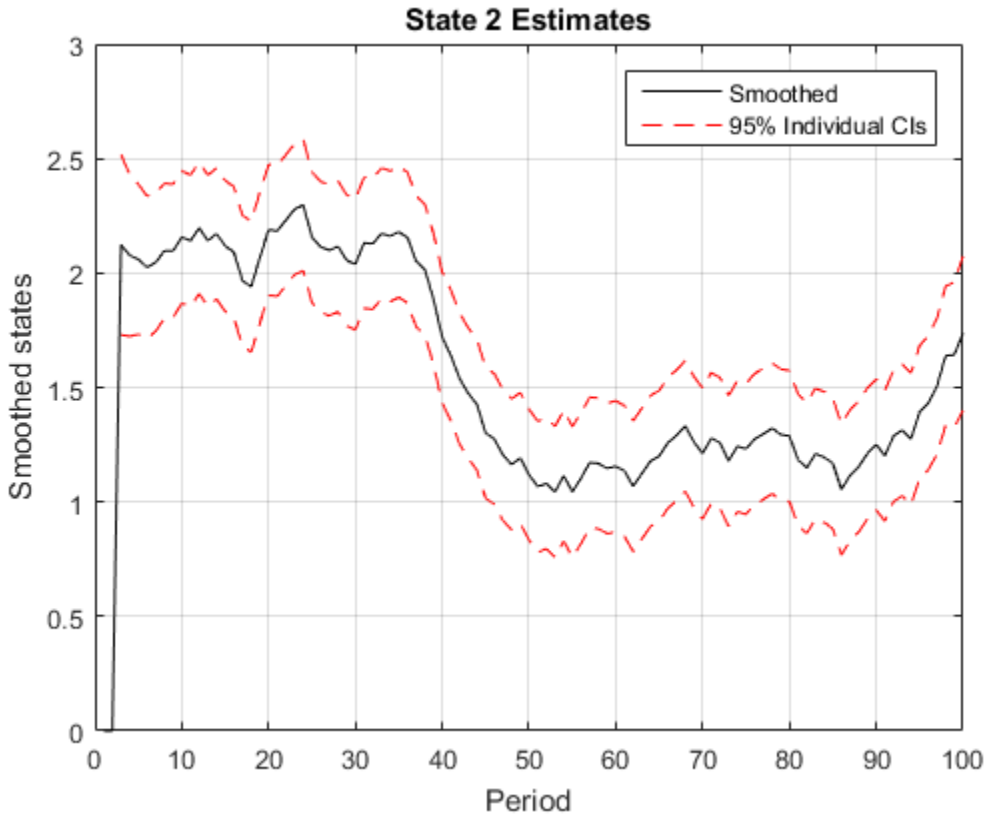
The first two rows of the table contain empty cells or zeros. These correspond to the observations required to initialize the diffuse Kalman filter. That is, `SwitchTime` is 2.

```
SwitchTime = 2;
```

Plot the smoothed states and their individual 95% Wald-type confidence intervals.

```
CI = nan(T,2,2);

for j = (SwitchTime + 1):T
    CovX = OutputTbl.SmoothedStatesCov{j};
    CI(j,:,1) = X(j,1) + 1.96*sqrt(CovX(1,1))*[-1 1];
    CI(j,:,2) = X(j,2) + 1.96*sqrt(CovX(2,2))*[-1 1];
end

figure;
plot(1:T,X(:,1),'k',1:T,CI(:,:,1),'--r');
xlabel('Period');
ylabel('Smoothed states');
title('State 1 Estimates')
legend('Smoothed','95% Individual CIs');
grid on;

figure;
plot(1:T,X(:,2),'k',1:T,CI(:,:,2),'--r');
xlabel('Period');
ylabel('Smoothed states');
title('State 2 Estimates')
legend('Smoothed','95% Individual CIs');
grid on;
```


State 1 Estimates
Smoothed
95% Individual CIs
Smoothed states
Period
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
0
10
20
30
40
50
60
70
80
90
100



- "Smooth Time-Varying Diffuse State-Space Model" on page 8-91
- "Filter Time-Varying Diffuse State-Space Model" on page 8-68

Algorithms

- The Kalman filter accommodates missing data by not updating filtered state estimates corresponding to missing observations. In other words, suppose there is a missing observation at period t. Then, the state forecast for period t based on the previous $t - 1$ observations and filtered state for period t are equivalent.

- For explicitly defined state-space models, `filter` applies all predictors to each response series. However, each response series has its own set of regression coefficients.
- The diffuse Kalman filter requires presample data. If missing observations begin the time series, then the diffuse Kalman filter must gather enough nonmissing observations to initialize the diffuse states.
- For diffuse state-space models, `filter` usually switches from the diffuse Kalman filter to the standard Kalman filter when the number of cumulative observations and the number of diffuse states are equal. However, if a diffuse state-space model has identifiability issues (e.g., the model is too complex to fit to the data), then `filter` might require more observations to initialize the diffuse states. In extreme cases, `filter` requires the entire sample.

References

[1] Durbin J., and S. J. Koopman. *Time Series Analysis by State Space Methods*. 2nd ed. Oxford: Oxford University Press, 2012.

See Also

dssm | estimate | filter | forecast | refine

More About

- "What Are State-Space Models?" on page 8-3
- "What Is the Kalman Filter?" on page 8-8

Introduced in R2015b

smooth

Class: ssm

Backward recursion of state-space models

Syntax

```
X = smooth(Mdl,Y)
X = smooth(Mdl,Y,Name,Value)
[X,logL,Output] = smooth( ___ )
```

Description

`X = smooth(Mdl,Y)` returns smoothed states (`X`) by performing backward recursion of the fully-specified state-space model `Mdl`. That is, `smooth` applies the standard Kalman filter using `Mdl` and the observed responses `Y`.

`X = smooth(Mdl,Y,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

If `Mdl` is not fully specified, then you must set the unknown parameters to known scalars using the `Params` `Name,Value` pair argument.

`[X,logL,Output] = smooth( ___ )` uses any of the input arguments in the previous syntaxes to additionally return the loglikelihood value (`logL`) and an output structure array (`Output`) containing:

- Smoothed states and their estimated covariance matrix
- Smoothed state disturbances and their estimated covariance matrix
- Smoothed observation innovations and their estimated covariance matrix
- The loglikelihood value
- The adjusted Kalman gain
- And a vector indicating which data the software used to filter

Input Arguments

`Mdl` — Standard state-space model

`ssm` model object

Standard state-space model, specified as an `ssm` model object returned by ssm or estimate.

If `Mdl` is not fully specified (that is, `Mdl` contains unknown parameters), then specify values for the unknown parameters using the `'Params'` name-value pair argument. Otherwise, the software issues an error. `estimate` returns fully-specified state-space models.

`Mdl` does not store observed responses or predictor data. Supply the data wherever necessary using the appropriate input or name-value pair arguments.

`Y` — Observed response data

numeric matrix | cell vector of numeric vectors

Observed response data to which `Mdl` is fit, specified as a numeric matrix or a cell vector of numeric vectors.

- If `Mdl` is time invariant with respect to the observation equation, then `Y` is a T-by-n matrix, where each row corresponds to a period and each column corresponds to a particular observation in the model. T is the sample size and m is the number of observations per period. The last row of `Y` contains the latest observations.
- If `Mdl` is time varying with respect to the observation equation, then `Y` is a T-by-1 cell vector. Each element of the cell vector corresponds to a period and contains an n_t-dimensional vector of observations for that period. The corresponding dimensions of the coefficient matrices in `Mdl.C{t}` and `Mdl.D{t}` must be consistent with the matrix in `Y{t}` for all periods. The last cell of `Y` contains the latest observations.

`NaN` elements indicate missing observations. For details on how the Kalman filter accommodates missing observations, see “Algorithms” on page 9-450.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

'Beta' — Regression coefficients

[ ] (default) | numeric matrix

Regression coefficients corresponding to predictor variables, specified as the comma-separated pair consisting of 'Beta' and a d-by-n numeric matrix. d is the number of predictor variables (see Predictors) and n is the number of observed response series (see Y).

If Mdl is an estimated state-space model, then specify the estimated regression coefficients stored in estParams.

'Params' — Values for unknown parameters

numeric vector

Values for unknown parameters in the state-space model, specified as the column-separated pair consisting of 'Params' and a numeric vector.

The elements of Params correspond to the unknown parameters in the state-space model matrices A, B, C, and D, and, optionally, the initial state mean Mean0 and covariance matrix Cov0.

- If you created Mdl explicitly (that is, by specifying the matrices without a parameter-to-matrix mapping function), then the software maps the elements of Params to NaNs in the state-space model matrices and initial state values. The software searches for NaNs column-wise following the order A, B, C, D, Mean0, and Cov0.
- If you created Mdl implicitly (that is, by specifying the matrices with a parameter-to-matrix mapping function), then you must set initial parameter values for the state-space model matrices, initial state values, and state types within the parameter-to-matrix mapping function.

If Mdl contains unknown parameters, then you must specify their values. Otherwise, the software ignores the value of Params.

Data Types: double

'Predictors' — Predictor variables in state-space model observation equation

[ ] (default) | numeric matrix

Predictor variables in the state-space model observation equation, specified as the comma-separated pair consisting of 'Predictors' and a T-by-d numeric matrix. T is the number of periods and d is the number of predictor variables. Row t corresponds to the observed predictors at period t (Z_t). The expanded observation equation is

$$y_t - Z_t\beta = Cx_t + Du_t.$$

That is, the software deflates the observations using the regression component. β is the time-invariant vector of regression coefficients that the software estimates with all other parameters.

If there are n observations per period, then the software regresses all predictor series onto each observation.

If you specify `Predictors`, then `Mdl` must be time invariant. Otherwise, the software returns an error.

By default, the software excludes a regression component from the state-space model.

Data Types: `double`

`'SquareRoot'` — Square root filter method flag

`false` (default) | `true`

Square root filter method flag, specified as the comma-separated pair consisting of `'SquareRoot'` and `true` or `false`. If `true`, then `estimate` applies the square root filter method when implementing the Kalman filter.

If you suspect that the eigenvalues of the filtered state or forecasted observation covariance matrices are close to zero, then specify `'SquareRoot',true`. The square root filter is robust to numerical issues arising from finite the precision of calculations, but requires more computational resources.

Example: `'SquareRoot',true`

Data Types: `logical`

`'Tolerance'` — Forecast uncertainty threshold

`0` (default) | nonnegative scalar

Forecast uncertainty threshold, specified as the comma-separated pair consisting of `'Tolerance'` and a nonnegative scalar.

If the forecast uncertainty for a particular observation is less than `Tolerance` during numerical estimation, then the software removes the uncertainty corresponding to the observation from the forecast covariance matrix before its inversion.

It is best practice to set `Tolerance` to a small number, for example, `1e-15`, to overcome numerical obstacles during estimation.

Example: `'Tolerance',le-15`

Data Types: `double`

`'Univariate'` — Univariate treatment of multivariate series flag

`false` (default) | `true`

Univariate treatment of a multivariate series flag, specified as the comma-separated pair consisting of `'Univariate'` and `true` or `false`. Univariate treatment of a multivariate series is also known as *sequential filtering*.

The univariate treatment can accelerate and improve numerical stability of the Kalman filter. However, all observation innovations must be uncorrelated. That is, D_tD_t' must be diagonal, where D_t, $t = 1,...,T$, is one of the following:

- The matrix `D{t}` in a time-varying state-space model
- The matrix `D` in a time-invariant state-space model

Example: `'Univariate',true`

Data Types: `logical`

Output Arguments

`X` — Smoothed states

matrix | cell vector of vectors

Smoothed states, returned as a matrix or a cell vector of matrices.

If `Mdl` is time invariant, then the number of rows of `X` is the sample size, and the number of columns of `X` is the number of states. The last row of `X` contains the latest, smoothed states.

If `Mdl` is time varying, then `X` is a cell vector with length equal to the sample size. Cell t of `X` contains a vector of smoothed states with length equal to the number of states in period t. The last cell of `X` contains the latest, smoothed states.

Data Types: `cell` | `double`

`logL` — Loglikelihood function value

scalar

Loglikelihood function value, returned as a scalar.

Missing observations do not contribute to the loglikelihood.

`Output` — Smoothing results by period

structure array

Smoothing results by period, returned as a structure array.

`Output` is a T-by-1 structure, where element t corresponds to the smoothing recursion at time t.

- If `Univariate` is `false` (it is by default), then the following table describes the fields of `Output`.

Field	Description	Estimate
`LogLikelihood`	Scalar loglikelihood objective function value	N/A
`SmoothedStates`	m_t-by-1 vector of smoothed states	$E(x_t \mid y_1,\ldots,y_T)$
`SmoothedStatesCov`	m_t-by-m_t variance-covariance matrix of the smoothed states	$Var(x_t \mid y_1,\ldots,y_T)$
`SmoothedStatesDisturb`	k_t-by-1 vector of smoothed state disturbances	$E(u_t \mid y_1,\ldots,y_T)$
`SmoothedStateDisturbCo`	k_t-by-k_t variance-covariance matrix of the smoothed, state disturbances	$Var(u_t \mid y_1,\ldots,y_T)$
`SmoothedObsInnov`	h_t-by-1 vector of smoothed observation innovations	$E(\varepsilon_t \mid y_1,\ldots,y_T)$
`SmoothedObsInnovCov`	h_t-by-h_t variance-covariance matrix of the smoothed, observation innovations	$Var(\varepsilon_t \mid y_1,\ldots,y_T)$
`KalmanGain`	m_t-by-n_t adjusted Kalman gain matrix	N/A
`DataUsed`	h_t-by-1 logical vector indicating whether the	N/A

Field	Description	Estimate
	software filters using a particular observation. For example, if observation i at time t is a `NaN`, then element i in `DataUsed` at time t is `0`.	

- If `Univariate` is `true`, then the fields of `Output` are the same as in the previous table, but the values in `KalmanGain` might vary.

Examples

Smooth States of Time-Invariant State-Space Model

Suppose that a latent process is an AR(1). Subsequently, the state equation is

$$x_t = 0.5x_{t-1} + u_t,$$

where u_t is Gaussian with mean 0 and standard deviation 0.5.

Generate a random series of 100 observations from x_t, assuming that the series starts at 1.5.

```
T = 100;
ARMdl = arima('AR',0.5,'Constant',0,'Variance',0.5^2);
x0 = 1.5;
rng(1); % For reproducibility
x = simulate(ARMdl,T,'Y0',x0);
```

Suppose further that the latent process is subject to additive measurement error. Subsequently, the observation equation is

$$y_t = x_t + \varepsilon_t,$$

where ε_t is Gaussian with mean 0 and standard deviation 0.05. Together, the latent process and observation equations compose a state-space model.

Use the random latent state process (`x`) and the observation equation to generate observations.

```
y = x + 0.05*randn(T,1);
```

Specify the four coefficient matrices.

```
A = 0.5;
B = 1;
C = 1;
D = 0.75;
```

Specify the state-space model using the coefficient matrices.

```
Mdl = ssm(A,B,C,D)
```

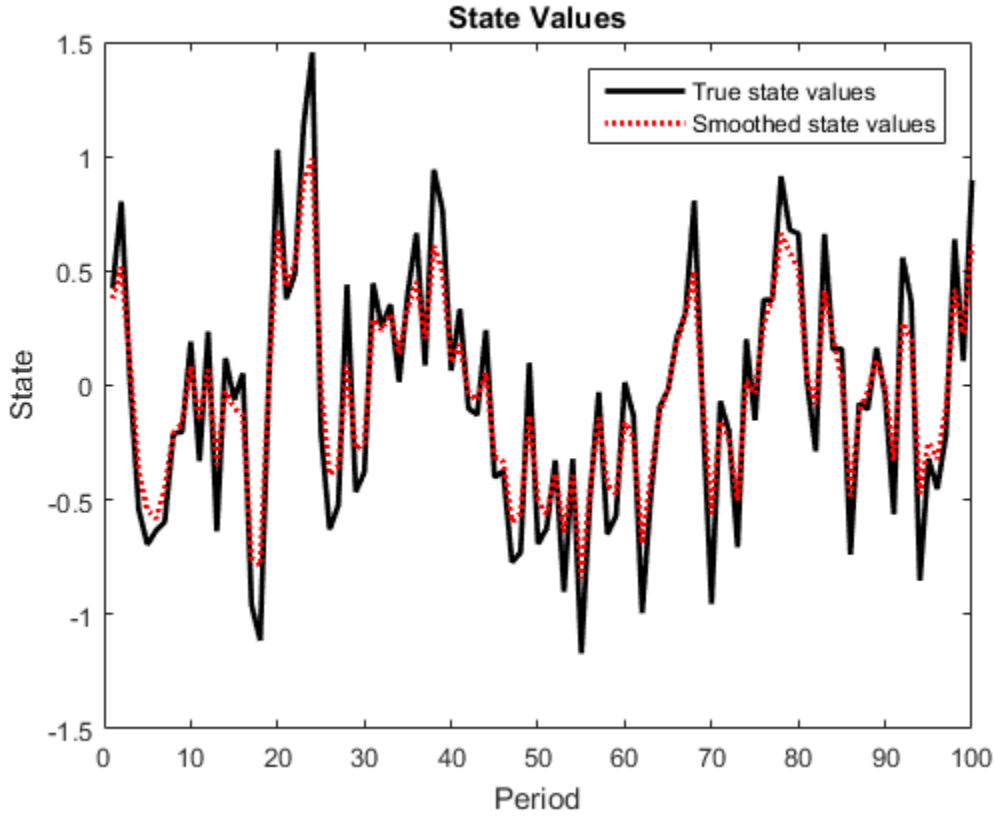
```
Mdl =


State-space model type: <a href="matlab: doc ssm">ssm</a>

State vector length: 1
Observation vector length: 1
State disturbance vector length: 1
Observation innovation vector length: 1
Sample size supported by model: Unlimited

State variables: x1, x2,...
State disturbances: u1, u2,...
Observation series: y1, y2,...
Observation innovations: e1, e2,...

State equation:
x1(t) = (0.50)x1(t-1) + u1(t)

Observation equation:
y1(t) = x1(t) + (0.75)e1(t)

Initial state distribution:

Initial state means
 x1
  0

Initial state covariance matrix
     x1
 x1  1.33

State types
```

```
     x1
 Stationary
```

`Mdl` is an `ssm` model. Verify that the model is correctly specified using the display in the Command Window. The software infers that the state process is stationary. Subsequently, the software sets the initial state mean and covariance to the mean and variance of the stationary distribution of an AR(1) model.

Smooth the states for periods 1 through 100. Plot the true state values and the smoothed states.

```
SmoothedX = smooth(Mdl,y);

figure
plot(1:T,x,'-k',1:T,SmoothedX,':r','LineWidth',2)
title({'State Values'})
xlabel('Period')
ylabel('State')
legend({'True state values','Smoothed state values'})
```



Smooth States of State-Space Model Containing Regression Component

Suppose that the linear relationship between the change in the unemployment rate and the nominal gross national product (nGNP) growth rate is of interest. Suppose further that the first difference of the unemployment rate is an ARMA(1,1) series. Symbolically, and in state-space form, the model is

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix} = \begin{bmatrix} \phi & \theta \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix} u_{1,t}$$
$$y_t - \beta Z_t = x_{1,t} + \sigma \varepsilon_t,$$

where:

- $x_{1,t}$ is the change in the unemployment rate at time t.
- $x_{2,t}$ is a dummy state for the MA(1) effect.
- $y_{1,t}$ is the observed unemployment rate being deflated by the growth rate of nGNP (Z_t).
- $u_{1,t}$ is the Gaussian series of state disturbances having mean 0 and standard deviation 1.
- ε_t is the Gaussian series of observation innovations having mean 0 and standard deviation σ.

Load the Nelson-Plosser data set, which contains the unemployment rate and nGNP series, among other things.

```
load Data_NelsonPlosser
```

Preprocess the data by taking the natural logarithm of the nGNP series, and the first difference of each series. Also, remove the starting `NaN` values from each series.

```
isNaN = any(ismissing(DataTable),2);       % Flag periods containing NaNs
gnpn = DataTable.GNPN(~isNaN);
u = DataTable.UR(~isNaN);
T = size(gnpn,1);                          % Sample size
Z = [ones(T-1,1) diff(log(gnpn))];
y = diff(u);
```

Though this example removes missing values, the software can accommodate series containing missing values in the Kalman filter framework.

Specify the coefficient matrices.

```
A = [NaN NaN; 0 0];
B = [1; 1];
C = [1 0];
D = NaN;
```

Specify the state-space model using `ssm`.

```
Mdl = ssm(A,B,C,D);
```

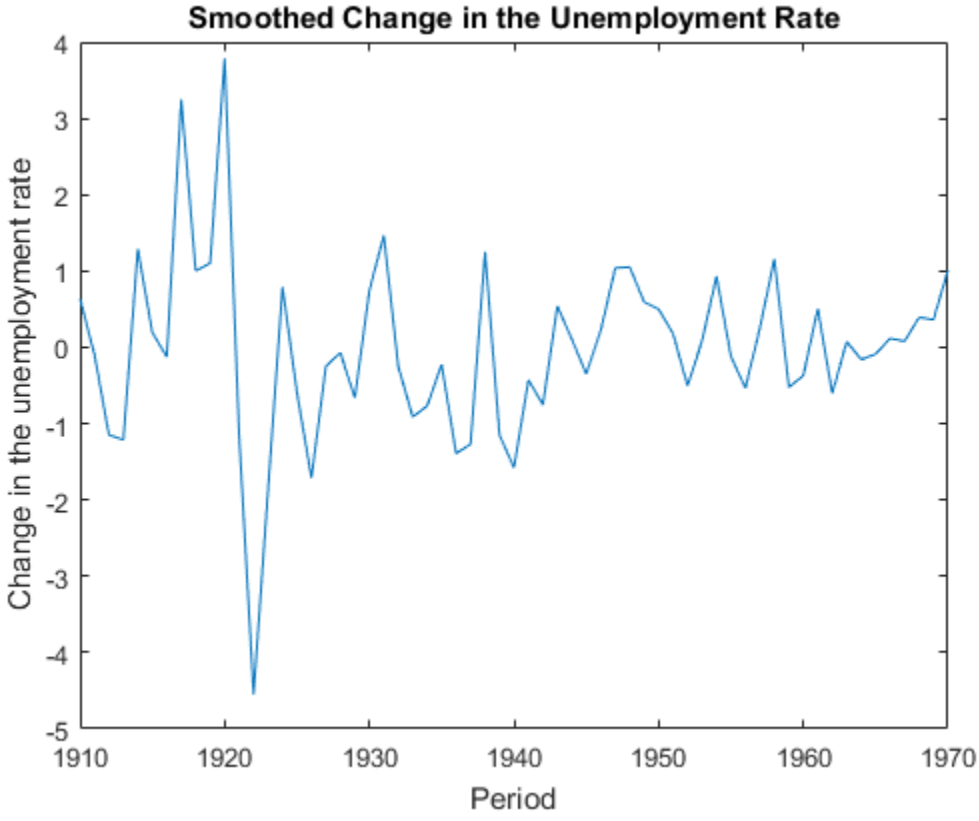
Estimate the model parameters. Specify the regression component and its initial value for optimization using the `'Predictors'` and `'Beta0'` name-value pair arguments, respectively. Restrict the estimate of σ to all positive, real numbers.

```
params0 = [0.3 0.2 0.2]; % Chosen arbitrarily
[EstMdl,estParams] = estimate(Mdl,y,params0,'Predictors',Z,...
    'Beta0',[0.1 0.2],'lb',[-Inf,-Inf,0,-Inf,-Inf]);


Method: Maximum likelihood (fmincon)
Sample size: 61
Logarithmic  likelihood:     -99.7245
Akaike   info criterion:      209.449
Bayesian info criterion:      220.003
           |      Coeff       Std Err    t Stat     Prob
-------------------------------------------------------------
 c(1)      |  -0.34098       0.29608    -1.15164  0.24948
 c(2)      |   1.05003       0.41377     2.53771  0.01116
 c(3)      |   0.48592       0.36790     1.32080  0.18657
 y <- z(1) |   1.36121       0.22338     6.09358   0
 y <- z(2) | -24.46711       1.60018   -15.29024   0
           |
           |    Final State   Std Dev     t Stat    Prob
 x(1)      |   1.01264       0.44690     2.26592  0.02346
 x(2)      |   0.77718       0.58917     1.31912  0.18713
```

`EstMdl` is an `ssm` model, and you can access its properties using dot notation.

Smooth the states. `EstMdl` does not store the data or the regression coefficients, so you must pass in them in using the name-value pair arguments `'Predictors'` and `'Beta'`, respectively. Plot the smoothed states. Recall that the first state is the change in the unemployment rate, and the second state helps build the first.

```
SmoothedX = smooth(EstMdl,y,'Predictors',Z,'Beta',estParams(end-1:end));

figure
plot(dates(end-(T-1)+1:end),SmoothedX(:,1));
xlabel('Period')
ylabel('Change in the unemployment rate')
title('Smoothed Change in the Unemployment Rate')
```



- "Smooth Time-Varying State-Space Model" on page 8-84
- "Compare Simulation Smoother to Smoothed States" on page 8-162

Algorithms

- The Kalman filter accommodates missing data by not updating filtered state estimates corresponding to missing observations. In other words, suppose there is a missing observation at period t. Then, the state forecast for period t based on the previous $t-1$ observations and filtered state for period t are equivalent.

- For explicitly defined state-space models, `ssm.smooth` applies all predictors to each response series. However, each response series has its own set of regression coefficients.

Tips

- `Mdl` does not store the response data, predictor data, and the regression coefficients. Supply the data wherever necessary using the appropriate input or name-value pair arguments.
- To accelerate estimation for low-dimensional, time-invariant models, set `'Univariate',true`. Using this specification, the software sequentially updates rather then updating all at once during the filtering process.

References

[1] Durbin J., and S. J. Koopman. *Time Series Analysis by State Space Methods*. 2nd ed. Oxford: Oxford University Press, 2012.

See Also

estimate | filter | forecast | refine | simsmooth | ssm

More About

- "What Are State-Space Models?" on page 8-3
- "What Is the Kalman Filter?" on page 8-8

ssm class

Create state-space model

Description

`ssm` creates a standard, linear, state-space model object with independent Gaussian state disturbances and observation innovations.

You can:

- Specify a time-invariant or time-varying model.
- Specify whether states are stationary, static, or nonstationary.
- Specify the state-transition, state-disturbance-loading, measurement-sensitivity, or observation-innovation matrices:
  - Explicitly by providing the matrices
  - Implicitly by providing a function that maps the parameters to the matrices, that is, a parameter-to-matrix mapping function

Once you have specified a model:

- If it contains unknown parameters, then pass the model and data to estimate, which estimates the parameters.
- If the state and observation matrices do not contain unknown parameters (for example, an estimated `ssm` model), then you can pass it to:
  - filter to implement forward recursion and obtain filtered estimates
  - forecast to obtain forecasted states and observations
  - smooth to implement backward recursion and obtain smoothed estimates
  - simulate to simulate states and observations from the state-space model
- `ssm` supports regression of exogenous predictors. To include a regression component that deflates the observations, see estimate, filter, forecast, and smooth.

Construction

`Mdl = ssm(A,B,C)` creates a state-space model (`Mdl`) using state-transition matrix `A`, state-disturbance-loading matrix `B`, and measurement-sensitivity matrix `C`.

`Mdl = ssm(A,B,C,D)` creates a state-space model using state-transition matrix `A`, state-disturbance-loading matrix `B`, measurement-sensitivity matrix `C`, and observation-innovation matrix `D`.

`Mdl = ssm(___,Name,Value)` uses any of the input arguments in the previous syntaxes and additional options that you specify by one or more `Name,Value` pair arguments.

`Name` can also be a property name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`Mdl = ssm(ParamMap)` creates a state-space model using a parameter-to-matrix mapping function (`ParamMap`) that you write. The function maps a vector of parameters to the matrices `A`, `B`, and `C`. Optionally, `ParamMap` can map parameters to `D`, `Mean0`, or `Cov0`. To specify the types of states, the function can return `StateType`. To accommodate a regression component in the observation equation, `ParamMap` can also return deflated observation data.

`Mdl = ssm(DSSMMdl)` converts a diffuse state-space model object (`DSSMMdl`) to a state-space model object (`Mdl`). `ssm` sets all initial variances of diffuse states in `SSMMdl.Cov0` to `1e07`.

Input Arguments

A — State-transition coefficient matrix

matrix | cell vector of matrices

State-transition coefficient matrix for explicit state-space model creation, specified as a matrix or cell vector of matrices.

The state-transition coefficient matrix, A_t, specifies how the states, x_t, are expected to transition from period $t-1$ to t, for all $t = 1,...,T$. That is, the expected state-transition equation at period t is $E(x_t \mid x_{t-1}) = A_t x_{t-1}$.

For time-invariant state-space models, specify `A` as an m-by-m matrix, where m is the number of states per period.

For time-varying state-space models, specify A as a T-dimensional cell array, where A{t} contains an m_t-by-m_{t-1} state-transition coefficient matrix. If the number of states changes from period $t-1$ to t, then $m_t \neq m_{t-1}$.

NaN values in any coefficient matrix indicate unique, unknown parameters in the state-space model. A contributes:

- sum(isnan(A(:))) unknown parameters to time-invariant state-space models. In other words, if the state-space model is time invariant, then the software uses the same unknown parameters defined in A at each period.
- numParamsA unknown parameters to time-varying state-space models, where numParamsA = sum(cell2mat(cellfun(@(x)sum(sum(isnan(x))),A,'UniformOutput',(
  In other words, if the state-space model is time varying, then the software assigns a new set of parameters for each matrix in A.

You cannot specify A and ParamMap simultaneously.

Data Types: double | cell

B — State-disturbance-loading coefficient matrix

matrix | cell vector of matrices

State-disturbance-loading coefficient matrix for explicit state-space model creation, specified as a matrix or cell vector of matrices.

The state disturbances, u_t, are independent Gaussian random variables with mean 0 and standard deviation 1. The state-disturbance-loading coefficient matrix, B_t, specifies the additive error structure in the state-transition equation from period $t-1$ to t, for all $t = 1,\ldots,T$. That is, the state-transition equation at period t is $x_t = A_t x_{t-1} + B_t u_t$.

For time-invariant state-space models, specify B as an m-by-k matrix, where m is the number of states and k is the number of state disturbances per period. B*B' is the state-disturbance covariance matrix for all periods.

For time-varying state-space models, specify B as a T-dimensional cell array, where B{t} contains an m_t-by-k_t state-disturbance-loading coefficient matrix. If the number of states or state disturbances changes at period t, then the matrix dimensions between B{t-1} and B{t} vary. B{t}*B{t}' is the state-disturbance covariance matrix for period t.

NaN values in any coefficient matrix indicate unique, unknown parameters in the state-space model. B contributes:

- `sum(isnan(B(:)))` unknown parameters to time-invariant state-space models. In other words, if the state-space model is time invariant, then the software uses the same unknown parameters defined in `B` at each period.
- `numParamsB` unknown parameters to time-varying state-space models, where `numParamsB = sum(cell2mat(cellfun(@(x)sum(sum(isnan(x))),B,'UniformOutput',0)))`. In other words, if the state-space model is time varying, then the software assigns a new set of parameters for each matrix in `B`.

You cannot specify `B` and `ParamMap` simultaneously.

Data Types: `double | cell`

`C` — Measurement-sensitivity coefficient matrix

matrix | cell vector of matrices

Measurement-sensitivity coefficient matrix for explicit state-space model creation, specified as a matrix or cell vector of matrices.

The measurement-sensitivity coefficient matrix, C_t, specifies how the states are expected to linearly combine at period t to form the observations, y_t, for all $t = 1,...,T$. That is, the expected observation equation at period t is $E(y_t | x_t) = C_t x_t$.

For time-invariant state-space models, specify `C` as an n-by-m matrix, where n is the number of observations and m is the number of states per period.

For time-varying state-space models, specify `C` as a T-dimensional cell array, where `C{t}` contains an n_t-by-m_t measurement-sensitivity coefficient matrix. If the number of states or observations changes at period t, then the matrix dimensions between `C{t-1}` and `C{t}` vary.

`NaN` values in any coefficient matrix indicate unique, unknown parameters in the state-space model. `C` contributes:

- `sum(isnan(C(:)))` unknown parameters to time-invariant state-space models. In other words, if the state-space model is time invariant, then the software uses the same unknown parameters defined in `C` at each period.
- `numParamsC` unknown parameters to time-varying state-space models, where `numParamsC = sum(cell2mat(cellfun(@(x)sum(sum(isnan(x))),C,'UniformOutput',0)))`. In other words, if the state-space model is time varying, then the software assigns a new set of parameters for each matrix in `C`.

You cannot specify `C` and `ParamMap` simultaneously.

Data Types: `double` | `cell`

`D` — Observation-innovation coefficient matrix

`[ ]` (default) | matrix | cell vector of matrices

Observation-innovation coefficient matrix for explicit state-space model creation, specified as a matrix or cell vector of matrices.

The observation innovations, ε_t, are independent Gaussian random variables with mean 0 and standard deviation 1. The observation-innovation coefficient matrix, D_t, specifies the additive error structure in the observation equation at period t, for all $t = 1,\ldots,T$. That is, the observation equation at period t is $y_t = C_t x_t + D_t \varepsilon_t$.

For time-invariant state-space models, specify `D` as an n-by-h matrix, where n is the number of observations and h is the number of observation innovations per period. `D*D'` is the observation-innovation covariance matrix for all periods.

For time-varying state-space models, specify `D` as a T-dimensional cell array, where `D{t}` contains an n_t-by-h_t matrix. If the number of observations or observation innovations changes at period t, then the matrix dimensions between `D{t-1}` and `D{t}` vary. `D{t}*D{t}'` is the observation-innovation covariance matrix for period `t`.

`NaN` values in any coefficient matrix indicate unique, unknown parameters in the state-space model. `D` contributes:

- `sum(isnan(D(:)))` unknown parameters to time-invariant state-space models. In other words, if the state-space model is time invariant, then the software uses the same unknown parameters defined in `D` at each period.
- `numParamsD` unknown parameters to time-varying state-space models, where `numParamsD = sum(cell2mat(cellfun(@(x)sum(sum(isnan(x))),D,'UniformOutput',0)))`. In other words, if the state-space model is time varying, then the software assigns a new set of parameters for each matrix in `D`.

By default, `D` is an empty matrix indicating no observation innovations in the state-space model.

You cannot specify `D` and `ParamMap` simultaneously.

Data Types: `double` | `cell`

`ParamMap` — Parameter-to-matrix mapping function

empty array ([ ]) (default) | function handle

Parameter-to-matrix mapping function for implicit state-space model creation, specified as a function handle.

`ParamMap` must be a function that takes at least one input argument and returns at least three output arguments. The requisite input argument is a vector of unknown parameters, and the requisite output arguments correspond to the coefficient matrices `A`, `B`, and `C`, respectively. If your parameter-to-mapping function requires the input parameter vector argument only, then implicitly create a state-space model by entering the following:

```
Mdl = ssm(@ParamMap)
```

In general, you can write an intermediate function, for example, `ParamFun`, using this syntax:

```
function [A,B,C,D,Mean0,Cov0,StateType,DeflateY] = ...
    ParamFun(params,...otherInputArgs...)
```

In this general case, create the state-space model by entering

```
Mdl = ssm(@(params)ParamMap(params,...otherInputArgs...))
```

However:

- Follow the order of the output arguments.
- `params` is a vector, and each element corresponds to an unknown parameter.
- `ParamFun` must return `A`, `B`, and `C`, which correspond to the state-transition, state-disturbance-loading, and measurement-sensitivity coefficient matrices, respectively.
- If you specify more input arguments than the parameter vector (`params`), such as observed responses and predictors, then implicitly create the state-space model using the syntax pattern

  ```
  Mdl = ssm(@(params)ParamFun(params,y,z))
  ```
- For the optional output arguments `D`, `Mean0`, `Cov0`, `StateType`, and `DeflateY`:

  - The optional output arguments correspond to the observation-innovation coefficient matrix `D` and the name-value pair arguments `Mean0`, `Cov0`, and `StateType`.

- To skip specifying an optional output argument, set the argument to `[]` in the function body. For example, to skip specifying `D`, then set `D = [];` in the function.
- `DeflateY` is the deflated-observation data, which accommodates a regression component in the observation equation. For example, in this function, which has a linear regression component, `Y` is the vector of observed responses and `Z` is the vector of predictor data.

  ```
  function [A,B,C,D,Mean0,Cov0,StateType,DeflateY] = ParamFun(params,Y,Z)
   ...
   DeflateY = Y - params(9) - params(10)*Z;
   ...
  end
  ```

- For the default values of `Mean0`, `Cov0`, and `StateType`, see "Algorithms" on page 9-996.

- It is best practice to:
  - Load the data to the MATLAB Workspace before specifying the model.
  - Create the parameter-to-matrix mapping function as its own file.

If you specify `ParamMap`, then you cannot specify any name-value pair arguments or any other input arguments.

Data Types: `function_handle`

`DSSMMdl` — Diffuse state-space model

`dssm` model object

Diffuse state-space model to convert to a state-space model, specified as a dssm model object.

`ssm` sets all initial variances of diffuse states in `DSSMMdl.Cov0` from `Inf` to `1e7`. Any diffuse states with variance other than `Inf` retain their values.

To apply the standard Kalman filter instead of the diffuse Kalman filter for filtering, smoothing, and parameter estimation, convert a diffuse state-space model to a state-space model.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`'Mean0'` — Initial state mean

numeric vector

Initial state mean for explicit state-space model creation, specified as the comma-separated pair consisting of `'Mean0'` and a numeric vector with length equal to the number of initial states. For the default values, see "Algorithms" on page 9-996.

If you specify `ParamMap`, then you cannot specify `Mean0`. Instead, specify the initial state mean in the parameter-to-matrix mapping function.

Data Types: `double`

`'Cov0'` — Initial state covariance matrix

square matrix

Initial state covariance matrix for explicit state-space model creation, specified as the comma-separated pair consisting of `'Cov0'` and a square matrix with dimensions equal to the number of initial states. For the default values, see "Algorithms" on page 9-996.

If you specify `ParamMap`, then you cannot specify `Cov0`. Instead, specify the initial state covariance in the parameter-to-matrix mapping function.

Data Types: `double`

`'StateType'` — Initial state distribution indicator

0 | 1 | 2

Initial state distribution indicator for explicit state-space model creation, specified as the comma-separated pair consisting of `'StateType'` and a numeric vector with length equal to the number of initial states. This table summarizes the available types of initial state distributions.

Value	Initial State Distribution Type
0	Stationary (for example, ARMA models)
1	The constant 1 (that is, the state is 1 with probability 1)
2	Diffuse or nonstationary (for example, random walk model, seasonal linear time series) or static state

For example, suppose that the state equation has two state variables: The first state variable is an AR(1) process, and the second state variable is a random walk. Specify the initial distribution types by setting `'StateType',[0; 2]`.

If you specify `ParamMap`, then you cannot specify `Mean0`. Instead, specify the initial state distribution indicator in the parameter-to-matrix mapping function.

For the default values, see "Algorithms" on page 9-996.

Data Types: `double`

Properties

A — State-transition coefficient matrix

matrix | cell vector of matrices | empty array ([ ])

State-transition coefficient matrix for explicitly created state-space models, specified as a matrix, a cell vector of matrices, or an empty array ([ ]). For implicitly created state-space models and before estimation, `A` is `[ ]` and read only.

The state-transition coefficient matrix, A_t, specifies how the states, x_t, are expected to transition from period $t-1$ to t, for all $t = 1,...,T$. That is, the expected state-transition equation at period t is $E(x_t \mid x_{t-1}) = A_t x_{t-1}$.

For time-invariant state-space models, `A` is an m-by-m matrix, where m is the number of states per period.

For time-varying state-space models, `A` is a T-dimensional cell array, where `A{t}` contains an m_t-by-m_{t-1} state-transition coefficient matrix. If the number of states changes from period $t-1$ to t, then $m_t \neq m_{t-1}$.

`NaN` values in any coefficient matrix indicate unknown parameters in the state-space model. `A` contributes:

- `sum(isnan(A(:)))` unknown parameters to time-invariant state-space models. In other words, if the state-space model is time invariant, then the software uses the same unknown parameters defined in `A` at each period.
- `numParamsA` unknown parameters to time-varying state-space models, where `numParamsA = sum(cell2mat(cellfun(@(x)sum(sum(isnan(x))),A,'UniformOutput',(`

In other words, if the state-space model is time varying, then the software assigns a new set of parameters for each matrix in `A`.

Data Types: `double` | `cell`

`B` — State-disturbance-loading coefficient matrix

matrix | cell vector of matrices | empty array (`[]`)

State-disturbance-loading coefficient matrix for explicitly created state-space models, specified as a matrix, a cell vector of matrices, or an empty array (`[]`). For implicitly created state-space models and before estimation, `B` is `[]` and read only.

The state disturbances, u_t, are independent Gaussian random variables with mean 0 and standard deviation 1. The state-disturbance-loading coefficient matrix, B_t, specifies the additive error structure in the state-transition equation from period $t - 1$ to t, for all $t = 1,...,T$. That is, the state-transition equation at period t is $x_t = A_t x_{t-1} + B_t u_t$.

For time-invariant state-space models, `B` is an m-by-k matrix, where m is the number of states and k is the number of state disturbances per period. `B*B'` is the state-disturbance covariance matrix for all periods.

For time-varying state-space models, `B` is a T-dimensional cell array, where `B{t}` contains an m_t-by-k_t state-disturbance-loading coefficient matrix. If the number of states or state disturbances changes at period t, then the matrix dimensions between `B{t-1}` and `B{t}` vary. `B{t}*B{t}'` is the state-disturbance covariance matrix for period `t`.

`NaN` values in any coefficient matrix indicate unknown parameters in the state-space model. `B` contributes:

- `sum(isnan(B(:)))` unknown parameters to time-invariant state-space models. In other words, if the state-space model is time invariant, then the software uses the same unknown parameters defined in `B` at each period.
- `numParamsB` unknown parameters to time-varying state-space models, where `numParamsB = sum(cell2mat(cellfun(@(x)sum(sum(isnan(x))),B,'UniformOutput',0)))`. In other words, if the state-space model is time varying, then the software assigns a new set of parameters for each matrix in `B`.

Data Types: `double` | `cell`

`C` — Measurement-sensitivity coefficient matrix

matrix | cell vector of matrices | empty array (`[]`)

Measurement-sensitivity coefficient matrix for explicitly created state-space models, specified as a matrix, a cell vector of matrices, or an empty array (`[]`). For implicitly created state-space models and before estimation, `C` is `[]` and read only.

The measurement-sensitivity coefficient matrix, C_t, specifies how the states are expected to combine linearly at period t to form the observations, y_t, for all $t = 1,...,T$. That is, the expected observation equation at period t is $E(y_t | x_t) = C_t x_t$.

For time-invariant state-space models, `C` is an n-by-m matrix, where n is the number of observations and m is the number of states per period.

For time-varying state-space models, `C` is a T-dimensional cell array, where `C{t}` contains an n_t-by-m_t measurement-sensitivity coefficient matrix. If the number of states or observations changes at period t, then the matrix dimensions between `C{t-1}` and `C{t}` vary.

`NaN` values in any coefficient matrix indicate unknown parameters in the state-space model. `C` contributes:

- `sum(isnan(C(:)))` unknown parameters to time-invariant state-space models. In other words, if the state-space model is time invariant, then the software uses the same unknown parameters defined in `C` at each period.
- `numParamsC` unknown parameters to time-varying state-space models, where `numParamsC = sum(cell2mat(cellfun(@(x)sum(sum(isnan(x))),C,'UniformOutput',0)))`. In other words, if the state-space model is time varying, then the software assigns a new set of parameters for each matrix in `C`.

Data Types: `double` | `cell`

`D` — Observation-innovation coefficient matrix

matrix | cell vector of matrices | empty array (`[]`)

Observation-innovation coefficient matrix for explicitly created state-space models, specified as a matrix, a cell vector of matrices, or an empty array (`[]`). For implicitly created state-space models and before estimation, `D` is `[]` and read only.

The observation innovations, ε_t, are independent Gaussian random variables with mean 0 and standard deviation 1. The observation-innovation coefficient matrix, D_t, specifies the additive error structure in the observation equation at period t, for all $t = 1,...,T$. That is, the observation equation at period t is $y_t = C_t x_t + D_t \varepsilon_t$.

For time-invariant state-space models, `D` is an n-by-h matrix, where n is the number of observations and h is the number of observation innovations per period. `D*D'` is the observation-innovation covariance matrix for all peroids.

For time-varying state-space models, `D` is a T-dimensional cell array, where `D{t}` contains an n_t-by-h_t matrix. If the number of observations or observation innovations changes at period t, then the matrix dimensions between `D{t-1}` and `D{t}` vary. `D{t}*D{t}'` is the state-disturbance covariance matrix for period `t`.

`NaN` values in any coefficient matrix indicate unknown parameters in the state-space model. `D` contributes:

- `sum(isnan(D(:)))` unknown parameters to time-invariant state-space models. In other words, if the state-space model is time invariant, then the software uses the same unknown parameters defined in `D` at each period.
- `numParamsD` unknown parameters to time-varying state-space models, where `numParamsD = sum(cell2mat(cellfun(@(x)sum(sum(isnan(x))),D,'UniformOutput',0)))`. In other words, if the state-space model is time varying, then the software assigns a new set of parameters for each matrix in `D`.

Data Types: `double` | `cell`

`Mean0` — Initial state mean

numeric vector | empty array (`[]`)

Initial state mean, specified as a numeric vector or an empty array (`[]`). `Mean0` has length equal to the number of initial states (`size(A,1)` or `size(A{1},1)`).

`Mean0` is the mean of the Gaussian distribution of the states at period 0.

For implicitly created state-space models and before estimation, `Mean0` is `[]` and read only. However, `estimate` specifies `Mean0` after estimation.

Data Types: `double`

`Cov0` — Initial state covariance matrix

square matrix | empty array (`[]`)

Initial state covariance matrix, specified as a square matrix or an empty array (`[]`). `Cov0` has dimensions equal to the number of initial states (`size(A,1)` or `size(A{1},1)`).

`Cov0` is the covariance of the Gaussian distribution of the states at period 0.

For implicitly created state-space models and before estimation, `Cov0` is `[]` and read only. However, `estimate` specifies `Cov0` after estimation.

Data Types: `double`

`StateType` — Initial state distribution type

numeric vector | empty array (`[]`)

Initial state distribution indicator, specified as a numeric vector or empty array (`[]`). `StateType` has length equal to the number of initial states.

For implicitly created state-space models or models with unknown parameters, `StateType` is `[]` and read only.

This table summarizes the available types of initial state distributions.

Value	Initial State Distribution Type
0	Stationary (e.g., ARMA models)
1	The constant 1 (that is, the state is 1 with probability 1)
2	Nonstationary (e.g., random walk model, seasonal linear time series) or static state

For example, suppose that the state equation has two state variables: The first state variable is an AR(1) process, and the second state variable is a random walk. Then, `StateType` is `[0; 2]`.

For nonstationary states, `ssm` sets `Cov0` to `1e7` by default. Subsequently, the software implements the Kalman filter for filtering, smoothing, and parameter estimation. This specification imposes relatively weak knowledge on the initial state values of diffuse states, and uses initial state covariance terms between all states.

Data Types: `double`

`ParamMap` — Parameter-to-matrix mapping function

function handle | empty array (`[]`)

Parameter-to-matrix mapping function, specified as a function handle or an empty array (`[]`). `ParamMap` completely specifies the structure of the state-space model. That is,

`ParamMap` defines `A`, `B`, `C`, `D`, and, optionally, `Mean0`, `Cov0`, and `StateType`. For explicitly created state-space models, `ParamMap` is `[]` and read only.

Data Types: `function_handle`

Methods

disp	Display summary information for state-space model
estimate	Maximum likelihood parameter estimation of state-space models
filter	Forward recursion of state-space models
forecast	Forecast states and observations of state-space models
refine	Refine initial parameters to aid state-space model estimation
simsmooth	State-space model simulation smoother
simulate	Monte Carlo simulation of state-space models
smooth	Backward recursion of state-space models

Definitions

Static State

A *static state* does not change in value throughout the sample, that is, $P\left(x_{t+1} = x_t\right) = 1$ for all $t = 1,\ldots,T$.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Explicitly Create Standard State-Space Model with Known and Unknown Parameters

Create a standard state-space model containing two independent, autoregressive states, and the observations are the deterministic sum of the two states. Symbolically, the system of equations is

$$\begin{bmatrix} x_{t,1} \\ x_{t,2} \end{bmatrix} = \begin{bmatrix} \phi_1 & 0 \\ 0 & \phi_2 \end{bmatrix} \begin{bmatrix} x_{t-1,1} \\ x_{t-1,2} \end{bmatrix} + \begin{bmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{bmatrix} \begin{bmatrix} u_{t,1} \\ u_{t,2} \end{bmatrix}$$

$$y_t = \begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} x_{t,1} \\ x_{t,2} \end{bmatrix}.$$

Specify the state-transition matrix.

```
A = [NaN 0; 0 NaN];
```

Specify the state-disturbance-loading matrix.

```
B = [NaN 0; 0 NaN];
```

Specify the measurement-sensitivity matrix.

```
C = [1 1];
```

Define the state-space model using `ssm`.

```
Mdl = ssm(A,B,C)
```

```
Mdl =


State-space model type: <a href="matlab: doc ssm">ssm</a>

State vector length: 2
Observation vector length: 1
State disturbance vector length: 2
Observation innovation vector length: 0
Sample size supported by model: Unlimited
Unknown parameters for estimation: 4

State variables: x1, x2,...
```

```
State disturbances: u1, u2,...
Observation series: y1, y2,...
Observation innovations: e1, e2,...
Unknown parameters: c1, c2,...

State equations:
x1(t) = (c1)x1(t-1) + (c3)u1(t)
x2(t) = (c2)x2(t-1) + (c4)u2(t)

Observation equation:
y1(t) = x1(t) + x2(t)

Initial state distribution:

Initial state means are not specified.
Initial state covariance matrix is not specified.
State types are not specified.
```

`Mdl` is an `ssm` model containing unknown parameters. A detailed summary of `Mdl` prints to the Command Window.

It is good practice to verify that the state and observation equations are correct. If the equations are not correct, then it might help to expand the state-space equation manually.

Explicitly Create State-Space Model with Observation Error

Create a state-space model containing two independent, autoregressive states, and the observations are the sum of the two states, plus Gaussian error. Symbolically, the equation is

$$\begin{bmatrix} x_{t,1} \\ x_{t,2} \end{bmatrix} = \begin{bmatrix} \phi_1 & 0 \\ 0 & \phi_2 \end{bmatrix} \begin{bmatrix} x_{t-1,1} \\ x_{t-1,2} \end{bmatrix} + \begin{bmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{bmatrix} \begin{bmatrix} u_{t,1} \\ u_{t,2} \end{bmatrix}$$

$$y_t = \begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} x_{t,1} \\ x_{t,2} \end{bmatrix} + \sigma_3 \varepsilon_t.$$

Define the state-transition matrix.

```
A = [NaN 0; 0 NaN];
```

Define the state-disturbance-loading matrix.

```
B = [NaN 0; 0 NaN];
```

Define the measurement-sensitivity matrix.

```
C = [1 1];
```

Define the observation-innovation matrix.

```
D = NaN;
```

Create the state-space model using `ssm`.

```
 Mdl = ssm(A,B,C,D)
```

```
Mdl =


State-space model type: <a href="matlab: doc ssm">ssm</a>

State vector length: 2
Observation vector length: 1
State disturbance vector length: 2
Observation innovation vector length: 1
Sample size supported by model: Unlimited
Unknown parameters for estimation: 5

State variables: x1, x2,...
State disturbances: u1, u2,...
Observation series: y1, y2,...
Observation innovations: e1, e2,...
Unknown parameters: c1, c2,...

State equations:
x1(t) = (c1)x1(t-1) + (c3)u1(t)
x2(t) = (c2)x2(t-1) + (c4)u2(t)

Observation equation:
y1(t) = x1(t) + x2(t) + (c5)e1(t)

Initial state distribution:

Initial state means are not specified.
Initial state covariance matrix is not specified.
State types are not specified.
```

`Mdl` is an `ssm` model containing unknown parameters. A detailed summary of `Mdl` prints to the Command Window.

It is good practice to verify that the state and observations equations are correct. If the equations are not correct, then it might help to expand the state-space equation manually.

Pass the data and `Mdl` to `estimate` to estimate the parameters.

Create Known State-Space Model with Initial State Values

Create a state-space model, where the state equation is an AR(2) model. The state disturbances are mean zero Gaussian random variables with standard deviation of 0.3. The observation equation is the difference between the current and previous state plus a mean zero Gaussian observation innovation with a standard deviation of 0.1. Symbolically, the state-space model is

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \\ x_{3,t} \end{bmatrix} = \begin{bmatrix} 0.6 & 0.2 & 0.5 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \\ x_{3,t-1} \end{bmatrix} + \begin{bmatrix} 0.3 \\ 0 \\ 0 \end{bmatrix} u_{1,t}$$

$$y_t = \begin{bmatrix} 1 & -1 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t} \\ x_{2,t} \\ x_{3,t} \end{bmatrix} + 0.1\varepsilon_t.$$

There are three states: $x_{1,t}$ is the AR(2) process, $x_{2,t}$ represents $x_{1,t-1}$, and $x_{3,t}$ is the AR(2) model constant.

Define the state-transition matrix.

```
A = [0.6 0.2 0.5; 1 0 0; 0 0 1];
```

Define the state-disturbance-loading matrix.

```
B = [0.3; 0; 0];
```

Define the measurement-sensitivity matrix.

```
C = [1 -1 0];
```

Define the observation-innovation matrix.

```
D = 0.1;
```

Use `ssm` to create the state-space model. Set the initial-state mean (`Mean0`) and covariance matrix (`Cov0`). Identify the type of initial state distributions (`StateType`) by noting the following:

- $x_{1,t}$ is a stationary, AR(2) process.
- $x_{2,t}$ is also a stationary, AR(2) process.
- $x_{3,t}$ is the constant 1 for all periods.

```
Mean0 = [0; 0; 1];                                          % The mean of the AR(2)
varAR2 = 0.3*(1 - 0.2)/((1 + 0.2)*((1 - 0.2)^2 - 0.6^2)); % The variance of the AR(2)
Cov1AR2 = 0.6*0.3/((1 + 0.2)*((1 - 0.2)^2) - 0.6^2);      % The covariance of the AR(2
Cov0 = zeros(3);
Cov0(1:2,1:2) = varAR2*eye(2) + Cov1AR2*flip(eye(2));
StateType = [0; 0; 1];
Mdl = ssm(A,B,C,D,'Mean0',Mean0,'Cov0',Cov0,'StateType',StateType)


Mdl =


State-space model type: <a href="matlab: doc ssm">ssm</a>

State vector length: 3
Observation vector length: 1
State disturbance vector length: 1
Observation innovation vector length: 1
Sample size supported by model: Unlimited

State variables: x1, x2,...
State disturbances: u1, u2,...
Observation series: y1, y2,...
Observation innovations: e1, e2,...

State equations:
x1(t) = (0.60)x1(t-1) + (0.20)x2(t-1) + (0.50)x3(t-1) + (0.30)u1(t)
x2(t) = x1(t-1)
x3(t) = x3(t-1)

Observation equation:
y1(t) = x1(t) - x2(t) + (0.10)e1(t)

Initial state distribution:
```

```
Initial state means
 x1  x2  x3
  0   0   1

Initial state covariance matrix
     x1    x2    x3
 x1  0.71  0.44   0
 x2  0.44  0.71   0
 x3   0     0     0

State types
     x1          x2          x3
 Stationary  Stationary  Constant
```

`Mdl` is an `ssm` model.

You can display properties of `Mdl` using dot notation. For example, display the initial state covariance matrix.

```
Mdl.Cov0
```

```
ans =

    0.7143    0.4412         0
    0.4412    0.7143         0
         0         0         0
```

Implicitly Create Time-Invariant State-Space Model

Use a parameter mapping function to create a time-invariant state-space model, where the state model is AR(1) model. The states are observed with bias, but without random error. Set the initial state mean and variance, and specify that the state is stationary.

Write a function that specifies how the parameters in `params` map to the state-space model matrices, the initial state values, and the type of state. Symbolically, the model is

$$\begin{aligned} x_t &= \phi x_{t-1} + \sigma u_t \\ y_t &= a x_t \end{aligned}.$$

```
% Copyright 2015 The MathWorks, Inc.

function [A,B,C,D,Mean0,Cov0,StateType] = timeInvariantParamMap(params)
% Time-invariant state-space model parameter mapping function example. This
% function maps the vector params to the state-space matrices (A, B, C, and
% D), the initial state value and the initial state variance (Mean0 and
% Cov0), and the type of state (StateType). The state model is AR(1)
% without observation error.
    varu1 = exp(params(2)); % Positive variance constraint
    A = params(1);
    B = sqrt(varu1);
    C = params(3);
    D = [];
    Mean0 = 0.5;
    Cov0 = 100;
    StateType = 0;
end
```

Save this code as a file named `timeInvariantParamMap.m` to a folder on your MATLAB® path.

Create the state-space model by passing the function `timeInvariantParamMap` as a function handle to `ssm`.

```
Mdl = ssm(@timeInvariantParamMap);
```

`ssm` implicitly creates the state-space model. Usually, you cannot verify implicitly defined state-space models.

Convert Diffuse to Standard State-Space Model

If you estimate, filter, or smooth a diffuse state-space model containing at least one diffuse state, then the software uses the diffuse Kalman filter. To use the standard Kalman filter instead, convert the diffuse state-space model to a standard state-space model. `ssm` attributes a large initial state variance (`1e7`) for diffuse states. A standard state-space model treatment results in an approximation to the results of the diffuse Kalman filter. However, `estimate` uses all of the data to fit the model, and `filter` and `smooth` return filtered and smoothed estimates for all periods, respectively.

Explicitly create a one-dimensional diffuse state-space model. Specify that the first state equation is $x_t = x_{t-1} + u_t$, and that the observation model is $y_t = x_t + \varepsilon_t$.

```
A = 1;
```

```
B = 1;
C = 1;
D = 1;
DSSMMdl = dssm(A,B,C,D)


DSSMMdl =


State-space model type: <a href="matlab: doc dssm">dssm</a>

State vector length: 1
Observation vector length: 1
State disturbance vector length: 1
Observation innovation vector length: 1
Sample size supported by model: Unlimited

State variables: x1, x2,...
State disturbances: u1, u2,...
Observation series: y1, y2,...
Observation innovations: e1, e2,...

State equation:
x1(t) = x1(t-1) + u1(t)

Observation equation:
y1(t) = x1(t) + e1(t)

Initial state distribution:

Initial state means
 x1
  0

Initial state covariance matrix
     x1
 x1  Inf

State types
    x1
 Diffuse
```

`DSSMMdl` is a `dssm` model object. Because the model does not contain any unknown parameters, `dssm` infers the initial state distribution and its parameters. In particular,

the initial state variance is `Inf` because the nonstationary state has a diffuse distribution by default.

Convert `DSSMMdl` to a standard state-space model.

```
Mdl = ssm(DSSMMdl)
```

```
Mdl =


State-space model type: <a href="matlab: doc ssm">ssm</a>

State vector length: 1
Observation vector length: 1
State disturbance vector length: 1
Observation innovation vector length: 1
Sample size supported by model: Unlimited

State variables: x1, x2,...
State disturbances: u1, u2,...
Observation series: y1, y2,...
Observation innovations: e1, e2,...

State equation:
x1(t) = x1(t-1) + u1(t)

Observation equation:
y1(t) = x1(t) + e1(t)

Initial state distribution:

Initial state means
 x1
  0

Initial state covariance matrix
     x1
 x1  1e+07

State types
    x1
 Diffuse
```

`Mdl` is an `ssm` model object. The structures of `Mdl` and `DSSMMdl` are equivalent, except that the initial state variance of the state in `Mdl` is `1e7`.

To see the difference between the two models, simulate 10 periods of data from a state-space model that is similar to `Mdl`, except it has known initial state mean of 5 and variance 2.

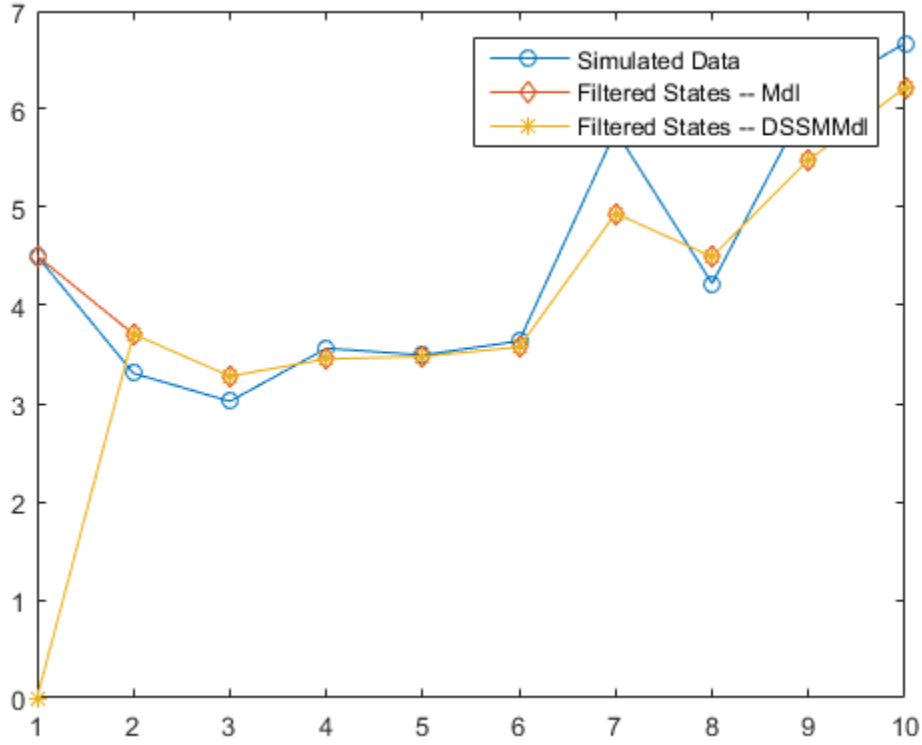
```
SimMdl = ssm(A,B,C,D,'Mean0',5,'Cov0',2,'StateType',2);
T = 10;
rng(1); % For reproducibility
y = simulate(SimMdl,T);
```

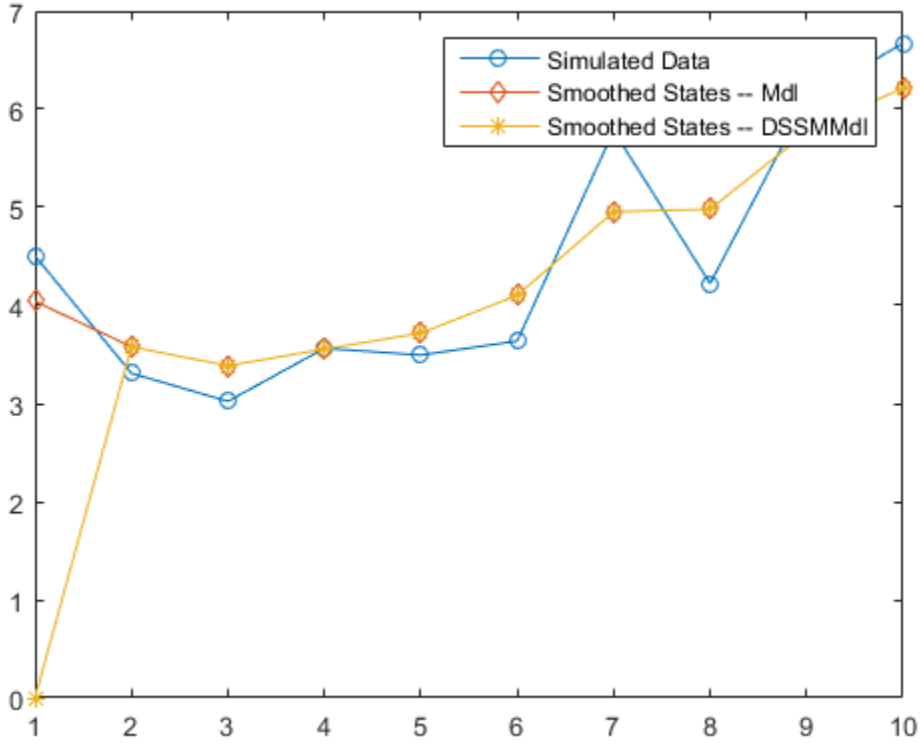
Obtain filtered and smoothed states from `Mdl` and `DSSMMdl` using the simulated data.

```
fMdl = filter(Mdl,y);
fDSSMMdl = filter(DSSMMdl,y);
sMdl = smooth(Mdl,y);
sDSSMMdl = smooth(DSSMMdl,y);
```

Plot the filtered and smoothed states.

```
figure;
plot(1:T,y,'-o',1:T,fMdl,'-d',1:T,fDSSMMdl,'-*');
legend('Simulated Data','Filtered States -- Mdl','Filtered States -- DSSMMdl');

figure;
plot(1:T,y,'-o',1:T,sMdl,'-d',1:T,sDSSMMdl,'-*');
legend('Simulated Data','Smoothed States -- Mdl','Smoothed States -- DSSMMdl');
```


Simulated Data
Filtered States -- Mdl
Filtered States -- DSSMMdl
0
1
2
3
4
5
6
7
1
2
3
4
5
6
7
8
9
10



Besides apparent transient behavior, the filtered and smoothed states between the standard and diffuse state-space models appear nearly equivalent. The slight difference occurs because `filter` and `smooth` set all diffuse state estimates in the diffuse state-space model to 0 while they implement the diffuse Kalman filter. Once the covariance matrices of the smoothed states attain full rank, `filter` and `smooth` switch to using the standard Kalman filter. In this case, the switching time occurs after the first period.

- “Implicitly Create Time-Invariant State-Space Model” on page 8-22
- “Implicitly Create Time-Varying State-Space Model” on page 8-32
- “Implicitly Create State-Space Model Containing Regression Component” on page 8-28
- “Create State-Space Model with Random State Coefficient” on page 8-38

Tip

Specify `ParamMap` in a more general or complex setting, where, for example:

- The initial state values are parameters.
- In time-varying models, you want to use the same parameters for more than one period.
- You want to impose parameter constraints.

Algorithms

- Default values for `Mean0` and `Cov0`:
  - If you explicitly specify the state-space model (that is, you provide the coefficient matrices `A`, `B`, `C`, and optionally `D`), then:
    - For stationary states, the software generates the initial value using the stationary distribution. If you provide all values in the coefficient matrices (that is, your model has no unknown parameters), then `ssm` generates the initial values. Otherwise, the software generates the initial values during estimation.
    - For states that are always the constant 1, `ssm` sets `Mean0` to 1 and `Cov0` to `0`.
    - For diffuse states, the software sets `Mean0` to 0 and `Cov0` to `1e7` by default.
  - If you implicitly create the state-space model (that is, you provide the parameter vector to the coefficient-matrices-mapping function `ParamMap`), then the software generates any initial values during estimation.
- For static states that do not equal 1 throughout the sample, the software cannot assign a value to the degenerate, initial state distribution. Therefore, set static states to `2` using the name-value pair argument `StateType`. Subsequently, the software treats static states as nonstationary and assigns the static state a diffuse initial distribution.
- It is best practice to set `StateType` for each state. By default, the software generates `StateType`, but this behavior might not be accurate. For example, the software cannot distinguish between a constant 1 state and a static state.
- The software cannot infer `StateType` from data because the data theoretically comes from the observation equation. The realizations of the state equation are unobservable.

- `ssm` models do not store observed responses or predictor data. Supply the data wherever necessary using the appropriate input or name-value pair arguments.
- Suppose that you want to create a state-space model using a parameter-to-matrix mapping function with this signature:

  ```
  [A,B,C,D,Mean0,Cov0,StateType,DeflateY] = paramMap(params,Y,Z)
  ```
  and you specify the model using an anonymous function

  ```
  Mdl = ssm(@(params)paramMap(params,Y,Z))
  ```
  The observed responses `Y` and predictor data `Z` are not input arguments in the anonymous function. If `Y` and `Z` exist in the MATLAB Workspace before you create `Mdl`, then the software establishes a link to them. Otherwise, if you pass `Mdl` to estimate, the software throws an error.

  The link to the data established by the anonymous function overrides all other corresponding input argument values of `estimate`. This distinction is important particularly when conducting a rolling window analysis. For details, see "Rolling-Window Analysis of Time-Series Models" on page 8-168.

Alternatives

- If the states are observable, and the state equation resembles:
  - An ARIMA model, then you can specify an arima model instead.
  - A regression model with ARIMA errors, then you can specify a regARIMA model instead.
  - A conditional variance model, then you can specify a `garch`, `egarch`, or `gjr` model instead.
  - A VAR model, then you can estimate such a model using `vgxvarx`.
- To impose no prior knowledge on the initial state values of diffuse states, and to implement the diffuse Kalman filter, create a dssm model object instead of an `ssm` model object.

References

[1] Durbin J., and S. J. Koopman. *Time Series Analysis by State Space Methods*. 2nd ed. Oxford: Oxford University Press, 2012.

See Also

dssm

More About

- "What Are State-Space Models?" on page 8-3
- "Rolling-Window Analysis of Time-Series Models" on page 8-168

toCellArray

Class: LagOp

Convert lag operator polynomial object to cell array

Syntax

```
[coefficients, lags] = toCellArray(A)
```

Description

`[coefficients, lags] = toCellArray(A)` converts a lag operator polynomial object *A(L)* to an equivalent cell array. `coefficients` is the cell array equivalent to the lag operator polynomial *A(L)*. `lags` is a vector of unique integer lags associated with the polynomial coefficients. Elements of lags are in ascending order. The first element of lags is the smaller of the smallest nonzero coefficient lag of the object and zero; the last element of lags is the degree of the polynomial. That is, `lags = [min(A.Lags,0), 1, 2, ... A.Degree]`.

Algorithms

`LagOp` objects implicitly store polynomial lags and corresponding coefficient matrices of zero-valued coefficients via lag-based indexing. However, cell arrays conform to traditional element indexing rules, and must explicitly store zero coefficient matrices.

The output cell array is equivalent to the input lag operator polynomial in the sense that the same lag operator is created when the output coefficients and lags are used to create a new `LagOp` object. That is, the following two statements produce the same polynomial *A(L)*:

```
[coefficients,lags] = toCellArray(A);
A = LagOp(coefficients,'Lags',lags);
```

Examples

Convert Lag Operator to a Cell Array

Create a `LagOp` polynomial and convert it to a cell array:

```
A = LagOp({0.8 1 0 .6});
B = toCellArray(A);
class(B)
```

```
ans =

cell
```

var2vec

Convert VAR model to VEC model

If any of the time series in a vector autoregression (VAR) model are cointegrated, then the VAR model is nonstationary. You can determine the error-correction coefficient by converting the VAR model to a vector error-correction (VEC) model. The error-correction coefficient matrix determines, on average, how the time series react to deviations from their long-run averages. The rank of the error-correction coefficient determines how many cointegrating relations there exist in the model.

Because `vgxvarx` is suitable for estimating VAR models in reduced form, you can convert an estimated VAR model to its VEC model equivalent using `var2vec`.

Syntax

```
[VEC,C] = var2vec(VAR)
```

Description

`[VEC,C] = var2vec(VAR)` returns the coefficient matrices (`VEC`) and the error-correction coefficient matrix (`C`) of the vector error-correction model equivalent to the vector autoregressive model with coefficient matrices (`VAR`). If the number of lags in the input vector autoregressive model is p, then the number of lags in the output vector error-correction model is $q = p - 1$.

Examples

Convert VAR Model to VEC Model Using Cell Arrays

Consider converting the following VAR(3) model to a VEC(2) model.

$$y_t = \begin{bmatrix} 0.5 \\ 1 \\ -2 \end{bmatrix} + \begin{bmatrix} 0.54 & 0.86 & -0.43 \\ 1.83 & 0.32 & 0.34 \\ -2.26 & -1.31 & 3.58 \end{bmatrix} y_{t-1} + \begin{bmatrix} 0.14 & -0.12 & 0.05 \\ 0.14 & 0.07 & 0.10 \\ 0.07 & 0.16 & 0.07 \end{bmatrix} y_{t-3} + \varepsilon_t.$$

Specify the coefficient matrices (A_1, A_2, and A_3) of the VAR(3) model terms y_{t-1}, y_{t-2}, and y_{t-3}.

```
A1 = [0.54  0.86 -0.43;
      1.83  0.32  0.34;
     -2.26 -1.31  3.58];
A2 = zeros(3);
A3 = [0.14 -0.12 0.05;
      0.14  0.07 0.10;
      0.07  0.16 0.07];
```

Pack the matrices into separate cells of a 3 dimensional cell vector. Put `A1` into the first cell, `A2` into the second cell, and `A3` into the third cell.

```
VAR = {A1 A2 A3};
```

Compute the coefficient matrices of Δy_{t-1} and Δy_{t-2}, and error-correction coefficient matrix of the equivalent VEC(2) model.

```
[VEC,C] = var2vec(VAR);
size(VEC)
```

```
ans =

     1     2
```

The specification of a cell array of matrices for the input argument indicates that the VAR(3) model is a reduced-form model expressed as a difference equation. `VAR{1}` is the coefficient of y_{t-1}, and subsequent elements correspond to subsequent lags.

`VEC` is a 1-by-2 cell vector of 3-by-3 coefficient matrices for the VEC(2) equivalent of the VAR(3) model. Because the VAR(3) model is in reduced form, the equivalent VEC model is also. That is, `VEC{1}` is the coefficient of Δy_{t-1}, and subsequent elements correspond to subsequent lags. The orientation of `VEC` corresponds to the orientation of `VAR`.

Display the VEC(2) model coefficients.

```
B1 = VEC{1}
B2 = VEC{2}
C
```

```
B1 =

   -0.1400    0.1200   -0.0500
   -0.1400   -0.0700   -0.1000
   -0.0700   -0.1600   -0.0700


B2 =

   -0.1400    0.1200   -0.0500
   -0.1400   -0.0700   -0.1000
   -0.0700   -0.1600   -0.0700


C =

   -0.3200    0.7400   -0.3800
    1.9700   -0.6100    0.4400
   -2.1900   -1.1500    2.6500
```

Since the constant offsets between the models are equivalent, the resulting VEC(2) model is

$$\begin{aligned}\Delta y_t &= \begin{bmatrix} 0.5 \\ 1 \\ -2 \end{bmatrix} + \begin{bmatrix} -0.14 & 0.12 & -0.05 \\ -0.14 & -0.07 & -0.10 \\ -0.07 & -0.16 & -0.07 \end{bmatrix} \Delta y_{t-1} + \begin{bmatrix} -0.14 & 0.12 & -0.05 \\ -0.14 & -0.07 & -0.10 \\ -0.07 & -0.16 & -0.07 \end{bmatrix} \Delta y_{t-2} \\ &+ \begin{bmatrix} -0.32 & 0.74 & -0.38 \\ 1.97 & -0.61 & 0.44 \\ -2.19 & -1.15 & 2.65 \end{bmatrix} y_{t-1} + \varepsilon_t.\end{aligned}$$

Convert Structural VAR Model to VEC Model Using Lag Operator Polynomials

Consider converting the following structural VAR(2) model to a structural VEC(1) model.

$$\begin{bmatrix} 0.54 & -2.26 \\ 1.83 & 0.86 \end{bmatrix} y_t = \begin{bmatrix} 0.32 & -0.43 \\ -1.31 & 0.34 \end{bmatrix} y_{t-1} + \begin{bmatrix} 0.07 & 0.07 \\ -0.01 & -0.02 \end{bmatrix} y_{t-2} + \varepsilon_t.$$

Specify the autoregressive coefficient matrices A_0, A_1, and A_2.

```
A0 = [0.54 -2.26;
```

```
      1.83  0.86];
A1 = [0.32 -0.43
     -1.31  0.34];
A2 = [0.07  0.07
     -0.01 -0.02];
```

Pack the matrices into separate cells of a 3 dimensional cell vector. Put `A0` into the first cell, `A1` into the second cell, and `A2` into the third cell. Negate the coefficients corresponding to all nonzero lag terms.

```
VARCoeff = {A0; -A1; -A2};
```

Create a lag operator polynomial that encompasses the autoregressive terms in the VAR(2) model.

```
VAR = LagOp(VARCoeff)
```

```
VAR =

    2-D Lag Operator Polynomial:
    ----------------------------
        Coefficients: [Lag-Indexed Cell Array with 3 Non-Zero Coefficients]
                Lags: [0 1 2]
              Degree: 2
           Dimension: 2
```

`VAR` is a `LagOp` lag operator polynomial. `VAR` specifies the VAR(2) model in lag operator notation, as in this equation

$$(A_0 - A_1L - A_2L^2)y_t = \varepsilon_t.$$

L is the lag operator. If you expand the quantity and solve for y_t, then the result is the VAR(2) model in difference-equation notation.

Compute the coefficient matrices of Δy_t and Δy_t, and the error-correction coefficient matrix of the equivalent VEC(1) model.

```
[VEC,C] = var2vec(VAR)
```

```
VEC =
```

```
    2-D Lag Operator Polynomial:
    ----------------------------
        Coefficients: [Lag-Indexed Cell Array with 2 Non-Zero Coefficients]
                Lags: [0 1]
              Degree: 1
           Dimension: 2

C =

   -0.1500    1.9000
   -3.1500   -0.5400
```

`VAR.Coefficients{0}` is A_0, the coefficient matrix of y_t. Subsequent elements in `VAR.Coefficients` correspond to subsequent lags in `VAR.Lags`.

`VEC` is the VEC(1) equivalent of the VAR(2) model. Because the VAR(2) model is structural, the equivalent VEC(1) model is as well. That is, `VEC.Coefficients{0}` is the coefficient of Δy_t, and subsequent elements correspond to subsequent lags in `VEC.Lags`.

Display the VEC model coefficients in difference-equation notation.

```
B0 = VEC.Coefficients{0}
B1 = -VEC.Coefficients{1}
C
```

```
B0 =

    0.5400   -2.2600
    1.8300    0.8600


B1 =

   -0.0700   -0.0700
    0.0100    0.0200


C =

   -0.1500    1.9000
   -3.1500   -0.5400
```

The resulting VEC(1) model is

$$\begin{bmatrix} 0.54 & -2.26 \\ 1.83 & 0.86 \end{bmatrix} \Delta y_t = \begin{bmatrix} -0.07 & -0.07 \\ 0.01 & 0.02 \end{bmatrix} \Delta y_{t-1} + \begin{bmatrix} -0.15 & 1.9 \\ -3.15 & -0.54 \end{bmatrix} y_{t-1} + \varepsilon_t.$$

Alternatively, reflect the lag operator polynomial `VEC` around lag 0 to obtain the difference-equation notation coefficients.

```
DiffEqnCoeffs = reflect(VEC);
B = toCellArray(DiffEqnCoeffs);
B{1} == B0
B{2} == B1
```

```
ans =

     1     1
     1     1


ans =

     1     1
     1     1
```

Both methods produce the same coefficients.

Convert VARMA Model to VEC Model

Approximate the coefficients of the VEC model that represents this stationary and invertible VARMA(8,4) model that is in lag operator form

$$\left\{ \begin{bmatrix} 1 & 0.2 & -0.1 \\ 0.03 & 1 & -0.15 \\ 0.9 & -0.25 & 1 \end{bmatrix} + \begin{bmatrix} 0.5 & -0.2 & -0.1 \\ -0.3 & -0.1 & 0.1 \\ 0.4 & -0.2 & -0.05 \end{bmatrix} L^4 + \begin{bmatrix} 0.05 & -0.02 & -0.01 \\ -0.1 & -0.01 & -0.001 \\ 0.04 & -0.02 & -0.005 \end{bmatrix} L^8 \right\} y_t =$$
$$\left\{ \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} + \begin{bmatrix} -0.02 & 0.03 & 0.3 \\ 0.003 & 0.001 & 0.01 \\ 0.3 & 0.01 & 0.01 \end{bmatrix} L^4 \right\} \varepsilon_t$$

where $y_t = [y_{1t} \quad y_{2t} \quad y_{3t}]'$ and $\varepsilon_t = [\varepsilon_{1t} \quad \varepsilon_{2t} \quad \varepsilon_{3t}]'$.

Create a cell vector containing the VAR coefficient matrices. Start with the coefficient of y_t, and then enter the rest in order by lag. Construct a vector that indicates the degree of the lag term for the corresponding coefficients.

```
VAR0 = {[1 0.2 -0.1; 0.03 1 -0.15; 0.9 -0.25 1],...
    [0.5 -0.2 -0.1; -0.3 -0.1 0.1; 0.4 -0.2 -0.05],...
    [0.05 -0.02 -0.01; -0.1 -0.01 -0.001; 0.04 -0.02 -0.005]};
var0Lags = [0 4 8];
```

Create a cell vector containing the VMA coefficients matrices. Start with the coefficient of ε_t, and then enter the rest in order by lag. Construct a vector that indicates the degree of the lag term for the corresponding coefficients.

```
VMA0 = {eye(3),...
    [-0.02 0.03 0.3; 0.003 0.001 0.01; 0.3 0.01 0.01]};
vma0Lags = [0 4];
```

`arma2ma` requires `LagOp` lag operator polynomials for input arguments that comprise structural VAR or VMA models. Construct separate `LagOp` polynomials that describe the VAR(8) and VMA(4) components of the VARMA(8,4) model.

```
VARLag = LagOp(VAR0,'Lags',var0Lags);
VMALag = LagOp(VMA0,'Lags',vma0Lags);
```

`VARLag` and `VMALag` are `LagOp` lag operator polynomials that describe the VAR and VMA components of the VARMA model.

Convert the VARMA(8,4) model to a VAR(p) model by obtaining the coefficients of the truncated approximation of the infinite-lag polynomial. Set `numLags` to return at most 12 lagged terms.

```
numLags = 12;
VAR = arma2ar(VARLag,VMALag,numLags)
```

```
VAR =

    3-D Lag Operator Polynomial:
    ----------------------------
        Coefficients: [Lag-Indexed Cell Array with 4 Non-Zero Coefficients]
```

```
              Lags: [0 4 8 12]
            Degree: 12
         Dimension: 3
```

`VAR` is a `LagOP` lag operator polynomial. All coefficients except those corresponding to lags 0, 4, 8, and 12 are 3-by-3 matrices of zeros. The coefficients in `VAR.Coefficients` comprise a structural VAR(12) model approximation of the original VARMA(8,4) model.

Compute the coefficients of the VEC(11) model equivalent to the resulting VAR(12) model.

```
[VEC,C] = var2vec(VAR)
```

```
VEC =

    3-D Lag Operator Polynomial:
    ----------------------------
        Coefficients: [Lag-Indexed Cell Array with 12 Non-Zero Coefficients]
                Lags: [0 1 2 3 4 5 6 7 8 9 10 11]
              Degree: 11
           Dimension: 3

C =

   -1.2998   -0.1019    0.5440
    0.3831   -0.8937    0.0603
   -0.9484    0.5068   -1.0876
```

`VEC` is a `LagOp` lag operator polynomial containing the coefficient matrices of the resulting VEC(11) model in `VEC.Coefficients`. `VEC.Coefficients{0}` is the coefficient of Δy_t, `Vec{1}` is the coefficient of Δy_{t-1}, and so on.

Display the nonzero coefficients of the resulting VEC model.

```
lag2Idx = VEC.Lags + 1; % Lags start at 0.  Add 1 to convert to indices.
VecCoeff = toCellArray(VEC);

for j = 1:numel(lag2Idx)
    fprintf('___________Lag %d__________\n',lag2Idx(j) - 1)
    fprintf('%8.3f %8.3f %8.3f \n',VecCoeff{lag2Idx(j)})
    fprintf    ('_____________________________\n')
```

```
end
```

```
____________Lag 0___________
   1.000    0.030    0.900
   0.200    1.000   -0.250
  -0.100   -0.150    1.000

_____________________________
____________Lag 1___________
  -0.300    0.413   -0.048
   0.098    0.106    0.257
   0.444   -0.090   -0.088

_____________________________
____________Lag 2___________
  -0.300    0.413   -0.048
   0.098    0.106    0.257
   0.444   -0.090   -0.088

_____________________________
____________Lag 3___________
  -0.300    0.413   -0.048
   0.098    0.106    0.257
   0.444   -0.090   -0.088

_____________________________
____________Lag 4___________
  -0.051    0.101    0.042
  -0.053    0.007   -0.011
   0.046    0.001   -0.116

_____________________________
____________Lag 5___________
  -0.051    0.101    0.042
  -0.053    0.007   -0.011
   0.046    0.001   -0.116

_____________________________
____________Lag 6___________
  -0.051    0.101    0.042
  -0.053    0.007   -0.011
   0.046    0.001   -0.116

_____________________________
____________Lag 7___________
  -0.051    0.101    0.042
  -0.053    0.007   -0.011
   0.046    0.001   -0.116

_____________________________
____________Lag 8___________
  -0.014   -0.000    0.010
   0.007    0.000    0.018
```

```
    0.034    0.001   -0.002

_____________________________
____________Lag 9____________
   -0.014   -0.000    0.010
    0.007    0.000    0.018
    0.034    0.001   -0.002

_____________________________
____________Lag 10___________
   -0.014   -0.000    0.010
    0.007    0.000    0.018
    0.034    0.001   -0.002

_____________________________
____________Lag 11___________
   -0.014   -0.000    0.010
    0.007    0.000    0.018
    0.034    0.001   -0.002

_____________________________
```

- "Determine Cointegration Rank of VEC Model" on page 7-114

Input Arguments

VAR — VAR(*p*) model coefficients

numeric vector | cell vector of square, numeric matrices | `LagOp` lag operator polynomial object

VAR(*p*) model coefficients, specified as a numeric vector, a cell vector of *n*-by-*n* numeric matrices, or a LagOp lag operator polynomial object.

- For a numeric vector specification:
  - The VAR(*p*) is a univariate time series.
  - `VAR` must be a length *p* numeric vector.
  - `VAR(j)` contains the scalar A_j, the coefficient of the lagged response y_{t-j}.
  - The coefficient of y_t (A_0) is `1`.
- For a cell vector specification:
  - `VAR` must have length *p*, and each cell contains an *n*-by-*n* numeric matrix ($n > 1$).
  - `VAR{j}` must contain A_j, the coefficient matrix of the lag term y_{t-j}.
  - `var2vec` assumes that the coefficient of y_t (A_0) is the *n*-by-*n* identity.

- For a `LagOp` lag operator polynomial specification:
  - `VAR.Degree` must be p.
  - `VAR.Coefficients{0}` is A_0, the coefficient of y_t. All other elements correspond to the coefficients of the subsequent lag terms. For example, `VAR.Coefficients{j}` is the coefficient matrix of y_{t-j}. `VAR.Lags` stores all nonzero lags.
  - To construct a model in reduced form, set `VAR.Coefficients{0}` to `eye(VAR.Dimension)`.

For example, consider converting

$$\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}y_t = \begin{bmatrix}0.1 & 0.2\\ 1 & 0.1\end{bmatrix}y_{t-1} + \begin{bmatrix}-0.1 & 0.01\\ 0.2 & -0.3\end{bmatrix}y_{t-2} + \varepsilon_t$$

to a VEC(1) model. The model is in difference-equation notation. You can convert the model by entering

```
VEC = var2vec({[0.1 0.2; 1 0.1], [-0.1 0.01; 0.2 -0.3]});
```

The VAR(2) model in lag operator notation is

$$\left(\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix} - \begin{bmatrix}0.1 & 0.2\\ 1 & 0.1\end{bmatrix} - \begin{bmatrix}-0.1 & 0.01\\ 0.2 & -0.3\end{bmatrix}L\right)y_t = \varepsilon_t.$$

The coefficient matrices of the lagged responses appear negated compared to the corresponding coefficients in difference-equation notation. To obtain the same result using `LagOp` lag operator polynomials, enter

```
VAR = LagOp({eye(2), -[0.1 0.2; 1 0.1], -[-0.1 0.01; 0.2 -0.3]});
VEC = var2vec(VAR);
```

Output Arguments

`VEC` — VEC(q) model coefficients of differenced responses

numeric vector | cell vector of square, numeric matrices | `LagOp` lag operator polynomial object

VEC(q) model coefficients of differenced responses, returned as a numeric vector, a cell vector of n-by-n numeric matrices, or a LagOp lag operator polynomial object. n is the number of time series in the VAR(p) model.

`VAR` and `VEC` share the same data type and orientation.

`var2vec` converts VAR(p) models to VEC($p - 1$) models. That is:

- If `VAR` is a cell or numeric vector, then `numel(VEC)` is `numel(VAR) - 1`.
- If `VAR` is a `LagOp` lag operator polynomial, then `VEC.Degree` is `VAR.Degree - 1`.

C — Error-correction coefficient

numeric matrix

Error-correction coefficient, returned as an n-by-n numeric matrix. n is the number of time series in the VAR model.

More About

Difference-Equation Notation

A VAR(p) or VEC(q) model written in *difference-equation notation* isolates the present value of the response vector and its structural coefficient matrix on the left side of the equation. The right side of the equation contains the sum of the lagged responses, their coefficient matrices, the present innovation vector, and, for VEC models, the error-correction term.

That is, a VAR(p) model written in difference-equation notation is

$$A_0 y_t = a + A_1 y_{t-1} + A_2 y_{t-2} + \ldots + A_p y_{t-p} + \varepsilon_t.$$

A VEC(q) model written in difference equation notation is

$$B_0 \Delta y_t = b + B_1 \Delta y_{t-1} + B_2 \Delta y_{t-2} + \ldots + B_q \Delta y_{t-q} + C y_{t-1} + \varepsilon_t.$$

For the variable and parameter definitions, see “VAR(p) Model” on page 9-1013 and “VEC(q) Model” on page 9-1014.

Lag Operator Notation

A VAR(p) or VEC(q) model written in *lag-operator notation* positions all response terms to the left side of the equation. The right side of the equation contains the model constant offset vector, the present innovation, and, for VEC models, the error-correction term.

That is, a VAR(p) model written in lag-operator notation is

$$A(L)y_t = a + \varepsilon_t$$

where $A(L) = A_0 - A_1L - A_2L^2 - \ldots - A_pL^p$ and $L^j y_t = y_{t-j}$.

A VEC(q) model written in difference equation notation is

$$B(L)\Delta y_t = b + Cy_{t-1} + \varepsilon_t$$

where $B(L) = B_0 - B_1L - B_2L^2 - \ldots - B_qL^q$.

For the variable and parameter definitions, see "VAR(p) Model" on page 9-1013 and "VEC(q) Model" on page 9-1014.

When comparing lag operator notation to difference-equation notation, the signs of the lag terms are opposites. For more details, see "Lag Operator Notation" on page 1-22.

VAR(p) Model

A *VAR(p) model* is a multivariate, autoregressive time series model that has this general form:

$$A_0 y_t = a + A_1 y_{t-1} + A_2 y_{t-2} + \ldots + A_p y_{t-p} + \varepsilon_t.$$

- y_t is an n-dimensional time series.
- A_0 is the n-by-n invertible structural coefficient matrix. For models in *reduced form*, $A_0 = I_n$, which is the n-dimensional identity matrix.
- a is an n-dimensional vector of constant offsets.
- A_j is the n-by-n coefficient matrix of y_{t-j}, $j = 1,\ldots,p$.

- ε_t is an n-dimensional innovations series. The innovations are serially uncorrelated, and have a multivariate normal distribution with mean 0 and n-by-n covariance matrix Σ.

VEC(*q*) Model

A *VEC(q) model* is a multivariate, autoregressive time series model that has this general form:

$$B_0\Delta y_t = b + B_1\Delta y_{t-1} + B_2\Delta y_{t-2} + \ldots + B_q\Delta y_{t-q} + Cy_{t-1} + \varepsilon_t.$$

- y_t is an n-dimensional time series.
- Δ is the first difference operator, that is, $\Delta y_t = y_t - y_{t-1}$.
- B_0 is the n-by-n invertible structural coefficient matrix. For models in *reduced form*, $B_0 = I_n$, which is the n-dimensional identity matrix.
- b is an n-dimensional vector of constant offsets.
- B_j is the n-by-n coefficient matrix of Δy_{t-j}, $j = 1,\ldots,q$.
- ε_t is an n-dimensional innovations series. The innovations are serially uncorrelated, and have a multivariate normal distribution with mean 0 and n-by-n covariance matrix Σ.
- C is the n-by-n error-correction or impact coefficient matrix.

Tips

- To accommodate structural VAR models, specify the input argument `VAR` as a LagOp lag operator polynomial.
- To access the cell vector of the lag operator polynomial coefficients of the output argument `VEC`, enter `toCellArray(VEC)`.
- To convert the model coefficients of the output argument from lag operator notation to the model coefficients in difference-equation notation, enter

  ```
  VECDEN = toCellArray(reflect(VEC));
  ```

  `VECDEN` is a cell vector containing p coefficients corresponding to the differenced response terms in `VEC.Lags` in difference-equation notation. The first element is the coefficient of Δy_t, the second element is the coefficient of Δy_{t-1}, and so on.
- Consider converting a VAR(p) model to a VEC(q) model. If the error-correction coefficient matrix (`C`) has:

  - Rank zero, then the converted VEC model is a stable VAR($p - 1$) model in terms of Δy_t.
  - Full rank, then the VAR(p) model is stable (i.e., has no unit roots) [2].
  - Rank r, such that $0 < r < n$, then the stable VEC model has r cointegrating relations.
- The constant offset of the converted VEC model is the same as the constant offset of the VAR model.

Algorithms

- `var2vec` does not impose stability requirements on the coefficients. To check for stability, use isStable.

  `isStable` requires a `LagOp` lag operator polynomial as an input argument. For example, to check whether `VAR`, the cell array of n-by-n numeric matrices, composes a stable time series, enter

  ```
  varLagOp = LagOp([eye(n) VAR]);
  isStable(varLagOp)
  ```

  A `0` indicates that the polynomial is not stable.

- "Lag Operator Notation" on page 1-22
- "Vector Autoregressive (VAR) Models" on page 7-3
- "Cointegration and Error Correction Analysis" on page 7-108

References

[1] Hamilton, J. D. *Time Series Analysis*. Princeton, NJ: Princeton University Press, 1994.

[2] Lutkepohl, H. "New Introduction to Multiple Time Series Analysis." Springer-Verlag, 2007.

See Also

`arma2ar` | `arma2ma` | isStable | LagOp | toCellArray | `vec2var` | `vgxvarx`

Introduced in R2015b

vartovec

Vector autoregression (VAR) to vector error-correction model (VEC)

Syntax

```
[VEC,C] = vartovec(VAR)
```

Description

Note: `vartovec` will be removed in a future release. Use `var2vec` instead.

Given a vector autoregression (VAR) model, `[VEC,C] = vartovec(VAR)` converts VAR to an equivalent vector error-correction (VEC) model. A VAR(p) model of a time series $y(t)$ has the form:

$$A_0 y(t) = A_1 y(t-1) + ... + A_p y(t-p) + \varepsilon(t)$$

The equivalent VEC(q) model, with $q = p - 1$, has the form:

$$B_0 z(t) = B_1 z(t-1) + ... + B_q z(t-q) + C y(t-1) + \varepsilon(t)$$

where $z(t) = y(t) - y(t-1)$ and C is the error-correction coefficient.

Input Arguments

VAR

The `VAR`(p) model to be converted to an equivalent `VEC`(q) model, with $q = p - 1$. VAR is specified by a (p + 1)-element cell vector of square matrices {`A0 A1 ... Ap`} associated with coefficients at lags 0, 1, ..., p. To represent a univariate model, `VAR` may be specified

as a double-precision vector. Alternatively, VAR may be specified as a LagOp object or a vgxset object.

Output Arguments

VEC

The VEC representation of the input VAR model. The data type and orientation of VEC is consistent with that of VAR

C

The error-correction coefficient. C is a square matrix the same size as the coefficients of the associated VEC.

Examples

Convert a VAR Model to a VEC Model

Specify a VAR(2) model of time series y_t:

$$y_t = A_1 y_{t-1} + A_2 y_{t-2} + \varepsilon_t$$

The coefficients are:

$$A_1 = \begin{bmatrix} -0.1 & 0.3 \\ 0.2 & -0.1 \end{bmatrix}$$

$$A_2 = \begin{bmatrix} -0.2 & 0.8 \\ -0.7 & -0.4 \end{bmatrix}$$

Enter the coefficients from the difference equation directly into a cell array:

```
VAR = {eye(2)  [-0.1 0.3 ; 0.2 -0.1] ...
```

```
      [-0.2 0.8 ; -0.7 -0.4]};
```

Use `vartovec` to convert the VAR(2) model to an equivalent VEC(1) model:

```
[VEC, C] = vartovec(VAR);
```

```
Warning: VARTOVEC will be removed in a future release. Use VAR2VEC instead.
```

Since the original VAR model was specified as a cell array, the VEC model is also a cell array. The error correction coefficient argument is a matrix.

You can express the same VAR(2) model as a lag operator polynomial:

$$\left(I - A_1 L - A_2 L^2\right) y_t = \varepsilon_t$$

Specify the model with the `LagOp` constructor:

```
VAR_LAG = LagOp({eye(2)  [0.1 -0.3 ; -0.2 0.1] ...
                [0.2 -0.8 ; 0.7 0.4]});
```

Use `vartovec` to convert the VAR(2) model to an equivalent VEC(1) model:

```
[VEC_LAG, C_LAG] = vartovec(VAR_LAG)
```

```
Warning: VARTOVEC will be removed in a future release. Use VAR2VEC instead.

VEC_LAG =

    2-D Lag Operator Polynomial:
    ----------------------------
        Coefficients: [Lag-Indexed Cell Array with 2 Non-Zero Coefficients]
                Lags: [0 1]
              Degree: 1
           Dimension: 2

C_LAG =

   -1.3000    1.1000
   -0.5000   -1.5000
```

Since the input model is a lag operator polynomial the output model is also is a lag operator polynomial. See “Specify Lag Operator Polynomials” on page 2-11 for more information on lag operator polynomials.

More About

Algorithms

- Written as a polynomial in the lag operator $Ly(t) = y(t-1)$, a VAR(p) model has the form:

  $$\left(A_0 - A_1 L - \ldots - A_p L^p\right) y(t) = A(L)\, y(t) = \varepsilon(t)$$

  The equivalent VEC(q) model has the form:

  $$\left(B_0 - B_1 L - \ldots - B_q L^q\right) z(t) = B(L)\, z(t) = C y(t-1) + \varepsilon(t)$$

  Thus, if `VAR` is specified as a `LagOp` object `A`, coefficients of lagged values of $y(t)$ must be represented by the opposite of their values in standard difference-equation form, and the output `VEC` will follow a similar sign convention
- If `VAR` is specified as a `vgxset` object, the conversion involves only the `AR0`, `AR`, and `nAR` components of the model. Other model components are unaffected.

References

[1] Hamilton, J. D. "Time Series Analysis." Princeton, NJ: Princeton University Press, 1994.

[2] Lutkepohl, H. "New Introduction to Multiple Time Series Analysis." Springer-Verlag, 2007.

See Also

`vec2var` | `var2vec` | LagOp | `vgxset`

Introduced in R2011a

vec2var

Convert VEC model to VAR model

Econometrics Toolbox multivariate time series model functions such as `vgxsim`, `vgxpred`, and `armairf` are appropriate for vector autoregression (VAR) models. To simulate, forecast, or generate impulse responses from a vector error-correction (VEC) model using `vgxsim`, `vgxpred`, or `armairf`, respectively, convert the VEC model to its equivalent VAR model representation.

Syntax

```
VAR = vec2var(VEC,C)
```

Description

`VAR = vec2var(VEC,C)` returns the coefficient matrices (`VAR`) of the vector autoregressive model equivalent to the vector error-correction model with coefficient matrices (`VEC`). If the number of lags in the input vector error-correction model is q, then the number of lags in the output vector error-correction model is $p = q + 1$.

Examples

Convert VEC Model to VAR Model Using Cell Arrays

Consider converting the following VEC(2) model to a VAR(3) model.

$$\begin{aligned}\Delta y_t &= \begin{bmatrix} 0.5 \\ 1 \\ -2 \end{bmatrix} + \begin{bmatrix} -0.14 & 0.12 & -0.05 \\ -0.14 & -0.07 & -0.10 \\ -0.07 & -0.16 & -0.07 \end{bmatrix} \Delta y_{t-1} + \begin{bmatrix} -0.14 & 0.12 & -0.05 \\ -0.14 & -0.07 & -0.10 \\ -0.07 & -0.16 & -0.07 \end{bmatrix} \Delta y_{t-2} \\ &+ \begin{bmatrix} -0.32 & 0.74 & -0.38 \\ 1.97 & -0.61 & 0.44 \\ -2.19 & -1.15 & 2.65 \end{bmatrix} y_{t-1} + \varepsilon_t .\end{aligned}$$

Specify the coefficient matrices (B_1 and B_2) of Δy_{t-1} and Δy_{t-2}, and the error-correction coefficient C.

```
B1 = [-0.14   0.12 -0.05;
      -0.14 -0.07 -0.10;
      -0.07  -0.16 -0.07];
B2 = [-0.14   0.12 -0.05;
      -0.14 -0.07 -0.10;
      -0.07  -0.16 -0.07];
C  = [-0.32  0.74 -0.38;
       1.97 -0.61  0.44;
     - 2.19 -1.15  2.65];
```

Pack the matrices into separate cells of a 2-dimensional cell vector. Put `B1` into the first cell and `B2` into the second cell.

```
VEC = {B1 B2};
```

Compute the coefficient matrices of the equivalent VAR(3) model.

```
VAR = vec2var(VEC,C);
size(VAR)
```

```
ans =

     1     3
```

The specification of a cell array of matrices for the input argument indicates that the VEC(2) model is in reduced form, and `VEC{1}` is the coefficient of Δy_{t-1}. Subsequent elements correspond to subsequent lags.

`VAR` is a 1-by-3 cell vector of 3-by-3 coefficient matrices for the VAR(3) equivalent of the VEC(2) model. Because the VEC(2) model is in reduced form, the equivalent VAR(3) model is as well. That is, `VAR{1}` is the coefficient of y_{t-1}, and subsequent elements correspond to subsequent lags. The orientation of `VAR` corresponds to the orientation of `VEC`.

Display the VAR(3) model coefficients.

```
A1 = VAR{1}
A2 = VAR{2}
A3 = VAR{3}
```

```
A1 =
```

```
    0.5400    0.8600   -0.4300
    1.8300    0.3200    0.3400
   -2.2600   -1.3100    3.5800

A2 =

     0     0     0
     0     0     0
     0     0     0

A3 =

    0.1400   -0.1200    0.0500
    0.1400    0.0700    0.1000
    0.0700    0.1600    0.0700
```

Since the constant offsets between the models are equivalent, the resulting VAR(3) model is

$$y_t = \begin{bmatrix} 0.5 \\ 1 \\ -2 \end{bmatrix} + \begin{bmatrix} 0.54 & 0.86 & -0.43 \\ 1.83 & 0.32 & 0.34 \\ -2.26 & -1.31 & 3.58 \end{bmatrix} y_{t-1} + \begin{bmatrix} 0.14 & -0.12 & 0.05 \\ 0.14 & 0.07 & 0.10 \\ 0.07 & 0.16 & 0.07 \end{bmatrix} y_{t-3} + \varepsilon_t.$$

Convert Structural VEC Model to VAR Model Using Lag Operator Polynomials

Consider converting the following structural VEC(1) model to a structural VAR(2) model.

$$\begin{bmatrix} 0.54 & -2.26 \\ 1.83 & 0.86 \end{bmatrix} \Delta y_t = \begin{bmatrix} -0.07 & -0.07 \\ 0.01 & 0.02 \end{bmatrix} \Delta y_{t-1} + \begin{bmatrix} -0.15 & 1.9 \\ -3.15 & -0.54 \end{bmatrix} y_{t-1} + \varepsilon_t.$$

Specify the coefficient matrices B_0 and B_1, and the error-correction coefficient C.

```
B0 = [0.54 -2.26;
      1.83  0.86];
B1 = [-0.07  -0.07
     0.01 0.02];
C = [-0.15  1.9;
     -3.15  -0.54];
```

Pack the matrices into separate cells of a 3-dimensional cell vector. Put `B0` into the first cell and `B1` into the second cell. Negate the coefficients corresponding to all nonzero differenced lag terms.

```
VECCoeff = {B0; -B1};
```

Create a lag operator polynomial that encompasses the autoregressive terms in the VEC(2) model.

```
VEC = LagOp(VECCoeff)
```

```
VEC =

    2-D Lag Operator Polynomial:
    ----------------------------
        Coefficients: [Lag-Indexed Cell Array with 2 Non-Zero Coefficients]
                Lags: [0 1]
              Degree: 1
           Dimension: 2
```

`VEC` is a `LagOp` lag operator polynomial, and specifies the autoregressive lag operator polynomial in this equation

$$(B_0 - B_1 L)\Delta y_t = C y_{t-1} + \varepsilon_t.$$

L is the lag operator. If you expand the quantity and solve for Δy_t, then the result is the VAR(2) model in difference-equation notation.

Compute the coefficient matrices of the equivalent VAR(2) model.

```
VAR = vec2var(VEC,C)
```

```
VAR =

    2-D Lag Operator Polynomial:
    ----------------------------
        Coefficients: [Lag-Indexed Cell Array with 3 Non-Zero Coefficients]
                Lags: [0 1 2]
              Degree: 2
           Dimension: 2
```

`VEC.Coefficients{0}` is A_0, the coefficient matrix of y_t. Subsequent elements in `VAR.Coefficients` correspond to subsequent lags in `VEC.Lags`.

VAR is the VAR(2) equivalent of the VEC(1) model. Becayse the VEC(1) model is structural, the equivalent VAR(2) is as well. That is, VAR.Coefficients{0} is the coefficient of y_t, and subsequent elements correspond to subsequent lags in VAR.Lags.

Display the VAR(2) model coefficients in difference-equation notation.

```
A0 = VAR.Coefficients{0}
A1 = -VAR.Coefficients{1}
A2 = -VAR.Coefficients{2}
```

```
A0 =

    0.5400   -2.2600
    1.8300    0.8600


A1 =

    0.3200   -0.4300
   -1.3100    0.3400


A2 =

    0.0700    0.0700
   -0.0100   -0.0200
```

The resulting VAR(3) model is

$$\begin{bmatrix} 0.54 & -2.26 \\ 1.83 & 0.86 \end{bmatrix} y_t = \begin{bmatrix} 0.32 & -0.43 \\ -1.31 & 0.34 \end{bmatrix} y_{t-1} + \begin{bmatrix} 0.07 & 0.07 \\ -0.01 & -0.02 \end{bmatrix} y_{t-2} + \varepsilon_t.$$

Alternatively, reflect the lag operator polynomial VAR around lag 0 to obtain the difference-equation notation coefficients.

```
DiffEqnCoeffs = reflect(VAR);
A = toCellArray(DiffEqnCoeffs);
A{1} == A0
A{2} == A1
A{3} == A2
```

```
ans =

     1     1
     1     1


ans =

     1     1
     1     1


ans =

     1     1
     1     1
```

Both methods produce the same coefficients.

Convert Structural VEC model to VMA Model

Approximate the coefficients of the structural VMA model that represents the structural VEC(8) model

$$\left\{\begin{bmatrix} 1 & 0.2 & -0.1 \\ 0.03 & 1 & -0.15 \\ 0.9 & -0.25 & 1 \end{bmatrix} + \begin{bmatrix} 0.5 & -0.2 & -0.1 \\ -0.3 & -0.1 & 0.1 \\ 0.4 & -0.2 & -0.05 \end{bmatrix} L^4 + \begin{bmatrix} 0.05 & -0.02 & -0.01 \\ -0.1 & -0.01 & -0.001 \\ 0.04 & -0.02 & -0.005 \end{bmatrix} L^8 \right\} \Delta y_t =$$
$$\begin{bmatrix} -0.02 & 0.03 & 0.3 \\ 0.05 & 0.1 & 0.01 \\ 0.3 & 0.01 & 0.01 \end{bmatrix} y_{t-1} + \varepsilon_t$$

where $\Delta y_t = [\Delta y_{t,1} \quad \Delta y_{t,2} \quad \Delta y_{t,3}]'$, $\varepsilon_t = [\varepsilon_{1t} \quad \varepsilon_{2t} \quad \varepsilon_{3t}]'$, and, for j = 1,2, and 3, $\Delta y_{t,j} = y_{t,j} - y_{t-1,j}$.

Create a cell vector containing the VEC(8) model coefficient matrices. Start with the coefficient of Δy_t, and then enter the rest in order by lag. Construct a vector that indicates the degree of the lag term for the corresponding coefficients.

```
VEC0 = {[1 0.2 -0.1; 0.03 1 -0.15; 0.9 -0.25 1],...
    [0.5 -0.2 -0.1; -0.3 -0.1 0.1; 0.4 -0.2 -0.05],...
    [0.05 -0.02 -0.01; -0.1 -0.01 -0.001; 0.04 -0.02 -0.005]};
```

```
vec0Lags = [0 4 8];
C = [-0.02 0.03 0.3; 0.05 0.1 0.01; 0.3 0.01 0.01];
```

`vec2var` requires a `LagOp` lag operator polynomial for an input argument that comprises a structural VEC(8) model. Construct a `LagOp` lag operator polynomial that describes the VEC(8) model autoregressive coefficient matrix component (i.e., the coefficients of Δy_t and its lags).

```
VECLag = LagOp(VEC0,'Lags',vec0Lags);
```

`VECLag` is a `LagOp` lag operator polynomial that describes the autoregressive component of the VEC(8) model.

Compute the coefficients of the VAR(9) model equivalent to the VEC(8) model.

```
VAR = vec2var(VECLag,C)
```

```
VAR =

    3-D Lag Operator Polynomial:
    ----------------------------
        Coefficients: [Lag-Indexed Cell Array with 6 Non-Zero Coefficients]
                Lags: [0 1 4 5 8 9]
              Degree: 9
           Dimension: 3
```

`VAR` is a `LagOp` lag operator polynomial. All coefficients except those corresponding to lags 0, 1, 4, 5, 8, and 9 are 3-by-3 matrices of zeros. The coefficients in `VAR` comprise a stable, structural VAR(9) model equivalent to the original VEC(8) model. The model is stable because the error-correction coefficient has full rank.

Compute the coefficients of the VMA model approximation to the resulting VAR(9) model. Set `numLags` to return at most 12 lags.

```
numLags = 12;
VMA = arma2ma(VAR,[],numLags);
```

`VMA` is a `LagOp` lag operator polynomial containing the coefficient matrices of the resulting VMA(12) model in `VMA.Coefficients`. `VMA{0}` is the coefficient of ε_t, `VMA{1}` is the coefficient of ε_{t-1}, and so on.

- “Simulate and Forecast a VEC Model” on page 7-129

- "Generate VEC Model Impulse Responses" on page 7-138

Input Arguments

`VEC` — VEC(*q*) model coefficients of differenced responses

numeric vector | cell vector of square, numeric matrices | `LagOp` lag operator polynomial object

VEC(*q*) model coefficients of differenced responses, specified as a numeric vector, a cell vector of *n*-by-*n* numeric matrices, or a LagOp lag operator polynomial object.

- For a numeric vector specification:
  - The VEC(*q*) is a univariate time series.
  - `VEC` must be a length *q* numeric vector.
  - `VEC(j)` contains the scalar B_j, the coefficient of the lagged difference Δy_{t-j}.
  - The coefficient of Δy_t (B_0) is `1`.
- For a cell vector specification:
  - `VEC` must have length *q*, and each cell contains an *n*-by-*n* numeric matrix ($n > 1$).
  - `VEC{j}` must contain B_j, the coefficient matrix of the lag term Δy_{t-j}.
  - `vec2var` assumes that the coefficient of Δy_t (B_0) is the *n*-by-*n* identity.
- For a `LagOp` lag operator polynomial specification:
  - `VEC.Degree` must be *q*.
  - `VEC.Coefficients{0}` is B_0, the coefficient of Δy_t. All other elements correspond to the coefficients of the subsequent lagged, differenced terms. For example, `VEC.Coefficients{j}` is the coefficient matrix of Δy_{t-j}. `VEC.Lags` stores all nonzero lags.
  - To construct a model in reduced form, set `VEC.Coefficients{0}` to `eye(VEC.Dimension)`.

For example, consider converting

$$\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}\Delta y_t = \begin{bmatrix}0.1 & 0.2\\ 1 & 0.1\end{bmatrix}\Delta y_{t-1} + \begin{bmatrix}-0.1 & 0.01\\ 0.2 & -0.3\end{bmatrix}\Delta y_{t-2} + \begin{bmatrix}0.5 & 0\\ -0.1 & 1\end{bmatrix}y_{t-1} + \varepsilon_t$$

to a VAR(3) model. The model is in difference-equation notation. You can convert the model by entering

```
VAR = vec2var({[0.1 0.2; 1 0.1], -[-0.1 0.01; 0.2 -0.3]},[0.5 0; -0.1 1]);
```

The VEC(2) model in lag operator notation is

$$\left(\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}-\begin{bmatrix}0.1 & 0.2\\ 1 & 0.1\end{bmatrix}L-\begin{bmatrix}-0.1 & 0.01\\ 0.2 & -0.3\end{bmatrix}L^2\right)\Delta y_t=\begin{bmatrix}0.5 & 0\\ -0.1 & 1\end{bmatrix}y_{t-1}+\varepsilon_t$$

The AR coefficient matrices of the lagged responses appear negated compared to the corresponding coefficients in difference-equation notation. To obtain the same result using `LagOP` lag operator polynomials, enter

```
VEC = LagOp({eye(2), -[0.1 0.2; 1 0.1], -[-0.1 0.01; 0.2 -0.3]});
C = [0.5 0; -0.1 1];
VAR = vec2var(VEC,C);
```

`C` — Error-correction coefficient

numeric matrix

Error-correction coefficient, specified as an n-by-n numeric matrix. n is the number of time series in the VEC model. The dimensions of `C` and the matrices composing `VEC` must be equivalent.

Data Types: `double`

Output Arguments

`VAR` — VAR(p) model coefficients

cell vector of square, numeric matrices | `LagOp` lag operator polynomial object | numeric vector

VAR(p) model coefficients, returned as a a numeric vector, cell vector of n-by-n numeric matrices, or a LagOp lag operator polynomial object. n is the number of time series in the VEC model.

`VEC` and `VAR` share the same data type and orientation.

`vec2var` converts VEC(q) models to VAR(q + 1) models. That is:

- If `VEC` is a cell or numeric vector, then `numel(VAR)` is `numel(VEC) + 1`.

- If `VEC` is a `LagOp` lag operator polynomial, then `VAR.Degree` is `VEC.Degree + 1`.

More About

Difference-Equation Notation

A VAR(p) or VEC(q) model written in *difference-equation notation* isolates the present value of the response vector and its structural coefficient matrix on the left side of the equation. The right side of the equation contains the sum of the lagged responses, their coefficient matrices, the present innovation vector, and, for VEC models, the error-correction term.

That is, a VAR(p) model written in difference-equation notation is

$$A_0 y_t = a + A_1 y_{t-1} + A_2 y_{t-2} + \ldots + A_p y_{t-p} + \varepsilon_t.$$

A VEC(q) model written in difference equation notation is

$$B_0 \Delta y_t = b + B_1 \Delta y_{t-1} + B_2 \Delta y_{t-2} + \ldots + B_q \Delta y_{t-q} + C y_{t-1} + \varepsilon_t.$$

For the variable and parameter definitions, see "VAR(p) Model" on page 9-1013 and "VEC(q) Model" on page 9-1014.

Lag Operator Notation

A VAR(p) or VEC(q) model written in *lag-operator notation* positions all response terms to the left side of the equation. The right side of the equation contains the model constant offset vector, the present innovation, and, for VEC models, the error-correction term.

That is, a VAR(p) model written in lag-operator notation is

$$A(L) y_t = a + \varepsilon_t$$

where $A(L) = A_0 - A_1 L - A_2 L^2 - \ldots - A_p L^p$ and $L^j y_t = y_{t-j}$.

A VEC(q) model written in difference equation notation is

$$B(L) \Delta y_t = b + C y_{t-1} + \varepsilon_t$$

where $B(L) = B_0 - B_1L - B_2L^2 - ... - B_qL^q$.

For the variable and parameter definitions, see "VAR(*p*) Model" on page 9-1013 and "VEC(*q*) Model" on page 9-1014.

When comparing lag operator notation to difference-equation notation, the signs of the lag terms are opposites. For more details, see "Lag Operator Notation" on page 1-22.

VAR(*p*) Model

A *VAR(p) model* is a multivariate, autoregressive time series model that has this general form:

$$A_0y_t = a + A_1y_{t-1} + A_2y_{t-2} + ... + A_py_{t-p} + \varepsilon_t.$$

- y_t is an n-dimensional time series.
- A_0 is the n-by-n invertible structural coefficient matrix. For models in *reduced form*, $A_0 = I_n$, which is the n-dimensional identity matrix.
- a is an n-dimensional vector of constant offsets.
- A_j is the n-by-n coefficient matrix of y_{t-j}, $j = 1,...,p$.
- ε_t is an n-dimensional innovations series. The innovations are serially uncorrelated, and have a multivariate normal distribution with mean 0 and n-by-n covariance matrix Σ.

VEC(*q*) Model

A *VEC(q) model* is a multivariate, autoregressive time series model that has this general form:

$$B_0\Delta y_t = b + B_1\Delta y_{t-1} + B_2\Delta y_{t-2} + ... + B_q\Delta y_{t-q} + Cy_{t-1} + \varepsilon_t.$$

- y_t is an n-dimensional time series.
- Δ is the first difference operator, that is, $\Delta y_t = y_t - y_{t-1}$.
- B_0 is the n-by-n invertible structural coefficient matrix. For models in *reduced form*, $B_0 = I_n$, which is the n-dimensional identity matrix.
- b is an n-dimensional vector of constant offsets.

- B_j is the n-by-n coefficient matrix of Δy_{t-j}, $j = 1,...,q$.
- ε_t is an n-dimensional innovations series. The innovations are serially uncorrelated, and have a multivariate normal distribution with mean 0 and n-by-n covariance matrix Σ.
- C is the n-by-n error-correction or impact coefficient matrix.

Tips

- To accommodate structural VEC models, specify the input argument `VEC` as a LagOp lag operator polynomial.
- To access the cell vector of the lag operator polynomial coefficients of the output argument `VAR`, enter `toCellArray(VAR)`.
- To convert the model coefficients of the output argument from lag operator notation to the model coefficients in difference-equation notation, enter

  ```
  VARDEN = toCellArray(reflect(VAR));
  ```

  `VARDEN` is a cell vector containing $q + 1$ coefficients corresponding to the response terms in `VAR.Lags` in difference-equation notation. The first element is the coefficient of y_t, the second element is the coefficient of y_{t-1}, and so on.
- The constant offset of the converted VAR model is the same as the constant offset of the VEC model.

Algorithms

- `vec2var` does not impose stability requirements on the coefficients. To check for stability, use isStable.

  `isStable` requires a `LagOp` lag operator polynomial as input. For example, to check whether `VAR`, the cell array of *n*-by*n* numeric matrices, composes a stable time series, enter

  ```
  varLagOp = LagOp([eye(n) var]);
  isStable(varLagOp)
  ```

  A `0` indicates that the polynomial is not stable. If `VAR` is a `LagOp` lag operator polynomial, then pass it to `isStable`.

- "Lag Operator Notation" on page 1-22
- "Vector Autoregressive (VAR) Models" on page 7-3
- "Cointegration and Error Correction Analysis" on page 7-108

References

[1] Hamilton, J. D. *Time Series Analysis*. Princeton, NJ: Princeton University Press, 1994.

[2] Lutkepohl, H. "New Introduction to Multiple Time Series Analysis." Springer-Verlag, 2007.

See Also

`arma2ar` | `arma2ma` | `armairf` | isStable | LagOp | toCellArray | `var2vec` | `vgxvarx`

Introduced in R2015b

vectovar

Vector error-correction (VEC) to vector autoregression (VAR)

Syntax

```
VAR = vectovar(VEC,C)
```

Description

Note: `vectovar` will be removed in a future release. Use `vec2var` instead.

Given a vector error-correction (VEC) model, `VAR = vectovar(VEC,C)` converts VEC to an equivalent vector autoregression (VAR) model. A VEC(q) model of a time series $y(t)$ has the form:

$$B_0 z(t) = B_1 z(t-1) + ... + B_q z(t-q) + C y(t-1) + \varepsilon(t)$$

where $z(t) = y(t) - y(t-1)$ and C is the error-correction coefficient. The equivalent VAR(p) model with $p = q + 1$ has the form:

$$A_0 y(t) = A_1 y(t-1) + ... + A_p y(t-p) + \varepsilon(t)$$

Input Arguments

VEC

The VEC(q) model to be converted to an equivalent VAR(p) model, with $p = q + 1$. VEC is specified by a (q + 1)-element cell vector of square matrices {`B0 B1 ... Bq`} associated with coefficients at lags 0, 1, ..., q. To represent a univariate model, VEC may be specified as a double-precision vector. Alternatively, VEC may be specified as a `LagOp` object or a `vgxset` object.

C

The error-correction coefficient. `C` is a square matrix the same size as the coefficients of the associated `VEC`.

Output Arguments

VAR

The VAR representation of the input VEC model. The data type and orientation of VAR is consistent with that of VEC.

Examples

Convert a VEC Model to a VAR Model

Specify a VEC(1) model of time series y_t:

$$\Delta y_t = B_1 \Delta y_{t-1} + C y_{t-1} + \varepsilon_t$$

The first-order lag coefficient is:

$$B_1 = \begin{bmatrix} 0.2 & -0.8 \\ 0.7 & 0.4 \end{bmatrix}$$

The error-correction coefficient is:

$$C = \begin{bmatrix} -1.3 & 1.1 \\ -0.5 & -1.5 \end{bmatrix}$$

Enter the coefficients from the difference equation (including the identity for B_0) directly into a cell array:

```
VEC = {eye(2)  [0.2 -0.8 ; 0.7 0.4]};
```

```
C   = [-1.3 1.1 ; -0.5 -1.5];
```

Use `vectovar` to convert the VEC(1) model to an equivalent VAR(2) model:

```
VAR = vectovar(VEC, C);
```

```
Warning: VECTOVAR will be removed in a future release. Use VEC2VAR instead.
```

Since the original VEC model was specified as a cell array, the VAR model is also a cell array. The output cell array contains A_0, A_1, and A_2:

```
A0 = VAR{1}
A1 = VAR{2}
A2 = VAR{3}
```

```
A0 =

     1     0
     0     1


A1 =

   -0.1000    0.3000
    0.2000   -0.1000


A2 =

   -0.2000    0.8000
   -0.7000   -0.4000
```

You can express the same VEC(1) model as a lag operator polynomial:

$$(I - B_1 L)\Delta y_t = C y_{t-1} + \varepsilon_t$$

To specify the VEC(1) model as a lag operator, use the `LagOp` constructor to create a lag operator polynomial object:

```
vec = LagOp({eye(2) [-0.2 0.8 ; -0.7 -0.4]});
```

Use `vectovar` to convert the VEC(1) model to an equivalent VAR(2) model:

Since the input model is a lag operator polynomial, so is the output model. The output model uses the same sign convention as the input model. Obtain the coefficient associated with the first and second lags of the VAR model by lag-based indexing:

```
VARFirstCoeff = var.Coefficients{1}
VARSecondCoeff = var.Coefficients{2}
```

```
VARFirstCoeff =

    0.1000   -0.3000
   -0.2000    0.1000


VARSecondCoeff =

    0.2000   -0.8000
    0.7000    0.4000
```

See “Specify Lag Operator Polynomials” on page 2-11 for more information on lag operator polynomials.

More About

Algorithms

- Written as a polynomial in the lag operator $Ly(t) = y(t-1)$, a VEC(q) model has the form:

  $$\left(B_0 - B_1L - \ldots - B_qL^q\right)z(t) = B(L)z(t) = Cy(t-1) + \varepsilon(t)$$

  The equivalent VAR(p) model has the form:

  $$\left(A_0 - A_1L - \ldots - A_pL^p\right)y(t) = A(L)y(t) = \varepsilon(t)$$

  Thus, if `VEC` is specified as a `LagOp` object `B`, coefficients of lagged values of $z(t)$ must be represented by the opposite of their values in standard difference-equation form. The output, `VAR`, will follow a similar sign convention.

- If `VEC` is specified as a `vgxset` object, the conversion involves only the `AR0`, `AR`, and `nAR` components of the model. Other model components are unaffected.

References

[1] Hamilton, J. D. "Time Series Analysis." Princeton, NJ: Princeton University Press, 1994.

[2] Lutkepohl, H. "New Introduction to Multiple Time Series Analysis." Springer-Verlag, 2007.

See Also

`var2vec` | `vec2var` | LagOp | `vgxset`

Introduced in R2011a

vgxar

Convert VARMA model to VAR model

Syntax

```
SpecAR = vgxar(Spec)

SpecAR = vgxar(Spec,nAR,ARlag,Cutoff)
```

Description

`vgxar` converts a VARMA model into a pure vector autoregressive (VAR) model. This function works only for VARMA models and does not handle exogenous variables (VARMAX models).

Required Input Argument

`Spec`	A multivariate time series specification structure for an n-dimensional VARMA time series process, as created by `vgxset`.

Optional Input Arguments

`nAR`	Number of AR lags for the output specification structure. `vgxar` truncates an infinite-order VAR model to `nAR` lags. If specific AR lags are not given by `ARlag`, the lags are `1:nAR`. To use `ARlag`, set `nAR` to `[]` or to the number of specific lags.
`ARlag`	A positive integer vector of specific AR lags for the output specification structure. `ARlag` must be of length `nAR`, unless `nAR` is `[]`.
`Cutoff`	The cutoff for the infinity norm below which trailing lags are removed. The default is `0`, which does not remove any lags and uses the values for `nAR` and `ARlag`.

If neither nAR nor ARlag is specified, vgxar uses the maximum lags of the AR or MA lags of the input Spec.

Note: If a large number of lags is needed to form a pure VAR representation (with unit roots close to 1), a large number of initial values is also needed for propagation.

Output Arguments

SpecAR	A transformed multivariate time series specification structure that consists of a pure vector autoregressive (VAR) model with nAR lags. Logical indicators for model parameter estimation ("solve" information) in Spec are not passed on to SpecAR.

Examples

Convert a VARMA Model to a VAR Model

Start with a 2-dimensional VARMA(2, 2) specification structure in Spec:

```
load Data_VARMA22
```

Convert Spec into a pure VAR(2) model in SpecAR:

```
SpecAR = vgxar(Spec);
```

Display the original specification structure in Spec and compare with the new specification structure in SpecAR:

```
vgxdisp(Spec, SpecAR)
```

```
  Model 1: 2-D VARMA(2,2) with No Additive Constant
           Conditional mean is AR-stable and is MA-invertible
  Model 2: 2-D VAR(2) with No Additive Constant
           Conditional mean is AR-stable and is MA-invertible
       Parameter        Model 1        Model 2
  -------------- -------------- --------------
      AR(1)(1,1)       0.373935       0.579177
           (1,2)       0.124043      -0.115882
```

```
          (2,1)       0.375488       0.287303
          (2,2)       0.259077       0.197368
     AR(2)(1,1)      0.0754758     -0.0426874
          (1,2)     -0.0972418      -0.015377
          (2,1)      0.0687406     -0.0176683
          (2,2)      0.0155532      0.0134923
     MA(1)(1,1)       0.205242
          (1,2)      -0.239925
          (2,1)     -0.0881847
          (2,2)     -0.0617094
     MA(2)(1,1)     -0.0682232
          (1,2)      0.0107276
          (2,1)      -0.155213
          (2,2)     -0.0040213
        Q(1,1)           0.08           0.08
        Q(2,1)           0.01           0.01
        Q(2,2)           0.03           0.03
```

Instead of just the default number of AR lags (which is two), obtain the first four AR lags in SpecAR:

```
SpecAR = vgxar(Spec, 4);
vgxdisp(Spec, SpecAR)
```

```
  Model 1: 2-D VARMA(2,2) with No Additive Constant
           Conditional mean is AR-stable and is MA-invertible
  Model 2: 2-D VAR(4) with No Additive Constant
           Conditional mean is AR-stable and is MA-invertible
       Parameter        Model 1        Model 2
  -------------- -------------- --------------
      AR(1)(1,1)       0.373935       0.579177
           (1,2)       0.124043      -0.115882
           (2,1)       0.375488       0.287303
           (2,2)       0.259077       0.197368
      AR(2)(1,1)      0.0754758     -0.0426874
           (1,2)     -0.0972418      -0.015377
           (2,1)      0.0687406     -0.0176683
           (2,2)      0.0155532      0.0134923
      AR(3)(1,1)             []      0.0409534
           (1,2)             []    -0.00362997
           (2,1)             []      0.0861962
           (2,2)             []     -0.0177161
      AR(4)(1,1)             []     0.00955252
           (1,2)             []    -0.00469931
           (2,1)             []      0.0022339
```

```
          (2,2)             []    -0.00374581
      MA(1)(1,1)       0.205242
           (1,2)      -0.239925
           (2,1)     -0.0881847
           (2,2)     -0.0617094
      MA(2)(1,1)     -0.0682232
           (1,2)      0.0107276
           (2,1)      -0.155213
           (2,2)     -0.0040213
          Q(1,1)           0.08           0.08
          Q(2,1)           0.01           0.01
          Q(2,2)           0.03           0.03
```

Obtain just the 99th lag and display the result:

```
SpecAR = vgxar(Spec, 1, 99);
vgxdisp(SpecAR);
```

```
  Model  : 2-D VAR(1) with No Additive Constant
           Conditional mean is AR-stable and is MA-invertible
           Autoregression lags: 99
  AR(99) Autoregression Matrix:
     8.06035e-45   -2.39247e-45
     1.44771e-44   -4.29698e-45
  Q Innovations Covariance:
            0.08           0.01
            0.01           0.03
```

See Also

`vgxset` | `vgxma`

Introduced in R2008b

vgxcount

Count VARMAX model parameters

Syntax

```
NumParam = vgxcount(Spec)
```

```
[NumParam,NumActive] = vgxcount(Spec)
```

Description

`vgxcount` counts the total and active parameters in a multivariate time series model.

The total number of parameters in a multivariate time series model includes all parameters in the conditional mean and conditional covariance models. If the innovations process has a full covariance, the total number of parameters is

$$n + nAR \cdot n^2 + nMA \cdot n^2 + nX + n(n+1) / 2$$

where n is the number of time series, nAR is the number of autoregressive lag matrices, nMA is the number of moving average lag matrices, and nX is the number of exogenous parameters. If the innovations process has a diagonal covariance, the total number of parameters is

$$n + nAR \cdot n^2 + nMA \cdot n^2 + nX + n$$

If the model does not have a constant (if `Spec.Constant` is `false`), then the total is reduced by n.

Note: The innovations covariance matrix is a symmetric matrix with at most $n(n + 1)/2$ unique elements.

Input Arguments

`Spec`	A multivariate time series specification structure for an n-dimensional time series process, as created by `vgxset`.

Output Arguments

`NumParam`	Total number of parameters in current model.
`NumActive`	Number of active (unrestricted) parameters in current model.

Examples

Count VAR Model Parameters

Start with a 2-dimensional VARMA(2, 2) specification structure in `Spec`:

```
load Data_VARMA22
```

Change the model to estimate only the diagonals of the AR matrices and count the total number of parameters in `NumParam` and the number of unrestricted parameters in `NumActive`:

```
Spec = vgxset(Spec,'ARsolve',{logical(eye(2)),logical(eye(2))});

[NumParam, NumActive] = vgxcount(Spec)
```

```
NumParam =

    19


NumActive =

    15
```

Introduced in R2008b

vgxdisp

Display VARMAX model parameters and statistics

Syntax

```
vgxdisp(Spec)
vgxdisp(SpecStd)
vgxdisp(Spec,SpecStd)
vgxdisp(Spec1,Spec2)
vgxdisp(Spec1Std,Spec2Std)
vgxdisp(Spec1,Spec1Std,Spec2,Spec2Std)
vgxdisp(Spec1,Spec2,Spec1Std,Spec2Std)
vgxdisp(Spec1,Spec1Std,Spec2,Spec2Std,...,Specn,SpecnStd)
vgxdisp(Spec1,Spec2,...,Specn,Spec1Std,Spec2Std,...,SpecnStdn)
vgxdisp(___,'Name1',Value1,'Name2',Value2,...)
```

Description

`vgxdisp` displays multivariate time series model parameters and standard errors in different formats.

- `vgxdisp(Spec)` displays a single specification structure `Spec`.
- `vgxdisp(SpecStd)` displays a single standard-error structure `SpecStd` with no *t*-statistics.
- `vgxdisp(Spec,SpecStd)` displays a single specification structure `SpecStd` with standard errors.
- `vgxdisp(Spec1,Spec2)` displays two specification structures `Spec1` and `Spec2` side-by-side. This option displays the specification structures in `table` format only.
- `vgxdisp(Spec1Std,Spec2Std)` displays two standard-error structures `Spec1Std` and `Spec2Std` side-by-side. This option displays the standard-error structures in `table` format only.
- `vgxdisp(Spec1,Spec1Std,Spec2,Spec2Std)` displays two specification structures side-by-side with standard errors. This option displays the specification structures in `table` format only.

- `vgxdisp(Spec1,Spec2,Spec1Std,Spec2Std)` also displays two specification structures side-by-side with standard errors. This option displays the specification structures in `table` format only.
- `vgxdisp(Spec1,Spec1Std,Spec2,Spec2Std,...,Spec`*n*`,Spec`*n*`Std)` displays *n* specification structures with standard errors. This option displays the specification structures in `table` format only.
- `vgxdisp(Spec1,Spec2,...,Spec`*n*`,Spec1Std,Spec2Std,...,SpecnStd`*n*`)` displays *n* specification structures with standard errors. This option displays the specification structures in `table` format only.
- `vgxdisp(___,'`*Name1*`',Value1,'`*Name2*`',Value2,...)` displays specification structures with additional with additional options specified by one or more `Name,Value` pair arguments.

Required Input Arguments

`Spec`	A multivariate time series specification structure for an *n*-dimensional time series process, as created by `vgxset`.
`SpecStd`	A multivariate time series specification structure that contains standard errors (or estimation errors) of estimated parameters for a companion *n*-dimensional time series process, as created by `vgxset`. Since the standard errors are maximum likelihood estimates, set the parameter `DoFAdj` to `true` to apply a degree-of-freedom adjustment and report ordinary least squares estimates.

If you input multiple specification structures, all must have the same dimension *n*. Pairs of specification structures and standard errors must be conformable. You can, however, specify different `AR` or `MA` lag structures for multiple specification structures, and if the inputs are exogenous, you can also specify different numbers of parameters.

If the specification structures do not set any logical indicators for model parameter estimation (“solve” information), `vgxdisp` assumes that every parameter is available for estimation. In this case, the degree-of-freedom adjustment that `vgxdisp` makes is the most conservative estimate for standard errors.

Optional Input Arguments

Specify the following optional input arguments as variable-length lists of matching parameter name/value pairs: `'Name1'`, `Value1`, `'Name2'`, `Value2`, ... and so on. The following rules apply when specifying parameter-name pairs:

- Specify the parameter name as a character string, followed by its corresponding parameter value.
- You can specify parameter name/value pairs in any order.
- Parameter names are case insensitive.
- You can specify unambiguous partial string matches.

The following table lists valid parameter names.

`DoFAdj`	Specifies whether `vgxdisp` adjusts for degrees of freedom in standard errors. Options are: • `true` — `vgxdisp` applies degree-of-freedom adjustment (least-squares estimation). • `false` — `vgxdisp` does not apply degree-of-freedom adjustment (maximum likelihood estimation).
`Format`	Specifies format in which model parameters and standard errors are displayed. Options are: • `'equation'` — `vgxdisp` displays model parameters and standard errors in canonical equation form. This is the default format for single models. • `'table'` — `vgxdisp` displays model parameters and standard errors in tabular form. This is the only option for multiple models.

Examples

Display VAR Models

Start with a 2-dimensional VARMA(2,2) data and display the specification `Spec`:

```
load Data_VARMA22
```

```
vgxdisp(Spec);
```

```
  Model  : 2-D VARMA(2,2) with No Additive Constant
           Conditional mean is AR-stable and is MA-invertible
  AR(1) Autoregression Matrix:
        0.373935       0.124043
        0.375488       0.259077
  AR(2) Autoregression Matrix:
       0.0754758     -0.0972418
       0.0687406      0.0155532
  MA(1) Moving Average Matrix:
        0.205242      -0.239925
      -0.0881847     -0.0617094
  MA(2) Moving Average Matrix:
      -0.0682232      0.0107276
       -0.155213     -0.0040213
  Q Innovations Covariance:
            0.08           0.01
            0.01           0.03
```

Assume that you have a 2-dimensional VAR(2) approximation of the original VARMA(2, 2) model estimated from time series data that is in the specification structure `EstSpec`:

```
vgxdisp(Spec, EstSpec);
```

```
  Model 1: 2-D VARMA(2,2) with No Additive Constant
           Conditional mean is AR-stable and is MA-invertible
  Model 2: 2-D VAR(2) with No Additive Constant
           Conditional mean is AR-stable and is MA-invertible
       Parameter        Model 1        Model 2
  -------------- -------------- --------------
      AR(1)(1,1)       0.373935       0.850166
           (1,2)       0.124043     -0.0498191
           (2,1)       0.375488       0.219381
           (2,2)       0.259077     -0.0227752
      AR(2)(1,1)      0.0754758      -0.294609
           (1,2)     -0.0972418       0.221336
           (2,1)      0.0687406       0.264504
           (2,2)      0.0155532      0.0819125
      MA(1)(1,1)       0.205242
           (1,2)      -0.239925
           (2,1)     -0.0881847
           (2,2)     -0.0617094
      MA(2)(1,1)     -0.0682232
           (1,2)      0.0107276
```

```
    (2,1)      -0.155213
    (2,2)     -0.0040213
  Q(1,1)           0.08       0.051844
  Q(2,1)           0.01     0.00711775
  Q(2,2)           0.03      0.0286081
```

Introduced in R2008b

vgxget

Get VARMAX model specification parameters

Syntax

```
ParameterValue = vgxget(Spec,'ParameterName')
```

Description

`ParameterValue = vgxget(Spec,'ParameterName')` returns the value `ParameterValue` of the model specification parameter *`ParameterName`* given in the multivariate time series specification structure `Spec`.

Input Arguments

`Spec`	A multivariate time series specification structure for an n-dimensional time series process, as created by `vgxset`.
`ParameterName`	The parameter of `Spec` whose value is returned by `vgxget`. Specify *`ParameterName`* as a case-insensitive character string or as a single-element cell array whose contents are a character array. You need only type the leading characters that uniquely identify the parameter.

Output Arguments

`ParameterValue`	The value of *`ParameterName`*.

Examples

Set VARMA Model Parameters to a Variable

Start with a 2-dimensional VARMA(2, 2) specification structure in `Spec`:

```
load Data_VARMA22
```

Obtain and display the string that contains the description of the model:

```
Model = vgxget(Spec, 'Model')
```

```
Model =

2-D VARMA(2,2) with No Additive Constant
```

Obtain and display the MA coefficients of the model:

```
MA = vgxget(Spec, 'MA');
MA{:}
```

```
ans =

    0.2052   -0.2399
   -0.0882   -0.0617


ans =

   -0.0682    0.0107
   -0.1552   -0.0040
```

See Also

vgxset

Introduced in R2008b

vgxinfer

Infer VARMAX model innovations

Syntax

```
[W,logL] = vgxinfer(Spec,Y)
```

```
[W,logL] = vgxinfer(Spec,Y,X,Y0,W0)
```

Description

`vgxinfer` infers the innovations from observations of a multivariate time series process specified by a VARMAX model.

Input Arguments

`Spec`	A model specification structure for a multidimensional VARMAX time series process, as produced by `vgxset` or `vgxvarx`.
`Y`	Response data. `Y` is a matrix or a 3-D array. If `Y` is a *numObs*-by-*numDims* matrix, it represents *numObs* observations of a single path of a *numDims*-dimensional time series. If `Y` is a *numObs*-by-*numDims*-by-*numPaths* array, it represents *numObs* observations of *numPaths* paths of a *numDims*-dimensional time series. Observations across paths are assumed to occur at the same time. The last observation is assumed to be the most recent.

Optional Input Arguments

`X`	Exogenous data. `X` is a cell vector or a cell matrix. Each cell contains a *numDims*-by-*numX* design matrix `X(t)` so that, for some b, `X(t)*`b is the regression component of a single *numDims*-dimensional observation `Y(t)` at time t. If `X` is a *numObs*-by-1 cell vector, it represents one path of the explanatory variables. If `X` is a *numObs*- by-*numXPaths* cell matrix, it represents *numXPaths* paths of the explanatory variables. If

Y has multiple paths, X must contain either a single path (applied to all paths in Y) or at least as many paths as in Y (extra paths are ignored).

YO — Presample response data. YO is a matrix or a 3-D array. If YO is a *numPresampleYObs*-by-*numDims* matrix, it represents *numPresampleYObs* observations of a single path of a *numDims*-dimensional time series. If YO is a *numPresampleYObs*-by-*numDims*-by-*numPreSampleYPaths* array, it represents *numPresampleYObs* observations of *numPreSampleYPaths* paths of a *numDims*-dimensional time series. If YO is empty or if *numPresampleYObs* is less than the maximum AR lag in Spec, presample values are padded with zeros. If *numPresampleYObs* is greater than the maximum AR lag, the most recent samples from the last rows of each path of YO are used. If Y has multiple paths, YO must contain either a single path (applied to all paths in Y) or at least as many paths as in Y (extra paths are ignored).

WO — Presample innovations data. WO is a matrix or a 3-D array. If WO is a *numPresampleWObs*-by-*numDims* matrix, it represents *numPresampleWObs* observations of a single path of a *numDims*-dimensional time series. If W0 is a *numPresampleWObs*-by-*numDims*-by-*numPreSampleWPaths* array, it represents *numPresampleWObs* observations of *numPreSampleWPaths* paths of a *numDims*-dimensional time series. If WO is empty or if *numPresampleWObs* is less than the maximum MA lag in Spec, presample values are padded with zeros. If *numPresampleWObs* is greater than the maximum MA lag, the most recent samples from the last rows of each path of WO are used. If Y has multiple paths, WO must contain either a single path (applied to all paths in Y) or at least as many paths as in Y (extra paths are ignored).

Output Arguments

W — Inferred innovations process, the same size as Y.

LogL — 1-by-*numPaths* vector containing the total loglikelihood of the response data in each path of Y.

Note: The functions vgxinfer and vgxproc are complementary. For example, given a specification structure Spec for a stable and invertible process and an innovations process W1, the code

```
Y = vgxproc(Spec,W1,X,Y0,W0);
W2 = vgxinfer(Spec,Y,X,Y0,W0);
```

produces an innovations process `W2` that is identical to `W1`. Differences can appear if the process in `Spec` fails to be either stable or invertible.

See Also

`vgxpred` | `vgxproc` | `vgxsim`

Introduced in R2008b

vgxloglik

VARMAX model loglikelihoods

Syntax

```
LLF = vgxloglik(Spec,W)
```

```
[LLF,CLLF] = vgxloglik(Spec,W)
```

Description

`vgxloglik` computes total and conditional loglikelihoods of a multivariate time series process.

Input Arguments

`Spec`	A multivariate time series specification structure for an n-dimensional time series process, as created by `vgxset`.
`W`	Innovations process. nP paths of an n-dimensional innovations process with T observations for each path, collected in a T-by-n-by-nP array. Times are ordered by row from oldest to most recent. The innovations covariance is assumed to be positive-definite. To obtain innovations given a specification structure and a path of a multiple time series process, use `vgxinfer`.

Output Arguments

`LLF`	Total loglikelihood function for T observations of an n-dimensional time series process. If `W` has nP paths, `LLF` is a 1-by-nP vector containing the total loglikelihood function for each path.
`CLLF`	Conditional loglikelihoods for T observations of an n-dimensional time series process. If `W` has nP paths, `CLLF` is a T-by-nP matrix containing the conditional loglikelihoods for each path. The total loglikelihood `LLF` is the sum of the T conditional loglikelihoods in `CLLF`.

Examples

Compute VARMA Model Loglikelihood

Start with a 2-dimensional VARMA(2, 2) specification structure in `Spec` with time series data and presample data:

```
load Data_VARMA22
```

Compute the total loglikelihood function given a specification structure in `Spec` and an innovations process derived from the time series data `Y` using the function `vgxinfer`:

```
W = vgxinfer(Spec, Y, [], Y0, W0);
LLF = vgxloglik(Spec, W)
```

```
LLF =

   17.8440
```

See Also

`vgxinfer`

Introduced in R2008b

vgxma

Convert VARMA model to VMA model

Syntax

```
SpecMA = vgxma(Spec)

SpecMA = vgxma(Spec,nMA,MAlag,Cutoff)
```

Description

`vgxma` converts a VARMA model into a pure vector moving average (VMA) model. This function works only for VARMA models and does not handle exogenous variables (VARMAX models).

Required Input Argument

`Spec`	A multivariate time series specification structure for an n-dimensional VARMA time series process, as created by `vgxset`.

Optional Input Arguments

`nMA`	Number of MA lags for the output specification structure. `vgxma` truncates an infinite-order VMA model to `nMA` lags. If specific MA lags are not given by `MAlag`, the lags are `1:nMA`. To use `MAlag`, set `nMA` to `[]` or to the number of specific lags.
`MAlag`	A positive integer vector of specific MA lags for the output specification structure. `MAlag` must be of length `nMA`, unless `nMA` is `[]`.
`Cutoff`	The cutoff for the infinity norm below which trailing lags are removed. The default is `0`, which does not remove any lags and uses the values for `nMA` and `MAlag`.

If neither nMA nor MAlag is specified, vgxma uses the maximum lags of the AR or MA lags of the input Spec.

Note: If a large number of lags is needed to form a pure VMA representation (with unit roots close to 1), a large number of initial values is also needed for propagation.

Output Arguments

SpecMA	A transformed multivariate time series specification structure that consists of a pure vector moving average (VMA) model with nMA lags. Logical indicators for model parameter estimation ("solve" information) in Spec are not passed on to SpecMA.

Examples

Convert a VARMA Model to a VMA Model

Start with a 2-dimensional VARMA(2, 2) specification structure in Spec:

```
load Data_VARMA22
```

Convert Spec into a pure VMA(2) model in SpecMA:

```
SpecMA = vgxma(Spec);
```

Display the original specification structure in Spec and compare with the new specification structure in SpecMA:

```
vgxdisp(Spec, SpecMA)
```

```
  Model 1: 2-D VARMA(2,2) with No Additive Constant
           Conditional mean is AR-stable and is MA-invertible
  Model 2: 2-D VMA(2) with No Additive Constant
           Conditional mean is AR-stable and is MA-invertible
       Parameter         Model 1        Model 2
  -------------- -------------- --------------
      AR(1)(1,1)       0.373935
           (1,2)       0.124043
           (2,1)       0.375488
```

```
          (2,2)       0.259077
      AR(2)(1,1)      0.0754758
          (1,2)     -0.0972418
          (2,1)      0.0687406
          (2,2)      0.0155532
      MA(1)(1,1)       0.205242       0.579177
          (1,2)      -0.239925      -0.115882
          (2,1)     -0.0881847       0.287303
          (2,2)     -0.0617094       0.197368
      MA(2)(1,1)     -0.0682232       0.259465
          (1,2)      0.0107276      -0.105364
          (2,1)      -0.155213       0.205435
          (2,2)     -0.0040213      0.0191531
         Q(1,1)           0.08           0.08
         Q(2,1)           0.01           0.01
         Q(2,2)           0.03           0.03
```

Obtain the first 4 MA lags in `SpecMA`:

```
SpecMA = vgxma(Spec, 4);
vgxdisp(Spec, SpecMA);
```

```
  Model 1: 2-D VARMA(2,2) with No Additive Constant
           Conditional mean is AR-stable and is MA-invertible
  Model 2: 2-D VMA(4) with No Additive Constant
           Conditional mean is AR-stable and is MA-invertible
       Parameter        Model 1        Model 2
  -------------- -------------- --------------
      AR(1)(1,1)       0.373935
          (1,2)       0.124043
          (2,1)       0.375488
          (2,2)       0.259077
      AR(2)(1,1)      0.0754758
          (1,2)     -0.0972418
          (2,1)      0.0687406
          (2,2)      0.0155532
      MA(1)(1,1)       0.205242       0.579177
          (1,2)      -0.239925      -0.115882
          (2,1)     -0.0881847       0.287303
          (2,2)     -0.0617094       0.197368
      MA(2)(1,1)     -0.0682232       0.259465
          (1,2)      0.0107276      -0.105364
          (2,1)      -0.155213       0.205435
          (2,2)     -0.0040213      0.0191531
      MA(3)(1,1)             []       0.138282
```

```
          (1,2)             []     -0.0649623
          (2,1)             []       0.194931
          (2,2)             []      -0.039497
     MA(4)(1,1)             []      0.0754946
          (1,2)             []      -0.039006
          (2,1)             []       0.123456
          (2,2)             []     -0.0415703
         Q(1,1)           0.08           0.08
         Q(2,1)           0.01           0.01
         Q(2,2)           0.03           0.03
```

Obtain just the 99th lag and display the result:

```
SpecMA = vgxma(Spec, 1, 99);
vgxdisp(SpecMA);
```

```
  Model  : 2-D VMA(1) with No Additive Constant
           Conditional mean is AR-stable and is MA-invertible
           Moving average lags: 99
  MA(99) Moving Average Matrix:
     2.09723e-30   -1.03631e-30
     3.16333e-30   -8.85453e-31
  Q Innovations Covariance:
            0.08           0.01
            0.01           0.03
```

See Also

vgxar

Introduced in R2008b

vgxplot

Plot VARMAX model responses

Syntax

```
vgxplot(Spec,Y)

vgxplot(Spec,Y,FY)

vgxplot(Spec,Y,FY,FYCov)
```

Description

`vgxplot` plots a multivariate time series process with optional error bands.

Required Input Arguments

`Spec`	A multivariate time series specification structure for an n-dimensional VARMA time series process, as created by `vgxset`.
`Y`	nPY observed paths of an n-dimensional time series process with T observations for each path, collected in a T-by-n-by-nPY array. Times are ordered by row from oldest to most recent. Plotted error bands are plus or minus one standard deviation of the one-period prediction error derived from `Spec`.

Optional Input Arguments

`FY`	$nPFY$ forecast paths of an n-dimensional time series process with FT observations for each path, collected in a FT-by-n-by-$nPFY$ array. Times are ordered by row from oldest to most recent. Plotted error bands are

	plus or minus one standard deviation of the cumulative forecast error derived from `FYCov`.
`FYCov`	A single path of forecast error covariances for an n-dimensional time series process with FT observations. `FYCov` is stored as an FT-cell vector with n-by-n forecast error covariance matrices in each cell for FT times. `FYCov` is the same if the underlying time series process has multiple paths, so only one path is necessary. Although multiple time series paths can be plotted, `FYCov` is based on calibration of a single path of the time series process. Plots with multiple forecast paths are displayed with error bands derived from `FYCov` that may not be valid for all paths. Nonetheless, the error bands enclose the envelope of multiple paths.

Examples

Forecast and Plot a Vector Autoregressive Process

Start with a 2-dimensional VARMA(2, 2) specification structure in `Spec` and time series data in `Y`:

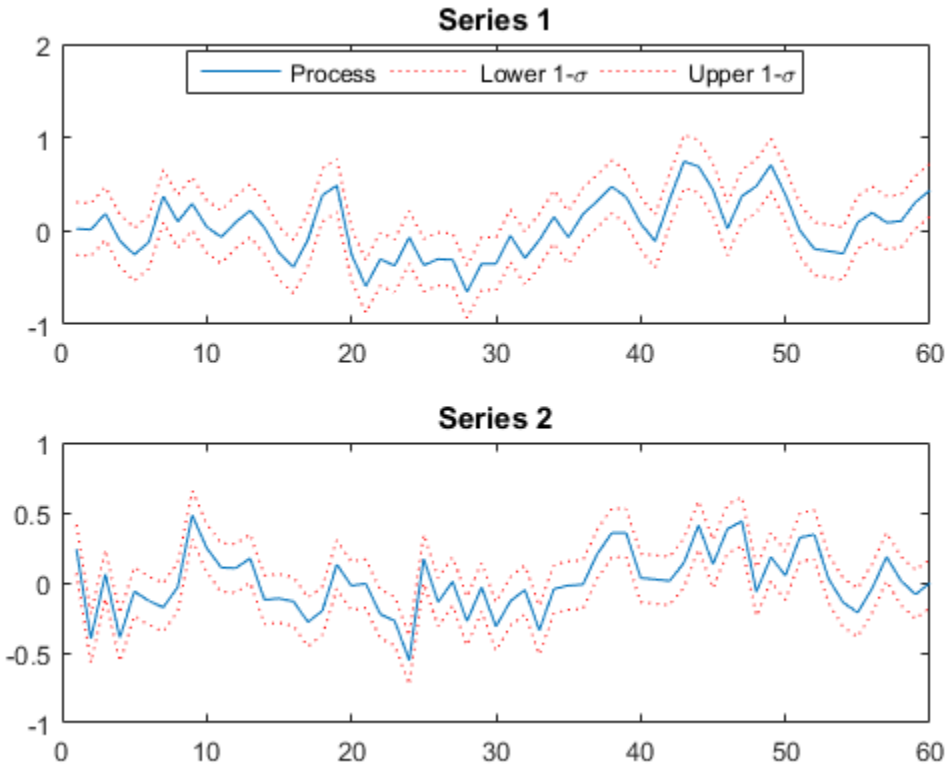
```
load Data_VARMA22
```

Propagate the time series forward 5 periods in `FY` and the forecast error covariance in `FYCov`:

```
[FY, FYCov] = vgxpred(Spec, 5, [], Y);
```

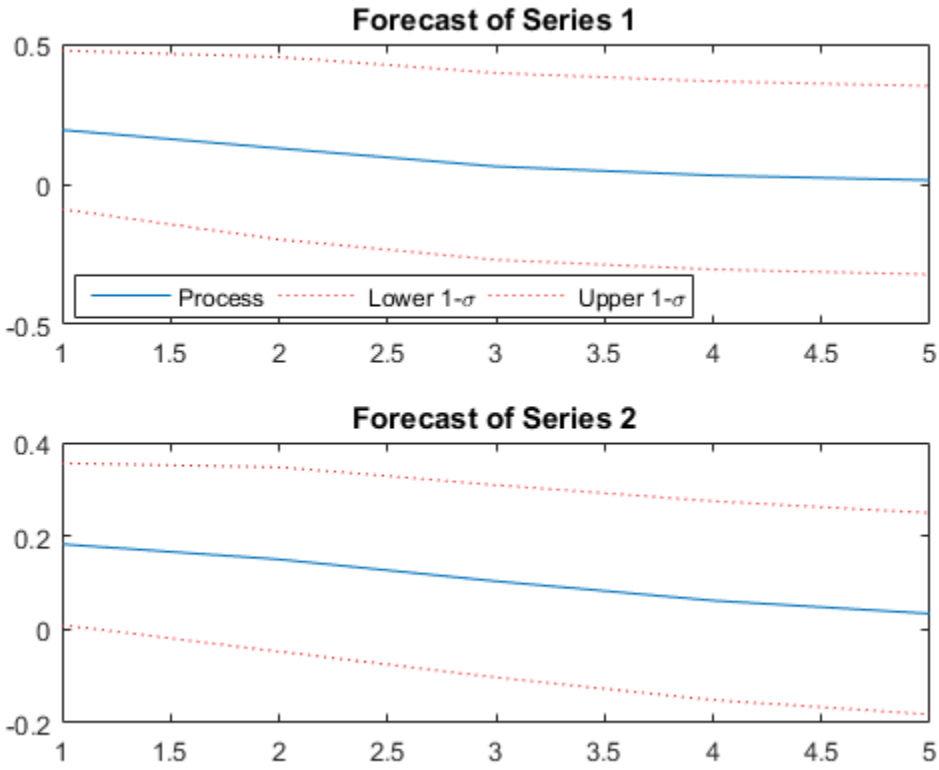
Plot just the times series process with 1-step prediction error bands:

```
vgxplot(Spec, Y);
```



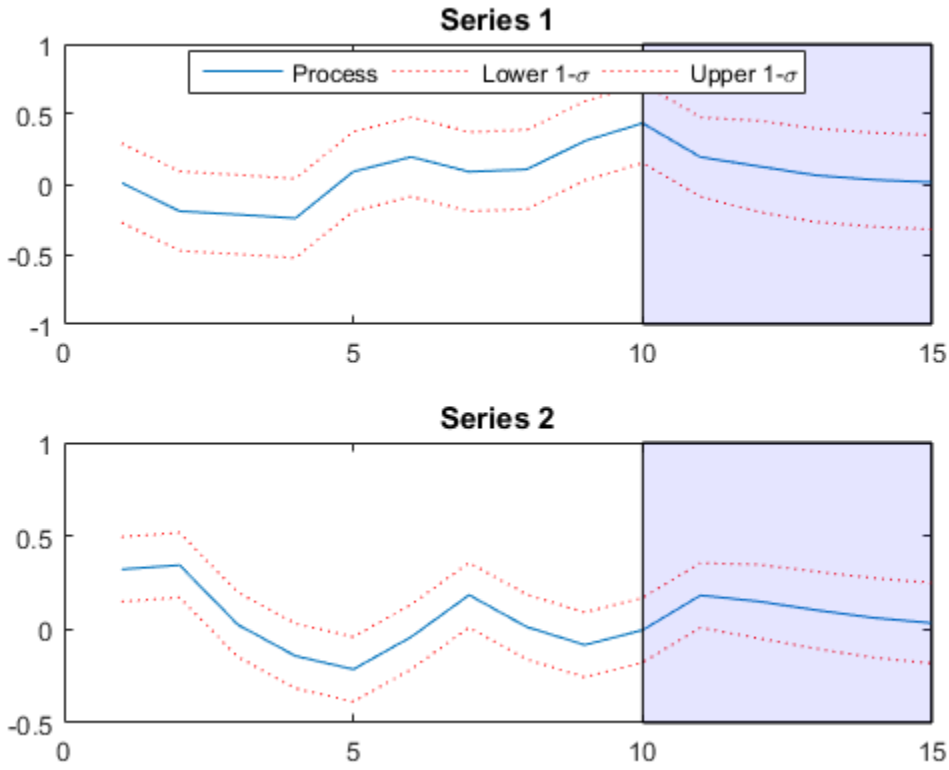
Plot just the forecast time series process with t-step prediction error bands:

```
vgxplot(Spec, [], FY, FYCov);
```



Plot both the time series process and its forecast with prediction errors (here the plot just displays the last 10 samples of the times series data):

```
vgxplot(Spec, Y(end-9:end,:), FY, FYCov);
```



Introduced in R2008b

vgxpred

Forecast VARMAX model responses

Syntax

```
FY = vgxpred(Spec,FT)

[FY,FYCov] = vgxpred(Spec,FT,FX,Y,W,NumPaths)
```

Description

`vgxpred` returns the transient response of a process during a forecast period with zero-valued innovations. To generate a process during a forecast period with simulated innovations, use `vgxsim`. To generate a process during a forecast period with known innovations, use `vgxproc`.

Input Arguments

`Spec`	A multivariate time series specification structure for an n-dimensional VARMA time series process, as created by `vgxset`.
`FT`	Number of forecast observations to be generated.
`FX`	nP paths of forecast regression design matrices associated with `FT` observations of an n-dimensional time series process, where each design matrix linearly relates nX exogenous inputs to each time series at each observation time. `FX` is an `FT`-by-nP matrix of cell arrays with n-by-nX design matrices in each cell. If `FY` has multiple paths, `FX` must contain either a single path or no fewer than the same number of paths as in `FY`. Extra paths are ignored.
`Y`	Presample time series process from the estimation period used for the forecast period. `Y` is a collection of nPY paths of an n-dimensional time series process with T observations for each path, collected in a T-by-n-by-nPY array. If `Y` has insufficient observations, the usual initialization methods for `vgxproc` and `vgxsim` apply.

`W`	Presample innovations process from the estimation period used for the forecast period. `W` is a collection of nPW paths of an n-dimensional innovations process with T observations for each path, collected in a T-by-n-by-nPW array. If `W` has insufficient observations, the usual initialization methods for `vgxproc` and `vgxsim` apply.
`NumPaths`	Number of paths to forecast. To generate multiple paths, you must use `NumPaths` to specify the number of paths. If `FX`, `Y`, and `W` have single paths and `NumPaths` > 1, every forecast is the same.

Output Arguments

`FY`	Forecast times-series process. `FY` is a collection of `NumPaths` paths of an n-dimensional time series process with `FT` observations for each path, collected in an `FT`-by-n-by-`NumPaths` array.
`FYCov`	Forecast error covariance matrices. `FYCov` is a single path of forecast error covariances for an n-dimensional time series process with `FT` observations. `FYCov` is collected in an `FT`-cell vector with n-by-n forecast error covariance matrices in each cell for $t = 1, \ldots, FT$. `FYCov{1}` is the one-period forecast covariance, `FYCov{2}` is the two-period forecast covariance, and so forth. `FYCov` is the same if multiple paths exist for the underlying time series process.

Examples

Plot VARMA Model Forecasts

Start with a 2-dimensional VARMA(2, 2) specification structure in `Spec` and times series and innovations process in `Y` and `W`:

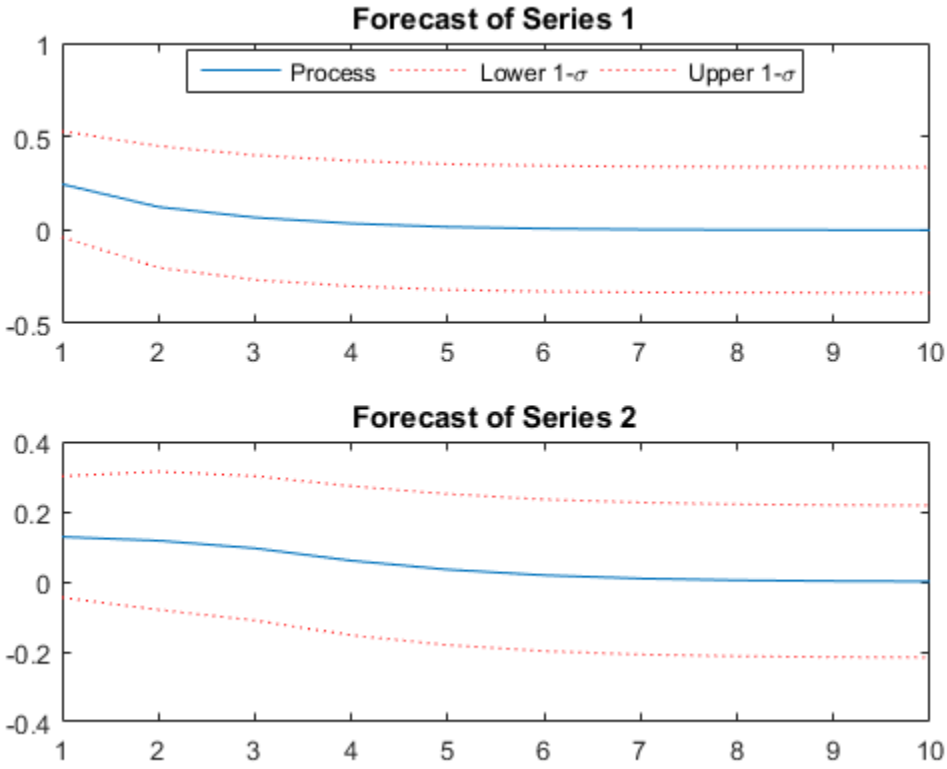
```
load Data_VARMA22
```

Forecast 10 samples into the future and use the time series and innovations process as presample data:

```
[FY, FYCov] = vgxpred(Spec, 10, [], Y, W);
```

Plot the transient response along with the predict-ahead forecast errors which are extracted from `FYCov` by the function `vgxplot`:

```
vgxplot(Spec, [], FY, FYCov);
```



See Also

`vgxinfer` | `vgxproc` | `vgxsim`

Introduced in R2008b

vgxproc

Generate VARMAX model responses from innovations

Syntax

```
[Y,logL] = vgxproc(Spec,W)
[Y,logL] = vgxproc(Spec,W,X,Y0,W0)
```

Description

`vgxproc` generates model responses using known innovations and a VARMAX model specification. To generate responses with simulated innovations, use `vgxsim`. To generate responses with zero-valued innovations, use `vgxpred`.

Input Arguments

`Spec`	A model specification structure for a multidimensional VARMAX time series process, as produced by `vgxset` or `vgxvarx`.
`W`	Innovations data, as produced by `vgxinfer`. `W` is a matrix or a 3-D array. If `W` is a *numObs*-by-*numDims* matrix, it represents *numObs* observations of a single path of a *numDims*-dimensional time series. If `W` is a *numObs*-by-*numDims*-by-*numPaths* array, it represents *numObs* observations of *numPaths* paths of a *numDims*-dimensional time series. Observations across paths are assumed to occur at the same time. The last observation is assumed to be the most recent.

Optional Input Arguments

`X`	Exogenous data. `X` is a cell vector or a cell matrix. Each cell contains a *numDims*-by-*numX* design matrix `X(t)` so that, for some b, `X(t)*b` is the regression component of a single *numDims*-dimensional observation `Y(t)` at time t. If `X` is a *numObs*-by-1 cell vector, it represents one path of the explanatory variables. If `X` is a *numObs*-by-*numXPaths* cell matrix, it represents *numXPaths* paths of the explanatory variables. If

W has multiple paths, X must contain either a single path (applied to all paths in W) or at least as many paths as in W (extra paths are ignored).

Y0 Presample response data. Y0 is a matrix or a 3-D array. If Y0 is a *numPresampleYObs*-by-*numDims* matrix, it represents *numPresampleYObs* observations of a single path of a *numDims*-dimensional time series. If Y0 is a *numPresampleYObs*-by-*numDims*-by-*numPreSampleYPaths* array, it represents *numPresampleYObs* observations of *numPreSampleYPaths* paths of a *numDims*-dimensional time series. If Y0 is empty or if *numPresampleYObs* is less than the maximum AR lag in Spec, presample values are padded with zeros. If *numPresampleYObs* is greater than the maximum AR lag, the most recent samples from the last rows of each path of Y0 are used. If W has multiple paths, Y0 must contain either a single path (applied to all paths in W) or at least as many paths as in W (extra paths are ignored).

W0 Presample innovations data. W0 is a matrix or a 3-D array. If W0 is a *numPresampleWObs*-by-*numDims* matrix, it represents *numPresampleWObs* observations of a single path of a *numDims*-dimensional time series. If W0 is a *numPresampleWObs*-by-*numDims*-by-*numPreSampleWPaths* array, it represents *numPresampleWObs* observations of *numPreSampleWPaths* paths of a *numDims*-dimensional time series. If W0 is empty or if *numPresampleWObs* is less than the maximum MA lag in Spec, presample values are padded with zeros. If *numPresampleWObs* is greater than the maximum MA lag, the most recent samples from the last rows of each path of W0 are used. If W has multiple paths, W0 must contain either a single path (applied to all paths in W) or at least as many paths as in W (extra paths are ignored).

Output Arguments

Y Response data, the same size as W.

LogL 1-by-*numPaths* vector containing the total loglikelihood of the response data in each path of Y.

Note: The functions vgxinfer and vgxproc are complementary. For example, given a specification structure Spec for a stable and invertible process and an innovations process W1, the code

```
Y = vgxproc(Spec,W1,X,Y0,W0);
W2 = vgxinfer(Spec,Y,X,Y0,W0);
```

produces an innovations process `W2` that is identical to `W1`. Differences can appear if the process in `Spec` fails to be either stable or invertible.

See Also

`vgxinfer` | `vgxpred` | `vgxsim`

Introduced in R2008b

vgxqual

Test VARMAX model for stability/invertibility

Syntax

```
[isStable,isInvertible] = vgxqual(Spec)

[isStable,isInvertible,AReig,MAeig] = vgxqual(Spec)
```

Description

`vgxqual` determines if a multivariate time series process is stable and invertible.

A process with non-constant exogenous inputs may not be stable or invertible. This cannot be determined from the specification structure unless the number of exogenous inputs is zero. `vgxqual` determines if the AR and the MA portions of a VARMAX model are stable and invertible *without considering exogenous inputs*. Thus it is more appropriate to call a multivariate time series process *AR-stable* if the AR portion is stable, and *MA-invertible* if the MA portion is invertible.

A stable VARMAX process is stationary, but the converse is not true. Although the terms *stable*, *stationary*, and *covariance-stationary* are often used interchangeably, a process is truly stationary if and only if its first and second moments are independent of time.

If a VARMAX model has no autoregressive terms, it is always AR-stable.

If a VARMAX model has no moving average terms, it is always MA-invertible.

Input Arguments

`Spec`	A multivariate time series specification structure for an n-dimensional time series process, as created by `vgxset`.

Output Arguments

`isStable`	Logical flag indicating if the multivariate time series process is stable. The flag is `true` if the process is stable, `false` otherwise.
`isInvertible`	Logical flag indicating if the multivariate time series process is invertible. The flag is `true` if the process is invertible, `false` otherwise.
`AReig`	Largest-magnitude eigenvalue for the AR portion of the multivariate time series process.
`MAeig`	Largest-magnitude eigenvalue for the MA portion of the multivariate time series process.

Examples

Verify the Stability and Invertibility of a VARMA Model

Start with a 2-dimensional VARMA(2,2) specification structure in `Spec`.

```
load Data_VARMA22
```

Although the display method for a `vgxset` object performs this test, the explicit test is:

```
[isStable, isInvertible] = vgxqual(Spec)
```

```
isStable =

     1


isInvertible =

     1
```

This shows that `Spec` is a model for an AR-stable and MA-invertible time series process.

Introduced in R2008b

vgxset

Set VARMAX model specification parameters

Syntax

```
Spec = vgxset('Name1',Value1,'Name2',Value2,...)
Spec = vgxset(OldSpec,'Name1',Value1,'Name2',Value2,...)
Spec = vgxset
```

Description

`vgxset` sets or modifies parameter values in a multivariate time series specification structure.

`Spec = vgxset('Name1',Value1,'Name2',Value2,...)` creates a multivariate time series specification structure `Spec` with parameters of specified name set to specified values. See "Name-Value Pair Arguments" on page 9-1074.

`Spec = vgxset(OldSpec,'Name1',Value1,'Name2',Value2,...)` modifies the multivariate time series specification structure `OldSpec`, changing the parameters of specified name to the specified values. See "Name-Value Pair Arguments" on page 9-1074.

`Spec = vgxset` creates a specification structure template for the default model (a univariate Gaussian noise process with no offset).

Input Arguments

`'Name'`	A character string naming a valid parameter of the output specification structure `Spec`.
`Value`	The value assigned to the corresponding parameter.
`OldSpec`	A specification structure to be modified, as created by `vgxset`.

Output Arguments

`Spec`	A specification encapsulating the style, orders, and parameters of the conditional mean of a multivariate time series model.

Name-Value Pair Arguments

Specify the following optional input arguments as variable-length lists of matching parameter name/value pairs: `'Name1'`, `Value1`, `'Name2'`, `Value2`, ... and so on. The following rules apply when specifying parameter-value pairs:

- Specify the parameter name as a character string, followed by its corresponding parameter value.
- You can specify parameter name/value pairs in any order.
- Parameter names are case insensitive.
- You can specify unambiguous partial string matches.

Model Information

Name	Value
`'Model'`	A string describing the model. The default is auto-generated, and depends on model parameters.
`'Info'`	A string with additional model information. The default is empty.
`'Series'`	A cell array of strings providing names for each time series. The default is empty.

Model Orders

Name	Value
`n`	A positive integer specifying the number of time series. The default is `1`.
`nAR`	A nonnegative integer specifying the number of AR lags. The default is `0`.
`nMA`	A nonnegative integer specifying the number of MA lags. The default is `0`.
`nX`	A nonnegative integer specifying the number regression parameters. The default is `0`.

Model Parameters

Name	Value
a	An n-vector of offset constants. The default is empty.
AR0	An n-by-n invertible matrix representing the zero-lag structural AR coefficients. The default is empty.
AR	An nAR-element cell array of n-by-n matrices of AR coefficients. The default is empty.
MA0	An n-by-n invertible matrix representing the zero-lag structural MA coefficients. The default is empty.
MA	An nMA-element cell array of n-by-n matrices of MA coefficients. The default is empty.
b	An nX-vector of regression coefficients. The default is empty.
Q	An n-by-n symmetric innovations covariance matrix. The default is empty.

Model Lags

Name	Value
ARlag	A monotonically increasing nAR-vector of AR lags. The default is empty.
MAlag	A monotonically increasing nMA-vector of MA lags. The default is empty.

Model Parameter Estimation

Name	Value
asolve	An n-vector of additive offset logical indicators. The default is empty.
ARsolve	An nAR-element cell array of n-by-n matrices of AR logical indicators. The default is empty.
AR0solve	An n-by-n matrix of AR0 logical indicators. The default is empty.
MAsolve	An nMA-element cell array of n-by-n matrices of MA logical indicators. The default is empty.
MA0solve	An n-by-n matrix of MA0 logical indicators. The default is empty.
bsolve	An nX-vector of regression logical indicators. The default is empty.

Name	Value
`Qsolve`	An `n`-by-`n` symmetric covariance matrix logical indicator. The default is empty.

Currently, `vgxvarx` cannot fit the following matrices:

- `AR0`
- `MA0`
- `MA`

Therefore, `vgxvarx` ignores `AR0solve`, `MA0solve`, and `MAsolve`. However, you are invited to examine the `Example_StructuralParams.m` file for one approach to fitting the `AR0` and `MA0` matrices. Enter `help Example_StructuralParams` at the MATLAB command line for information.

Furthermore, `vgxvarx` also ignores `Qsolve`. `vgxvarx` can fit either a diagonal or a full `Q` matrix; see "Optional Input Arguments" on page 9-1082.

Model Offset

Name	Value
`Constant`	Additive offset logical indicator. The default is `false`.

Examples

Specify VARMA Models and Adjust Parameter Values

You can set up a multiple time series specification structure in three ways. The first and most direct approach is to set all model parameters directly. The following command creates a specification structure for a 2-dimensional VARMAX(2,1,2) model with constant offset:

```
Spec = vgxset('a', [0.1; -0.1], ...
 'b', [1.1, 0.9], ...
 'AR', {[0.6, 0.3; -0.4, 0.7], [0.3, 0.1; 0.05, 0.2]}, ...
    'MA', [0.5, -0.1; 0.07, 0.2], ...
 'Q', [0.03, 0.004; 0.004, 0.006])
```

```
Spec =

      Model: 2-D VARMAX(2,1,2) with Additive Constant
          n: 2
        nAR: 2
        nMA: 1
         nX: 2
          a: [0.1 -0.1] additive constants
         AR: {2x1 cell} stable autoregressive process
         MA: {1x1 cell} invertible moving average process
          b: [1.1 0.9] regression coefficients
          Q: [2x2] covariance matrix
```

Notice that multiple lag coefficients are specified as matrices in each cell of a vector cell array and that a single lag can be entered as a matrix.

A second approach creates the same specification structure for a 2-dimensional VARMAX(2,1,2) model with constant offset. In this case, however, no parameter values have been set:

```
Spec = vgxset('n',2,'nAR',2,'nMA',1,'nX',2, ...
    'Constant',true)
```

```
Spec =

      Model: 2-D VARMAX(2,1,2) with Additive Constant
          n: 2
        nAR: 2
        nMA: 1
         nX: 2
          a: []
         AR: {}
         MA: {}
          b: []
          Q: []
```

Note that this approach requires you to specify explicitly the inclusion of an additive constant.

Given this specification structure, you can fill in the parameters or use calibration methods to fill in the rest of the structure.

The third way to set up a specification structure is to specify which parameters in a model that you would like to estimate. The following command creates a specification structure for our 2-dimensional model:

```
Spec = vgxset('ARsolve', repmat({true(2)}, 2, 1), ...
    'MAsolve',true(2),'bsolve',true(2, 1), ...
    'asolve',true(2,1))
```

```
Spec =

      Model: 2-D VARMAX(2,1,2) with Additive Constant
          n: 2
        nAR: 2
        nMA: 1
         nX: 2
          a: []
     asolve: [1 1 logical] additive constant indicators
         AR: {}
    ARsolve: {2x1 cell of logicals} autoregressive lag indicators
         MA: {}
    MAsolve: {1x1 cell of logicals} moving average lag indicators
          b: []
     bsolve: [1 1 logical] regression coefficient indicators
          Q: []
```

Notice that the dimensions of the model have been specified implicitly from the "solve" flags which are displayed since they were set directly. The solve flags were not displayed in prior examples since, by default, the "solve" flags are implicitly true.

Finally, you can change a specification structure by passing it into the function `vgxset`. For example, if you start with a fully-qualified VARMAX(2, 1, 2) model,

```
Spec = vgxset('a', [0.1; -0.1], ...
         'b', [1.1, 0.9], ...
        'AR', {[0.6, 0.3; -0.4, 0.7], [0.3, 0.1; 0.05, 0.2]}, ...
         'MA', [0.5, -0.1; 0.07, 0.2], ...
          'Q', [0.03, 0.004; 0.004, 0.006])
```

```
Spec =

      Model: 2-D VARMAX(2,1,2) with Additive Constant
          n: 2
```

```
        nAR: 2
        nMA: 1
         nX: 2
          a: [0.1 -0.1] additive constants
         AR: {2x1 cell} stable autoregressive process
         MA: {1x1 cell} invertible moving average process
          b: [1.1 0.9] regression coefficients
          Q: [2x2] covariance matrix
```

you can remove exogenous inputs from the model to convert it into a VARMA(2, 1) model with:

```
Spec = vgxset(Spec, 'nX', 0, 'b', [])
```

```
Spec =

      Model: 2-D VARMA(2,1) with Additive Constant
          n: 2
        nAR: 2
        nMA: 1
         nX: 0
          a: [0.1 -0.1] additive constants
         AR: {2x1 cell} stable autoregressive process
         MA: {1x1 cell} invertible moving average process
          Q: [2x2] covariance matrix
```

Notice that you must remove all information about the exogenous inputs which means both the dimensions `'nX'` and, if nonempty, the parameters `'b'`.

See Also

`vgxget` | `vgxsim` | `vgxvarx`

Introduced in R2008b

vgxsim

Simulate VARMAX model responses

Syntax

```
Y = vgxsim(Spec,numObs)

[Y,W] = vgxsim(Spec,numObs,X,Y0,W0,numPaths)
```

Description

`vgxsim` simulates sample paths and innovations of a multidimensional VARMAX time series process.

Input Arguments

`Spec`	A model specification structure for a multidimensional VARMAX time series process, as produced by `vgxset` or `vgxvarx`.
`numObs`	Positive integer indicating the number of observations generated for each path of output arguments `Y` and `W`.

Optional Input Arguments

`X`	Exogenous data. `X` is a cell vector or a cell matrix. Each cell contains a *numDims*-by-*numX* design matrix `X(t)` so that, for some *b*, `X(t)*b` is the regression component of a single *numDims*-dimensional response `Y(t)` at time *t*. If `X` is a *numObs*-by-1 cell vector, it represents one path of the explanatory variables. If `X` is a *numObs*- by-*numXPaths* cell matrix, it represents *numXPaths* paths of the explanatory variables. If `Y` has multiple paths, `X` must contain either a single path (applied to all paths in `Y`) or at least as many paths as in `Y` (extra paths are ignored).

`Y0`	Presample response data. `Y0` is a matrix or a 3-D array. If `Y0` is a *numPresampleYObs*-by-*numDims* matrix, it represents *numPresampleYObs* observations of a single path of a *numDims*-dimensional time series. If `Y0` is a *numPresampleYObs*-by-*numDims*-by-*numPreSampleYPaths* array, it represents *numPresampleYObs* observations of *numPreSampleYPaths* paths of a *numDims*-dimensional time series. If `Y0` is empty or if *numPresampleYObs* is less than the maximum `AR` lag in `Spec`, presample values are padded with zeros. If *numPresampleYObs* is greater than the maximum `AR` lag, the most recent samples from the last rows of each path of `Y0` are used. If `Y` has multiple paths, `Y0` must contain either a single path (applied to all paths in `Y`) or at least as many paths as in `Y` (extra paths are ignored).
`W0`	Presample innovations data. `W0` is a matrix or a 3-D array. If `W0` is a *numPresampleWObs*-by-*numDims* matrix, it represents *numPresampleWObs* observations of a single path of a *numDims*-dimensional time series. If `W0` is a *numPresampleWObs*-by-*numDims*-by-*numPreSampleWPaths* array, it represents *numPresampleWObs* observations of *numPreSampleWPaths* paths of a *numDims*-dimensional time series. If `W0` is empty or if *numPresampleWObs* is less than the maximum `MA` lag in `Spec`, presample values are padded with zeros. If *numPresampleWObs* is greater than the maximum `MA` lag, the most recent samples from the last rows of each path of `W0` are used. If `Y` has multiple paths, `W0` must contain either a single path (applied to all paths in `Y`) or at least as many paths as in `Y` (extra paths are ignored).
`numPaths`	Number of simulated paths. The default is 1. It is necessary to specify `numPaths` to generate multiple paths. If `numPaths` > 1 and `X`, `Y`, and `W` are single paths, all forecasts are identical.

Output Arguments

`Y`	*numObs*-by-*numDims*-by-*numPaths* array of simulated paths.
`W`	*numObs*-by-*numDims*-by-*numPaths* array of innovations.

See Also

`vgxinfer` | `vgxpred` | `vgxproc`

Introduced in R2008b

vgxvarx

Estimate VARX model parameters

Syntax

```
EstSpec = vgxvarx(Spec,Y)

[EstSpec,EstStdErrors,LLF,W] = vgxvarx(Spec,Y,X,Y0,'Name1',Value1,'Name2',Value2,...);
```

Description

`vgxvarx` estimates parameters of VAR and VARX models using maximum likelihood estimation.

Required Input Arguments

`Spec`	A multivariate time series specification structure for an n-dimensional time series process, as created by `vgxset`. `Spec` must contain model dimensions, a lag structure (if any), and parameter estimation mappings (if any). It is not necessary to have initial values for any parameters to be estimated.
`Y`	A single path of an n-dimensional time series process with T observations for each path, collected in a T-by-n matrix. If `Y` contains multiple paths, `vgxvarx` uses only the first path to estimate parameters in `Spec`.

Optional Input Arguments

`X`	Exogenous inputs. nPX paths of regression design matrices associated with T observations of an n-dimensional time series process, where each design matrix linearly relates nX exogenous inputs to each time series at each observation time. `X` is a T-by-nPX matrix of cell arrays with n-by-nX design matrices in each cell. If `Y` has multiple paths, `X` must contain either a single

	path or no fewer than the same number of paths as in `Y`. Extra paths are ignored.
`Y0`	Presample time series process. *nPY0* presample paths of an *n*-dimensional time series process with *TY0* samples for each path, collected in a *TY0*-by-*n*-by-*nPY0* array. If `Y0` is empty or if *TY0* is less than the maximum AR lag in `Spec`, presample values are padded with zeros. If *TY0* is greater than the maximum AR lag, the most recent samples from the last rows of each path of `Y0` are used. If `Y0` has multiple paths, `Y0` must contain either a single path or no fewer than the same number of paths as in `Y`. Extra paths are ignored.

Specify the following optional input arguments as variable-length lists of matching parameter name/value pairs: `'Name1'`, `Value1`, `'Name2'`, `Value2`, ... and so on. The following rules apply when specifying parameter-name pairs:

- Specify the parameter name as a character string, followed by its corresponding parameter value.
- You can specify parameter name/value pairs in any order.
- Parameter names are case insensitive.
- You can specify unambiguous partial string matches.

The following table lists valid parameter name/value pairs.

Name	Value
`'CovarType'`	Form of the estimated covariance matrix. • `'full'` — estimate the full covariance matrix. This is the default. • `'diagonal'` — estimate a diagonal covariance matrix. This value overrides the value of `Qsolve` in `Spec`.
`'StdErrType'`	Form of the estimated standard errors. • `'mean'` — estimate only the standard errors associated with the parameters of the conditional mean. This is the default. • `'all'` — estimate the standard errors for all parameters, including the parameters for the innovations covariance. • `'none'` — do not estimate the standard errors.

Name	Value
'IgnoreMA'	Use of moving average terms in the specification structure. • 'no' — interpret moving average terms in the specification structure as errors. This is the default. • 'yes' — ignore moving average terms in the specification structure. For example, if a VARMA (1,1) model is specified with IgnoreMA set to 'yes', vgxvarx treats the model as a VAR(1) model with no moving average terms. If IgnoreMA is set to 'no', vgxvarx will not calibrate the model and will produce an error.
'MaxIter'	Maximum number of iterations for parameter estimation. The default is 1000. For ordinary least-squares (OLS) estimates of the parameters, set 'MaxIter' to 1. For feasible generalized least-squares (FGLS) estimates, set 'MaxIter' to 2. This parameter is used only for regression with exogenous inputs.
'TolParam'	Convergence tolerance for parameter estimation. The default is sqrt(eps).
'TolObj'	Convergence tolerance for parameter estimates. The default is eps^(3/4).

Output Arguments

EstSpec	A multivariate time series specification structure for an n-dimensional time series process that contains parameter estimates for a VAR or VARX model, as created by vgxset.
EstStdErrors	A multivariate time series specification structure containing standard errors of estimated parameters for the n-dimensional time series process EstSpec. EstStdErrors is not a specification of a VAR or VARX model; it contains standard errors that are mapped to the equivalent model parameters in EstSpec. If StdErrType is set to 'none', EstStdErrors is vgxset.empty. The standard errors are maximum likelihood estimates, so a degree-of-freedom adjustment is necessary to form ordinary least

	squares estimates. To adjust standard errors for degrees-of-freedom, multiply them by $\sqrt{\frac{T}{(T - NumActive - 1)}},$ where T is the number of observations of the time series process and *NumActive* is the number of unrestricted parameters that `vgxvarx` estimates.
`LLF`	The loglikelihood function with the maximum likelihood estimates of the model parameters from `EstSpec`.
`W`	Estimated innovations process. Since the estimation is based on a single path of `Y`, `W` contains the inferred path of an n-dimensional innovations process with T observations given the estimated model is `EstSpec`.

Examples

Estimate a Vector Autoregressive Process

Start with a 2-dimensional VARMA(2, 2) specification structure in `Spec` with presample data for the time series and innovations process:

```
load Data_VARMA22
```

The process in `Y` was generated with the known specification structure in `Spec` and is a VARMA(2, 2) process. Since `vgxvarx` calibrates VARX models, ignore the moving average component of the model and fit a pure VAR(2) model as an approximation. The function `vgxvarx` generates parameter estimates in `EstSpec` and standard errors in `EstStdErrors`:

```
[EstSpec, EstStdErrors] = vgxvarx(vgxar(Spec), Y, [], Y0);
```

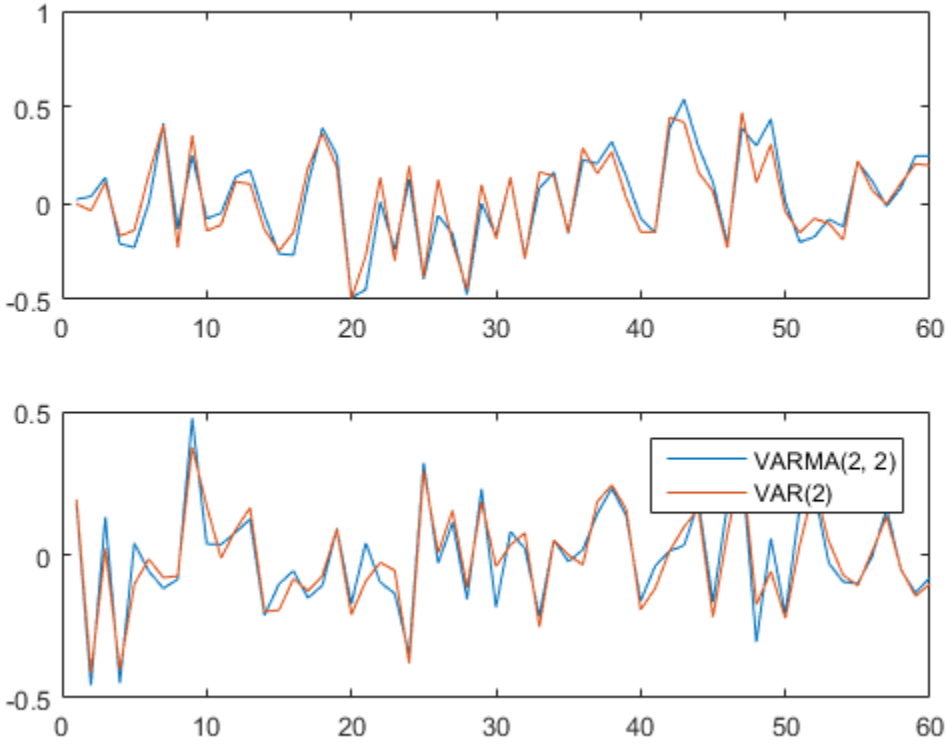
Use `vgxdisp` to display the estimated parameters along with standard errors and t-statistics:

```
vgxdisp(EstSpec, EstStdErrors)
```

```
Model  : 2-D VAR(2) with No Additive Constant
         Conditional mean is AR-stable and is MA-invertible
         Standard errors without DoF adjustment (maximum likelihood)
     Parameter          Value     Std. Error    t-Statistic
-------------- -------------- -------------- --------------
    AR(1)(1,1)       0.850166        0.12583        6.75649
         (1,2)     -0.0498191       0.163542      -0.304625
         (2,1)       0.219381      0.0934711        2.34705
         (2,2)     -0.0227752       0.121486      -0.187472
    AR(2)(1,1)      -0.294609       0.145514       -2.02461
         (1,2)       0.221336       0.148174        1.49376
         (2,1)       0.264504       0.108094        2.44699
         (2,2)      0.0819125       0.110069        0.74419
        Q(1,1)       0.051844
        Q(2,1)     0.00711775
        Q(2,2)      0.0286081
```

To see qualitatively how well the VARX approximation worked, infer the innovations process from the calibrated model and compare with the known innovations process from the VARMA(2, 2) process in `W`:

```
EstW = vgxinfer(EstSpec, Y, [], Y0, W0);

subplot(2,1,1);
plot([ W(:,1), EstW(:,1) ]);
subplot(2,1,2);
plot([ W(:,2), EstW(:,2) ]);
legend('VARMA(2, 2)', 'VAR(2)');
```



Try to calibrate a VAR(2) model with a restricted model such that the cross-terms in the AR lag matrices are fixed at 0 (this model treats each time series in `Y` as a separate VAR model but with a joint distribution for the innovations process):

```
SpecX = vgxset(vgxar(Spec),'AR',repmat({eye(2)},2,1), ...
 'ARsolve', repmat({ logical(eye(2)) }, 2, 1));

[EstSpecX, EstStdErrorsX] = vgxvarx(SpecX, Y, [], Y0);
```

Compare the calibrated restricted VAR(2) model with the previous unrestricted VAR(2) model to see that the cross-terms in the AR lag matrices have been fixed at 0 in the restricted model:

```
vgxdisp(EstSpecX, EstSpec);
```

```
Model 1: 2-D VAR(2) with No Additive Constant
         Conditional mean is AR-stable and is MA-invertible
Model 2: 2-D VAR(2) with No Additive Constant
         Conditional mean is AR-stable and is MA-invertible
     Parameter        Model 1        Model 2
-------------- -------------- --------------
    AR(1)(1,1)       0.789031       0.850166
         (1,2)              0     -0.0498191
         (2,1)              0       0.219381
         (2,2)       0.284343     -0.0227752
    AR(2)(1,1)      -0.251377      -0.294609
         (1,2)              0       0.221336
         (2,1)              0       0.264504
         (2,2)       0.178158      0.0819125
        Q(1,1)      0.0541056       0.051844
        Q(2,1)     0.00806393     0.00711775
        Q(2,2)      0.0410274      0.0286081
```

Introduced in R2008b

vratiotest

Variance ratio test for random walk

Syntax

```
h = vratiotest(y)
h = vratiotest(y,'ParameterName',ParameterValue,...)
[h,pValue] = vratiotest(...)
[h,pValue,stat] = vratiotest(...)
[h,pValue,stat,cValue] = vratiotest(...)
[h,pValue,stat,cValue,ratio] = vratiotest(...)
```

Description

`h = vratiotest(y)` assesses the null hypothesis of a random walk in a univariate time series `y`.

`h = vratiotest(y,'ParameterName',ParameterValue,...)` accepts optional inputs as one or more comma-separated parameter-value pairs. `'ParameterName'` is the name of the parameter inside single quotation marks. `ParameterValue` is the value corresponding to `'ParameterName'`. Specify parameter-value pairs in any order; names are case-insensitive. Perform multiple tests by passing a vector value for any parameter. Multiple tests yield vector results.

`[h,pValue] = vratiotest(...)` returns *p*-values of the test statistics.

`[h,pValue,stat] = vratiotest(...)` returns the test statistics.

`[h,pValue,stat,cValue] = vratiotest(...)` returns critical values for the tests.

`[h,pValue,stat,cValue,ratio] = vratiotest(...)` returns a vector of ratios.

Input Arguments

`y`

Vector of time-series data. The last element is the most recent observation. The test ignores `NaN` values, which indicate missing entries.

The input series `y` is in levels. To convert a return series `r` to levels, define `y(1)` and let `y = cumsum([y(1);r])`.

Name-Value Pair Arguments

`'alpha'`

Scalar or vector of nominal significance levels for the tests. Set values between `0` and `1`.

The test is two-tailed, so `vratiotest` rejects the random-walk null when the test statistic is outside of the critical interval `[-cValue,cValue]`. Each tail outside of the critical interval has probability `alpha`/2.

Default: `0.05`

`'IID'`

Scalar or vector of Boolean values indicating whether to assume independent identically distributed (IID) innovations.

To strengthen the null model and assume that the $e(t)$ are independent and identically distributed (IID), set `IID` to `true`.

The IID assumption is often unreasonable for long-term macroeconomic or financial price series. Rejection of the random-walk null due to heteroscedasticity is not interesting for these cases.

Default: `false`

`'period'`

Scalar or vector of integers greater than one and less than half the number of observations in `y`, indicating the period `q` used to create overlapping return horizons for the variance ratio.

When the period `q` has the default value of `2`, the first-order autocorrelation of the returns is asymptotically equal to `ratio–1`.

The test finds the largest integer `n` such that `n*q` ≤ `T–1`, where `T` is the sample size. It then discards the final `(T–1)–n*q` observations. To include these final observations, discard the initial `(T–1)–n*q` observations in `y` before running the test.

Default: `2`

Output Arguments

`h`

Vector of Boolean decisions for the tests, with length equal to the number of tests. Values of `h` equal to `1` indicate rejection of the random-walk null in favor of the alternative. Values of `h` equal to `0` indicate a failure to reject the random-walk null.

`pValue`

Vector of *p*-values of the test statistics, with length equal to the number of tests. Values are standard normal probabilities.

`stat`

Vector of test statistics, with length equal to the number of tests. Statistics are asymptotically standard normal.

`cValue`

Vector of critical values for the tests, with length equal to the number of tests. Values are for standard normal probabilities.

`ratio`

Vector of variance ratios, with length equal to the number of tests. Each ratio is the ratio of:

- The variance of the *q*-fold overlapping return horizon
- *q* times the variance of the return series

For a random walk, these ratios are asymptotically equal to one. For a mean-reverting series, the ratios are less than one. For a mean-averting series, the ratios are greater than one.

Definitions

The variance ratio test assesses the null hypothesis that a univariate time series y is a random walk. The null model is
$y(t) = c + y(t–1) + e(t)$,

where c is a drift constant and $e(t)$ are uncorrelated innovations with zero mean.

- When `IID` is `false`, the alternative is that the $e(t)$ are correlated.
- When `IID` is `true`, the alternative is that the $e(t)$ are either dependent or not identically distributed (for example, heteroscedastic).

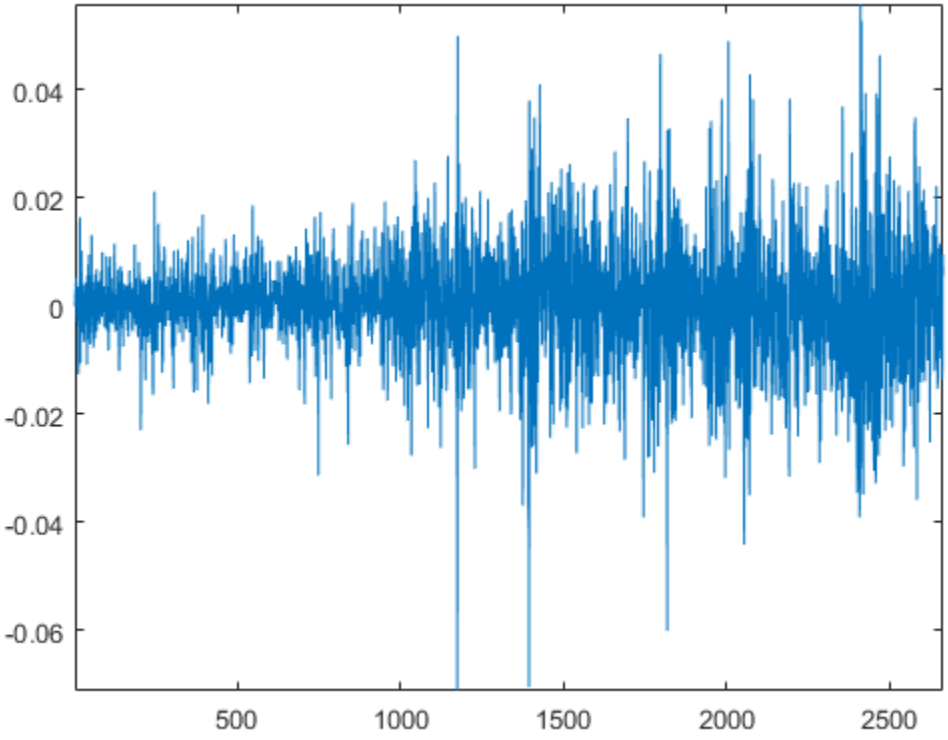
Examples

Assess Whether a Series Is a Random Walk

Test whether a US equity index is a random walk using various step sizes. Perform the test with and without the assumption that the innovations are independent and identically distributed.

Load the global large-cap equity indices data set. Focus on the daily S & P 500 index (`SP`).

```
load Data_GlobalIdx1
logSP = log(DataTable.SP);

figure
plot(diff(logSP))
axis tight
```



The plot indicates possible conditional heteroscedasticity.

Test whether the series is a random walk using various periods and whether the innovations are independent and identically distributed.

```
q = [2 4 8 2 4 8];
flag = logical([1 1 1 0 0 0]);
[h,pValue,stat,cValue,ratio] = ...
        vratiotest(logSP,'period',q,'IID',flag)
rho1 = ratio(1)-1 % First-order autocorrelation of returns
```

```
h =
```

```
     0     0     1     0     0     0

pValue =

    0.5670    0.3307    0.0309    0.7004    0.5079    0.1303

stat =

    0.5724   -0.9727   -2.1579    0.3847   -0.6621   -1.5128

cValue =

    1.9600    1.9600    1.9600    1.9600    1.9600    1.9600

ratio =

    1.0111    0.9647    0.8763    1.0111    0.9647    0.8763

rho1 =

    0.0111
```

`h` indicates that the test fails to reject that the series is a random walk at 5% level, except in the case where `period` = 8 and `IID` = `true`. This rejection is likely due to the test not accounting for the heteroscedasticity.

More About

Algorithms

The `vratiotest` test statistics are based on a ratio of variance estimates of returns $r(t) = y(t)–y(t–1)$ and period q return horizons $r(t) + ... + r(t–q+1)$. Overlapping horizons increase the efficiency of the estimator and add power to the test. Under either null, uncorrelated innovations $e(t)$ imply that the period q variance is asymptotically equal to q times the period 1 variance. The variance of the ratio, however, depends on the degree of heteroscedasticity, and, therefore, on the null.

Rejection of the null due to dependence of the innovations does not imply that the $e(t)$ are correlated. Dependence allows that nonlinear functions of the $e(t)$ are correlated, even when the $e(t)$ are not. For example, it can hold that $\text{Cov}[e(t),e(t–k)] = 0$ for all $k \neq 0$, while $\text{Cov}[e(t)^2,e(t–k)^2] \neq 0$ for some $k \neq 0$.

Cecchetti and Lam [2] show that sequential testing using multiple values of q results in small-sample size distortions beyond those that result from the asymptotic approximation of critical values.

- "Unit Root Nonstationarity" on page 3-34

References

[1] Campbell, J. Y., A. W. Lo, and A. C. MacKinlay. Chapter 12. "The Econometrics of Financial Markets." *Nonlinearities in Financial Data*. Princeton, NJ: Princeton University Press, 1997.

[2] Cecchetti, S. G., and P. S. Lam. "Variance-Ratio Tests: Small-Sample Properties with an Application to International Output Data." *Journal of Business and Economic Statistics*. Vol. 12, 1994, pp. 177–186.

[3] Cochrane, J. "How Big is the Random Walk in GNP?" *Journal of Political Economy*. Vol. 96, 1988, pp. 893–920.

[4] Faust, J. "When Are Variance Ratio Tests for Serial Dependence Optimal?" *Econometrica*. Vol. 60, 1992, pp. 1215–1226.

[5] Lo, A. W., and A. C. MacKinlay. "Stock Market Prices Do Not Follow Random Walks: Evidence from a Simple Specification Test." *Review of Financial Studies*. Vol. 1, 1988, pp. 41–66.

[6] Lo, A. W., and A. C. MacKinlay. "The Size and Power of the Variance Ratio Test." *Journal of Econometrics*. Vol. 40, 1989, pp. 203–238.

[7] Lo, A. W., and A. C. MacKinlay. *A Non-Random Walk Down Wall St.* Princeton, NJ: Princeton University Press, 2001.

See Also

`adftest` | `pptest` | `kpsstest` | `lmctest`

Introduced in R2009b

waldtest

Wald test of model specification

Syntax

```
h = waldtest(r,R,EstCov)
h = waldtest(r,R,EstCov,alpha)
[h,pValue] = waldtest( ___ )
[h,pValue,stat,cValue] = waldtest( ___ )
```

Description

`h = waldtest(r,R,EstCov)` returns a logical value (`h`) with the rejection decision from conducting a Wald test of model specification.

`waldtest` constructs the test statistic using the restriction function and its Jacobian, and the value of the unrestricted model covariance estimator, all evaluated at the unrestricted parameter estimates (`r`, `R`, and `EstCov`, respectively).

- If any input argument is a cell vector of length $k > 1$, then the other input arguments must be cell arrays of length k. `waldtest(r,R,EstCov)` treats each cell as a separate, independent test, and returns a vector of rejection decisions.
- If any input argument is a row vector, then the software returns output arguments as row vectors.

`h = waldtest(r,R,EstCov,alpha)` returns the rejection decision of the Wald test conducted at significance level `alpha`.

`[h,pValue] = waldtest( ___ )` returns the rejection decision and p-value (`pValue`) for the hypothesis test, using any of the input arguments in the previous syntaxes.

`[h,pValue,stat,cValue] = waldtest( ___ )` additionally returns the test statistic (`stat`) and critical value (`cValue`) for the hypothesis test.

Examples

Assess Model Specifications Using the Wald Test

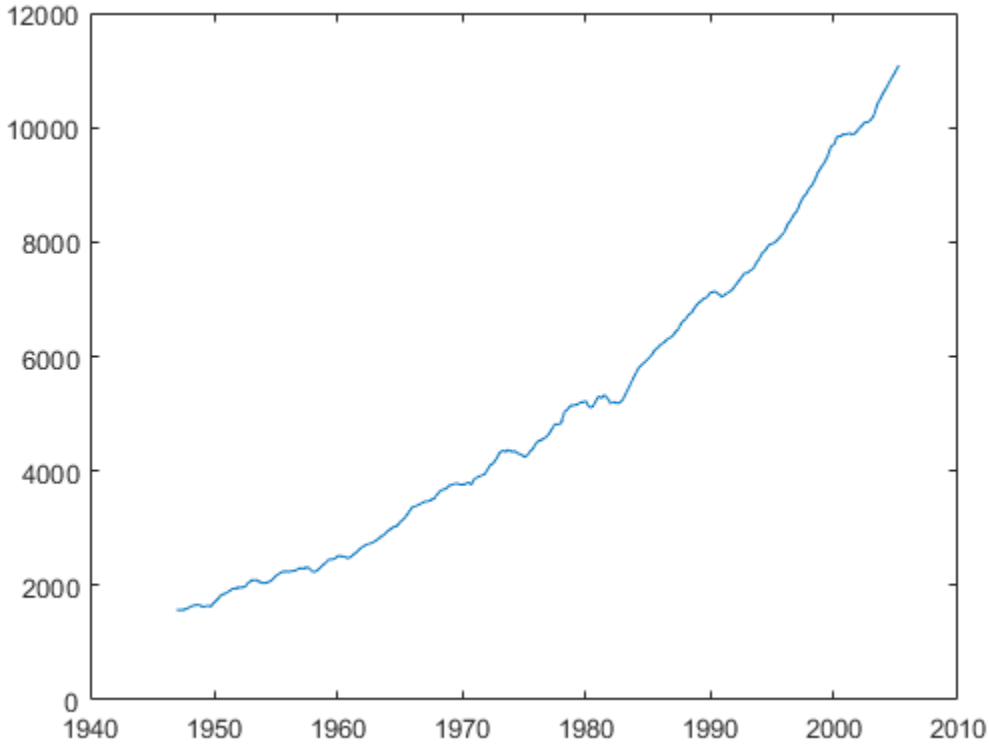
Check for significant lag effects in a time series regression model.

Load the U.S. GDP data set.

```
load Data_GDP
```

Plot the GDP against time.

```
plot(dates,Data)
datetick
```



The series seems to increase exponentially.

Transform the data using the natural logarithm.

```
logGDP = log(Data);
```

`logGDP` is increasing in time, so assume that there is a significant lag 1 effect. To use the Wald test to check if there is a significant lag 2 effect, you need the:

- Estimated coefficients of the unrestricted model
- Restriction function evaluated at the unrestricted model coefficient values
- Jacobian of the restriction function evaluated at the unrestricted model coefficient values
- Estimated, unrestricted parameter covariance matrix.

The unrestricted model is

$$y_t = \beta_0 + \beta_1 y_{t-1} + \beta_2 y_{t-2} + \varepsilon_t.$$

Estimate the coefficients of the unrestricted model.

```
LagLGDP = lagmatrix(logGDP,1:2);
UMdl = fitlm(table(LagLGDP(:,1),LagLGDP(:,2),logGDP));
```

`UMdl` is a fitted `LinearModel` model. It contains, among other things, the fitted coefficients of the unrestricted model.

The restriction is $\beta_2 = 0$. Therefore, the restiction function (r) and Jacobian (R) are:

- $r = \beta_2$
- $R = [0 \quad 0 \quad 1]$

Specify r, R, and the estimated, unrestricted parameter covariance matrix.

```
r = UMdl.Coefficients.Estimate(3);
R = [0 0 1];
EstParamCov = UMdl.CoefficientCovariance;
```

Test for a significant lag 2 effect using the Wald test.

```
[h,pValue] = waldtest(r,R,EstParamCov)
```

```
h =

     1


pValue =

   1.2521e-07
```

`h = 1` indicates that the null, restricted hypothesis ($\beta_2 = 0$) should be rejected in favor of the alternative, unrestricted hypothesis. `pValue` is quite small, which suggests that there is strong evidence for this result.

Assess Conditional Heteroscedasticity Using the Wald Test

Test whether there are significant ARCH effects in a simulated response series using `waldtest`.

Suppose that the model for the simulated data is AR(1) with an ARCH(1) variance. Symbolically, the model is

$$y_t = 0.9y_{t-1} + \varepsilon_t,$$

where

- $\varepsilon_t = w_t\sqrt{h_t}$
- $h_t = 1 + 0.5\varepsilon_{t-1}^2$
- w_t is Gaussian with mean 0 and variance 1.

Specify the model for the simulated data.

```
VarMdl = garch('ARCH',0.5,'Constant',1);
Mdl = arima('Constant',0,'Variance',VarMdl,'AR',0.9);
```

`Mdl` is a fully specified AR(1) model with an ARCH(1) variance.

Simulate presample and effective sample responses from `Mdl`.

```
T = 100;
rng(1);  % For reproducibility
n = 2;   % Number of presample observations required for the Jacobian
[y,epsilon,condVariance] = simulate(Mdl,T + n);
```

```
psI = 1:n;              % Presample indices
esI = (n + 1):(T + n); % Estimation sample indices
```

`epsilon` is the random path of innovations from `VarMdl`. The software filters `epsilon` through `Mdl` to yield the random response path `y`.

Specify the unrestricted model assuming that the conditional mean model is

$$y_t = c + \phi_1 y_{t-1} + \varepsilon_t,$$

where $h_t = \alpha_0 + \alpha_1 \varepsilon_{t-1}^2$. Fit the simulated data (`y`) to the unrestricted model using the presample observations.

```
UVarMdl = garch(0,1);
UMdl = arima('ARLags',1,'Variance',UVarMdl);
[UEstMdl,UEstParamCov] = estimate(UMdl,y(esI),'Y0',y(psI),...
    'E0',epsilon(psI),'V0',condVariance(psI),'Display','off');
```

`UEstMdl` is the the fitted, unrestricted model, and `UEstParamCov` is the estimated parameter covariance of the unrestricted model parameters.

The null hypothesis is that $\alpha_1 = 0$, i.e., the restricted model is AR(1) with Gaussian innovations that have mean 0 and constant variance. Therefore, the restriction function is $r(\theta) = \alpha_1$, where $\theta = [c, \phi_1, \alpha_0, \alpha_1]'$. The components of the Wald test are:

- The restriction function evaluated at the unrestricted parameter estimates is $r = \hat{\alpha}_1$.
- The Jacobian of *r* evaluated at the unrestricted model parameters is $R = \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix}$.
- The unrestricted model estimated parameter covariance matrix is `UEstParamCov`.

Specify *r* and *R*.

```
r = UEstMdl.Variance.ARCH{1};
R = [0, 0, 0, 1];
```

Test the null hypothesis that $\alpha_1 = 0$ at the 1% significance level using `waldtest`.

```
[h,pValue,stat,cValue] = waldtest(r,R,UEstParamCov,0.01)
```

```
h =
```

```
     0

pValue =

    0.0549

stat =

    3.6846

cValue =

    6.6349
```

`h = 0` indicates that the null, restricted model should not be rejected in favor of the alternative, unrestricted model. This result is consistent with the model for the simulated data.

Test Among Multiple Nested Model Specifications

Assess model specifications by testing down among multiple restricted models using simulated data. The true model is the ARMA(2,1)

$$y_t = 3 + 0.9y_{t-1} - 0.5y_{t-2} + \varepsilon_t + 0.7\varepsilon_{t-1},$$

where ε_t is Gaussian with mean 0 and variance 1.

Specify the true ARMA(2,1) model, and simulate 100 response values.

```
TrueMdl = arima('AR',{0.9,-0.5},'MA',0.7,...
    'Constant',3,'Variance',1);
T = 100;
rng(1); % For reproducibility
y = simulate(TrueMdl,T);
```

Specify the unrestricted model and the names of the candidate models for testing down.

```
UMdl = arima(2,0,2);
RMdlNames = {'ARMA(2,1)','AR(2)','ARMA(1,2)','ARMA(1,1)',...
    'AR(1)','MA(2)','MA(1)'};
```

`UMdl` is the unrestricted, ARMA(2,2) model. `RMdlNames` is a cell array of strings containing the names of the restricted models.

Fit the unrestricted model to the simulated data.

```
[UEstMdl,UEstParamCov] = estimate(UMdl,y,'Display','off');
```

`UEstMdl` is the fitted, unrestricted model, and `UEstParamCov` is the estimated parameter covariance matrix.

The unrestricted model has six parameters. To construct the restriction function and its Jacobian, you must know the order of the parameters in `UEstParamCov`. For this `arima` model, the order is $[c, \phi_1, \phi_2, \theta_1 \theta_2, \sigma^2]$.

Each candidate model corresponds to a restriction function. Put the restriction function vectors into separate cells of a cell vector.

```
rf1 = UEstMdl.MA{2};                            % ARMA(2,1)
rf2 = cell2mat(UEstMdl.MA)';                    % AR(2)
rf3 = UEstMdl.AR{2};                            % ARMA(1,2)
rf4 = [UEstMdl.AR{2};UEstMdl.MA{2}]';           % ARMA(1,1)
rf5 = [UEstMdl.AR{2};cell2mat(UEstMdl.MA)'];    % AR(1)
rf6 = cell2mat(UEstMdl.AR)';                    % MA(2)
rf7 = [cell2mat(UEstMdl.AR)';UEstMdl.MA{2}];    % MA(1)
r = {rf1;rf2;rf3;rf4;rf5;rf6;rf7};
```

`r` is a 7-by-1 cell vector of vectors corresponding to the restriction function for the candidate models.

Put the Jacobian of each restriction function into separate, corresponding cells of a cell vector. The order of the elements in the Jacobian must correspond to the order of the elements in `UEstParamCov`.

```
J1 = [0 0 0 0 1 0];                             % ARMA(2,1)
J2 = [0 0 0 1 0 0; 0 0 0 0 1 0];                % AR(2)
J3 = [0 1 0 0 0 0];                             % ARMA(1,2)
J4 = [0 1 0 0 0 0; 0 0 0 0 1 0];                % ARMA(1,1)
J5 = [0 1 0 0 0 0; 0 0 0 1 0 0; 0 0 0 0 1 0]; % AR(1)
J6 = [1 0 0 0 0 0; 0 1 0 0 0 0];                % MA(2)
J7 = [1 0 0 0 0 0; 0 1 0 0 0 0; 0 0 0 0 1 0]; % MA(1)
R = {J1;J2;J3;J4;J5;J6;J7};
```

`R` is a 7-by-1 cell vector of vectors corresponding to the restriction function for the candidate models.

Put the estimated parameter covariance matrix in each cell of a 7-by-1 cell vector.

```
EstCov = cell(7,1); % Preallocate
for j = 1:length(EstCov)
    EstCov{j} = UEstParamCov;
end
```

Apply the Wald test at a 1% significance level to find the appropriate, restricted model specifications.

```
alpha = .01;
h = waldtest(r,R,EstCov,alpha);
RestrictedModels = RMdlNames(~h)
```

```
RestrictedModels =

    'ARMA(2,1)'    'ARMA(1,2)'    'ARMA(1,1)'    'MA(2)'    'MA(1)'
```

`RestrictedModels` lists the most appropriate restricted models.

You can test down again, but use ARMA(2,1) as the unrestricted model. In this case, you must remove MA(2) from the possible restricted models.

Conduct a Wald Test Using Nonlinear Restriction Functions

Test whether the parameters of a nested model have a nonlinear relationship.

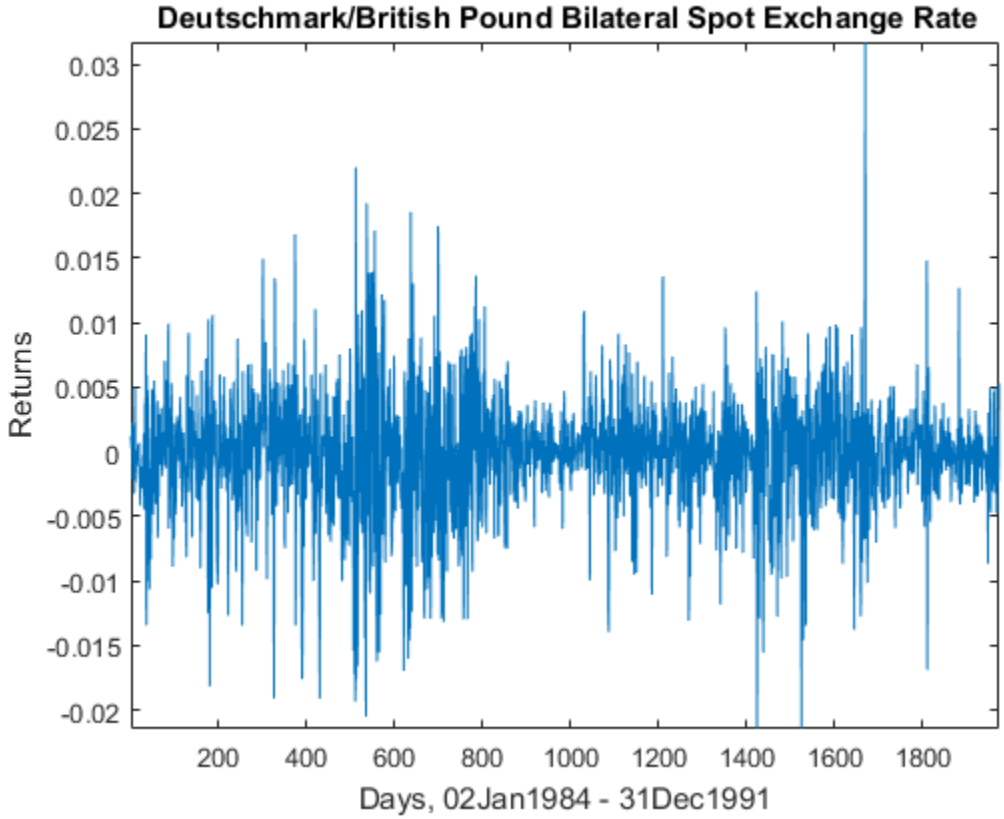
Load the Deutschmark/British Pound bilateral spot exchange rate data set.

```
load Data_MarkPound
```

The data set (`Data`) contains a time series of prices.

Convert the prices to returns, and plot the return series.

```
returns = price2ret(Data);

figure
plot(returns)
axis tight
ylabel('Returns')
xlabel('Days, 02Jan1984 - 31Dec1991')
title('{\bf Deutschmark/British Pound Bilateral Spot Exchange Rate}')
```



The returns series shows signs of heteroscedasticity.

Suppose that a GARCH(1,1) model is an appropriate model for the data. Fit a GARCH(1,1) model to the data including a constant.

```
Mdl = garch(1,1);
[EstMdl,EstParamCov] = estimate(Mdl,returns);
g1 = EstMdl.GARCH{1};
a1 = EstMdl.ARCH{1};
```

```
    GARCH(1,1) Conditional Variance Model:
    ----------------------------------------
    Conditional Probability Distribution: Gaussian
```

```
                          Standard          t
    Parameter       Value          Error       Statistic
    -----------   -----------   ------------   -----------
    Constant    1.05346e-06   3.50483e-07        3.00575
    GARCH{1}       0.806576     0.0129095        62.4794
     ARCH{1}       0.154357     0.0115746        13.3358
```

`g1` is the estimated GARCH effect, and `a1` is the estimated ARCH effect.

The following might represent relationships between the GARCH and ARCH coefficients:

- $\gamma_1\alpha_1 = 1$
- $\gamma_1 + \alpha_1 = 1$

where γ_1 is the GARCH effect and α_1 is the ARCH effect. Specify these relationships as the restriction function $r(\theta) = 0$, evaluated at the unrestricted model parameter estimates. This specification defines a nested, restricted model.

```
r = [g1*a1; g1+a1] - 1;
```

Specify the Jacobian of the restriction function vector.

```
R = [0, a1, g1;0, 1, 1];
```

Conduct a Wald test to assess whether there is sufficient evidence to reject the restricted model.

```
[h,pValue,stat,cValue] = waldtest(r,R,EstParamCov)
```

```
h =

     1


pValue =

     0


stat =

   1.4594e+04
```

```
cValue =

    5.9915
```

`h = 1` indicates that there is sufficient evidence to reject the restricted model in favor of the unrestricted model. `pValue = 0` indicates that the evidence for rejecting the restricted model is strong.

- "Conduct a Wald Test" on page 3-74
- "Classical Model Misspecification Tests"
- "Time Series Regression IX: Lag Order Selection"

Input Arguments

`r` — Restriction functions

scalar | vector | cell vector of scalars or vectors

Restriction functions corresponding to restricted models, specified as a scalar, vector, or cell vector of scalars or vectors.

- If `r` is a q-vector or a singleton cell array containing a q-vector, then the software conducts one Wald test. q must be less than the number of unrestricted model parameters.
- If `r` is a cell vector of length $k > 1$, and cell j contains a q_j-vector, $j = 1,...,k$, then the software conducts k independent Wald tests. Each q_j must be less than the number of unrestricted model parameters.

Data Types: `double` | `cell`

`R` — Restriction function Jacobians

row vector | matrix | cell vector of row vectors or matrices

Restriction function Jacobians, specified as a row vector, matrix, or cell vector of row vectors or matrices.

- Suppose $r_1,...,r_q$ are the q restriction functions, and the unrestricted model parameters are $\theta_1,...,\theta_p$. Then, the restriction function Jacobian is

$$R = \begin{pmatrix} \frac{\partial r_1}{\partial \theta_1} & \cdots & \frac{\partial r_1}{\partial \theta_p} \\ \vdots & \ddots & \vdots \\ \frac{\partial r_q}{\partial \theta_1} & \cdots & \frac{\partial r_q}{\partial \theta_p} \end{pmatrix}.$$

- If `R` is a q-by-p matrix or a singleton cell array containing a q-by-p matrix, then the software conducts one Wald test. q must be less than p, which is the number of unrestricted model parameters.
- If `R` is a cell vector of length $k > 1$, and cell j contains a q_j-by-p_j matrix, $j = 1,...,k$, then the software conducts k independent Wald tests. Each q_j must be less than p_j, which is the number of unrestricted parameters in model j.

Data Types: `double` | `cell`

`EstCov` — Unrestricted model parameter covariance estimate

matrix | cell vector of matrices

Unrestricted model parameter covariance estimates, specified as a matrix or cell vector of matrices.

- If `EstCov` is a p-by-p matrix or a singleton cell array containing a p-by-p matrix, then the software conducts one Wald test. p is the number of unrestricted model parameters.
- If `EstCov` is a cell vector of length $k > 1$, and cell j contains a p_j-by-p_j matrix, $j = 1,...,k$, then the software conducts k independent Wald tests. Each p_j is the number of unrestricted parameters in model j.

Data Types: `double` | `cell`

`alpha` — Nominal significance levels

0.05 (default) | scalar | vector

Nominal significance levels for the hypothesis tests, specified as a scalar or vector.

Each element of `alpha` must be greater than 0 and less than 1.

When conducting $k > 1$ tests,

- If `alpha` is a scalar, then the software expands it to a k-by-1 vector.

- If `alpha` is a vector, then it must have length k.

Data Types: `double`

Output Arguments

`h` — Test rejection decisions

logical | vector of logicals

Test rejection decisions, returned as a logical value or vector of logical values with a length equal to the number of tests that the software conducts.

- `h = 1` indicates rejection of the null, restricted model in favor of the alternative, unrestricted model.
- `h = 0` indicates failure to reject the null, restricted model.

`pValue` — Test statistic *p*-values

scalar | vector

Test statistic p-values, returned as a scalar or vector with a length equal to the number of tests that the software conducts.

`stat` — Test statistics

scalar | vector

Test statistics, returned as a scalar or vector with a length equal to the number of tests that the software conducts.

`cValue` — Critical values

scalar | vector

Critical values determined by `alpha`, returned as a scalar or vector with a length equal to the number of tests that the software conducts.

More About

Wald Test

The *Wald test* compares specifications of nested models by assessing the significance of q parameter restrictions to an extended model with p unrestricted parameters.

The test statistic is

$$W = r'\left(R\Sigma_{\hat{\theta}}R'\right)^{-1}r,$$

where

- r is the restriction function that specifies restrictions of the form $r(\theta) = 0$ on parameters θ in the unrestricted model, evaluated at the unrestricted model parameter estimates. In other words, r maps the p-dimensional parameter space to the q-dimensional restriction space.

  In practice, r is a q-by-1 vector, where $q < p$.

  Usually, $r = \hat{\theta} - \theta_0$, where $\hat{\theta}$ is the unrestricted model parameter estimates for the restricted parameters and θ_0 holds the values of the restricted model parameters under the null hypothesis.
- R is the restriction function Jacobian evaluated at the unrestricted model parameter estimates.
- $\hat{\Sigma}_{\hat{\theta}}$ is the unrestricted model parameter covariance estimator evaluated at the unrestricted model parameter estimates.
- W has an asymptotic chi-square distribution with q degrees of freedom.

When W exceeds a critical value in its asymptotic distribution, the test rejects the null, restricted hypothesis in favor of the alternative, unrestricted hypothesis. The nominal significance level (α) determines the critical value.

Note: Wald tests depend on the algebraic form of the restrictions. For example, you can express the restriction $ab = 1$ as $a - 1/b = 0$, or $b - 1/a = 0$, or $ab - 1 = 0$. Each formulation leads to different test statistics.

Tips

- Estimate unrestricted univariate linear time series models, such as `arima` or `garch`, or time series regression models (`regARIMA`) using `estimate`. Estimate unrestricted multivariate linear time series models using `vgxvarx`.

`estimate` and `vgxvarx` return parameter estimates and their covariance estimates, which you can process and use as inputs to `waldtest`.

- If you cannot easily compute restricted parameter estimates, then use `waldtest`. By comparison:
  - `lratiotest` requires both restricted and unrestricted parameter estimates.
  - `lmtest` requires restricted parameter estimates.

Algorithms

- `waldtest` performs multiple, independent tests when the restriction function vector, its Jacobian, and the unrestricted model parameter covariance matrix (`r`, `R`, and `EstCov`, respectively) are equal-length cell vectors.
  - If `EstCov` is the same for all tests, but `r` varies, then `waldtest` "tests down" against multiple restricted models.
  - If `EstCov` varies among tests, but `r` does not, then `waldtest` "tests up" against multiple unrestricted models.
  - Otherwise, `waldtest` compares model specifications pair-wise.
- `alpha` is nominal in that it specifies a rejection probability in the asymptotic distribution. The actual rejection probability is generally greater than the nominal significance.
- The Wald test rejection error is generally greater than the likelihood ratio and Lagrange multiplier test rejection errors.

- Using garch Objects
- "Model Comparison Tests" on page 3-65

References

[1] Davidson, R. and J. G. MacKinnon. *Econometric Theory and Methods*. Oxford, UK: Oxford University Press, 2004.

[2] Godfrey, L. G. *Misspecification Tests in Econometrics*. Cambridge, UK: Cambridge University Press, 1997.

[3] Greene, W. H. *Econometric Analysis*. 6th ed. Upper Saddle River, NJ: Pearson Prentice Hall, 2008.

[4] Hamilton, J. D. *Time Series Analysis*. Princeton, NJ: Princeton University Press, 1994.

See Also

arima | estimate | `estimate` | estimate | `garch` | `lmtest` | regARIMA | `vgxvarx`

Introduced in R2009a

Data Sets and Examples

Econometrics Toolbox includes the sample data sets and examples in the following tables.

Generally, the data sets contain individual data variables, description variables with references, and tablular arrays encapsulating the data set and its description, as appropriate. To load a data set into the workspace, type

`load Data_Filename`,
where `Data_Filename` is one of the files listed in the table.

Data Set Name	Description
`Data_Canada`	Canadian inflation and interest rates, 1954–1994
`Data_Consumption`	U.S. food consumption, 1927-1962
`Data_CreditDefaults`	Investment-grade corporate bond defaults and four predictors, 1984–2004
`Data_Danish`	Danish stock returns, bond yields, 1922–1999
`Data_EquityIdx`	U.S. equity indices, 1990–2001
`Data_FXRates`	Currency exchange rates, 1979–1998
`Data_GDP`	U.S. Gross Domestic Product, 1947–2005
`Data_GlobalIdx1`	Global large-cap equity indices, 1993–2003
`Data_GNP`	U.S. Gross National Product, 1947–2005
`Data_Income1`	Simulated data on income and education
`Data_Income2`	Average annual earnings by educational attainment in eight workforce age categories
`Data_JAustralian`	Johansen's Australian data, 1972–1991
`Data_JDanish`	Johansen's Danish data, 1974–1987
`Data_MarkPound`	Deutschmark/British Pound foreign-exchange rate, 1984–1991
`Data_NelsonPlosser`	Macroeconomic series of Nelson and Plosser, 1860–1970

Data Set Name	Description
`Data_SchwertMacro`	Macroeconomic series of Schwert, 1947–1985
`Data_SchwertStock`	Indices of U.S. stock prices, 1871–2008
`Data_TBill`	Three-month U.S. treasury bill secondary market rates, 1947–2005
`Data_USEconModel`	US Macroeconomic series, 1947–2009
`Data_VARMA22`	Two-dimensional VARMA(2,2) specification
`Data_Recessions`	U.S. recession start and end dates, 1857–2011

After loading the data set, you can display information about the data set, e.g., the meanings of the variables, by entering `Description` in the command line.

Example Name	Description
`Demo_ClassicalTests`	Performing classical model misspecification tests
`Demo_DieboldLiModel`	Using the State-Space Model (SSM) and Kalman filter to fit the Diebold-Li yields-only model to yield curves derived from government bond data
`Demo_HPFilter`	Using the Hodrick-Prescott filter to reproduce their original result
`Demo_RiskFHS`	Using bootstrapping and filtered historical simulation to evaluate market risk
`Demo_RiskEVT`	Using extreme value theory and copulas to evaluate market risk
`Demo_TSReg1`	Introducing basic assumptions behind multiple linear regression models
`Demo_TSReg2`	Detecting correlation among predictors and accommodate problems of large estimator variance
`Demo_TSReg3`	Detecting influential observations in time series data and accommodate their effect on multiple linear regression models

Example Name	Description
`Demo_TSReg4`	Investigating trending variables, spurious regression, and methods of accommodation in multiple linear regression models
`Demo_TSReg5`	Selecting a parsimonious set of predictors with high statistical significance for multiple linear regression models
`Demo_TSReg6`	Evaluating model assumptions and investigate respecification opportunities by examining the series of residuals
`Demo_TSReg7`	Presenting the basic setup for producing conditional and unconditional forecasts from multiple linear regression models
`Demo_TSReg8`	Examining how lagged predictors affect least-squares estimation of multiple linear regression models
`Demo_TSReg9`	Illustrating predictor history selection for multiple linear regression models
`Demo_TSReg10`	Estimating multiple linear regression models of time series data in the presence of heteroscedastic or autocorrelated innovations
`Demo_USEconModel`	Modeling the United States economy

Glossary

Akaike information criteria (AIC)

A model-order selection criterion based on parsimony. More complicated models are penalized for the inclusion of additional parameters. *See also* **Bayesian information criteria (BIC)**.

antithetic sampling

A variance reduction technique that pairs a sequence of independent normal random numbers with a second sequence obtained by negating the random numbers of the first. The first sequence simulates increments of one path of Brownian motion, and the second sequence simulates increments of its reflected, or antithetic, path. These two paths form an antithetic pair independent of any other pair.

AR

Autoregressive. AR models include past observations of the dependent variable in the forecast of future observations.

ARCH

Autoregressive Conditional Heteroscedasticity. A time series technique that uses past observations of the variance to forecast future variances. *See also* **GARCH**.

ARMA

Autoregressive Moving Average. A time series model that includes both AR and MA components. *See also* **AR** and **MA**.

autocorrelation function (ACF)

Correlation sequence of a random time series with itself. See also **cross-correlation function (XCF)**.

autoregressive

See **AR**.

Bayesian information criteria (BIC)

A model-order selection criterion based on parsimony. More complicated models are penalized for the inclusion of additional parameters. Since BIC imposes a greater penalty for additional parameters than AIC, BIC always provides a model with a number of parameters no greater than that chosen by AIC. *See also* **Akaike information criteria (AIC)**.

Brownian motion

A zero-mean continuous-time stochastic process with independent increments (also known as a *Wiener process*).

conditional — Time series technique with explicit dependence on the past sequence of observations.

conditional mean — time series model for forecasting the expected value of the return series itself.

conditional variance — Time series model for forecasting the expected value of the variance of the return series.

cross-correlation function (XCF) — Correlation sequence between two random time series. *See also* **autocorrelation function (ACF)**.

diffusion — The function that characterizes the random (stochastic) portion of a stochastic differential equation. *See also* **stochastic differential equation**.

discretization error — Errors that may arise due to discrete-time sampling of continuous stochastic processes.

drift — The function that characterizes the deterministic portion of a stochastic differential equation. *See also* **stochastic differential equation**.

equality constraint — A constraint, imposed during parameter estimation, by which a parameter is held fixed at a user-specified value.

Euler approximation — A simulation technique that provides a discrete-time approximation of a continuous-time stochastic process.

excess kurtosis — A characteristic, relative to a standard normal probability distribution, in which an area under the probability density function is reallocated from the center of the distribution to the tails (fat tails). Samples obtained from distributions with excess kurtosis have a higher probability of containing outliers than samples drawn from a normal (Gaussian) density. Time series that exhibit a fat tail distribution are often referred to as leptokurtic.

explanatory variables — Time series used to explain the behavior of another observed series of interest. Explanatory variables are typically incorporated into a regression framework.

fat tails — *See* **excess kurtosis**.

GARCH — Generalized autoregressive conditional heteroscedasticity. A time series technique that uses past observations of the variance and variance forecast to forecast future variances. *See also* **ARCH**.

heteroscedasticity — Time-varying, or time-dependent, variance.

homoscedasticity — Time-independent variance. The Econometrics Toolbox software also refers to homoscedasticity as constant conditional variance.

i.i.d. — Independent, identically distributed.

innovations — A sequence of unanticipated shocks, or disturbances. The Econometrics Toolbox software uses innovations and residuals interchangeably.

leptokurtic — *See* **excess kurtosis**.

MA — Moving average. MA models include past observations of the innovations noise process in the forecast of future observations of the dependent variable of interest.

MMSE — Minimum mean square error. A technique designed to minimize the variance of the estimation or forecast error. *See also* **RMSE**.

moving average — *See* **MA**.

objective function — The function to numerically optimize. In the Econometrics Toolbox software, the objective function is the loglikelihood function of a random process.

partial autocorrelation function (PACF) — Correlation sequence estimated by fitting successive order autoregressive models to a random time series by least squares. The PACF is useful for identifying the order of an autoregressive model.

path — A random trial of a time series process.

proportional sampling — A stratified sampling technique that ensures that the proportion of random draws matches its theoretical probability. One of the most common examples of proportional sampling involves stratifying the terminal value of a price process in which each sample path is associated with a single stratified terminal value such that the number of paths equals the number of strata.

See also **stratified sampling**.

p-value — The lowest level of significance at which a test statistic is significant.

realization — *See* **path**.

residuals — *See* **innovations**.

RMSE — Root mean square error. The square root of the mean square error. *See also* **MMSE**.

standardized innovations — The innovations divided by the corresponding conditional standard deviation.

stochastic differential equation — A generalization of an ordinary differential equation, with the addition of a noise process, that yields random variables as solutions.

strata — *See* **stratified sampling**.

stratified sampling — A variance reduction technique that constrains a proportion of sample paths to specific subsets (or *strata*) of the sample space.

time series — Discrete-time sequence of observations of a random process. The type of time series of interest in the Econometrics Toolbox software is typically a series of returns, or relative changes of some underlying price series.

transient — A response, or behavior, of a time series that is heavily dependent on the initial conditions chosen to begin a recursive calculation. The transient response is typically

	undesirable, and initially masks the true steady-state behavior of the process of interest.
trial	The result of an independent random experiment that computes the average or expected value of a variable of interest and its associated confidence interval.
unconditional	Time series technique in which explicit dependence on the past sequence of observations is ignored. Equivalently, the time stamp associated with any observation is ignored.
variance reduction	A sampling technique in which a given sequence of random variables is replaced with another of the same expected value but smaller variance. Variance reduction techniques increase the efficiency of Monte Carlo simulation.
volatility	The risk, or uncertainty, measure associated with a financial time series. The Econometrics Toolbox software associates volatility with standard deviation.
Wiener process	*See* **Brownian motion**.