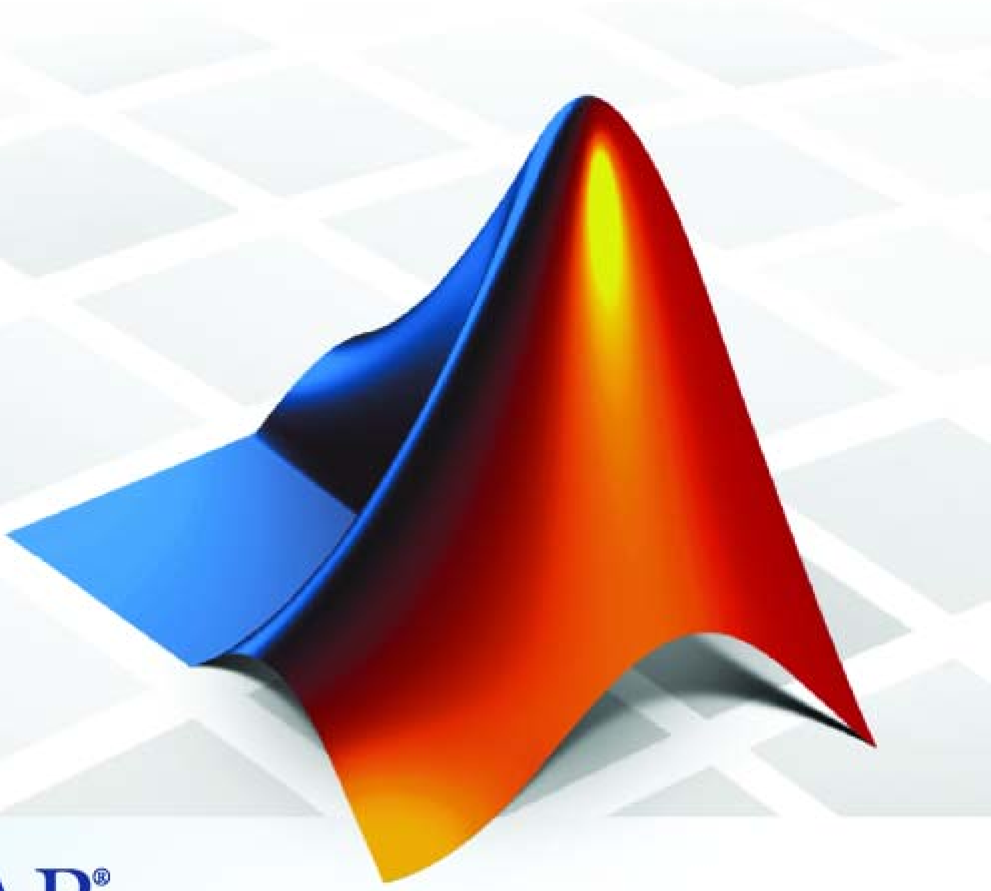
Statistics Toolbox 6
User's Guide



MATLAB®

The MathWorks

Accelerating the pace of engineering and science

How to Contact The MathWorks

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	Technical Support
`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

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

For contact information about worldwide offices, see the MathWorks Web site.

Statistics Toolbox User's Guide

Revision History

September 1993	First printing	Version 1.0
March 1996	Second printing	Version 2.0
January 1997	Third printing	Version 2.11
November 2000	Fourth printing	Revised for Version 3.0 (Release 12)
May 2001	Fifth printing	Minor revisions
July 2002	Sixth printing	Revised for Version 4.0 (Release 13)
February 2003	Online only	Revised for Version 4.1 (Release 13.0.1)
June 2004	Seventh printing	Revised for Version 5.0 (Release 14)
October 2004	Online only	Revised for Version 5.0.1 (Release 14SP1)
March 2005	Online only	Revised for Version 5.0.2 (Release 14SP2)
September 2005	Online only	Revised for Version 5.1 (Release 14SP3)
March 2006	Online only	Revised for Version 5.2 (Release 2006a)
September 2006	Online only	Revised for Version 5.3 (Release 2006b)
March 2007	Eighth printing	Revised for Version 6.0 (Release 2007a)

Contents

6 Hypothesis Tests

7 Linear Models

8 Nonlinear Models

9 Multivariate Statistics

Getting Started

What Is Statistics Toolbox?

Statistics Toolbox extends MATLAB® to support a wide range of common statistical tasks. The toolbox contains two categories of tools:

- Building-block statistical functions for use in MATLAB programming
- Graphical user interfaces (GUIs) for interactive use of the functions

Code for the building-block functions is open and extensible. You can use the MATLAB Editor to review, copy, and edit the M-file code for any function. You can extend the toolbox by copying code to new M-files or by writing M-files that call toolbox functions.

Toolbox GUIs allow you to perform statistical visualization and analysis without writing code. You interact with the GUIs through controls such as sliders, push buttons, and input fields, and the GUIs interact with the building-block functions in the background.

Primary Topic Areas

Descriptive Statistics

Statistics Toolbox includes functions for computing common measures of location, scale, and shape of a numerical data sample. The functions allow for convenient handling of multidimensional data and missing data values.

Statistical Visualization

Statistics Toolbox adds many specialized statistical plots to the plot types already found in MATLAB. Relevant functions accept grouping variables for the simultaneous visualization of different data groups. Interactive features allow you to explore data sets and experiment with different data models.

Probability Distributions

Statistics Toolbox supports computations involving over 30 different common probability distributions, plus custom distributions which you can define. For each distribution, a selection of relevant functions is available, including density functions, cumulative distribution functions, parameter estimation functions, and random number generators. The toolbox also supports nonparametric methods for density estimation.

Hypothesis Tests

Statistics Toolbox provides functions that implement many common hypothesis tests, including distribution tests, analysis of variance tests, and tests of randomness.

Linear Models

In the area of linear regression, Statistics Toolbox has functions to compute parameter estimates, predicted values, and confidence intervals for simple and multiple regression, stepwise regression, ridge regression, and regression using response surface models. In the area of analysis of variance (ANOVA), Statistics Toolbox has functions to perform one-way, two-way, and higher-way ANOVA, analysis of covariance (ANOCOVA), multivariate analysis of variance (MANOVA), and multiple comparisons of the estimates produced by ANOVA and ANOCOVA functions.

Nonlinear Models

For nonlinear regression models, Statistics Toolbox provides additional parameter estimation functions and tools for interactive prediction and visualization of multidimensional nonlinear fits. The toolbox also includes functions that create classification and regression trees to approximate regression relationships.

Multivariate Statistics

Statistics Toolbox supports methods for the visualization and analysis of multidimensional data, including principal components analysis, factor analysis, one-way multivariate analysis of variance, cluster analysis, and classical multidimensional scaling.

Statistical Process Control

In the area of process control and quality management, Statistics Toolbox provides functions for creating a variety of control charts, performing process capability studies, and evaluating Design for Six Sigma (DFSS) methodologies.

Design of Experiments

Statistics Toolbox provides tools for generating and augmenting full and fractional factorial designs, response surface designs, and D-optimal designs. The toolbox also provides functions for the optimal assignment of units with fixed covariates.

Hidden Markov Models

Statistics Toolbox provides functions for the analysis of hidden Markov models, including the generation of random data, maximum likelihood estimation of model parameters, calculation of most probable state sequences, and calculation of posterior state probabilities.

Data Sets

The following data sets are provided with Statistics Toolbox.

`acetylene.mat`	Chemical reaction data with correlated predictors
`carbig.mat`	Measurements of large model cars, 1970–1982
`carsmall.mat`	Measurements of small model cars, 1970–1982
`census.mat`	U.S. census data from 1790 to 1980
`cereal.mat`	Breakfast cereal ingredients
`cities.mat`	Quality of life ratings for U.S. metropolitan areas
`discrim.mat`	A version of `cities.mat` used for discriminant analysis
`examgrades.mat`	Exam grades on a scale of 0–100
`fisheriris.mat`	Fisher's iris data (1936)
`gas.mat`	Gasoline prices around the state of Massachusetts in 1993
`hald.mat`	Heat of cement vs. mix of ingredients
`hogg.mat`	Bacteria counts in different shipments of milk
`kmeansdata.mat`	Four-dimensional clustered data
`lawdata.mat`	Grade point average and LSAT test scores from 15 law schools
`mileage.mat`	Mileage data for three car models from two factories
`moore.mat`	Biochemical oxygen demand on five predictors
`morse.mat`	Recognition of Morse code distinctions by non-coders
`parts.mat`	Dimensional run out on 36 circular parts
`polydata.mat`	Data for `polytool` demo
`popcorn.mat`	Popcorn yield by popper type and brand
`reaction.mat`	Reaction kinetics data for Hougen-Watson model

`sat.dat`	Scholastic Aptitude Test averages by gender and test (table)
`sat2.dat`	Scholastic Aptitude Test averages by gender and test (csv)
`stockreturns.mat`	Simulated stock return data for factor analysis example

Organizing Data

Introduction

In MATLAB, data is placed into “data containers” in the form of workspace variables. All workspace variables organize data into some form of array. For statistical purposes, arrays are viewed as tables of values.

MATLAB variables use different structures to organize data:

- 2-D numerical arrays (matrices) organize observations and measured variables by rows and columns, respectively. (See “Data Structures” in the MATLAB documentation.)
- Multidimensional arrays organize multidimensional observations or experimental designs. (See “Multidimensional Arrays” in the MATLAB documentation.)
- Cell and structure arrays organize heterogeneous data of different types, sizes, units, etc. (See “Cell Arrays” and “Structures” in the MATLAB documentation.)

Data types determine the kind of data variables contain. (See “Data Types” in the MATLAB documentation.)

These basic MATLAB container variables are reviewed, in a statistical context, in the section on “MATLAB Arrays” on page 2-4.

These variables are not specifically designed for statistical data, however. Statistical data generally involves observations of multiple variables, with measurements of heterogeneous type and size. Data may be numerical, categorical, or in the form of descriptive metadata. Fitting statistical data into basic MATLAB variables, and accessing it efficiently, can be cumbersome.

Statistics Toolbox offers two additional types of container variables specifically designed for statistical data:

- “Categorical Arrays” on page 2-13 accommodate data in the form of discrete levels, together with its descriptive metadata.
- “Dataset Arrays” on page 2-28 encapsulate heterogeneous data and metadata, including categorical data, which is accessed and manipulated using familiar methods analogous to those for numerical matrices.

These statistical container variables are discussed in the section on "Statistical Arrays" on page 2-11.

MATLAB Arrays

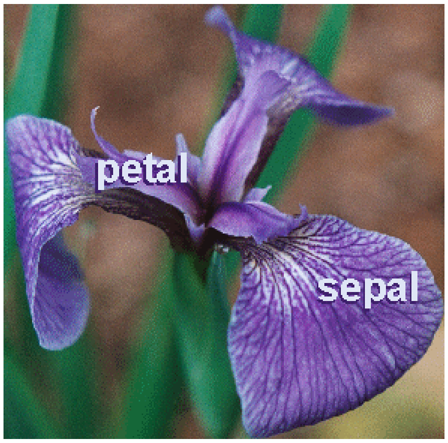
This section describes the array-based organization of data in MATLAB and the statistical functions that operate on data arrays.

Numerical Data

In MATLAB, two-dimensional numerical arrays (matrices) containing statistical data use rows to represent observations and columns to represent measured variables. For example,

```
load fisheriris % Fisher's iris data (1936)
```

loads the variables `meas` and `species` into the MATLAB workspace. The `meas` variable is a 150-by-4 numerical matrix, representing 150 observations of 4 different measured variables (by column: sepal length, sepal width, petal length, and petal width, respectively).



The observations in meas are of three different species of iris (setosa, versicolor, and virginica), which can be separated from one another using the 150-by-1 cell array of strings species:

```
setosa_indices = strcmp('setosa',species);
setosa = meas(setosa_indices,:);
```

The resulting setosa variable is 50-by-4, representing 50 observations of the 4 measured variables for iris setosa.

To access and display the first five observations in the setosa data, use row, column parenthesis indexing:

```
SetosaObs = setosa(1:5,:)
SetosaObs =
    5.1000    3.5000    1.4000    0.2000
    4.9000    3.0000    1.4000    0.2000
    4.7000    3.2000    1.3000    0.2000
    4.6000    3.1000    1.5000    0.2000
    5.0000    3.6000    1.4000    0.2000
```

The data are organized into a table with implicit column headers “Sepal Length,” “Sepal Width,” “Petal Length,” and “Petal Width.” Implicit row headers are “Observation 1,” “Observation 2,” “Observation 3,” etc.

Similarly, 50 observations for iris versicolor and iris virginica can be extracted from the meas container variable:

```
versicolor_indices = strcmp('versicolor',species);
versicolor = meas(versicolor_indices,:);

virginica_indices = strcmp('virginica',species);
virginica = meas(virginica_indices,:);
```

Because the data sets for the three species happen to be of the same size, they can be reorganized into a single 50-by-4-by-3 multidimensional array:

```
iris = cat(3,setosa,versicolor,virginica);
```

The iris array is a three-layer table with the same implicit row and column headers as the setosa, versicolor, and virginica arrays. The implicit layer names, along the third dimension, are “Setosa,” “Versicolor,” and “Virginica.” The utility of such a multidimensional organization depends on assigning meaningful properties of the data to each dimension.

To access and display data in a multidimensional array, use parenthesis indexing, as for 2-D arrays. The following gives the first five observations of sepal lengths in the setosa data:

```
SetosaSL = iris(1:5,1,1)
SetosaSL =
    5.1000
    4.9000
    4.7000
    4.6000
    5.0000
```

Multidimensional arrays provide a natural way to organize numerical data for which the observations, or experimental designs, have many dimensions. If, for example, data with the structure of `iris` are collected by multiple observers, in multiple locations, over multiple dates, the entirety of the data can be organized into a single higher dimensional array with dimensions for “Observer,” “Location,” and “Date.” Likewise, an experimental design calling for m observations of n p-dimensional variables could be stored in an m-by-n-by-p array.

Numerical arrays have limitations when organizing more general statistical data. One limitation is the implicit nature of the metadata. Another is the requirement that multidimensional data be of commensurate size across all dimensions. If variables have different lengths, or the number of variables differs by layer, then multidimensional arrays must be artificially padded with `NaN`s to indicate “missing values.” These limitations are addressed by dataset arrays (see “Dataset Arrays” on page 2-28), which are specifically designed for statistical data.

Heterogeneous Data

Two data types in MATLAB—cell arrays and structure arrays—provide container variables that allow you to combine metadata with variables of different types and sizes.

The data in the variables `setosa`, `versicolor`, and `virginica` created in "Numerical Data" on page 2-4 can be organized in a cell array, as follows:

```
iris1 = cell(51,5,3); % Container variable

obsnames = strcat({'Obs'},num2str((1:50)','%d'));
iris1(2:end,1,:) = repmat(obsnames,[1 1 3]);

varnames = {'SepalLength','SepalWidth',...
            'PetalLength','PetalWidth'};
iris1(1,2:end,:) = repmat(varnames,[1 1 3]);

iris1(2:end,2:end,1) = num2cell(setosa);
iris1(2:end,2:end,2) = num2cell(versicolor);
iris1(2:end,2:end,3) = num2cell(virginica);

iris1{1,1,1} = 'Setosa';
iris1{1,1,2} = 'Versicolor';
iris1{1,1,3} = 'Virginica';
```

To access and display the cells, use parenthesis indexing. The following displays the first five observations in the setosa sepal data:

```
SetosaSLSW = iris1(1:6,1:3,1)
SetosaSLSW =
    'Setosa'    'SepalLength'    'SepalWidth'
    'Obs1'      [     5.1000]    [    3.5000]
    'Obs2'      [     4.9000]    [         3]
    'Obs3'      [     4.7000]    [    3.2000]
    'Obs4'      [     4.6000]    [    3.1000]
    'Obs5'      [          5]    [    3.6000]
```

Here, the row and column headers have been explicitly labeled with metadata.

To extract the data subset, use row, column curly brace indexing:

```
subset = reshape([iris1{2:6,2:3,1}],5,2)
subset =
    5.1000    3.5000
    4.9000    3.0000
    4.7000    3.2000
    4.6000    3.1000
    5.0000    3.6000
```

While cell arrays are useful for organizing heterogeneous data, they may be cumbersome when it comes to manipulating and analyzing the data. Statistical functions in MATLAB and Statistics Toolbox do not accept data in the form of cell arrays. For processing, data must be extracted from the cell array to a numerical container variable, as in the preceding example. The indexing can become complicated for large, heterogeneous data sets. This shortcoming of cell arrays is addressed directly by dataset arrays (see "Dataset Arrays" on page 2-28), which are designed to store general statistical data and provide easy access.

The data in the preceding example can also be organized in a structure array, as follows:

```
iris2.data = cat(3,setosa,versicolor,virginica);
iris2.varnames = {'SepalLength','SepalWidth',...
                  'PetalLength','PetalWidth'};
iris2.obsnames = strcat({'Obs'},num2str((1:50)','%d'));
iris2.species = {'setosa','versicolor','virginica'};
```

The data subset is then returned using a combination of dot and parenthesis indexing:

```
subset = iris2.data(1:5,1:2,1)
subset =
    5.1000    3.5000
    4.9000    3.0000
    4.7000    3.2000
    4.6000    3.1000
    5.0000    3.6000
```

For statistical data, structure arrays have many of the same shortcomings of cell arrays. Once again, dataset arrays (see "Dataset Arrays" on page 2-28), designed specifically for general statistical data, address these shortcomings.

Statistical Functions

One of the advantages of working in MATLAB is that functions operate on entire arrays of data, not just on single scalar values. The functions are said to be *vectorized*. Vectorization allows for both efficient problem formulation, using array-based data, and efficient computation, using vectorized statistical functions.

When statistical functions in MATLAB and Statistics Toolbox operate on a vector of numerical data (either a row vector or a column vector), they return a single computed statistic:

```
% Fisher's setosa data:
load fisheriris
setosa_indices = strcmp('setosa',species);
setosa = meas(setosa_indices,:);

% Single variable from the data:
setosa_sepal_length = setosa(:,1);

% Standard deviation of the variable:
std(setosa_sepal_length)
ans =
    0.3525
```

When statistical functions operate on a matrix of numerical data, they treat the columns independently, as separate measured variables, and return a vector of statistics—one for each variable:

```
std(setosa)
ans =
    0.3525    0.3791    0.1737    0.1054
```

The four standard deviations are for measurements of sepal length, sepal width, petal length, and petal width, respectively.

Compare this to

```
std(setosa(:))
ans =
    1.8483
```

which gives the standard deviation across the entire array (all measurements).

Compare the preceding statistical calculations to the more generic mathematical operation

```
sin(setosa)
```

This operation returns a 50-by-4 array the same size as `setosa`. The `sin` function is vectorized in a different way than the `std` function, computing one scalar value for each element in the array.

Statistical functions in MATLAB and Statistics Toolbox, like `std`, must be distinguished from general mathematical functions like `sin`. Both are vectorized, and both are useful for working with array-based data, but only statistical functions summarize data across observations (rows) while preserving variables (columns). This property of statistical functions may be explicit, as with `std`, or implicit, as with `regress`. To see how a particular function handles array-based data, consult its reference page.

Statistical functions in MATLAB expect data input arguments to be in the form of numerical arrays. If data is stored in a cell or structure array, it must be extracted to a numerical array, via indexing, for processing. Functions in Statistics Toolbox are more flexible. Many Statistics Toolbox functions accept data input arguments in the form of both numerical arrays and dataset arrays (see "Dataset Arrays" on page 2-28), which are specifically designed for storing general statistical data.

Statistical Arrays

Introduction

As discussed in “MATLAB Arrays” on page 2-4, MATLAB offers array types for numerical, logical, and character data, as well as cell and structure arrays for heterogeneous collections of data.

Statistics Toolbox offers two additional types of arrays specifically designed for statistical data:

- “Categorical Arrays” on page 2-13
- “Dataset Arrays” on page 2-28

Categorical arrays store data with values in a discrete set of levels. Each level is meant to capture a single, defining characteristic of an observation. If no ordering is encoded in the levels, the data and the array are *nominal*. If an ordering is encoded, the data and the array are *ordinal*.

Categorical arrays also store labels for the levels. Nominal labels typically suggest the type of an observation, while ordinal labels suggest the position or rank.

Dataset arrays collect heterogeneous statistical data and metadata, including categorical data, into a single container variable. Like the numerical matrices discussed in “Numerical Data” on page 2-4, dataset arrays can be viewed as tables of values, with rows representing different observations and columns representing different measured variables. Like the cell and structure arrays discussed in “Heterogeneous Data” on page 2-7, dataset arrays can accommodate variables of different types, sizes, units, etc.

Dataset arrays combine the organizational advantages of these basic MATLAB data types while addressing their shortcomings with respect to storing complex statistical data.

Both categorical and dataset arrays have associated families of functions for assembling, accessing, manipulating, and processing the collected data. Basic array operations parallel those for numerical, cell, and structure arrays.

Categorical Arrays

Categorical Data

Categorical data take on values from only a finite, discrete set of categories or *levels*. Levels may be determined before the data are collected, based on the application, or they may be determined by the distinct values in the data when converting them to categorical form. Predetermined levels, such as a set of states or numerical intervals, are independent of the data they contain. Any number of values in the data may attain a given level, or no data at all. Categorical data show which measured values share common levels, and which do not.

Levels may have associated *labels*. Labels typically express a defining characteristic of an observation, captured by its level.

If no ordering is encoded in the levels, the data are *nominal*. Nominal labels typically indicate the type of an observation. Examples of nominal labels are {false, true}, {male, female}, and {Afghanistan, ... , Zimbabwe}. For nominal data, the numeric or lexicographic order of the labels is irrelevant—Afghanistan is not considered to be less than, equal to, or greater than Zimbabwe.

If an ordering is encoded in the levels—for example, if levels labeled “red”, “green”, and “blue” represent wavelengths—the data are *ordinal*. Labels for ordinal levels typically indicate the position or rank of an observation. Examples of ordinal labels are {0, 1}, {mm, cm, m, km}, and {poor, satisfactory, outstanding}. The ordering of the levels may or may not correspond to the numeric or lexicographic order of the labels.

Categorical Arrays

Categorical data can be represented in MATLAB using integer arrays, but this method has a number of drawbacks. First, it removes all of the useful metadata that might be captured in labels for the levels. Labels must be stored separately, in character arrays or cell arrays of strings. Secondly, this method suggests that values stored in the integer array have their usual numeric meaning, which, for categorical data, they may not. Finally, integer types have a fixed set of levels (for example, `-128:127` for all `int8` arrays), which cannot be changed.

Categorical arrays, available in Statistics Toolbox, are specifically designed for storing, manipulating, and processing categorical data and metadata. Unlike integer arrays, each categorical array has its own set of levels, which can be changed. Categorical arrays also accommodate labels for levels in a natural way. Like numerical arrays, categorical arrays take on different shapes and sizes, from scalars to *N*-D arrays.

Organizing data in a categorical array can be an end in itself. Often, however, categorical arrays are used for further statistical processing. They can be used to index into other variables, creating subsets of data based on the category of observation, or they can be used with statistical functions that accept categorical inputs. For examples, see “Grouped Data” on page 2-41.

Categorical arrays come in two types, depending on whether the collected data is understood to be nominal or ordinal. Nominal arrays are constructed with the `nominal` function; ordinal arrays are constructed with the `ordinal` function. For example,

```
load fisheriris
ndata = nominal(species,{'A','B','C'});
```

creates a nominal array with levels `A`, `B`, and `C` from the `species` data in `fisheriris.mat`, while

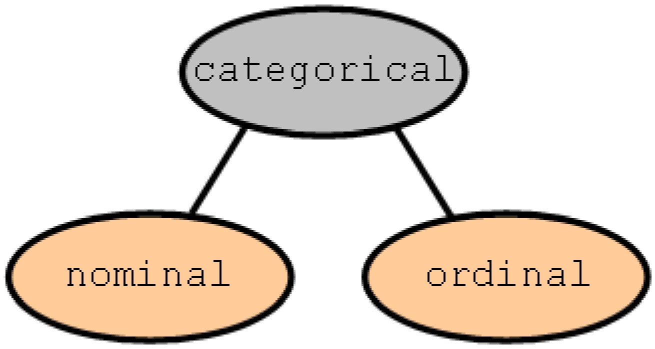
```
odata = ordinal(ndata,{},{'C','A','B'});
```

encodes an ordering of the levels with `C < A < B`. See “Using Categorical Arrays” on page 2-21, and the reference pages for `nominal` and `ordinal`, for further examples.

Functions associated with categorical arrays are used to display, summarize, convert, concatenate, and access the collected data. Examples include `disp`, `summary (categorical)`, `double`, `horzcat`, and `getlabels`, respectively. Many of these functions are invoked using operations analogous to those for numerical arrays, and do not need to be called directly. (For example, `horzcat` is invoked by `[]`.) Other functions are used to manipulate levels and labels and must be called directly (for example, `addlevels` and `setlabels`). There are functions that apply to both nominal and ordinal arrays (for example, `getlabels`), functions that apply only to one type (for example, `sortrows (ordinal)`), and functions that are applied differently to the two types (for example, `horzcat`). For a complete list of functions with descriptions of their use, see "Categorical Array Operations" on page 2-16.

Categorical arrays are implemented as *objects* in MATLAB, and the associated functions are their *methods*. It is not necessary to understand MATLAB objects and methods to make use of categorical arrays—in fact, categorical arrays are designed to behave as much as possible like other, familiar MATLAB arrays.

However, understanding the class structure of categorical arrays can be helpful when selecting an appropriate method. The data type `categorical` is an abstract class that defines properties and methods common to both the `nominal` and `ordinal` classes. Never call the constructor for the `categorical` class directly. Instead, use either the `nominal` or `ordinal` constructor. The `nominal` and `ordinal` classes are subclasses derived directly from the parent class `categorical`.



Categorical Array Operations

The tables in this section list available methods for categorical (ordinal and nominal) arrays. Many of the methods are invoked by familiar MATLAB operators and do not need to be called directly. For full descriptions of individual methods, type one of the following, depending on the class:

```
help ordinal/methodname
help nominal/methodname
```

Methods with supporting reference pages, including examples, are linked from the tables. "Using Categorical Arrays" on page 2-21 contains an extended example that makes use of many categorical methods.

The following table lists methods available for all categorical arrays (nominal and ordinal).

Categorical Method	Description
`addlevels`	Add levels to categorical array.
`cellstr`	Convert categorical array to cell array of strings.
`char`	Convert categorical array to character array.
`circshift`	Shift categorical array circularly.
`ctranspose`	Transpose categorical matrix. This method is invoked by the `'` operator.
`disp`	Display categorical array, without printing array name.
`display`	Display categorical array, printing array name. This method is invoked when the name of a categorical array is entered at the command prompt.
`double`	Convert categorical array to `double` array.
`droplevels`	Remove levels from categorical array.
`end`	Last index in indexing expression for categorical array.
`flipdim`	Flip categorical array along specified dimension.
`fliplr`	Flip categorical matrix in left/right direction.
`flipud`	Flip categorical matrix in up/down direction.

Categorical Method	Description
`getlabels`	Get level labels of categorical array.
`int8`	Convert categorical array to `int8` array.
`int16`	Convert categorical array to `int16` array.
`int32`	Convert categorical array to `int32` array.
`int64`	Convert categorical array to `int64` array.
`ipermute`	Inverse permute dimensions of categorical array.
`isempty`	True for empty categorical array.
`isequal`	True if categorical arrays are equal.
`islevel`	Test for categorical array levels.
`isscalar`	True if categorical array is scalar.
`isundefined`	True for elements of categorical array that are undefined.
`isvector`	True if categorical array is vector.
`length`	Length of categorical array.
`levelcounts`	Element counts by level for categorical array.
`ndims`	Number of dimensions of categorical array.
`numel`	Number of elements in categorical array.
`permute`	Permute dimensions of categorical array.
`reorderlevels`	Reorder levels in categorical array.
`repmat`	Replicate and tile a categorical array.
`reshape`	Change size of categorical array.
`rot90`	Rotate categorical matrix 90 degrees.
`setlabels`	Relabel levels for categorical array.
`shiftdim`	Shift dimensions of categorical array.
`single`	Convert categorical array to `single` array.
`size`	Size of categorical array.

Categorical Method	Description
`squeeze`	Squeeze singleton dimensions from categorical array.
`subsasgn`	Subscripted assignment for categorical array. This method is invoked by parenthesis indexing, as described in "Accessing Categorical Arrays" on page 2-23.
`subsref`	Subscripted reference for categorical array. This method is invoked by parenthesis indexing, as described in "Accessing Categorical Arrays" on page 2-23.
`summary (categorical)`	Summary of categorical array.
`times`	Product of categorical arrays. This method is invoked by the `.*` operator.
`transpose`	Transpose categorical matrix. This method is invoked by the `.'` operator.
`uint8`	Convert categorical array to `uint8` array.
`uint16`	Convert categorical array to `uint16` array.
`uint32`	Convert categorical array to `uint32` array.
`uint64`	Convert categorical array to `uint64` array.
`unique`	Unique values in categorical array.

The following table lists additional methods for nominal arrays.

Nominal Method	Description
`cat`	Concatenate nominal arrays. The `horzcat` and `vertcat` methods implement special cases.
`eq`	Equality for nominal array.
`horzcat`	Horizontal concatenation for nominal arrays. This method is invoked by square brackets, as described in "Combining Categorical Arrays" on page 2-24.

Nominal Method	**Description**
`intersect`	Set intersection for nominal arrays.
`ismember`	True for set member.
`mergelevels`	Merge levels of nominal array.
`ne`	Not equal for nominal arrays. This method is invoked by the ~= operator.
`nominal`	Create nominal array.
`setdiff`	Set difference for nominal arrays.
`setxor`	Set exclusive or for nominal arrays.
`union`	Set union for nominal arrays.
`vertcat`	Vertical concatenation for nominal arrays. This method is invoked by square brackets, as described in "Combining Categorical Arrays" on page 2-24.

The following table lists additional methods for ordinal arrays.

Ordinal Method	**Description**
`cat`	Concatenate ordinal arrays. The `horzcat` and `vertcat` methods implement special cases.
`eq`	Equality for ordinal arrays. This method is invoked by the == operator.
`ge`	Greater than or equal to for ordinal arrays. This method is invoked by the >= operator.
`gt`	Greater than for ordinal arrays. This method is invoked by the > operator.
`horzcat`	Horizontal concatenation for ordinal arrays. This method is invoked by square brackets, as described in "Combining Categorical Arrays" on page 2-24.
`intersect`	Set intersection for ordinal arrays.

Ordinal Method	Description
`ismember`	True for set member.
`issorted`	True for sorted ordinal array.
`le`	Less than or equal to for ordinal arrays. This method is invoked by the <= operator.
`lt`	Less than for ordinal arrays. This method is invoked by the < operator.
`max`	Largest element in ordinal array.
`mergelevels`	Merge levels of ordinal array.
`min`	Smallest element in ordinal array.
`ne`	Not equal for ordinal arrays. This method is invoked by the ~= operator.
`ordinal`	Create ordinal array.
`setdiff`	Set difference for ordinal arrays.
`setxor`	Set exclusive or for ordinal arrays.
`sort`	Sort ordinal array in ascending or descending order.
`sortrows (ordinal)`	Sort rows of ordinal matrix in ascending order.
`union`	Set union for ordinal arrays.
`vertcat`	Vertical concatenation for ordinal arrays. This method is invoked by square brackets, as described in "Combining Categorical Arrays" on page 2-24.

Using Categorical Arrays

This section provides an extended tutorial example demonstrating the use of categorical arrays and associated functions. The example introduces many available functions, but is not meant to be comprehensive. "Categorical Array Operations" on page 2-16 contains a complete list of available functions, with descriptions. For examples detailing the use of particular functions, alone or in combination with other functions, see the corresponding reference pages.

Constructing Categorical Arrays (p. 2-21)	The `nominal` and `ordinal` constructors
Accessing Categorical Arrays (p. 2-23)	Indexing methods
Combining Categorical Arrays (p. 2-24)	Concatenation methods
Computing with Categorical Arrays (p. 2-26)	Subsetting and grouping

Constructing Categorical Arrays. Load the 150-by-4 numerical array `meas` and the 150-by-1 cell array of strings `species`:

```
load fisheriris % Fisher's iris data (1936)
```

The data are 150 observations of four measured variables (by column number: sepal length, sepal width, petal length, and petal width, respectively) over three species of iris (setosa, versicolor, and virginica).

Use `nominal` to create a nominal array from `species`:

```
n1 = nominal(species);
```

Open `species` and `n1` side by side in the Array Editor (see "Viewing and Editing Workspace Variables with the Array Editor"). Note that the string information in `species` has been converted to categorical form, leaving only information on which data share the same values, indicated by the labels for the levels.

By default, levels are labeled with the distinct values in the data (in this case, the strings in `species`). Alternate labels are given with additional input arguments to the `nominal` constructor:

```
n2 = nominal(species,{'species1','species2','species3'});
```

Open `n2` in the Array Editor, and compare it with `species` and `n1`. The levels have been relabeled.

Suppose that the data are considered to be ordinal. A characteristic of the data that is not reflected in the labels is the diploid chromosome count, which orders the levels corresponding to the three species as follows:

species1 < species3 < species2

The `ordinal` constructor is used to cast `n2` as an ordinal array:

```
o1 = ordinal(n2,{},{'species1','species3','species2'});
```

The second input argument to `ordinal` is the same as for `nominal`—a list of labels for the levels in the data. If it is unspecified, as above, the labels are inherited from the data, in this case `n2`. The third input argument of `ordinal` indicates the ordering of the levels, in ascending order.

When displayed side by side in the Array Editor, `o1` does not appear any different than `n2`. This is because the data in `o1` have not been sorted. It is important to recognize the difference between the ordering of the levels in an ordinal array and sorting the actual data according to that ordering. The `sort` function sorts ordinal data in ascending order:

```
o2 = sort(o1);
```

When displayed in the Array Editor, `o2` shows the data sorted by diploid chromosome count.

To find which elements moved up in the sort, use the `<` operator for ordinal arrays:

```
moved_up = (o1 < o2);
```

The operation returns a logical array `moved_up`, indicating which elements have moved up (the data for `species3`).

Use the getlabels function to display the labels for the levels in ascending order:

```
labels2 = getlabels(o2)
labels2 =
    'species1'    'species3'    'species2'
```

The sort function reorders the display of the data, but not the order of the levels. To reorder the levels, use reorderlevels:

```
o3 = reorderlevels(o2,labels2([1 3 2]));
labels3 = getlabels(o3)
labels3 =
    'species1'    'species2'    'species3'
o4 = sort(o3);
```

These operations return the levels in the data to their original ordering, by species number, and then sort the data for display purposes.

Accessing Categorical Arrays. Categorical arrays are accessed using parenthesis indexing, with syntax that parallels similar operations for numerical arrays (see "Numerical Data" on page 2-4).

Parenthesis indexing on the right-hand side of an assignment is used to extract the lowest 50 elements from the ordinal array o4:

```
low50 = o4(1:50);
```

Suppose you want to categorize the data in o4 with only two levels: low (the data in low50) and high (the rest of the data). One way to do this is to use an assignment with parenthesis indexing on the left-hand side:

```
o5 = o4; % Copy o4
o5(1:50) = 'low';
Warning: Categorical level 'low' being added.
o5(51:end) = 'high';
Warning: Categorical level 'high' being added.
```

Note the warnings: the assignments move data to new levels. The old levels, though empty, remain:

```
getlabels(o5)
```

```
ans =
    'species1' 'species2' 'species3' 'low' 'high'
```

The old levels are removed using `droplevels`:

```
o5 = droplevels(o5,{'species1','species2','species3'});
```

Another approach to creating two categories in `o5` from the three categories in `o4` is to merge levels, using `mergelevels`:

```
o5 = mergelevels(o4,{'species1'},'low');
o5 = mergelevels(o5,{'species2','species3'},'high');

getlabels(o5)
ans =
    'low'    'high'
```

The merged levels are removed and replaced with the new levels.

Combining Categorical Arrays. Categorical arrays are concatenated using square brackets. Again, the syntax parallels similar operations for numerical arrays (see "Numerical Data" on page 2-4). There are, however, restrictions:

- Only categorical arrays of the same type can be combined. You cannot concatenate a nominal array with an ordinal array.
- Only ordinal arrays with the same levels, in the same order, can be combined.
- Nominal arrays with different levels can be combined to produce a nominal array whose levels are the union of the levels in the component arrays.

First use the `ordinal` constructor to create ordinal arrays from the variables for sepal length and sepal width in `meas`. Categorize the data as `short` or `long` depending on whether they are below or above the median of the variable, respectively:

```
sl = meas(:,1); % Sepal length data
sw = meas(:,2); % Sepal width data
SL1 = ordinal(sl,{'short','long'},[],...
              [min(sl),median(sl),max(sl)]);
SW1 = ordinal(sw,{'short','long'},[],...
              [min(sw),median(sw),max(sw)]);
```

Because SL1 and SW1 are ordinal arrays with the same levels, in the same order, they can be concatenated:

```
S1 = [SL1,SW1];
S1(1:10,:)
ans =
     short      long
     short      long
     short      long
     short      long
     short      long
     short      long
     short      long
     short      long
     short      short
     short      long
```

The result is an ordinal array S1 with two columns.

If, on the other hand, the measurements are cast as nominal, different levels can be used for the different variables, and the two nominal arrays can still be combined:

```
SL2 = nominal(sl,{'short','long'},[],...
              [min(sl),median(sl),max(sl)]);
SW2 = nominal(sw,{'skinny','wide'},[],...
              [min(sw),median(sw),max(sw)]);
S2 = [SL2,SW2];
getlabels(S2)
ans =
    'short' 'long' 'skinny' 'wide'
S2(1:10,:)
ans =
     short      wide
     short      wide
     short      wide
     short      wide
     short      wide
     short      wide
    short      wide
    short      wide
    short      skinny
    short      wide
```

Computing with Categorical Arrays. Categorical arrays are used to index into other variables, creating subsets of data based on the category of observation, and they are used with statistical functions that accept categorical inputs, such as those described in "Grouped Data" on page 2-41.

The `ismember` function is used to create logical variables based on the category of observation. For example, the following creates a logical index the same size as `species` that is `true` for observations of iris setosa and `false` elsewhere. Recall that `n1 = nominal(species)`:

```
SetosaObs = ismember(n1,'setosa');
```

Since the code above compares elements of `n1` to a single value, the same operation is carried out by the equality operator:

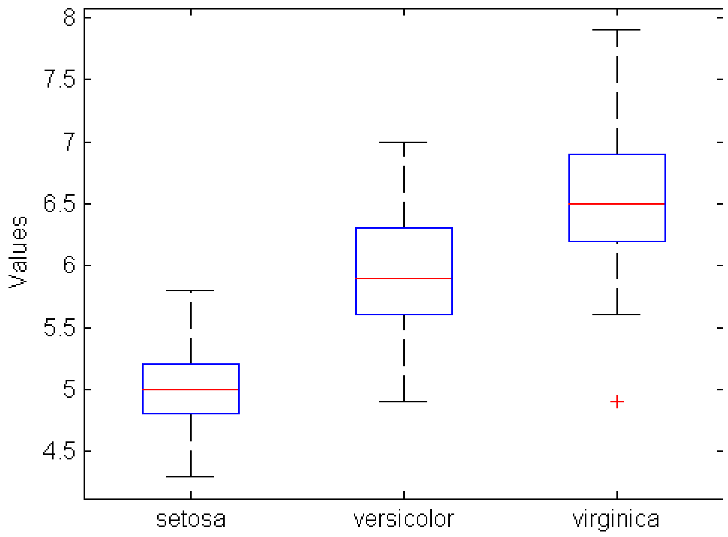
```
SetosaObs = (n1 == 'setosa');
```

The `SetosaObs` variable is used to index into `meas` to extract only the setosa data:

```
SetosaData = meas(SetosaObs,:);
```

Categorical arrays are also used as grouping variables. The following plot summarizes the sepal length data in `meas` by category:

```
boxplot(sl,n1)
```


Values
8
7.5
7
6.5
6
5.5
5
4.5
setosa
versicolor
virginica

Dataset Arrays

Statistical Data

MATLAB has “data containers” suitable for completely homogeneous data (numeric, character, and logical arrays) and for completely heterogeneous data (cell and structure arrays). Statistical data, however, are often a mixture of homogeneous variables of heterogeneous types and sizes. Dataset arrays are suitable containers for this kind of data.

Dataset arrays can be viewed as tables of values, with rows representing different observations or cases and columns representing different measured variables. In this sense, dataset arrays are analogous to the numerical arrays for statistical data discussed in “Numerical Data” on page 2-4. Basic methods for creating and manipulating dataset arrays parallel the syntax of corresponding methods for numerical arrays.

While each column of a dataset array must be a variable of a single type, each row may contain an observation consisting of measurements of different types. In this sense, dataset arrays lie somewhere between variables that enforce complete homogeneity on the data and those that enforce nothing. Because of the potentially heterogeneous nature of the data, dataset arrays have indexing methods with syntax that parallels corresponding methods for cell and structure arrays (see “Heterogeneous Data” on page 2-7).

Dataset Arrays

Dataset arrays are MATLAB variables created with the `dataset` function, and then manipulated with associated `dataset` functions.

For example, the following creates a dataset array from observations that are a combination of categorical and numerical measurements:

```
load fisheriris
NumObs = size(meas,1);
ObsNames = strcat({'Obs'},num2str((1:NumObs)','%d'));
iris = dataset({nominal(species),'species'},...
               {meas,'SL','SW','PL','PW'},...
               'obsnames',ObsNames);
iris(1:5,:)
ans =
            species    SL     SW     PL     PW
    Obs1    setosa     5.1    3.5    1.4    0.2
    Obs2    setosa     4.9      3    1.4    0.2
    Obs3    setosa     4.7    3.2    1.3    0.2
    Obs4    setosa     4.6    3.1    1.5    0.2
    Obs5    setosa       5    3.6    1.4    0.2
```

When creating a dataset array, variable names and observation names can be assigned together with the data. Other metadata associated with the array can be assigned with the `set` function and accessed with the `get` function. For example:

```
iris = set(iris,'Description','Fisher''s Iris Data');
get(iris)
   Description: 'Fisher's Iris Data'
   Units: {}
   DimNames: {'Observations' 'Variables'}
   UserData: []
   ObsNames: {150x1 cell}
   VarNames: {'species' 'SL' 'SW' 'PL' 'PW'}
```

See "Using Dataset Arrays" on page 2-33 and the reference page for `dataset` for further examples.

The following table lists the accessible properties of dataset arrays.

Dataset Property	Value
`Description`	A string describing the data set. The default is an empty string.
`Units`	A cell array of strings giving the units of the variables in the data set. The number of strings must equal the number of variables. Strings may be empty. The default is an empty cell array.
`DimNames`	A cell array of two strings giving the names of the rows and columns, respectively, of the data set. The default is `{'Observations' 'Variables'}`.
`UserData`	Any MATLAB variable containing additional information to be associated with the data set. The default is an empty array.
`ObsNames`	A cell array of nonempty, distinct strings giving the names of the observations in the data set. The number of strings must equal the number of observations. The default is an empty cell array.
`VarNames`	A cell array of nonempty, distinct strings giving the names of the variables in the data set. The number of strings must equal the number of variables. The default is the cell array of string names for the variables used to create the data set.

Functions associated with dataset arrays are used to display, summarize, convert, concatenate, and access the collected data. Examples include `disp`, `summary (dataset)`, `double`, `horzcat`, and `get`, respectively. Many of these functions are invoked using operations analogous to those for numerical arrays, and do not need to be called directly. (For example, `horzcat` is invoked by `[]`.) Other functions access the collected data and must be called directly (for example, `grpstats` and `replacedata`). For a complete list of functions with descriptions of their use, see "Dataset Array Operations" on page 2-31.

Dataset arrays are implemented as *objects* in MATLAB, and the associated functions are their *methods*. It isn't necessary to understand MATLAB objects and methods to make use of dataset arrays—in fact, dataset arrays are designed to behave as much as possible like other, familiar MATLAB arrays.

Dataset Array Operations

The table in this section lists available methods for dataset arrays. Many of the methods are invoked by familiar MATLAB operators and do not need to be called directly. For full descriptions of individual methods, type

```
help dataset/methodname
```

Methods with supporting reference pages, including examples, are linked from the table. "Using Dataset Arrays" on page 2-33 contains an extended example that makes use of many dataset methods.

Dataset Method	Description
`cat`	Concatenate dataset arrays. The `horzcat` and `vertcat` methods implement special cases.
`dataset`	Create dataset array.
`datasetfun`	Apply function to each variable of dataset array.
`disp`	Display dataset array, without printing data set name.
`display`	Display dataset array, printing data set name. This method is invoked when the name of a dataset array is entered at the command prompt.
`double`	Convert dataset variables to `double` array.
`end`	Last index in indexing expression for dataset array.
`get`	Get dataset array property.
`grpstats (dataset)`	A version of the `grpstats` function that accepts dataset arrays and categorical grouping variables as inputs.
`horzcat`	Horizontal concatenation for dataset arrays (add variables). This method is invoked by square brackets, as described in "Combining Dataset Arrays" on page 2-37.
`isempty`	True for empty dataset array.
`join`	Merge observations from two dataset arrays.
`length`	Length of dataset array.
`ndims`	Number of dimensions of dataset array.

Dataset Method	Description
`numel`	Number of elements in dataset array.
`replacedata`	Convert array to dataset variables.
`set`	Set dataset array property value.
`single`	Convert dataset variables to `single` array.
`size`	Size of dataset array.
`sortrows (dataset)`	Sort rows of dataset array.
`subsasgn`	Subscripted assignment for dataset array. This method is invoked by the parenthesis, dot, and curly brace indexing described in "Accessing Dataset Arrays" on page 2-35.
`subsref`	Subscripted reference for dataset array. This method is invoked by the parenthesis, dot, and curly brace indexing described in "Accessing Dataset Arrays" on page 2-35.
`summary (dataset)`	Print summary statistics for dataset array.
`unique`	Unique observations in dataset.
`vertcat`	Vertical concatenation for dataset arrays (add observations). This method is invoked by square brackets, as described in "Combining Dataset Arrays" on page 2-37.

Using Dataset Arrays

This section provides an extended tutorial example demonstrating the use of dataset arrays and associated functions. The example introduces many available functions, but is not meant to be comprehensive. "Dataset Array Operations" on page 2-31 contains a complete list of available functions, with descriptions. For examples detailing the use of particular functions, alone or in combination with other functions, see the corresponding reference pages.

Constructing Dataset Arrays. Load the 150-by-4 numerical array `meas` and the 150-by-1 cell array of strings `species`:

```
load fisheriris % Fisher's iris data (1936)
```

The data are 150 observations of four measured variables (by column number: sepal length, sepal width, petal length, and petal width, respectively) over three species of iris (setosa, versicolor, and virginica).

Create a dataset array `iris` from the data, assigning variable names `species`, `SL`, `SW`, `PL`, and `PW` and observation names `Obs1`, `Obs2`, `Obs3`, etc.:

```
NumObs = size(meas,1);
ObsNames = strcat({'Obs'},num2str((1:NumObs)','%d'));
iris = dataset({nominal(species),'species'},...
              {meas,'SL','SW','PL','PW'},...
              'obsnames',ObsNames);
```

```
iris(1:5,:)
ans =
            species    SL     SW     PL     PW
    Obs1    setosa     5.1    3.5    1.4    0.2
    Obs2    setosa     4.9      3    1.4    0.2
    Obs3    setosa     4.7    3.2    1.3    0.2
    Obs4    setosa     4.6    3.1    1.5    0.2
    Obs5    setosa       5    3.6    1.4    0.2
```

The cell array of strings `species` is first converted to a categorical array of type `nominal` before inclusion in the dataset array. For further information on categorical arrays, see "Categorical Arrays" on page 2-13.

Use the `set` function to set properties of the array:

```
desc = 'Fisher''s iris data (1936)';
units = [{''} repmat({'cm'},1,4)];
info = 'http://en.wikipedia.org/wiki/R.A._Fisher';

iris = set(iris,'Description',desc,...
                'Units',units,...
                'UserData',info);
```

Use the `get` function to view properties of the array:

```
get(iris)
    Description: 'Fisher's iris data (1936)'
          Units: {''  'cm'  'cm'  'cm'  'cm'}
       DimNames: {'Observations'  'Variables'}
       UserData: 'http://en.wikipedia.org/wiki/R.A._Fisher'
       ObsNames: {150x1 cell}
       VarNames: {'species'  'SL'  'SW'  'PL'  'PW'}

get(iris(1:5,:),'ObsNames')
ans =
    'Obs1'
    'Obs2'
    'Obs3'
    'Obs4'
    'Obs5'
```

For a table of accessible properties of dataset arrays, with descriptions, see “Dataset Arrays” on page 2-29.

Accessing Dataset Arrays. Dataset arrays support multiple types of indexing. Like the numerical matrices described in “Numerical Data” on page 2-4, parenthesis `()` indexing is used to access data subsets. Like the cell and structure arrays described in “Heterogeneous Data” on page 2-7, dot `.` indexing is used to access data variables and curly brace `{}` indexing is used to access data elements.

Use parenthesis indexing to assign a subset of the data in `iris` to a new dataset array `iris1`:

```
iris1 = iris(1:5,2:3)
iris1 =
             SL     SW
    Obs1    5.1    3.5
    Obs2    4.9      3
    Obs3    4.7    3.2
    Obs4    4.6    3.1
    Obs5      5    3.6
```

Similarly, use parenthesis indexing to assign new data to the first variable in `iris1`:

```
iris1(:,1) = dataset([5.2;4.9;4.6;4.6;5])
iris1 =
             SL     SW
    Obs1    5.2    3.5
    Obs2    4.9      3
    Obs3    4.6    3.2
    Obs4    4.6    3.1
    Obs5      5    3.6
```

Variable and observation names can also be used to access data:

```
SepalObs = iris1({'Obs1','Obs3','Obs5'},'SL')
SepalObs =
            SL
    Obs1    5.1
    Obs3    4.7
    Obs5      5
```

Dot indexing is used to access variables in a dataset array, and can be combined with other indexing methods. For example, the `zscore` function is applied to the data in `SepalObs` as follows:

```
ScaledSepalObs = zscore(iris1.SL([1 3 5]))
ScaledSepalObs =
    0.8006
   -1.1209
    0.3203
```

The following code extracts the sepal lengths in `iris1` corresponding to sepal widths greater than 3:

```
BigSWLengths = iris1.SL(iris1.SW > 3)
BigSWLengths =
    5.2000
    4.6000
    4.6000
    5.0000
```

Dot indexing also allows entire variables to be deleted from a dataset array:

```
iris1.SL = []
iris1 =
            SW
    Obs1    3.5
    Obs2      3
    Obs3    3.2
    Obs4    3.1
    Obs5    3.6
```

Dynamic variable naming works for dataset arrays just as it does for structure arrays. For example, the units of the `SW` variable are changed in `iris1` as follows:

```
varname = 'SW';
iris1.(varname) = iris1.(varname)*10
iris1 =
            SW
    Obs1    35
    Obs2    30
    Obs3    32
    Obs4    31
    Obs5    36
iris1 = set(iris1,'Units',{'mm'});
```

Curly brace indexing is used to access individual data elements. The following are equivalent:

```
iris1{1,1}
ans =
    35

iris1{'Obs1','SW'}
ans =
    35
```

Combining Dataset Arrays. Combine two dataset arrays into a single dataset array using square brackets:

```
SepalData = iris(:,{'SL','SW'});
PetalData = iris(:,{'PL','PW'});
newiris = [SepalData,PetalData];
size(newiris)
ans =
   150   4
```

For horizontal concatenation, as in the preceding example, the number of observations in the two dataset arrays must agree. Observations are matched up by name (if given), regardless of their order in the two data sets.

The following concatenates variables within a dataset array and then deletes the component variables:

```
newiris.SepalData = [newiris.SL,newiris.SW];
newiris.PetalData = [newiris.PL,newiris.PW];
```

```
newiris(:,{'SL','SW','PL','PW'}) = [];
size(newiris)
ans =
   150   2
size(newiris.SepalData)
ans =
   150   2
```

`newiris` is now a 150-by-2 dataset array containing two 150-by-2 numerical arrays as variables.

Vertical concatenation is also handled in a manner analogous to numerical arrays:

```
newobs = dataset({[5.3 4.2; 5.0 4.1],'PetalData'},...
                 {[5.5 2; 4.8 2.1],'SepalData'});
newiris = [newiris;newobs];
size(newiris)
ans =
   152     2
```

For vertical concatenation, as in the preceding example, the names of the variables in the two dataset arrays must agree. Variables are matched up by name, regardless of their order in the two data sets.

Expansion of variables is also accomplished using direct assignment to new rows:

```
newiris(153,:) = dataset({[5.1 4.0],'PetalData'},...
                         {[5.1 4.2],'SepalData'});
```

A different type of concatenation is performed by the `join` function, which takes the data in one dataset array and assigns it to the rows of another dataset array, based on matching values in a common key variable. For example, the following creates a dataset array with diploid chromosome counts for each species of iris:

```
snames = nominal({'setosa';'versicolor';'virginica'});
CC = dataset({snames,'species'},{[38;108;70],'cc'})
CC = 
    species       cc 
    setosa         38
    versicolor    108
    virginica      70
```

This data is broadcast to the rows of `iris` using `join`:

```
iris2 = join(iris,CC);
iris2([1 2 51 52 101 102],:)
ans = 
           species       SL     SW     PL     PW     cc 
 Obs1      setosa        5.1    3.5    1.4    0.2     38
 Obs2      setosa        4.9      3    1.4    0.2     38
 Obs51     versicolor      7    3.2    4.7    1.4    108
 Obs52     versicolor    6.4    3.2    4.5    1.5    108
 Obs101    virginica     6.3    3.3      6    2.5     70
 Obs102    virginica     5.8    2.7    5.1    1.9     70
```

Computing with Dataset Arrays. The `summary (dataset)` function provides summary statistics for the component variables of a dataset array:

```
summary(newiris)
Fisher's iris data (1936)
SepalData: [153x2 double]
     min        4.3000          2
     1st Q      5.1000     2.8000
     median     5.8000          3
     3rd Q      6.4000     3.3250
     max        7.9000     4.4000
PetalData: [153x2 double]
     min             1     0.1000
     1st Q      1.6000     0.3000
     median     4.4000     1.3000
     3rd Q      5.1000     1.8000
     max        6.9000     4.2000
```

To apply other statistical functions, use dot indexing to access relevant variables:

```
SepalMeans = mean(newiris.SepalData)
SepalMeans =
    5.8294    3.0503
```

The same result is obtained with the `datasetfun` function, which applies functions to dataset array variables:

```
means = datasetfun(@mean,newiris,'UniformOutput',false)
means =
    [1x2 double]    [1x2 double]
SepalMeans = means{1}
SepalMeans =
    5.8294    3.0503
```

An alternative approach is to cast data in a dataset array as `double` and apply statistical functions directly. Compare the following two methods for computing the covariance of the length and width of the `SepalData` in `newiris`:

```
covs = datasetfun(@cov,newiris,'UniformOutput',false)
covs =
    [2x2 double]    [2x2 double]
SepalCovs = covs{1}
SepalCovs =
    0.6835   -0.0373
   -0.0373    0.2054

SepalCovs = cov(double(newiris(:,1)))
SepalCovs =
    0.6835   -0.0373
   -0.0373    0.2054
```

Grouped Data

Grouping Variables

Grouping variables are utility variables used to indicate which elements in a data set are to be considered together when computing statistics and creating visualizations. They may be numeric vectors, string arrays, cell arrays of strings, or categorical arrays.

Grouping variables have the same length as the variables (columns) in a data set. Observations (rows) *i* and *j* are considered to be in the same group if the values of the corresponding grouping variable are identical at those indices.

For example, the following loads the 150-by-4 numerical array `meas` and the 150-by-1 cell array of strings `species` into the workspace:

```
load fisheriris % Fisher's iris data (1936)
```

The data are 150 observations of four measured variables (by column number: sepal length, sepal width, petal length, and petal width, respectively) over three species of iris (setosa, versicolor, and virginica). To group the observations by species, the following are all acceptable (and equivalent) grouping variables:

```
group1 = species; % Cell array of strings
group2 = grp2idx(species) % Numeric vector
group3 = char(species); % Character array
group4 = nominal(species); % Categorical array
```

These grouping variables can be supplied as input arguments to any of the functions described in "Functions for Grouped Data" on page 2-42. Examples are given in "Using Grouping Variables" on page 2-43.

Functions for Grouped Data

The following table lists functions in Statistics Toolbox that accept a grouping variable `group` as an input argument. The grouping variable may be in the form of a vector, string array, cell array of strings, or categorical array, as described in "Grouping Variables" on page 2-41.

For a full description of the syntax of any particular function, and examples of its use, consult its reference page, linked from the table. "Using Grouping Variables" on page 2-43 also includes examples.

Function	Basic Syntax for Grouped Data
`andrewsplot`	`andrewsplot(X, ... ,'Group',group)`
`anova1`	`p = anova1(X,group)`
`anovan`	`p = anovan(x,group)`
`aoctool`	`aoctool(x,y,group)`
`boxplot`	`boxplot(x,group)`
`classify`	`class = classify(sample,training,group)`
`controlchart`	`controlchart(x,group)`
`crosstab`	`crosstab(group1,group2)`
`dummyvar`	`D = dummyvar(group)`
`gagerr`	`gagerr(x,group)`
`gplotmatrix`	`gplotmatrix(x,y,group)`
`grp2idx`	`[G,GN] = grp2idx(group)`
`grpstats`	`means = grpstats(X,group)`
`gscatter`	`gscatter(x,y,group)`
`interactionplot`	`interactionplot(X,group)`
`kruskalwallis`	`p = kruskalwallis(X,group)`
`maineffectsplot`	`maineffectsplot(X,group)`
`manova1`	`d = manova1(X,group)`
`multivarichart`	`multivarichart(x,group)`
`parallelcoords`	`parallelcoords(X, ... ,'Group',group)`

Function	Basic Syntax for Grouped Data
`silhouette`	`silhouette(X,group)`
`tabulate`	`tabulate(group)`
`treefit`	`T = treefit(X,y,'cost',S)` or `T = treefit(X,y,'priorprob',S)`, where `S.group = group`
`vartestn`	`vartestn(X,group)`

Using Grouping Variables

This section provides an example demonstrating the use of grouping variables and associated functions. Grouping variables are introduced in "Grouping Variables" on page 2-41. A list of functions accepting grouping variables as input arguments is given in "Functions for Grouped Data" on page 2-42.

Load the 150-by-4 numerical array `meas` and the 150-by-1 cell array of strings `species`:

```
load fisheriris % Fisher's iris data (1936)
```

The data are 150 observations of four measured variables (by column number: sepal length, sepal width, petal length, and petal width, respectively) over three species of iris (setosa, versicolor, and virginica).

Create a categorical array (see "Categorical Arrays" on page 2-13) from `species` to use as a grouping variable:

```
group = nominal(species);
```

While `species`, as a cell array of strings, is itself a grouping variable, the categorical array has the advantage that it can be easily manipulated with categorical methods. (See "Categorical Array Operations" on page 2-16.)

Compute some basic statistics for the data (median and interquartile range), by group, using the `grpstats` function:

```
[order,number,group_median,group_iqr] = ...
grpstats(meas,group,{'gname','numel',@median,@iqr})
order =
        'setosa'
        'versicolor'
        'virginica'
    number =
        50    50    50    50
        50    50    50    50
        50    50    50    50
    group_median =
        5.0000    3.4000    1.5000    0.2000
        5.9000    2.8000    4.3500    1.3000
        6.5000    3.0000    5.5500    2.0000
    group_iqr =
        0.4000    0.5000    0.2000    0.1000
        0.7000    0.5000    0.6000    0.3000
        0.7000    0.4000    0.8000    0.5000
```

The statistics appear in 3-by-4 arrays, corresponding to the 3 groups (categories) and 4 variables in the data. The order of the groups (not encoded in the nominal array `group`) is indicated by the group names in `order`.

To improve the labeling of the data, create a dataset array (see "Dataset Arrays" on page 2-28) from `meas`:

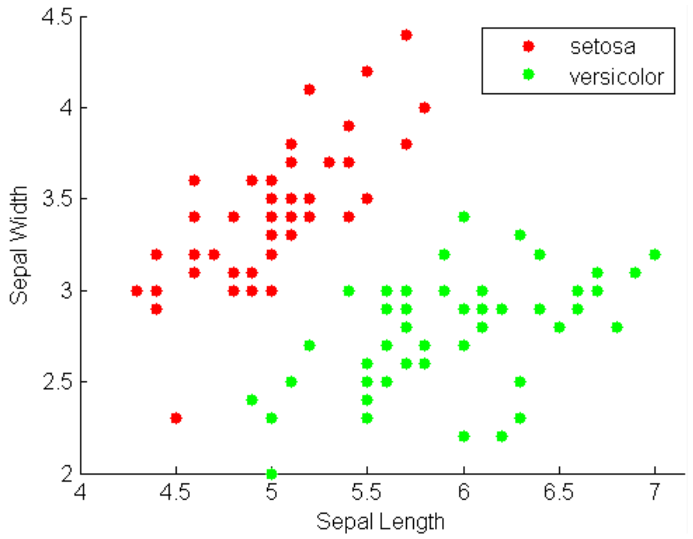
```
NumObs = size(meas,1);
ObsNames = strcat({'Obs'},num2str((1:NumObs)','%d'));
iris = dataset({group,'species'},...
               {meas,'SL','SW','PL','PW'},...
               'obsnames',ObsNames);
```

When you call `grpstats` with a dataset array as an argument, you invoke the `grpstats` method of the `dataset` class, `grpstats (dataset)`, rather than the regular `grpstats` function. The method has a slightly different syntax than the regular `grpstats` function, but it returns the same results, with better labeling:

```
stats = grpstats(iris,'species',{@median,@iqr})
stats =
                  species       GroupCount
    setosa        setosa        50
    versicolor    versicolor    50
    virginica     virginica     50

                  median_SL    iqr_SL
    setosa          5          0.4
    versicolor    5.9          0.7
    virginica     6.5          0.7

                  median_SW    iqr_SW
    setosa        3.4          0.5
    versicolor    2.8          0.5
    virginica       3          0.4

                  median_PL    iqr_PL
    setosa         1.5         0.2
    versicolor    4.35         0.6
    virginica     5.55         0.8

                  median_PW    iqr_PW
    setosa        0.2          0.1
    versicolor    1.3          0.3
    virginica       2          0.5
```

Grouping variables are also used to create visualizations based on categories of observations. The following scatter plot, created with the `gscatter` function, shows the correlation between sepal length and sepal width in two species of iris. The `ismember` function is used to subset the two species from `group`:

```
subset = ismember(group,{'setosa','versicolor'});
scattergroup = group(subset);
gscatter(iris.SL(subset),...
         iris.SW(subset),...
         scattergroup)
xlabel('Sepal Length')
ylabel('Sepal Width')
```


setosa
versicolor
Sepal Width
Sepal Length
4.5
4
3.5
3
2.5
2
4
4.5
5
5.5
6
6.5
7

Descriptive Statistics

Introduction

A first step in data analysis is often to produce useful summaries of data characteristics. This section introduces basic methods for producing summary statistics and plots.

Statistics Toolbox functions are introduced in the following sections:

- “Measures of Central Tendency” on page 3-3
- “Measures of Dispersion” on page 3-5
- “Data with Missing Values” on page 3-7
- “Graphical Descriptions” on page 3-9
- “The Bootstrap” on page 3-18

Note For information on creating summaries of data by group, see “Grouped Data” on page 2-41.

Measures of Central Tendency

The purpose of measures of central tendency is to locate the data values on the number line. Another term for these statistics is *measures of location*.

The following table lists the functions that calculate the measures of central tendency.

Function Name	Description
`geomean`	Geometric mean
`harmmean`	Harmonic mean
`mean`	Arithmetic average (in MATLAB)
`median`	50th percentile (in MATLAB)
`mode`	Most frequent value (in MATLAB)
`trimmean`	Trimmed mean

The average is a simple and popular estimate of location. If the data sample comes from a normal distribution, then the sample mean is also optimal (MVUE of μ).

Unfortunately, outliers, data entry errors, or glitches exist in almost all real data. The sample mean is sensitive to these problems. One bad data value can move the average away from the center of the rest of the data by an arbitrarily large distance.

The median and trimmed mean are two measures that are resistant (robust) to outliers. The median is the 50th percentile of the sample, which will only change slightly if you add a large perturbation to any value. The idea behind the trimmed mean is to ignore a small percentage of the highest and lowest values of a sample when determining the center of the sample.

The geometric mean and harmonic mean, like the average, are not robust to outliers. They are useful when the sample is distributed lognormal or heavily skewed.

The following example shows the behavior of the measures of location for a sample with one outlier.

```
x = [ones(1,6) 100]

x =
     1     1     1     1     1     1   100

locate = [geomean(x) harmmean(x) mean(x) median(x)...
          trimmean(x,25)]

locate =
    1.9307    1.1647   15.1429    1.0000    1.0000
```

You can see that the mean is far from any data value because of the influence of the outlier. The median and trimmed mean ignore the outlying value and describe the location of the rest of the data values.

Measures of Dispersion

The purpose of measures of dispersion is to find out how spread out the data values are on the number line. Another term for these statistics is measures of spread.

The table gives the function names and descriptions.

Function Name	Description
`iqr`	Interquartile range
`mad`	Mean absolute deviation
`range`	Range
`std`	Standard deviation (in MATLAB)
`var`	Variance (in MATLAB)

The range (the difference between the maximum and minimum values) is the simplest measure of spread. But if there is an outlier in the data, it will be the minimum or maximum value. Thus, the range is not robust to outliers.

The standard deviation and the variance are popular measures of spread that are optimal for normally distributed samples. The sample variance is the MVUE of the normal parameter σ^2. The standard deviation is the square root of the variance and has the desirable property of being in the same units as the data. That is, if the data is in meters, the standard deviation is in meters as well. The variance is in meters^2, which is more difficult to interpret.

Neither the standard deviation nor the variance is robust to outliers. A data value that is separate from the body of the data can increase the value of the statistics by an arbitrarily large amount.

The mean absolute deviation (MAD) is also sensitive to outliers. But the MAD does not move quite as much as the standard deviation or variance in response to bad data.

The interquartile range (IQR) is the difference between the 75th and 25th percentile of the data. Since only the middle 50% of the data affects this measure, it is robust to outliers.

The following example shows the behavior of the measures of dispersion for a sample with one outlier.

```
x = [ones(1,6) 100]

x =
     1     1     1     1     1     1   100

stats = [iqr(x) mad(x) range(x) std(x)]

stats =
         0   24.2449   99.0000   37.4185
```

Data with Missing Values

Most real-world data sets have one or more missing elements. It is convenient to code missing entries in a matrix as `NaN` (Not a Number).

Here is a simple example.

```
m = magic(3);
m([1 5]) = [NaN NaN]

m =
   NaN    1    6
     3  NaN    7
     4    9    2
```

Any arithmetic operation that involves the missing values in this matrix yields `NaN`, as below.

```
sum(m)

ans =
   NaN NaN 15
```

Removing cells with `NaN` would destroy the matrix structure. Removing whole rows that contain `NaN` would discard real data. Instead, Statistics Toolbox has a variety of functions listed in the following table that are similar to other MATLAB functions, but that treat `NaN` values as missing and therefore ignore them in the calculations.

```
nansum(m)

ans =
     7    10    15
```

Function	Description
`nanmax`	Maximum ignoring NaNs
`nanmean`	Mean ignoring NaNs
`nanmedian`	Median ignoring NaNs
`nanmin`	Minimum ignoring NaNs
`nanstd`	Standard deviation ignoring NaNs
`nansum`	Sum ignoring NaNs

In addition, other Statistics Toolbox functions operate only on the numeric values, ignoring `NaN`s. These include `iqr`, `kurtosis`, `mad`, `prctile`, `range`, `skewness`, and `trimmean`.

Graphical Descriptions

Trying to describe a data sample with two numbers, a measure of location and a measure of spread, is frugal but may be misleading. Here are some other approaches:

- "Quantiles and Percentiles" on page 3-9
- "Probability Density Estimation" on page 3-10
- "Empirical Cumulative Distribution Function" on page 3-15

Quantiles and Percentiles

Quantiles and percentiles provide information about the shape of data as well as its location and spread.

The *quantile* of order p ($0 \le p \le 1$) is the smallest x value where the cumulative distribution function equals or exceeds p. The function `quantile` computes quantiles as follows:

1. n sorted data points are the $0.5/n$, $1.5/n$, ..., $(n–0.5)/n$ quantiles.
2. Linear interpolation is used to compute intermediate quantiles.
3. The data `min` or `max` are assigned to quantiles outside the range.
4. Missing values are treated as `NaN`, and removed from the data.

Percentiles, computed by the `prctile` function, are quantiles for a certain percentage of the data, specified for $0 \le p \le 100$.

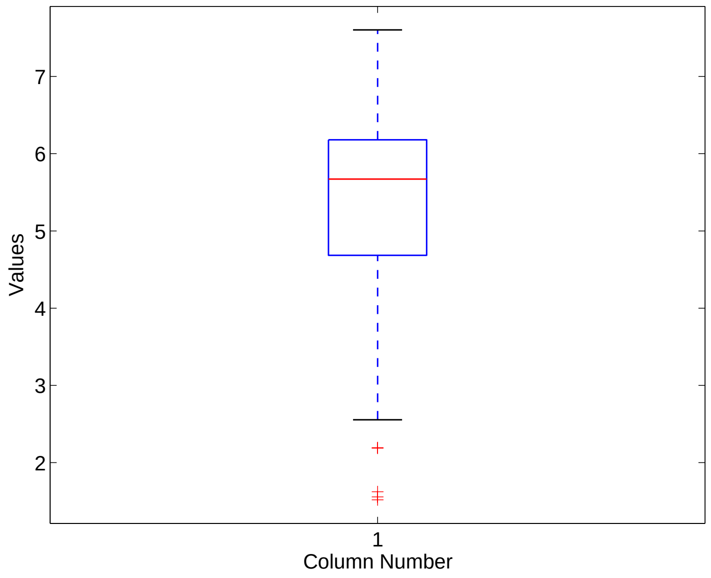
The following example shows the result of looking at every quartile (quantiles with orders that are multiples of 0.25) of a sample containing a mixture of two distributions.

```
x = [normrnd(4,1,1,100) normrnd(6,0.5,1,200)];
p = 100*(0:0.25:1);
y = prctile(x,p);
z = [p;y]
z =
          0   25.0000   50.0000   75.0000  100.0000
```

```
    1.5172    4.6842    5.6706    6.1804    7.6035
```

A box plot helps to visualize the statistics:

```
boxplot(x)
```

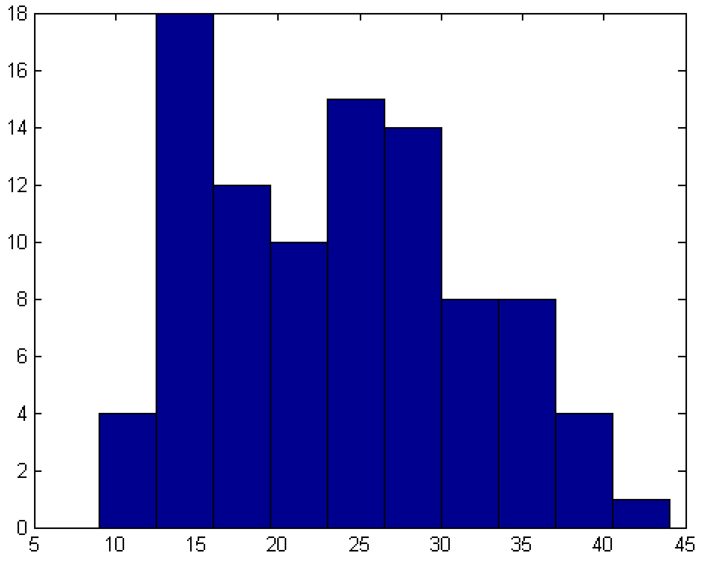


The long lower tail and plus signs show the lack of symmetry in the sample values. For more information on box plots, see “Box Plots” on page 4-6.

Probability Density Estimation

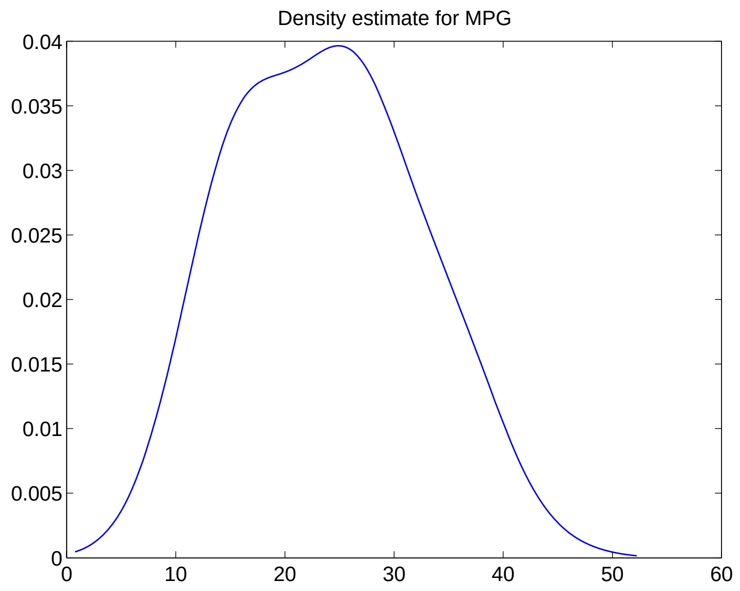
The distribution of data can be described graphically with a histogram:

```
cars = load('carsmall','MPG','Origin');
MPG = cars.MPG;
hist(MPG)
```



You can also describe a data distribution by estimating its density. The `ksdensity` function does this using a kernel smoothing method. A nonparametric density estimate of the data above, using the default kernel and bandwidth, is given by:

```
[f,x] = ksdensity(MPG);
plot(x,f);
title('Density estimate for MPG')
```



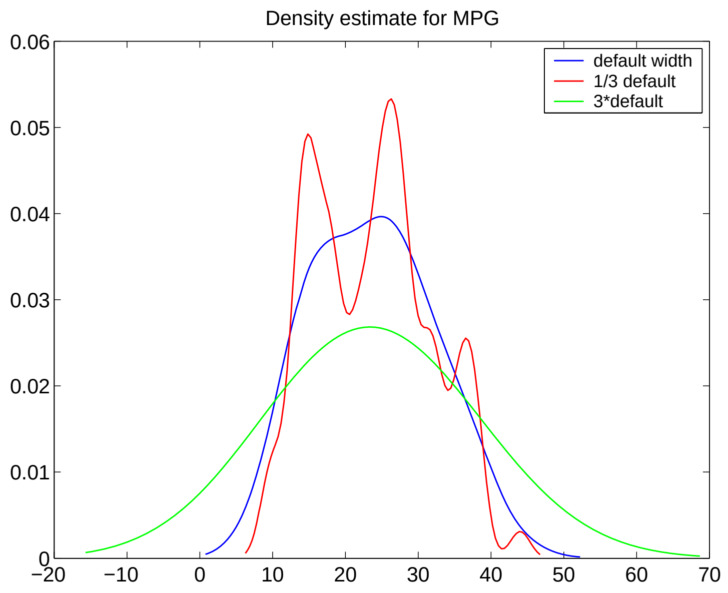
Kernel Bandwidth

The choice of kernel bandwidth controls the smoothness of the probability density curve. The following graph shows the density estimate for the same mileage data using different bandwidths. The default bandwidth is in blue and looks like the preceding graph. Estimates for smaller and larger bandwidths are in red and green.

The first call to `ksdensity` returns the default bandwidth, `u`, of the kernel smoothing function. Subsequent calls modify this bandwidth.

```
[f,x,u] = ksdensity(MPG);
plot(x,f)
title('Density estimate for MPG')
hold on
[f,x] = ksdensity(MPG,'width',u/3);
plot(x,f,'r');
[f,x] = ksdensity(MPG,'width',u*3);
plot(x,f,'g');
```

```
legend('default width','1/3 default','3*default')
hold off
```



The default bandwidth seems to be doing a good job—reasonably smooth, but not so smooth as to obscure features of the data. This bandwidth is the one that is theoretically optimal for estimating densities for the normal distribution.

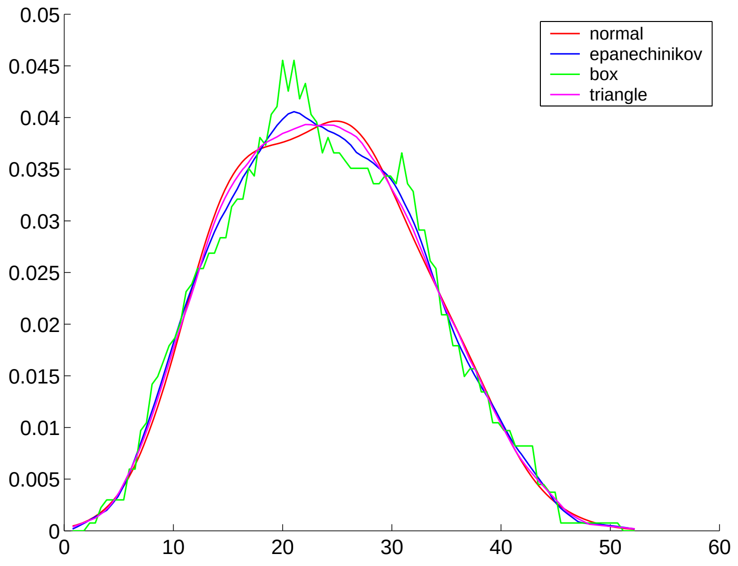
The green curve shows a density with the kernel bandwidth set too high. This curve smooths out the data so much that the end result looks just like the kernel function. The red curve has a smaller bandwidth and is rougher looking than the blue curve. It may be too rough, but it does provide an indication that there might be two major peaks rather than the single peak of the blue curve. A reasonable choice of width might lead to a curve that is intermediate between the red and blue curves.

Kernel Smoothing Function

You can also specify a kernel function by supplying either the function name or a function handle. The four preselected functions, `'normal'`, `'epanechnikov'`, `'box'`, and `'triangle'`, are all scaled to have standard deviation equal to 1, so they perform a comparable degree of smoothing.

Using default bandwidths, you can now plot the same mileage data, using each of the available kernel functions.

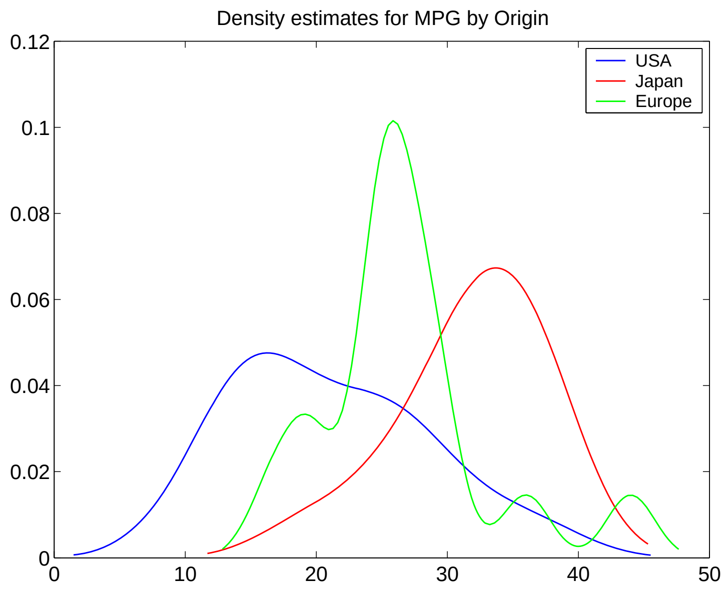
```
hname = {'normal' 'epanechnikov' 'box' 'triangle'};
hold on;
colors = {'r' 'b' 'g' 'm'};
for j=1:4
    [f,x] = ksdensity(MPG,'kernel',hname{j});
    plot(x,f,colors{j});
end
legend(hname{:});
hold off
```



The density estimates are roughly comparable, but the box kernel produces a density that is rougher than the others.

Usefulness of Smooth Density Estimates

In addition to the aesthetic appeal of the smooth density estimate, there are other appeals as well. While it is difficult to overlay two histograms to compare them, you can easily overlay smooth density estimates. For example, the following graph shows the MPG distributions for cars from different countries of origin.



For piecewise probability density estimation, using kernel smoothing in the center of the distribution and Pareto distributions in the tails, see "Fitting Piecewise Distributions" on page 5-106 and `paretotails`.

Empirical Cumulative Distribution Function

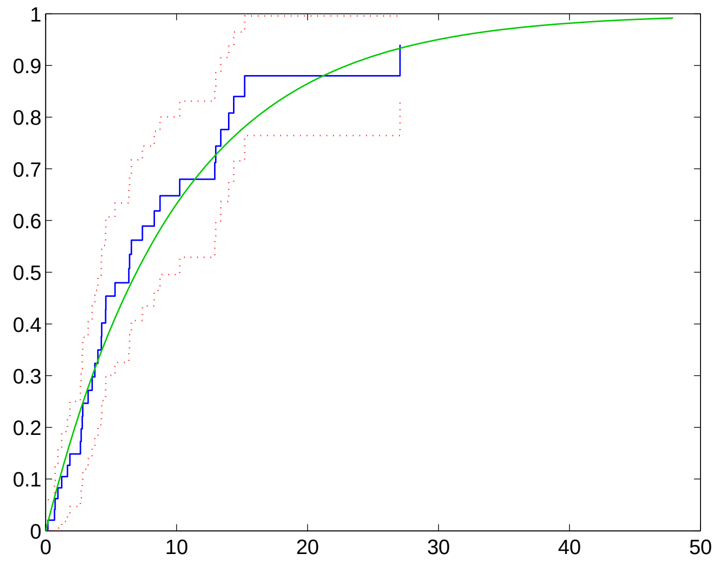
The `ksdensity` function described in the last section produces an empirical version of a probability density function (pdf). That is, instead of selecting a density with a particular parametric form and estimating the parameters, it produces a nonparametric density estimate that tries to adapt itself to the data.

Similarly, it is possible to produce an empirical version of the cumulative distribution function (cdf). The `ecdf` function computes this empirical cdf. It returns the values of a function F such that $F(x)$ represents the proportion of observations in a sample less than or equal to x.

The idea behind the empirical cdf is simple. It is a function that assigns probability $1/n$ to each of n observations in a sample. Its graph has a stair-step appearance. If a sample comes from a distribution in a parametric family (such as a normal distribution), its empirical cdf is likely to resemble the parametric distribution. If not, its empirical distribution still gives an estimate of the cdf for the distribution that generated the data.

The following example generates 20 observations from a normal distribution with mean 10 and standard deviation 2. You can use `ecdf` to calculate the empirical cdf and `stairs` to plot it. Then you overlay the normal distribution curve on the empirical function.

```
x = normrnd(10,2,20,1);[f,xf] = ecdf(x);
stairs(xf,f)
xx=linspace(5,15,100);
yy = normcdf(xx,10,2);
hold on; plot(xx,yy,'r:'); hold off
legend('Empirical cdf','Normal cdf',2)
```



The empirical cdf is especially useful in survival analysis applications. In such applications the data may be censored, that is, not observed exactly. Some individuals may fail during a study, and you can observe their failure time exactly. Other individuals may drop out of the study, or may not fail until after the study is complete. The `ecdf` function has arguments for dealing with censored data. In addition, you can use the `coxphfit` function with individuals that have predictors that are not the same.

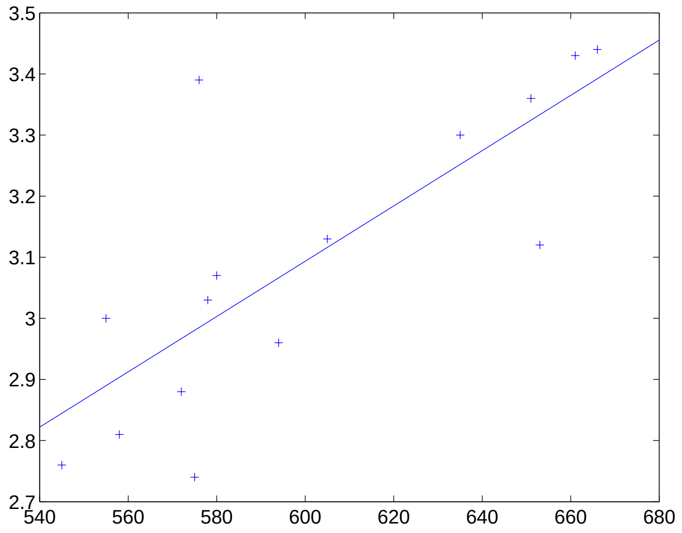
For piecewise probability density estimation, using the empirical cdf in the center of the distribution and Pareto distributions in the tails, see "Fitting Piecewise Distributions" on page 5-106 and `paretotails`.

The Bootstrap

The bootstrap is a procedure that involves choosing random samples with replacement from a data set and analyzing each sample the same way. Sampling with replacement means that every sample is returned to the data set after sampling. So a particular data point from the original data set could appear multiple times in a given bootstrap sample. The number of elements in each bootstrap sample equals the number of elements in the original data set. The range of sample estimates you obtain enables you to establish the uncertainty of the quantity you are estimating.

Here is an example taken from Efron and Tibshirani [18] comparing Law School Admission Test (LSAT) scores and subsequent law school grade point average (GPA) for a sample of 15 law schools.

```
load lawdata
plot(lsat,gpa,'+')
lsline
```



The least squares fit line indicates that higher LSAT scores go with higher law school GPAs. But how certain is this conclusion? The plot provides some intuition, but nothing quantitative.

You can calculate the correlation coefficient of the variables using the `corr` function.

```
rhohat = corr(lsat,gpa)

rhohat =

    0.7764
```

Now you have a number, 0.7764, describing the positive connection between LSAT and GPA, but though 0.7764 may seem large, you still do not know if it is statistically significant.

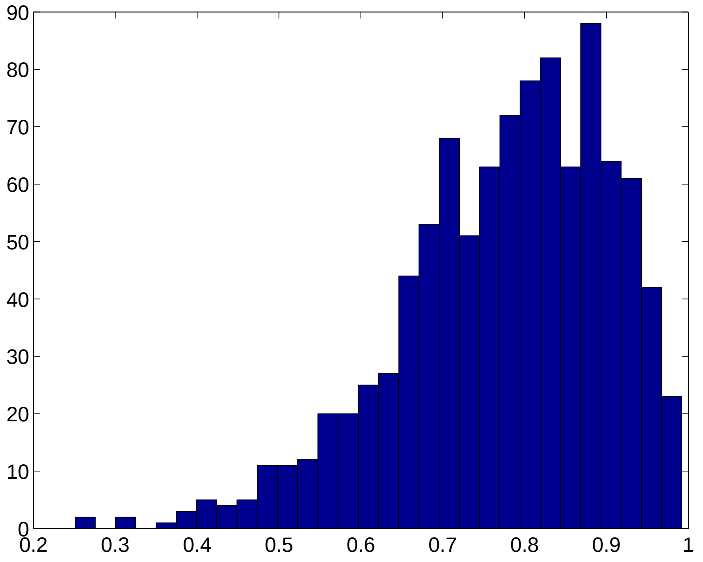
Using the `bootstrp` function you can resample the `lsat` and `gpa` vectors as many times as you like and consider the variation in the resulting correlation coefficients.

Here is an example.

```
rhos1000 = bootstrp(1000,'corr',lsat,gpa);
```

This command resamples the `lsat` and `gpa` vectors 1000 times and computes the `corr` function on each sample. Here is a histogram of the result.

```
hist(rhos1000(:,2),30)
```



Nearly all the estimates lie on the interval [0.4 1.0].

Bootstrap Confidence Intervals

It is often desirable to construct a confidence interval for a parameter estimate in statistical inferences. Using the `bootci` function, you can use bootstrapping to obtain a confidence interval. The confidence interval for the `lsat` and `gpa` data is computed as:

```
ci = bootci(5000,@corr,lsat,gpa)

ci =

    0.3265
    0.9389
```

Therefore, a 95% confidence interval for the correlation coefficient between LSAT and GPA is [0.33 0.94]. This is strong quantitative evidence that LSAT and subsequent GPA are positively correlated. Moreover, this evidence does not require any strong assumptions about the probability distribution of the correlation coefficient.

Although the `bootci` function computes the Bias Corrected and accelerated (BCa) interval as the default type, it is also able to compute various other types of bootstrap confidence intervals, such as the studentized bootstrap confidence interval.

Statistical Visualization

Introduction

Statistics Toolbox adds many data visualization functions to the extensive graphics capabilities already in MATLAB. Of general use are:

- *Scatter plots* are a basic visualization tool for multivariate data. They are used to identify relationships among variables. Grouped versions of these plots use different plotting symbols to indicate group membership. The `gname` function can label points on these plots with a text label or an observation number.
- *Box plots* display a five number summary of a set of data: the median, the two ends of the interquartile range (the box), and two extreme values (the whiskers) above and below the box. Because they show less detail than histograms, box plots are most useful for side-by-side comparisons of two distributions.
- *Distribution plots* help you identify an appropriate distribution family for your data. They include normal and Weibull probability plots, quantile-quantile plots, and empirical cumulative distribution plots.

These plots are described further in the sections:

- "Scatter Plots" on page 4-3
- "Box Plots" on page 4-6
- "Distribution Plots" on page 4-8

Advanced visualization functions for specialized statistical analyses are listed under Statistical Visualization.

Note For information on creating visualizations of data by group, see "Grouped Data" on page 2-41.

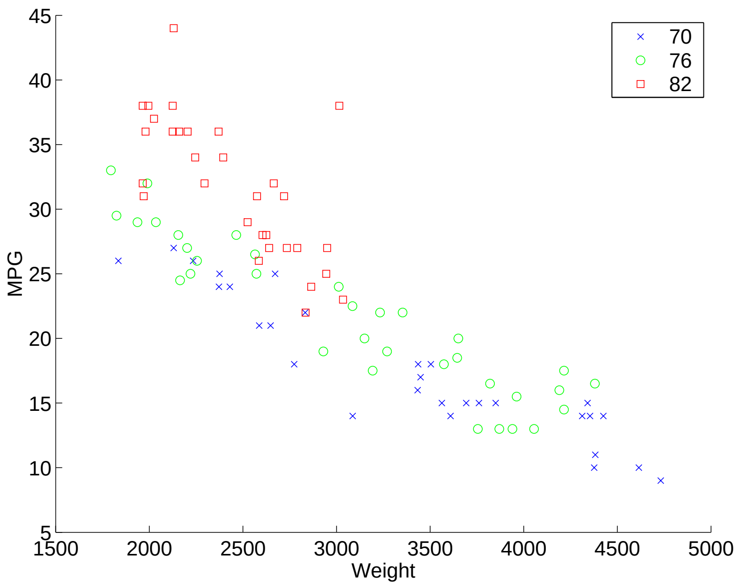
Scatter Plots

A scatter plot is a simple plot of one variable against another. The MATLAB `plot` and `scatter` functions can produce scatter plots. The MATLAB `plotmatrix` function can produce a matrix of such plots showing the relationship between several pairs of variables.

Statistics Toolbox adds the functions `gscatter` and `gplotmatrix` to produce grouped versions of these plots. These are useful for determining whether the values of two variables or the relationship between those variables is the same in each group.

Suppose you want to examine the weight and mileage of cars from three different model years.

```
load carsmall
gscatter(Weight,MPG,Model_Year,'','xos')
```

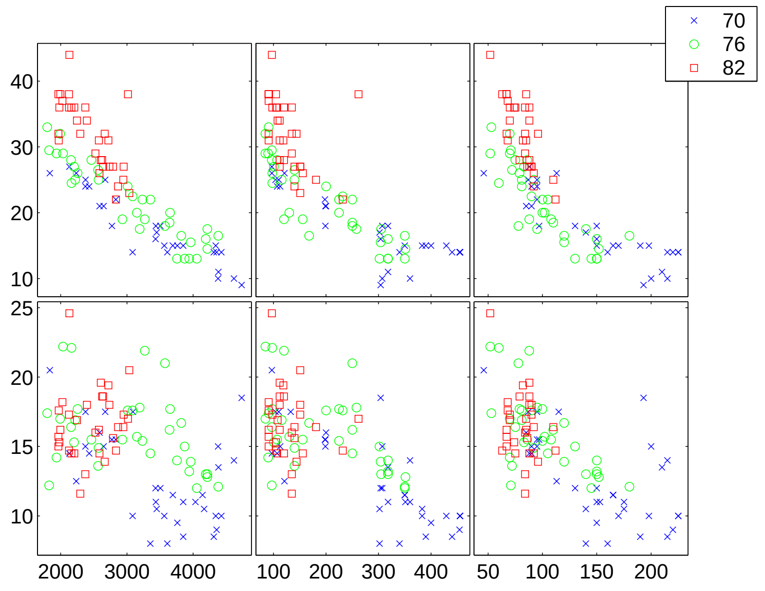


This shows that not only is there a strong relationship between the weight of a car and its mileage, but also that newer cars tend to be lighter and have better gas mileage than older cars.

The default arguments for `gscatter` produce a scatter plot with the different groups shown with the same symbol but different colors. The last two arguments above request that all groups be shown in default colors and with different symbols.

The `carsmall` data set contains other variables that describe different aspects of cars. You can examine several of them in a single display by creating a grouped plot matrix.

```
xvars = [Weight Displacement Horsepower];
yvars = [MPG Acceleration];
gplotmatrix(xvars,yvars,Model_Year,'','xos')
```



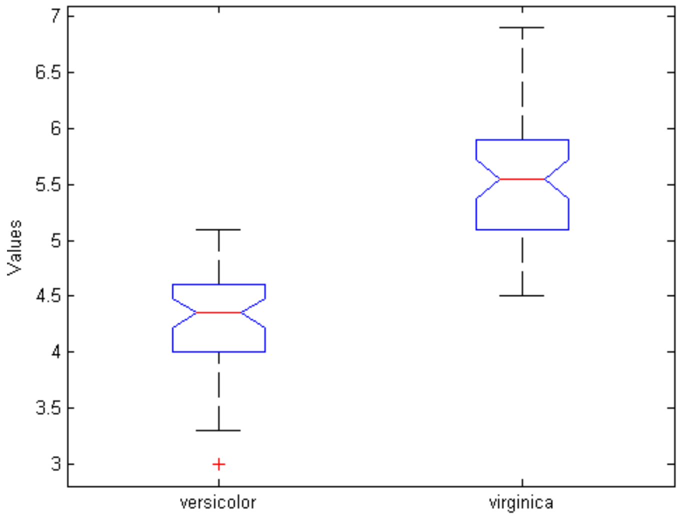
The upper right subplot displays MPG against Horsepower, and shows that over the years the horsepower of the cars has decreased but the gas mileage has improved.

The gplotmatrix function can also graph all pairs from a single list of variables, along with histograms for each variable. See "Multivariate Analysis of Variance" on page 9-23.

Box Plots

The graph below, created with the `boxplot` command, compares petal lengths in samples from two species of iris.

```
load fisheriris
s1 = meas(51:100,3);
s2 = meas(101:150,3);
boxplot([s1 s2],'notch','on',...
        'labels',{'versicolor','virginica'})
```



This plot has the following features:

- The tops and bottoms of each “box” are the 25th and 75th percentiles of the samples, respectively. The distances between the tops and bottoms are the interquartile ranges.
- The line in the middle of each box is the sample median. If the median is not centered in the box, it shows sample skewness.
- The ”whiskers” are lines extending above and below each box. Whiskers are drawn from the ends of the interquartile ranges to the furthest observations within the whisker length (the *adjacent values*).

- Observations beyond the whisker length are marked as outliers. By default, an outlier is a value that is more than 1.5 times the interquartile range away from the top or bottom of the box, but this value can be adjusted with additional input arguments. Outliers are displayed with a red + sign.
- Notches display the variability of the median between samples. The width of a notch is computed so that box plots whose notches do not overlap (as above) have different medians at the 5% significance level. The significance level is based on a normal distribution assumption, but comparisons of medians are reasonably robust for other distributions. Comparing box-plot medians is like a visual hypothesis test, analogous to the *t* test used for means.

Distribution Plots

There are several types of plots for assessing the distribution of statistics and data samples, as described in the following sections:

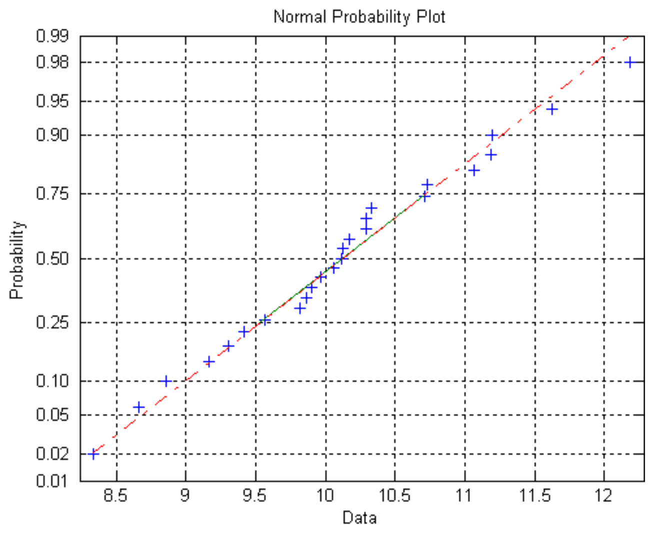
- “Normal Probability Plots” on page 4-8
- “Quantile-Quantile Plots” on page 4-10
- “Empirical Cumulative Distribution Function Plots” on page 4-12
- “Other Probability Plots” on page 4-13

Normal Probability Plots

Normal probability plots are used to assess whether data comes from a normal distribution. Many statistical procedures make the assumption that an underlying distribution is normal, so normal probability plots can provide some assurance that the assumption is justified, or else provide a warning of problems with the assumption. An analysis of normality typically combines normal probability plots with hypothesis tests for normality, as described in Chapter 6, “Hypothesis Tests”.

The following example shows a normal probability plot created with the `normplot` function.

```
x = normrnd(10,1,25,1);
normplot(x)
```

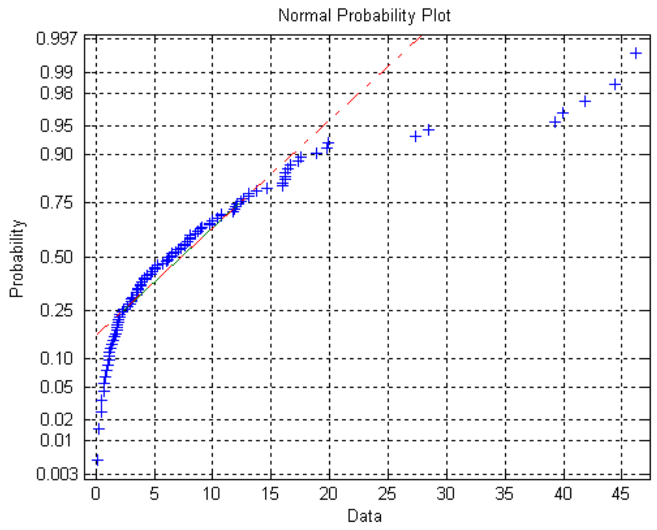


The plus signs plot the empirical probability versus the data value for each point in the data. A solid line connects the 25th and 75th percentiles in the data, and a dashed line extends it to the ends of the data. The y-axis values are probabilities from zero to one, but the scale is not linear. The distance between tick marks on the y-axis matches the distance between the quantiles of a normal distribution. The quantiles are close together near the median (probability = 0.5) and stretch out symmetrically as you move away from the median.

In a normal probability plot, if all the data points fall near the line, an assumption of normality is reasonable. Otherwise, the points will curve away from the line, and an assumption of normality is not justified.

For example:

```
x = exprnd(10,100,1);
normplot(x)
```



The plot is strong evidence that the underlying distribution is not normal.

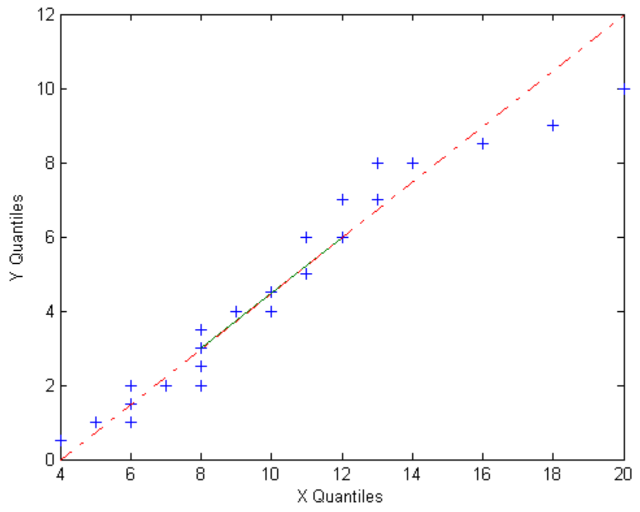
Quantile-Quantile Plots

Quantile-quantile plots are used to determine whether two samples come from the same distribution family. They are scatter plots of quantiles computed from each sample, with a line drawn between the first and third quartiles. If the data falls near the line, it is reasonable to assume that the two samples come from the same distribution. The method is robust with respect to changes in the location and scale of either distribution.

To create a quantile-quantile plot, use the `qqplot` function.

The following example shows a quantile-quantile plot of two samples from Poisson distributions.

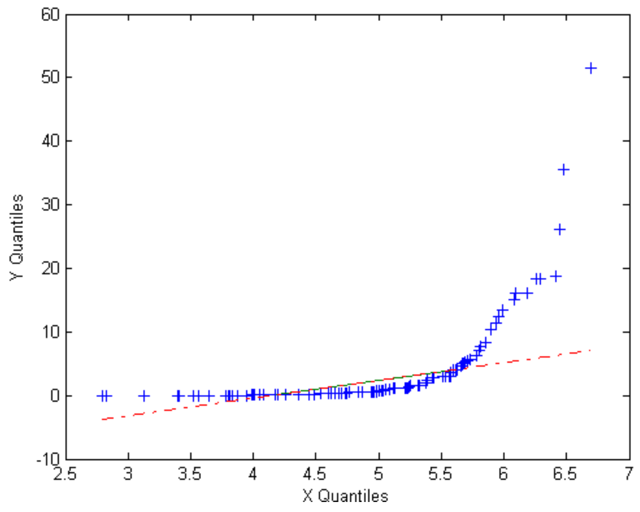
```
x = poissrnd(10,50,1);
y = poissrnd(5,100,1);
qqplot(x,y);
```



Even though the parameters and sample sizes are different, the approximate linear relationship suggests that the two samples may come from the same distribution family. As with normal probability plots, hypothesis tests, as described in Chapter 6, “Hypothesis Tests”, can provide additional justification for such an assumption. For statistical procedures that depend on the two samples coming from the same distribution, however, a linear quantile-quantile plot is often sufficient.

The following example shows what happens when the underlying distributions are not the same.

```
x = normrnd(5,1,100,1);
y = wblrnd(2,0.5,100,1);
qqplot(x,y);
```



These samples clearly are not from the same distribution family.

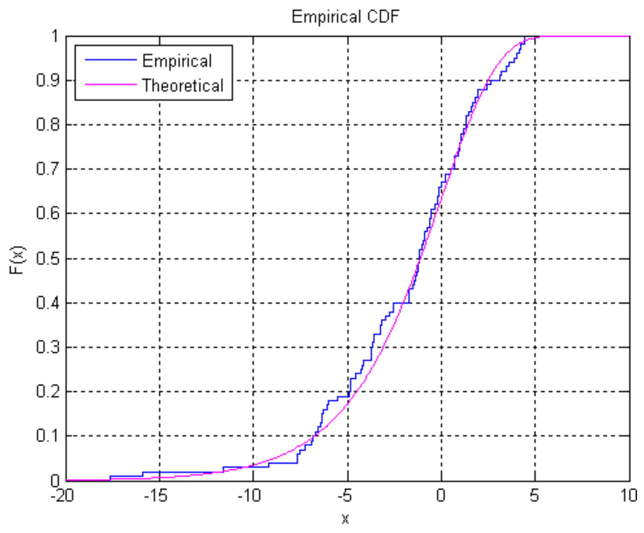
Empirical Cumulative Distribution Function Plots

An empirical cumulative distribution function (cdf) plot shows the proportion of data less than each x value, as a function of x. The scale on the y-axis is linear; in particular, it is not scaled to any particular distribution. Empirical cdf plots are used to compare data cdfs to cdfs for particular distributions.

To create an empirical cdf plot, use the `cdfplot` function (or `ecdf` and `stairs`).

The following example compares the empirical cdf for a sample from an extreme value distribution with a plot of the cdf for the sampling distribution. In practice, the sampling distribution would be unknown, and would be chosen to match the empirical cdf.

```
y = evrnd(0,3,100,1);
cdfplot(y)
hold on
x = -20:0.1:10;
f = evcdf(x,0,3);
plot(x,f,'m')
legend('Empirical','Theoretical','Location','NW')
```



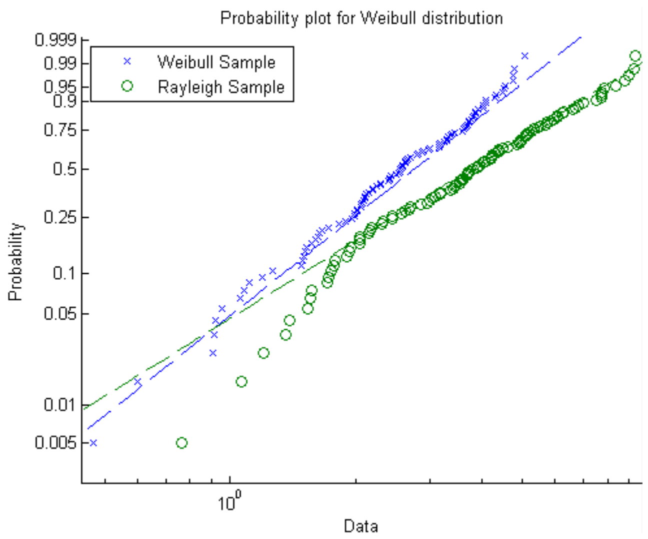
Other Probability Plots

A probability plot, like the normal probability plot, is just an empirical cdf plot scaled to a particular distribution. The *y*-axis values are probabilities from zero to one, but the scale is not linear. The distance between tick marks is the distance between quantiles of the distribution. In the plot, a line is drawn between the first and third quartiles in the data. If the data falls near the line, it is reasonable to choose the distribution as a model for the data.

To create probability plots for different distributions, use the `probplot` function.

For example, the following plot assesses two samples, one from a Weibull distribution and one from a Rayleigh distribution, to see if they may have come from a Weibull population.

```
x1 = wblrnd(3,3,100,1);
x2 = raylrnd(3,100,1);
probplot('weibull',[x1 x2])
legend('Weibull Sample','Rayleigh Sample','Location','NW')
```



The plot gives justification for modeling the first sample with a Weibull distribution; much less so for the second sample.

A distribution analysis typically combines probability plots with hypothesis tests for a particular distribution, as described in Chapter 6, “Hypothesis Tests”.

Probability Distributions

Introduction

A typical data sample is distributed over a range of values, with some values occurring more frequently than others. Some of the variability may be the result of measurement error or sampling effects. For large random samples, however, the distribution of the data typically reflects the variability of the source population and can be used to model the data-producing process.

Statistics computed from data samples also vary from sample to sample. Modeling distributions of statistics is important for drawing inferences from statistical summaries of data.

Probability distributions are theoretical distributions, based on assumptions about a source population. They assign probability to the event that a random variable, such as a data value or a statistic, takes on a specific, discrete value, or falls within a specified range of continuous values.

Choosing a model often means choosing a parametric family of probability distributions and then adjusting the parameters to fit the data. The choice of an appropriate distribution family may be based on *a priori* knowledge, such as matching the mechanism of a data-producing process to the theoretical assumptions underlying a particular family, or *a posteriori* knowledge, such as information provided by probability plots and distribution tests. Parameters can then be found that achieve the *maximum likelihood* of producing the data.

When the source population is unavailable for analysis or repeated sampling (as, for example, with historical data), *nonparametric models*, such as those produced by `ksdensity`, may be appropriate. These models make no assumptions about the mechanism producing the data or the form of the underlying distribution, so no parameter estimates are made. Nonparametric models are appropriate when data or statistics do not follow any standard probability distribution (as, for example, with multimodal data).

Once a model is chosen, *random number generators* produce random values with the specified probability distribution. Random number generators are used in *Monte Carlo simulations* of the original data-producing process.

Supported Distributions

Probability distributions supported by Statistics Toolbox are cross-referenced with their supporting functions and GUIs in the following tables:

- "Continuous Distributions (Data)" on page 5-4
- "Continuous Distributions (Statistics)" on page 5-6
- "Discrete Distributions" on page 5-7
- "Multivariate Distributions" on page 5-8

The tables use the following abbreviations for distribution functions:

- **pdf** — Probability density functions
- **cdf** — Cumulative distribution functions
- **inv** — Inverse cumulative distribution functions
- **stat** — Distribution statistics functions
- **fit** — Distribution fitting functions
- **like** — Negative log-likelihood functions
- **rnd** — Random number generators

Note Supported distributions are described more fully in the "Distribution Reference" on page 5-9.

Continuous Distributions (Data)

Name	pdf	cdf	inv	stat	fit	like	rnd
Beta	betapdf, pdf	betacdf, cdf	betainv, icdf	betastat	betafit, mle	betalike	betarnd, random, randtool
Birnbaum-Saunders					dfittool		
Exponential	exppdf, pdf	expcdf, cdf	expinv, icdf	expstat	expfit, mle, dfittool	explike	exprnd, random, randtool
Extreme value	evpdf, pdf	evcdf, cdf	evinv, icdf	evstat	evfit, mle, dfittool	evlike	evrnd, random, randtool
Gamma	gampdf, pdf	gamcdf, cdf	gaminv, icdf	gamstat	gamfit, mle, dfittool	gamlike	gamrnd, randg, random, randtool
Generalized extreme value	gevpdf, pdf	gevcdf, cdf	gevinv, icdf	gevstat	gevfit, dfittool	gevlike	gevrnd, random, randtool
Generalized Pareto	gppdf, pdf	gpcdf, cdf	gpinv, icdf	gpstat	gpfit, dfittool	gplike	gprnd, random, randtool
Inverse Gaussian					dfittool		
Johnson system							johnsrnd
Logistic					dfittool		
Loglogistic					dfittool		
Lognormal	lognpdf, pdf	logncdf, cdf	logninv, icdf	lognstat	lognfit, mle, dfittool	lognlike	lognrnd, random, randtool

Name	pdf	cdf	inv	stat	fit	like	rnd
Nakagami					`dfittool`		
Non-parametric	`ksdensity`	`ksdensity`	`ksdensity`		`ksdensity`, `dfittool`		
Normal (Gaussian)	`normpdf`, `pdf`	`normcdf`, `cdf`	`norminv`, `icdf`	`normstat`	`normfit`, `mle`, `dfittool`	`normlike`	`normrnd`, `randn`, `random`, `randtool`
Pearson system							`pearsrnd`
Rayleigh	`raylpdf`, `pdf`	`raylcdf`, `cdf`	`raylinv`, `icdf`	`raylstat`	`raylfit`, `mle`, `dfittool`		`raylrnd`, `random`, `randtool`
Rician					`dfittool`		
t location-scale					`dfittool`		
Uniform (continuous)	`unifpdf`, `pdf`	`unifcdf`, `cdf`	`unifinv`, `icdf`	`unifstat`	`unifit`, `mle`		`unifrnd`, `rand`, `random`
Weibull	`wblpdf`, `pdf`	`wblcdf`, `cdf`	`wblinv`, `icdf`	`wblstat`	`wblfit`, `mle`, `dfittool`	`wbllike`	`wblrnd`, `random`

Continuous Distributions (Statistics)

Name	pdf	cdf	inv	stat	fit	like	rnd
Chi-square	`chi2pdf`, `pdf`	`chi2cdf`, `cdf`	`chi2inv`, `icdf`	`chi2stat`			`chi2rnd`, `random`, `randtool`
F	`fpdf`, `pdf`	`fcdf`, `cdf`	`finv`, `icdf`	`fstat`			`frnd`, `random`, `randtool`
Noncentral chi-square	`ncx2pdf`, `pdf`	`ncx2cdf`, `cdf`	`ncx2inv`, `icdf`	`ncx2stat`			`ncx2rnd`, `random`, `randtool`
Noncentral *F*	`ncfpdf`, `pdf`	`ncfcdf`, `cdf`	`ncfinv`, `icdf`	`ncfstat`			`ncfrnd`, `random`, `randtool`
Noncentral *t*	`nctpdf`, `pdf`	`nctcdf`, `cdf`	`nctinv`, `icdf`	`nctstat`			`nctrnd`, `random`, `randtool`
Student's *t*	`tpdf`, `pdf`	`tcdf`, `cdf`	`tinv`, `icdf`	`tstat`			`trnd`, `random`, `randtool`

Discrete Distributions

Name	pdf	cdf	inv	stat	fit	like	rnd
Binomial	`binopdf`, `pdf`	`binocdf`, `cdf`	`binoinv`, `icdf`	`binostat`	`binofit`, `mle`, `dfittool`		`binornd`, `random`, `randtool`
Bernoulli					`mle`		
Geometric	`geopdf`, `pdf`	`geocdf`, `cdf`	`geoinv`, `icdf`	`geostat`	`mle`		`geornd`, `random`, `randtool`
Hyper-geometric	`hygepdf`, `pdf`	`hygecdf`, `cdf`	`hygeinv`, `icdf`	`hygestat`			`hygernd`, `random`
Multinomial	`mnpdf`						`mnrnd`
Negative binomial	`nbinpdf`, `pdf`	`nbincdf`, `cdf`	`nbininv`, `icdf`	`nbinstat`	`nbinfit`, `mle`, `dfittool`		`nbinrnd`, `random`, `randtool`
Poisson	`poisspdf`, `pdf`	`poisscdf`, `cdf`	`poissinv`, `icdf`	`poisstat`	`poissfit`, `mle`, `dfittool`		`poissrnd`, `random`, `randtool`
Uniform (discrete)	`unidpdf`, `pdf`	`unidcdf`, `cdf`	`unidinv`, `icdf`	`unidstat`	`mle`		`unidrnd`, `random`, `randtool`

Multivariate Distributions

Name	pdf	cdf	inv	stat	fit	like	rnd
Gaussian copula	`copulapdf`	`copulacdf`		`copulastat`			`copularnd`
t copula	`copulapdf`	`copulacdf`		`copulastat`			`copularnd`
Clayton copula	`copulapdf`	`copulacdf`		`copulastat`			`copularnd`
Frank copula	`copulapdf`	`copulacdf`		`copulastat`			`copularnd`
Gumbel copula	`copulapdf`	`copulacdf`		`copulastat`			`copularnd`
Inverse Wishart							`iwishrnd`
Multivariate normal	`mvnpdf`	`mvncdf`					`mvnrnd`
Multivariate *t*	`mvtpdf`	`mvtcdf`					`mvtrnd`
Wishart							`wishrnd`

Distribution Reference

This section provides reference information on the following probability distributions supported by Statistics Toolbox functions and GUIs:

Bernoulli Distribution

Definition of the Bernoulli Distribution

The Bernoulli distribution is a special case of the binomial distribution, with $n = 1$.

Beta Distribution

Definition of the Beta Distribution

The beta pdf is

$$y = f(x|a,b) = \frac{1}{B(a,b)}x^{a-1}(1-x)^{b-1}I_{(0,1)}(x)$$

where $B(\cdot)$ is the Beta function. The indicator function $I_{(0,1)}(x)$ ensures that only values of x in the range (0 1) have nonzero probability.

Background on the Beta Distribution

The beta distribution describes a family of curves that are unique in that they are nonzero only on the interval (0 1). A more general version of the function assigns parameters to the endpoints of the interval.

The beta cdf is the same as the incomplete beta function.

The beta distribution has a functional relationship with the t distribution. If Y is an observation from Student's t distribution with ν degrees of freedom, then the following transformation generates X, which is beta distributed.

$$X = \frac{1}{2}+\frac{1}{2}\frac{Y}{\sqrt{\nu+Y^2}}$$

If $Y \sim t(\nu)$, then $X \sim \beta\left(\frac{\nu}{2},\frac{\nu}{2}\right)$

Statistics Toolbox uses this relationship to compute values of the t cdf and inverse function as well as generating t distributed random numbers.

Parameter Estimation for the Beta Distribution

Suppose you are collecting data that has hard lower and upper bounds of zero and one respectively. Parameter estimation is the process of determining the parameters of the beta distribution that fit this data best in some sense.

One popular criterion of goodness is to maximize the likelihood function. The likelihood has the same form as the beta pdf. But for the pdf, the parameters are known constants and the variable is x. The likelihood function reverses the roles of the variables. Here, the sample values (the x's) are already observed. So they are the fixed constants. The variables are the unknown parameters. Maximum likelihood estimation (MLE) involves calculating the values of the parameters that give the highest likelihood given the particular set of data.

The function `betafit` returns the MLEs and confidence intervals for the parameters of the beta distribution. Here is an example using random numbers from the beta distribution with $a = 5$ and $b = 0.2$.

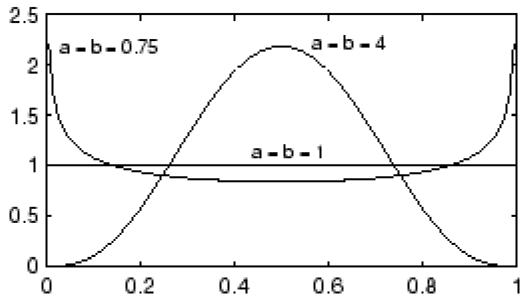
```
r = betarnd(5,0.2,100,1);
[phat, pci] = betafit(r)

phat =
    4.5330    0.2301

pci =
    2.8051    0.1771
    6.2610    0.2832
```

The MLE for parameter a is 4.5330, compared to the true value of 5. The 95% confidence interval for a goes from 2.8051 to 6.2610, which includes the true value.

Similarly the MLE for parameter b is 0.2301, compared to the true value of 0.2. The 95% confidence interval for b goes from 0.1771 to 0.2832, which also includes the true value. In this made-up example you know the "true value." In experimentation you do not.

Example and Plot of the Beta Distribution

The shape of the beta distribution is quite variable depending on the values of the parameters, as illustrated by the plot below.



The constant pdf (the flat line) shows that the standard uniform distribution is a special case of the beta distribution.

Binomial Distribution

Definition of the Binomial Distribution

The binomial pdf is

$$f(k \mid n, p) = \binom{n}{k} p^k (1-p)^{n-k}$$

where k is the number of successes in n trials of a Bernoulli process with probability of success p.

The binomial distribution is discrete, defined for integers k = 0, 1, 2, ... n, where it is nonzero.

Background of the Binomial Distribution

The binomial distribution models the total number of successes in repeated trials from an infinite population under the following conditions:

- Only two outcomes are possible on each of n trials.
- The probability of success for each trial is constant.
- All trials are independent of each other.

James Bernoulli derived the binomial distribution in 1713. Earlier, Blaise Pascal had considered the special case where p = 1/2.

The binomial distribution is a generalization of the Bernoulli distribution; it generalizes to the multinomial distribution.

Parameter Estimation for the Binomial Distribution

Suppose you are collecting data from a widget manufacturing process, and you record the number of widgets within specification in each batch of 100. You might be interested in the probability that an individual widget is within specification. Parameter estimation is the process of determining the parameter, p, of the binomial distribution that fits this data best in some sense.

One popular criterion of goodness is to maximize the likelihood function. The likelihood has the same form as the binomial pdf above. But for the pdf, the parameters (n and p) are known constants and the variable is x. The likelihood function reverses the roles of the variables. Here, the sample values (the x's) are already observed. So they are the fixed constants. The variables are the unknown parameters. MLE involves calculating the value of p that give the highest likelihood given the particular set of data.

The function `binofit` returns the MLEs and confidence intervals for the parameters of the binomial distribution. Here is an example using random numbers from the binomial distribution with $n = 100$ and $p = 0.9$.

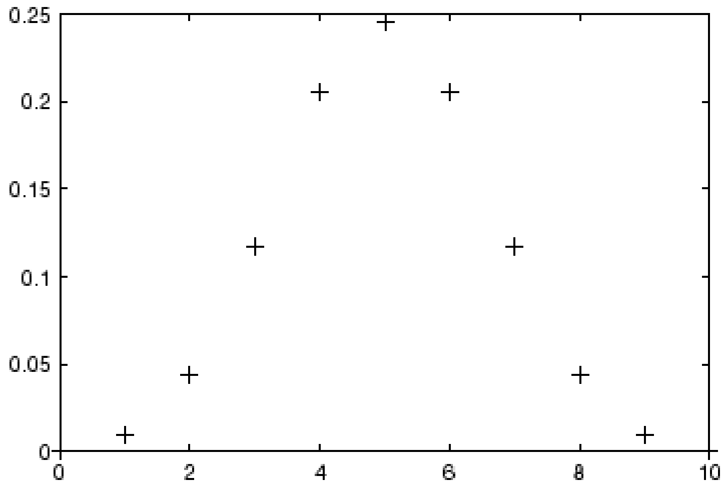
```
r = binornd(100,0.9)

r =
    88

[phat, pci] = binofit(r,100)

phat =
    0.8800

pci =
    0.7998
    0.9364
```

The MLE for parameter p is 0.8800, compared to the true value of 0.9. The 95% confidence interval for p goes from 0.7998 to 0.9364, which includes the true value. In this made-up example you know the "true value" of p. In experimentation you do not.

Example and Plot of the Binomial Distribution

The following commands generate a plot of the binomial pdf for $n = 10$ and $p = 1/2$.

```
x = 0:10;
y = binopdf(x,10,0.5);
plot(x,y,'+')
```



0.25
0.2
0.15
0.1
0.05
0
0
2
4
6
8
10

Birnbaum-Saunders Distribution

Definition of the Birnbaum-Saunders Distribution

The Birnbaum-Saunders distribution has the density function

$$\frac{1}{\sqrt{2\pi}}\exp\left\{-\frac{\left(\sqrt{x/\beta}-\sqrt{\beta/x}\right)^2}{2\gamma^2}\right\}\left(\frac{\left(\sqrt{x/\beta}+\sqrt{\beta/x}\right)}{2\gamma x}\right)$$

with scale parameter $\beta > 0$ and shape parameter $\gamma > 0$, for $x > 0$.

If x has a Birnbaum-Saunders distribution with parameters β and γ, then

$$\frac{1}{\gamma}\left(\sqrt{x/\beta}+\sqrt{\beta/x}\right)$$

has a standard normal distribution.

Background on the Birnbaum-Saunders Distribution

The Birnbaum-Saunders distribution was originally proposed as a lifetime model for materials subject to cyclic patterns of stress and strain, where the ultimate failure of the material comes from the growth of a prominent flaw. In materials science, Miner's Rule suggests that the damage occurring after n cycles, at a stress level with an expected lifetime of N cycles, is proportional to n / N. Whenever Miner's Rule applies, the Birnbaum-Saunders model is a reasonable choice for a lifetime distribution model.

Parameter Estimation for the Birnbaum-Saunders Distribution

See `mle`, `dfittool`.

Chi-Square Distribution

Definition of the Chi-Square Distribution

The χ^2 pdf is

$$y = f(x|\nu) = \frac{x^{(\nu-2)/2}e^{-x/2}}{2^{\frac{\nu}{2}}\Gamma(\nu/2)}$$

where $\Gamma(\cdot)$ is the Gamma function, and ν is the degrees of freedom.

Background of the Chi-Square Distribution

The χ^2 distribution is a special case of the gamma distribution where $b = 2$ in the equation for gamma distribution below.

$$y = f(x|a,b) = \frac{1}{b^a\Gamma(a)}x^{a-1}e^{-\frac{x}{b}}$$

The χ^2 distribution gets special attention because of its importance in normal sampling theory. If a set of n observations is normally distributed with variance σ^2, and s^2 is the sample standard deviation, then

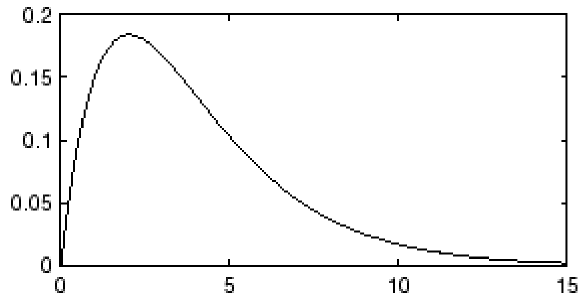
$$\frac{(n-1)s^2}{\sigma^2} \sim \chi^2(n-1)$$

Statistics Toolbox uses the above relationship to calculate confidence intervals for the estimate of the normal parameter σ^2 in the function `normfit`.

Example and Plot of the Chi-Square Distribution

The χ^2 distribution is skewed to the right especially for few degrees of freedom (ν). The plot shows the χ^2 distribution with four degrees of freedom.

```
x = 0:0.2:15;
y = chi2pdf(x,4);
plot(x,y)
```


0.2
0.15
0.1
0.05
0
0
5
10
15

Copulas

See the “Copulas” on page 5-174 entry in “Random Number Generation” on page 5-158.

Custom Distributions

User-defined custom distributions, created using M-files and function handles, are supported by the Statistics Toolbox functions `pdf`, `cdf`, `icdf`, and `mle`, and the Statistics Toolbox GUI `dfittool`.

Exponential Distribution

Definition of the Exponential Distribution

The exponential pdf is

$$y = f(x|\mu) = \frac{1}{\mu}e^{-\frac{x}{\mu}}$$

Background of the Exponential Distribution

Like the chi-square distribution, the exponential distribution is a special case of the gamma distribution (obtained by setting $a = 1$)

$$y = f(x|a,b) = \frac{1}{b^a\Gamma(a)}x^{a-1}e^{-\frac{x}{b}}$$

where $\Gamma(\cdot)$ is the Gamma function.

The exponential distribution is special because of its utility in modeling events that occur randomly over time. The main application area is in studies of lifetimes.

Parameter Estimation for the Exponential Distribution

Suppose you are stress testing light bulbs and collecting data on their lifetimes. You assume that these lifetimes follow an exponential distribution. You want to know how long you can expect the average light bulb to last. Parameter estimation is the process of determining the parameters of the exponential distribution that fit this data best in some sense.

One popular criterion of goodness is to maximize the likelihood function. The likelihood has the same form as the exponential pdf above. But for the pdf, the parameters are known constants and the variable is x. The likelihood function reverses the roles of the variables. Here, the sample values (the x's) are already observed. So they are the fixed constants. The variables are the unknown parameters. MLE involves calculating the values of the parameters that give the highest likelihood given the particular set of data.

The function `expfit` returns the MLEs and confidence intervals for the parameters of the exponential distribution. Here is an example using random numbers from the exponential distribution with μ = 700.

```
lifetimes = exprnd(700,100,1);
[muhat, muci] = expfit(lifetimes)

muhat =

  672.8207

muci =

  547.4338
  810.9437
```

The MLE for parameter μ is 672, compared to the true value of 700. The 95% confidence interval for μ goes from 547 to 811, which includes the true value.

In the life tests you do not know the true value of μ so it is nice to have a confidence interval on the parameter to give a range of likely values.

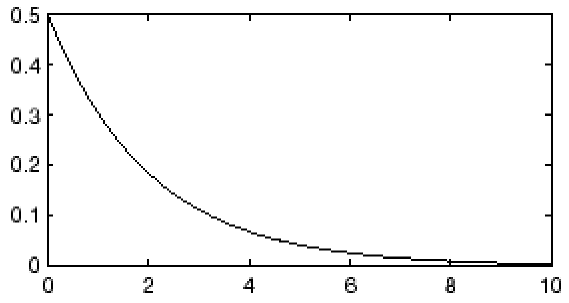
Example and Plot of the Exponential Distribution

For exponentially distributed lifetimes, the probability that an item will survive an extra unit of time is independent of the current age of the item. The example shows a specific case of this special property.

```
l = 10:10:60;
lpd = l+0.1;
deltap = (expcdf(lpd,50)-expcdf(l,50))./(1-expcdf(l,50))

deltap =
    0.0020    0.0020    0.0020    0.0020    0.0020    0.0020
```

The following commands generate a plot of the exponential pdf with its parameter (and mean), μ, set to 2.

```
x = 0:0.1:10;
y = exppdf(x,2);
plot(x,y)
```


0.5
0.4
0.3
0.2
0.1
0
0
2
4
6
8
10

Extreme Value Distribution

Definition of the Extreme Value Distribution

The probability density function for the extreme value distribution with location parameter μ and scale parameter σ is

$$y = f(x|\mu,\sigma) = \sigma^{-1}\exp\left(\frac{x-\mu}{\sigma}\right)\exp\left(-\exp\left(\frac{x-\mu}{\sigma}\right)\right)$$

If T has a Weibull distribution with parameters a and b, as described in "Weibull Distribution" on page 5-90, then log T has an extreme value distribution with parameters $\mu = \log a$ and sim$\sigma = 1/b$.

Background of the Extreme Value Distribution

Extreme value distributions are often used to model the smallest or largest value among a large set of independent, identically distributed random values representing measurements or observations. The extreme value distribution used in Statistics Toolbox is appropriate for modeling the smallest value from a distribution whose tails decay exponentially fast, for example, the normal distribution. It can also model the largest value from a distribution, such as the normal or exponential distributions, by using the negative of the original values.

For example, the values generated by the following code have approximately an extreme value distribution.

```
xmin = min(randn(1000,5), [], 1);
negxmax = -max(randn(1000,5), [], 1);
```

Although the extreme value distribution is most often used as a model for extreme values, you can also use it as a model for other types of continuous data. For example, extreme value distributions are closely related to the Weibull distribution. If T has a Weibull distribution, then log(T) has a type 1 extreme value distribution.

Parameter Estimation for the Extreme Value Distribution

The function `evfit` returns the maximum likelihood estimates (MLEs) and confidence intervals for the parameters of the extreme value distribution. The following example shows how to fit some sample data using evfit, including estimates of the mean and variance from the fitted distribution.

Suppose you want to model the size of the smallest washer in each batch of 1000 from a manufacturing process. If you believe that the sizes are independent within and between each batch, you can fit an extreme value distribution to measurements of the minimum diameter from a series of eight experimental batches. The following code returns the MLEs of the distribution parameters as parmhat and the confidence intervals as the columns of parmci.

```
x = [19.774 20.141 19.44 20.511 21.377 19.003 19.66 18.83];
[parmhat, parmci] = evfit(x)

parmhat =
   20.2506    0.8223

parmci =
   19.644 0.49861
   20.857 1.3562
```

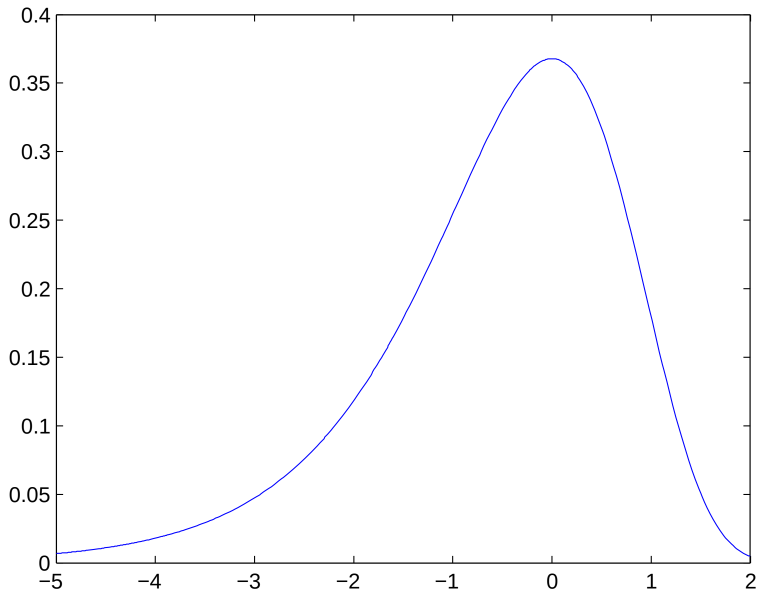
You can find mean and variance of the extreme value distribution with these parameters using the function evstat.

```
[meanfit, varfit] = evstat(parmhat(1),parmhat(2))

meanfit =
   19.776

varfit =
   1.1123
```

Plot of the Extreme Value Distribution

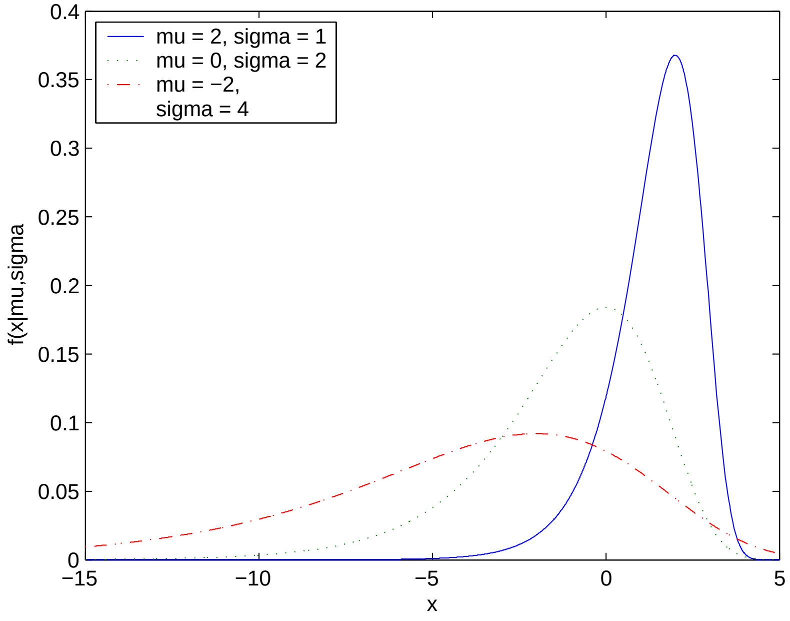
The following code generates a plot of the pdf for the extreme value distribution.

```
t = [-5:.01:2];
y = evpdf(t);
plot(t, y)
```



The extreme value distribution is skewed to the left, and its general shape remains the same for all parameter values. The location parameter, `mu`, shifts the distribution along the real line, and the scale parameter, `sigma`, expands or contracts the distribution. This example plots the probability function for different combinations of `mu` and `sigma`.

```
x = -15:.01:5;
plot(x,evpdf(x,2,1),'-', x,evpdf(x,0,2),':',
x,evpdf(x,-2,4),'-.');
legend({'mu = 2, sigma = 1' 'mu = 0, sigma = 2' 'mu = -2,'...
'sigma = 4'},2)
xlabel('x')
ylabel('f(x|mu,sigma')
```


mu = 2, sigma = 1
mu = 0, sigma = 2
mu = −2, sigma = 4
f(x|mu,sigma)
x
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
−15
−10
−5
0
5

F Distribution

Definition of the F Distribution

The pdf for the *F* distribution is

$$y = f(x|\nu_1,\nu_2) = \frac{\Gamma\left[\frac{(\nu_1+\nu_2)}{2}\right]}{\Gamma\left(\frac{\nu_1}{2}\right)\Gamma\left(\frac{\nu_2}{2}\right)}\left(\frac{\nu_1}{\nu_2}\right)^{\frac{\nu_1}{2}}\frac{x^{\frac{\nu_1-2}{2}}}{\left[1+\left(\frac{\nu_1}{\nu_2}\right)x\right]^{\frac{\nu_1+\nu_2}{2}}}$$

where $\Gamma(\cdot)$ is the Gamma function.

Background of the *F* distribution

The *F* distribution has a natural relationship with the chi-square distribution. If χ_1 and χ_2 are both chi-square with ν_1 and ν_2 degrees of freedom respectively, then the statistic *F* below is *F*-distributed.

$$F(\nu_1,\nu_2) = \frac{\frac{\chi_1}{\nu_1}}{\frac{\chi_2}{\nu_2}}$$

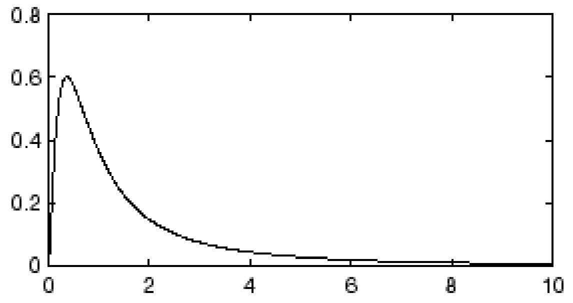
The two parameters, ν_1 and ν_2, are the numerator and denominator degrees of freedom. That is, ν_1 and ν_2 are the number of independent pieces of information used to calculate χ_1 and χ_2, respectively.

Example and Plot of the F Distribution

The most common application of the *F* distribution is in standard tests of hypotheses in analysis of variance and regression.

The plot shows that the *F* distribution exists on the positive real numbers and is skewed to the right.

```
x = 0:0.01:10;
y = fpdf(x,5,3);
plot(x,y)
```


0.8
0.6
0.4
0.2
0
0
2
4
6
8
10

Gamma Distribution

Definition of the Gamma Distribution

The gamma pdf is

$$y = f(x|a,b) = \frac{1}{b^a\Gamma(a)}x^{a-1}e^{-\frac{x}{b}}$$

where $\Gamma(\cdot)$ is the Gamma function.

Background of the Gamma Distribution

The gamma distribution models sums of exponentially distributed random variables.

The gamma distribution family is based on two parameters. The chi-square and exponential distributions, which are children of the gamma distribution, are one-parameter distributions that fix one of the two gamma parameters.

The gamma distribution has the following relationship with the incomplete Gamma function.

$$f(x \mid a,b) = \mathrm{gammainc}(\frac{x}{b},a)$$

For $b = 1$ the functions are identical.

When a is large, the gamma distribution closely approximates a normal distribution with the advantage that the gamma distribution has density only for positive real numbers.

Parameter Estimation for the Gamma Distribution

Suppose you are stress testing computer memory chips and collecting data on their lifetimes. You assume that these lifetimes follow a gamma distribution. You want to know how long you can expect the average computer memory chip to last. Parameter estimation is the process of determining the parameters of the gamma distribution that fit this data best in some sense.

One popular criterion of goodness is to maximize the likelihood function. The likelihood has the same form as the gamma pdf above. But for the pdf, the parameters are known constants and the variable is x. The likelihood function reverses the roles of the variables. Here, the sample values (the x's) are already observed. So they are the fixed constants. The variables are the unknown parameters. MLE involves calculating the values of the parameters that give the highest likelihood given the particular set of data.

The function gamfit returns the MLEs and confidence intervals for the parameters of the gamma distribution. Here is an example using random numbers from the gamma distribution with $a = 10$ and $b = 5$.

```
lifetimes = gamrnd(10,5,100,1);
[phat, pci] = gamfit(lifetimes)

phat =

    10.9821    4.7258

pci =

    7.4001    3.1543
   14.5640    6.2974
```

Note phat(1) = $\hat{a}$ and phat(2) = $\hat{b}$. The MLE for parameter a is 10.98, compared to the true value of 10. The 95% confidence interval for a goes from 7.4 to 14.6, which includes the true value.

Similarly the MLE for parameter b is 4.7, compared to the true value of 5. The 95% confidence interval for b goes from 3.2 to 6.3, which also includes the true value.

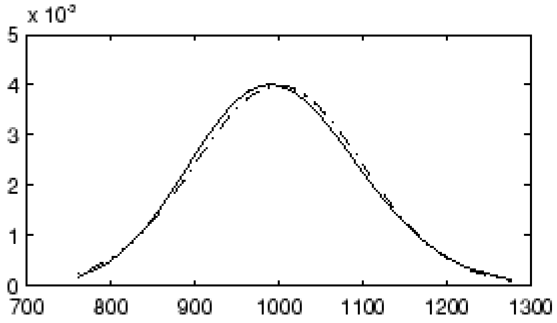
In the life tests you do not know the true value of a and b so it is nice to have a confidence interval on the parameters to give a range of likely values.

Example and Plot of the Gamma Distribution

In the example the gamma pdf is plotted with the solid line. The normal pdf has a dashed line type.

```
x = gaminv((0.005:0.01:0.995),100,10);
```

```
y = gampdf(x,100,10);
y1 = normpdf(x,1000,100);
plot(x,y,'-',x,y1,'-.')
```



Generalized Extreme Value Distribution

Definition of the Generalized Extreme Value Distribution

The probability density function for the generalized extreme value distribution with location parameter μ, scale parameter σ, and shape parameter $k \neq 0$ is

$$y = f(x|k,\mu,\sigma) = \left(\frac{1}{\sigma}\right)\exp\left(-\left(1+k\frac{(x-\mu)}{\sigma}\right)^{-\frac{1}{k}}\right)\left(1+k\frac{(x-\mu)}{\sigma}\right)^{-1-\frac{1}{k}}$$

for

$$1 + k\frac{(x-\mu)}{\sigma} > 0$$

`k > 0` corresponds to the Type II case, while `k < 0` corresponds to the Type III case. In the limit for `k = 0`, corresponding to the Type I case, the density is

$$y = f(x|0,\mu,\sigma) = \left(\frac{1}{\sigma}\right)\exp\left(-\exp\left(-\frac{(x-\mu)}{\sigma}\right)-\frac{(x-\mu)}{\sigma}\right)$$

Background of the Generalized Extreme Value Distribution

Like the extreme value distribution, the generalized extreme value distribution is often used to model the smallest or largest value among a large set of independent, identically distributed random values representing measurements or observations. For example, you might have batches of 1000 washers from a manufacturing process. If you record the size of the largest washer in each batch, the data are known as block maxima (or minima if you record the smallest). You can use the generalized extreme value distribution as a model for those block maxima.

The generalized extreme value combines three simpler distributions into a single form, allowing a continuous range of possible shapes that includes all three of the simpler distributions. You can use any one of those distributions to model a particular dataset of block maxima. The generalized extreme value distribution allows you to "let the data decide" which distribution is appropriate.

The three cases covered by the generalized extreme value distribution are often referred to as the Types I, II, and III. Each type corresponds to the limiting distribution of block maxima from a different class of underlying distributions. Distributions whose tails decrease exponentially, such as the normal, lead to the Type I. Distributions whose tails decrease as a polynomial, such as Student's t, lead to the Type II. Distributions whose tails are finite, such as the beta, lead to the Type III.

Types I, II, and III are sometimes also referred to as the Gumbel, Frechet, and Weibull types, though this terminology can be slightly confusing. The Type I (Gumbel) and Type III (Weibull) cases actually correspond to the mirror images of the usual Gumbel and Weibull distributions, for example, as computed by the functions `evcdf` and `evfit` , or `wblcdf` and `wblfit`, respectively. Finally, the Type II (Frechet) case is equivalent to taking the reciprocal of values from a standard Weibull distribution.

Parameter Estimation for the Generalized Extreme Value Distribution

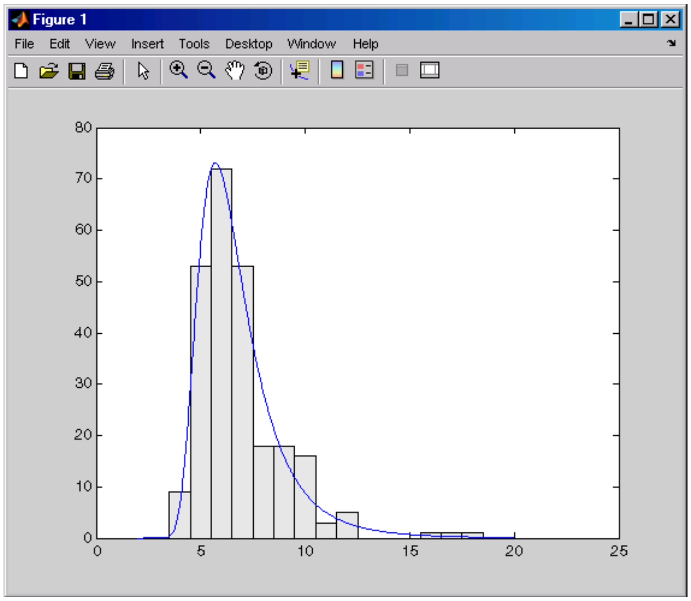
If you generate 250 blocks of 1000 random values drawn from Student's t distribution with 5 degrees of freedom, and take their maxima, you can fit a generalized extreme value distribution to those maxima.

```
blocksize = 1000;
nblocks = 250;
t = trnd(5,blocksize,nblocks);
x = max(t); % 250 column maxima
paramEsts = gevfit(x)
paramEsts =

    0.1507    1.2712    5.8816
```

Notice that the shape parameter estimate (the first element) is positive, which is what you would expect based on block maxima from a Student's t distribution.

```
hist(x,2:20);
set(get(gca,'child'),'FaceColor',[.9 .9 .9])
```

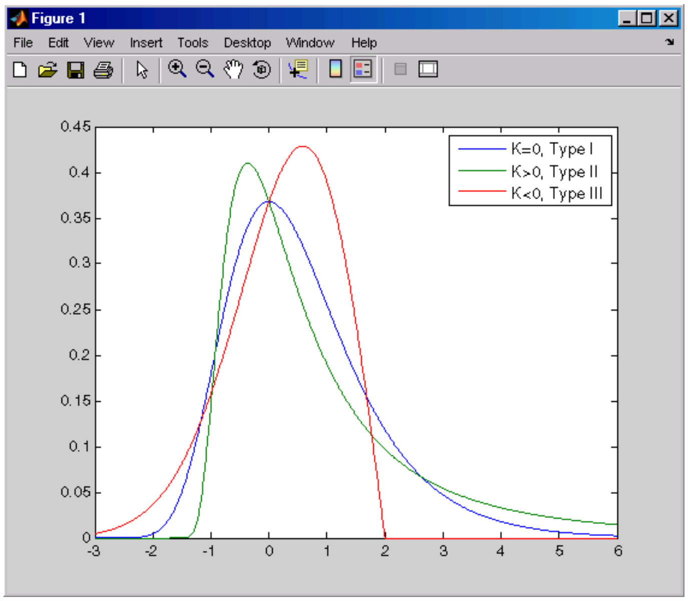
```
xgrid = linspace(2,20,1000);
line(xgrid,nblocks*...
     gevpdf(xgrid,paramEsts(1),paramEsts(2),paramEsts(3)));
```



Plot of the Generalized Extreme Value Distribution

The following code generates examples of probability density functions for the three basic forms of the generalized extreme value distribution.

```
x = linspace(-3,6,1000);
y1 = gevpdf(x,-.5,1,0);
y2 = gevpdf(x,0,1,0);
y3 = gevpdf(x,.5,1,0)
plot(x,y1,'-', x,y2,'-', x,y3,'-')
legend({'K<0, Type III' 'K=0, Type I' 'K>0, Type II'});
```



Notice that for k > 0, the distribution has zero probability density for x such that

$$x < -\frac{\sigma}{k} + \mu$$

For k < 0, the distribution has zero probability density for

$$x > -\frac{\sigma}{k} + \mu$$

In the limit for k = 0, there is no upper or lower bound.

Generalized Pareto Distribution

Definition of the Generalized Pareto Distribution

The probability density function for the generalized Pareto distribution with shape parameter $k \neq 0$, scale parameter σ, and threshold parameter θ, is

$$y = f(x|k,\sigma,\theta) = \left(\frac{1}{\sigma}\right)\left(1 + k\frac{(x-\theta)}{\sigma}\right)^{-1-\frac{1}{k}}$$

for $\theta < x$, when $k > 0$, or for $\theta < x < -\frac{\sigma}{k}$ when $k < 0$.

In the limit for $k = 0$, the density is

$$y = f(x|0,\sigma,\theta) = \left(\frac{1}{\sigma}\right)e^{-\frac{(x-\theta)}{\sigma}}$$

for $\theta < x$.

If $k = 0$ and $\theta = 0$, the generalized Pareto distribution is equivalent to the exponential distribution. If $k > 0$ and $\theta = \sigma$, the generalized Pareto distribution is equivalent to the Pareto distribution.

Background of the Generalized Pareto Distribution

Like the exponential distribution, the generalized Pareto distribution is often used to model the tails of another distribution. For example, you might have washers from a manufacturing process. If random influences in the process lead to differences in the sizes of the washers, a standard probability distribution, such as the normal, could be used to model those sizes. However, while the normal distribution might be a good model near its mode, it might not be a good fit to real data in the tails and a more complex model might be needed to describe the full range of the data. On the other hand, only recording the sizes of washers larger (or smaller) than a certain threshold means you can fit a separate model to those tail data, which are known as *exceedences*. You can use the generalized Pareto distribution in this way, to provide a good fit to extremes of complicated data.

The generalized Pareto distribution allows a continuous range of possible shapes that includes both the exponential and Pareto distributions as special cases. You can use either of those distributions to model a particular dataset of exceedences. The generalized extreme value distribution allows you to "let the data decide" which distribution is appropriate.

The generalized Pareto distribution has three basic forms, each corresponding to a limiting distribution of exceedence data from a different class of underlying distributions.

- Distributions whose tails decrease exponentially, such as the normal, lead to a generalized Pareto shape parameter of zero.
- Distributions whose tails decrease as a polynomial, such as Student's t, lead to a positive shape parameter.
- Distributions whose tails are finite, such as the beta, lead to a negative shape parameter.

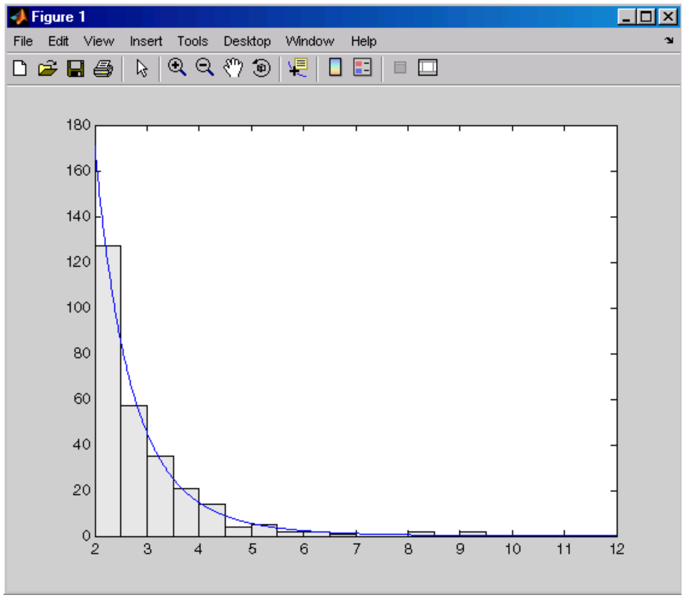
Parameter Estimation for the Generalized Pareto Distribution

If you generate a large number of random values from a Student's t distribution with 5 degrees of freedom, and then discard everything less than 2, you can fit a generalized Pareto distribution to those exceedences.

```
t = trnd(5,5000,1);
y = t(t > 2) - 2;
paramEsts = gpfit(y)
paramEsts =

    0.1598    0.7968
```

Notice that the shape parameter estimate (the first element) is positive, which is what you would expect based on exceedences from a Student's t distribution.

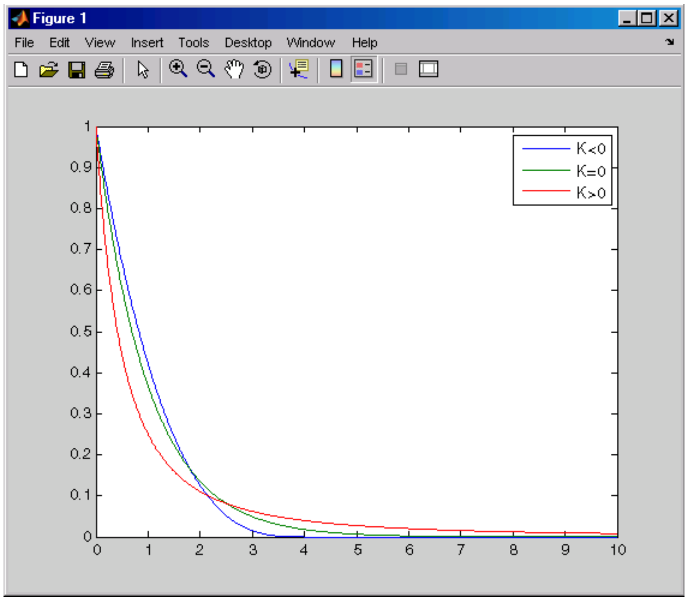
```
hist(y+2,2.25:.5:11.75);
set(get(gca,'child'),'FaceColor',[.9 .9 .9])
xgrid = linspace(2,12,1000);
line(xgrid,.5*length(y)*...
     gppdf(xgrid,paramEsts(1),paramEsts(2),2));
```



Plot of the Generalized Pareto Distribution

The following code generates examples of the probability density functions for the three basic forms of the generalized Pareto distribution.

```
x = linspace(0,10,1000);
y1 = gppdf(x,-.25,1,0);
y2 = gppdf(x,0,1,0);
y3 = gppdf(x,1,1,0)
plot(x,y1,'-', x,y2,'-', x,y3,'-')
legend({'K<0' 'K=0' 'K>0'});
```



Notice that for $k < 0$, the distribution has zero probability density for $x > -\frac{\sigma}{k}$, while for $k \geq 0$, there is no upper bound.

Geometric Distribution

Definition of the Geometric Distribution

The geometric pdf is

$$y = f(x|p) = pq^x I_{(0, 1, \ldots)}(x)$$

where $q = 1 - p$. The geometric distribution is a special case of the negative binomial distribution, with $r = 1$.

Background of the Geometric Distribution

The geometric distribution is discrete, existing only on the nonnegative integers. It is useful for modeling the runs of consecutive successes (or failures) in repeated independent trials of a system.

The geometric distribution models the number of successes before one failure in an independent succession of tests where each test results in success or failure.

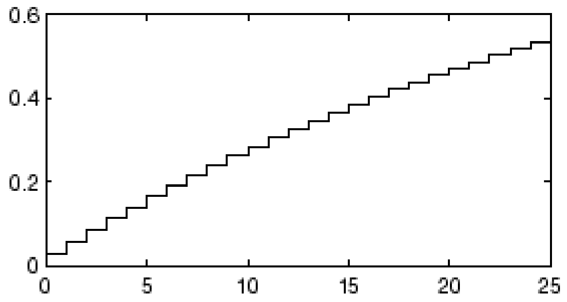
Example and Plot of the Geometric Distribution

Suppose the probability of a five-year-old battery failing in cold weather is 0.03. What is the probability of starting 25 consecutive days during a long cold snap?

```
1 - geocdf(25,0.03)

ans =

    0.4530
```

The plot shows the cdf for this scenario.

```
x = 0:25;
y = geocdf(x,0.03);
stairs(x,y)
```


0.6
0.4
0.2
0
0
5
10
15
20
25

Hypergeometric Distribution

Definition of the Hypergeometric Distribution

The hypergeometric pdf is

$$y = f(x|M,K,n) = \frac{\binom{K}{x}\binom{M-K}{n-x}}{\binom{M}{n}}$$

Background of the Hypergeometric Distribution

The hypergeometric distribution models the total number of successes in a fixed-size sample drawn without replacement from a finite population.

The distribution is discrete, existing only for nonnegative integers less than the number of samples or the number of possible successes, whichever is greater. The hypergeometric distribution differs from the binomial only in that the population is finite and the sampling from the population is without replacement.

The hypergeometric distribution has three parameters that have direct physical interpretations.

- *M* is the size of the population.
- *K* is the number of items with the desired characteristic in the population.
- *n* is the number of samples drawn.

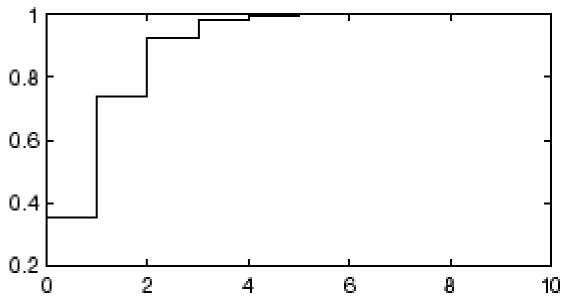
Sampling "without replacement" means that once a particular sample is chosen, it is removed from the relevant population for all subsequent selections.

Example and Plot of the Hypergeometric Distribution

The plot shows the cdf of an experiment taking 20 samples from a group of 1000 where there are 50 items of the desired type.

```
x = 0:10;
y = hygecdf(x,1000,50,20);
```

```
stairs(x,y)
```



Inverse Gaussian Distribution

Definition of the Inverse Gaussian Distribution

The inverse Gaussian distribution has the density function

$$\sqrt{\frac{\lambda}{2\pi x^3}}\exp\left\{-\frac{\lambda}{2\mu^2 x}(x-\mu)^2\right\}$$

Background on the Inverse Gaussian Distribution

Also known as the Wald distribution, the inverse Gaussian is used to model nonnegative positively skewed data. The distribution originated in the theory of Brownian motion, but has been used to model diverse phenomena. Inverse Gaussian distributions have many similarities to standard Gaussian (normal) distributions, which lead to applications in inferential statistics.

Parameter estimation for the Inverse Gaussian Distribution

See `mle`, `dfittool`.

Inverse Wishart Distribution

Definition of the Inverse Wishart Distribution

The inverse Wishart distribution is based on the Wishart distribution. If a random matrix has a Wishart distribution with parameters Σ^{-1} and ν, then the inverse of that random matrix has an inverse Wishart distribution with parameters Σ and ν. The mean of the distribution is given by

$$\frac{\Sigma}{\nu - d - 1}$$

Statistics Toolbox only supports random matrix generation for the inverse Wishart, and only for nonsingular Σ and ν greater than $d - 1$.

Johnson System of Distributions

See “Pearson and Johnson Systems of Distributions” on page 5-169.

Logistic Distribution

Definition of the Logistic Distribution

The logistic distribution has the density function

$$\frac{e^{\frac{x-\mu}{\sigma}}}{\sigma\left(1+e^{\frac{x-\mu}{\sigma}}\right)^2}$$

with location parameter μ and scale parameter $\sigma > 0$, for all real x.

Background on the Logistic Distribution

The logistic distribution originated with Verhulst's work on demography in the early 1800s. The distribution has been used for various growth models, and is used in logistic regression. It has longer tails and a higher kurtosis than the normal distribution.

Parameter estimation for the Logistic Distribution

See `mle`, `dfittool`.

Loglogistic Distribution

Definition of the Loglogistic Distribution

The variable x has a loglogistic distribution with location parameter μ and scale parameter $\sigma > 0$ if $\ln x$ has a logistic distribution with parameters μ and σ. The relationship is similar to that between the lognormal and normal distribution.

Parameter estimation for the Loglogistic Distribution

See `mle`, `dfittool`.

Lognormal Distribution

Definition of the Lognormal Distribution

The lognormal pdf is

$$y = f(x|\mu,\sigma) = \frac{1}{x\sigma\sqrt{2\pi}}e^{\frac{-(\ln x-\mu)^2}{2\sigma^2}}$$

Background of the Lognormal Distribution

The normal and lognormal distributions are closely related. If X is distributed lognormally with parameters μ and σ, then log(X) is distributed normally with mean μ and standard deviation σ.

The mean m and variance v of a lognormal random variable are functions of μ and σ that can be calculated with the `lognstat` function. They are:

$$m = \exp(\mu+\sigma^2/2)$$
$$v = \exp(2\mu+\sigma^2)\exp(\sigma^2-1)$$

A lognormal distribution with mean m and variance v has parameters

$$\mu = \log(m^2/\sqrt{v+m^2})$$
$$\sigma = \sqrt{\log(v/m^2+1)}$$

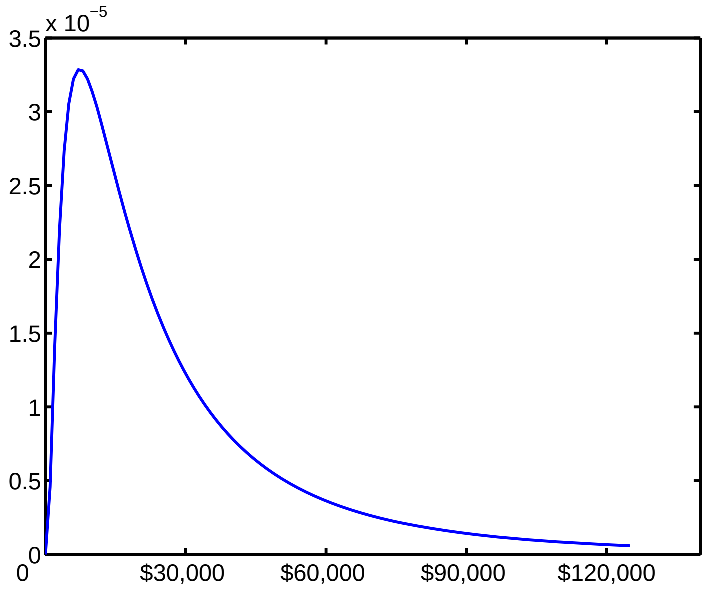
The lognormal distribution is applicable when the quantity of interest must be positive, since log(X) exists only when X is positive.

Example and Plot of the Lognormal Distribution

Suppose the income of a family of four in the United States follows a lognormal distribution with μ = log(20,000) and σ^2 = 1.0. Plot the income density.

```
x = (10:1000:125010)';
y = lognpdf(x,log(20000),1.0);
plot(x,y)
```

```
set(gca,'xtick',[0 30000 60000 90000 120000])
set(gca,'xticklabel',str2mat('0','$30,000','$60,000',...
                                   '$90,000','$120,000'))
```



Multinomial Distribution

Definition of the Multinomial Distribution

The multinomial pdf is

$$f(x\,|\,n,p) = \frac{n!}{x_1!\cdots x_k!}\,p_1^{x_1}\cdots p_k^{x_k}$$

where $x = (x_1, \ldots, x_k)$ gives the number of each of k outcomes in n trials of a process with fixed probabilities $p = (p_1, \ldots, p_k)$ of individual outcomes in any one trial. The vector x has non-negative integer components that sum to n. The vector p has non-negative integer components that sum to 1.

Background of the Multinomial Distribution

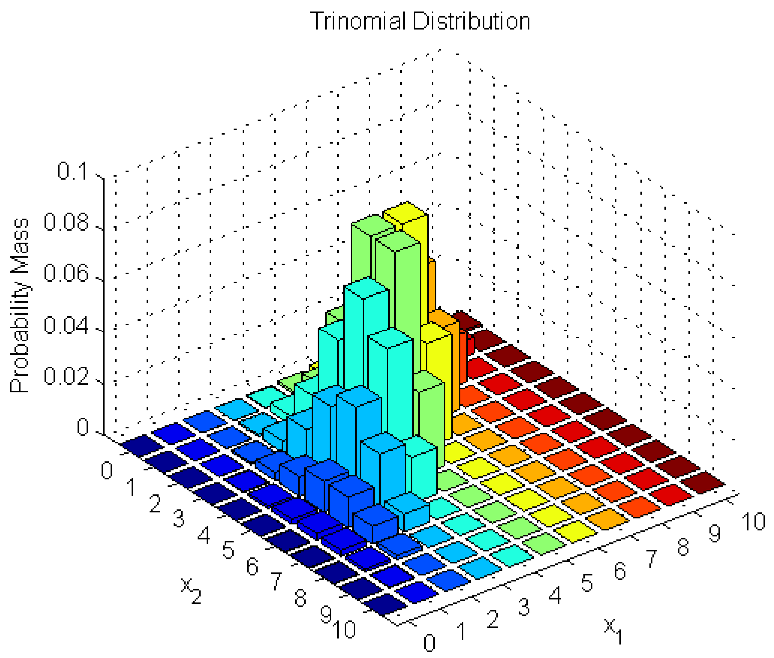
The multinomial distribution is a generalization of the binomial distribution. The binomial distribution gives the probability of the number of “successes” and “failures” in n independent trials of a two-outcome process. The probability of “success” and “failure” in any one trial is given by the fixed probabilities p and $q = 1–p$. The multinomial distribution gives the probability of each combination of outcomes in n independent trials of a k-outcome process. The probability of each outcome in any one trial is given by the fixed probabilities $p_1, \ldots, p_\mathrm{k}$.

The expected value of outcome i is np_i. The variance of outcome i is $np_i(1 - p_i)$. The covariance of outcomes i and j is $-np_ip_j$ for distinct i and j.

Example and Plot of the Multinomial Distribution

```
% Compute the distribution
p = [1/2 1/3 1/6]; % Outcome probabilities
n = 10; % Sample size
x1 = 0:n;
x2 = 0:n;
[X1,X2] = meshgrid(x1,x2);
X3 = n-(X1+X2);
Y = mnpdf([X1(:),X2(:),X3(:)],repmat(p,(n+1)^2,1));
```

```
% Plot the distribution
Y = reshape(Y,n+1,n+1);
bar3(Y)
set(gca,'XTickLabel',0:n)
set(gca,'YTickLabel',0:n)
xlabel('x_1')
ylabel('x_2')
zlabel('Probability Mass')
```



Note that the visualization does not show x_3, which is determined by the constraint $x_1 + x_2 + x_3 = n$.

Multivariate Normal Distribution

Definition of the Multivariate Normal Distribution

The probability density function of the d-dimensional multivariate normal distribution is given by

$$y = f(x,\mu,\Sigma) = \frac{e^{-(x-\mu)\,\Sigma^{-1}(x-\mu)'/2}}{|\Sigma|^{1/2}\;\frac{1}{\sqrt{(2\pi)^{d}}}}$$

where x and μ are 1-by-d vectors and Σ is a d-by-d symmetric positive definite matrix. While it is possible to define the multivariate normal for singular Σ, the density cannot be written as above. Statistics Toolbox supports only random vector generation for the singular case. Note that while most textbooks define the multivariate normal with x and μ oriented as column vectors, for the purposes of data analysis software, it is more convenient to orient them as row vectors, and Statistics Toolbox uses that orientation.

Background of the Multivariate Normal Distribution

The multivariate normal distribution is a generalization of the univariate normal to two or more variables. It is a distribution for random vectors of correlated variables, each element of which has a univariate normal distribution. In the simplest case, there is no correlation among variables, and elements of the vectors are independent univariate normal random variables.

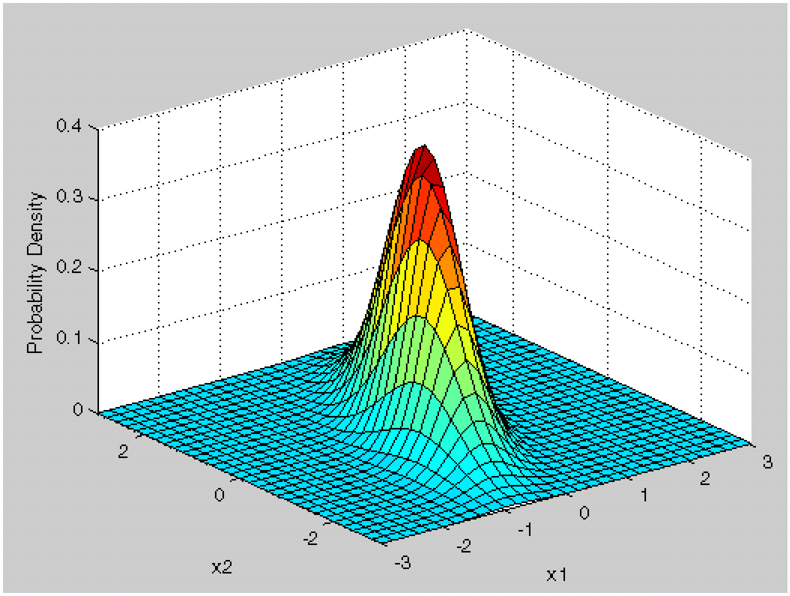
The multivariate normal distribution is parameterized with a mean vector, μ, and a covariance matrix, Σ. These are analogous to the mean μ and standard deviation σ parameters of a univariate normal distribution. The diagonal elements of Σ contain the variances for each variable, while the off-diagonal elements of Σ contain the covariances between variables.

The multivariate normal distribution is often used as a model for multivariate data, primarily because it is one of the few multivariate distributions that is tractable to work with.

Example and Plot of the Multivariate Normal Distribution

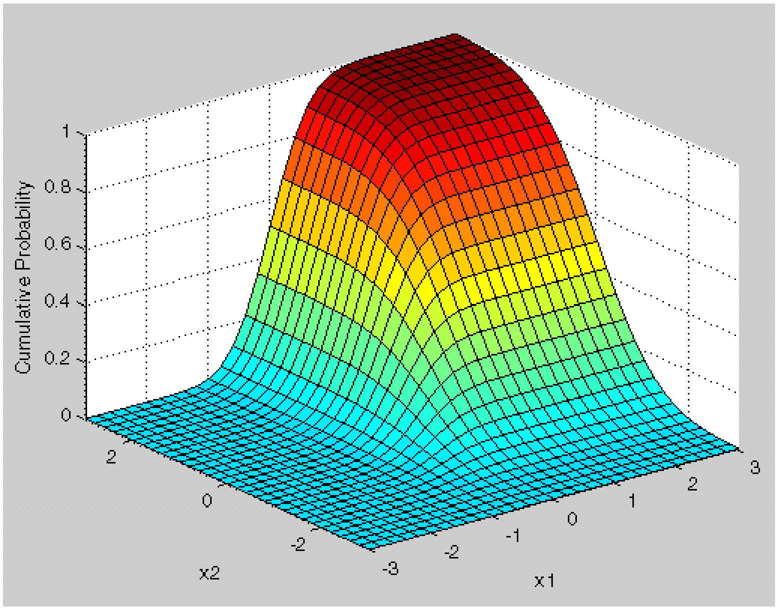
This example shows the probability density function (pdf) and cumulative distribution function (cdf) for a bivariate normal distribution with unequal standard deviations. You can use the multivariate normal distribution in a higher number of dimensions as well, although visualization is not easy.

```
mu = [0 0];
Sigma = [.25 .3; .3 1];
x1 = -3:.2:3; x2 = -3:.2:3;
[X1,X2] = meshgrid(x1,x2);
F = mvnpdf([X1(:) X2(:)],mu,Sigma);
F = reshape(F,length(x2),length(x1));
surf(x1,x2,F);
caxis([min(F(:))-.5*range(F(:)),max(F(:))]);
axis([-3 3 -3 3 0 .4])
xlabel('x1'); ylabel('x2'); zlabel('Probability Density');
```



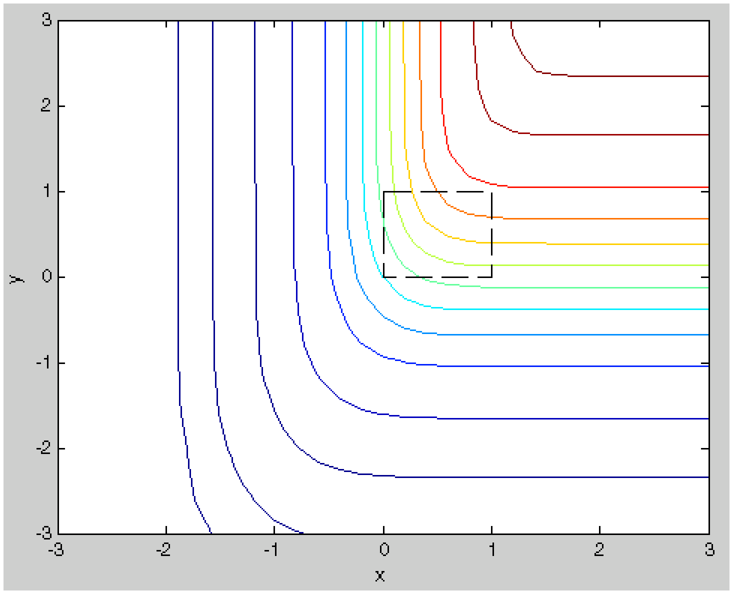
```
F = mvncdf([X1(:) X2(:)],mu,Sigma);
F = reshape(F,length(x2),length(x1));
surf(x1,x2,F);
caxis([min(F(:))-.5*range(F(:)),max(F(:))]);
```

```
axis([-3 3 -3 3 0 1])
xlabel('x1'); ylabel('x2'); zlabel('Cumulative Probability');
```



Since the bivariate normal distribution is defined on the plane, you can also compute cumulative probabilities over rectangular regions. For example, this contour plot illustrates the computation that follows, of the probability contained within the unit square.

```
contour(x1,x2,F,[.0001 .001 .01 .05:.1:.95 .99 .999 .9999]);
xlabel('x'); ylabel('y');
line([0 0 1 1 0],[1 0 0 1 1],'linestyle','--','color','k');
```



```
mvncdf([0 0],[1 1],mu,Sigma)

ans =
        0.20974
```

Computing a multivariate cumulative probability very precisely can be significantly more work than computing a univariate probability. Therefore, the `mvncdf` function computes values to less than full machine precision by default, and returns an estimate of the error as an optional second output. You can also specify a maximum error tolerance; see `mvncdf`.

```
[F,err] = mvncdf([0 0],[1 1],mu,Sigma)

F =
        0.20974
err =
         1e-008
```

Multivariate *t* Distribution

Definition of the Multivariate Student's t Distribution

The probability density function of the d-dimensional multivariate Student's t distribution is given by

$$y = f(x, P, \nu) = \frac{1}{|\Sigma|^{1/2}} \frac{1}{\sqrt{(\nu\pi)^{d}}} \frac{\Gamma((\nu+d)/2)}{\Gamma(\nu/2)} \left(1 + \frac{x' P^{-1} x}{\nu}\right)^{-(\nu+d)/2}$$

where x is a 1-by-d vector, P is a d-by-d symmetric, positive definite matrix, and ν is a positive scalar. While it is possible to define the multivariate Student's t for singular P, the density cannot be written as above. For the singular case, Statistics Toolbox only supports random number generation. Note that while most textbooks define the multivariate Student's t with x oriented as a column vector, for the purposes of data analysis software, it is more convenient to orient x as a row vector, and Statistics Toolbox uses that orientation.

Background of the Multivariate Student's t Distribution

The multivariate Student's t distribution is a generalization of the univariate Student's t to two or more variables. It is a distribution for random vectors of correlated variables, each element of which has a univariate Student's t distribution. In the same way as the univariate Student's t distribution can be constructed by dividing a standard univariate normal random variable by the square root of a univariate chi-square random variable, the multivariate Student's t distribution can be constructed by dividing a multivariate normal random vector having zero mean and unit variances by a univariate chi-square random variable.

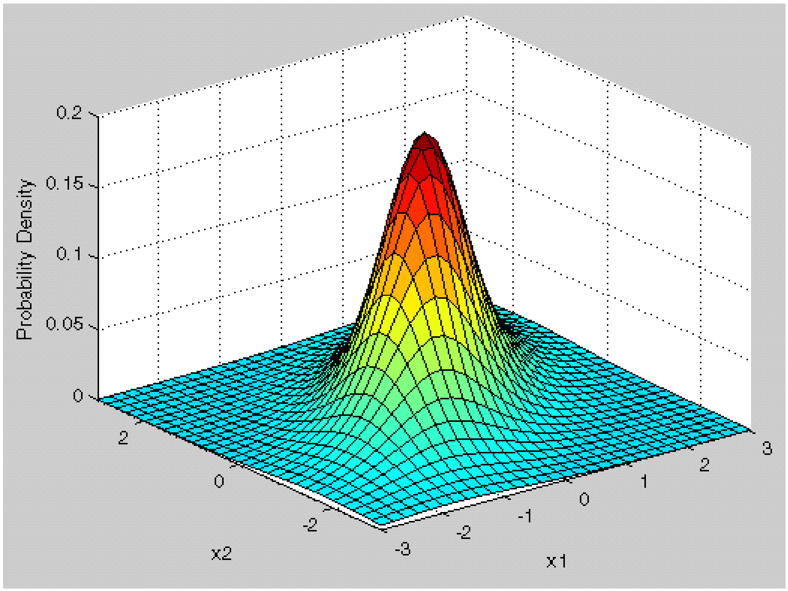
The multivariate Student's t distribution is parameterized with a correlation matrix, P, and a positive scalar degrees of freedom parameter, ν. ν is analogous to the degrees of freedom parameter of a univariate Student's t distribution. The off-diagonal elements of P contain the correlations between variables. Note that when P is the identity matrix, variables are uncorrelated; however, they are not independent.

The multivariate Student's t distribution is often used as a substitute for the multivariate normal distribution in situations where it is known that the marginal distributions of the individual variables have fatter tails than the normal.

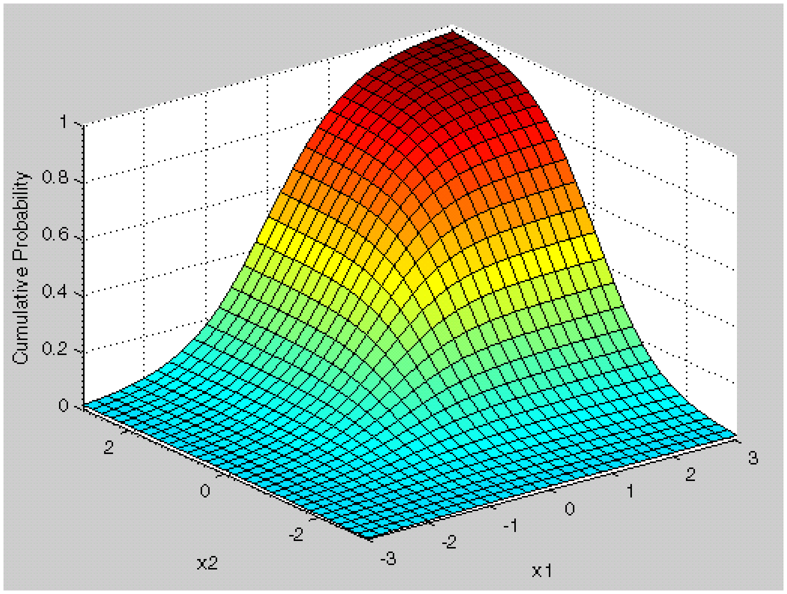
Example and Plot of the Multivariate Student's *t* Distribution

This example shows the probability density function (pdf) and cumulative distribution function (cdf) for a bivariate Student's *t* distribution. You can use the multivariate Student's *t* distribution in a higher number of dimensions as well, although visualization is not easy.

```
Rho = [1 .6; .6 1];
nu = 5;
x1 = -3:.2:3; x2 = -3:.2:3;
[X1,X2] = meshgrid(x1,x2);
F = mvtpdf([X1(:) X2(:)],Rho,nu);
F = reshape(F,length(x2),length(x1));
surf(x1,x2,F);
caxis([min(F(:))-.5*range(F(:)),max(F(:))]);
axis([-3 3 -3 3 0 .2])
xlabel('x1'); ylabel('x2'); zlabel('Probability Density');
```

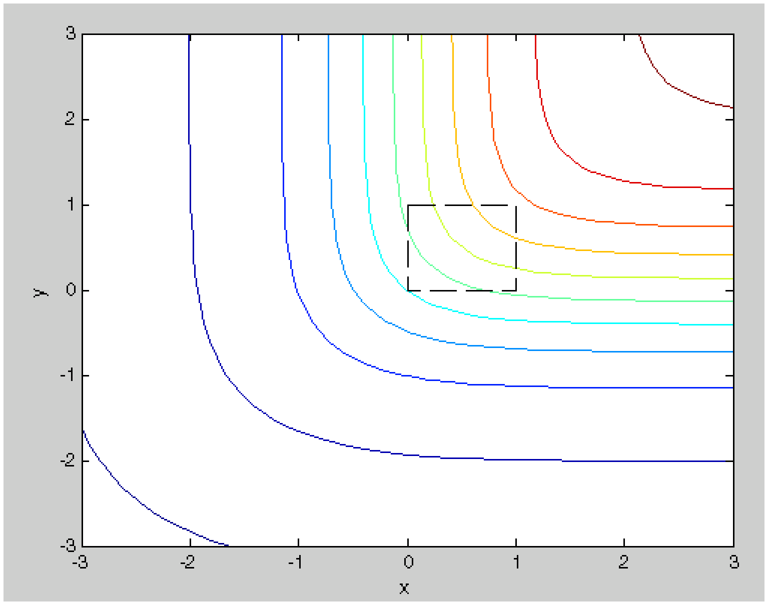


```
F = mvtcdf([X1(:) X2(:)],Rho,nu);
F = reshape(F,length(x2),length(x1));
surf(x1,x2,F);
caxis([min(F(:))-.5*range(F(:)),max(F(:))]);
axis([-3 3 -3 3 0 1])
xlabel('x1'); ylabel('x2'); zlabel('Cumulative Probability');
```



Since the bivariate Student's *t* distribution is defined on the plane, you can also compute cumulative probabilities over rectangular regions. For example, this contour plot illustrates the computation that follows, of the probability contained within the unit square.

```
contour(x1,x2,F,[.0001 .001 .01 .05:.1:.95 .99 .999 .9999]);
xlabel('x'); ylabel('y');
line([0 0 1 1 0],[1 0 0 1 1],'linestyle','--','color','k');
```



```
mvtcdf([0 0],[1 1],Rho,nu)

ans =
      0.14013
```

Computing a multivariate cumulative probability very precisely can be significantly more work than computing a univariate probability. Therefore, the `mvtcdf` function computes values to less than full machine precision by default, and returns an estimate of the error as an optional second output. You can also specify a maximum error tolerance; see `mvtcdf`.

```
[F,err] = mvtcdf([0 0],[1 1],Rho,nu)

F =
      0.14013
err =
       1e-008
```

Nakagami Distribution

Definition of the Nakagami Distribution

The Nakagami distribution has the density function

$$2\left(\frac{\mu}{\omega}\right)^{\mu}\frac{1}{\Gamma(\mu)}x^{(2\mu-1)}e^{-\frac{\mu}{\omega}x^2}$$

with shape parameter μ and scale parameter ω > 0, for $x > 0$. If x has a Nakagami distribution with parameters μ and ω, then x^2 has a gamma distribution with shape parameter μ and scale parameter ω/μ.

Background on the Nakagami Distribution

In communications theory, Nakagami distributions, Rician distributions, and Rayleigh distributions are used to model scattered signals that reach a receiver by multiple paths. Depending on the density of the scatter, the signal will display different fading characteristics. Rayleigh and Nakagami distributions are used to model dense scatters, while Rician distributions model fading with a stronger line-of-sight. Nakagami distributions can be reduced to Rayleigh distributions, but give more control over the extent of the fading.

Parameter estimation for the Nakagami Distribution

See `mle`, `dfittool`.

Negative Binomial Distribution

Definition of the Negative Binomial Distribution

When the r parameter is an integer, the negative binomial pdf is

$$y = f(x|r,p) = \binom{r+x-1}{x} p^r q^x I_{(0,1,\ldots)}(x)$$

where $q = 1 - p$. When r is not an integer, the binomial coefficient in the definition of the pdf is replaced by the equivalent expression

$$\frac{\Gamma(r+x)}{\Gamma(r)\Gamma(x+1)}$$

Background of the Negative Binomial Distribution

In its simplest form (when r is an integer), the negative binomial distribution models the number of failures x before a specified number of successes is reached in a series of independent, identical trials. Its parameters are the probability of success in a single trial, p, and the number of successes, r. A special case of the negative binomial distribution, when $r = 1$, is the geometric distribution, which models the number of failures before the first success.

More generally, r can take on non-integer values. This form of the negative binomial distribution has no interpretation in terms of repeated trials, but, like the Poisson distribution, it is useful in modeling count data. The negative binomial distribution is more general than the Poisson distribution because it has a variance that is greater than its mean, making it suitable for count data that do not meet the assumptions of the Poisson distribution. In the limit, as r increases to infinity, the negative binomial distribution approaches the Poisson distribution.

Parameter Estimation for the Negative Binomial Distribution

Suppose you are collecting data on the number of auto accidents on a busy highway, and would like to be able to model the number of accidents per day. Because these are count data, and because there are a very large number of cars and a small probability of an accident for any specific car, you might think to use the Poisson distribution. However, the probability of having an accident is likely to vary from day to day as the weather and amount of traffic change, and so the assumptions needed for the Poisson distribution are not met. In particular, the variance of this type of count data sometimes exceeds the mean by a large amount. The data below exhibit this effect: most days have few or no accidents, and a few days have a large number.

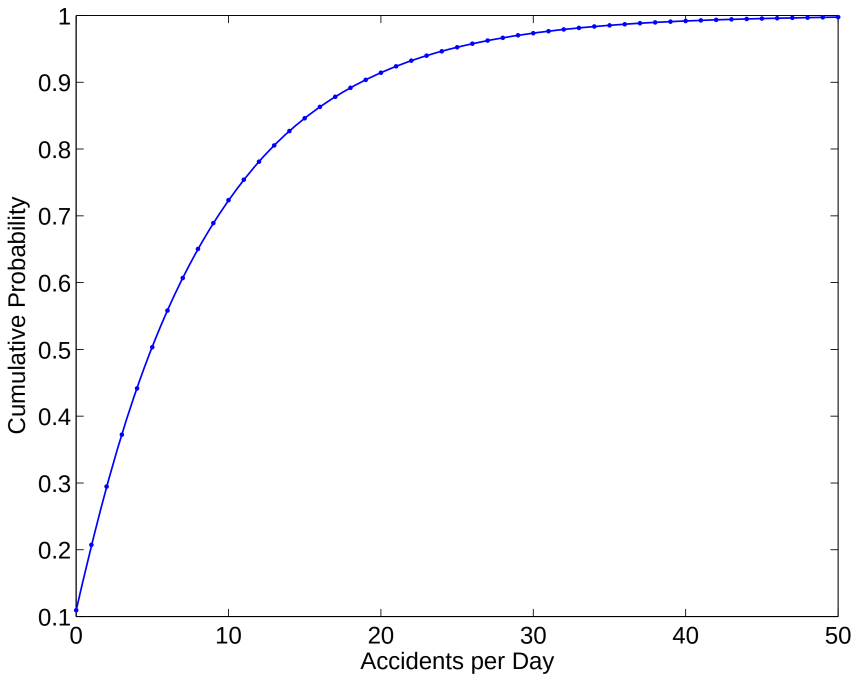
```
accident = [2  3  4  2  3  1  12  8  14  31  23  1  10  7  0];
mean(accident)
ans =

       8.0667

var(accident)
ans =

       79.352
```

The negative binomial distribution is more general than the Poisson, and is often suitable for count data when the Poisson is not. The function `nbinfit` returns the maximum likelihood estimates (MLEs) and confidence intervals for the parameters of the negative binomial distribution. Here are the results from fitting the `accident` data:

```
[phat,pci] = nbinfit(accident)
phat =
        1.006      0.11088
pci =
     0.015286   0.00037634
       1.9967      0.22138
```

It is difficult to give a physical interpretation in this case to the individual parameters. However, the estimated parameters can be used in a model for the number of daily accidents. For example, a plot of the estimated cumulative probability function shows that while there is an estimated 10% chance of no accidents on a given day, there is also about a 10% chance that there will be 20 or more accidents.

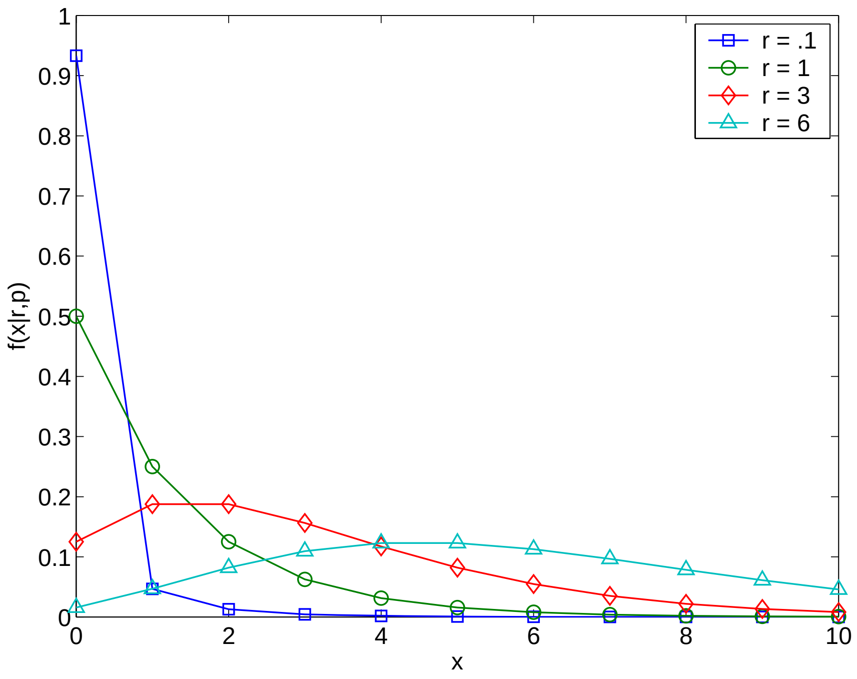
```
plot(0:50,nbincdf(0:50,phat(1),phat(2)),'.-');
xlabel('Accidents per Day')
ylabel('Cumulative Probability')
```



Example and Plot of the Negative Binomial Distribution

The negative binomial distribution can take on a variety of shapes ranging from very skewed to nearly symmetric. This example plots the probability function for different values of `r`, the desired number of successes: `.1`, `1`, `3`, `6`.

```
x = 0:10;
plot(x,nbinpdf(x,.1,.5),'s-', ...
     x,nbinpdf(x,1,.5),'o-', ...
     x,nbinpdf(x,3,.5),'d-', ...
     x,nbinpdf(x,6,.5),'^-');
legend({'r = .1' 'r = 1' 'r = 3' 'r = 6'})
xlabel('x')
ylabel('f(x|r,p')
```



r = .1
r = 1
r = 3
r = 6
f(x|r,p)
x
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
0
2
4
6
8
10

Noncentral Chi-Square Distribution

Definition of the Noncentral Chi-Square Distribution

There are many equivalent formulas for the noncentral chi-square distribution function. One formulation uses a modified Bessel function of the first kind. Another uses the generalized Laguerre polynomials. Statistics Toolbox computes the cumulative distribution function values using a weighted sum of χ^2 probabilities with the weights equal to the probabilities of a Poisson distribution. The Poisson parameter is one-half of the noncentrality parameter of the noncentral chi-square

$$F(x|\nu,\delta) = \sum_{j=0}^{\infty}\left(\frac{\left(\frac{1}{2}\delta\right)^j}{j!}e^{-\frac{\delta}{2}}\right)Pr[\chi^2_{\nu+2j} \le x]$$

where δ is the noncentrality parameter.

Background of the Noncentral Chi-Square Distribution

The χ^2 distribution is actually a simple special case of the noncentral chi-square distribution. One way to generate random numbers with a χ^2 distribution (with ν degrees of freedom) is to sum the squares of ν standard normal random numbers (mean equal to zero.)

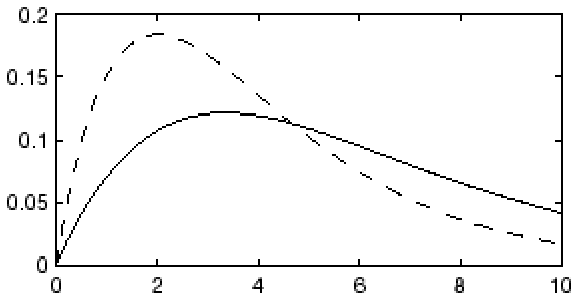
What if the normally distributed quantities have a mean other than zero? The sum of squares of these numbers yields the noncentral chi-square distribution. The noncentral chi-square distribution requires two parameters: the degrees of freedom and the noncentrality parameter. The noncentrality parameter is the sum of the squared means of the normally distributed quantities.

The noncentral chi-square has scientific application in thermodynamics and signal processing. The literature in these areas may refer to it as the Ricean or generalized Rayleigh distribution.

Example of the Noncentral Chi-Square Distribution

The following commands generate a plot of the noncentral chi-square pdf.

```
x = (0:0.1:10)';
p1 = ncx2pdf(x,4,2);
p = chi2pdf(x,4);
plot(x,p,'-',x,p1,'-')
```



Noncentral *F* Distribution

Definition of the Noncentral *F* Distribution

Similar to the noncentral χ^2 distribution, the toolbox calculates noncentral *F* distribution probabilities as a weighted sum of incomplete beta functions using Poisson probabilities as the weights.

$$F(x|\nu_1, \nu_2, \delta) = \sum_{j=0}^{\infty}\left(\frac{\left(\frac{1}{2}\delta\right)^j}{j!}e^{-\frac{\delta}{2}}\right)I\left(\frac{\nu_1 \cdot x}{\nu_2 + \nu_1 \cdot x}\middle|\frac{\nu_1}{2} + j, \frac{\nu_2}{2}\right)$$

$I(x|a,b)$ is the incomplete beta function with parameters a and b, and δ is the noncentrality parameter.

Background of the Noncentral *F* Distribution

As with the χ^2 distribution, the *F* distribution is a special case of the noncentral *F* distribution. The *F* distribution is the result of taking the ratio of χ^2 random variables each divided by its degrees of freedom.

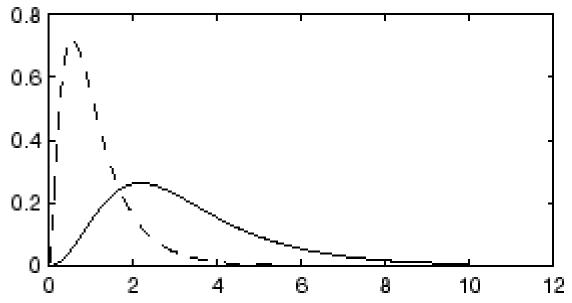
If the numerator of the ratio is a noncentral chi-square random variable divided by its degrees of freedom, the resulting distribution is the noncentral *F* distribution.

The main application of the noncentral *F* distribution is to calculate the power of a hypothesis test relative to a particular alternative.

Example and Plot of the Noncentral *F* Distribution

The following commands generate a plot of the noncentral *F* pdf.

```
x = (0.01:0.1:10.01)';
p1 = ncfpdf(x,5,20,10);
p  = fpdf(x,5,20);
plot(x,p,'-',x,p1,'-')
```



0.8
0.6
0.4
0.2
0
0
2
4
6
8
10
12

Noncentral *t* Distribution

Definition of the Noncentral *t* Distribution

The most general representation of the noncentral t distribution is quite complicated. Johnson and Kotz [27] give a formula for the probability that a noncentral t variate falls in the range $[-t, t]$.

$$Pr((-t) < x < t \,|\, (\nu, \delta)) = \sum_{j=0}^{\infty} \left(\frac{\left(\frac{1}{2}\delta^2\right)^j}{j!} e^{-\frac{\delta^2}{2}} \right) I\left(\frac{x^2}{\nu + x^2} \,\middle|\, \frac{1}{2} + j, \frac{\nu}{2} \right)$$

$I(x\,|\,a,b)$ is the incomplete beta function with parameters a and b, δ is the noncentrality parameter, and ν is the number of degrees of freedom.

Background of the Noncentral *t* Distribution

The noncentral t distribution is a generalization of Student's t distribution.

Student's t distribution with $n - 1$ degrees of freedom models the t-statistic

$$t = \frac{\bar{x} - \mu}{s / \sqrt{n}}$$

where $\bar{x}$ is the sample mean and s is the sample standard deviation of a random sample of size n from a normal population with mean μ. If the population mean is actually μ_0, then the t-statistic has a noncentral t distribution with noncentrality parameter

$$\delta = \frac{\mu_0 - \mu}{\sigma / \sqrt{n}}$$

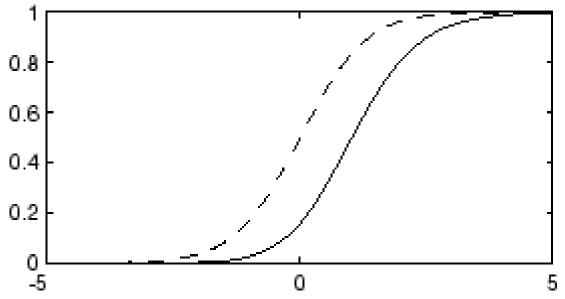
The noncentrality parameter is the normalized difference between μ_0 and μ.

The noncentral t distribution gives the probability that a t test will correctly reject a false null hypothesis of mean μ when the population mean is actually μ_0; that is, it gives the power of the t test. The power increases as the difference $\mu_0 - \mu$ increases, and also as the sample size n increases.

Example and Plot of the Noncentral *t* Distribution

The following commands generate a plot of the noncentral *t* pdf.

```
x = (-5:0.1:5)';
p1 = nctcdf(x,10,1);
p = tcdf(x,10);
plot(x,p,'-',x,p1,'-')
```



Nonparametric Distributions

See the discussion of `ksdensity` in “Probability Density Estimation” on page 3-10.

Normal Distribution

Definition of the Normal Distribution

The normal pdf is

$$y = f(x|\mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(x-\mu)^2}{2\sigma^2}}$$

Background of the Normal Distribution

The normal distribution is a two-parameter family of curves. The first parameter, μ, is the mean. The second, σ, is the standard deviation. The standard normal distribution (written $\Phi(x)$) sets μ to 0 and σ to 1.

$\Phi(x)$ is functionally related to the error function, *erf*.

$$erf(x) = 2\Phi(x\sqrt{2}) - 1$$

The first use of the normal distribution was as a continuous approximation to the binomial.

The usual justification for using the normal distribution for modeling is the Central Limit Theorem, which states (roughly) that the sum of independent samples from any distribution with finite mean and variance converges to the normal distribution as the sample size goes to infinity.

Parameter Estimation for the Normal Distribution

To use statistical parameters such as mean and standard deviation reliably, you need to have a good estimator for them. The maximum likelihood estimates (MLEs) provide one such estimator. However, an MLE might be biased, which means that its expected value of the parameter might not equal the parameter being estimated. For example, an MLE is biased for estimating the variance of a normal distribution. An unbiased estimator that is commonly used to estimate the parameters of the normal distribution is the *minimum variance unbiased estimator* (*MVUE*). The MVUE has the minimum variance of all unbiased estimators of a parameter.

The MVUEs of parameters μ and σ^2 for the normal distribution are the sample mean and variance. The sample mean is also the MLE for μ. The following are two common formulas for the variance.

$$1)\quad s^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

$$2)\quad s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

where

$$\bar{x} = \sum_{i=1}^{n}\frac{x_i}{n}$$

Equation 1 is the maximum likelihood estimator for σ^2, and equation 2 is the MVUE.

As an example, suppose you want to estimate the mean, μ, and the variance, σ^2, of the heights of all fourth grade children in the United States. The function `normfit` returns the MVUE for μ, the square root of the MVUE for σ^2, and confidence intervals for μ and σ^2. Here is a playful example modeling the heights in inches of a randomly chosen fourth grade class.

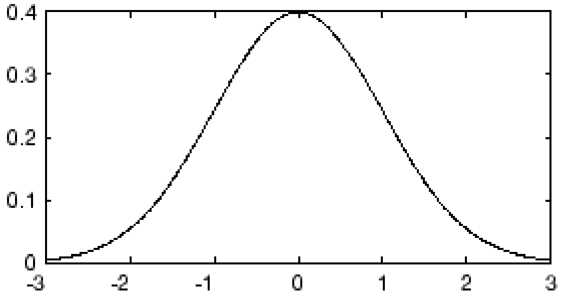
```
height = normrnd(50,2,30,1);            % Simulate heights.
[mu,s,muci,sci] = normfit(height)

mu =
   50.2025

s =
    1.7946

muci =
   49.5210
   50.8841

sci =
        1.4292
        2.4125
```

Note that `s^2` is the MVUE of the variance.

```
  s^2

  ans =
        3.2206
```

Example and Plot of the Normal Distribution

The plot shows the bell curve of the standard normal pdf, with $\mu = 0$ and $\sigma = 1$.



Pearson System of Distributions

See “Pearson and Johnson Systems of Distributions” on page 5-169.

Poisson Distribution

Definition of the Poisson Distribution

The Poisson pdf is

$$y = f(x|\lambda) = \frac{\lambda^x}{x!}e^{-\lambda}I_{(0,1,\ldots)}(x)$$

Background of the Poisson Distribution

The Poisson distribution is appropriate for applications that involve counting the number of times a random event occurs in a given amount of time, distance, area, etc. Sample applications that involve Poisson distributions include the number of Geiger counter clicks per second, the number of people walking into a store in an hour, and the number of flaws per 1000 feet of video tape.

The Poisson distribution is a one-parameter discrete distribution that takes nonnegative integer values. The parameter, λ, is both the mean and the variance of the distribution. Thus, as the size of the numbers in a particular sample of Poisson random numbers gets larger, so does the variability of the numbers.

As Poisson showed, the Poisson distribution is the limiting case of a binomial distribution where N approaches infinity and p goes to zero while $Np = \lambda$.

The Poisson and exponential distributions are related. If the number of counts follows the Poisson distribution, then the interval between individual counts follows the exponential distribution.

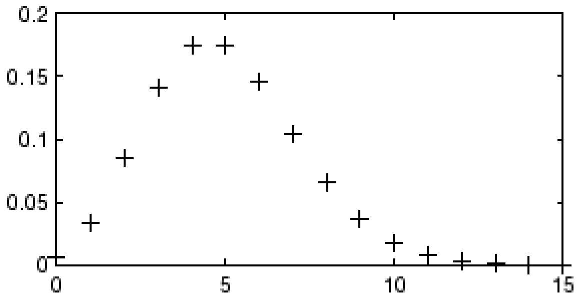
Parameter Estimation for the Poisson Distribution

The MLE and the MVUE of the Poisson parameter, λ, is the sample mean. The sum of independent Poisson random variables is also Poisson distributed with the parameter equal to the sum of the individual parameters. Statistics Toolbox makes use of this fact to calculate confidence intervals λ. As λ gets large the Poisson distribution can be approximated by a normal distribution with $\mu = \lambda$ and $\sigma^2 = \lambda$. Statistics Toolbox uses this approximation for calculating confidence intervals for values of λ greater than 100.

Example and Plot of the Poisson Distribution

The plot shows the probability for each nonnegative integer when λ = 5.

```
x = 0:15;
y = poisspdf(x,5);
plot(x,y,'+')
```



Rayleigh Distribution

Definition of the Rayleigh Distribution

The Rayleigh pdf is

$$y = f(x \mid b) = \frac{x}{b^2} e^{\left(\frac{-x^2}{2b^2}\right)}$$

Background of the Rayleigh Distribution

The Rayleigh distribution is a special case of the Weibull distribution. If A and B are the parameters of the Weibull distribution, then the Rayleigh distribution with parameter b is equivalent to the Weibull distribution with parameters $A = \sqrt{2}b$ and $B = 2$.

If the component velocities of a particle in the x and y directions are two independent normal random variables with zero means and equal variances, then the distance the particle travels per unit time is distributed Rayleigh.

In communications theory, Nakagami distributions, Rician distributions, and Rayleigh distributions are used to model scattered signals that reach a receiver by multiple paths. Depending on the density of the scatter, the signal will display different fading characteristics. Rayleigh and Nakagami distributions are used to model dense scatters, while Rician distributions model fading with a stronger line-of-sight. Nakagami distributions can be reduced to Rayleigh distributions, but give more control over the extent of the fading.

Parameter Estimation for the Rayleigh Distribution

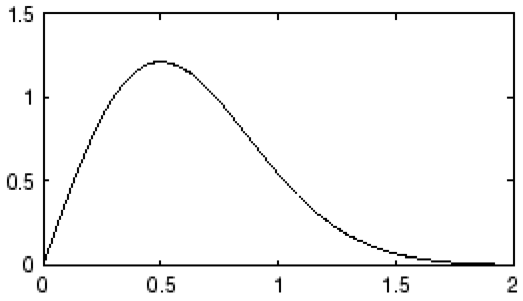
The `raylfit` function returns the MLE of the Rayleigh parameter. This estimate is

$$b = \sqrt{\frac{1}{2n}\sum_{i=1}^{n} x_i^2}$$

Example and Plot of the Rayleigh Distribution

The following commands generate a plot of the Rayleigh pdf.

```
x = [0:0.01:2];
p = raylpdf(x,0.5);
plot(x,p)
```



Rician Distribution

Definition of the Rician Distribution

The Rician distribution has the density function

$$I_0\left(\frac{xs}{\sigma^2}\right)\frac{x}{\sigma^2}e^{-\left(\frac{x^2+s^2}{2\sigma^2}\right)}$$

with noncentrality parameter $s \geq 0$ and scale parameter $\sigma > 0$, for $x > 0$. I_0 is the zero-order modified Bessel function of the first kind. If x has a Rician distribution with parameters s and σ, then $(x/\sigma)^2$ has a noncentral chi-square distribution with two degrees of freedom and noncentrality parameter $(s/\sigma)^2$.

Background on the Rician Distribution

In communications theory, Nakagami distributions, Rician distributions, and Rayleigh distributions are used to model scattered signals that reach a receiver by multiple paths. Depending on the density of the scatter, the signal will display different fading characteristics. Rayleigh and Nakagami distributions are used to model dense scatters, while Rician distributions model fading with a stronger line-of-sight. Nakagami distributions can be reduced to Rayleigh distributions, but give more control over the extent of the fading.

Parameter estimation for the Rician Distribution

See `mle`, `dfittool`.

Student's *t* Distribution

Definition of Student's *t* Distribution

Student's t pdf is

$$y = f(x|\nu) = \frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\Gamma\left(\frac{\nu}{2}\right)} \frac{1}{\sqrt{\nu\pi}} \frac{1}{\left(1+\frac{x^2}{\nu}\right)^{\frac{\nu+1}{2}}}$$

where $\Gamma(\cdot)$ is the Gamma function.

Background of Student's *t* Distribution

The t distribution is a family of curves depending on a single parameter ν (the degrees of freedom). As ν goes to infinity, the t distribution approaches the standard normal distribution.

W. S. Gossett discovered the distribution through his work at the Guinness brewery. At the time, Guinness did not allow its staff to publish, so Gossett used the pseudonym "Student."

If x is a random sample of size n from a normal distribution with mean μ, then the statistic

$$t = \frac{\bar{x} - \mu}{s / \sqrt{n}}$$

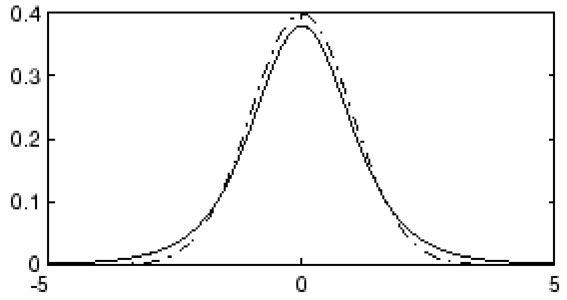
where $\bar{x}$ is the sample mean and s is the sample standard deviation, has Student's t distribution with $n - 1$ degrees of freedom.

Example and Plot of Student's *t* Distribution

The plot compares the t distribution with $\nu = 5$ (solid line) to the shorter tailed, standard normal distribution (dashed line).

```
x = -5:0.1:5;
y = tpdf(x,5);
```

```
z = normpdf(x,0,1);
plot(x,y,'-',x,z,'-.')
```



t Location-Scale Distribution

Definition of the *t* Location-Scale Distribution

The t location-scale distribution has the density function

$$\frac{\Gamma\left(\frac{\upsilon+1}{2}\right)}{\sigma\sqrt{\upsilon\pi}\,\Gamma\left(\frac{\upsilon}{2}\right)}\left[\frac{\upsilon+\left(\frac{x-\mu}{\sigma}\right)^2}{\upsilon}\right]^{-\left(\frac{\upsilon+1}{2}\right)}$$

with location parameter μ, scale parameter $\sigma > 0$, and shape parameter $\nu > 0$. If x has a t location-scale distribution, with parameters μ, σ, and ν, then

$$\frac{x-\mu}{\sigma}$$

has a Student's t distribution with ν degrees of freedom.

Background of the *t* Location-Scale Distribution

The t location-scale distribution is useful for modeling data distributions with heavier tails (more prone to outliers) than the normal distribution. It approaches the normal distribution as ν approaches infinity, and smaller values of ν yield heavier tails.

Parameter estimation for the *t* Location-Scale Distribution

See `mle`, `dfittool`.

Uniform Distribution (Continuous)

Definition of the Continuous Uniform Distribution

The uniform cdf is

$$p = F(x|a,b) = \frac{x-a}{b-a} I_{[a,\,b]}(x)$$

Background of the Continuous Uniform Distribution

The uniform distribution (also called rectangular) has a constant pdf between its two parameters a (the minimum) and b (the maximum). The standard uniform distribution (a = 0 and b = 1) is a special case of the beta distribution, obtained by setting both of its parameters to 1.

The uniform distribution is appropriate for representing the distribution of round-off errors in values tabulated to a particular number of decimal places.

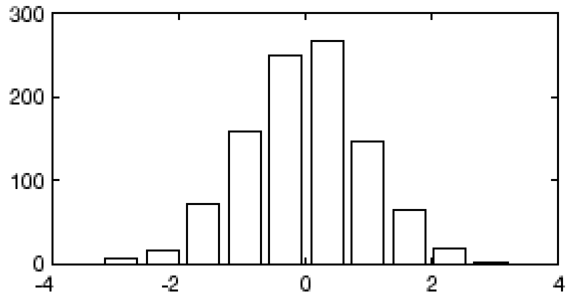
Parameter Estimation for the Continuous Uniform Distribution

The sample minimum and maximum are the MLEs of a and b respectively.

Example and Plot of the Continuous Uniform Distribution

The example illustrates the inversion method for generating normal random numbers using `rand` and `norminv`. Note that the MATLAB function, `randn`, does not use inversion since it is not efficient for this case.

```
u = rand(1000,1);
x = norminv(u,0,1);
hist(x)
```



Uniform Distribution (Discrete)

Definition of the Discrete Uniform Distribution

The discrete uniform pdf is

$$y = f(x|N) = \frac{1}{N} I_{(1,\ldots,N)}(x)$$

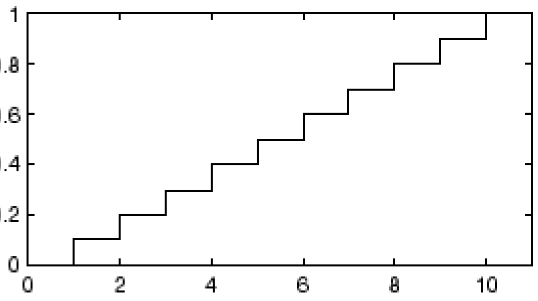
Background of the Discrete Uniform Distribution

The discrete uniform distribution is a simple distribution that puts equal weight on the integers from one to N.

Example and Plot of the Discrete Uniform Distribution

As for all discrete distributions, the cdf is a step function. The plot shows the discrete uniform cdf for $N = 10$.

```
x = 0:10;
y = unidcdf(x,10);
stairs(x,y)
set(gca,'Xlim',[0 11])
```



Pick a random sample of 10 from a list of 553 items:

```
numbers = unidrnd(553,1,10)
numbers =
   293   372     5   213    37   231   380   326   515   468
```

Weibull Distribution

Definition of the Weibull Distribution

The Weibull pdf is

$$y = f(x|a,b) = ba^{-b}x^{b-1}e^{-\left(\frac{x}{a}\right)^b} I_{(0,\infty)}(x)$$

Background of the Weibull Distribution

Waloddi Weibull offered the distribution that bears his name as an appropriate analytical tool for modeling the breaking strength of materials. Current usage also includes reliability and lifetime modeling. The Weibull distribution is more flexible than the exponential for these purposes.

To see why, consider the hazard rate function (instantaneous failure rate). If $f(t)$ and $F(t)$ are the pdf and cdf of a distribution, then the hazard rate is

$$h(t) = \frac{f(t)}{1-F(t)}$$

Substituting the pdf and cdf of the exponential distribution for $f(t)$ and $F(t)$ above yields a constant. The example below shows that the hazard rate for the Weibull distribution can vary.

Parameter Estimation for the Weibull Distribution

Suppose you want to model the tensile strength of a thin filament using the Weibull distribution. The function `wblfit` gives maximum likelihood estimates and confidence intervals for the Weibull parameters.

```
strength = wblrnd(0.5,2,100,1);       % Simulated strengths.
[p,ci] = wblfit(strength)

p =
0.4715    1.9811


ci =
        0.4248    1.7067
        0.5233    2.2996
```

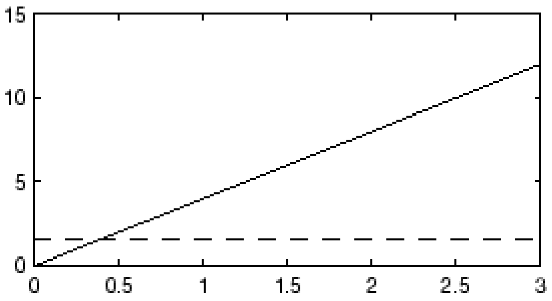
The default 95% confidence interval for each parameter contains the true value.

Example and Plot of the Weibull Distribution

The exponential distribution has a constant hazard function, which is not generally the case for the Weibull distribution.

The plot shows the hazard functions for exponential (dashed line) and Weibull (solid line) distributions having the same mean life. The Weibull hazard rate here increases with age (a reasonable assumption).

```
  t = 0:0.1:4.5;
  h1 = exppdf(t,0.6267) ./ (1-expcdf(t,0.6267));
  h2 = wblpdf(t,2,2) ./ (1-wblcdf(t,2,2));
  plot(t,h1,'-',t,h2,'-')
```



Wishart Distribution

Definition of the Wishart Distribution

The probability density function of the d-dimensional Wishart distribution is given by

$$y = f(X, \Sigma, \nu) = \frac{X^{((\nu\text{-}d\text{-}1)/2)} \cdot e^{\left(-\mathrm{trace}\left(\Sigma^{-1}X\right)/2\right)}}{2^{(\nu d)/2} \cdot \pi^{(d(d\text{-}1))/2} \cdot \Sigma^{\nu/2} \cdot \Gamma(\nu/2) \cdot \ldots \cdot \Gamma(\nu\text{-}(d\text{-}1))/2}$$

where X and Σ are d-by-d symmetric positive definite matrices, and ν is a scalar greater than $d - 1$. While it is possible to define the Wishart for singular Σ or for integer $\nu \leq d - 1$, the density cannot be written as above. Statistics Toolbox only supports random matrix generation for the Wishart, including the singular cases.

Background of the Wishart Distribution

The Wishart distribution is a generalization of the univariate chi-square distribution to two or more variables. It is a distribution for symmetric positive semidefinite matrices, typically covariance matrices, the diagonal elements of which are each chi-square random variables. In the same way as the chi-square distribution can be constructed by summing the squares of independent, identically distributed, zero-mean univariate normal random variables, the Wishart distribution can be constructed by summing the inner products of independent, identically distributed, zero-mean multivariate normal random vectors.

The Wishart distribution is parameterized with a symmetric, positive semidefinite matrix, Σ, and a positive scalar degrees of freedom parameter, ν. ν is analogous to the degrees of freedom parameter of a univariate chi-square distribution, and $\Sigma\nu$ is the mean of the distribution.

The Wishart distribution is often used as a model for the distribution of the sample covariance matrix for multivariate normal random data, after scaling by the sample size.

Example of the Wishart Distribution

If x is a bivariate normal random vector with mean zero and covariance matrix

$$\Sigma = \begin{pmatrix} 1 & .5 \\ .5 & 2 \end{pmatrix}$$

then you can use the Wishart distribution to generate a sample covariance matrix without explicitly generating x itself. Notice how the sampling variability is quite large when the degrees of freedom is small.

```
mu = [0 0];
Sigma = [1 .5; .5 2];
n = 10; S1 = wishrnd(Sigma,n)/(n-1)

S1 =
       1.7959      0.64107
      0.64107       1.5496

n = 1000; S2 = wishrnd(Sigma,n)/(n-1)

S2 =
       0.9842      0.50158
      0.50158       2.1682
```

Distribution Functions

For each distribution supported by Statistics Toolbox, a selection of the following types of distribution functions is available for statistical programming. This section gives a general overview of the use of each type of function, independent of the particular distribution. For specific functions available for specific distributions, see "Supported Distributions" on page 5-3.

Probability Density Functions

Probability density functions (pdfs) for supported distributions in Statistics Toolbox all end with `pdf`, as in `binopdf` or `exppdf`. Specific function names for specific distributions can be found in “Supported Distributions” on page 5-3.

Each function represents a parametric family of distributions. Input arguments are arrays of outcomes followed by a list of parameter values specifying a particular member of the distribution family.

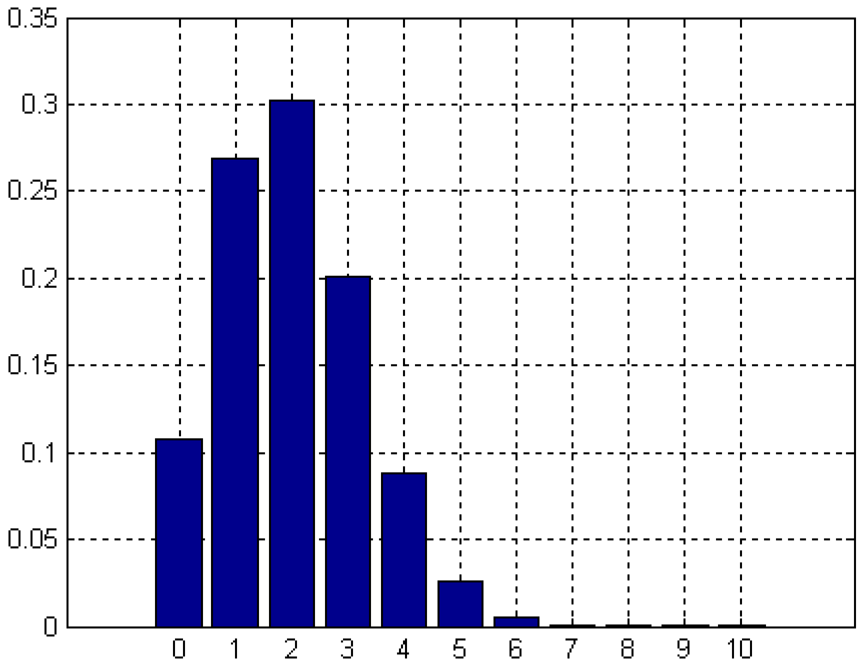
For discrete distributions, the pdf assigns a probability to each outcome. In this context, the pdf is often called a *probability mass function* (*pmf*).

For example, the discrete binomial pdf

$$f(k) = \binom{n}{k} p^k (1-p)^{n-k}$$

assigns probability to the event of k successes in n trials of a Bernoulli process (such as coin flipping) with probability p of success at each trial. Each of the integers k = 0, 1, 2, ..., n is assigned a positive probability, with the sum of the probabilities equal to 1. The probabilities are computed with the `binopdf` function:

```
p = 0.2; % Probability of success for each trial
n = 10; % Number of trials
k = 0:n; % Outcomes
m = binopdf(k,n,p); % Probability mass vector
bar(k,m) % Visualize the probability distribution
grid on
```



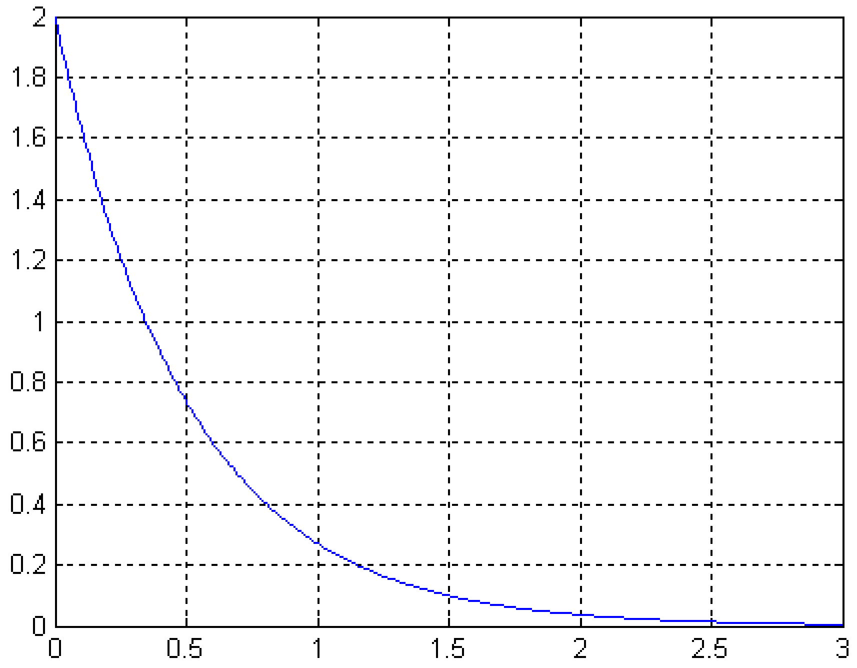
For continuous distributions, the pdf assigns a probability *density* to each outcome. The probability of any single outcome is zero. The pdf must be integrated over a set of outcomes to compute the probability that an outcome falls within that set. The integral over the entire set of outcomes is 1.

For example, the continuous exponential pdf

$$f(t) = \lambda e^{-\lambda t}$$

is used to model the probability that a process with constant failure rate λ will have a failure within time t . Each time $t > 0$ is assigned a positive probability density. Densities are computed with the `exppdf` function:

```
lambda = 2; % Failure rate
t = 0:0.01:3; % Outcomes
f = exppdf(t,1/lambda); Probability density vector
plot(t,f)% Visualize the probability distribution
grid on
```



Probabilities for continuous pdfs can be computed with the `quad` function. In the example above, the probability of failure in the time interval [0, 1] is computed as follows:

```
f_lambda = @(t)exppdf(t,1/lambda); % Pdf with fixed lambda
P = quad(f_lambda,0,1) % Integrate from 0 to 1
P =
    0.8647
```

Alternatively, the cumulative distribution function (cdf) for the exponential function, `expcdf`, can be used:

```
P = expcdf(1,1/lambda) % Cumulative probability from 0 to 1
P =
    0.8647
```

Cumulative Distribution Functions

Cumulative distribution functions (cdfs) for supported distributions in Statistics Toolbox all end with `cdf`, as in `binocdf` or `expcdf`. Specific function names for specific distributions can be found in "Supported Distributions" on page 5-3.

Each function represents a parametric family of distributions. Input arguments are arrays of outcomes followed by a list of parameter values specifying a particular member of the distribution family.

For discrete distributions, the cdf F is related to the pdf f by

$$F(x) = \sum_{y \le x} f(y)$$

For continuous distributions, the cdf F is related to the pdf f by

$$F(x) = \int_{-\infty}^{x} f(y)dy$$

Cdfs are used to compute probabilities of events. In particular, if F is a cdf and x and y are outcomes, then

- $P(y \le x) = F(x)$
- $P(y \ge x) = 1 - F(x)$
- $P(x_1 \le y \le x_2) = F(x_2) - F(x_1)$

For example, the t-statistic

$$t = \frac{\bar{x} - \mu}{s / \sqrt{n}}$$

follows a Student's t distribution with $n - 1$ degrees of freedom when computed from repeated random samples from a normal population with mean μ. Here $\bar{x}$ is the sample mean, s is the sample standard deviation, and n is the sample size. The probability of observing a *t*-statistic greater than or equal to the value computed from a sample can be found with the `tcdf` function:

```
mu = 1; % Population mean
sigma = 2; % Population standard deviation
n = 100; % Sample size
x = normrnd(mu,sigma,n,1); % Random sample from population
xbar = mean(x); % Sample mean
s = std(x); % Sample standard deviation
t = (xbar-mu)/(s/sqrt(n)) % t-statistic
t =
    0.2489
p = 1-tcdf(t,n-1) % Probability of larger t-statistic
p =
    0.4020
```

This probability is the same as the *p*-value returned by a *t*-test of the null hypothesis that the sample comes from a normal population with mean μ:

```
[h,ptest] = ttest(x,mu,0.05,'right')
h =
     0
ptest =
    0.4020
```

Inverse Cumulative Distribution Functions

Inverse cumulative distribution functions for supported distributions in Statistics Toolbox all end with `inv`, as in `binoinv` or `expinv`. Specific function names for specific distributions can be found in "Supported Distributions" on page 5-3.

Each function represents a parametric family of distributions. Input arguments are arrays of cumulative probabilities between 0 and 1 followed by a list of parameter values specifying a particular member of the distribution family.

For continuous distributions, the inverse cdf returns the unique outcome whose cdf value is the input cumulative probability.

For example, the `expinv` function can be used to compute inverses of exponential cumulative probabilities:

```
x = 0.5:0.2:1.5 % Outcomes
x =
    0.5000  0.7000  0.9000  1.1000  1.3000  1.5000
p = expcdf(x,1) % Cumulative probabilities
p =
    0.3935  0.5034  0.5934  0.6671  0.7275  0.7769
expinv(p,1) % Return original outcomes
ans =
    0.5000  0.7000  0.9000  1.1000  1.3000  1.5000
```

For discrete distributions, there may be no outcome whose cdf value is the input cumulative probability. In these cases, the inverse cdf returns the first outcome whose cdf value equals or exceeds the input cumulative probability.

For example, the `binoinv` function can be used to compute inverses of binomial cumulative probabilities:

```
x = 0.5:0.2:1.5 % Outcomes
x =
    0.5000  0.7000  0.9000  1.1000  1.3000  1.5000
p = binocdf(x,10,0.2) % Cumulative probabilities
p =
    0.1074  0.1074  0.1074  0.3758  0.3758  0.3758
```

```
>> binoinv(p,10,0.2) % Return binomial outcomes
ans =
     0  0  0  1  1  1
```

The inverse cdf is useful in hypothesis testing, where critical outcomes of a test statistic are computed from cumulative significance probabilities. For example, `norminv` can be used to compute a 95% confidence interval under the assumption of normal variability:

```
p = [0.025 0.975]; % Interval containing 95% of [0,1]
x = norminv(p,0,1) % Assume standard normal variability
x =
   -1.9600  1.9600 % 95% confidence interval

n = 20; % Sample size
y = normrnd(8,1,n,1); % Random sample (assume mean is unknown)
ybar = mean(y);
ci = ybar + (1/sqrt(n))*x % Confidence interval for mean
ci =
    7.6779  8.5544
```

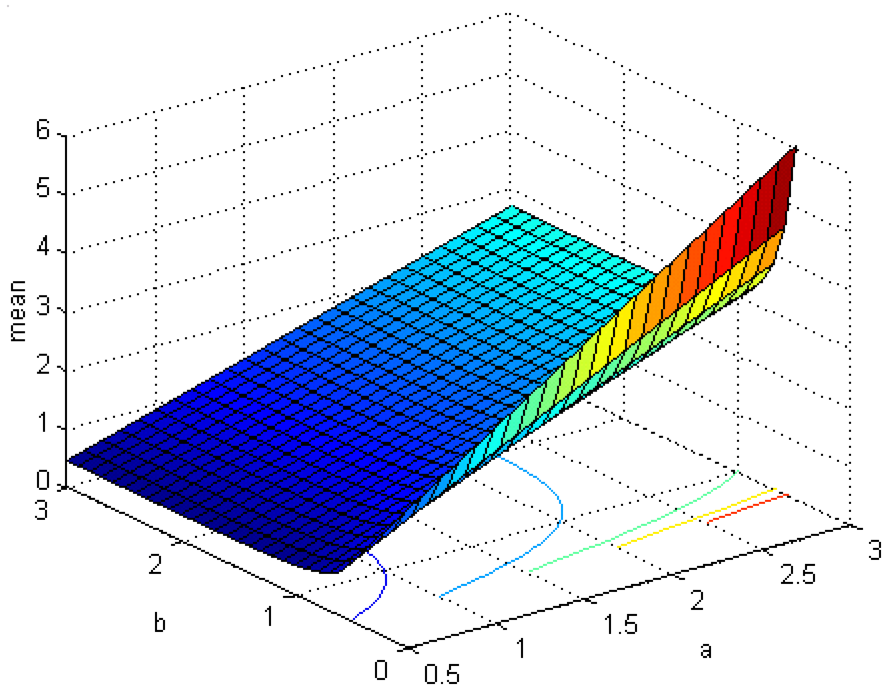
Distribution Statistics Functions

Distribution statistics functions for supported distributions in Statistics Toolbox all end with `stat`, as in `binostat` or `expstat`. Specific function names for specific distributions can be found in "Supported Distributions" on page 5-3.

Each function represents a parametric family of distributions. Input arguments are lists of parameter values specifying a particular member of the distribution family. Functions return the mean and variance of the distribution, as a function of the parameters.

For example, the `wblstat` function can be used to visualize the mean of the Weibull distribution as a function of its two distribution parameters:

```
a = 0.5:0.1:3;
b = 0.5:0.1:3;
[A,B] = meshgrid(a,b);
M = wblstat(A,B);
surfc(A,B,M)
```



mean
6
5
4
3
2
1
0
3
2
b
1
0
0.5
1
1.5
2
a
2.5
3

Distribution Fitting Functions

Fitting Supported Distributions

Distribution fitting functions for supported distributions in Statistics Toolbox all end with `fit`, as in `binofit` or `expfit`. Specific function names for specific distributions can be found in "Supported Distributions" on page 5-3.

Each function represents a parametric family of distributions. Input arguments are arrays of data, presumed to be samples from some member of the selected distribution family. Functions return maximum likelihood estimates (MLEs) of distribution parameters, that is, parameters for the distribution family member with the maximum likelihood of producing the data as a random sample.

The Statistics Toolbox function `mle` is a convenient front end to the individual distribution fitting functions, and more. The function computes MLEs for distributions beyond those for which Statistics Toolbox provides specific pdf functions.

For some pdfs, MLEs can be given in closed form and computed directly. For other pdfs, a search for the maximum likelihood must be employed. The search can be controlled with an `options` input argument, created using the `statset` function. For efficient searches, it is important to choose a reasonable distribution model and set appropriate convergence tolerances.

MLEs can be heavily biased, especially for small samples. As sample size increases, however, MLEs become unbiased minimum variance estimators with approximate normal distributions. This is used to compute confidence bounds for the estimates.

For example, consider the following distribution of means from repeated random samples of an exponential distribution:

```
mu = 1; % Population parameter
n = 1e3; % Sample size
ns = 1e4; % Number of samples
```

```
samples = exprnd(mu,n,ns); % Population samples
means = mean(samples); % Sample means
```

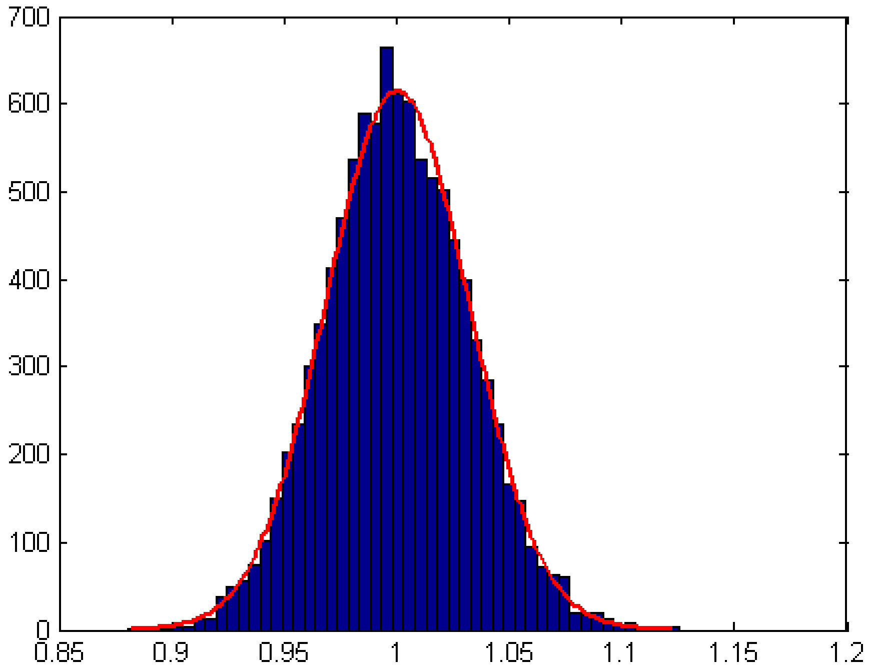
The Central Limit Theorem says that the means will be approximately normally distributed, regardless of the distribution of the data in the samples. The `normfit` function can be used to find the normal distribution that best fits the means:

```
[muhat,sigmahat,muci,sigmaci] = normfit(means)
muhat =
    1.0003
sigmahat =
    0.0319
muci =
    0.9997
    1.0010
sigmaci =
    0.0314
    0.0323
```

The function returns MLEs for the mean and standard deviation and their 95% confidence intervals.

To visualize the distribution of sample means together with the fitted normal distribution, you must scale the fitted pdf, with area = 1, to the area of the histogram being used to display the means:

```
numbins = 50;
hist(means,numbins)
hold on
[bincounts,binpositions] = hist(means,numbins);
binwidth = binpositions(2) - binpositions(1);
histarea = binwidth*sum(bincounts);
x = binpositions(1):0.001:binpositions(end);
y = normpdf(x,muhat,sigmahat);
plot(x,histarea*y,'r','LineWidth',2)
```



Fitting Piecewise Distributions

The parametric methods discussed in “Fitting Supported Distributions” on page 5-104 fit data samples with smooth distributions that have a relatively low-dimensional set of parameters controlling their shape. These methods work well in many cases, but there is no guarantee that a given sample will be described accurately by any of the supported distributions in Statistics Toolbox.

The empirical distributions computed by `ecdf` and discussed in “Empirical Cumulative Distribution Function” on page 3-15 assign equal probability to each observation in a sample, providing an exact match of the sample distribution. However, the distributions are not smooth, especially in the tails where data may be sparse.

The `paretotails` function fits a distribution by piecing together the empirical distribution in the center of the sample with smooth Pareto distributions in the tails.

The output of `paretotails` is an object with associated methods to evaluate the cdf, inverse cdf, and other functions of the fitted distribution. The `paretotails` class is a subclass of the `piecewisedistribution` class, and many of its methods are derived from that class. Never call the constructor for the `piecewisedistribution` class directly. Instead, use the subclass constructor `paretotails`.

The tables below list methods for the `piecewisedistribution` and `paretotails` classes. For full descriptions of individual methods, type one of the following, depending on the class:

```
help piecewisedistribution/methodname
help paretotails/methodname
```

Methods with supporting reference pages, including examples, are linked from the tables. An example follows the tables.

The following table lists methods available for all `piecewisedistribution` objects.

Piecewise Distribution Method	Description
`boundary`	Boundary points of piecewise distribution segments.
`cdf (piecewisedistribution)`	Cumulative distribution function for piecewise distribution.
`disp`	Display `piecewisedistribution` object, without printing object name.
`display`	Display `piecewisedistribution` object, printing object name. This method is invoked when the name of a `piecewisedistribution` object is entered at the command prompt.
`icdf (piecewisedistribution)`	Inverse cumulative distribution function for piecewise distribution.
`nsegments`	Number of segments of piecewise distribution.

Piecewise Distribution Method	Description
`pdf (piecewisedistribution)`	Probability density function for piecewise distribution.
`random (piecewisedistribution)`	Random numbers from piecewise distribution.
`segment`	Segment of piecewise distribution containing input values.

The following table lists additional methods for `paretotails` objects.

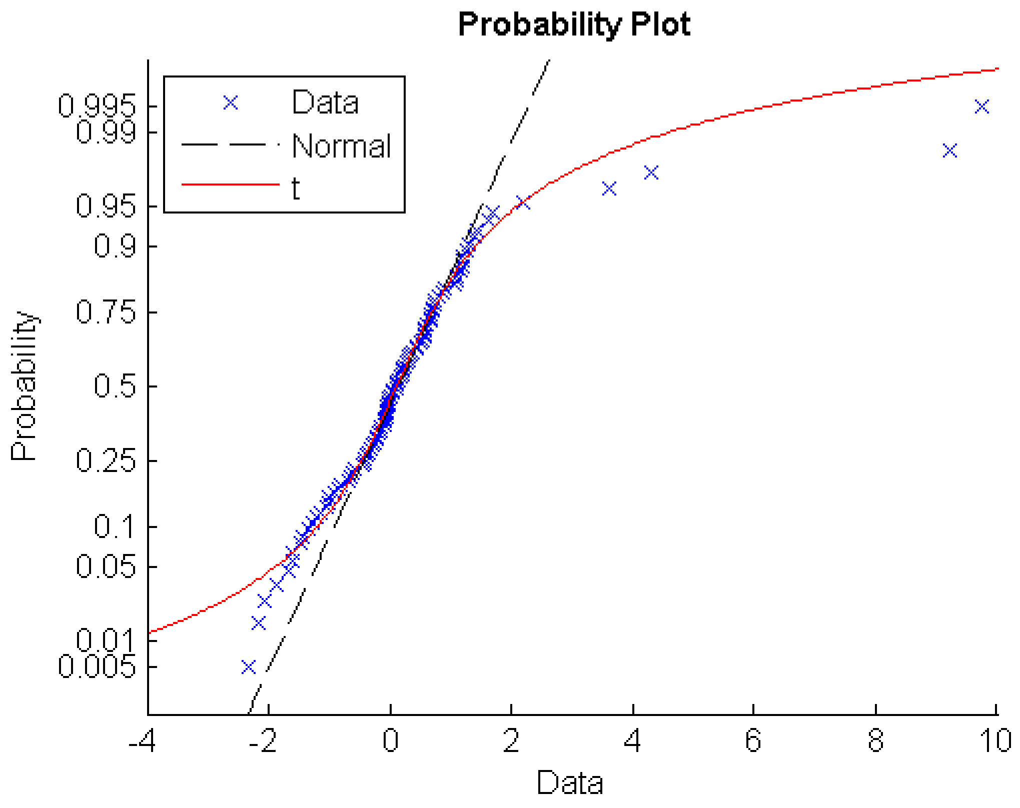
Pareto Tails Method	Description
`lowerparams`	Parameters of generalized Pareto distribution lower tail.
`paretotails`	Construct Pareto tails object.
`subsref`	Subscripted reference for `paretotails` object. This method is invoked by parenthesis indexing, as demonstrated in the example below.
`upperparams`	Parameters of generalized Pareto distribution upper tail.

As an example, consider the following data, with about 20% outliers:

```
left_tail = -exprnd(1,10,1);
right_tail = exprnd(5,10,1);
center = randn(80,1);
data = [left_tail;center;right_tail];
```

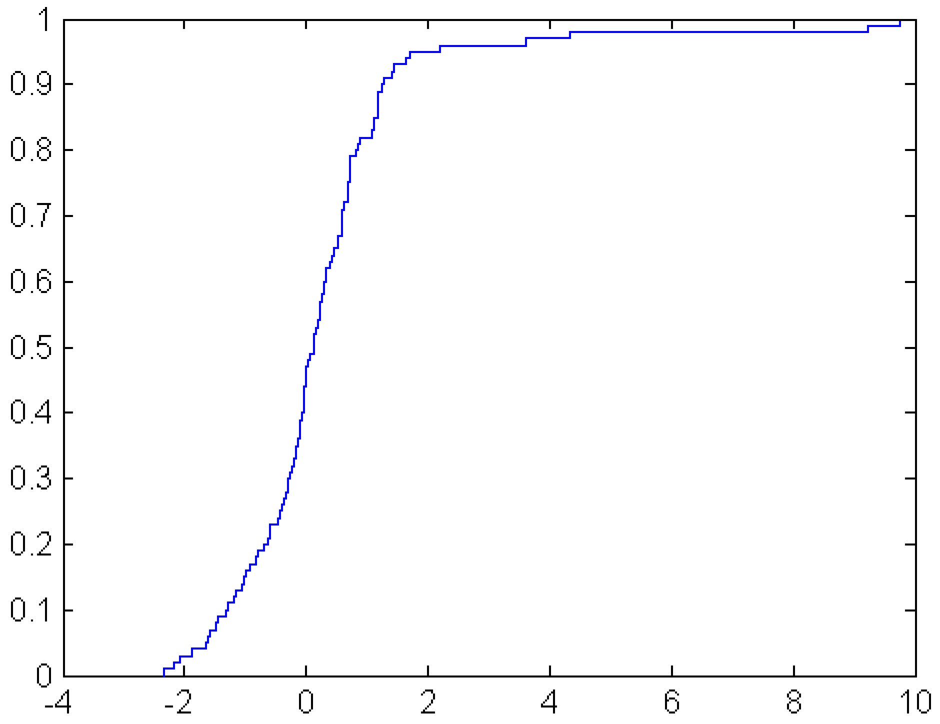
Neither a normal distribution nor a *t* distribution fits the tails very well:

```
probplot(data);
p = mle(data,'dist','tlo');
h = probplot(gca,@(data,mu,sig,df))
cdf('tlocationscale',data,mu,sig,df),p);
set(h,'color','r','linestyle','-')
title('{\bf Probability Plot}')
legend('Data','Normal','t','Location','NW')
```



On the other hand, the empirical distribution provides a perfect fit, but the outliers make the tails very discrete:

```
ecdf(data)
```

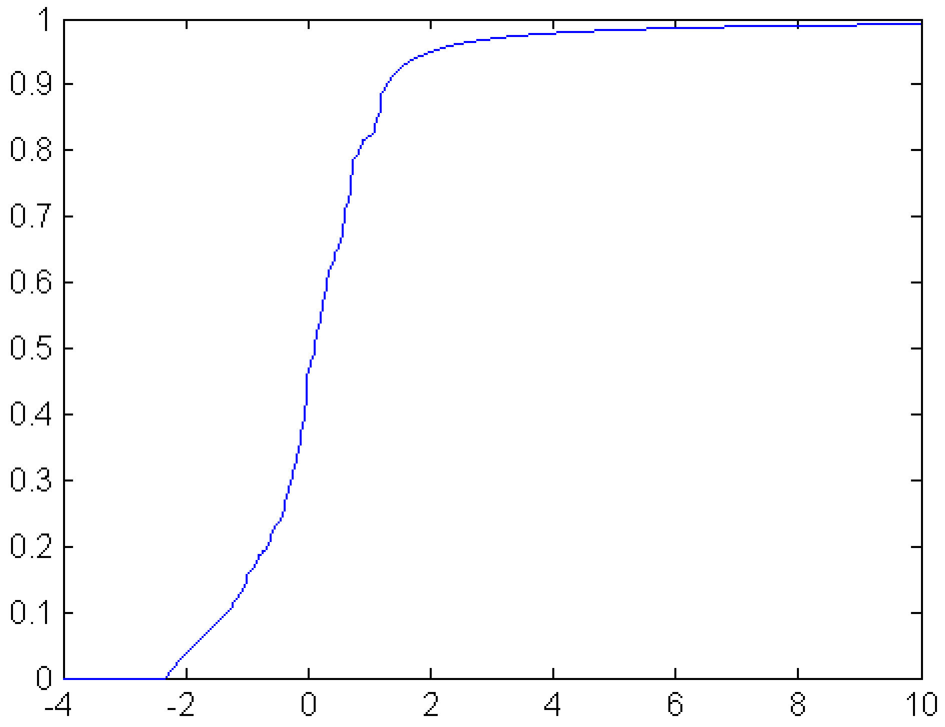


Random samples generated from this distribution by inversion might include, for example, values around 4.33 and 9.25, but nothing in-between.

The `paretotails` function fits a distribution by piecing together the empirical distribution in the center of the sample with smooth Pareto distributions in the tails. This provides a single, well-fit model for the entire sample. The following uses generalized Pareto distributions (GPDs) for the lower and upper 10% of the data:

```
pfit = paretotails(data,0.1,0.9)
pfit =
Piecewise distribution with 3 segments
 -Inf < x < -1.30726 (0 < p < 0.1)
        lower tail, GPD(-1.10167,1.12395)

 -1.30726 < x < 1.27213 (0.1 < p < 0.9)
        interpolated empirical cdf
```

```
     1.27213 < x < Inf (0.9 < p < 1)
           upper tail, GPD(1.03844,0.726038)

   x = -4:0.01:10;
   plot(x,pfit(x))
```



Note that the fit object `pfit` returned by `paretotails` allows for functional syntax of the form `pfit(x)` for evaluating the piecewise cdf. You can access other information about the fit using the methods listed in the tables above. Options for `paretotails` also allow for kernel smoothing of the center of the cdf.

Negative Log-Likelihood Functions

Negative log-likelihood functions for supported distributions in Statistics Toolbox all end with `like`, as in `explike`. Specific function names for specific distributions can be found in "Supported Distributions" on page 5-3.

Each function represents a parametric family of distributions. Input arguments are lists of parameter values specifying a particular member of the distribution family followed by an array of data. Functions return the negative log-likelihood of the parameters, given the data.

Negative log-likelihood functions are used as objective functions in search algorithms such as the one implemented by the `fminsearch` function in MATLAB. Additional search algorithms are implemented by functions in Optimization Toolbox and Genetic Algorithm and Direct Search Toolbox.

When used to compute maximum likelihood estimates (MLEs), negative log-likelihood functions allow you to choose a search algorithm and exercise low-level control over algorithm execution. By contrast, the functions discussed in "Distribution Fitting Functions" on page 5-104 use preset algorithms with options limited to those set by the `statset` function.

Likelihoods are conditional probability densities. A parametric family of distributions is specified by its pdf $f(x,a)$, where x and a represent the variables and parameters, respectively. When a is fixed, the pdf is used to compute the density at x, $f(x\,|\,a)$. When x is fixed, the pdf is used to compute the *likelihood* of the parameters a, $f(a\,|\,x)$. The joint likelihood of the parameters over an independent random sample X is

$$L(a) = \prod_{x \in X} f(a \mid x)$$

Given X, MLEs maximize $L(a)$ over all possible a.

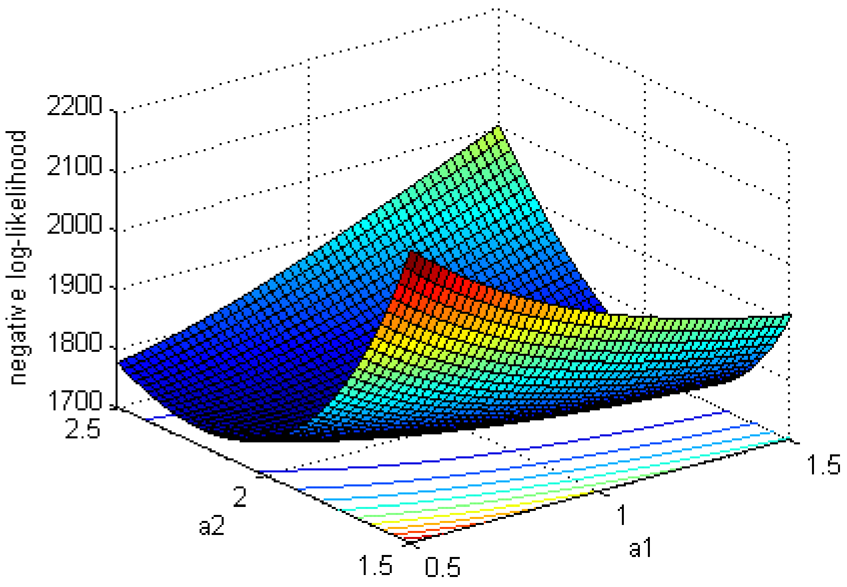
In numerical algorithms, the log-likelihood function, $\log(L(a))$, is (equivalently) optimized. The logarithm transforms the product of potentially small likelihoods into a sum of logs, which is easier to distinguish from 0 in computation. For convenience, the negative log-likelihood functions in Statistics Toolbox return the *negative* of this sum, since the optimization algorithms to which the values are passed typically search for minima rather than maxima.

For example, use `gamrnd` to generate a random sample from a specific gamma distribution:

```
a = [1,2];
X = gamrnd(a(1),a(2),1e3,1);
```

Given `X`, the `gamlike` function can be used to visualize the likelihood surface in the neighborhood of `a`:

```
mesh = 50;
delta = 0.5;
a1 = linspace(a(1)-delta,a(1)+delta,mesh);
a2 = linspace(a(2)-delta,a(2)+delta,mesh);
logL = zeros(mesh); % Preallocate memory
for i = 1:mesh
    for j = 1:mesh
        logL(i,j) = gamlike([a1(i),a2(j)],X);
    end
end

[A1,A2] = meshgrid(a1,a2);
surfc(A1,A2,logL)
```



The MATLAB `fminsearch` function can be used to search for the minimum of the likelihood surface:

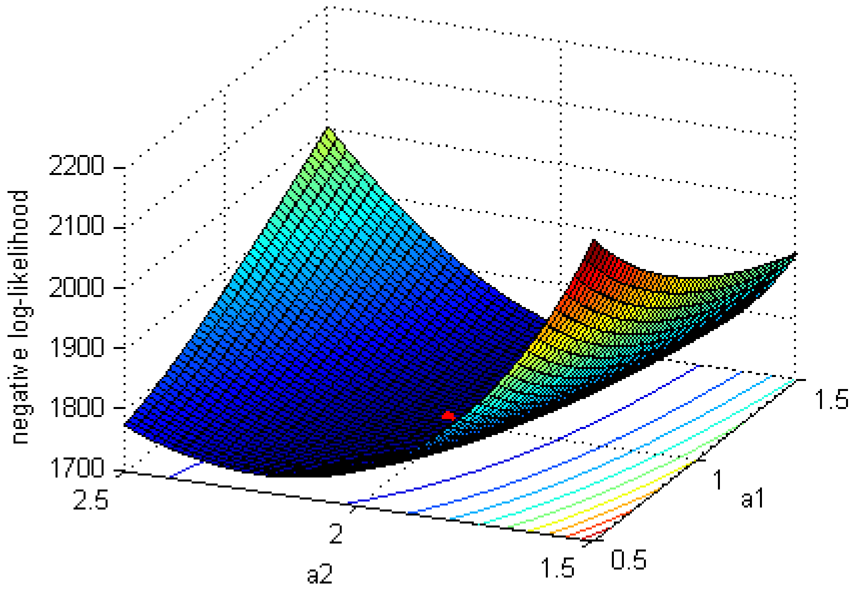
```
LL = @(u)gamlike([u(1),u(2)],X); % Likelihood given X
MLES = fminsearch(LL,[1,2])
MLES =
    1.0231    1.9729
```

These can be compared to the MLEs returned by the `gamfit` function, which uses a combination search and solve algorithm:

```
ahat = gamfit(X)
ahat =
    1.0231    1.9728
```

The MLEs can be added to the surface plot (rotated to show the minimum):

```
hold on
plot3(MLES(1),MLES(2),LL(MLES),...
      'ro','MarkerSize',5,...
      'MarkerFaceColor','r')
```


negative log-likelihood
2200
2100
2000
1900
1800
1700
2.5
2
1.5
a2
0.5
1
1.5
a1

Random Number Generators

Random number generators (RNGs) for supported distributions in Statistics Toolbox all end with `rnd`, as in `binornd` or `exprnd`. Specific RNG names for specific distributions can be found in "Supported Distributions" on page 5-3.

Each RNG represents a parametric family of distributions. Input arguments are lists of parameter values specifying a particular member of the distribution family followed by the dimensions of an array. RNGs return random numbers from the specified distribution in an array of the specified dimensions.

RNGs in Statistics Toolbox depend on the MATLAB base generators `rand` and/or `randn`, using the techniques discussed in "Methods of Random Number Generation" on page 5-158 to generate random numbers from particular distributions. Dependencies of specific RNGs are listed in the table below.

MATLAB resets the state of the base RNGs each time it is started. Thus, by default, dependent RNGs in Statistics Toolbox will generate the same sequence of values with each MATLAB session. To change this behavior, the state of the base RNGs must be set explicitly in `startup.m` or in the relevant program code. States can be set to fixed values, for reproducibility, or to time-dependent values, to assure new random sequences. For details on setting the state and the method used by the base RNGs, see `rand` and `randn`.

For example, to simulate flips of a biased coin:

```
p = 0.55; % Probability of heads
n = 100; % Number of flips per trial
N = 1e3; % Number of trials
rand('state',sum(100*clock)) % Set base generator
sims = unifrnd(0,1,n,N) < p; % 1 for heads; 0 for tails
```

The empirical probability of heads for each trial is given by:

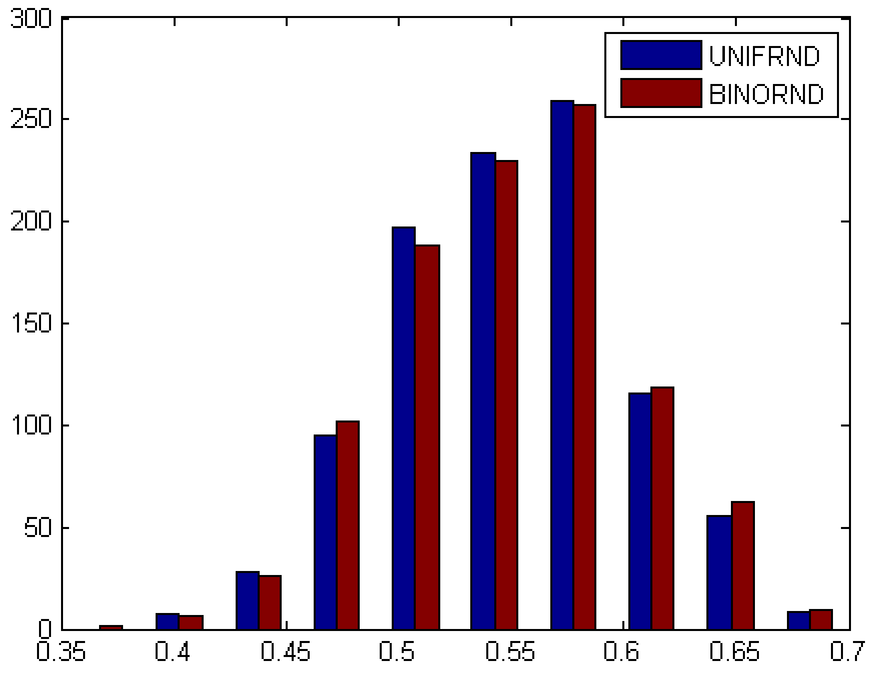
```
phat = sum(sims)/n;
```

The probability of heads for each trial can also be simulated by:

```
prand = = binornd(n,p,1,N)/n;
```

You can compare the two simulations with a histogram:

```
hist([phat' prand'])
legend('UNIFRND','BINORND')
```



Dependencies of the Random Number Generators

The following table lists the dependencies of the RNGs in Statistics Toolbox on the MATLAB base RNGs `rand` and/or `randn`. Set the states and methods of the RNGs in the right-hand column to assure reproducibility/variability of the outputs of the RNGs in the left-hand column.

RNG	MATLAB Base RNG
`betarnd`	`rand, randn`
`binornd`	`rand`
`chi2rnd`	`rand, randn`
`evrnd`	`rand`
`exprnd`	`rand`
`frnd`	`rand, randn`
`gamrnd`	`rand, randn`
`geornd`	`rand`
`gevrnd`	`rand`
`gprnd`	`rand`
`hygernd`	`rand`
`iwishrnd`	`rand, randn`
`johnsrnd`	`randn`
`lhsdesign`	`rand`
`lhsnorm`	`rand`
`lognrnd`	`randn`
`mhsample`	`rand` or `randn`, depending on the RNG given for the proposal distribution
`mvnrnd`	`randn`
`mvtrnd`	`rand, randn`
`nbinrnd`	`rand, randn`
`ncfrnd`	`rand, randn`

RNG	MATLAB Base RNG
`nctrnd`	`rand, randn`
`ncx2rnd`	`randn`
`normrnd`	`randn`
`pearsrnd`	`rand` or `randn`, depending on the distribution type
`poissrnd`	`rand, randn`
`random`	`rand` or `randn`, depending on the specified distribution
`randsample`	`rand`
`raylrnd`	`randn`
`slicesample`	`rand`
`trnd`	`rand, randn`
`unidrnd`	`rand`
`unifrnd`	`rand`
`wblrnd`	`rand`
`wishrnd`	`rand, randn`

Distribution GUIs

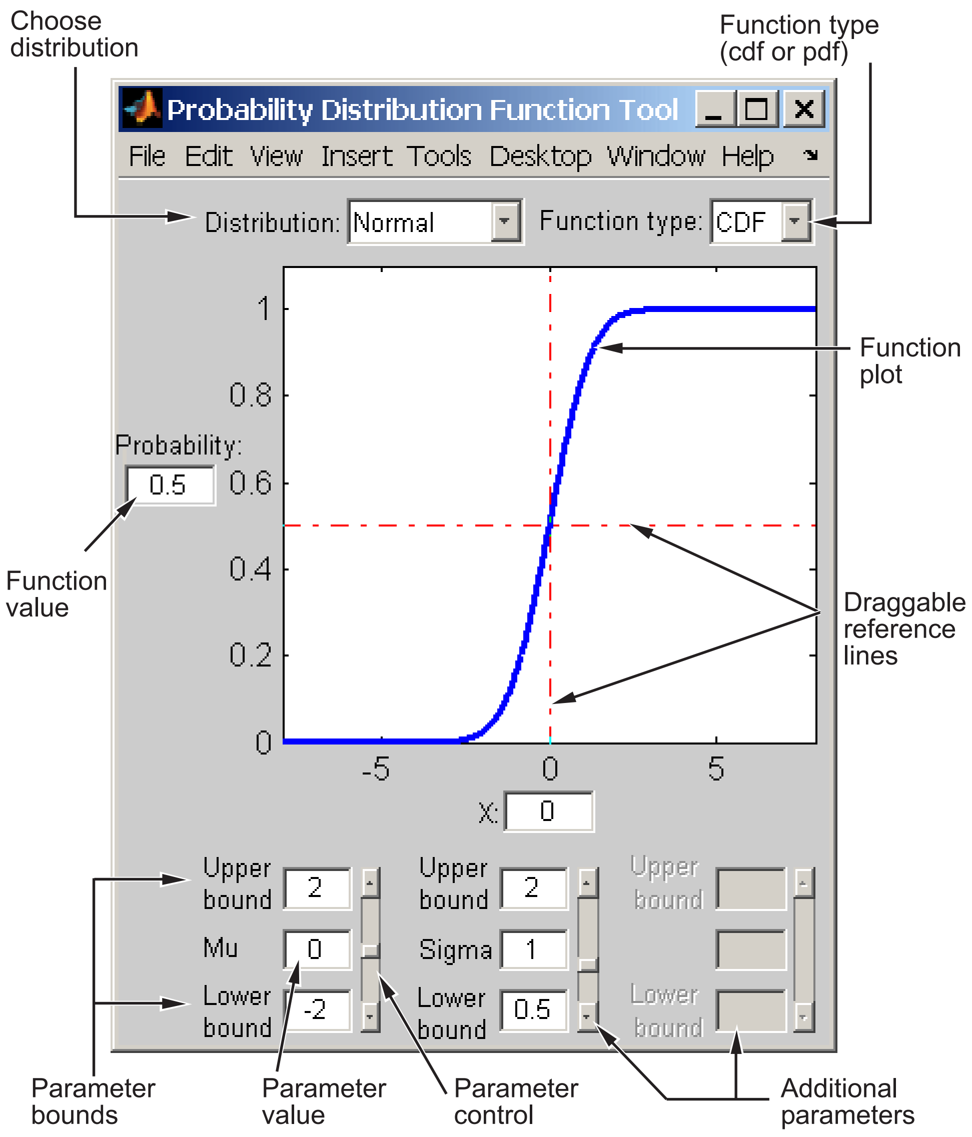
The following sections describe GUIs in Statistics Toolbox that provide convenient, interactive access to the distribution functions described in "Distribution Functions" on page 5-94:

Distribution Function Tool (p. 5-120)	Interactive effects of parameter changes to pdfs, cdfs
Distribution Fitting Tool (p. 5-122)	Interactive distribution fitting
Random Number Generation Tool (p. 5-155)	Interactive random number generation from supported distributions

Distribution Function Tool

To interactively see the influence of parameter changes on the shapes of the pdfs and cdfs of distributions supported by Statistics Toolbox, use the Probability Distribution Function Tool.

Run the tool by typing `disttool` at the command line. You can also run it from the Demos tab in the Help browser.



Start by selecting a distribution. Then choose the function type: probability density function (pdf) or cumulative distribution function (cdf).

Once the plot displays, you can

- Calculate a new function value by typing a new `x` value in the text box on the *x*-axis, dragging the vertical reference line, or clicking in the figure where you want the line to be. The new function value displays in the text box to the left of the plot.
- For cdf plots, find critical values corresponding to a specific probability by typing the desired probability in the text box on the *y*-axis or by dragging the horizontal reference line.
- Use the controls at the bottom of the window to set parameter values for the distribution and to change their upper and lower bounds.

Distribution Fitting Tool

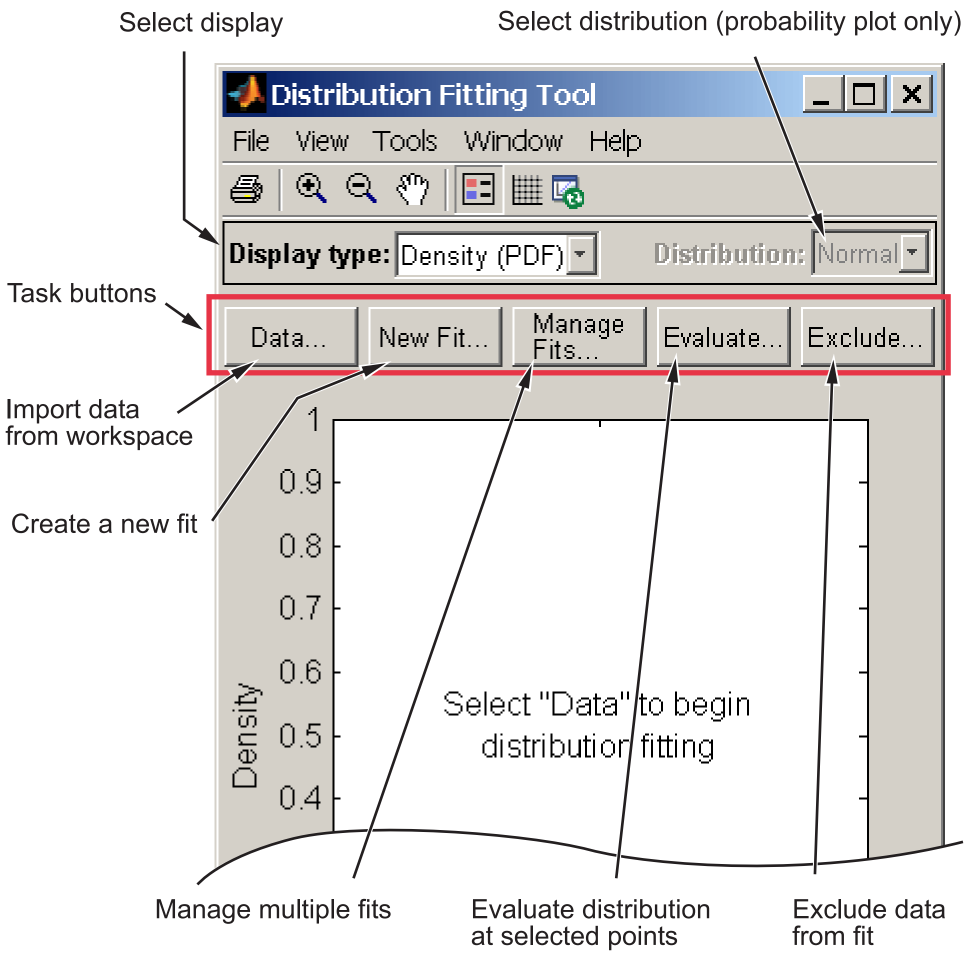
The Distribution Fitting Tool is a GUI for fitting univariate distributions to data. This section describes how to use the Distribution Fitting Tool and covers the following topics:

Main Window of the Distribution Fitting Tool

To open the Distribution Fitting Tool, enter the command

```
dfittool
```

The following figure shows the main window of the Distribution Fitting Tool.


Select display
Select distribution (probability plot only)
Distribution Fitting Tool
File View Tools Window Help
Display type: Density (PDF)
Distribution: Normal
Task buttons
Data...
New Fit...
Manage Fits...
Evaluate...
Exclude...
Import data from workspace
Create a new fit
Density
1
0.9
0.8
0.7
0.6
0.5
0.4
Select "Data" to begin distribution fitting
Manage multiple fits
Evaluate distribution at selected points
Exclude data from fit

Plot Buttons. Buttons at the top of the tool allow you to adjust the plot displayed in the main window:

- — Toggle the legend on (default) or off.
- — Toggle grid lines on or off (default).
- — Restore default axes limits.

Display Type. The **Display Type** field specifies the type of plot displayed in the main window. Each type corresponds to a probability function, for example, a probability density function. The following display types are available:

- `Density (PDF)` — Displays a probability density function (PDF) plot for the fitted distribution.
- `Cumulative probability (CDF)` — Displays a cumulative probability plot of the data.
- `Quantile (inverse CDF)` — Displays a quantile (inverse CDF) plot.
- `Probability plot` — Displays a probability plot.
- `Survivor function` — Displays a survivor function plot of the data.
- `Cumulative hazard` — Displays a cumulative hazard plot of the data.

Task Buttons. The task buttons enable you to perform the tasks necessary to fit distributions to data. Each button opens a new window in which you perform the task. The buttons include

- **Data** — Import and manage data sets. See "Creating and Managing Data Sets" on page 5-131.
- **New Fit** — Create new fits. See "Creating a New Fit" on page 5-135.
- **Manage Fits** — Manage existing fits. See "Managing Fits" on page 5-141.
- **Evaluate** — Evaluate fits at any points you choose. See "Evaluating Fits" on page 5-143.
- **Exclude** — Create rules specifying which values to exclude when fitting a distribution. See "Excluding Data" on page 5-146.

Display Pane. The display pane displays plots of the data sets and fits you create. Whenever you make changes in one of the task windows, the results are updated in the display pane.

Menu Options. The Distribution Fitting Tool menus contain items that enable you to do the following:

- Save and load sessions. See "Saving and Loading Sessions" on page 5-151.
- Generate an M-file with which you can fit distributions to data and plot the results independently of the Distribution Fitting Tool. See "Generating an M-File to Fit and Plot Distributions" on page 5-152.
- Define and import custom distributions. See "Using Custom Distributions" on page 5-153.

Example: Fitting a Distribution

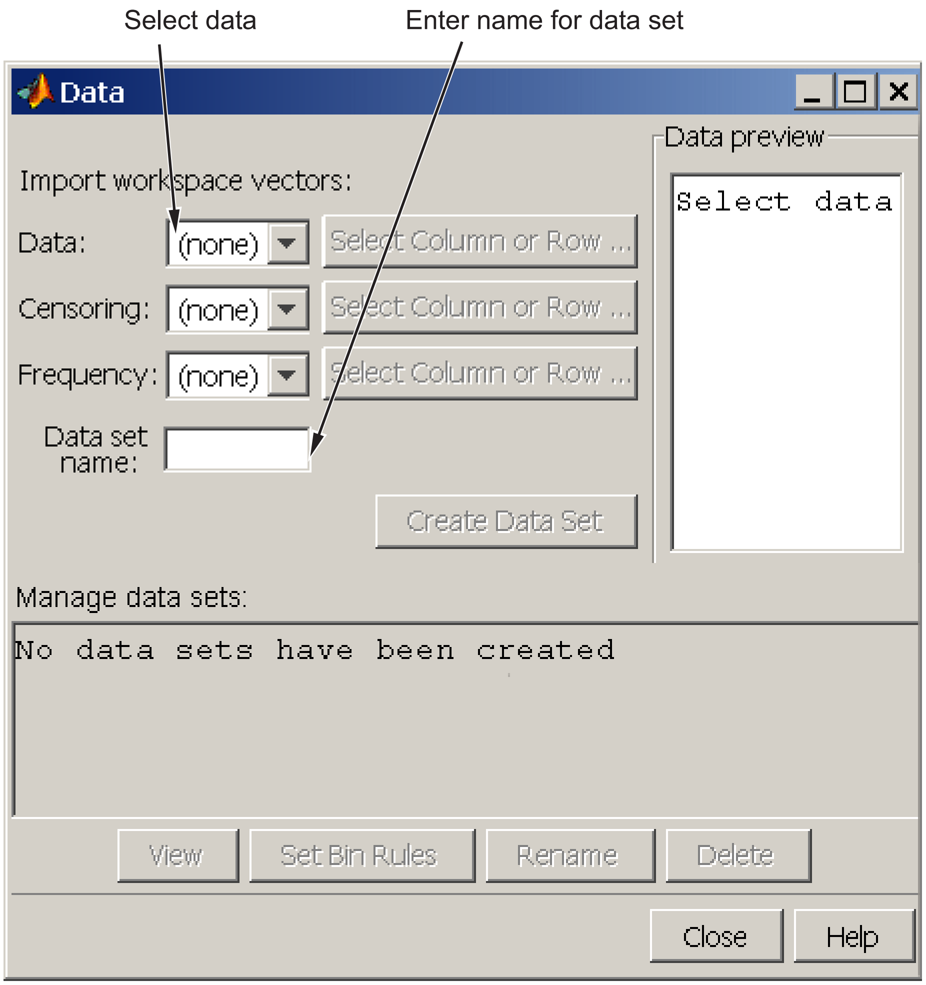
This section presents an example that illustrates how to use the Distribution Fitting Tool. The example involves the following steps:

- "Create Random Data for the Example" on page 5-125
- "Import Data into the Distribution Fitting Tool" on page 5-125
- "Create a New Fit" on page 5-128

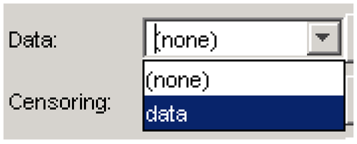
Create Random Data for the Example. To try the example, first generate some random data to which you will fit a distribution. The following command generates a vector `data`, of length `100`, whose entries are random numbers from a normal distribution with mean`.36` and standard deviation `1.4`.

```
data = normrnd(.36, 1.4, 100, 1);
```

Import Data into the Distribution Fitting Tool. To import the vector `data` into the Distribution Fitting Tool, click the **Data** button in main window. This opens the window shown in the following figure.

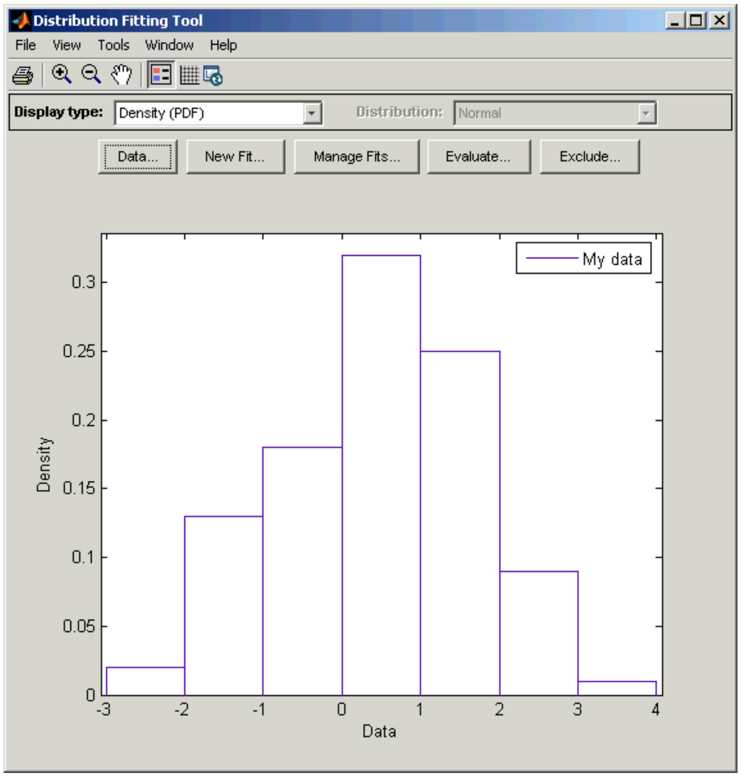


The **Data** field displays all numeric arrays in the MATLAB workspace. Select `data` from the drop-down list, as shown in the following figure.



This displays a histogram of the data in the **Data preview** pane.

In the **Data set name** field, type a name for the data set, such as `My data`, and click **Create Data Set** to create the data set. The main window of the Distribution Fitting Tool now displays a larger version of the histogram in the **Data preview** pane, as shown in the following figure.

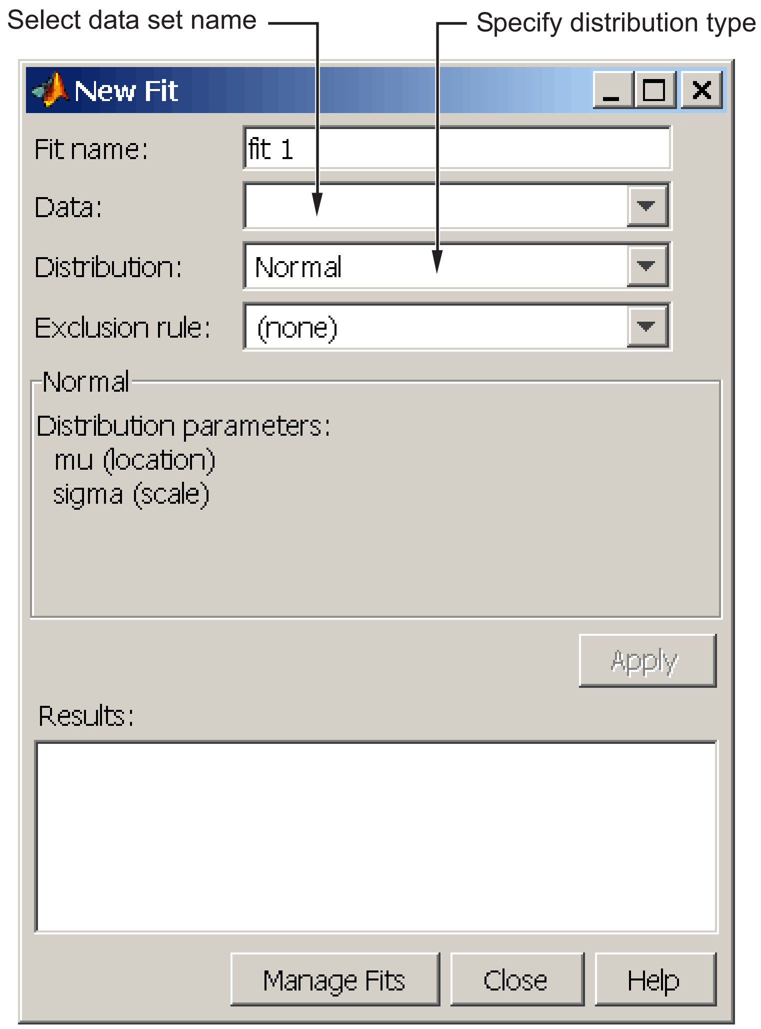


Display type: Density (PDF)
Distribution: Normal

Histogram of the Data

Note Because the example uses random data, you might see a slightly different histogram if you try this example for yourself.

Create a New Fit. To fit a distribution to the data, click **New Fit** in the main window of the Distribution Fitting Tool. This opens the window shown in the following figure.



Fit name: fit 1

Data:

Distribution: Normal

Exclusion rule: (none)

Normal

Distribution parameters:
mu (location)
sigma (scale)

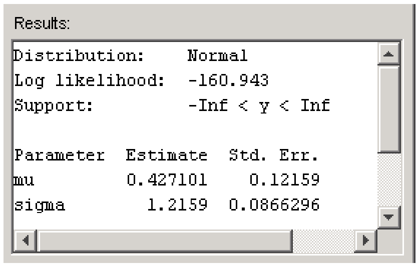
To fit a normal distribution, the default entry of the **Distribution** field, to `My data`:

1 Enter a name for the fit, such as `My fit`, in the **Fit name** field.

2 Select `My data` from the drop-down list in the **Data** field.

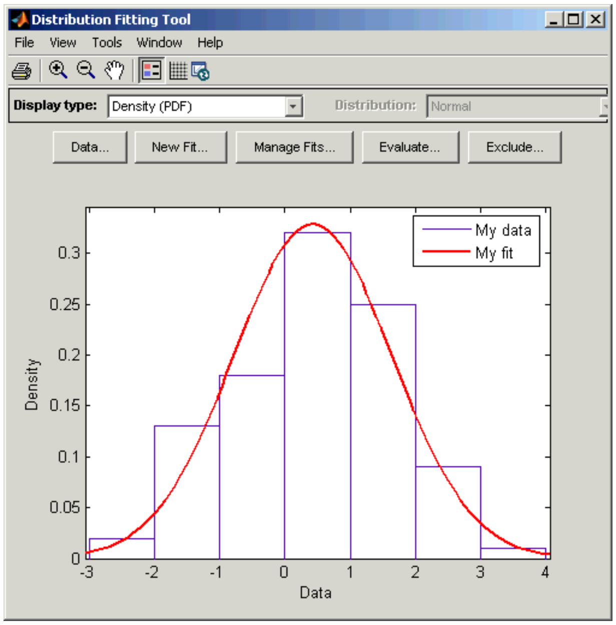
3 Click **Apply**.

The **Results** pane displays the mean and standard deviation of the normal distribution that best fits `My data`, as shown in the following figure.



```
Distribution:    Normal
Log likelihood:  -160.943
Support:         -Inf < y < Inf

Parameter  Estimate  Std. Err.
mu         0.427101    0.12159
sigma        1.2159  0.0866296
```

The main window of the Distribution Fitting Tool displays a plot of the normal distribution with this mean and standard deviation, as shown in the following figure.

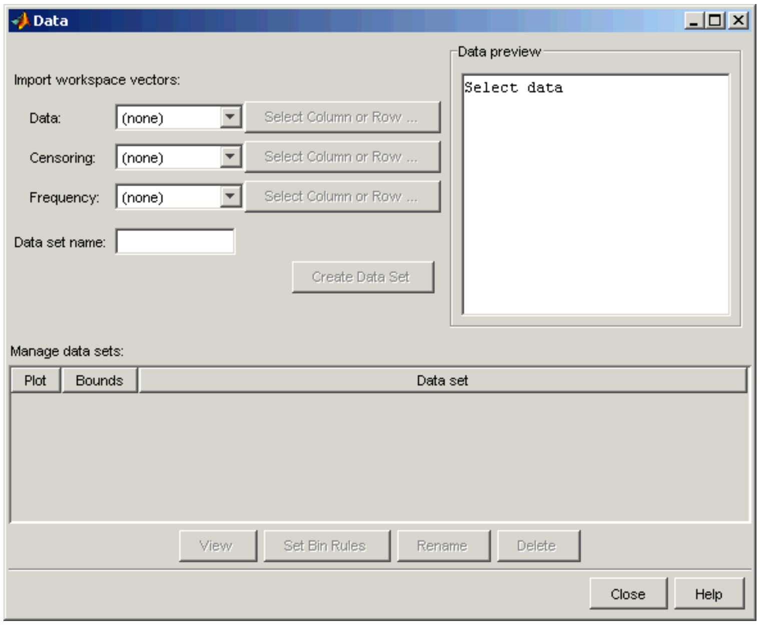


Plot of the Distribution and Data

Creating and Managing Data Sets

This section describes how create and manage data sets.

To begin, click the **Data** button in the main window of the Distribution Fitting Tool to open the Data window shown in the following figure.



Importing Data. The **Import workspace vectors** pane enables you to create a data set by importing a vector from the MATLAB workspace. The following sections describe the fields of the **Import workspace vectors** pane.

Data

The drop-down list in the **Data** field contains the names of all matrices and vectors, other than 1-by-1 matrices (scalars) in the MATLAB workspace. Select the array containing the data you want to fit. The actual data you import must be a vector. If you select a matrix in the **Data** field, the first column of the matrix is imported by default. To select a different column or row of the matrix, click **Select Column or Row**. This displays the matrix in the Array Editor, where you can select a row or column by highlighting it with the mouse.

Alternatively, you can enter any valid MATLAB expression in the **Data** field.

When you select a vector in the **Data** field, a histogram of the data is displayed in the **Data preview** pane.

Censoring

If some of the points in the data set are censored, enter a Boolean vector, of the same size as the data vector, specifying the censored entries of the data. A `1` in the censoring vector specifies that the corresponding entry of the data vector is censored, while a `0` specifies that the entry is not censored. If you enter a matrix, you can select a column or row by clicking **Select Column or Row**. If you do not want to censor any data, leave the **Censoring** field blank.

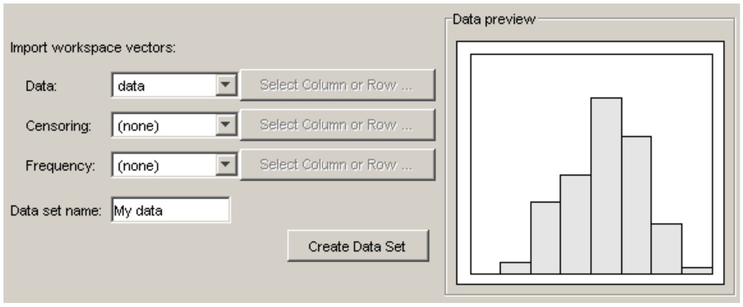
Frequency

Enter a vector of positive integers of the same size as the data vector to specify the frequency of the corresponding entries of the data vector. For example, a value of `7` in the 15th entry of frequency vector specifies that there are 7 data points corresponding to the value in the 15th entry of the data vector. If all entries of the data vector have frequency 1, leave the **Frequency** field blank.

Data name

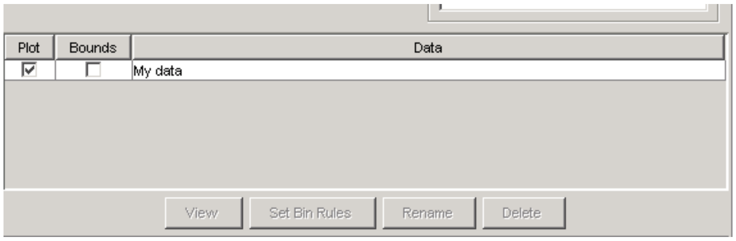
Enter a name for the data set you import from the workspace, such as `My data`.

As an example, if you create the vector `data` described in "Example: Fitting a Distribution" on page 5-125, and select it in the **Data** field, the upper half of the Data window appears as in the following figure.



After you have entered the information in the preceding fields, click **Create Data Set** to create the data set `My data`.

Managing Data Sets. The **Manage data sets** pane enables you to view and manage the data sets you create. When you create a data set, its name appears in the **Data sets** list. The following figure shows the **Manage data sets** pane after creating the data set `My data`.



For each data set in the **Data sets** list, you can

- Select the **Plot** check box to display a plot of the data in the main Distribution Fitting Tool window. When you create a new data set, **Plot** is selected by default. Clearing the **Plot** check box removes the data from the plot in the main window. You can specify the type of plot displayed in the **Display Type** field in the main window.
- If **Plot** is selected, you can also select **Bounds** to display confidence interval bounds for the plot in the main window. These bounds are pointwise confidence bounds around the empirical estimates of these functions. The bounds are only displayed when you set **Display Type** in the main window to one of the following:

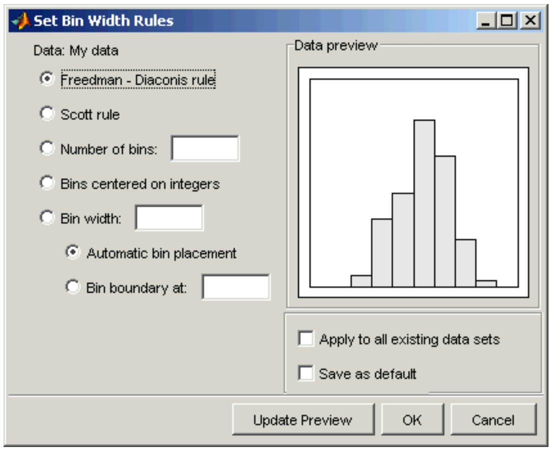
- `Cumulative probability (CDF)`
- `Survivor function`
- `Cumulative hazard`

The Distribution Fitting Tool cannot display confidence bounds on density (PDF), quantile (inverse CDF), or probability plots. Clearing the **Bounds** check box removes the confidence bounds from the plot in the main window.

When you select a data set from the list, the following buttons are enabled:

- **View** — Displays the data in a table in a new window.
- **Set Bin Rules** — Defines the histogram bins used in a density (PDF) plot.
- **Rename** — Renames the data set.
- **Delete** — Deletes the data set.

Setting Bin Rules. To set bin rules for the histogram of a data set, click **Set Bin Rules**. This opens the dialog box shown in the following figure.


Set Bin Width Rules

Data: My data

[✓ Freedman - Diaconis rule]

Scott rule

Number of bins:

Bins centered on integers

Bin width:

[✓ Automatic bin placement]

Bin boundary at:

Data preview

☐ Apply to all existing data sets

☐ Save as default

You can select from the following rules:

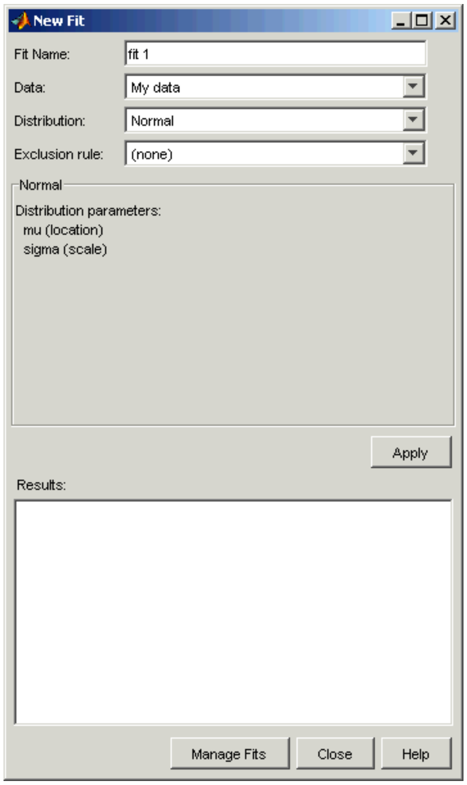
- **Freedman-Diaconis** rule — Algorithm that chooses bin widths and locations automatically, based on the sample size and the spread of the data. This rule, which is the default, is suitable for many kinds of data.
- **Scott rule** — Algorithm intended for data that are approximately normal. The algorithm chooses bin widths and locations automatically.
- **Number of bins** — Enter the number of bins. All bins have equal widths.
- **Bins centered on integers** — Specifies bins centered on integers.
- **Bin width** — Enter the width of each bin. If you select this option, you can make the following choices:
  - **Automatic bin placement** — Places the edges of the bins at integer multiples of the **Bin width**.
  - **Bin boundary at** — Enter a scalar to specify the boundaries of the bins. The boundary of each bin is equal to this scalar plus an integer multiple of the **Bin width**.

The Set Bin Width Rules dialog box also provides the following options:

- **Apply to all existing data sets** — When selected, the rule is applied to all data sets. Otherwise, the rule is only applied to the data set currently selected in the Data window.
- **Save as default** — When selected, the current rule is applied to any new data sets that you create. You can also set default bin width rules by selecting `Set Default Bin Rules` from the **Tools** menu in the main window.

Creating a New Fit

This section describes how to create a new fit. To begin, click the **New Fit** button at the top of the main window to open a New Fit window. If you created the data set `My data`, as described in "Example: Fitting a Distribution" on page 5-125, `My data` appears in the **Data** field, as shown in the following figure.



Fit Name: fit 1

Data: My data

Distribution: Normal

Exclusion rule: (none)

Normal

Distribution parameters:
mu (location)
sigma (scale)

Fit Name. Enter a name for the fit in the **Fit Name** field.

Data. The **Data** field contains a drop-down list of the data sets you have created. Select the data set to which you want to fit a distribution.

Distribution. Select the type of distribution you want to fit from the **Distribution** drop-down list. See "Available Distributions" on page 5-137 for a list of distributions supported by the Distribution Fitting Tool.

Note Only the distributions that apply to the values of the selected data set are displayed in the **Distribution** field. For example, positive distributions are not displayed when the data include values that are zero or negative.

You can specify either a parametric or a nonparametric distribution. When you select a parametric distribution from the drop-down list, a description of its parameters is displayed in the pane below the **Exclusion rule** field. The Distribution Fitting Tool estimates these parameters to fit the distribution to the data set. When you select `Nonparametric fit`, options for the fit appear in the pane, as described in "Options for Nonparametric Fits" on page 5-139.

Exclusion Rule. You can specify a rule to exclude some the data in the **Exclusion rule** field. You can create an exclusion rule by clicking **Exclude** in the main window of the Distribution Fitting Tool. For more information, see "Excluding Data" on page 5-146.

Apply the New Fit. Click **Apply** to fit the distribution. For a parametric fit, the **Results** pane displays the values of the estimated parameters. For a nonparametric fit, the **Results** pane displays information about the fit.

When you click **Apply**, the main window of Distribution Fitting Tool displays a plot of the distribution, along with the corresponding data.

Note When you click **Apply**, the title of the window changes to Edit Fit. You can now make changes to the fit you just created and click **Apply** again to save them. After closing the Edit Fit window, you can reopen it from the Fit Manager window at any time to edit the fit.

Available Distributions. This section lists the distributions available in the Distribution Fitting Tool.

Most, but not all, of the distributions available in the Distribution Fitting Tool are supported elsewhere in Statistics Toolbox (see "Supported Distributions" on page 5-3), and have dedicated distribution fitting functions. These functions are used to compute the majority of the fits in the Distribution Fitting Tool, and are referenced in the list below.

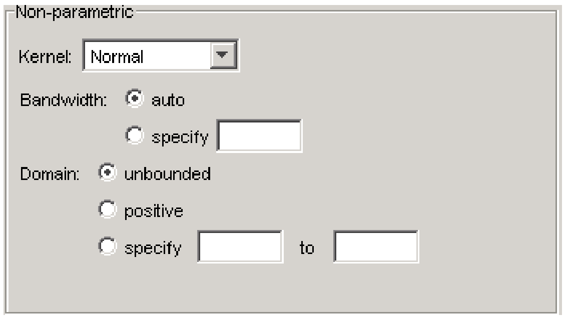
Other fits are computed using functions internal to the Distribution Fitting Tool. Distributions that do not have corresponding fitting functions in Statistics Toolbox are described in "Additional Distributions Available in the Distribution Fitting Tool" on page 5-155.

Not all of the distributions listed below are available for all data sets. The Distribution Fitting Tool determines the extent of the data (nonnegative, unit interval, etc.) and displays appropriate distributions in the **Distribution** drop-down list. Distribution data ranges are given parenthetically in the list below.

- Beta (unit interval values) distribution, fit using the function `betafit`.
- Binomial (nonnegative values) distribution, fit using the function `binopdf`.
- Birnbaum-Saunders (positive values) distribution.
- Exponential (nonnegative values) distribution, fit using the function `expfit`.
- Extreme value (all values) distribution, fit using the function `evfit`.
- Gamma (positive values) distribution, fit using the function `gamfit`.
- Generalized extreme value (all values) distribution, fit using the function `gevfit`.
- Generalized Pareto (all values) distribution, fit using the function `gpfit`.
- Inverse Gaussian (positive values) distribution.
- Logistic (all values) distribution.
- Loglogistic (positive values) distribution.
- Lognormal (positive values) distribution, fit using the function `lognfit`.
- Nakagami (positive values) distribution.
- Negative binomial (nonnegative values) distribution, fit using the function `nbinpdf`.
- Nonparametric (all values) distribution, fit using the function `ksdensity`. See "Options for Nonparametric Fits" on page 5-139 for a description of available options.
- Normal (all values) distribution, fit using the function `normfit`.
- Poisson (nonnegative integer values) distribution, fit using the function `poisspdf`.
- Rayleigh (positive values) distribution using the function `raylfit`.
- Rician (positive values) distribution.

- *t* location-scale (all values) distribution.
- Weibull (positive values) distribution using the function `wblfit`.

Options for Nonparametric Fits. When you select `Non-parametric` in the **Distribution** field, a set of options appears in the pane below **Exclusion rule**, as shown in the following figure.



Kernel: Normal

Bandwidth: (● auto) ○ specify

Domain: (● unbounded) ○ positive ○ specify

The options for nonparametric distributions are

- **Kernel** — Type of kernel function to use. The options are
  - `Normal`
  - `Box`
  - `Triangle`
  - `Epanechnikov`
- **Bandwidth** — The bandwidth of the kernel smoothing window. Select **auto** for a default value that is optimal for estimating normal densities. This value is displayed in the **Fit results** pane after you click **Apply**. Select **specify** and enter a smaller value to reveal features such as multiple modes or a larger value to make the fit smoother.
- **Domain** — The allowed *x*-values for the density. The options are
  - **unbounded** — The density extends over the whole real line.
  - **positive** — The density is restricted to positive values.
  - **specify** — Enter lower and upper bounds for the domain of the density.

When you select **positive** or **specify**, the nonparametric fit has zero probability outside the specified domain.

Displaying Results

This section explains the different ways to display results in the main window of the Distribution Fitting Tool. The main window displays plots of

- The data sets for which you select **Plot** in the Data window.
- The fits for which you select **Plot** in the Fit Manager window.
- Confidence bounds for
  - Data sets for which you select **Bounds** in the Data window.
  - Fits for which you select **Bounds** in the Fit Manager.

Display Type. The **Display Type** field in the main window specifies the type of plot displayed. Each type corresponds to a probability function, for example, a probability density function. The following display types are available:

- `Density (PDF)` — Displays a probability density function (PDF) plot for the fitted distribution. The main window displays data sets using a probability histogram, in which the height of each rectangle is the fraction of data points that lie in the bin divided by the width of the bin. This makes the sum of the areas of the rectangles equal to 1.
- `Cumulative probability (CDF)` — Displays a cumulative probability plot of the data. The main window displays data sets using a cumulative probability step function. The height of each step is the cumulative sum of the heights of the rectangles in the probability histogram.
- `Quantile (inverse CDF)` — Displays a quantile (inverse CDF) plot.
- `Probability plot` — Displays a probability plot of the data. You can specify the type of distribution used to construct the probability plot in the **Distribution** field, which is only available when you select `Probability plot`. The choices for the distribution are
  - `Exponential`
  - `Extreme value`
  - `Logistic`

  - `Log-Logistic`
  - `Lognormal`
  - `Normal`
  - `Rayleigh`
  - `Weibull`

  In addition to these choices, you can create a probability plot against a parametric fit that you create in the New Fit panel. These fits are added at the bottom of the Distribution drop-down list when you create them.

- `Survivor function` — Displays a survivor function plot of the data.
- `Cumulative hazard` — Displays a cumulative hazard plot of the data.

Note Some of these distributions are not available if the plotted data includes `0` or negative values.

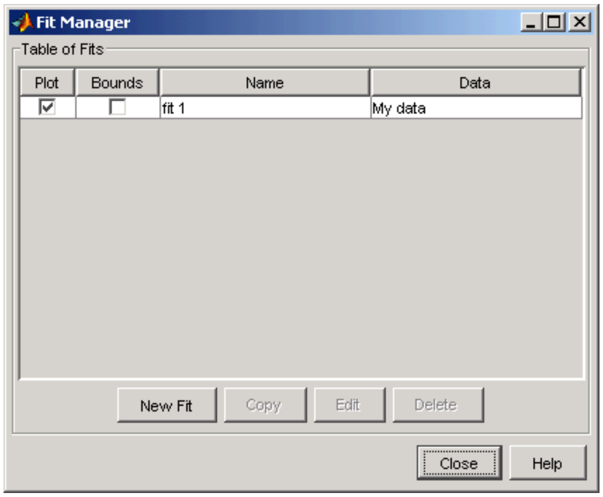
Confidence Bounds. You can display confidence bounds for data sets and fits, provided that you set **Display Type** to `Cumulative probability (CDF)`, `Survivor function`, `Cumulative hazard`, or `Quantile` for fits only.

- To display bounds for a data set, select **Bounds** next to the data set in the **Data sets** pane of the Data window.
- To display bounds for a fit, select **Bounds** next to the fit in the **Fit Manager** window. Confidence bounds are not available for all fit types.

To set the confidence level for the bounds, select `Confidence Level` from the **View** menu in the main window and choose from the options.

Managing Fits

This section describes how to manage fits that you have created. To begin, click the **Manage Fits** button in the main window of the Distribution Fitting Tool. This opens the Fit Manager window as shown in the following figure.



Plot	Bounds	Name	Data
✓	☐	fit 1	My data

The **Table of fits** displays a list of the fits you create.

Plot. Select **Plot** to display a plot of the fit in the main window of the Distribution Fitting Tool. When you create a new fit, **Plot** is selected by default. Clearing the **Plot** check box removes the fit from the plot in the main window.

Bounds. If **Plot** is selected, you can also select **Bounds** to display confidence bounds in the plot. The bounds are displayed when you set **Display Type** in the main window to one of the following:

- `Cumulative probability (CDF)`
- `Quantile (inverse CDF)`
- `Survivor function`
- `Cumulative hazard`

The Distribution Fitting Tool cannot display confidence bounds on density (PDF) or probability plots. In addition, bounds are not supported for nonparametric fits and some parametric fits.

Clearing the **Bounds** check box removes the confidence intervals from the plot in the main window.

When you select a fit in the **Table of fits**, the following buttons are enabled below the table:

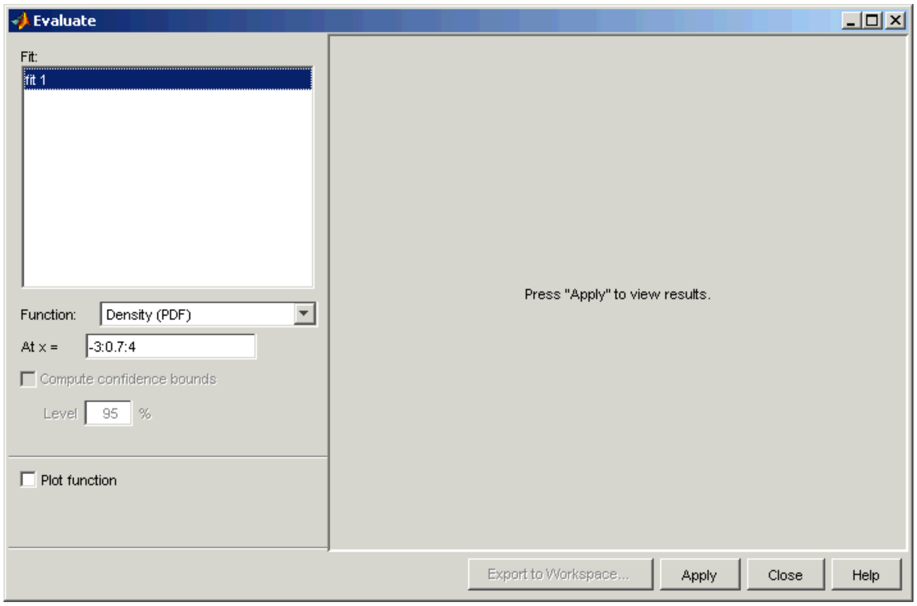
- **New Fit** — Opens a New Fit window.
- **Copy** — Creates a copy of the selected fit.
- **Edit** — Opens an Edit Fit window, where you can edit the fit.

Note You can only edit the currently selected fit in the Edit Fit window. To edit a different fit, select it in the **Table of fits** and click **Edit** to open another Edit Fit window.

- **Delete** — Deletes the selected fit.

Evaluating Fits

The Evaluate window enables you to evaluate any fit at whatever points you choose. To open the window, click the **Evaluate** button in the main window of the Distribution Fitting Tool. The following figure shows the Evaluate window.



The Evaluate window contains the following items:

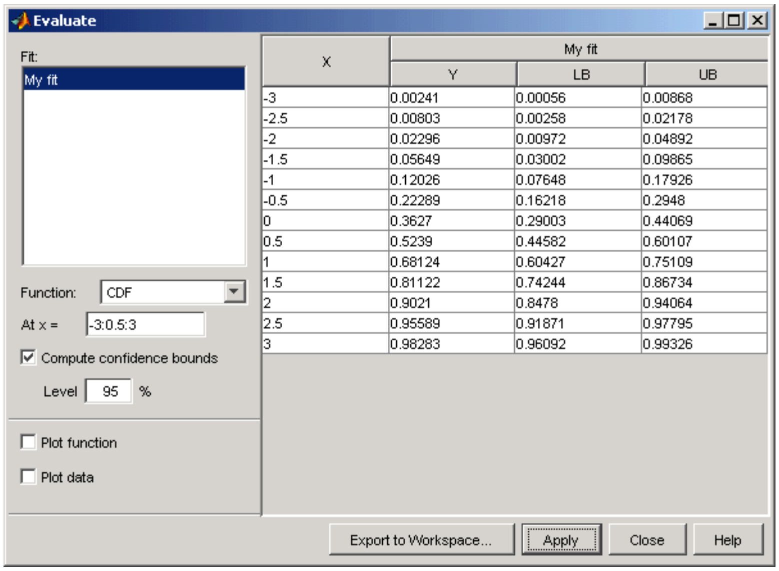
- **Fit** pane — Displays the names of existing fits. Select one or more fits that you want to evaluate. Using your platform specific functionality, you can select multiple fits.
- **Function** — Select the type of probability function you want to evaluate for the fit. The available functions are
  - `Density (PDF)` — Computes a probability density function.
  - `Cumulative probability (CDF)` — Computes a cumulative probability function.
  - `Quantile (inverse CDF)` — Computes a quantile (inverse CDF) function.
  - `Survivor function` — Computes a survivor function.
  - `Cumulative hazard` — Computes a cumulative hazard function.
  - `Hazard rate` — Computes the hazard rate.

- **At x =** — Enter a vector of points at which you want to evaluate the distribution function. If you change **Function** to `Quantile (inverse CDF)`, the field name changes to **At p =** and you enter a vector of probability values.
- **Compute confidence bounds** — Select this box to compute confidence bounds for the selected fits. The check box is only enabled if you set **Function** to one of the following:
  - `Cumulative probability (CDF)`
  - `Quantile (inverse CDF)`
  - `Survivor function`
  - `Cumulative hazard`

  The Distribution Fitting Tool cannot compute confidence bounds for nonparametric fits and for some parametric fits. In these cases, the tool returns `NaN` for the bounds.
- **Level** — Set the level for the confidence bounds.
- **Plot function** — Select this box to display a plot of the distribution function, evaluated at the points you enter in the **At x =** field, in a new window.

Note The settings for **Compute confidence bounds**, **Level**, and **Plot function** do not affect the plots that are displayed in the main window of the Distribution Fitting Tool. The settings only apply to plots you create by clicking **Plot function** in the Evaluate window.

Click **Apply** to apply these settings to the selected fit. The following figure shows the results of evaluating the cumulative density function for the fit `My fit`, created in “Example: Fitting a Distribution” on page 5-125, at the points in the vector `-3:0.5:3`.



X	My fit		
	Y	LB	UB
-3	0.00241	0.00056	0.00868
-2.5	0.00803	0.00258	0.02178
-2	0.02296	0.00972	0.04892
-1.5	0.05649	0.03002	0.09865
-1	0.12026	0.07648	0.17926
-0.5	0.22289	0.16218	0.2948
0	0.3627	0.29003	0.44069
0.5	0.5239	0.44582	0.60107
1	0.68124	0.60427	0.75109
1.5	0.81122	0.74244	0.86734
2	0.9021	0.8478	0.94064
2.5	0.95589	0.91871	0.97795
3	0.98283	0.96092	0.99326

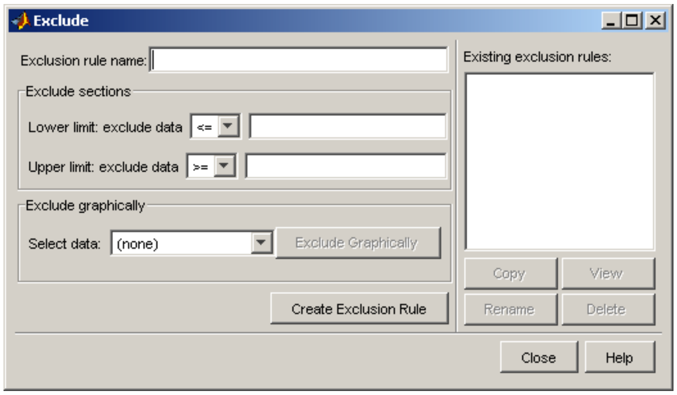
The window displays the following values in the columns of the table to the right of the **Fit** pane:

- X — The entries of the vector you enter in **At x =** field
- Y — The corresponding values of the CDF at the entries of X
- LB — The lower bounds for the confidence interval, if you select **Compute confidence bounds**
- UB — The upper bounds for the confidence interval, if you select **Compute confidence bounds**

To save the data displayed in the Evaluate window, click **Export to Workspace**. This saves the values in the table to a matrix in the MATLAB workspace.

Excluding Data

To exclude values from fit, click the **Exclude** button in the main window of the Distribution Fitting Tool. This opens the Exclude window, in which you can create rules for excluding specified values. You can use these rules to exclude data when you create a new fit in the New Fit window. The following figure shows the Exclude window.



Lower limit: exclude data <=

Upper limit: exclude data >=

Select data: (none)

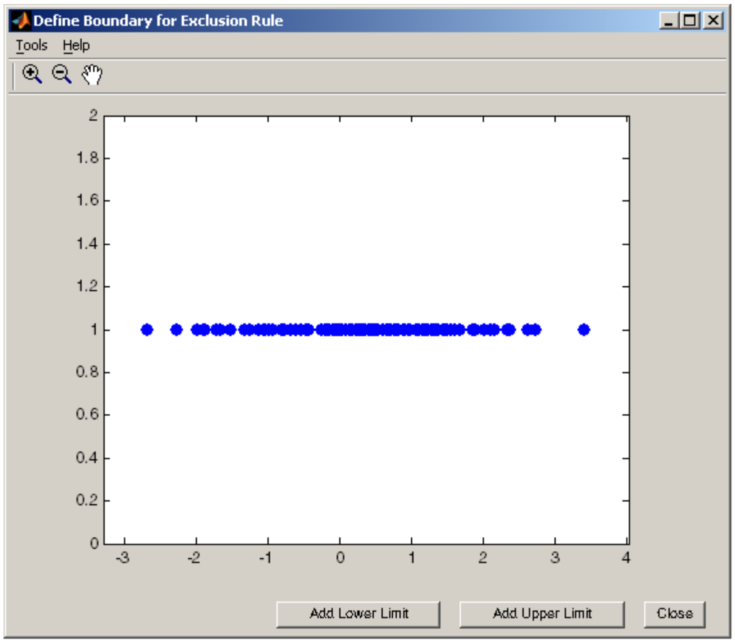
The following sections describe how to create an exclusion rule.

Exclusion Rule Name. Enter a name for the exclusion rule in the **Exclusion rule name** field.

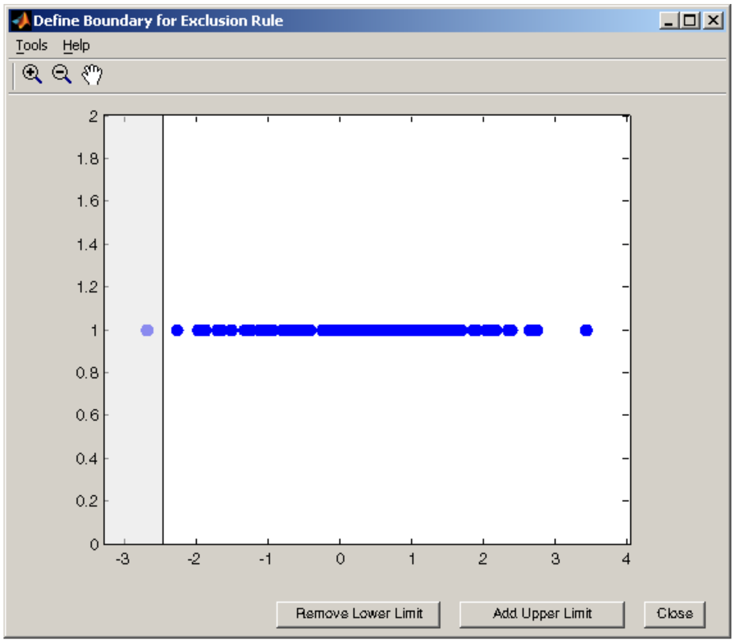
Exclude Sections. In the **Exclude sections** pane, you can specify bounds for the excluded data:

- In the **Lower limit: exclude Y** drop-down list, select <= or < from the drop-down list and enter a scalar in the field to the right. This excludes values that are either less than or equal to or less than that scalar, respectively.
- In the **Upper limit: exclude Y** drop-down list, select >= or > from the drop-down list and enter a scalar in the field to the right to exclude values that are either greater than or equal to or greater than the scalar, respectively.

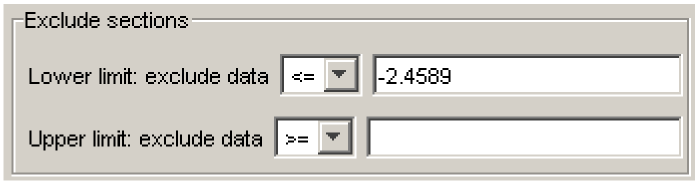
Exclude Graphically. The **Exclude Graphically** button enables you to define the exclusion rule by displaying a plot of the values in a data set and selecting the bounds for the excluded data with the mouse. For example, if you created the data set `My data`, described in "Creating and Managing Data Sets" on page 5-131, select it from the drop-down list next to **Exclude graphically** and then click the **Exclude graphically** button. This displays the values in `My data` in a new window as shown in the following figure.



To set a lower limit for the boundary of the excluded region, click **Add Lower Limit**. This displays a vertical line on the left side of the plot window. Move the line with the mouse to the point you where you want the lower limit, as shown in the following figure.



Moving the vertical line changes the value displayed in the **Lower limit: exclude data** field in the Exclude window, as shown in the following figure.



Lower limit: exclude data <= -2.4589

Upper limit: exclude data >=

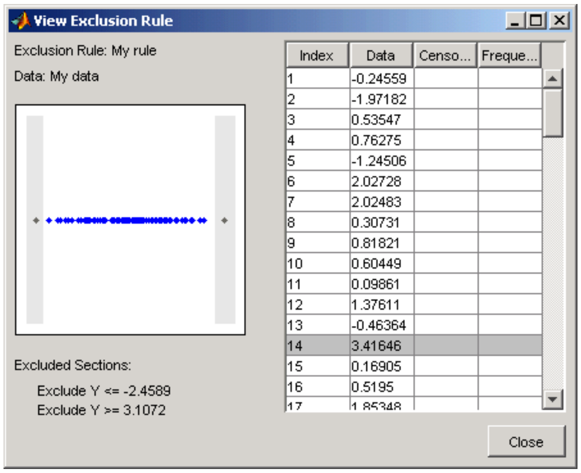
The value displayed corresponds to the *x*-coordinate of the vertical line.

Similarly, you can set the upper limit for the boundary of the excluded region by clicking **Add Upper Limit** and moving the vertical line that appears at the right side of the plot window. After setting the lower and upper limits, click **Close** and return to the Exclude window.

Create Exclusion Rule. Once you have set the lower and upper limits for the boundary of the excluded data, click **Create Exclusion Rule** to create the new rule. The name of the new rule now appears in the **Existing exclusion rules** pane.

When you select an exclusion rule in the **Existing exclusion rules** pane, the following buttons are enabled:

- **Copy** — Creates a copy of the rule, which you can then modify. To save the modified rule under a different name, click **Create Exclusion Rule**.
- **View** — Opens a new window in which you can see which data points are excluded by the rule. The following figure shows a typical example.



Exclusion Rule: My rule

Data: My data

Excluded Sections:

Exclude Y <= -2.4589

Exclude Y >= 3.1072

Index	Data	Censo...	Freque...
1	-0.24559		
2	-1.97182		
3	0.53547		
4	0.76275		
5	-1.24506		
6	2.02728		
7	2.02483		
8	0.30731		
9	0.81821		
10	0.60449		
11	0.09861		
12	1.37611		
13	-0.46364		
14	3.41646		
15	0.16905		
16	0.5195		
17	1.85348		

The shaded areas in the plot graphically display which data points are excluded. The table to the right lists all data points. The shaded rows indicate excluded points:

- **Rename** — Renames the rule
- **Delete** — Deletes the rule

Once you define an exclusion rule, you can use it when you fit a distribution to your data. The rule does not exclude points from the display of the data set.

Saving and Loading Sessions

This section explains how to save your work in the current Distribution Fitting Tool session and then load it in a subsequent session, so that you can continue working where you left off.

Saving a Session. To save the current session, select `Save Session` from the **File** menu in the main window. This opens a dialog box that prompts you to enter a filename, such as `my_session.dfit`, for the session. Clicking **Save** saves the following items created in the current session:

- Data sets
- Fits
- Exclusion rules
- Plot settings
- Bin width rules

Loading a Session. To load a previously saved session, select `Load Session` from the **File** menu in the main window and enter the name of a previously saved session. Clicking **Open** restores the information from the saved session to the current session of the Distribution Fitting Tool.

Generating an M-File to Fit and Plot Distributions

The `Generate M-file` option in the **File** menu enables you to create an M-file that

- Fits the distributions used in the current session to any data vector in the MATLAB workspace.
- Plots the data and the fits.

After you end the current session, you can use the M-file to create plots in a standard MATLAB figure window, without having to reopen the Distribution Fitting Tool.

As an example, assuming you created the fit described in "Creating a New Fit" on page 5-135, do the following steps:

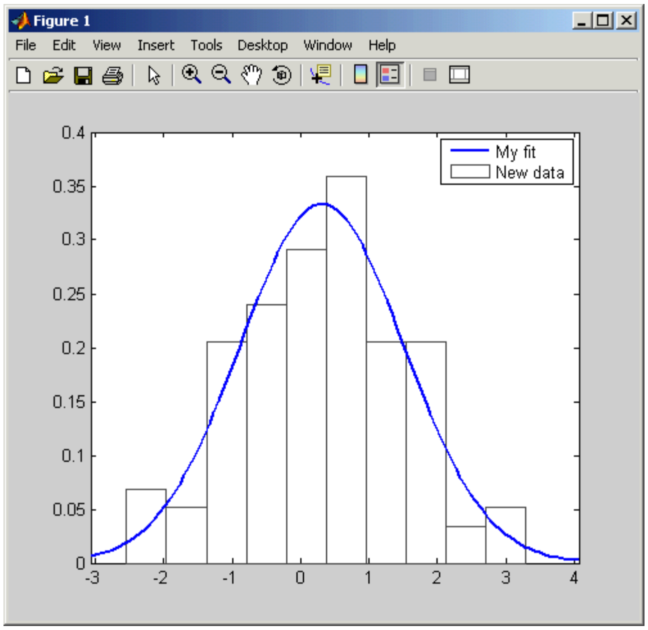
1 Select `Generate M-file` from the **File** menu.

2 Save the M-file as `normal_fit.m` in a directory on the MATLAB path.

You can then apply the function `normal_fit` to any vector of data in the MATLAB workspace. For example, the following commands

```
new_data = normrnd(4.1, 12.5, 100, 1);
normal_fit(new_data)
legend('New Data', 'My fit')
```

fit a normal distribution to a data set and generate a plot of the data and the fit.



Note By default, the M-file labels the data in the legend using the same name as the data set in the Distribution Fitting Tool. You can change the label using the `legend` command, as illustrated by the preceding example.

Using Custom Distributions

This section explains how to use custom distributions with the Distribution Fitting Tool.

Defining Custom Distributions. To define a custom distribution, select `Define Custom Distribution` from the **File** menu. This opens an M-file template in the MATLAB editor. You then edit this M-file so that it computes the distribution you want.

The template includes example code that computes the Laplace distribution, beginning at the lines

```
%                                  -
%    Remove the following return statement to define the
%    Laplace distributon
%                                  -
return
```

To use this example, simply delete the command `return` and save the M-file. If you save the template in a directory on the MATLAB path, under its default name `dfittooldists.m`, the Distribution Fitting Tool reads it in automatically when you start the tool. You can also save the template under a different name, such as `laplace.m`, and then import the custom distribution as described in the following section.

Importing Custom Distributions. To import a custom distribution, select `Import Custom Distributions` from the **File** menu. This opens a dialog box in which you can select the M-file that defines the distribution. For example, if you created the file `laplace.m`, as described in the preceding section, you can enter `laplace.m` and select **Open** in the dialog box. The **Distribution** field of the New Fit window now contains the option `Laplace`.

Additional Distributions Available in the Distribution Fitting Tool

The following distributions are available in the Distribution Fitting Tool, but do not have dedicated distribution functions as described in "Distribution Functions" on page 5-94. The distributions can be used with the functions `pdf`, `cdf`, `icdf`, and `mle` in a limited capacity. See the reference pages for these functions for details on the limitations.

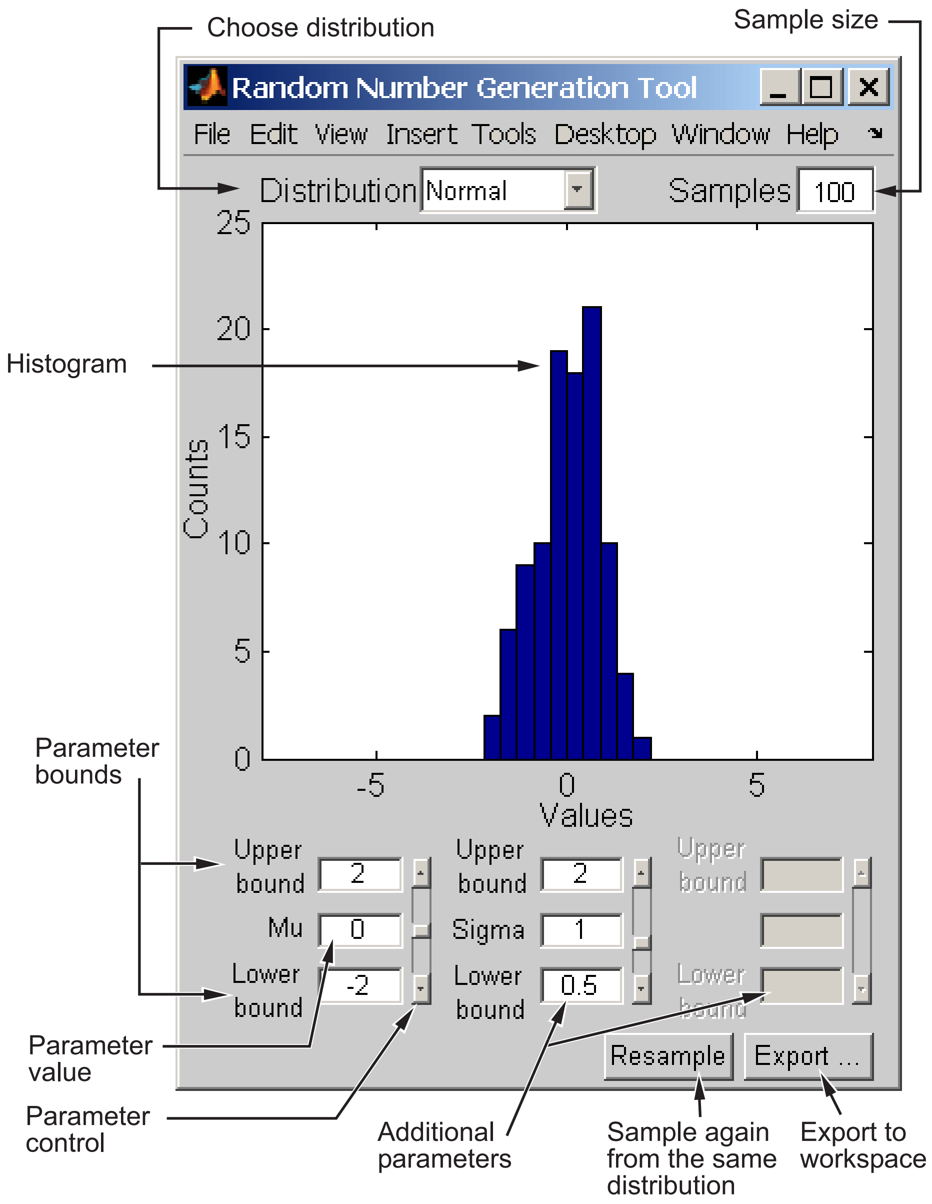
- "Birnbaum-Saunders Distribution" on page 5-18
- "Inverse Gaussian Distribution" on page 5-47
- "Loglogistic Distribution" on page 5-51
- "Logistic Distribution" on page 5-50
- "Nakagami Distribution" on page 5-64
- "Rician Distribution" on page 5-84
- "*t* Location-Scale Distribution" on page 5-87

For a complete list of the distributions available for use with the Distribution Fitting Tool, see "Supported Distributions" on page 5-3. Distributions listing `dfittool` in the **fit** column of the tables in that section can be used with the Distribution Fitting Tool.

Random Number Generation Tool

The Random Number Generation Tool is a graphical user interface that generates random samples from specified probability distributions and displays the samples as histograms. Use the tool to explore the effects of changing parameters and sample size on the distributions.

Run the tool by typing `randtool` at the command line. You can also run it from the `Demos` tab in the Help browser.



Distribution: Normal
Samples: 100
Upper bound: 2
Mu: 0
Lower bound: -2
Upper bound: 2
Sigma: 1
Lower bound: 0.5

Start by selecting a distribution, then enter the desired sample size.

You can also

- Use the controls at the bottom of the window to set parameter values for the distribution and to change their upper and lower bounds.
- Draw another sample from the same distribution, with the same size and parameters.
- Export the current sample to your workspace. A dialog box enables you to provide a name for the sample.

Random Number Generation

Random number generators for supported distributions are discussed in "Random Number Generators" on page 5-116.

A GUI for generating random numbers from supported distributions is discussed in "Random Number Generation Tool" on page 5-155.

This section discusses additional topics in random number generation.

Methods of Random Number Generation (p. 5-158)	Programming methods for random number generators
Additional Random Number Generators (p. 5-167)	Additional random number generators available in Statistics Toolbox
Copulas (p. 5-174)	Simulating dependent random variables using copulas

Methods of Random Number Generation

A working definition of *randomness* was given in 1951 by Berkeley Professor D. H. Lehmer, an early pioneer in computing:

A random sequence is a vague notion... in which each term is unpredictable to the uninitiated and whose digits pass a certain number of tests traditional with statisticians...

Mathematical definitions of randomness use notions of information content, noncomputability, and stochasticity, among other things. The various definitions, however, do not always agree on which sequences are random and which are not.

Practical methods for generating random numbers from specific distributions usually start with uniform random numbers. Once you have a uniform random number generator, like the MATLAB `rand` function, you can produce random numbers from other distributions using the methods described below.

Direct Methods

Direct methods make direct use of the definition of the distribution.

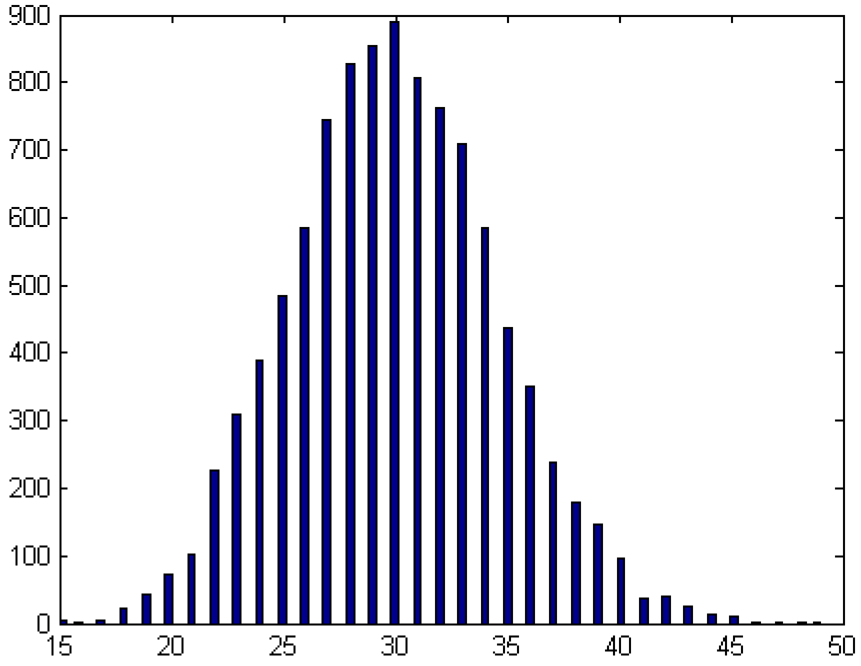
For example, consider binomial random numbers. You can think of a binomial random number as the number of heads in N tosses of a coin with probability p of a heads on any toss. If you generate N uniform random numbers on the interval (0,1) and count the number less than p, then the count is a binomial random number with parameters N and p.

The following function is a simple implementation of a binomial RNG using this approach:

```
function X = directbinornd(N,p,m,n)

X = zeros(m,n); % Preallocate memory
for i = 1:m*n
    u = rand(N,1);
    X(i) = sum(u < p);
end
```

For example,

```
X = directbinornd(100,0.3,1e4,1);
hist(X,101)
```



The Statistics Toolbox function `binornd` uses a modified direct method, based on the definition of a binomial random variable as the sum of Bernoulli random variables.

The method above is easily converted to a random number generator for the Poisson distribution with parameter λ. Recall that the Poisson distribution is the limiting case of the binomial distribution as N approaches infinity, p approaches zero, and Np is held fixed at λ. To generate Poisson random numbers, you could create a version of the above generator that inputs λ rather than N and p, and then internally sets N to some large number and p to λ/N.

The Statistics Toolbox function `poissrnd` actually uses two direct methods: a waiting time method for small values of λ, and a method due to Ahrens and Dieter for larger values of λ.

Inversion Methods

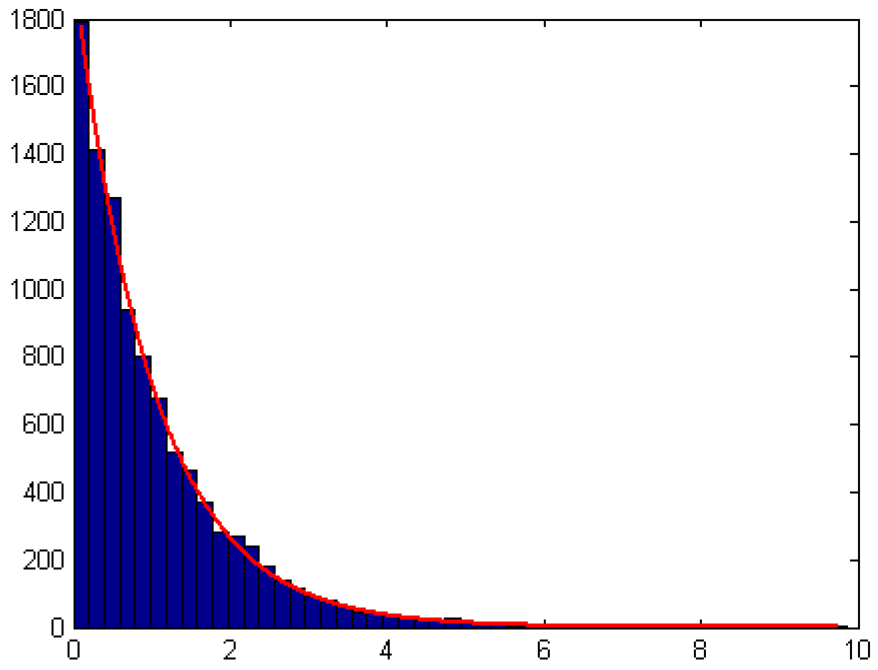
Inversion methods are based on the observation that continuous cumulative distribution functions (cdfs) range uniformly over the interval (0,1). If u is a uniform random number on (0,1), then a random number X from a continuous distribution with specified cdf F can be obtained using $X = F^{-1}(U)$.

For example, the following code generates random numbers from a specific exponential distribution using the inverse cdf and the MATLAB uniform random number generator `rand`:

```
mu = 1;
X = expinv(rand(1e4,1),mu);
```

The distribution of the generated random numbers can be compared to the pdf of the specified exponential distribution. The pdf, with area = 1, must be scaled to the area of the histogram used to display the distribution:

```
numbins = 50;
hist(X,numbins)
hold on
[bincounts,binpositions] = hist(X,numbins);
binwidth = binpositions(2) - binpositions(1);
histarea = binwidth*sum(bincounts);
x = binpositions(1):0.001:binpositions(end);
y = exppdf(x,mu);
plot(x,histarea*y,'r','LineWidth',2)
```



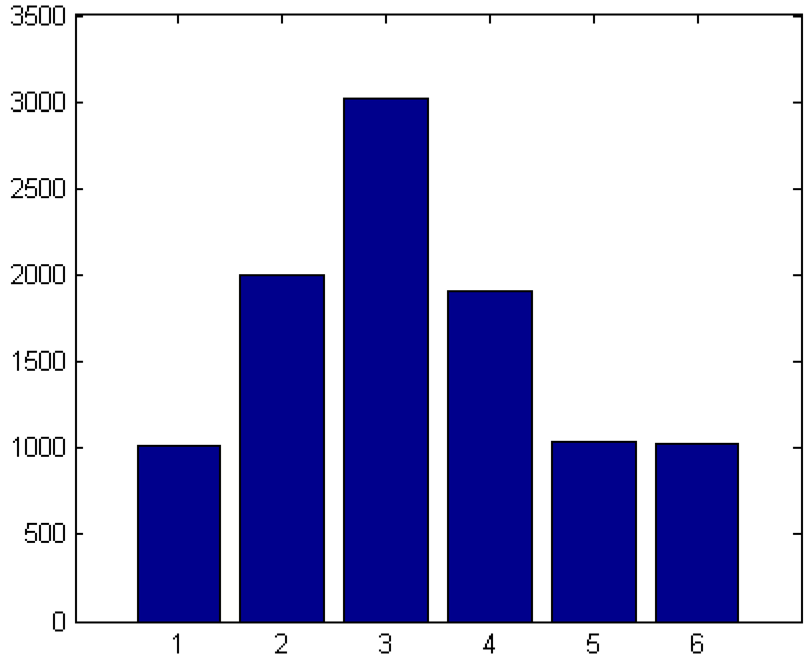
Inversion methods can be adapted to discrete distributions. Suppose you want a random number X from a discrete distribution with a probability mass vector $P(X = x_i) = p_i$, where $x_0 < x_1 < x_2 < \ldots$. You could generate a uniform random number u on (0,1) and then set $X = x_i$ if $F(x_{i-1}) < u < F(x_i)$.

For example, the following function implements an inversion method for a discrete distribution with probability mass vector p:

```
function X = discreteinvrnd(p,m,n)

X = zeros(m,n); % Preallocate memory
for i = 1:m*n
    u = rand;
    I = find(u < cumsum(p));
    X(i) = min(I);
end
```

The function can be used to generate random numbers from any discrete distribution:

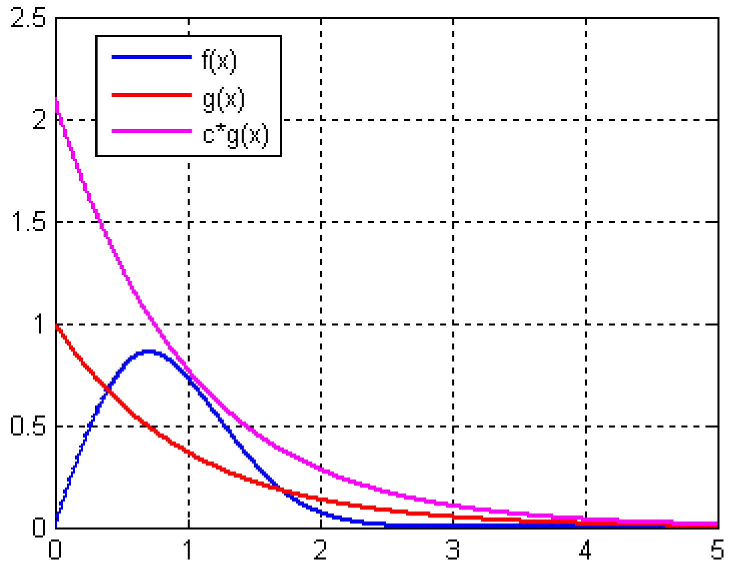
```
p = [0.1 0.2 0.3 0.2 0.1 0.1]; % Probability mass vector
X = discreteinvrnd(p,1e4,1);
[n,x] = hist(X,length(p));
bar(1:length(p),n)
```



Acceptance-Rejection Methods

The functional form of some distributions makes it difficult or time-consuming to generate random numbers using direct or inversion methods. Acceptance-rejection methods can provide a good solution in these cases.

Acceptance-rejection methods also begin with uniform random numbers, but they require the availability of an additional random number generator. If the goal is to generate a random number from a continuous distribution with pdf f, acceptance-rejection methods first generate a random number from a continuous distribution with pdf g satisfying $f(x) \le cg(x)$ for some c and all x.



A continuous acceptance-rejection RNG proceeds as follows:

1 Choose a density g.

2 Find a constant c such that $f(x) / g(x) \le c$ for all x.

3 Generate a uniform random number u.

4 Generate a random number v from g.

5 If $c^*u \le f(v) / g(v)$, accept and return v.

6 Otherwise, reject v and go to 3.

For efficiency, you need a cheap method for generating random numbers from g, and the scalar c should be small. The expected number of iterations to produce a random number is c.

The following function implements an acceptance-rejection method for generating random numbers from pdf f, given f, g, the RNG `grnd` for g, and the constant c:

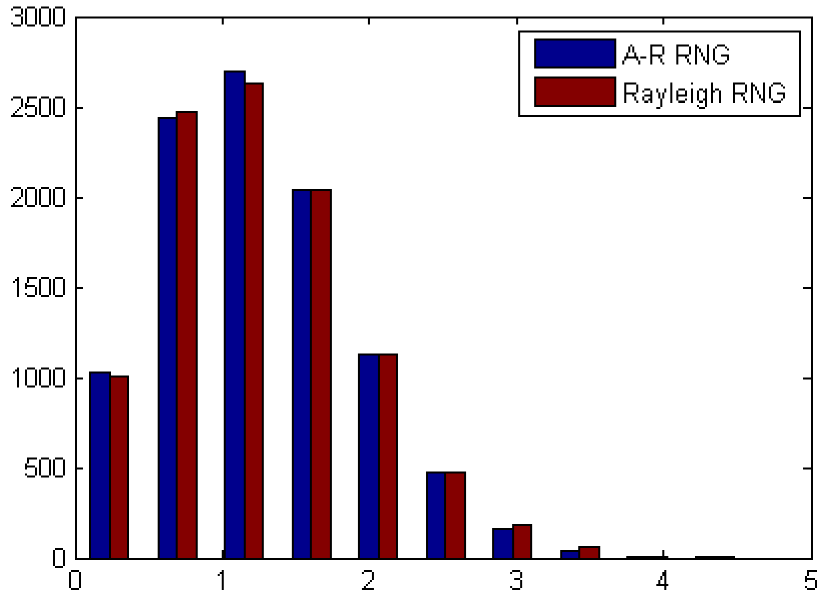
```
function X = accrejrnd(f,g,grnd,c,m,n)

X = zeros(m,n); % Preallocate memory
for i = 1:m*n
    accept = false;
    while accept == false
        u = rand();
        v = grnd();
        if c*u <= f(v)/g(v)
           X(i) = v;
           accept = true;
        end
    end
end
```

For example, the function $f(x) = xe^{-x^2/2}$ satisfies the conditions for a pdf on $[0,\infty)$ (nonnegative and integrates to 1). The exponential pdf with mean 1, $f(x) = e^{-x}$, dominates g for c greater than about 2.2. Thus, you can use `rand` and `exprnd` to generate random numbers from f:

```
f = @(x)x.*exp(-(x.^2)/2);
g = @(x)exp(-x);
grnd = @()exprnd(1);
X = accrejrnd(f,g,grnd,2.2,1e4,1);
```

The pdf f is actually a Rayleigh distribution with shape parameter 1. The distribution of random numbers generated by the acceptance-rejection method can be compared to those generated by `raylrnd`:

```
Y = raylrnd(1,1e4,1);
hist([X Y])
legend('A-R RNG','Rayleigh RNG')
```



The Statistics Toolbox function `raylrnd` uses a transformation method, expressing a Rayleigh random variable in terms of a chi-square random variable, which can be computed using `randn`.

Acceptance-rejection methods can be adapted to discrete distributions. In this case, the goal is to generate random numbers from a distribution with probability mass $P_p(X = i) = p_i$, assuming you have a method for generating random numbers from a distribution with probability mass $P_q(X = i) = q_i$. The RNG proceeds as follows:

1. Choose a density P_q.
2. Find a constant c such that $p_i / q_i \leq c$ for all i .
3. Generate a uniform random number u.
4. Generate a random number v from P_q.
5. If $c{*}u \leq p_v / q_v$, accept and return v.
6. Otherwise, reject v and go to 3.

Additional Random Number Generators

In addition to the direct, inverse, and acceptance-rejection methods described in "Methods of Random Number Generation" on page 5-158, Statistics Toolbox offers Markov chain Monte-Carlo methods and the Pearson and Johnson systems of distributions for generating random numbers from any distribution.

Markov Chain Samplers

In Bayesian data analysis, it is difficult to sample from the posterior distribution if it is in a nonstandard form. To generate random numbers for a nonstandard form, Markov chain algorithms draw dependent samples whose stationary distribution is the posterior distribution. Two algorithms are provided here—Metropolis-Hastings and slice sampling.

Metropolis-Hastings Algorithm. The Metropolis-Hastings algorithm draws samples from a distribution that is only known up to a constant. Random numbers are generated from a distribution with a probability density function that is equal to or proportional to a proposal function.

The following steps are used to generate random numbers:

1. Assume a initial value $x(t)$.
2. Draw a sample, $y(t)$, from a proposal distribution $q(y \mid x(t))$.
3. Accept $y(t)$ as the next sample $x(t+1)$ with probability $r(x(t),y(t))$, and keep $x(t)$ as the next sample $x(t+1)$ with probability $1-r(x(t),y(t))$, where

$$r(x,y)=min\left\{\frac{f(y)}{f(x)}\frac{q(x\mid y)}{q(y\mid x)},1\right\}$$

4. Increment $t \to t+1$, and repeat steps 2 and 3 until the desired number of samples are obtained.

You can generate random numbers using the Metropolis-Hastings method with the `mhsample` function. To produce quality samples efficiently with Metropolis-Hastings algorithm, it is crucial to select a good proposal distribution. If it is difficult to find an efficient proposal distribution, you can use the slice sampling algorithm without explicitly specifying a proposal distribution.

Slice Sampling Algorithm. In instances where it is difficult to find an efficient Metropolis-Hastings proposal distribution, there are a few algorithms, such as the slice sampling algorithm, that do not require an explicit specification for the proposal distribution. The slice sampling algorithm draws samples from the region under the density function using a sequence of vertical and horizontal steps. First, it selects a height at random between 0 and the density function $f(x)$. Then, it selects a new x value at random by sampling from the horizontal "slice" of the density above the selected height. A similar slice sampling algorithm is used for a multivariate distribution.

If a function $f(x)$ proportional to the density function is given, the following steps are used to generate random numbers:

1. Assume a initial value $x(t)$ within the domain of $f(x)$.

2. Draw a real value y uniformly from $(0,f(x(t)))$, thereby defining a horizontal "slice" as $S = \{x: y < f(x)\}$.

3. Find an interval $I = (L,R)$ around $x(t)$ that contains all, or much of the "slice" S.

4. Draw the new point $x(t+1)$ within this interval.

5. Increment $t \rightarrow t+1$ and repeat steps 2 through 4 until the desired number of samples are obtained.

Slice sampling can generate random numbers from a distribution with an arbitrary form of the density function, provided that an efficient numerical procedure is available to find the interval $I = (L,R)$, which is the "slice" of the density.

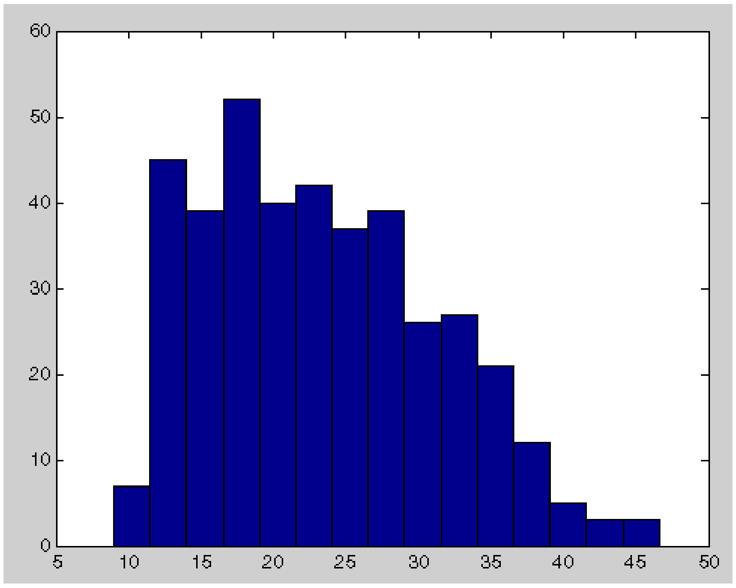
You can generate random numbers using the slice sampling method with the `slicesample` function.

Pearson and Johnson Systems of Distributions

In many simulation applications, you need to generate random inputs that are similar to existing data. One simple way to do that is to resample from the original data, using the `randsample` function. You might also fit a parametric distribution from one of the families described in the "Distribution Reference" on page 5-9, and then generate random values from that distribution. However, choosing a suitable family can sometimes be difficult. The Pearson and Johnson systems can help by making such a choice unnecessary. Each is a flexible parametric family of distributions that includes a wide range of distribution shapes, and it is often possible to find a distribution within one of these two systems that provides a good match to your data.

As an example, load the `carbig` dataset, which includes a variable `MPG` containing measurements of the gas mileage for each car.

```
load carbig
MPG = MPG(~isnan(MPG));
hist(MPG,15);
```



The Pearson System of Distributions. The statistician Karl Pearson devised a system, or family, of distributions that includes a unique distribution corresponding to every valid combination of mean, standard deviation, skewness, and kurtosis. If you compute sample values for each of these moments from data, it is easy to find the distribution in the Pearson system that matches these four moments and to generate a random sample.

The Pearson system embeds seven basic types of distribution together in a single parametric framework. It includes common distributions such as the normal and t distributions, simple transformations of standard distributions such as a shifted and scaled beta distribution and the inverse gamma distribution, and one distribution—the Type IV—that is not a simple transformation of any standard distribution.

For a given set of moments, there are distributions that are not in the system that also have those same first four moments, and the distribution in the Pearson system may not be a good match to your data, particularly if the data are multimodal. But the system does cover a wide range of distribution shapes, including both symmetric and skewed distributions.

To generate a sample from the Pearson distribution that closely matches the `MPG` data, simply compute the four sample moments and treat those as distribution parameters.

```
moments = {mean(MPG),std(MPG),skewness(MPG),kurtosis(MPG)};
[r,type] = pearsrnd(moments{:},10000,1);
```

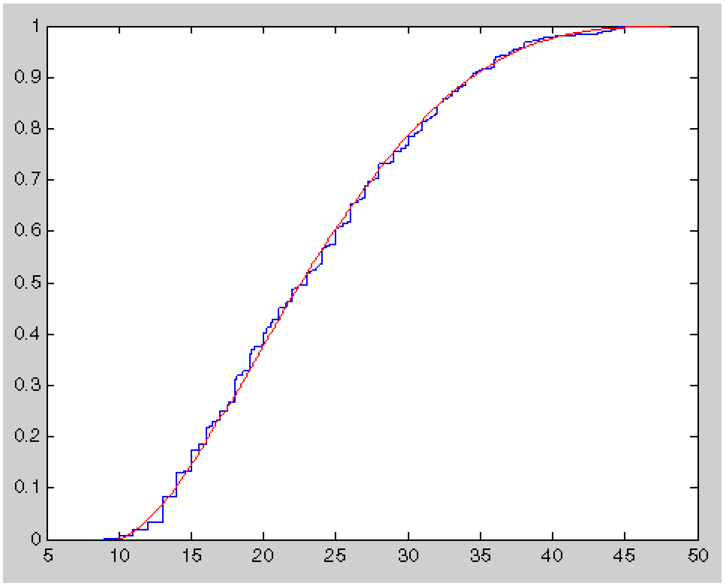
The optional second output from `pearsrnd` indicates which type of distribution within the Pearson system matches the combination of moments.

```
type

type =

     1
```

In this case, `pearsrnd` has determined that the data are best described with a Type I Pearson distribution, which is a shifted, scaled beta distribution.

Verify that the sample resembles the original data by overlaying the empirical cumulative distribution functions.

```
ecdf(MPG);
[Fi,xi] = ecdf(r);
hold on, stairs(xi,Fi,'r'); hold off
```



The Johnson System of Distributions. Statistician Norman Johnson devised a different system of distributions that also includes a unique distribution for every valid combination of mean, standard deviation, skewness, and kurtosis. However, since it is more natural to describe distributions in the Johnson system using quantiles, working with this system is different than working with the Pearson system.

The Johnson system is based on three possible transformations of a normal random variable, plus the identity transformation. The three nontrivial cases are known as `SL`, `SU`, and `SB`, corresponding to exponential, logistic, and hyperbolic sine transformations. All three can be written as

$$X = \gamma + \delta \cdot \Gamma(\frac{(Z-\xi)}{\lambda})$$

where Z is a standard normal random variable, Γ is the transformation, and γ, δ, ξ, and λ are scale and location parameters. The fourth case, `SN`, is the identity transformation.

To generate a sample from the Johnson distribution that matches the `MPG` data, first define the four quantiles to which the four evenly spaced standard normal quantiles of -1.5, -0.5, 0.5, and 1.5 should be transformed. That is, you compute the sample quantiles of the data for the cumulative probabilities of 0.067, 0.309, 0.691, and 0.933.

```
probs = normcdf([-1.5 -0.5 0.5 1.5])

probs =
     0.066807      0.30854      0.69146      0.93319

quantiles = quantile(MPG,probs)

quantiles =

   13.0000   18.0000   27.2000   36.0000
```

Then treat those quantiles as distribution parameters.

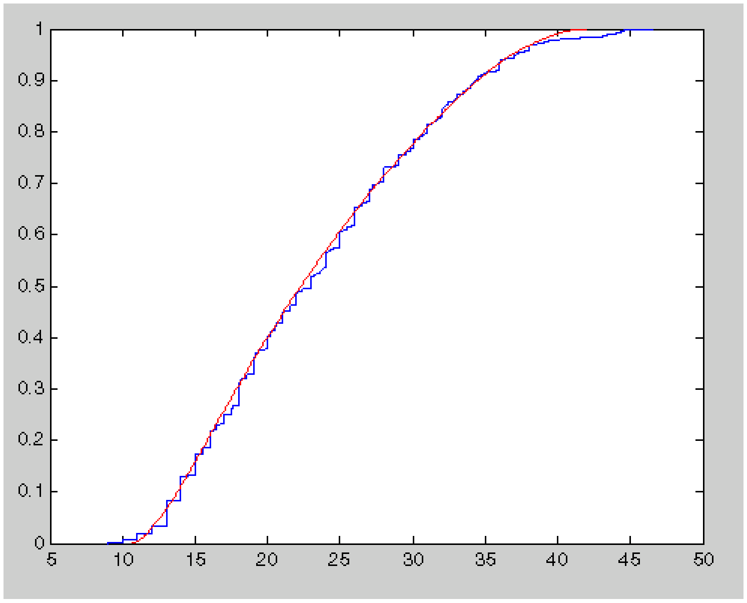
```
[r1,type] = johnsrnd(quantiles,10000,1);
```

The optional second output from `johnsrnd` indicates which type of distribution within the Johnson system matches the quantiles.

```
type

type =

SB
```

You can verify that the sample resembles the original data by overlaying the empirical cumulative distribution functions.

```
ecdf(MPG);
[Fi,xi] = ecdf(r1);
hold on, stairs(xi,Fi,'r'); hold off
```

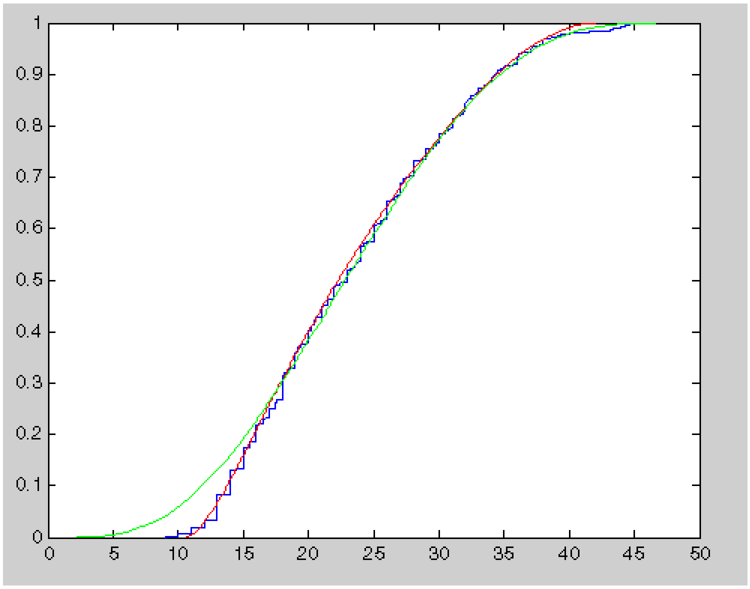


In some applications, it may be important to match the quantiles better in some regions of the data than in others. To do that, specify four evenly spaced standard normal quantiles at which you want to match the data, instead of the default -1.5, -0.5, 0.5, and 1.5. For example, you might care more about matching the data in the right tail than in the left, and so you would specify standard normal quantiles that emphasizes the right tail.

```
qnorm = [-.5 .25 1 1.75]
probs = normcdf(qnorm);
qemp = quantile(MPG,probs);
r2 = johnsrnd([qnorm; qemp],10000,1);

qnorm =
         -0.5          0.25             1          1.75
```

However, while the new sample matches the original data better in the right tail, it matches much worse in the left tail.

```
[Fj,xj] = ecdf(r2);
hold on, stairs(xj,Fj,'g'); hold off
```



Copulas

Statistics Toolbox provides functions to create sequences of random data according to many common univariate distributions. The toolbox also includes functions to generate random data from several multivariate distributions, such as `mvnrnd` for the multivariate normal and `mvtrnd` for the multivariate t. However, these standard multivariate distributions do not allow for cases with complicated relationships among the variables or where the individual variables are from different distributions.

Copulas are functions that describe dependencies among variables, and provide a way to create distributions to model correlated multivariate data. Using a copula, a data analyst can construct a multivariate distribution by specifying marginal univariate distributions, and then choose a particular copula to provide a correlation structure between variables. Bivariate distributions, as well as distributions in higher dimensions, are possible. This section discusses how to use copulas to generate dependent multivariate random data in MATLAB, using Statistics Toolbox.

Dependence Between Simulation Inputs

One of the design decisions for a Monte-Carlo simulation is a choice of probability distributions for the random inputs. Selecting a distribution for each individual variable is often straightforward, but deciding what dependencies should exist between the inputs may not be. Ideally, input data to a simulation should reflect what is known about dependence among the real quantities being modeled. However, there may be little or no information on which to base any dependence in the simulation. In such cases, it is useful to experiment with different possibilities in order to determine the model's sensitivity.

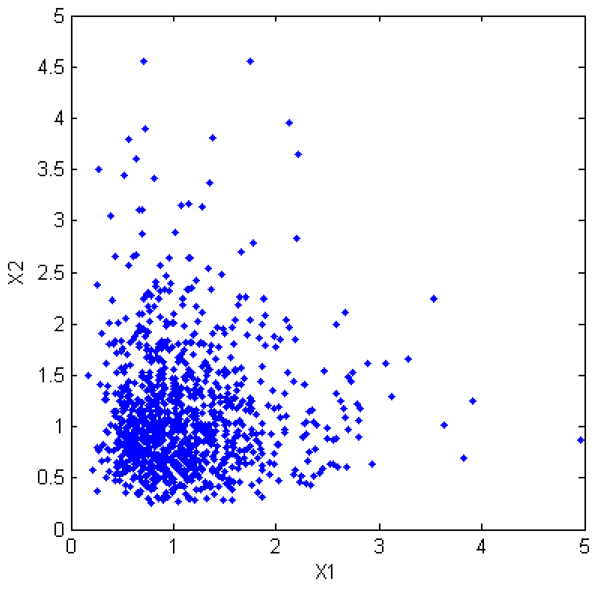
It can be difficult to actually generate random inputs with dependence when they have distributions that are not from a standard multivariate distribution. Further, some of the standard multivariate distributions can model only very limited types of dependence. It is always possible to make the inputs independent, and while that is a simple choice, it is not always sensible and can lead to the wrong conclusions.

For example, a Monte-Carlo simulation of financial risk might have two random inputs that represent different sources of insurance losses. These inputs might be modeled as lognormal random variables. A reasonable question to ask is how dependence between these two inputs affects the results of the simulation. Indeed, it might be known from real data that the same random conditions affect both sources and ignoring that in the simulation could lead to the wrong conclusions.

The `lognrnd` function is used to simulate independent lognormal random variables. In the example below, the `mvnrnd` function is used to generate `n` pairs of independent normal random variables, and then exponentiate them. Notice that the covariance matrix used here is diagonal, i.e., independence between the columns of `Z`.

```
n = 1000;
sigma = .5;
SigmaInd = sigma.^2 .* [1 0; 0 1]

SigmaInd =
        0.25           0
           0        0.25
```

```
ZInd = mvnrnd([0 0], SigmaInd, n);
XInd = exp(ZInd);
plot(XInd(:,1),XInd(:,2),'.'); axis equal; axis([0 5 0 5]);
xlabel('X1'); ylabel('X2');
```

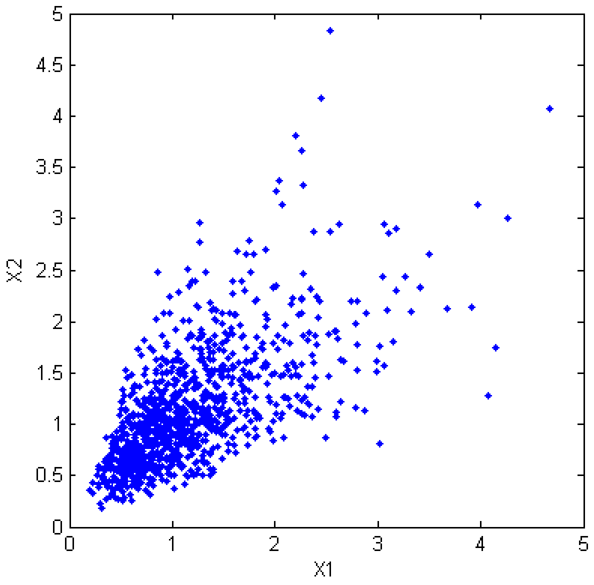


Dependent bivariate lognormal random variables are also easy to generate, using a covariance matrix with nonzero off-diagonal terms.

```
rho = .7;
SigmaDep = sigma.^2 .* [1 rho; rho 1]

SigmaDep =
          0.25         0.175
         0.175          0.25

ZDep = mvnrnd([0 0], SigmaDep, n);
XDep = exp(ZDep);
```

A second scatter plot demonstrates the difference between these two bivariate distributions.

```
plot(XDep(:,1),XDep(:,2),'.');
axis equal; axis([0 5 0 5]);
xlabel('X1'); ylabel('X2');
```



It is clear that there is a tendency in the second data set for large values of X1 to be associated with large values of X2, and similarly for small values. This dependence is determined by the correlation parameter, ρ, of the underlying bivariate normal. The conclusions drawn from the simulation could well depend on whether or not X1 and X2 were generated with dependence. The bivariate lognormal distribution is a simple solution in this case, and of course easily generalizes to higher dimensions in cases where the marginal distributions are different lognormals. Other multivariate distributions also exist. For example, the multivariate t and the Dirichlet distributions are used to simulate dependent t and beta random variables, respectively. But the list of simple multivariate distributions is not long, and they only apply in cases where the marginals are all in the same family (or even the exact same distributions). This can be a serious limitation in many situations.

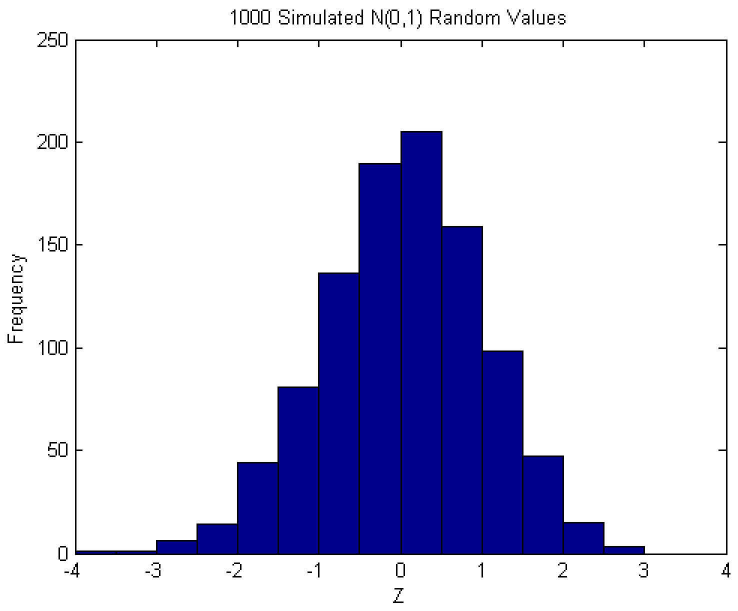
A More General Method for Constructing Dependent Bivariate Distributions

Although the construction discussed in the previous section creates a bivariate lognormal that is simple, it serves to illustrate a method which is more generally applicable. First, generate pairs of values from a bivariate normal distribution. There is statistical dependence between these two variables, and each has a normal marginal distribution. Next, apply a transformation (the exponential function) separately to each variable, changing the marginal distributions into lognormals. The transformed variables still have a statistical dependence.

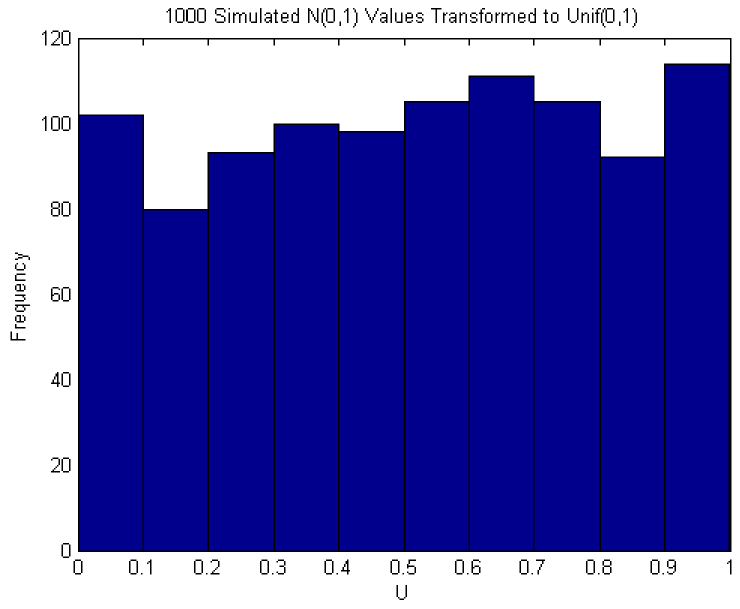
If a suitable transformation could be found, this method could be generalized to create dependent bivariate random vectors with other marginal distributions. In fact, a general method of constructing such a transformation does exist, although it is not as simple as exponentiation alone.

By definition, applying the normal cumulative distribution function (cdf), denoted here by Φ, to a standard normal random variable results in a random variable that is uniform on the interval [0, 1]. To see this, if `Z` has a standard normal distribution, then the cdf of `U` $= \Phi$`(Z)` is
`Pr{U` $\leq$ `u} = Pr{`Φ`(Z)` $\leq$ `u} = Pr{Z` $\leq$ Φ^{-1}`(u)} = u`,
and that is the cdf of a Unif(0,1) random variable. Histograms of some simulated normal and transformed values demonstrate that fact.

```
n = 1000;
z = normrnd(0,1,n,1);
hist(z,-3.75:.5:3.75); xlim([-4 4]);
title('1000 Simulated N(0,1) Random Values');
xlabel('Z'); ylabel('Frequency');
```

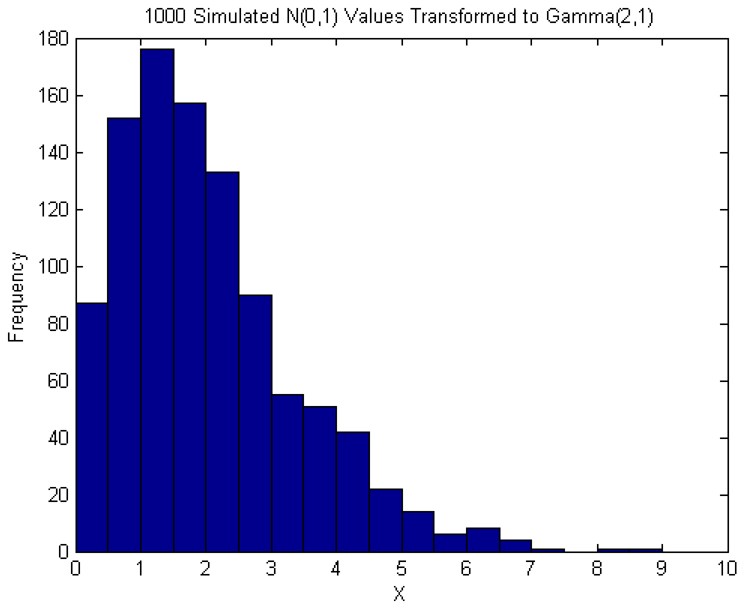


```
u = normcdf(z);
hist(u,.05:.1:.95);
title('1000 Simulated N(0,1) Values Transformed to Unif(0,1)');
xlabel('U'); ylabel('Frequency');
```



Borrowing from the theory of univariate random number generation, applying the inverse cdf of any distribution, F, to a Unif(0,1) random variable results in a random variable whose distribution is exactly F. This is known as the Inversion Method. The proof is essentially the opposite of the above proof for the forward case. Another histogram illustrates the transformation to a gamma distribution.

```
x = gaminv(u,2,1);
hist(x,.25:.5:9.75);
title('1000 Simulated N(0,1) Values Transformed to Gamma(2,1)');
xlabel('X'); ylabel('Frequency');
```



This two-step transformation can be applied to each variable of a standard bivariate normal, creating dependent random variables with arbitrary marginal distributions. Because the transformation works on each component separately, the two resulting random variables need not even have the same marginal distributions. The transformation is defined as

$$Z = [Z1\ Z2] \sim N([0\ 0], \begin{bmatrix} 1 & \rho \\ \rho & 1 \end{bmatrix})$$

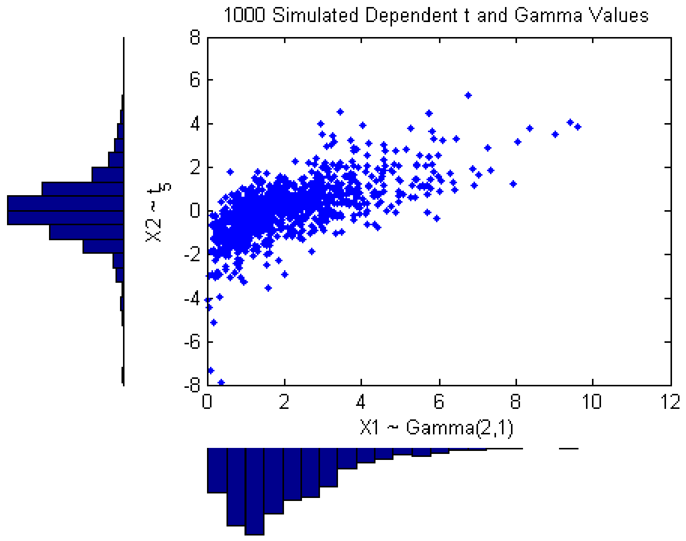
$$U = [\Phi(Z1)\ \Phi(Z2)]$$

$$X = [G1(U1)\ G2(U2)]$$

where `G1` and `G2` are inverse cdfs of two possibly different distributions. For example, you can generate random vectors from a bivariate distribution with t_5 and Gamma(2,1) marginals.

```
n = 1000; rho = .7; Z = mvnrnd([0 0],
[1 rho; rho 1], n); U = normcdf(Z);
X = [gaminv(U(:,1),2,1) tinv(U(:,2),5)];
```

This plot has histograms alongside a scatter plot to show both the marginal distributions, and the dependence.



Rank Correlation Coefficients

Dependence between X1 and X2 in this construction is determined by the correlation parameter, ρ, of the underlying bivariate normal. However, it is not true that the linear correlation of X1 and X2 is ρ. For example, in the original lognormal case, there is a closed form for that correlation:

$$\mathrm{cor}(X1,X2) = \frac{(e^{\rho\sigma^2} - 1)}{(e^{\sigma^2} - 1)}$$

which is strictly less than ρ, unless ρ is exactly one. In more general cases such as the Gamma/t construction above, the linear correlation between X1 and X2 is difficult or impossible to express in terms of ρ, but simulations can be used to show that the same effect happens.

That is because the linear correlation coefficient expresses the linear dependence between random variables, and when nonlinear transformations are applied to those random variables, linear correlation is not preserved. Instead, a rank correlation coefficient, such as Kendall's τ or Spearman's ρ, is more appropriate.

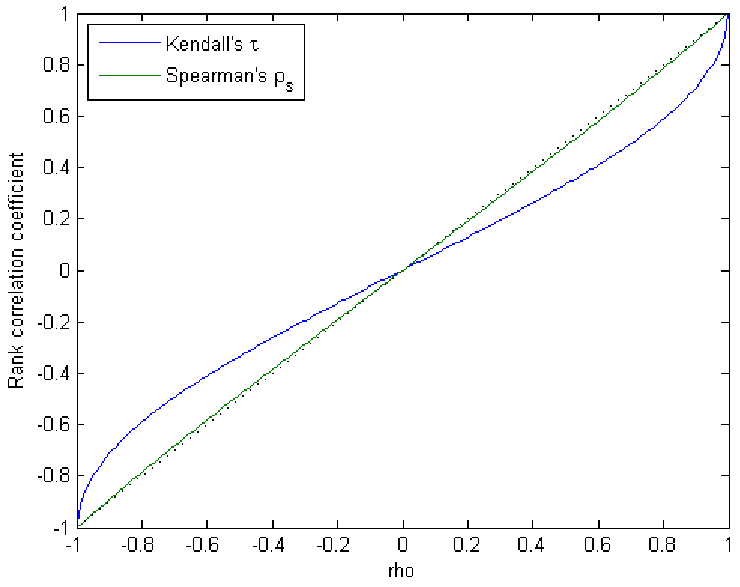
Roughly speaking, these rank correlations measure the degree to which large or small values of one random variable associate with large or small values of another. However, unlike the linear correlation coefficient, they measure the association only in terms of ranks. As a consequence, the rank correlation is preserved under any monotonic transformation. In particular, the transformation method just described preserves the rank correlation. Therefore, knowing the rank correlation of the bivariate normal `Z` exactly determines the rank correlation of the final transformed random variables, `X`. While the linear correlation coefficient, ρ, is still needed to parameterize the underlying bivariate normal, Kendall's τ or Spearman's ρ are more useful in describing the dependence between random variables, because they are invariant to the choice of marginal distribution.

It turns out that for the bivariate normal, there is a simple one-to-one mapping between Kendall's τ or Spearman's ρ, and the linear correlation coefficient ρ:

$$\tau = \frac{2}{\pi}\arcsin(\rho) \quad \text{or} \quad \rho = \sin(\tau\frac{\pi}{2})$$

$$\rho_s = \frac{6}{\pi}\arcsin(\frac{\rho}{2}) \quad \text{or} \quad \rho = 2\sin(\rho_s\frac{\pi}{6})$$

```
rho = -1:.01:1;
tau = 2.*asin(rho)./pi;
rho_s = 6.*asin(rho./2)./pi;
subplot(1,1,1);
plot(rho,tau,'-',rho,rho_s,'-',[-1 1],[-1 1],'k:');
axis([-1 1 -1 1]);
xlabel('rho');
ylabel('Rank correlation coefficient');
legend('Kendall''s \tau', ...
       'Spearman''s \rho_s', ...
       'location','northwest');
```



Thus, it is easy to create the desired rank correlation between `X1` and `X2`, regardless of their marginal distributions, by choosing the correct ρ parameter value for the linear correlation between `Z1` and `Z2`.

Notice that for the multivariate normal distribution, Spearman's rank correlation is almost identical to the linear correlation. However, this is not true once you transform to the final random variables.

Copulas

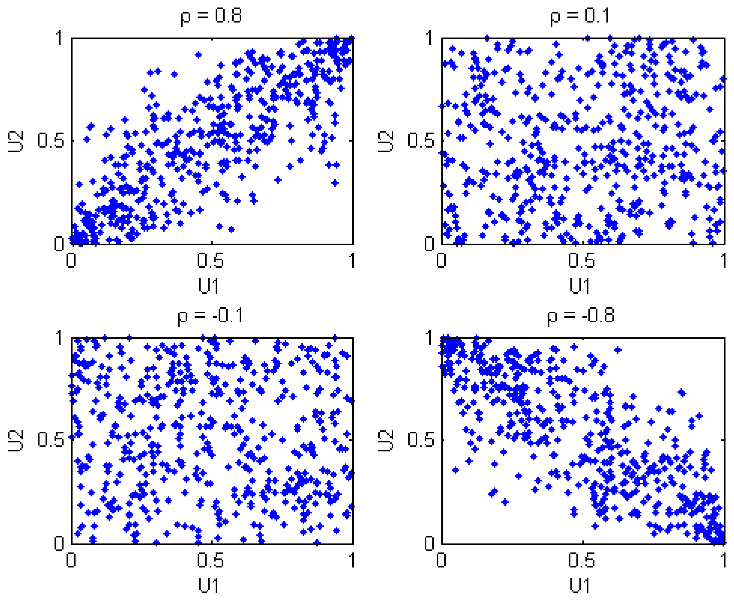
The first step of the construction described in the previous section defines what is known as a *copula*, specifically, a bivariate Gaussian copula. A copula is a multivariate probability distribution, where each random variable has a uniform marginal distribution on the unit interval [0,1]. These variables may be completely independent, deterministically related (e.g., `U2 = U1`), or anything in between. Because of the possibility for dependence among variables, you can use a copula to construct a new multivariate distribution for dependent variables. By transforming each of the variables in the copula separately using the inversion method, possibly using different cdfs, the resulting distribution can have arbitrary marginal distributions. Such multivariate distributions are often useful in simulations, when you know that the different random inputs are not independent of each other.

Statistics Toolbox provides functions to compute the probability density function (pdf) and the cumulative distribution function (cdf) for Gaussian copulas, functions to compute rank correlations from linear correlations and vice versa, and a function to generate random vectors. For example, use the `copularnd` function to create scatter plots of random values from a bivariate Gaussian copula for various levels of ρ, to illustrate the range of different dependence structures. The family of bivariate Gaussian copulas is parameterized by the linear correlation matrix:

$$P = \begin{pmatrix} 1 & \rho \\ \rho & 1 \end{pmatrix}$$

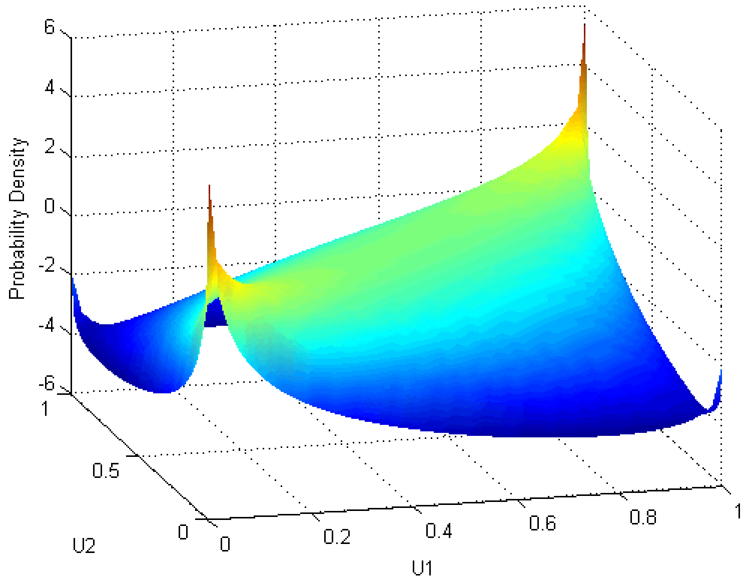
`U1` and `U2` approach linear dependence as ρ approaches ±1, and approach complete independence as ρ approaches zero.

```
n = 500;
U = copularnd('Gaussian',[1 .8; .8 1], n);
subplot(2,2,1); plot(U(:,1),U(:,2),'.');
title('\rho = 0.8'); xlabel('U1'); ylabel('U2');
U = copularnd('Gaussian', [1 .1; .1 1], n);
subplot(2,2,2); plot(U(:,1),U(:,2),'.');
title('\rho = 0.1'); xlabel('U1'); ylabel('U2');
U = copularnd('Gaussian', [1 -.1; -.1 1], n);
subplot(2,2,3); plot(U(:,1),U(:,2),'.');
title('\rho = -0.1'); xlabel('U1'); ylabel('U2');
U = copularnd('Gaussian', [1 -.8; -.8 1], n);
subplot(2,2,4); plot(U(:,1),U(:,2),'.');
title('\rho = -0.8'); xlabel('U1'); ylabel('U2');
```

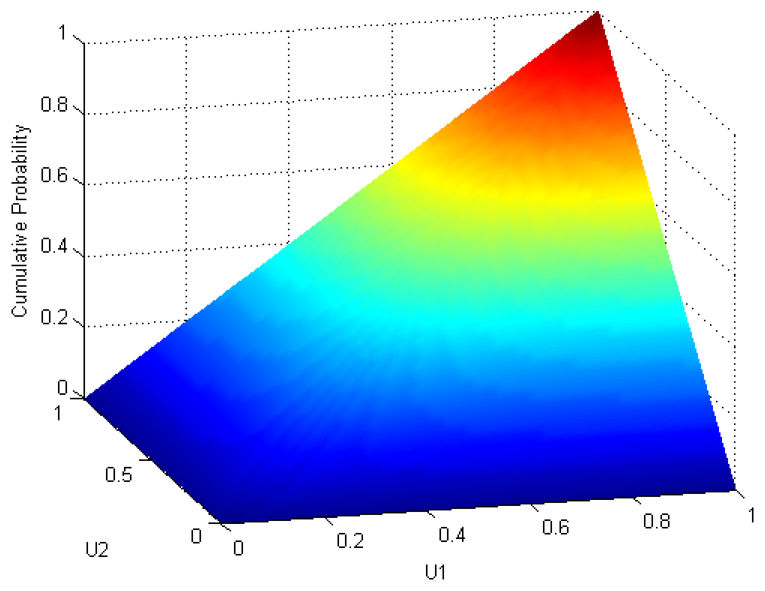


The dependence between U1 and U2 is completely separate from the marginal distributions of X1=G(U1) and X2 = G(U2). X1 and X2 can be given any marginal distributions, and still have the same rank correlation. This is one of the main appeals of copulas—they allow this separate specification of dependence and marginal distribution. You can also compute the pdf and the cdf for a copula. For example, these plots show the pdf and cdf for $\rho = .8$.

```
u1 = linspace(1e-3,1-1e-3,50);
u2 = linspace(1e-3,1-1e-3,50);
subplot(1,1,1);
[U1,U2] = meshgrid(u1,u2);
Rho = [1 .8; .8 1];
f = copulapdf('t',[U1(:) U2(:)],Rho,5);
f = reshape(f,size(U1));
surf(u1,u2,log(f),'FaceColor','interp','EdgeColor','none');
view([-15,20]);
xlabel('U1'); ylabel('U2'); zlabel('Probability Density');
```



```
u1 = linspace(1e-3,1-1e-3,50);
u2 = linspace(1e-3,1-1e-3,50);
[U1,U2] = meshgrid(u1,u2);
F = copulacdf('t',[U1(:) U2(:)],Rho,5);
F = reshape(F,size(U1));
surf(u1,u2,F,'FaceColor','interp','EdgeColor','none');
view([-15,20]);
xlabel('U1'); ylabel('U2'); zlabel('Cumulative Probability');
```



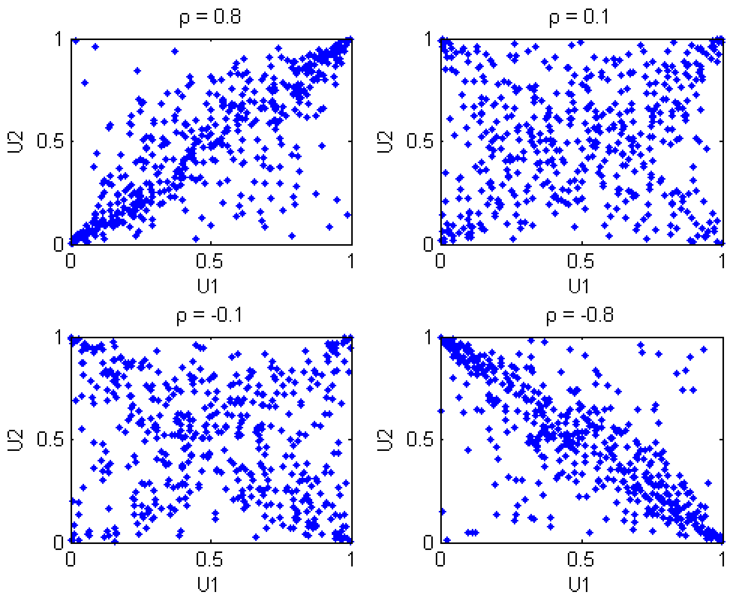
t Copulas

A different family of copulas can be constructed by starting from a bivariate t distribution, and transforming using the corresponding t cdf. The bivariate t distribution is parameterized with P, the linear correlation matrix, and ν, the degrees of freedom. Thus, for example, you can speak of a t_1 or a t_5 copula, based on the multivariate t with one and five degrees of freedom, respectively.

Just as for Gaussian copulas, Statistics Toolbox provides functions for t copulas to compute the pdf, cdf, and rank correlations; and to generate random vectors. For example, use the `copularnd` function to create scatter plots of random values from a bivariate t_1 copula for various levels of ρ, to illustrate the range of different dependence structures.

```
n = 500;
nu = 1;
U = copularnd('t', [1 .8; .8 1], nu, n);
subplot(2,2,1); plot(U(:,1),U(:,2),'.');
title('\rho = 0.8'); xlabel('U1'); ylabel('U2');
U = copularnd('t', [1 .1; .1 1], nu, n);
```

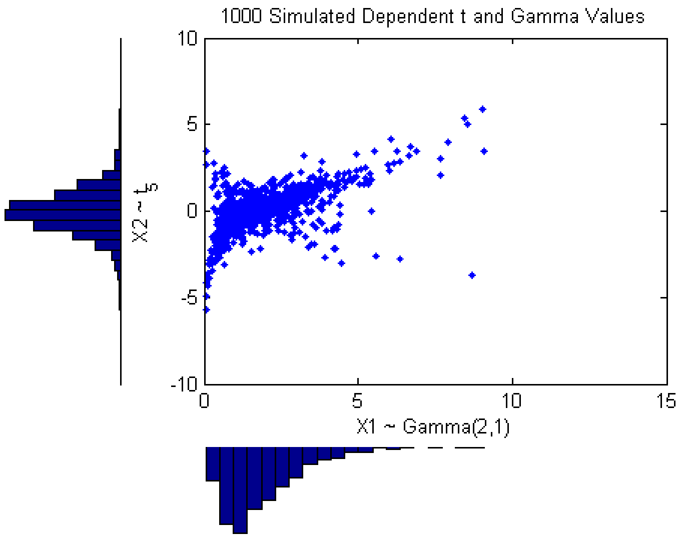
```
subplot(2,2,2); plot(U(:,1),U(:,2),'.');
title('\rho = 0.1'); xlabel('U1'); ylabel('U2');
U = copularnd('t', [1 -.1; -.1 1], nu, n);
subplot(2,2,3); plot(U(:,1),U(:,2),'.');
title('\rho = -0.1'); xlabel('U1'); ylabel('U2');
U = copularnd('t', [1 -.8; -.8 1], nu, n);
subplot(2,2,4); plot(U(:,1),U(:,2),'.');
title('\rho = -0.8'); xlabel('U1'); ylabel('U2');
```



A t copula has uniform marginal distributions for `U1` and `U2`, just as a Gaussian copula does. The rank correlation τ or ρ_s between components in a t copula is also the same function of ρ as for a Gaussian. However, as these plots demonstrate, a t_1 copula differs quite a bit from a Gaussian copula, even when their components have the same rank correlation. The difference is in their dependence structure. Not surprisingly, as the degrees of freedom parameter ν is made larger, a t_ν copula approaches the corresponding Gaussian copula.

As with a Gaussian copula, any marginal distributions can be imposed over a t copula. For example, using a t copula with 1 degree of freedom, you can again generate random vectors from a bivariate distribution with Gamma(2,1) and t_5 marginals:

```
n = 1000;
rho = .7;
nu = 1;
U = copularnd('t', [1 rho; rho 1], nu, n);
X = [gaminv(U(:,1),2,1) tinv(U(:,2),5)];
```

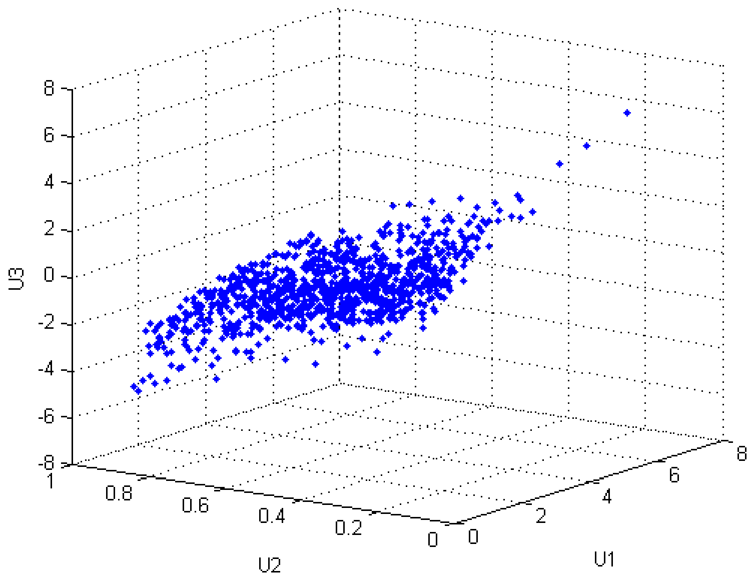


Compared to the bivariate Gamma/*t* distribution constructed earlier, which was based on a Gaussian copula, the distribution constructed here, based on a t_1 copula, has the same marginal distributions and the same rank correlation between variables but a very different dependence structure. This illustrates the fact that multivariate distributions are not uniquely defined by their marginal distributions, or by their correlations. The choice of a particular copula in an application may be based on actual observed data, or different copulas may be used as a way of determining the sensitivity of simulation results to the input distribution.

Copulas in Higher Dimensions

The Gaussian and t copulas are known as elliptical copulas. It is easy to generalize elliptical copulas to a higher number of dimensions. For example, simulate data from a trivariate distribution with Gamma(2,1), Beta(2,2), and t_5 marginals using a Gaussian copula as follows:

```
n = 1000;
Rho = [1 .4 .2; .4 1 -.8; .2 -.8 1];
U = copularnd('Gaussian', Rho, n);
X = [gaminv(U(:,1),2,1) betainv(U(:,2),2,2) tinv(U(:,3),5)];
subplot(1,1,1);
plot3(X(:,1),X(:,2),X(:,3),'.');
grid on; view([-55, 15]);
xlabel('X1'); ylabel('X2'); zlabel('X3');
```



Notice that the relationship between the linear correlation parameter ρ and, for example, Kendall's τ, holds for each entry in the correlation matrix P used here. You can verify that the sample rank correlations of the data are approximately equal to the theoretical values.

```
tauTheoretical = 2.*asin(Rho)./pi
tauTheoretical =
           1      0.26198      0.12819
     0.26198            1     -0.59033
     0.12819     -0.59033            1

tauSample = corr(X, 'type','Kendall')

tauSample =
           1      0.27254      0.12701
     0.27254            1     -0.58182
     0.12701     -0.58182            1
```

Archimedean Copulas

Statistics Toolbox also supports three bivariate Archimedean copula families: the Clayton, the Frank, and the Gumbel. These are one-parameter families that are defined directly in terms of their cdfs, rather than being defined constructively using a standard multivariate distribution.

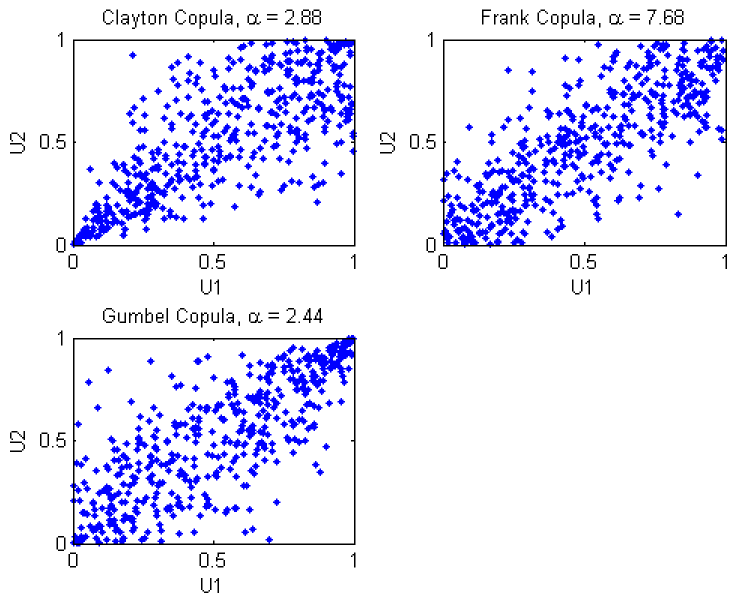
To compare these three Archimedean copulas to the Gaussian and *t* bivariate copulas, first use the `copulastat` function to find the rank correlation for a Gaussian or t copula with linear correlation parameter of 0.8, then use the `copulaparam` function to find the Clayton copula parameter that corresponds to that rank correlation.

```
tau = copulastat('Gaussian', .8 ,'type', 'kendall')

tau =
     0.59033

alpha = copulaparam('Clayton', tau, 'type', 'kendall')

alpha =
        2.882
```

Finally, plot a random sample from the Clayton copula. Repeat the same procedure for the Frank and Gumbel copulas.

```
n = 500;
U = copularnd('Clayton', alpha, n);
```

```
subplot(2,2,1); plot(U(:,1),U(:,2),'.');
title(sprintf('Clayton Copula, \\alpha = %.2f',alpha)); ...
      xlabel('U1'); ylabel('U2');
alpha = copulaparam('Frank', tau, 'type', 'kendall');
U = copularnd('Frank', alpha, n);
subplot(2,2,2); plot(U(:,1),U(:,2),'.');
title(sprintf('Frank Copula, \\alpha = %.2f',alpha)); ...
      xlabel('U1'); ylabel('U2');
alpha = copulaparam('Gumbel', tau, 'type', 'kendall');
U = copularnd('Gumbel', alpha, n);
subplot(2,2,3); plot(U(:,1),U(:,2),'.');
title(sprintf('Gumbel Copula, \\alpha = %.2f',alpha)); ...
      xlabel('U1'); ylabel('U2');
```



Copulas and Nonparametric Marginal Distributions

To simulate dependent multivariate data using a copula, you must specify each of the following:

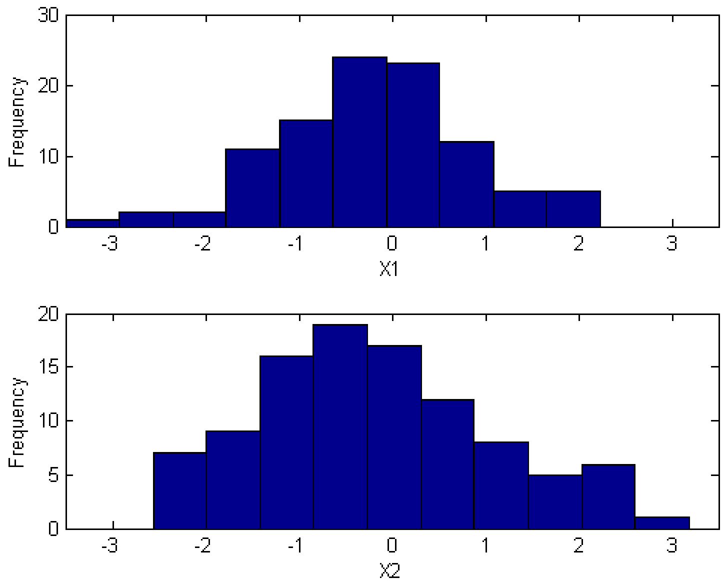
1 The copula family (and any shape parameters)

2 The rank correlations among variables

3 Marginal distributions for each variable

Suppose you have return data for two stocks, and would like to run a Monte Carlo simulation with inputs that follow the same distributions as the data.

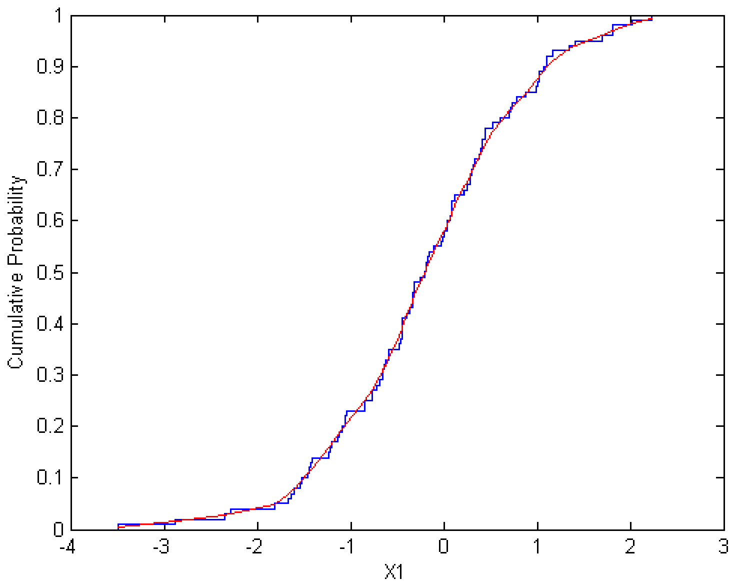
```
load stockreturns
nobs = size(stocks,1);
subplot(2,1,1); hist(stocks(:,1),10); xlim([-3.5 3.5]);
xlabel('X1'); ylabel('Frequency');
subplot(2,1,2); hist(stocks(:,2),10); xlim([-3.5 3.5]);
xlabel('X2'); ylabel('Frequency');
```



You could fit a parametric model separately to each dataset, and use those estimates as the marginal distributions. However, a parametric model may not be sufficiently flexible. Instead, you can use an nonparametric model to transform to the marginal distributions. All that is needed is a way to compute the inverse cdf for the nonparametric model.

The simplest nonparametric model is the empirical cdf, as computed by the `ecdf` function. For a discrete marginal distribution, this is appropriate. However, for a continuous distribution, it is a good idea to use a model that is smoother than the step function computed by `ecdf`. One way to do that is to estimate the empirical cdf, and interpolate between the midpoints of the steps with a piecewise linear function. Another way is to use kernel smoothing. For example, compare the empirical cdf to a kernel smoothed cdf estimate for the first variable.

```
[Fi,xi] = ecdf(stocks(:,1));
subplot(1,1,1);
stairs(xi,Fi,'b');
hold on
Fi_sm = ksdensity(stocks(:,1),xi,'function','cdf','width',.15);
plot(xi,Fi_sm,'r-');
hold off
xlabel('X1'); ylabel('Cumulative Probability');
```



For the simulation, you might want to experiment with different copulas and correlations. Here, you'll use a bivariate *t* copula with a fairly small degrees of freedom parameter. For the correlation parameter, you can compute the rank correlation of the data, and then find the corresponding linear correlation parameter for the *t* copula.

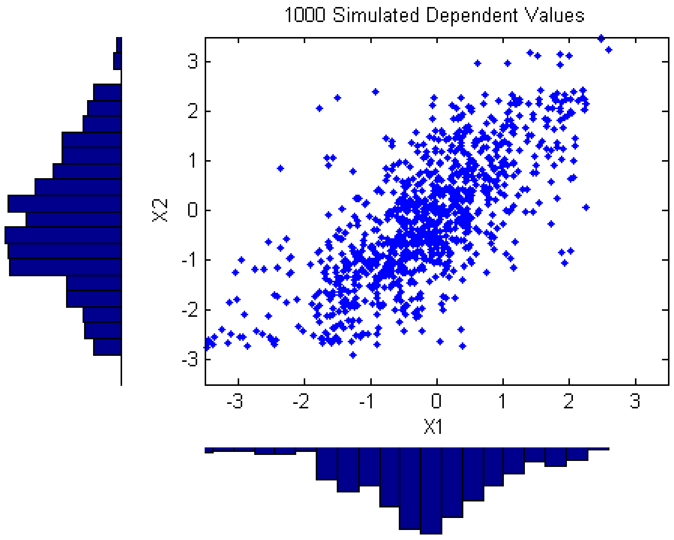
```
nu = 5;
tau = corr(stocks(:,1),stocks(:,2),'type','kendall')

tau =
      0.51798

rho = copulaparam('t', tau, nu, 'type','kendall')

rho =
      0.72679
```

Next, generate random values from the *t* copula, and transform using the nonparametric inverse cdfs. The `ksdensity` function allows you to make a kernel estimate of distribution, and evaluate the inverse cdf at the copula points all in one step.

```
n = 1000;
U = copularnd('t',[1 rho; rho 1],nu,n);
X1 = ksdensity(stocks(:,1),U(:,1),...
               'function','icdf','width',.15);
X2 = ksdensity(stocks(:,2),U(:,2),...
               'function','icdf','width',.15);
```

Alternatively, when you have a large amount of data or need to simulate more than one set of values, it may be more efficient to compute the inverse cdf over a grid of values in the interval (0,1), and use interpolation to evaluate it at the copula points.

```
p = linspace(.00001, .99999, 1000);
G1 = ksdensity(stocks(:,1) ,p ,'function', 'icdf', 'width',.15);
X1 = interp1(p, G1, U(:,1), 'spline');
G2 = ksdensity(stocks(:,2) ,p ,'function', 'icdf', 'width',.15);
X2 = interp1(p, G2, U(:,2), 'spline');
```



Notice that the marginal histograms of the simulated data are a smoothed version of the histograms for the original data. The amount of smoothing is controlled by the bandwidth input to `ksdensity`.

Hypothesis Tests

Introduction

Hypothesis testing is a common method of drawing inferences about a population based on statistical evidence from a sample.

As an example, suppose someone says that at a certain time in the state of Massachusetts the average price of a gallon of regular unleaded gas was \$1.15. How could you determine the truth of the statement? You could try to find prices at every gas station in the state at the time. That approach would be definitive, but it could be time-consuming, costly, or even impossible.

A simpler approach would be to find prices at a small number of randomly selected gas stations around the state, and then compute the sample average.

Sample averages differ from one another due to chance variability in the selection process. Suppose your sample average comes out to be \$1.18. Is the \$0.03 difference an artifact of random sampling or significant evidence that the average price of a gallon of gas was in fact greater than \$1.15? Hypothesis testing is a statistical method for making such decisions.

Hypothesis Test Terminology

All hypothesis tests share the same basic terminology and structure.

- A *null hypothesis* is an assertion about a population that you would like to test. It is "null" in the sense that it often represents a status quo belief, such as the absence of a characteristic or the lack of an effect. It may be formalized by asserting that a population parameter, or a combination of population parameters, has a certain value. In the example given in the "Introduction" on page 6-2, the null hypothesis would be that the average price of gas across the state was \$1.15. This is written H_0: $\mu = 1.15$.
- An *alternative hypothesis* is a contrasting assertion about the population that can be tested against the null hypothesis. In the example given in the "Introduction" on page 6-2, possible alternative hypotheses are:

  H_1: $\mu \neq 1.15$ — State average was different from \$1.15 (two-tailed test)

  H_1: $\mu > 1.15$ — State average was greater than \$1.15 (right-tail test)

  H_1: $\mu < 1.15$ — State average was less than \$1.15 (left-tail test)
- To conduct a hypothesis test, a random sample from the population is collected and a relevant *test statistic* is computed to summarize the sample. This statistic varies with the type of test, but its distribution under the null hypothesis must be known (or assumed).
- The *p-value* of a test is the probability, under the null hypothesis, of obtaining a value of the test statistic as extreme or more extreme than the value computed from the sample.
- The *significance level* of a test is a threshold of probability α agreed to before the test is conducted. A typical value of α is 0.05. If the p-value of a test is less than α, the test rejects the null hypothesis. If the p-value is greater than α, there is insufficient evidence to reject the null hypothesis. Note that lack of evidence for rejecting the null hypothesis is not evidence for accepting the null hypothesis. Also note that substantive "significance" of an alternative cannot be inferred from the statistical significance of a test.
- The significance level α can be interpreted as the probability of rejecting the null hypothesis when it is actually true—a *type I error*. The distribution of the test statistic under the null hypothesis determines the probability α of a type I error. Even if the null hypothesis is not rejected, it may still be false—a *type II error*. The distribution of the test statistic under the

alternative hypothesis determines the probability β of a type II error. Type II errors are often due to small sample sizes. The *power* of a test, $1 - \beta$, is the probability of correctly rejecting a false null hypothesis.

- Results of hypothesis tests are often communicated with a *confidence interval*. A confidence interval is an estimated range of values with a specified probability of containing the true population value of a parameter. Upper and lower bounds for confidence intervals are computed from the sample estimate of the parameter and the known (or assumed) sampling distribution of the estimator. A typical assumption is that estimates will be normally distributed with repeated sampling (as dictated by the Central Limit Theorem). Wider confidence intervals correspond to poor estimates (smaller samples); narrow intervals correspond to better estimates (larger samples). If the null hypothesis asserts the value of a population parameter, the test rejects the null hypothesis when the hypothesized value lies outside the computed confidence interval for the parameter.

Hypothesis Test Assumptions

Different hypothesis tests make different assumptions about the distribution of the random variable being sampled in the data. These assumptions must be considered when choosing a test and when interpreting the results.

For example, the *z*-test (`ztest`) and the *t*-test (`ttest`) both assume that the data are independently sampled from a normal distribution. Statistics Toolbox offers a number of functions for testing this assumption, such as `chi2gof`, `jbtest`, `lillietest`, and `normplot`.

Both the *z*-test and the *t*-test are relatively robust with respect to departures from this assumption, so long as the sample size *n* is large enough. Both tests compute a sample mean $\bar{x}$, which, by the Central Limit Theorem, has an approximately normal sampling distribution with mean equal to the population mean μ, regardless of the population distribution being sampled.

The difference between the *z*-test and the *t*-test is in the assumption of the standard deviation σ of the underlying normal distribution. A *z*-test assumes that σ is known; a *t*-test does not. As a result, a *t*-test must compute an estimate *s* of the standard deviation from the sample.

Test statistics for the *z*-test and the *t*-test are, respectively,

$$z = \frac{\bar{x} - \mu}{\sigma / \sqrt{n}}$$

$$t = \frac{\bar{x} - \mu}{s / \sqrt{n}}$$

Under the null hypothesis that the population is distributed with mean μ, the *z*-statistic has a standard normal distribution, *N*(0,1). Under the same null hypothesis, the *t*-statistic has Student's *t* distribution with $n - 1$ degrees of freedom. For small sample sizes, Student's *t* distribution is flatter and wider than *N*(0,1), compensating for the decreased confidence in the estimate *s*. As sample size increases, however, Student's *t* distribution approaches the standard normal distribution, and the two tests become essentially equivalent.

Knowing the distribution of the test statistic under the null hypothesis allows for accurate calculation of *p*-values. Interpreting *p*-values in the context of the test assumptions allows for critical analysis of test results.

Assumptions underlying each of the hypothesis tests in Statistics Toolbox are given in the reference page for the implementing function.

Example: Hypothesis Testing

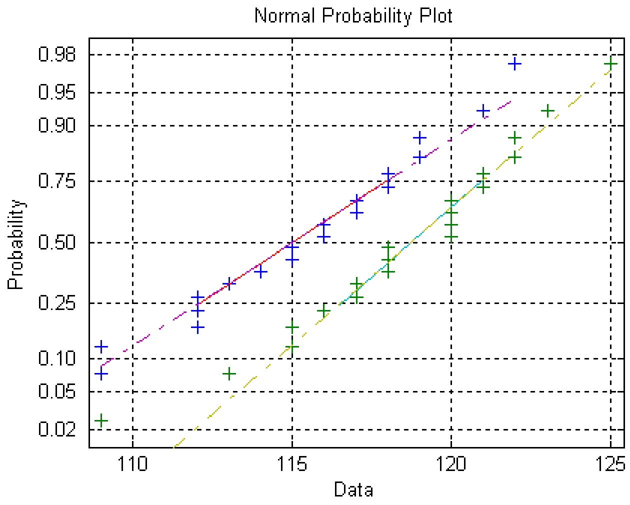
This example uses the gas price data in the file `gas.mat`. The file contains two random samples of prices for a gallon of gas around the state of Massachusetts in 1993. The first sample, `price1`, contains 20 random observations around the state on a single day in January. The second sample, `price2`, contains 20 random observations around the state one month later.

```
load gas
prices = [price1 price2];
```

As a first step, you might want to test the assumption that the samples come from normal distributions.

A normal probability plot gives a quick idea.

```
normplot(prices)
```



Both scatters approximately follow straight lines through the first and third quartiles of the samples, indicating approximate normal distributions. The February sample (the right-hand line) shows a slight departure from normality in the lower tail. A shift in the mean from January to February is evident.

A hypothesis test can be used to quantify the test of normality. Since each sample is relatively small, a Lilliefors test is recommended.

```
lillietest(price1)
ans =
     0
lillietest(price2)
ans =
     0
```

The default significance level of `lillietest` is 5%. The logical 0 returned by each test indicates a failure to reject the null hypothesis that the samples are normally distributed. This failure may reflect normality in the population or it may reflect a lack of strong evidence against the null hypothesis due to the small sample size.

Now compute the sample means:

```
sample_means = mean(prices)
sample_means =
  115.1500  118.5000
```

You might want to test the null hypothesis that the mean price across the state on the day of the January sample was $1.15. If you know that the standard deviation in prices across the state has historically, and consistently, been $0.04, then a *z*-test is appropriate.

```
[h,pvalue,ci] = ztest(price1/100,1.15,0.04)
h =
     0
pvalue =
    0.8668
ci =
    1.1340    1.1690
```

The logical output `h` = 0 indicates a failure to reject the null hypothesis at the default significance level of 5%. This is a consequence of the high probability under the null hypothesis, indicated by the *p*-value, of observing a value as extreme or more extreme of the *z*-statistic computed from the sample. The 95% confidence interval on the mean [1.1340 1.1690] includes the hypothesized population mean of $1.15.

Does the later sample offer stronger evidence for rejecting a null hypothesis of a state-wide average price of $1.15 in February? The shift shown in the probability plot and the difference in the computed sample means suggest this. The shift might indicate a significant fluctuation in the market, raising questions about the validity of using the historical standard deviation. If a known standard deviation cannot be assumed, a *t*-test is more appropriate.

```
[h,pvalue,ci] = ttest(price2/100,1.15)
h =
     1
pvalue =
   4.9517e-04
ci =
    1.1675    1.2025
```

The logical output `h = 1` indicates a rejection of the null hypothesis at the default significance level of 5%. In this case, the 95% confidence interval on the mean does not include the hypothesized population mean of $1.15.

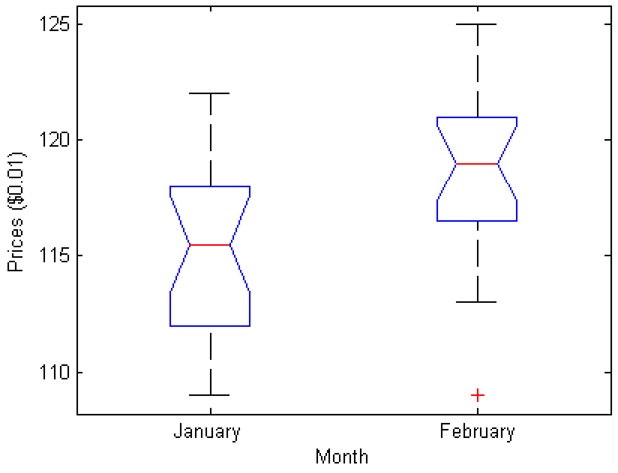
You might want to investigate the shift in prices a little more closely. The function `ttest2` tests if two independent samples come from normal distributions with equal but unknown standard deviations and the same mean, against the alternative that the means are unequal.

```
[h,sig,ci] = ttest2(price1,price2)
h =
     1
sig =
    0.0083
ci =
   -5.7845   -0.9155
```

The null hypothesis is rejected at the default 5% significance level, and the confidence interval on the difference of means does not include the hypothesized value of 0.

A notched box plot is another way to visualize the shift.

```
boxplot(prices,1)
set(gca,'XtickLabel',str2mat('January','February'))
xlabel('Month')
ylabel('Prices ($0.01)')
```



The plot displays the distribution of the samples around their medians. The heights of the notches in each box are computed so that the side-by-side boxes have nonoverlapping notches when their medians are different at a default 5% significance level. The computation is based on an assumption of normality in the data, but the comparison is reasonably robust for other distributions. The side-by-side plots provide a kind of visual hypothesis test, comparing medians rather than means. The plot above appears to barely reject the null hypothesis of equal medians.

The nonparametric Wilcoxon rank sum test, implemented by the function `ranksum`, can be used to quantify the test of equal medians. It tests if two independent samples come from identical continuous (not necessarily normal) distributions with equal medians, against the alternative that they do not have equal medians.

```
[p,h] = ranksum(price1, price2)
p =
    0.0092
h =
      1
```

The test rejects the null hypothesis of equal medians at the default 5% significance level.

Available Hypothesis Tests

Note In addition to the functions listed below, Statistics Toolbox also includes functions for analysis of variance (ANOVA), which perform hypothesis tests in the context of linear modeling. These functions are discussed in the Chapter 7, "Linear Models" section of the documentation.

Function	Description
`ansaribradley`	Ansari-Bradley test. Tests if two independent samples come from the same distribution, against the alternative that they come from distributions that have the same median and shape but different variances.
`chi2gof`	Chi-square goodness-of-fit test. Tests if a sample comes from a specified distribution, against the alternative that it does not come from that distribution.
`dwtest`	Durbin-Watson test. Tests if the residuals from a linear regression are independent, against the alternative that there is autocorrelation among them.
`jbtest`	Jarque-Bera test. Tests if a sample comes from a normal distribution with unknown mean and variance, against the alternative that it does not come from a normal distribution.
`kstest`	One-sample Kolmogorov-Smirnov test. Tests if a sample comes from a continuous distribution with specified parameters, against the alternative that it does not come from that distribution.
`kstest2`	Two-sample Kolmogorov-Smirnov test. Tests if two samples come from the same continuous distribution, against the alternative that they do not come from the same distribution.
`lillietest`	Lilliefors test. Tests if a sample comes from a distribution in the normal family, against the alternative that it does not come from a normal distribution.

Function	Description
`ranksum`	Wilcoxon rank sum test. Tests if two independent samples come from identical continuous distributions with equal medians, against the alternative that they do not have equal medians.
`runstest`	Runs test. Tests if a sequence of values comes in random order, against the alternative that the ordering is not random.
`signrank`	One-sample or paired-sample Wilcoxon signed rank test. Tests if a sample comes from a continuous distribution symmetric about a specified median, against the alternative that it does not have that median.
`signtest`	One-sample or paired-sample sign test. Tests if a sample comes from an arbitrary continuous distribution with a specified median, against the alternative that it does not have that median.
`ttest`	One-sample or paired-sample *t*-test. Tests if a sample comes from a normal distribution with unknown variance and a specified mean, against the alternative that it does not have that mean.
`ttest2`	Two-sample *t*-test. Tests if two independent samples come from normal distributions with unknown but equal (or, optionally, unequal) variances and the same mean, against the alternative that the means are unequal.
`vartest`	One-sample chi-square variance test. Tests if a sample comes from a normal distribution with specified variance, against the alternative that it comes from a normal distribution with a different variance.
`vartest2`	Two-sample *F*-test for equal variances. Tests if two independent samples come from normal distributions with the same variance, against the alternative that they come from normal distributions with different variances.

Function	Description
`vartestn`	Bartlett multiple-sample test for equal variances. Tests if multiple samples come from normal distributions with the same variance, against the alternative that they come from normal distributions with different variances.
`ztest`	One-sample z-test. Tests if a sample comes from a normal distribution with known variance and specified mean, against the alternative that it does not have that mean.

Linear Models

Introduction

Linear models represent the relationship between a continuous response variable and one or more predictor variables (either continuous or categorical) in the form $y = X\beta + \epsilon$, where

- y is an n-by-1 vector of observations of the response variable.
- X is the n-by-p design matrix determined by the predictors.
- β is a p-by-1 vector of unknown parameters to be estimated.
- ε is an n-by-1 vector of independent, identically distributed random disturbances.

The general form of the linear model is used to solve a variety of "Linear Regression" on page 7-3 and "Analysis of Variance" on page 7-32 problems.

For examples of simple linear models using MATLAB functions, see *MATLAB Data Analysis*.

Linear Regression

Multiple Linear Regression

The purpose of multiple linear regression is to establish a quantitative relationship between a group of predictor variables (the columns of X) and a response, y. This relationship is useful for

- Understanding which predictors have the greatest effect.
- Knowing the direction of the effect (i.e., increasing x increases/decreases y).
- Using the model to predict future values of the response when only the predictors are currently known.

The following sections explain multiple linear regression in greater detail:

Mathematical Foundations of Multiple Linear Regression

The linear model takes its common form

$$y = X\beta + \varepsilon$$

where:

- y is an n-by-1 vector of observations.
- X is an n-by-p matrix of regressors.

- β is a *p*-by-1 vector of parameters.
- ε is an *n*-by-1 vector of random disturbances.

The solution to the problem is a vector, *b*, which estimates the unknown vector of parameters, β. The least squares solution is

$$b = \hat{\beta} = (X^T X)^{-1} X^T y$$

This equation is useful for developing later statistical formulas, but has poor numeric properties. `regress` uses QR decomposition of *X* followed by the backslash operator to compute *b*. The QR decomposition is not necessary for computing *b*, but the matrix *R* is useful for computing confidence intervals.

You can plug *b* back into the model formula to get the predicted *y* values at the data points.

$$\hat{y} = Xb = Hy$$

$$H = X(X^T X)^{-1} X^T$$

Note Statisticians use a hat (circumflex) over a letter to denote an estimate of a parameter or a prediction from a model. The projection matrix *H* is called the *hat matrix*, because it puts the "hat" on *y*.

The residuals are the difference between the observed and predicted *y* values.

$$r = y - \hat{y} = (I - H)y$$

The residuals are useful for detecting failures in the model assumptions, since they correspond to the errors, ε, in the model equation. By assumption, these errors each have independent normal distributions with mean zero and a constant variance.

The residuals, however, are correlated and have variances that depend on the locations of the data points. It is a common practice to scale ("Studentize") the residuals so they all have the same variance.

In the equation below, the scaled residual, t_i, has a Student's t distribution with (n-p-1) degrees of freedom

$$t_i = \frac{r_i}{\hat{\sigma}_{(i)}\sqrt{1-h_i}}$$

where

$$\hat{\sigma}^2_{(i)} = \frac{\|r\|^2}{n-p-1} - \frac{r_i^2}{(n-p-1)(1-h_i)}$$

and:

- t_i is the scaled residual for the ith data point.
- r_i is the raw residual for the ith data point.
- n is the sample size.
- p is the number of parameters in the model.
- h_i is the ith diagonal element of H.

The left-hand side of the second equation is the estimate of the variance of the errors excluding the ith data point from the calculation.

A hypothesis test for outliers involves comparing t_i with the critical values of the t distribution. If t_i is large, this casts doubt on the assumption that this residual has the same variance as the others.

A confidence interval for the mean of each error is

$$c_i = r_i \pm t_{\left(1-\frac{\alpha}{2},\nu\right)}\hat{\sigma}_{(i)}\sqrt{1-h_i}$$

Confidence intervals that do not include zero are equivalent to rejecting the hypothesis (at a significance probability of α) that the residual mean is zero. Such confidence intervals are good evidence that the observation is an outlier for the given model.

Example: Multiple Linear Regression

The example comes from Chatterjee and Hadi in a paper on regression diagnostics. The data set (originally from Moore) has five predictor variables and one response.

```
load moore
X = [ones(size(moore,1),1) moore(:,1:5)];
```

Matrix `X` has a column of ones, and then one column of values for each of the five predictor variables. The column of ones is necessary for estimating the *y*-intercept of the linear model.

```
y = moore(:,6);
[b,bint,r,rint,stats] = regress(y,X);
```

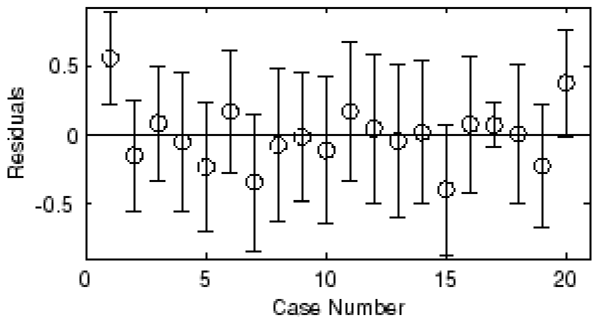
The *y*-intercept is `b(1)`, which corresponds to the column index of the column of ones.

```
stats
stats =
    0.8107   11.9886    0.0001    0.0685
```

The elements of the vector `stats` are the regression R^2 statistic, the F statistic (for the hypothesis test that all the regression coefficients are zero), the p-value associated with this F statistic, and an estimate of the error variance.

R^2 is 0.8107 indicating the model accounts for over 80% of the variability in the observations. The F statistic of about 12 and its p-value of 0.0001 indicate that it is highly unlikely that all of the regression coefficients are zero. The error variance of 0.0685 indicates that there a small random variability between the variable and the regression function.

```
rcoplot(r,rint)
```



The plot shows the residuals plotted in case order (by row). The 95% confidence intervals about these residuals are plotted as error bars. The first observation is an outlier since its error bar does not cross the zero reference line.

In problems with just a single predictor, it is simpler to use the `polytool` function. This function can form an *X* matrix with predictor values, their squares, their cubes, and so on.

Polynomial Curve Fitting Demo

The `polytool` demo is an interactive graphic environment for polynomial curve fitting and prediction. You can use `polytool` to do curve fitting and prediction for any set of *x*-*y* data, but, for the sake of demonstration, Statistics Toolbox provides a data set (`polydata.mat`) to illustrate some basic concepts.

With the `polytool` demo you can

- Plot the data, the fitted polynomial, and global confidence bounds on a new predicted value.
- Change the degree of the polynomial fit.
- Evaluate the polynomial at a specific *x*-value, or drag the vertical reference line to evaluate the polynomial at varying *x*-values.
- Display the predicted *y*-value and its uncertainty at the current *x*-value.
- Control the confidence bounds and choose between least squares or robust fitting.
- Export fit results to the workspace.

Note From the command line, you can call `polytool` and specify the data set, the order of the polynomial, and the confidence intervals, as well as labels to replace **X Values** and **Y Values**. See the `polytool` function reference page for details.

The following sections explore the use of `polytool`:

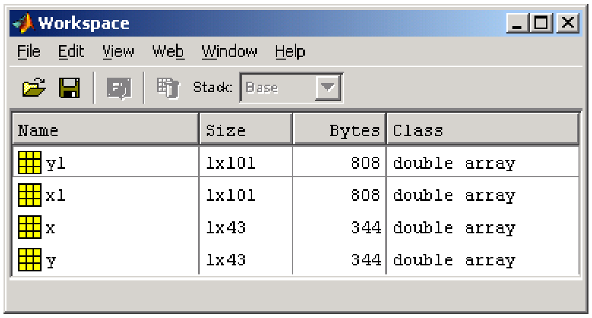
- "Fitting a Polynomial" on page 7-8
- "Confidence Bounds" on page 7-11

Fitting a Polynomial.

1 **Load the data.** Before you start the demonstration, you must first load a data set. This example uses `polydata.mat`. For this data set, the variables x and y are observations made with error from a cubic polynomial. The variables `x1` and `y1` are data points from the "true" function without error.

```
load polydata
```

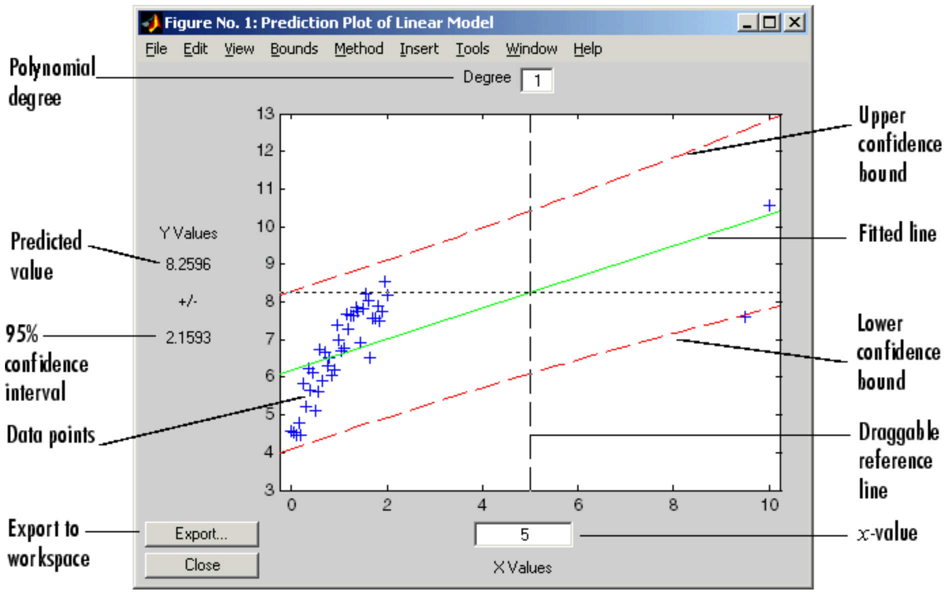
Your variables appear in the Workspace browser.



Name	Size	Bytes	Class
y1	1x101	808	double array
x1	1x101	808	double array
x	1x43	344	double array
y	1x43	344	double array

2 **Try a linear fit.** Run `polytool` and provide it with the data to which the polynomial is fit. Because this code does not specify the degree of the polynomial, `polytool` does a linear fit to the data.

```
polytool(x,y)
```



The linear fit is not very good. The bulk of the data with *x*-values between 0 and 2 has a steeper slope than the fitted line. The two points to the right are dragging down the estimate of the slope.

3 **Try a cubic fit.** In the **Degree** text box at the top, type 3 for a cubic model. Then, drag the vertical reference line to the *x*-value of 2 (or type 2 in the **X Values** text box).

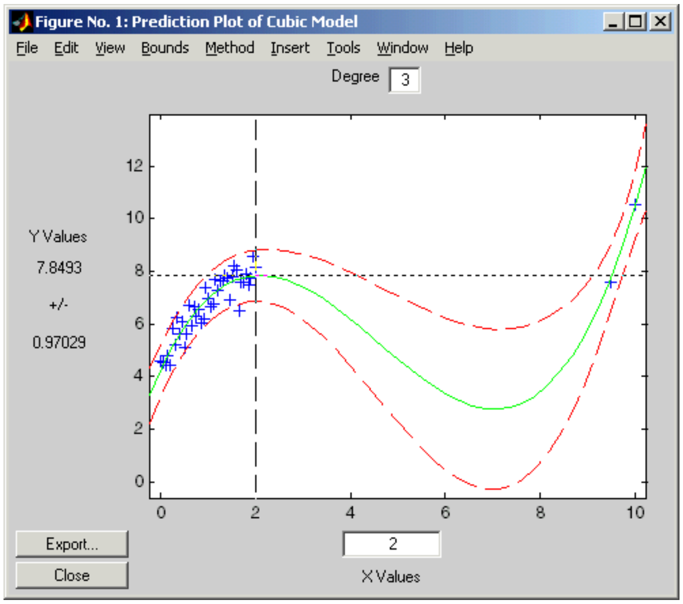


Figure No. 1: Prediction Plot of Cubic Model
File Edit View Bounds Method Insert Tools Window Help
Degree 3
Y Values
7.8493
+/-
0.97029
Export...
Close
2
X Values

This graph shows a much better fit to the data. The confidence bounds are closer together indicating that there is less uncertainty in prediction. The data at both ends of the plot track the fitted curve.

4 **Finally, overfit the data.** If the cubic polynomial is a good fit, it is tempting to try a higher order polynomial to see if even more precise predictions are possible. Since the true function is cubic, this amounts to overfitting the data. Use the data entry box for degree and type 5 for a quintic model.

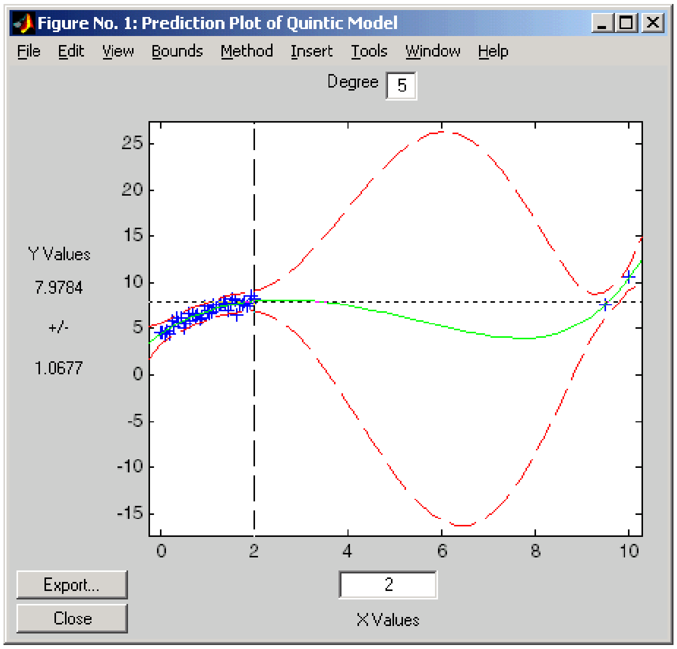


Figure No. 1: Prediction Plot of Quintic Model
File Edit View Bounds Method Insert Tools Window Help
Degree 5
Y Values
7.9784
+/-
1.0677
25
20
15
10
5
0
-5
-10
-15
0
2
4
6
8
10
Export...
Close
2
X Values

As measured by the confidence bounds, the fit is precise near the data points. But, in the region between the data groups, the uncertainty of prediction rises dramatically.

This bulge in the confidence bounds happens because the data really does not contain enough information to estimate the higher order polynomial terms precisely, so even interpolation using polynomials can be risky in some cases.

Confidence Bounds. By default, the confidence bounds are nonsimultaneous bounds for a new observation. What does this mean? Let $p(x)$ be the true but unknown function you want to estimate. The graph contains the following three curves:

- $f(x)$, the fitted function
- $l(x)$, the lower confidence bounds
- $u(x)$, the upper confidence bounds

Suppose you plan to take a new observation at the value x_{n+1}. Call it $y_{n+1}(x_{n+1})$. This new observation has its own error ε_{n+1}, so it satisfies the equation

$$y_{n+1}(x_{n+1}) = p(x_{n+1}) + \varepsilon_{n+1}$$

What are the likely values for this new observation? The confidence bounds provide the answer. The interval [l_{n+1} , u_{n+1}] is a 95% confidence bound for $y_{n+1}(x_{n+1})$.

These are the default bounds, but the **Bounds** menu on the `polytool` figure window provides options for changing the meaning of these bounds. This menu has options that enable you to specify whether the bounds should be simultaneous or not, and whether the bounds are to apply to the estimated function, i.e., curve, or to a new observation. Using these options you can produce any of the following types of confidence bounds.

Confidence bound table of one heading row, four data rows, and three columns.

Is Simultaneous	For Quantity	Yields Confidence Bounds for
Nonsimultaneous	Observation	$y_{n+1}(x_{n+1})$ (default)
Nonsimultaneous	Curve	$p(x_{n+1})$
Simultaneous	Observation	$y_{n+1}(x)$, globally for any x
Simultaneous	Curve	$p(x)$, simultaneously for all x

Quadratic Response Surface Models

Response Surface Methodology (RSM) is a tool for understanding the quantitative relationship between multiple input variables and one output variable.

Consider one output, z, as a polynomial function of two inputs, x and y. The function $z = f(x,y)$ describes a two-dimensional surface in the space (x,y,z). In general, you can have as many input variables as you want and the resulting surface becomes a hypersurface. Also, you can have multiple output variables with a separate hypersurface for each one.

For three inputs (x_1, x_2, x_3), the equation of a quadratic response surface is

$$\begin{aligned} y = {} & b_0 + b_1x_1 + b_2x_2 + b_3x_3 + \ldots && \text{(linear terms)} \\ & + b_{12}x_1x_2 + b_{13}x_1x_3 + b_{23}x_2x_3 + \ldots && \text{(interaction terms)} \\ & + b_{11}x_1^2 + b_{22}x_2^2 + b_{33}x_3^2 && \text{(quadratic terms)} \end{aligned}$$

It is difficult to visualize a k-dimensional surface in k+1 dimensional space for $k > 2$.

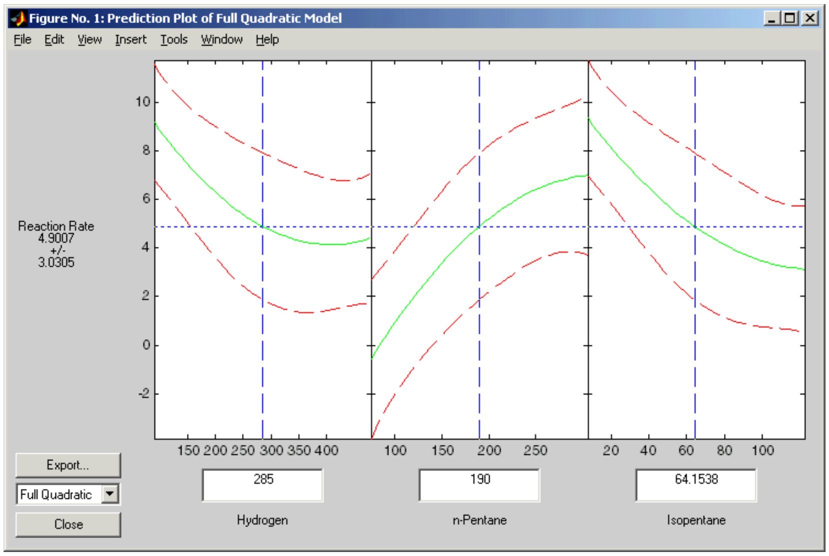
Exploring Graphs of Multidimensional Polynomials

The function `rstool` performs an interactive fit and plot of a multidimensional response surface model (RSM). Note that, in general, this GUI provides an environment for exploration of the graph of a multidimensional polynomial.

You can learn about `rstool` by trying the commands below. The chemistry behind the data in `reaction.mat` deals with reaction kinetics as a function of the partial pressure of three chemical reactants: hydrogen, n-pentane, and isopentane.

```
load reaction
rstool(reactants,rate,'quadratic',0.01,xn,yn)
```

`rstool` displays a “vector” of three plots. The dependent variable of all three plots is the reaction rate. The first plot has hydrogen as the independent variable. The second and third plots have n-pentane and isopentane respectively.

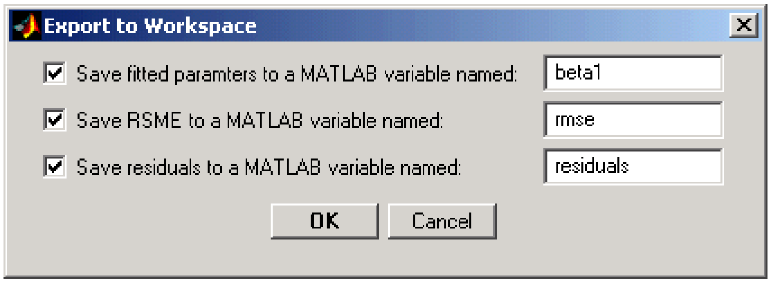


Each plot shows the fitted relationship of the reaction rate to the independent variable at a fixed value of the other two independent variables. The fixed value of each independent variable is in an editable text box below each axis, and is marked by a vertical dashed blue line. You can change the fixed value of any independent variable by either typing a new value in the box or by dragging any of the three vertical lines to a new position.

When you change the value of an independent variable, all the plots update to show the current picture at the new point in the space of the independent variables.

Note that while this example only uses three inputs (`reactants`) and one output (`rate`), `rstool` can accommodate an arbitrary number of inputs and outputs. Interpretability may be limited by the size of your monitor for large numbers of inputs or outputs.

Exporting Variables to the Workspace. Click **Export** to save variables in the GUI to the base workspace.



Export to Workspace

- [✓] Save fitted paramters to a MATLAB variable named: beta1
- [✓] Save RSME to a MATLAB variable named: rmse
- [✓] Save residuals to a MATLAB variable named: residuals

Fitted parameters, i.e., coefficients, appear in the following order. Some polynomial models use a subset of these terms but keep them in this order.

1 Constant term

2 Linear terms

3 Interaction terms formed by taking pairwise products of the columns of the input matrix

4 Squared terms

Changing the Order of the Polynomial. Below the **Export** button, there is a pop-up menu that enables you to change the polynomial model. If you use the commands above, this menu has the string `Full Quadratic` already selected. The choices are:

- `Linear` — includes constant and linear terms.
- `Pure Quadratic` — includes constant, linear and squared terms.
- `Interactions` — includes constant, linear, and cross product terms.
- `Full Quadratic` — includes interactions and squared terms.
- `User Specified` — available only if you provide a matrix of model terms as the third argument to `rstool`. See the `rstool` and `x2fx` function reference pages for details.)

The `rstool` GUI is used by the `rsmdemo` function to visualize the results of a designed experiment for studying a chemical reaction. See “Design of Experiments Demo” on page 11-11.

Stepwise Regression

Stepwise regression is a technique for choosing the variables, i.e., terms, to include in a multiple regression model. Forward stepwise regression starts with no model terms. At each step it adds the most statistically significant term (the one with the highest F statistic or lowest p-value) until there are none left. Backward stepwise regression starts with all the terms in the model and removes the least significant terms until all the remaining terms are statistically significant. It is also possible to start with a subset of all the terms and then add significant terms or remove insignificant terms.

An important assumption behind the method is that some input variables in a multiple regression do not have an important explanatory effect on the response. If this assumption is true, then it is a convenient simplification to keep only the statistically significant terms in the model.

One common problem in multiple regression analysis is multicollinearity of the input variables. The input variables may be as correlated with each other as they are with the response. If this is the case, the presence of one input variable in the model may mask the effect of another input. Stepwise regression might include different variables depending on the choice of starting model and inclusion strategy.

Statistics Toolbox includes two functions for performing stepwise regression:

- `stepwise` — an interactive graphical tool that enables you to explore stepwise regression.
- `stepwisefit` — a command-line tool for performing stepwise regression. You can use `stepwisefit` to return the results of a stepwise regression to the MATLAB workspace.

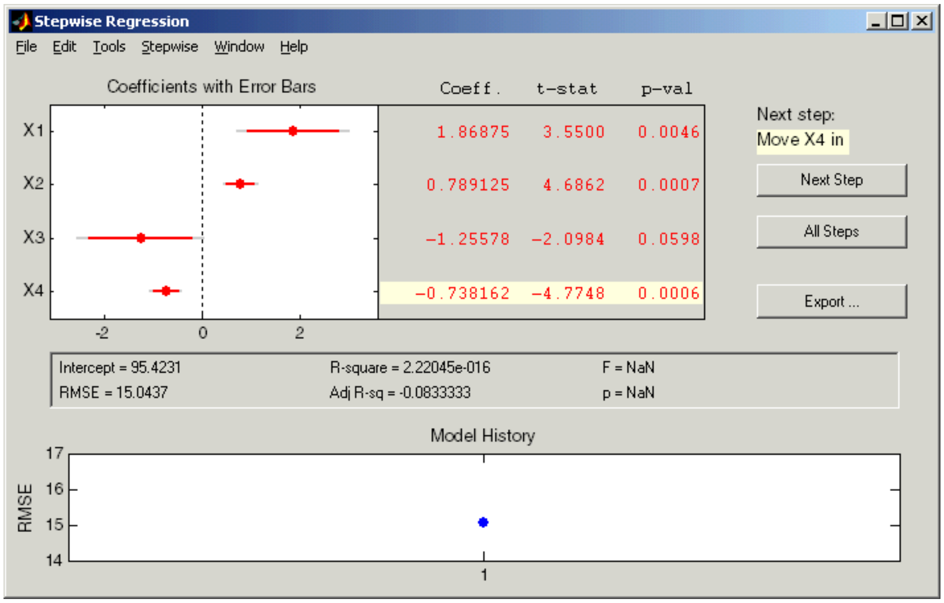
Stepwise Regression Demo

The `stepwise` function provides an interactive graphical interface that you can use to compare competing models.

This example uses the Hald ([21], p. 167) data set. The Hald data come from a study of the heat of reaction of various cement mixtures. There are four components in each mixture, and the amount of heat produced depends on the amount of each ingredient in the mixture.

Here are the commands to get started.

```
load hald
stepwise(ingredients,heat)
```



	Coeff.	t-stat	p-val
X1	1.86875	3.5500	0.0046
X2	0.789125	4.6862	0.0007
X3	-1.25578	-2.0984	0.0598
X4	-0.738162	-4.7748	0.0006

```
Intercept = 95.4231     R-square = 2.22045e-016     F = NaN
RMSE = 15.0437          Adj R-sq = -0.0833333       p = NaN
```

For each term on the *y*-axis, the plot shows the regression (least squares) coefficient as a dot with horizontal bars indicating confidence intervals. Blue dots represent terms that are in the model, while red dots indicate terms that are not currently in the model. The horizontal bars indicate 90% (colored) and 95% (grey) confidence intervals.

To the right of each bar, a table lists the value of the regression coefficient for that term, along with its t-statistic and p-value. The coefficient for a term that is not in the model is the coefficient that would result from adding that term to the current model.

From the **Stepwise** menu, select `Scale Inputs` to center and normalize the columns of the input matrix to have a standard deviation of 1.

Note When you call the `stepwise` function, you can also specify the initial state of the model and the confidence levels to use. See the `stepwise` function reference page for details.

Additional Diagnostic Statistics. Several diagnostic statistics appear below the plot.

- Intercept — the estimated value of the constant term
- RMSE — the root mean squared error of the current model
- R-square — the amount of response variability explained by the model
- Adjusted R-square — the R-square statistic adjusted for the residual degrees of freedom
- F — the overall F statistic for the regression
- P — the associated significance probability

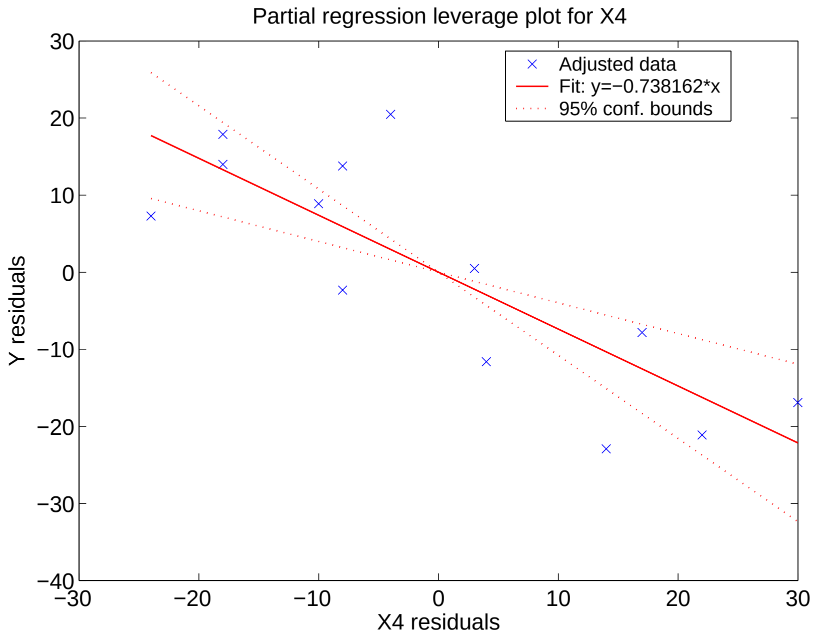
Moving Terms In and Out of the Model. There are two ways you can move terms in and out of the model:

- Click on a line in the plot or in the table to toggle the state of the corresponding term. The resulting change to the model depends on the color of the line:
  - Clicking a blue line, corresponding to a term currently in the model, removes the term from the model and changes the line to red.
  - Clicking a red line, corresponding to a term currently not in the model, adds the term to the model and changes the line to blue.
- Select the recommended step shown under **Next Step** to the right of the table. The recommended step is either to add the most statistically significant term, or to remove the least significant term. Click **Next Step** to perform the recommended step. After you do so, the `stepwise` GUI displays the next term to add or remove. When there are no more recommended steps, the GUI displays "Move no terms."

  Alternatively, you can perform all the recommended steps at once by clicking **All Steps.**

Assessing the Effect of Adding a Term. The demo can produce a partial regression leverage plot for the term you choose. If the term is not in the model, the plot shows the effect of adding it by plotting the residuals of the terms that are in the model against the residuals of the chosen term. If the term is in the model, the plot shows the effect of adding it if it were not already in the model. That is, the demo plots the residuals of all *other* terms in the model against the residuals of the chosen term.

From the **Stepwise** menu, select `Added Variable Plot` to display a list of terms. Select the term for which you want a plot, and click **OK**. This example selects `X4`, the recommended term in the figure above.



Model History. The Model History plot shows the RMSE for every model generated during the current session. Click one of the dots to return to the model at that point in the analysis.

Exporting Variables. The **Export** pop-up menu enables you to export variables from the `stepwise` function to the base workspace. Check the variables you want to export and, optionally, change the variable name in the corresponding edit box. Click **OK**.

Generalized Linear Models

So far, the functions in this section have dealt with models that have a linear relationship between the response and one or more predictors. Sometimes you may have a nonlinear relationship instead. To fit nonlinear models you can use the functions described in Chapter 8, "Nonlinear Models". However, there are some nonlinear models, known as generalized linear models, that you can fit using simpler linear methods. To understand generalized linear models, first review the linear models you have seen so far. Each of these models has the following three characteristics:

- The response has a normal distribution with mean μ.
- A coefficient vector b defines a linear combination $X*b$ of the predictors X.
- The model equates the two as $\mu = X*b$.

In generalized linear models, these characteristics are generalized as follows:

- The response has a distribution that may be normal, binomial, Poisson, gamma, or inverse Gaussian, with parameters including a mean µ.
- A coefficient vector b defines a linear combination $X*b$ of the predictors X.
- A link function $f(\cdot)$ defines the link between the two as $f(\mu) = X*b$.

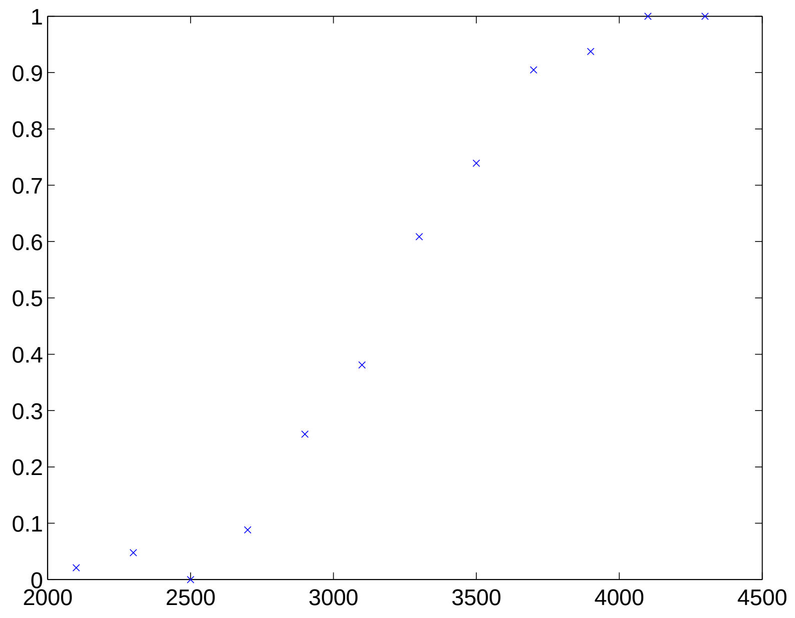
The following sections explore these models in greater detail:

- "Example: Generalized Linear Models" on page 7-20
- "Generalized Linear Model Demo" on page 7-25

Example: Generalized Linear Models

For example, consider the following data derived from the `carbig` data set, in which the cars have various weights. You record the total number of cars of each weight and the number qualifying as poor-mileage cars because their miles per gallon value is below some target. Assume that you don't know the miles per gallon for each car, only the number passing the test. It might be reasonable to assume that the value of the variable `poor` follows a binomial distribution with parameter `N=total` and with a `p` parameter that depends on the car weight. A plot shows that the proportion of poor-mileage cars follows a nonlinear S-shape.

```
w = [2100 2300 2500 2700 2900 3100...
     3300 3500 3700 3900 4100 4300]';
poor = [1 2 0 3 8 8 14 17 19 15 17 21]';
total = [48 42 31 34 31 21 23 23 21 16 17 21]';

[w poor total]
ans =
        2100           1          48
        2300           2          42
        2500           0          31
        2700           3          34
        2900           8          31
        3100           8          21
        3300          14          23
        3500          17          23
        3700          19          21
        3900          15          16
        4100          17          17
        4300          21          21

plot(w,poor./total,'x')
```



This shape is typical of graphs of proportions, as they have natural boundaries at 0.0 and 1.0.

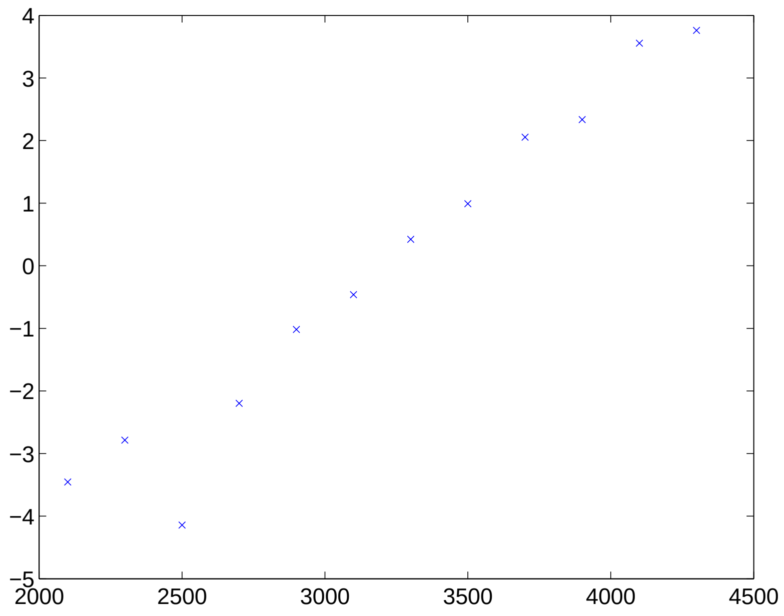
A linear regression model would not produce a satisfactory fit to this graph. Not only would the fitted line not follow the data points, it would produce invalid proportions less than 0 for light cars, and higher than 1 for heavy cars.

There is a class of regression models for dealing with proportion data. The logistic model is one such model. It defines the relationship between proportion p and weight w to be

$$\log\left(\frac{p}{1-p}\right) = b_1 + b_2 w$$

Is this a good model for the data? It would be helpful to graph the data on this scale, to see if the relationship appears linear. However, some of the proportions are 0 and 1, so you cannot explicitly evaluate the left-hand-side of the equation. A useful trick is to compute adjusted proportions by adding small increments to the poor and total values—say a half observation to poor and a full observation to total. This keeps the proportions within range. A graph now shows a more nearly linear relationship.

```
padj = (poor+.5) ./ (total+1);
plot(w,log(padj./(1-padj)),'x')
```



You can use the glmfit function to fit this logistic model.

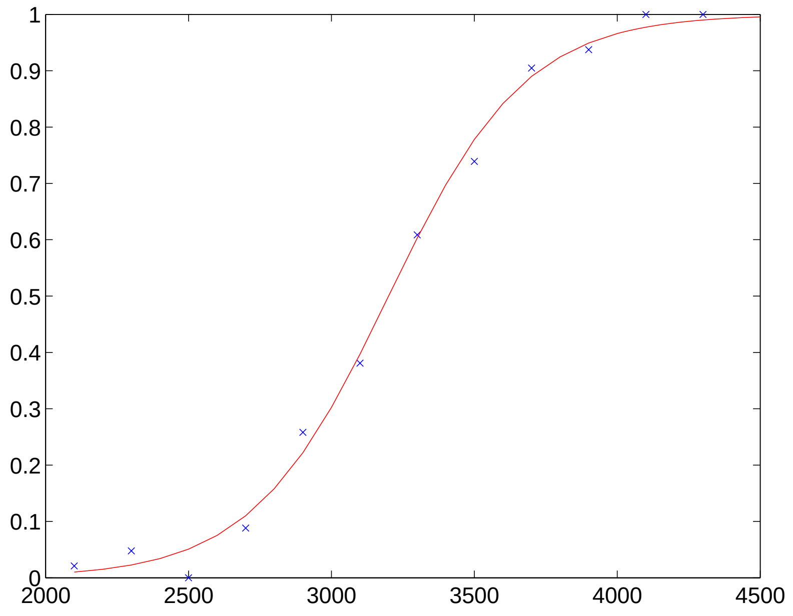
```
b = glmfit(w,[poor total],'binomial')

b =
  -13.3801
    0.0042
```

To use these coefficients to compute a fitted proportion, you have to invert the logistic relationship. Some simple algebra shows that the logistic equation can also be written as

$$p = \frac{1}{1 + \exp(-b_1 - b_2 w)}$$

Fortunately, the function `glmval` can decode this link function to compute the fitted values. Using this function, you can graph fitted proportions for a range of car weights, and superimpose this curve on the original scatter plot.

```
x = 2100:100:4500;
y = glmval(b,x,'logit');
plot(w,poor./total,'x',x,y,'r-')
```



Generalized linear models can fit a variety of distributions with a variety of relationships between the distribution parameters and the predictors.

Generalized Linear Model Demo

The `glmdemo` function begins a slide show describing generalized linear models. It presents examples of what functions and distributions are available with generalized linear models. It then presents an example where traditional linear least squares fitting is not appropriate, and shows how to use the `glmfit` function to fit a logistic regression model and the `glmval` function to compute predictions from that model. (See the `glmfit` and `glmval` function reference pages for details.)

To run `glmdemo` from the command line, type `playshow glmdemo`.

Robust and Nonparametric Methods

As mentioned in the previous sections, regression and analysis of variance procedures depend on certain assumptions, such as a normal distribution for the error term. Sometimes such an assumption is not warranted. For example, if the distribution of the errors is asymmetric or prone to extreme outliers, that is a violation of the assumption of normal errors.

Statistics Toolbox has a robust regression function that is useful when there may be outliers. Robust methods are designed to be relatively insensitive to large changes in a small part of the data.

Statistics Toolbox also has nonparametric versions of the one-way and two-way analysis of variance functions. Unlike classical tests, nonparametric tests make only mild assumptions about the data, and are appropriate when the distribution of the data is not normal. On the other hand, they are less powerful than classical methods for normally distributed data.

The following sections describe the robust regression and nonparametric functions in greater detail:

- "Robust Regression" on page 7-26
- "Kruskal-Wallis Test" on page 7-29
- "Friedman's Test" on page 7-30

Both of the nonparametric functions described here can return a `stats` structure that you can use as input to the `multcompare` function to perform multiple comparisons.

Robust Regression

"Example: Multiple Linear Regression" on page 7-6 shows that there is an outlier when you use ordinary least squares regression to model a response as a function of five predictors. How does that outlier affect the results?

There is a type of regression known as "robust" regression that can be used to limit the effect of outliers. The idea is to assign a weight to each point so that outliers are given reduced weight. This makes the results less sensitive to the presence of outliers. The weighting is done automatically and iteratively as follows. In the first iteration, the fit is an ordinary least squares fit with each point having the same weight. Then new weights are computed to give lower weight to points that are far from their predicted values, and the fit is repeated using these weights. The process continues until it converges.

So, to determine how the outlier affects the results in this example, first estimate the coefficients using the `robustfit` function.

```
load moore
x = moore(:,1:5);
y = moore(:,6);
[br,statsr] = robustfit(x,y);
br
br =
   -1.7742
    0.0000
    0.0009
    0.0002
    0.0062
    0.0001
```

Compare these estimates to those you obtain from the `regress` function.

```
b
b =
   -2.1561
   -0.0000
    0.0013
    0.0001
    0.0079
    0.0001
```

To understand why the two differ, it is helpful to look at the weight variable from the robust fit. It measures how much weight was given to each point during the final iteration of the fit. In this case, the first point had a very low weight so it was effectively ignored.

```
statsr.w'
ans =
  Columns 1 through 7
    0.0577  0.9977  0.9776  0.9455  0.9687  0.8734  0.9177
  Columns 8 through 14
    0.9990  0.9653  0.9679  0.9768  0.9882  0.9998  0.9979
  Columns 15 through 20
    0.8185  0.9757  0.9875  0.9991  0.9021  0.6953
```

Robust Fitting Demo. The `robustdemo` function presents a simple comparison of least squares and robust fits for a response and a single predictor. You can use data provided by the demo or supply your own. See the `robustdemo` function reference page for information about using your own data:

1. **Start the demo.** To begin using `robustdemo` with the built-in sample data, simply type the function name.

   ```
   robustdemo
   ```

   The resulting figure presents a scatter plot with two fitted lines. One line is the fit from an ordinary least squares regression. The other is from a robust regression. Along the bottom of the figure are the equations for the fitted line and the estimated error standard deviation for each fit.

   The effect of any point on the least squares fit depends on the residual and leverage for that point. The residual is the vertical distance from the point to the line. The leverage is a measure of how far the point is from the center of the *x* data.

   The effect of any point on the robust fit also depends on the weight assigned to the point. Points far from the line get lower weight.

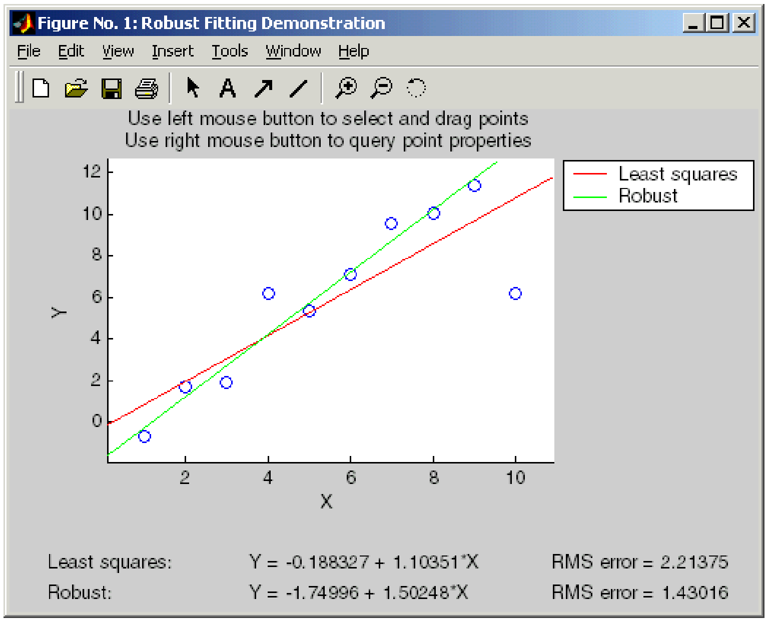


Figure No. 1: Robust Fitting Demonstration
File Edit View Insert Tools Window Help
Use left mouse button to select and drag points
Use right mouse button to query point properties
Least squares
Robust
Y
X
12
10
8
6
4
2
0
2
4
6
8
10

Least squares: Y = -0.188327 + 1.10351*X RMS error = 2.21375
Robust: Y = -1.74996 + 1.50248*X RMS error = 1.43016

2 **Compare effects of leverage and weight.** Use the right mouse button to click on any point and see its least squares leverage and robust weight.

   In this example, the right-most point has a leverage value of `0.35`. It is also far from the line, so it exerts a large influence on the least squares fit. It has a small weight, though, so it is effectively excluded from the robust fit.

3 **See how changes in data affect the two fits.** Using the left mouse button, select any point, and drag it to a new location while holding the left button down. When you release the point, both fits update.

   Bringing the right-most point closer to the line makes the two fitted lines nearly identical. Now, the point has nearly full weight in the robust fit.

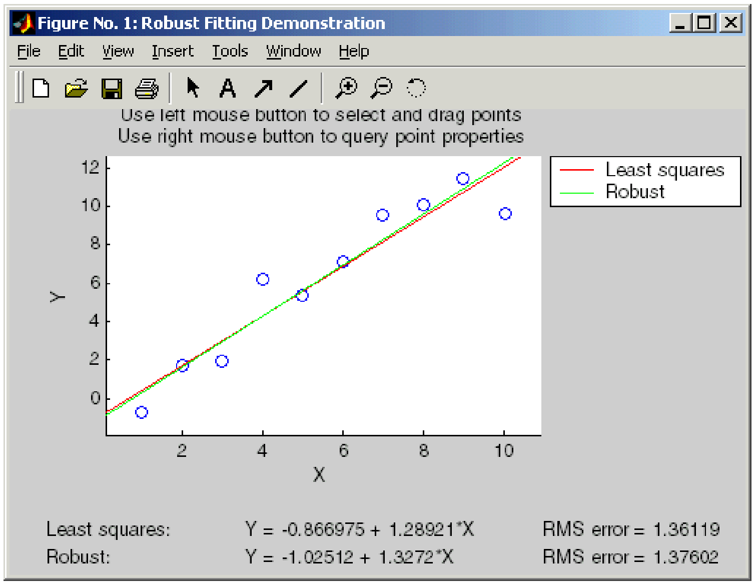


Figure No. 1: Robust Fitting Demonstration
File Edit View Insert Tools Window Help
Use left mouse button to select and drag points
Use right mouse button to query point properties
Least squares
Robust
Y
X

Least squares: Y = -0.866975 + 1.28921*X RMS error = 1.36119
Robust: Y = -1.02512 + 1.3272*X RMS error = 1.37602

Kruskal-Wallis Test

The example "Example: One-Way ANOVA" on page 7-33 uses one-way analysis of variance to determine if the bacteria counts of milk varied from shipment to shipment. The one-way analysis rests on the assumption that the measurements are independent, and that each has a normal distribution with a common variance and with a mean that was constant in each column. You can conclude that the column means were not all the same. The following example repeats that analysis using a nonparametric procedure.

The Kruskal-Wallis test is a nonparametric version of one-way analysis of variance. The assumption behind this test is that the measurements come from a continuous distribution, but not necessarily a normal distribution. The test is based on an analysis of variance using the ranks of the data values, not the data values themselves. Output includes a table similar to an ANOVA table, and a box plot.

You can run this test as follows:

```
p = kruskalwallis(hogg)
p =
    0.0020
```

The low p-value means the Kruskal-Wallis test results agree with the one-way analysis of variance results.

Friedman's Test

The example "Example: Two-Way ANOVA" on page 7-38 uses two-way analysis of variance to study the effect of car model and factory on car mileage. The example tests whether either of these factors has a significant effect on mileage, and whether there is an interaction between these factors. The conclusion of the example is there is no interaction, but that each individual factor has a significant effect. The next example examines whether a nonparametric analysis leads to the same conclusion.

Friedman's test is a nonparametric test for data having a two-way layout (data grouped by two categorical factors). Unlike two-way analysis of variance, Friedman's test does not treat the two factors symmetrically and it does not test for an interaction between them. Instead, it is a test for whether the columns are different after adjusting for possible row differences. The test is based on an analysis of variance using the ranks of the data across categories of the row factor. Output includes a table similar to an ANOVA table.

You can run Friedman's test as follows.

```
p = friedman(mileage, 3)

ans =

  7.4659e-004
```

Recall the classical analysis of variance gave a p-value to test column effects, row effects, and interaction effects. This p-value is for column effects. Using either this p-value or the p-value from ANOVA ($p < 0.0001$), you conclude that there are significant column effects.

In order to test for row effects, you need to rearrange the data to swap the roles of the rows in columns. For a data matrix x with no replications, you could simply transpose the data and type

```
p = friedman(x')
```

With replicated data it is slightly more complicated. A simple way is to transform the matrix into a three-dimensional array with the first dimension representing the replicates, swapping the other two dimensions, and restoring the two-dimensional shape.

```
x = reshape(mileage, [3 2 3]);
x = permute(x, [1 3 2]);
x = reshape(x, [9 2])
x =
   33.3000   32.6000
   33.4000   32.5000
   32.9000   33.0000
   34.5000   33.4000
   34.8000   33.7000
   33.8000   33.9000
   37.4000   36.6000
   36.8000   37.0000
   37.6000   36.7000

friedman(x, 3)

ans =

    0.0082
```

Again, the conclusion is similar to that of the classical analysis of variance. Both this p-value and the one from ANOVA (p = 0.0039) lead you to conclude that there are significant row effects.

You cannot use Friedman's test to test for interactions between the row and column factors.

Analysis of Variance

One-Way Analysis of Variance

The purpose of one-way ANOVA is to find out whether data from several groups have a common mean. That is, to determine whether the groups are actually different in the measured characteristic.

One-way ANOVA is a simple special case of the linear model. The one-way ANOVA form of the model is

$$y_{ij} = \alpha_{.j} + \varepsilon_{ij}$$

where:

- y_{ij} is a matrix of observations in which each column represents a different group.
- $\alpha_{.j}$ is a matrix whose columns are the group means. (The "dot j" notation means that α applies to all rows of the jth column. That is, the value α_{ij} is the same for all i.)
- ε_{ij} is a matrix of random disturbances.

The model assumes that the columns of y are a constant plus a random disturbance. You want to know if the constants are all the same.

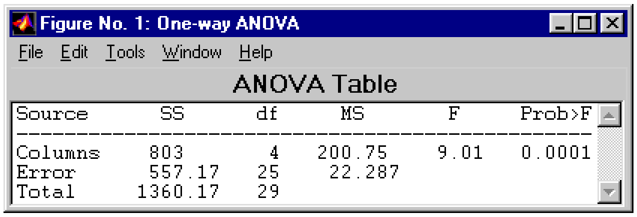
The following sections explore one-way ANOVA in greater detail:

Example: One-Way ANOVA

The data below comes from a study by Hogg and Ledolter [23] of bacteria counts in shipments of milk. The columns of the matrix hogg represent different shipments. The rows are bacteria counts from cartons of milk chosen randomly from each shipment. Do some shipments have higher counts than others?

```
load hogg
hogg

hogg =

    24    14    11     7    19
    15     7     9     7    24
    21    12     7     4    19
    27    17    13     7    15
    33    14    12    12    10
    23    16    18    18    20

[p,tbl,stats] = anova1(hogg);
p

p =
   1.1971e-04
```

The standard ANOVA table has columns for the sums of squares, degrees of freedom, mean squares (SS/df), F statistic, and p-value.



ANOVA Table

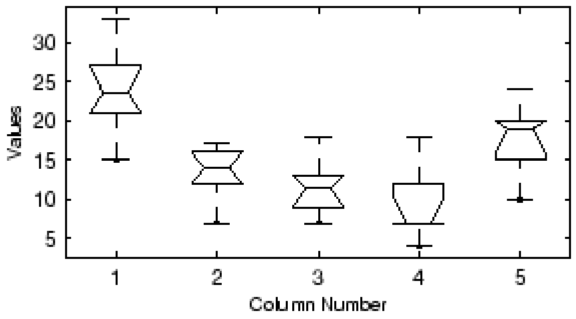
```
Source       SS      df      MS       F      Prob>F
---------------------------------------------------
Columns     803       4    200.75    9.01    0.0001
Error       557.17   25     22.287
Total      1360.17   29
```

You can use the F statistic to do a hypothesis test to find out if the bacteria counts are the same. anova1 returns the p-value from this hypothesis test.

In this case the p-value is about 0.0001, a very small value. This is a strong indication that the bacteria counts from the different tankers are not the same. An F statistic as extreme as the observed F would occur by chance only once in 10,000 times if the counts were truly equal.

The p-value returned by `anova1` depends on assumptions about the random disturbances ε_{ij} in the model equation. For the p-value to be correct, these disturbances need to be independent, normally distributed, and have constant variance. See "Robust and Nonparametric Methods" on page 7-25 for a nonparametric function that does not require a normal assumption.

You can get some graphical assurance that the means are different by looking at the box plots in the second figure window displayed by `anova1`. Note however that the notches are used for a comparison of medians, not a comparison of means. For more information on this display, see "Box Plots" on page 4-6.



Multiple Comparisons

Sometimes you need to determine not just whether there are any differences among the means, but specifically which pairs of means are significantly different. It is tempting to perform a series of t tests, one for each pair of means, but this procedure has a pitfall.

In a t test, you compute a t statistic and compare it to a critical value. The critical value is chosen so that when the means are really the same (any apparent difference is due to random chance), the probability that the t statistic will exceed the critical value is small, say 5%. When the means are different, the probability that the statistic will exceed the critical value is larger.

In this example there are five means, so there are 10 pairs of means to compare. It stands to reason that if all the means are the same, and if there is a 5% chance of incorrectly concluding that there is a difference in one pair, then the probability of making at least one incorrect conclusion among all 10 pairs is much larger than 5%.

Fortunately, there are procedures known as *multiple comparison procedures* that are designed to compensate for multiple tests.

Example: Multiple Comparisons

You can perform a multiple comparison test using the multcompare function and supplying it with the stats output from anova1.

```
[c,m] = multcompare(stats)
c =
    1.0000    2.0000    2.4953   10.5000   18.5047
    1.0000    3.0000    4.1619   12.1667   20.1714
    1.0000    4.0000    6.6619   14.6667   22.6714
    1.0000    5.0000   -2.0047    6.0000   14.0047
    2.0000    3.0000   -6.3381    1.6667    9.6714
    2.0000    4.0000   -3.8381    4.1667   12.1714
    2.0000    5.0000  -12.5047   -4.5000    3.5047
    3.0000    4.0000   -5.5047    2.5000   10.5047
    3.0000    5.0000  -14.1714   -6.1667    1.8381
    4.0000    5.0000  -16.6714   -8.6667   -0.6619
m =
   23.8333    1.9273
   13.3333    1.9273
   11.6667    1.9273
    9.1667    1.9273
   17.8333    1.9273
```

The first output from multcompare has one row for each pair of groups, with an estimate of the difference in group means and a confidence interval for that group. For example, the second row has the values

```
1.0000    3.0000    4.1619   12.1667   20.1714
```

indicating that the mean of group 1 minus the mean of group 3 is estimated to be 12.1667, and a 95% confidence interval for this difference is [4.1619, 20.1714]. This interval does not contain 0, so you can conclude that the means of groups 1 and 3 are different.

The second output contains the mean and its standard error for each group.

It is easier to visualize the difference between group means by looking at the graph that `multcompare` produces.

There are five groups. The graph instructs you to **Click on the group you want to test**. Three groups have slopes significantly different from group one.

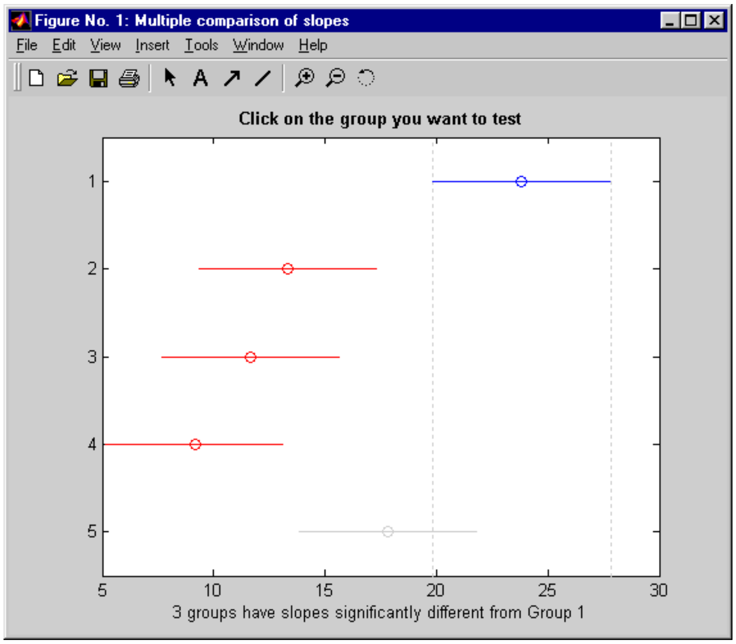


Figure No. 1: Multiple comparison of slopes
File Edit View Insert Tools Window Help
Click on the group you want to test
1
2
3
4
5
5
10
15
20
25
30
3 groups have slopes significantly different from Group 1

The graph shows that group 1 is significantly different from groups 2, 3, and 4. By using the mouse to select group 4, you can determine that it is also significantly different from group 5. Other pairs are not significantly different.

Two-Way Analysis of Variance

The purpose of two-way ANOVA is to find out whether data from several groups have a common mean. One-way ANOVA and two-way ANOVA differ in that the groups in two-way ANOVA have two categories of defining characteristics instead of one.

Suppose an automobile company has two factories, and each factory makes the same three models of car. It is reasonable to ask if the gas mileage in the cars varies from factory to factory as well as from model to model. There are two predictors, factory and model, to explain differences in mileage.

There could be an overall difference in mileage due to a difference in the production methods between factories. There is probably a difference in the mileage of the different models (irrespective of the factory) due to differences in design specifications. These effects are called *additive*.

Finally, a factory might make high mileage cars in one model (perhaps because of a superior production line), but not be different from the other factory for other models. This effect is called an *interaction*. It is impossible to detect an interaction unless there are duplicate observations for some combination of factory and car model.

Two-way ANOVA is a special case of the linear model. The two-way ANOVA form of the model is

$$y_{ijk} = \mu + \alpha_{.j} + \beta_{i.} + \gamma_{ij} + \varepsilon_{ijk}$$

where, with respect to the automobile example above:

- y_{ijk} is a matrix of gas mileage observations (with row index *i*, column index *j*, and repetition index *k*).
- μ is a constant matrix of the overall mean gas mileage.
- $\alpha_{.j}$ is a matrix whose columns are the deviations of each car's gas mileage (from the mean gas mileage μ) that are attributable to the car's *model*. All values in a given column of $\alpha_{.j}$ are identical, and the values in each row of $\alpha_{.j}$ sum to 0.
- $\beta_{i.}$ is a matrix whose rows are the deviations of each car's gas mileage (from the mean gas mileage μ) that are attributable to the car's *factory*. All

values in a given row of $\beta_{i.}$ are identical, and the values in each column of $\beta_{i.}$ sum to 0.

- γ_{ij} is a matrix of interactions. The values in each row of γ_{ij} sum to 0, and the values in each column of γ_{ij} sum to 0.
- ε_{ijk} is a matrix of random disturbances.

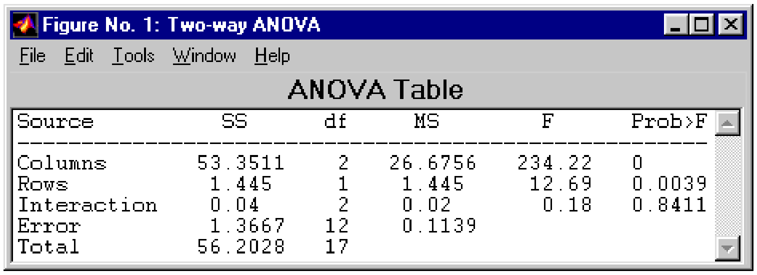
Example: Two-Way ANOVA

The purpose of the example is to determine the effect of car model and factory on the mileage rating of cars.

```
load mileage
mileage

mileage =

   33.3000   34.5000   37.4000
   33.4000   34.8000   36.8000
   32.9000   33.8000   37.6000
   32.6000   33.4000   36.6000
   32.5000   33.7000   37.0000
   33.0000   33.9000   36.7000

cars = 3;
[p,tbl,stats] = anova2(mileage,cars);
p

p =
    0.0000    0.0039    0.8411
```

There are three models of cars (columns) and two factories (rows). The reason there are six rows in `mileage` instead of two is that each factory provides three cars of each model for the study. The data from the first factory is in the first three rows, and the data from the second factory is in the last three rows.

The standard ANOVA table has columns for the sums of squares, degrees-of-freedom, mean squares (SS/df), F statistics, and p-values.



```
                        ANOVA Table
Source          SS      df     MS        F      Prob>F
------------------------------------------------------
Columns      53.3511     2   26.6756   234.22   0
Rows          1.445      1    1.445     12.69   0.0039
Interaction   0.04       2    0.02       0.18   0.8411
Error         1.3667    12    0.1139
Total        56.2028    17
```

You can use the F statistics to do hypotheses tests to find out if the mileage is the same across models, factories, and model-factory pairs (after adjusting for the additive effects). `anova2` returns the p-value from these tests.

The p-value for the model effect is zero to four decimal places. This is a strong indication that the mileage varies from one model to another. An F statistic as extreme as the observed F would occur by chance less than once in 10,000 times if the gas mileage were truly equal from model to model. If you used the `multcompare` function to perform a multiple comparison test, you would find that each pair of the three models is significantly different.

The p-value for the factory effect is 0.0039, which is also highly significant. This indicates that one factory is out-performing the other in the gas mileage of the cars it produces. The observed p-value indicates that an F statistic as extreme as the observed F would occur by chance about four out of 1000 times if the gas mileage were truly equal from factory to factory.

There does not appear to be any interaction between factories and models. The p-value, 0.8411, means that the observed result is quite likely (84 out 100 times) given that there is no interaction.

The p-values returned by `anova2` depend on assumptions about the random disturbances ε_{ijk} in the model equation. For the p-values to be correct these disturbances need to be independent, normally distributed, and have constant variance. See "Robust and Nonparametric Methods" on page 7-25 for nonparametric methods that do not require a normal distribution.

In addition, `anova2` requires that data be *balanced*, which in this case means there must be the same number of cars for each combination of model and factory. The next section discusses a function that supports unbalanced data with any number of predictors.

N-Way Analysis of Variance

You can use N-way ANOVA to determine if the means in a set of data differ when grouped by multiple factors. If they do differ, you can determine which factors or combinations of factors are associated with the difference.

N-way ANOVA is a generalization of two-way ANOVA. For three factors, the model can be written

$$y_{ijkl} = \mu + \alpha_{.j.} + \beta_{i..} + \gamma_{..k} + (\alpha\beta)_{ij.} + (\alpha\gamma)_{i.k} + (\beta\gamma)_{.jk} + (\alpha\beta\gamma)_{ijk} + \varepsilon_{ijkl}$$

In this notation parameters with two subscripts, such as $(\alpha\beta)_{ij.}$, represent the interaction effect of two factors. The parameter $(\alpha\beta\gamma)_{ijk}$ represents the three-way interaction. An ANOVA model can have the full set of parameters or any subset, but conventionally it does not include complex interaction terms unless it also includes all simpler terms for those factors. For example, one would generally not include the three-way interaction without also including all two-way interactions.

The `anovan` function performs N-way ANOVA. Unlike the `anova1` and `anova2` functions, `anovan` does not expect data in a tabular form. Instead, it expects a vector of response measurements and a separate vector (or text array) containing the values corresponding to each factor. This input data format is more convenient than matrices when there are more than two factors or when the number of measurements per factor combination is not constant.

The following examples explore `anovan` in greater detail:

- "Example: N-Way ANOVA with a Small Data Set" on page 7-40
- "Example: N-Way ANOVA with a Large Data Set" on page 7-42
- "Example: ANOVA with Random Effects" on page 7-46

Example: N-Way ANOVA with a Small Data Set

Consider the following two-way example using `anova2`.

```
m = [23 15 20;27 17 63;43 3 55;41 9 90]
m =
    23    15    20
    27    17    63
    43     3    55
    41     9    90

anova2(m,2)

ans =
    0.0197    0.2234    0.2663
```

The factor information is implied by the shape of the matrix `m` and the number of measurements at each factor combination (2). Although `anova2` does not actually require arrays of factor values, for illustrative purposes you could create them as follows.

```
cfactor = repmat(1:3,4,1)

cfactor =
     1     2     3
     1     2     3
     1     2     3
     1     2     3

rfactor = [ones(2,3); 2*ones(2,3)]

rfactor =

     1     1     1
     1     1     1
     2     2     2
     2     2     2
```

The `cfactor` matrix shows that each column of `m` represents a different level of the column factor. The `rfactor` matrix shows that the top two rows of `m` represent one level of the row factor, and bottom two rows of `m` represent a second level of the row factor. In other words, each value `m(i,j)` represents an observation at column factor level `cfactor(i,j)` and row factor level `rfactor(i,j)`.

To solve the above problem with anovan, you need to reshape the matrices m, cfactor, and rfactor to be vectors.

```
m = m(:);
cfactor = cfactor(:);
rfactor = rfactor(:);

[m cfactor rfactor]

ans =

    23     1     1
    27     1     1
    43     1     2
    41     1     2
    15     2     1
    17     2     1
     3     2     2
     9     2     2
    20     3     1
    63     3     1
    55     3     2
    90     3     2

anovan(m,{cfactor rfactor},2)

ans =

    0.0197
    0.2234
    0.2663
```

Example: N-Way ANOVA with a Large Data Set

The previous example used anova2 to study a small data set measuring car mileage. This example illustrates how to analyze a larger set of car data with mileage and other information on 406 cars made between 1970 and 1982. First, load the data set and look at the variable names.

```
load carbig
whos
```

```
  Name                  Size           Bytes  Class

  Acceleration     406x1             3248  double array
  Cylinders        406x1             3248  double array
  Displacement     406x1             3248  double array
  Horsepower       406x1             3248  double array
  MPG              406x1             3248  double array
  Model            406x36           29232  char array
  Model_Year       406x1             3248  double array
  Origin           406x7             5684  char array
  Weight           406x1             3248  double array
  cyl4             406x5             4060  char array
  org              406x7             5684  char array
  when             406x5             4060  char array
```

The example focusses on four variables. `MPG` is the number of miles per gallon for each of 406 cars (though some have missing values coded as `NaN`). The other three variables are factors: `cyl4` (four-cylinder car or not), `org` (car originated in Europe, Japan, or the USA), and `when` (car was built early in the period, in the middle of the period, or late in the period).

First, fit the full model, requesting up to three-way interactions and Type 3 sums-of-squares.

```
varnames = {'Origin';'4Cyl';'MfgDate'};
anovan(MPG,{org cyl4 when},3,3,varnames)

ans =
    0.0000
       NaN
         0
    0.7032
    0.0001
    0.2072
    0.6990
```

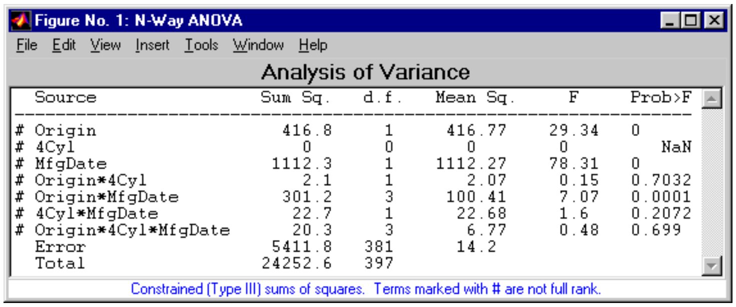


Figure No. 1: N-Way ANOVA
File Edit View Insert Tools Window Help

Analysis of Variance

```
  Source                     Sum Sq.   d.f.   Mean Sq.     F      Prob>F
------------------------------------------------------------------------
# Origin                       416.8     1     416.77    29.34   0
# 4Cyl                           0       0       0        0         NaN
# MfgDate                     1112.3     1    1112.27    78.31   0
# Origin*4Cyl                    2.1     1       2.07     0.15   0.7032
# Origin*MfgDate               301.2     3     100.41     7.07   0.0001
# 4Cyl*MfgDate                  22.7     1      22.68     1.6    0.2072
# Origin*4Cyl*MfgDate           20.3     3       6.77     0.48   0.699
  Error                       5411.8   381      14.2
  Total                      24252.6   397
```

Constrained (Type III) sums of squares. Terms marked with # are not full rank.

Note that many terms are marked by a # symbol as not having full rank, and one of them has zero degrees of freedom and is missing a p-value. This can happen when there are missing factor combinations and the model has higher-order terms. In this case, the cross-tabulation below shows that there are no cars made in Europe during the early part of the period with other than four cylinders, as indicated by the 0 in `table(2,1,1)`.

```
[table, chi2, p, factorvals] = crosstab(org,when,cyl4)

table(:,:,1) =

    82    75    25
     0     4     3
     3     3     4

table(:,:,2) =
    12    22    38
    23    26    17
    12    25    32

chi2 =

  207.7689

p =

     0

factorvals =
'USA'         'Early'     'Other'
'Europe'      'Mid'       'Four'
'Japan'       'Late'           []
```

Consequently it is impossible to estimate the three-way interaction effects, and including the three-way interaction term in the model makes the fit singular.

Using even the limited information available in the ANOVA table, you can see that the three-way interaction has a p-value of 0.699, so it is not significant. So this time you examine only two-way interactions.

```
[p,tbl,stats,terms] = anovan(MPG,{org cyl4 when},2,3,varnames);
terms

terms =
     1     0     0
     0     1     0
     0     0     1
     1     1     0
     1     0     1
     0     1     1
```

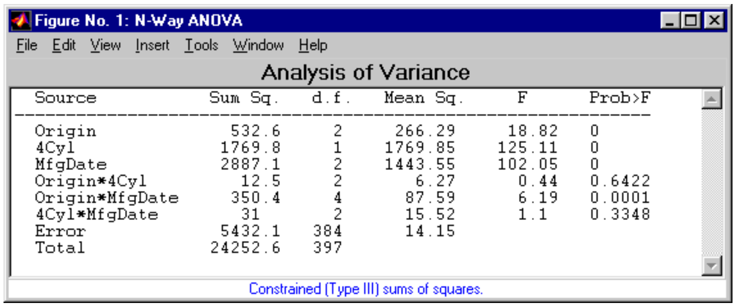


Figure No. 1: N-Way ANOVA
File Edit View Insert Tools Window Help

Analysis of Variance

```
Source           Sum Sq.   d.f.   Mean Sq.     F      Prob>F
-------------------------------------------------------------
Origin             532.6     2     266.29    18.82    0
4Cyl              1769.8     1    1769.85   125.11    0
MfgDate           2887.1     2    1443.55   102.05    0
Origin*4Cyl         12.5     2       6.27     0.44    0.6422
Origin*MfgDate     350.4     4      87.59     6.19    0.0001
4Cyl*MfgDate        31       2      15.52     1.1     0.3348
Error             5432.1   384      14.15
Total            24252.6   397
```

Constrained (Type III) sums of squares.

Now all terms are estimable. The p-values for interaction term 4 (`Origin*4Cyl`) and interaction term 6 (`4Cyl*MfgDate`) are much larger than a typical cutoff value of 0.05, indicating these terms are not significant. You could choose to omit these terms and pool their effects into the error term. The output `terms` variable returns a matrix of codes, each of which is a bit pattern representing a term. You can omit terms from the model by deleting their entries from terms and running anovan again, this time supplying the resulting vector as the model argument.

```
terms([4 6],:) = []

terms =

     1     0     0
     0     1     0
     0     0     1
     1     0     1

anovan(MPG,{org cyl4 when},terms,3,varnames)

ans =

  1.0e-003 *

    0.0000
         0
         0
    0.1140
```

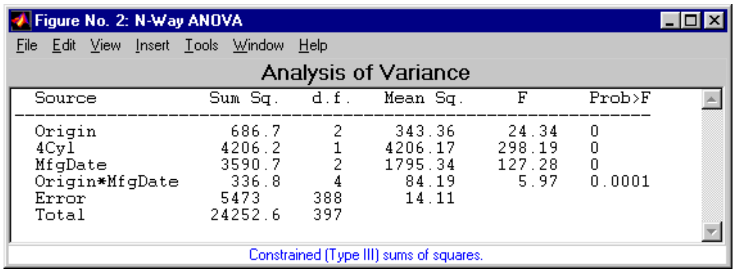


Figure No. 2: N-Way ANOVA
File Edit View Insert Tools Window Help

Analysis of Variance

```
Source            Sum Sq.   d.f.   Mean Sq.     F      Prob>F
--------------------------------------------------------------
Origin              686.7     2     343.36    24.34   0
4Cyl               4206.2     1    4206.17   298.19   0
MfgDate            3590.7     2    1795.34   127.28   0
Origin*MfgDate      336.8     4      84.19     5.97   0.0001
Error              5473     388      14.11
Total             24252.6   397
```

Constrained (Type III) sums of squares.

Now you have a more parsimonious model indicating that the mileage of these cars seems to be related to all three factors, and that the effect of the manufacturing date depends on where the car was made.

Example: ANOVA with Random Effects

In an ordinary ANOVA model, each grouping variable represents a fixed factor. The levels of that factor are a fixed set of values. Your goal is to determine whether different factor levels lead to different response values. This section presents an example that shows how to use anovan to fit models where a factor's levels represent a random selection from a larger (infinite) set of possible levels.

This section covers the following topics:

Setting Up the Model. To set up the example, first load the data, which is stored in a 6-by-3 matrix, `mileage`.

```
load mileage
```

The `anova2` function works only with balanced data, and it infers the values of the grouping variables from the row and column numbers of the input matrix. The `anovan` function, on the other hand, requires you to explicitly create vectors of grouping variable values. To create these vectors, do the following steps:

1. Create an array indicating the factory for each value in `mileage`. This array is 1 for the first column, 2 for the second, and 3 for the third.

   ```
   factory  = repmat(1:3,6,1);
   ```

2. Create an array indicating the car model for each mileage value. This array is 1 for the first three rows of mileage, and 2 for the remaining three rows.

   ```
   carmod = [ones(3,3); 2*ones(3,3)];
   ```

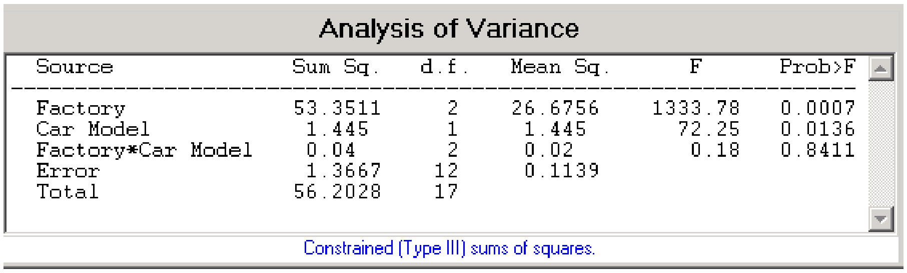
3. Turn these matrices into vectors and display them.

   ```
   mileage = mileage(:);
   factory = factory(:);
   carmod = carmod(:);
   [mileage factory carmod]
   ```

```
ans =

   33.3000    1.0000    1.0000
   33.4000    1.0000    1.0000
   32.9000    1.0000    1.0000
   32.6000    1.0000    2.0000
   32.5000    1.0000    2.0000
   33.0000    1.0000    2.0000
   34.5000    2.0000    1.0000
   34.8000    2.0000    1.0000
   33.8000    2.0000    1.0000
   33.4000    2.0000    2.0000
   33.7000    2.0000    2.0000
   33.9000    2.0000    2.0000
   37.4000    3.0000    1.0000
   36.8000    3.0000    1.0000
   37.6000    3.0000    1.0000
   36.6000    3.0000    2.0000
   37.0000    3.0000    2.0000
   36.7000    3.0000    2.0000
```

Fitting a Random Effects Model. Continuing the example from the preceding section, suppose you are studying a few factories but you want information about what would happen if you build these same car models in a different factory—either one that you already have or another that you might construct. To get this information, fit the analysis of variance model, specifying a model that includes an interaction term and that the `factory` factor is random.

```
[pvals,tbl,stats] = anovan(mileage, {factory carmod}, ...
'model',2, 'random',1,'varnames',{'Factory' 'Car Model'});
```


Analysis of Variance

```
  Source                Sum Sq.   d.f.   Mean Sq.       F      Prob>F
-----------------------------------------------------------------------
  Factory               53.3511     2    26.6756    1333.78   0.0007
  Car Model              1.445      1     1.445       72.25   0.0136
  Factory*Car Model      0.04       2     0.02         0.18   0.8411
  Error                  1.3667    12     0.1139
  Total                 56.2028    17
```

Constrained (Type III) sums of squares.

In the fixed effects version of this fit, which you get by omitting the inputs `'random',1` in the preceding code, the effect of car model is significant, with a p-value of 0.0039. But in this example, which takes into account the random variation of the effect of the variable `'Car Model'` from one factory to another, the effect is still significant, but with a higher p-value of 0.0136.

F Statistics for Models with Random Effects. The F statistic in a model having random effects is defined differently than in a model having all fixed effects. In the fixed effects model, you compute the F statistic for any term by taking the ratio of the mean square for that term with the mean square for error. In a random effects model, however, some F statistics use a different mean square in the denominator.

In the example described in "Setting Up the Model" on page 7-47, the effect of the variable `'Factory'` could vary across car models. In this case, the interaction mean square takes the place of the error mean square in the F statistic. The F statistic for factory is

```
F = 1.445 / 0.02

F =

   72.2500
```

The degrees of freedom for the statistic are the degrees of freedom for the numerator (1) and denominator (2) mean squares. Therefore the p-value for the statistic is

```
pval = 1 - fcdf(F,1,2)

pval =

    0.0136
```

With random effects, the expected value of each mean square depends not only on the variance of the error term, but also on the variances contributed by the random effects. You can see these dependencies by writing the expected values as linear combinations of contributions from the various model terms. To find the coefficients of these linear combinations, enter `stats.ems`, which returns the `ems` field of the `stats` structure.

```
stats.ems

ans =

    6.0000    0.0000    3.0000    1.0000
    0.0000    9.0000    3.0000    1.0000
    0.0000    0.0000    3.0000    1.0000
         0         0         0    1.0000
```

To see text representations of the linear combinations, enter

```
stats.txtems

ans =

    '6*V(Factory)+3*V(Factory*Car Model)+V(Error)'
    '9*Q(Car Model)+3*V(Factory*Car Model)+V(Error)'
    '3*V(Factory*Car Model)+V(Error)'
    'V(Error)'
```

The expected value for the mean square due to car model (second term) includes contributions from a quadratic function of the car model effects, plus three times the variance of the interaction term's effect, plus the variance of the error term. Notice that if the car model effects were all zero, the expression would reduce to the expected mean square for the third term (the interaction term). That is why the F statistic for the car model effect uses the interaction mean square in the denominator.

In some cases there is no single term whose expected value matches the one required for the denominator of the F statistic. In that case, the denominator is a linear combination of mean squares. The `stats` structure contains fields giving the definitions of the denominators for each F statistic. The `txtdenom` field, `stats.txtdenom`, gives a text representation, and the `denom` field gives a matrix that defines a linear combination of the variances of terms in the model. For balanced models like this one, the `denom` matrix, `stats.denom`, contains zeros and ones, because the denominator is just a single term's mean square.

```
stats.txtdenom

ans =
    'MS(Factory*Car Model)'
    'MS(Factory*Car Model)'
    'MS(Error)'

stats.denom

ans =

   -0.0000    1.0000    0.0000
    0.0000    1.0000   -0.0000
    0.0000         0    1.0000
```

Variance Components. For the model described in "Setting Up the Model" on page 7-47, consider the mileage for a particular car of a particular model made at a random factory. The variance of that car is the sum of components, or contributions, one from each of the random terms.

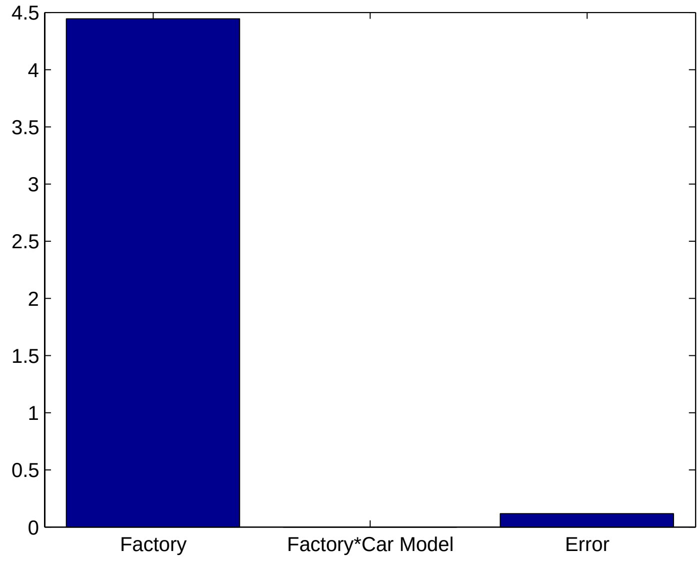
```
stats.rtnames

ans =

    'Factory'
    'Factory*Car Model'
    'Error'
```

You do not know those variances, but you can estimate them from the data. Recall that the `ems` field of the `stats` structure expresses the expected value of each term's mean square as a linear combination of unknown variances for random terms, and unknown quadratic forms for fixed terms. If you take the expected mean square expressions for the random terms, and equate those expected values to the computed mean squares, you get a system of equations that you can solve for the unknown variances. These solutions are the variance component estimates. The `varest` field contains a variance component estimate for each term. The `rtnames` field contains the names of the random terms.

```
stats.varest

ans =
 4.4426
-0.0313
 0.1139
```

Under some conditions, the variability attributed to a term is unusually low, and that term's variance component estimate is negative. In those cases it is common to set the estimate to zero, which you might do, for example, to create a bar graph of the components.

```
bar(max(0,stats.varest))
set(gca,'xtick',1:3,'xticklabel',stats.rtnames)
bar(max(0,stats.varest))
```



You can also compute confidence bounds for the variance estimate. The anovan function does this by computing confidence bounds for the variance expected mean squares, and finding lower and upper limits on each variance component containing all of these bounds. This procedure leads to a set of bounds that is conservative for balanced data. (That is, 95% confidence bounds will have a probability of at least 95% of containing the true variances if the number of observations for each combination of grouping variables is the same.) For unbalanced data, these are approximations that are not guaranteed to be conservative.

```
[{'Term' 'Estimate' 'Lower' 'Upper'};
 stats.rtnames, num2cell([stats.varest stats.varci])]

ans =

    'Term'                   'Estimate'    'Lower'     'Upper'
    'Factory'                [  4.4426]    [1.0736]    [175.6038]
    'Factory*Car Model'      [ -0.0313]    [   NaN]    [     NaN]
    'Error'                  [  0.1139]    [0.0586]    [  0.3103]
```

Other ANOVA Models

The anovan function also has arguments that enable you to specify two other types of model terms. First, the 'nested' argument specifies a matrix that indicates which factors are nested within other factors. A nested factor is one that takes different values within each level its nested factor.

For example, the mileage data from the previous section assumed that the two car models produced in each factory were the same. Suppose instead, each factory produced two distinct car models for a total of six car models, and we numbered them 1 and 2 for each factory for convenience. Then, the car model is nested in factory. A more accurate and less ambiguous numbering of car model would be as follows:

Factory	Car Model
1	1
1	2
2	3
2	4
3	5
3	6

However, it is common with nested models to number the nested factor the same way in each nested factor.

Second, the `'continuous'` argument specifies that some factors are to be treated as continuous variables. The remaining factors are categorical variables. Although the `anovan` function can fit models with multiple continuous and categorical predictors, the simplest model that combines one predictor of each type is known as an *analysis of covariance* model. The next section describes a specialized tool for fitting this model.

Analysis of Covariance

Analysis of covariance is a technique for analyzing grouped data having a response (y, the variable to be predicted) and a predictor (x, the variable used to do the prediction). Using analysis of covariance, you can model y as a linear function of x, with the coefficients of the line possibly varying from group to group.

The aoctool Demo

The `aoctool` demo is an interactive graphical environment for fitting and prediction with analysis of covariance (ANOCOVA) models. It is similar to the `polytool` demo. The `aoctool` function fits the following models for the ith group:

Same mean	$y = \alpha + \varepsilon$
Separate means	$y = (\alpha + \alpha_i) + \varepsilon$
Same line	$y = \alpha + \beta x + \varepsilon$
Parallel lines	$y = (\alpha + \alpha_i) + \beta x + \varepsilon$
Separate lines	$y = (\alpha + \alpha_i) + (\beta + \beta_i)x + \varepsilon$

In the parallel lines model, for example, the intercept varies from one group to the next, but the slope is the same for each group. In the same mean model, there is a common intercept and no slope. In order to make the group coefficients well determined, the demo imposes the constraints

$$\sum \alpha_j = \sum \beta_j = 0$$

The following sections provide an illustrative example.

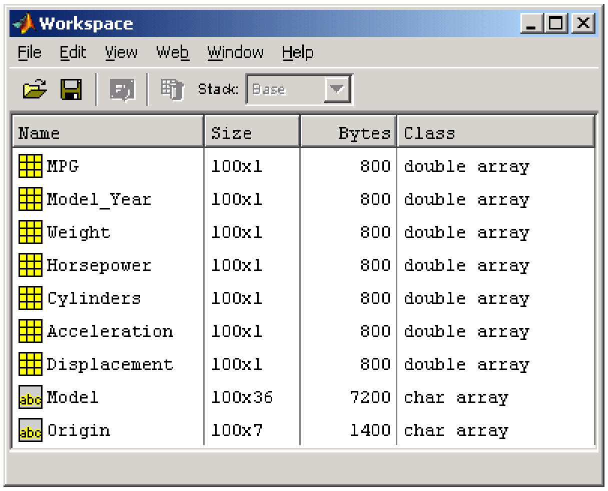
- "Exploring the aoctool Interface" on page 7-55
- "Confidence Bounds" on page 7-58
- "Multiple Comparisons" on page 7-60

Exploring the aoctool Interface.

1 **Load the data.** Statistics Toolbox has a small data set with information about cars from the years 1970, 1976, and 1982. This example studies the relationship between the weight of a car and its mileage, and whether this relationship has changed over the years. To start the demonstration, load the data set.

```
load carsmall
```

The Workspace browser shows the variables in the data set.



Name	Size	Bytes	Class
MPG	100x1	800	double array
Model_Year	100x1	800	double array
Weight	100x1	800	double array
Horsepower	100x1	800	double array
Cylinders	100x1	800	double array
Acceleration	100x1	800	double array
Displacement	100x1	800	double array
Model	100x36	7200	char array
Origin	100x7	1400	char array

You can also use `aoctool` with your own data.

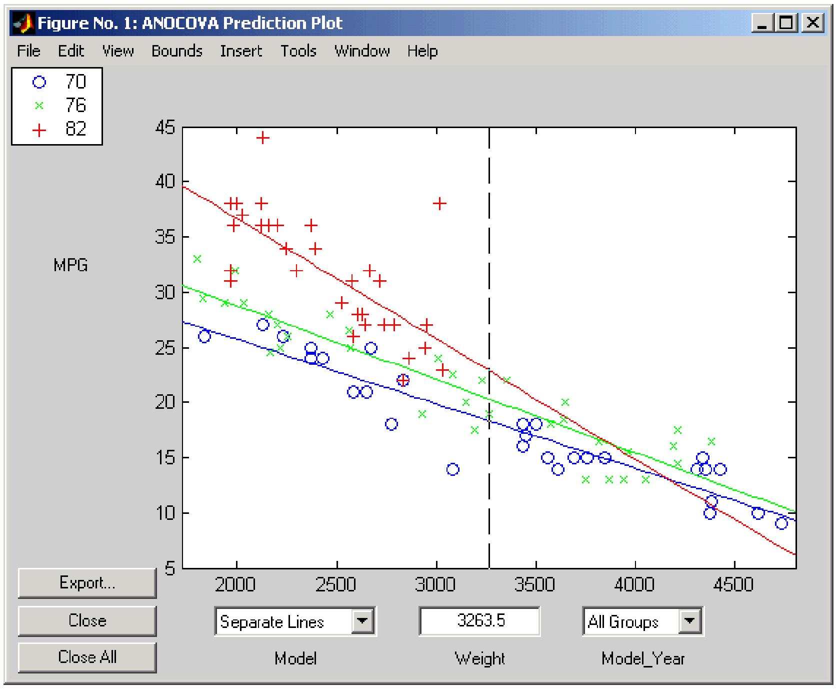
2 **Start the tool.** The following command calls `aoctool` to fit a separate line to the column vectors `Weight` and `MPG` for each of the three model group defined in `Model_Year`. The initial fit models the *y* variable, `MPG`, as a linear function of the *x* variable, `Weight`.

```
[h,atab,ctab,stats] = aoctool(Weight,MPG,Model_Year);
```

```
Note: 6 observations with missing values have been removed.
```

See the `aoctool` function reference page for detailed information about calling `aoctool`.

3 **Examine the output.** The graphical output consists of a main window with a plot, a table of coefficient estimates, and an analysis of variance table. In the plot, each `Model_Year` group has a separate line. The data points for each group are coded with the same color and symbol, and the fit for each group has the same color as the data points.



The coefficients of the three lines appear in the figure titled ANOCOVA Coefficients. You can see that the slopes are roughly -0.0078, with a small deviation for each group:

Model year 1970: $y = (45.9798 - 8.5805) + (-0.0078 + 0.002)x + \varepsilon$

Model year 1976: $y = (45.9798 - 3.8902) + (-0.0078 + 0.0011)x + \varepsilon$

Model year 1982: $y = (45.9798 + 12.4707) + (-0.0078 - 0.0031)x + \varepsilon$

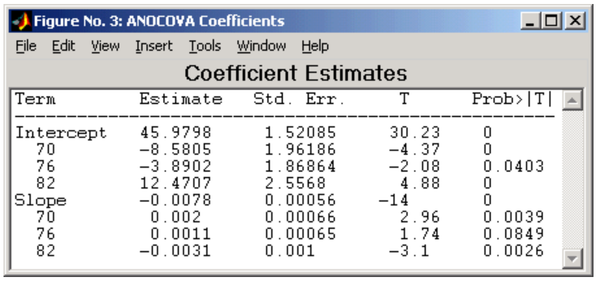


Figure No. 3: ANOCOVA Coefficients

Coefficient Estimates

```
Term          Estimate    Std. Err.      T      Prob>|T|
--------------------------------------------------------
Intercept     45.9798     1.52085      30.23    0
  70          -8.5805     1.96186      -4.37    0
  76          -3.8902     1.86864      -2.08    0.0403
  82          12.4707     2.5568        4.88    0
Slope         -0.0078     0.00056     -14       0
  70           0.002      0.00066       2.96    0.0039
  76           0.0011     0.00065       1.74    0.0849
  82          -0.0031     0.001        -3.1     0.0026
```

Because the three fitted lines have slopes that are roughly similar, you may wonder if they really are the same. The `Model_Year*Weight` interaction expresses the difference in slopes, and the ANOVA table shows a test for the significance of this term. With an F statistic of 5.23 and a p-value of 0.0072, the slopes are significantly different.

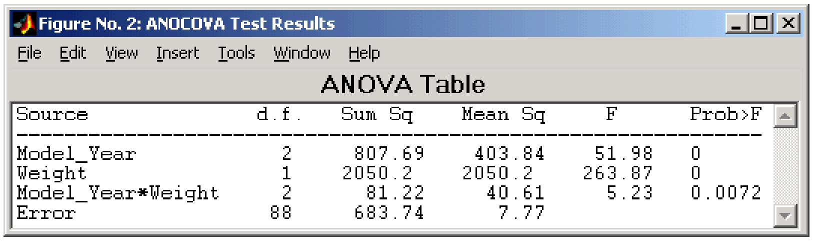
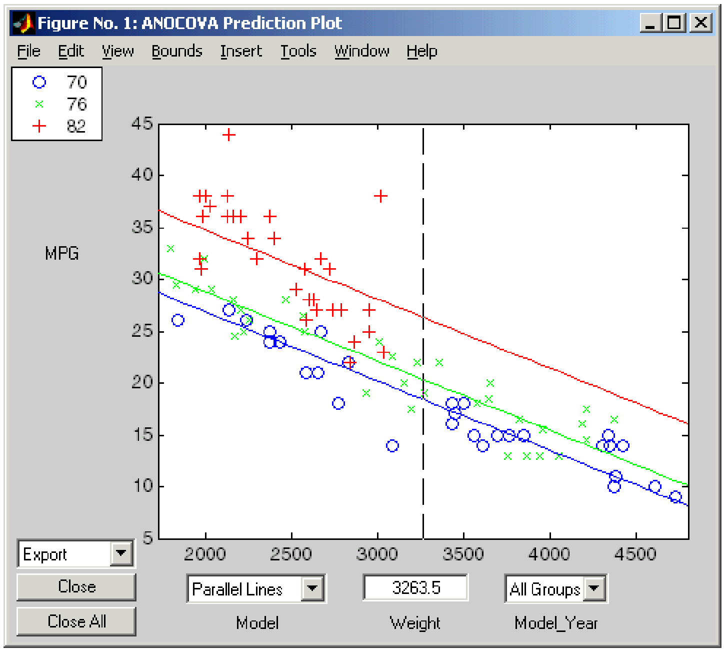


Figure No. 2: ANOCOVA Test Results

ANOVA Table

```
Source               d.f.    Sum Sq     Mean Sq      F       Prob>F
--------------------------------------------------------------------
Model_Year             2      807.69     403.84     51.98    0
Weight                 1     2050.2     2050.2     263.87    0
Model_Year*Weight      2       81.22      40.61      5.23    0.0072
Error                 88      683.74       7.77
```

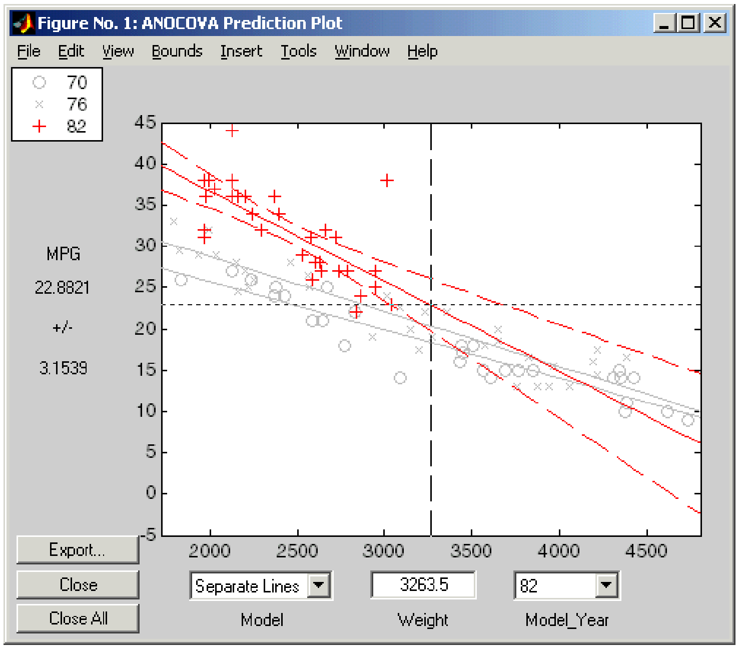
4 **Constrain the slopes to be the same.** To examine the fits when the slopes are constrained to be the same, return to the ANOCOVA Prediction Plot window and use the **Model** pop-up menu to select a `Parallel Lines` model. The window updates to show the following graph.



Though this fit looks reasonable, it is significantly worse than the `Separate Lines` model. Use the **Model** pop-up menu again to return to the original model.

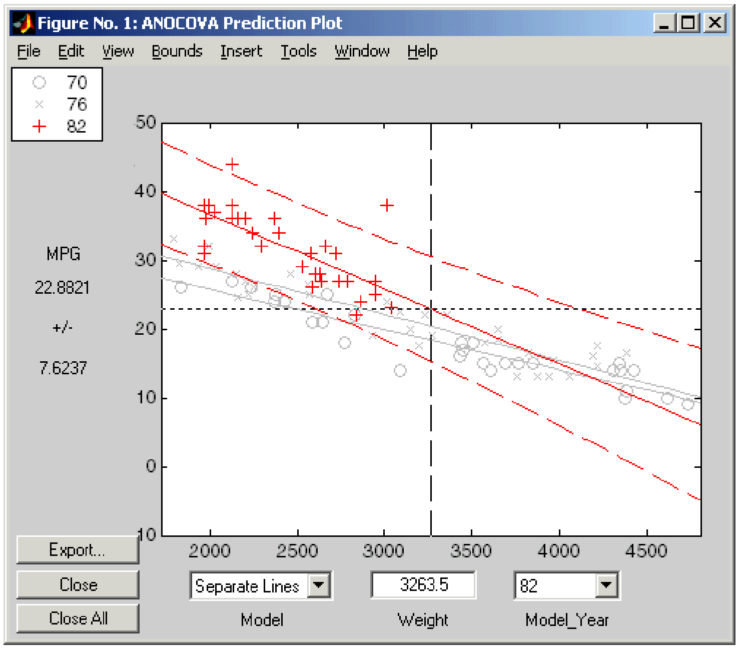
Confidence Bounds. The example in "The aoctool Demo" on page 7-54 provides estimates of the relationship between `MPG` and `Weight` for each `Model_Year`, but how accurate are these estimates? To find out, you can superimpose confidence bounds on the fits by examining them one group at a time.

1 In the **Model_Year** menu at the lower right of the figure, change the setting from `All Groups` to 82. The data and fits for the other groups are dimmed, and confidence bounds appear around the 82 fit.



The dashed lines form an envelope around the fitted line for model year 82. Under the assumption that the true relationship is linear, these bounds provide a 95% confidence region for the true line. Note that the fits for the other model years are well outside these confidence bounds for `Weight` values between `2000` and `3000`.

2 Sometimes it is more valuable to be able to predict the response value for a new observation, not just estimate the average response value. Use the `aoctool` function **Bounds** menu to change the definition of the confidence bounds from `Line` to `Observation`. The resulting wider intervals reflect the uncertainty in the parameter estimates as well as the randomness of a new observation.



Like the polytool function, the aoctool function has cross hairs that you can use to manipulate the Weight and watch the estimate and confidence bounds along the *y*-axis update. These values appear only when a single group is selected, not when All Groups is selected.

Multiple Comparisons. You can perform a multiple comparison test by using the stats output structure from aoctool as input to the multcompare function. The multcompare function can test either slopes, intercepts, or population marginal means (the predicted MPG of the mean weight for each group). The example in “The aoctool Demo” on page 7-54 shows that the slopes are not all the same, but could it be that two are the same and only the other one is different? You can test that hypothesis.

```
multcompare(stats,0.05,'on','','s')

ans =
    1.0000    2.0000   -0.0012    0.0008    0.0029
    1.0000    3.0000    0.0013    0.0051    0.0088
    2.0000    3.0000    0.0005    0.0042    0.0079
```

This matrix shows that the estimated difference between the intercepts of groups 1 and 2 (1970 and 1976) is 0.0008, and a confidence interval for the difference is [-0.0012, 0.0029]. There is no significant difference between the two. There are significant differences, however, between the intercept for 1982 and each of the other two. The graph shows the same information.

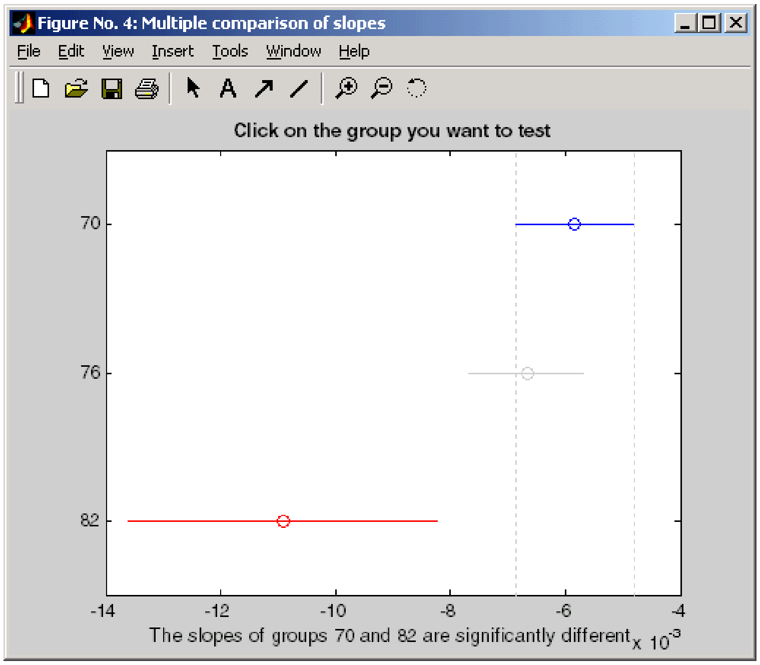


Figure No. 4: Multiple comparison of slopes
File Edit View Insert Tools Window Help
Click on the group you want to test
70
76
82
-14
-12
-10
-8
-6
-4
The slopes of groups 70 and 82 are significantly different
x 10^-3

Note that the `stats` structure was created in the initial call to the `aoctool` function, so it is based on the initial model fit (typically a separate-lines model). If you change the model interactively and want to base your multiple comparisons on the new model, you need to run `aoctool` again to get another `stats` structure, this time specifying your new model as the initial model.

Nonlinear Models

Parametric Models

Introduction

Parametric nonlinear models represent the relationship between a continuous response variable and one or more predictor variables (either continuous or categorical) in the form $y = f(X, \beta) + \epsilon$, where

- y is an n by-1 vector of observations of the response variable.
- X is an n-by-p design matrix determined by the predictors.
- β is a p-by-1 vector of unknown parameters to be estimated.
- f is any function of X and β.
- ε is an n-by-1 vector of independent, identically distributed random disturbances.

Nonlinear Regression

The Hougen-Watson model (Bates and Watts, [2], pp. 271–272) for reaction kinetics is an example of a parametric nonlinear model. The form of the model is

$$rate = \frac{\beta_1 \cdot x_2 - x_3/\beta_5}{1 + \beta_2 \cdot x_1 + \beta_3 \cdot x_2 + \beta_4 \cdot x_3}$$

where *rate* is the reaction rate, x_1, x_2, and x_3 are concentrations of hydrogen, n-pentane, and isopentane, respectively, and β_1, β_2, ... , β_5 are the unknown parameters.

The file `reaction.mat` contains simulated data from a reaction appropriate for this model:

```
load reaction

who
Your variables are:
beta        rate        xn
model       reactants   yn
```

The variables are

- `rate` — A 13-by-1 vector of observed reaction rates
- `reactants` — A 13-by-3 matrix of reactant concentrations
- `beta` — A 5-by-1 vector of initial parameter estimates
- `model` — The name of an M-file function for the model
- `xn` — The names of the reactants
- `yn` — The name of the response

The M-file function for the model is `hougen`, which looks like this:

```
type hougen

function yhat = hougen(beta,x)
%HOUGEN Hougen-Watson model for reaction kinetics.
%   YHAT = HOUGEN(BETA,X) gives the predicted values of the
%   reaction rate, YHAT, as a function of the vector of
%   parameters, BETA, and the matrix of data, X.
%   BETA must have five elements and X must have three
%   columns.
%
%   The model form is:
%   y = (b1*x2 - x3/b5)./(1+b2*x1+b3*x2+b4*x3)

b1 = beta(1);
b2 = beta(2);
b3 = beta(3);
b4 = beta(4);
b5 = beta(5);

x1 = x(:,1);
```

```
x2 = x(:,2);
x3 = x(:,3);

yhat = (b1*x2 - x3/b5)./(1+b2*x1+b3*x2+b4*x3);
```

The function `nlinfit` is used to find least-squares parameter estimates for nonlinear models. It uses the Gauss-Newton algorithm with Levenberg-Marquardt modifications for global convergence.

`nlinfit` requires the predictor data, the responses, and an initial guess of the unknown parameters. It also requires a function handle to a function that takes the predictor data and parameter estimates and returns the responses predicted by the model.

To fit the `reaction` data, call `nlinfit` using the following syntax:

```
load reaction
betahat = nlinfit(reactants,rate,@hougen,beta)
betahat =
    1.2526
    0.0628
    0.0400
    0.1124
    1.1914
```

The output vector `betahat` contains the parameter estimates.

Confidence Intervals for Parameter Estimates

To compute confidence intervals for the parameter estimates, use the function `nlparci`, together with additional outputs from `nlinfit`:

```
[betahat,resid,J] = nlinfit(reactants,rate,@hougen,beta);
betaci = nlparci(betahat,resid,J)
betaci =
   -0.7467    3.2519
   -0.0377    0.1632
   -0.0312    0.1113
   -0.0609    0.2857
   -0.7381    3.1208
```

The columns of the output betaci contain the lower and upper bounds, respectively, of the (default) 95% confidence intervals for each parameter.

Confidence Intervals for Predicted Responses

The function nlpredci is used to compute confidence intervals for predicted responses:

```
[yhat,delta] = nlpredci(@hougen,reactants,betahat,resid,J);
opd = [rate yhat delta]
opd =
    8.5500    8.2937    0.9178
    3.7900    3.8584    0.7244
    4.8200    4.7950    0.8267
    0.0200   -0.0725    0.4775
    2.7500    2.5687    0.4987
   14.3900   14.2227    0.9666
    2.5400    2.4393    0.9247
    4.3500    3.9360    0.7327
   13.0000   12.9440    0.7210
    8.5000    8.2670    0.9459
    0.0500   -0.1437    0.9537
   11.3200   11.3484    0.9228
    3.1300    3.3145    0.8418
```

The output opd contains the observed rates in the first column and the predicted rates in the second column. The (default) 95% simultaneous confidence intervals on the predictions are the values in the second column ± the values in the third column. These are not intervals for new observations at the predictors, even though most of the confidence intervals do contain the original observations.

Interactive Nonlinear Regression

Calling nlintool opens a graphical user interface (GUI) for interactive exploration of multidimensional nonlinear functions, and for fitting parametric nonlinear models. The GUI calls nlinfit, and requires the same inputs. The interface is analogous to polytool and rstool for polynomial models.

Open `nlintool` with the `reaction` data and the `hougen` model by typing

```
nlintool(reactants,rate,@hougen,beta,0.01,xn,yn)
```

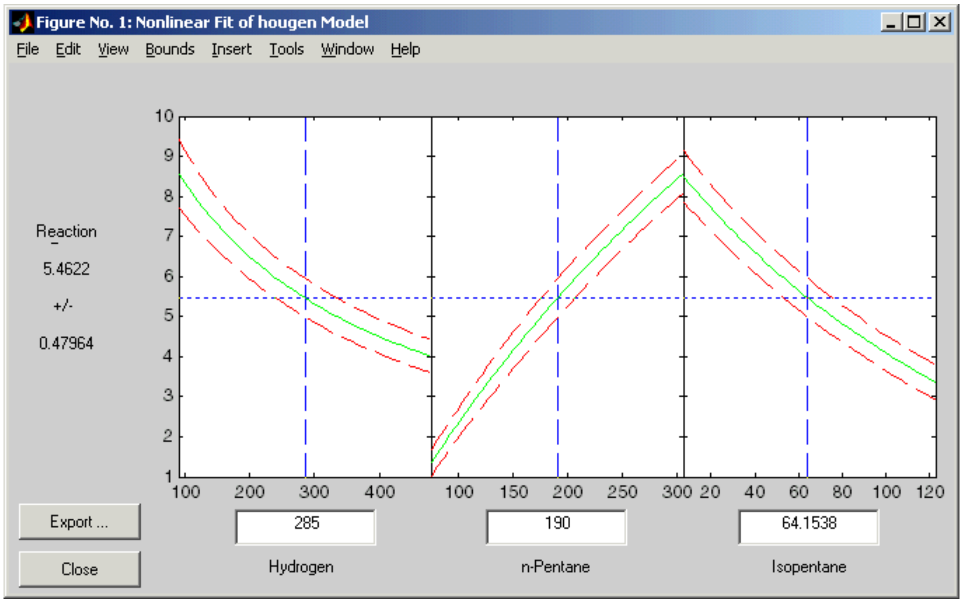


Figure No. 1: Nonlinear Fit of hougen Model
File Edit View Bounds Insert Tools Window Help
Reaction
5.4622
+/-
0.47964
Export ...
Close
285
Hydrogen
190
n-Pentane
64.1538
Isopentane

You see three plots. The response variable for all plots is the reaction rate, plotted in green. The red lines show confidence intervals on predicted responses. The first plot shows hydrogen as the predictor, the second shows *n*-pentane, and the third shows isopentane.

Each plot displays the fitted relationship of the reaction rate to one predictor at a fixed value of the other two predictors. The fixed values are in the text boxes below each predictor axis. Change the fixed values by typing in a new value or by dragging the vertical lines in the plots to new positions. When you change the value of a predictor, all the plots update to display the model at the new point in predictor space.

While this example uses only three predictors, `nlintool` can accommodate any number of predictors.

Nonparametric Models

Introduction

Parametric models specify the form of the relationship between predictors and a response, as in the Hougen-Watson model described in "Parametric Models" on page 8-2. In many cases, however, the form of the relationship is unknown, and a parametric model requires assumptions and simplifications. Regression trees offer a nonparametric alternative. When response data is categorical, classification trees are a natural modification.

Note This section demonstrates methods for classification and regression tree objects created with the `classregtree` constructor. These methods supersede the functions `treefit`, `treedisp`, `treeval`, `treefit`, `treeprune`, and `treetest`, which are maintained in Statistics Toolbox only for backwards compatibility.

Algorithm Reference

The algorithms used by the classification and regression tree functions in Statistics Toolbox are based on those in Breiman, L., et al., *Classification and Regression Trees*, Chapman & Hall, Boca Raton, 1993.

Regression Trees

This example uses the data on cars in `carsmall.mat` to create a regression tree for predicting mileage using measurements of weight and the number of cylinders as predictors. Note that, in this case, one predictor (weight) is continuous and the other (cylinders) is categorical. The response (mileage) is continuous.

Load the data and use the `classregtree` constructor to create the regression tree:

```
load carsmall

t = classregtree([Weight, Cylinders],MPG,...
                 'cat',2,'splitmin',20,...
                 'names',{'Weight','Cylinders'})
t =
Decision tree for regression
 1  if Weight<3085.5 then node 2 else node 3
 2  if Weight<2371 then node 4 else node 5
 3  if Cylinders=8 then node 6 else node 7
 4  if Weight<2162 then node 8 else node 9
 5  if Cylinders=6 then node 10 else node 11
 6  if Weight<4381 then node 12 else node 13
 7  fit = 19.2778
 8  fit = 33.3056
 9  fit = 29.6111
10  fit = 23.25
11  if Weight<2827.5 then node 14 else node 15
12  if Weight<3533.5 then node 16 else node 17
13  fit = 11
14  fit = 27.6389
15  fit = 24.6667
16  fit = 16.6
17  fit = 14.3889
```

t is a classregtree object and can be operated on with any of the methods of the class.

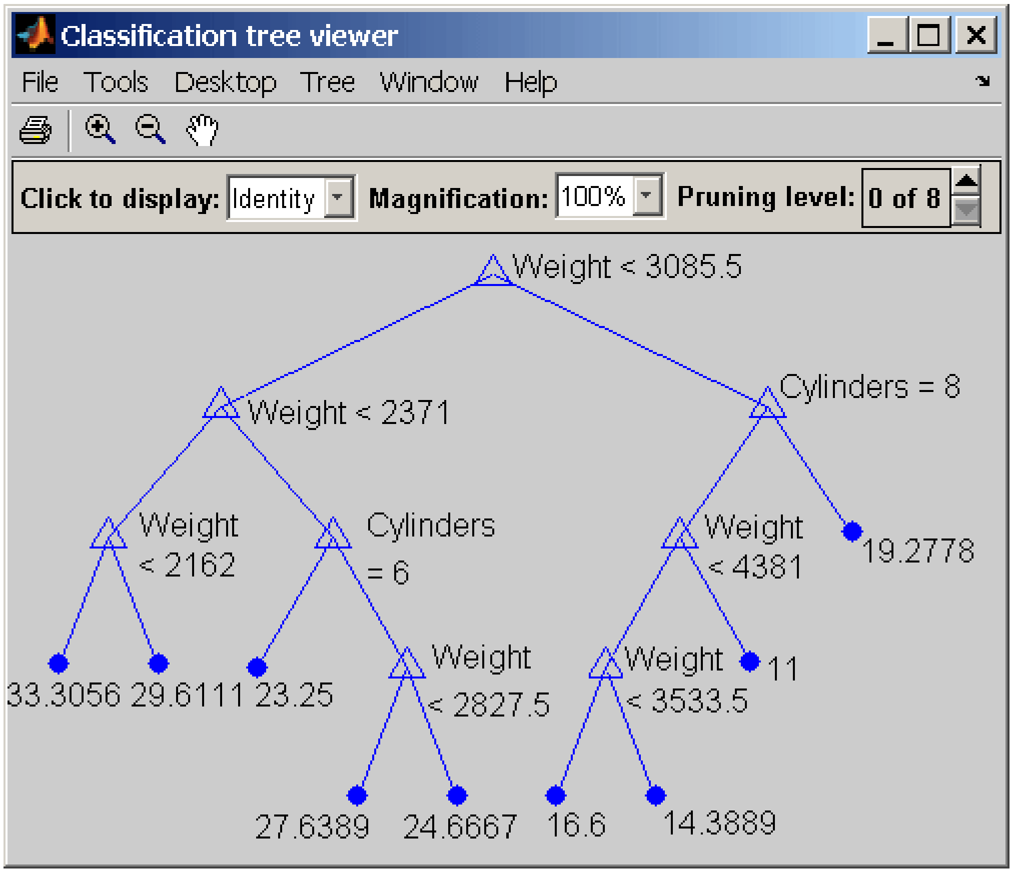
Use the type method to show the type of the tree:

```
treetype = type(t)
treetype =
regression
```

classregtree creates a regression tree because MPG is a numerical vector, and the response is assumed to be continuous.

To view the tree, use the view method:

```
view(t)
```



The tree predicts the response values at the circular leaf nodes based on a series of questions about the car at the triangular branching nodes. A `true` answer to any question follows the branch to the left; a `false` follows the branch to the right.

Use the tree to predict the mileage for a 2000-pound car with either 4, 6, or 8 cylinders:

```
mileage2K = t([2000 4; 2000 6; 2000 8])
mileage2K =
   33.3056
   33.3056
   33.3056
```

Note that the object allows for functional evaluation, of the form `t(X)`. This is a shorthand way of calling the `eval` method.

The predicted responses computed above are all the same. This is because they follow a series of splits in the tree that depend only on weight, terminating at the left-most leaf node in the view above. A 4000-pound car, following the right branch from the top of the tree, leads to different predicted responses:

```
mileage4K = t([4000 4; 4000 6; 4000 8])
mileage4K =
   19.2778
   19.2778
   14.3889
```

You can use a variety of other methods, such as `cutvar`, `cuttype`, and `cutcategories`, to get more information about the split at node 3 that distinguishes the 8-cylinder car:

```
var3 = cutvar(t,3) % What variable determines the split?
var3 =
    'Cylinders'

type3 = cuttype(t,3) % What type of split is it?
type3 =
    'categorical'

c = cutcategories(t,3) % Which classes are sent to the left
                       % child node, and which to the right?
c =
    [8]    [1x2 double]
c{1}
ans =
     8
c{2}
ans =
     4     6
```

Regression trees fit the original (training) data well, but may do a poor job of predicting new values. Lower branches, especially, may be strongly affected by outliers. A simpler tree often avoids over-fitting. To find the best regression tree, employing the techniques of resubstitution and cross-validation, use the `test` method.

Classification Trees

This example uses Fisher's iris data in `fisheriris.mat` to create a classification tree for predicting species using measurements of sepal length, sepal width, petal length, and petal width as predictors. Note that, in this case, the predictors are continuous and the response is categorical.

Load the data and use the `classregtree` constructor to create the classification tree:

```
load fisheriris

t = classregtree(meas,species,'names',{'SL' 'SW' 'PL' 'PW'})
t =
Decision tree for classification
1  if PL<2.45 then node 2 else node 3
2  class = setosa
3  if PW<1.75 then node 4 else node 5
4  if PL<4.95 then node 6 else node 7
5  class = virginica
6  if PW<1.65 then node 8 else node 9
7  class = virginica
8  class = versicolor
9  class = virginica
```

`t` is a `classregtree` object and can be operated on with any of the methods of the class.

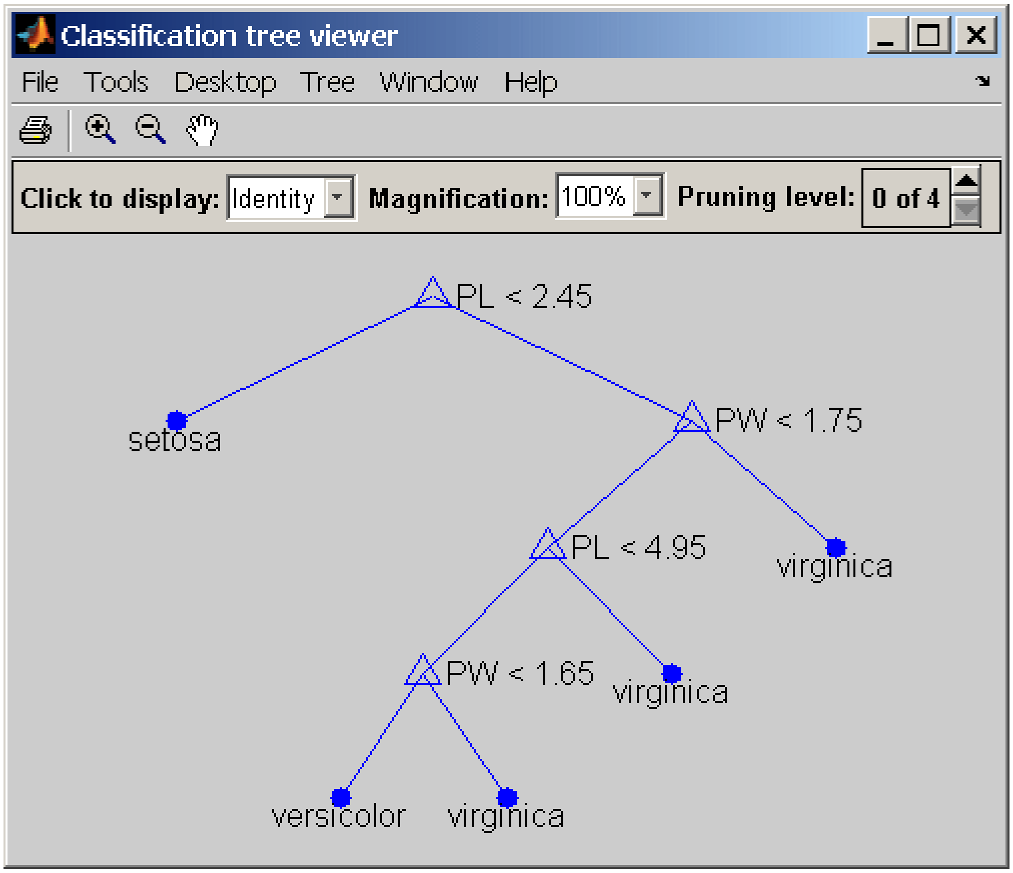
Use the `type` method to show the type of the tree:

```
treetype = type(t)
treetype =
classification
```

`classregtree` creates a classification tree because `species` is a cell array of strings, and the response is assumed to be categorical.

To view the tree, use the `view` method:

```
view(t)
```



The tree predicts the response values at the circular leaf nodes based on a series of questions about the iris at the triangular branching nodes. A `true` answer to any question follows the branch to the left; a `false` follows the branch to the right.

The tree does not use sepal measurements for predicting species. These can go unmeasured in new data, and be entered as `NaN` values for predictions. For example, to use the tree to predict the species of an iris with petal length `4.8` and petal width `1.6`, type

```
predicted = t([NaN NaN 4.8 1.6])
predicted =
    'versicolor'
```

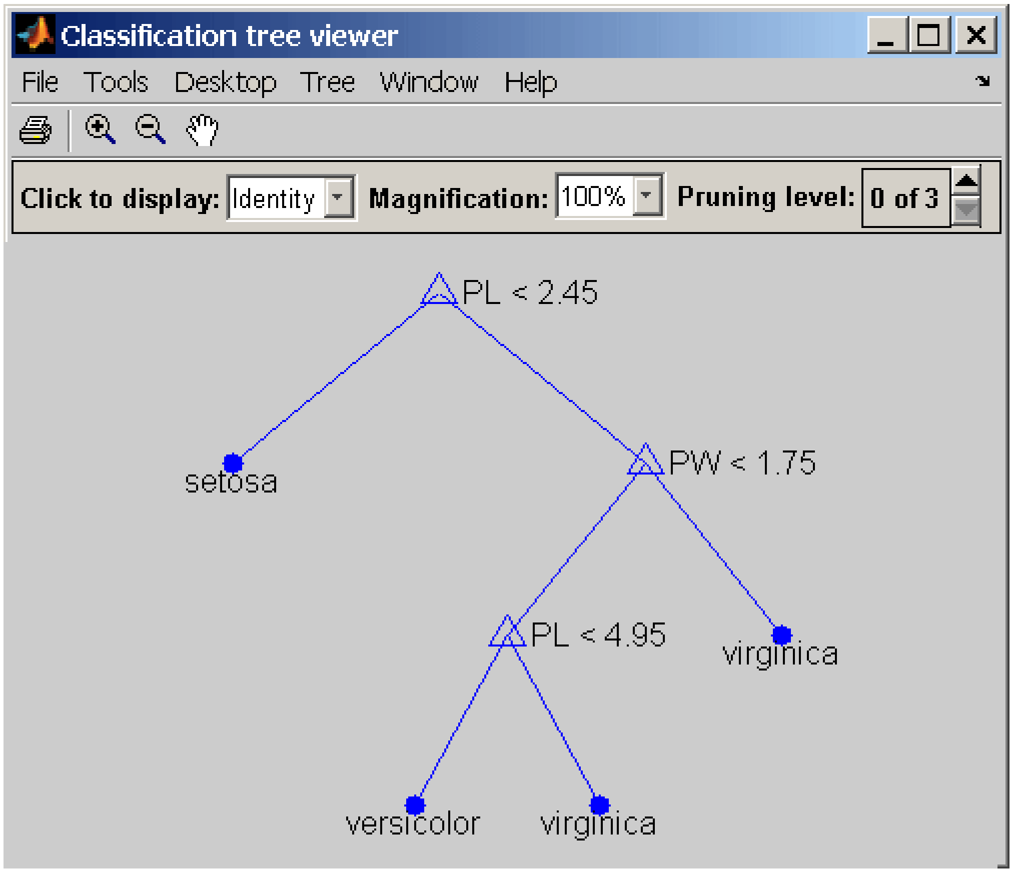
Note that the object allows for functional evaluation, of the form `t(X)`. This is a shorthand way of calling the `eval` method. The predicted species is the left-hand leaf node at the bottom of the tree in the view above.

You can use a variety of other methods, such as `cutvar` and `cuttype`, to get more information about the split at node 6 that makes the final distinction between `versicolor` and `virginica`:

```
var6 = cutvar(t,6) % What variable determines the split?
var6 =
    'PW'

type6 = cuttype(t,6) % What type of split is it?
type6 =
    'continuous'
```

Classification trees fit the original (training) data well, but may do a poor job of classifying new values. Lower branches, especially, may be strongly affected by outliers. A simpler tree often avoids over-fitting. The `prune` method can be used to find the next largest tree from an optimal pruning sequence:

```
pruned = prune(t,'level',1)
pruned =
Decision tree for classification
1  if PL<2.45 then node 2 else node 3
2  class = setosa
3  if PW<1.75 then node 4 else node 5
4  if PL<4.95 then node 6 else node 7
5  class = virginica
6  class = versicolor
7  class = virginica

view(pruned)
```



To find the best classification tree, employing the techniques of resubstitution and cross-validation, use the `test` method.

Multivariate Statistics

Principal Components Analysis

One of the difficulties inherent in multivariate statistics is the problem of visualizing data that has many variables. In MATLAB, the `plot` command displays a graph of the relationship between two variables. The `plot3` and `surf` commands display different three-dimensional views. But when there are more than three variables, it is more difficult to visualize their relationships.

Fortunately, in data sets with many variables, groups of variables often move together. One reason for this is that more than one variable might be measuring the same driving principle governing the behavior of the system. In many systems there are only a few such driving forces. But an abundance of instrumentation enables you to measure dozens of system variables. When this happens, you can take advantage of this redundancy of information. You can simplify the problem by replacing a group of variables with a single new variable.

Principal components analysis is a quantitatively rigorous method for achieving this simplification. The method generates a new set of variables, called *principal components*. Each principal component is a linear combination of the original variables. All the principal components are orthogonal to each other, so there is no redundant information. The principal components as a whole form an orthogonal basis for the space of the data.

There are an infinite number of ways to construct an orthogonal basis for several columns of data. What is so special about the principal component basis?

The first principal component is a single axis in space. When you project each observation on that axis, the resulting values form a new variable. And the variance of this variable is the maximum among all possible choices of the first axis.

The second principal component is another axis in space, perpendicular to the first. Projecting the observations on this axis generates another new variable. The variance of this variable is the maximum among all possible choices of this second axis.

The full set of principal components is as large as the original set of variables. But it is commonplace for the sum of the variances of the first few principal components to exceed 80% of the total variance of the original data. By examining plots of these few new variables, researchers often develop a deeper understanding of the driving forces that generated the original data.

You can use the function `princomp` to find the principal components. The following sections provide an example and explain the four outputs of `princomp`:

- "Example: Principal Components Analysis" on page 9-3
- "The Principal Component Coefficients (First Output)" on page 9-6
- "The Component Scores (Second Output)" on page 9-7
- "The Component Variances (Third Output)" on page 9-10
- "Hotelling's T2 (Fourth Output)" on page 9-12
- "Visualizing the Results of a Principal Components Analysis — The Biplot" on page 9-12

To use `princomp`, you need to have the actual measured data you want to analyze. However, if you lack the actual data, but have the sample covariance or correlation matrix for the data, you can still use the function `pcacov` to perform a principal components analysis. See the reference page for `pcacov` for a description of its inputs and outputs.

Example: Principal Components Analysis

Consider a sample application that uses nine different indices of the quality of life in 329 U.S. cities. These are climate, housing, health, crime, transportation, education, arts, recreation, and economics. For each index, higher is better. For example, a higher index for crime means a lower crime rate.

Start by loading the data in `cities.mat`.

```
load cities
whos
  Name              Size          Bytes  Class
  categories        9x14            252  char array
  names          329x43         28294  char array
  ratings        329x9          23688  double array
```

The whos command generates a table of information about all the variables in the workspace.

The cities data set contains three variables:

- categories, a string matrix containing the names of the indices
- names, a string matrix containing the 329 city names
- ratings, the data matrix with 329 rows and 9 columns

The categories variable has the following values:

```
categories
categories =
   climate
   housing
   health
   crime
   transportation
   education
   arts
   recreation
   economics
```

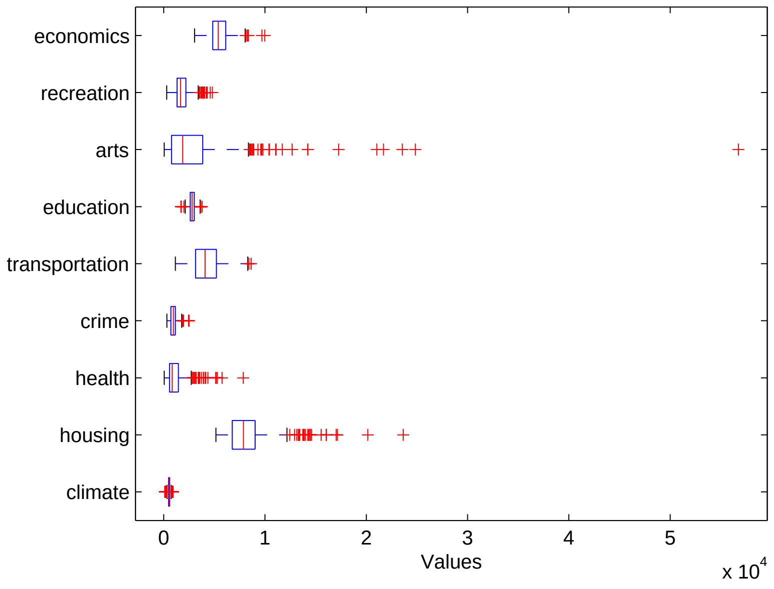
The first five rows of names are

```
first5 = names(1:5,:)
first5 =
   Abilene, TX
   Akron, OH
   Albany, GA
   Albany-Troy, NY
   Albuquerque, NM
```

To get a quick impression of the ratings data, make a box plot.

```
boxplot(ratings,'orientation','horizontal','labels',categories)
```

This command generates the plot below. Note that there is substantially more variability in the ratings of the arts and housing than in the ratings of crime and climate.



Ordinarily you might also graph pairs of the original variables, but there are 36 two-variable plots. Perhaps principal components analysis can reduce the number of variables you need to consider.

Sometimes it makes sense to compute principal components for raw data. This is appropriate when all the variables are in the same units. Standardizing the data is often preferable when the variables are in different units or when the variance of the different columns is substantial (as in this case).

You can standardize the data by dividing each column by its standard deviation.

```
stdr = std(ratings);
sr = ratings./repmat(stdr,329,1);
```

Now you are ready to find the principal components.

```
[coefs,scores,variances,t2] = princomp(sr);
```

The following sections explain the four outputs from `princomp`.

The Principal Component Coefficients (First Output)

The first output of the `princomp` function, `coefs`, contains the coefficients for nine principal components. These are the linear combinations of the original variables that generate the new variables.

The first three principal component coefficient vectors are

```
c3 = coefs(:,1:3)
c3 =
    0.2064    0.2178   -0.6900
    0.3565    0.2506   -0.2082
    0.4602   -0.2995   -0.0073
    0.2813    0.3553    0.1851
    0.3512   -0.1796    0.1464
    0.2753   -0.4834    0.2297
    0.4631   -0.1948   -0.0265
    0.3279    0.3845   -0.0509
    0.1354    0.4713    0.6073
```

The largest coefficients in the first column (first principal component) are the third and seventh elements, corresponding to the variables `health` and `arts`. All the coefficients of the first principal component have the same sign, making it a weighted average of all the original variables.

Because the principal components are unit length and orthogonal, premultiplying the matrix `c3` by its transpose yields the identity matrix.

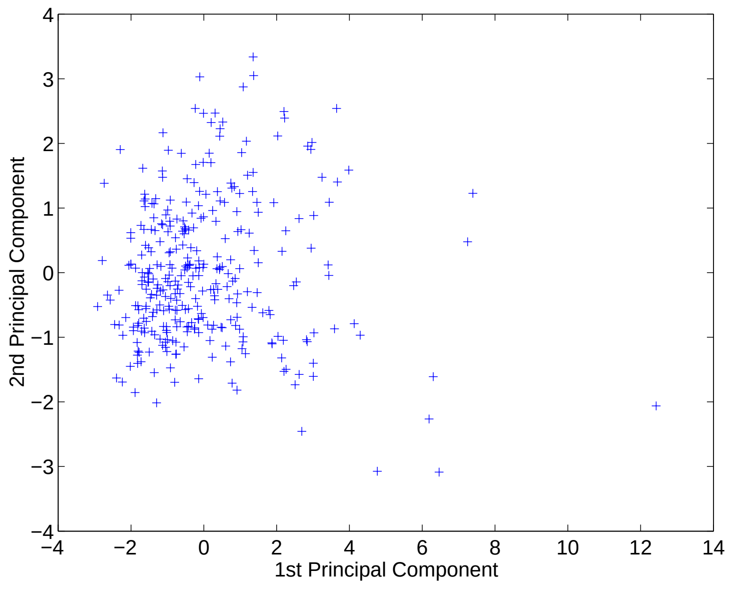
```
I = c3'*c3
I =
    1.0000   -0.0000   -0.0000
   -0.0000    1.0000   -0.0000
   -0.0000   -0.0000    1.0000
```

The Component Scores (Second Output)

The second output, `scores`, is the original data mapped into the new coordinate system defined by the principal components. This output is the same size as the input data matrix.

A plot of the first two columns of `scores` shows the ratings data projected onto the first two principal components. `princomp` computes the scores to have mean zero.

```
plot(scores(:,1),scores(:,2),'+')
xlabel('1st Principal Component');
ylabel('2nd Principal Component');
```



Note the outlying points in the right half of the plot.

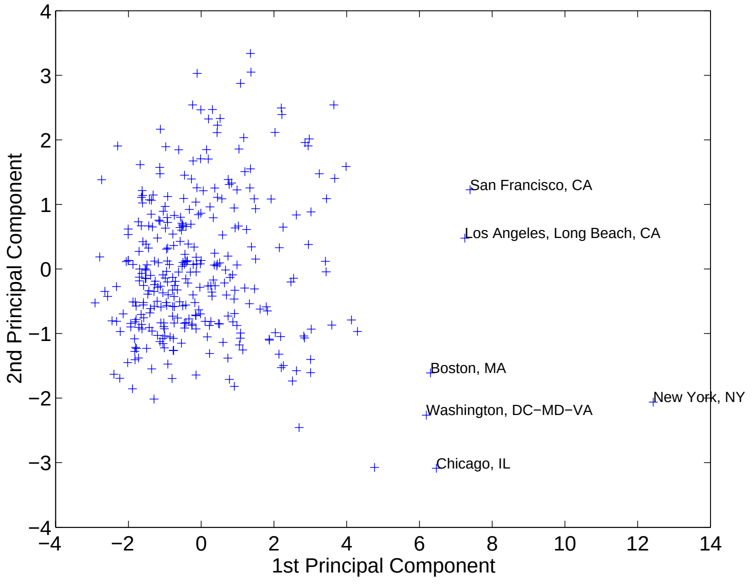
While it is possible to create a three-dimensional plot using three columns of `scores`, the examples in this section create two-dimensional plots, which are easier to describe.

The function gname is useful for graphically identifying a few points in a plot like this. You can call gname with a string matrix containing as many case labels as points in the plot. The string matrix names works for labeling points with the city names.

```
gname(names)
```

Move your cursor over the plot and click once near each point in the right half. As you click each point, MATLAB labels it with the proper row from the names string matrix. When you are finished labeling points, press the **Return** key.

Here is the resulting plot.



The labeled cities are some of the biggest population centers in the United States. They are definitely different from the remainder of the data, so perhaps they should be considered separately. To remove the labeled cities from the data, first identify their corresponding row numbers as follows:

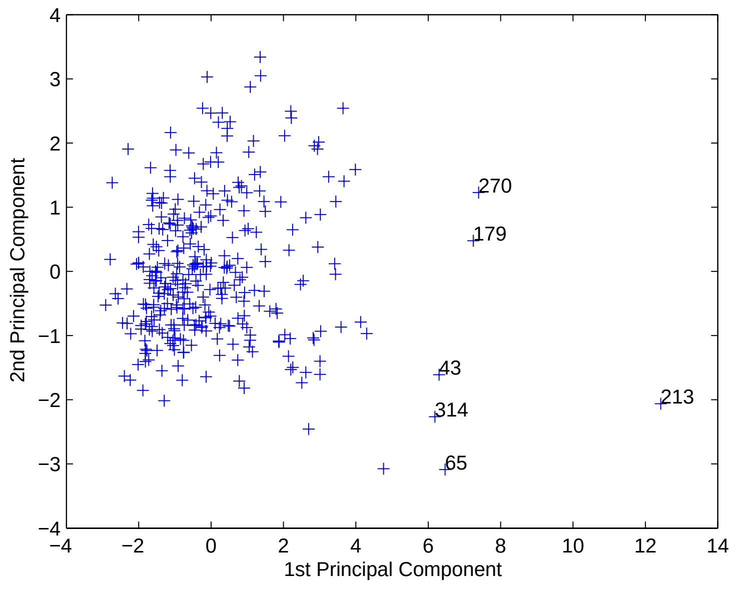
1 Close the plot window.

2 Redraw the plot by entering

```
plot(scores(:,1),scores(:,2),'+')
xlabel('1st Principal Component');
ylabel('2nd Principal Component');
```

3 Enter `gname` without any arguments.

4 Click near the points you labeled in the preceding figure. This labels the points by their row numbers, as shown in the following figure.



Then you can create an index variable containing the row numbers of all the metropolitan areas you choose.

```
metro = [43 65 179 213 234 270 314];
names(metro,:)
ans =
   Boston, MA
   Chicago, IL
   Los Angeles, Long Beach, CA
   New York, NY
   Philadelphia, PA-NJ
   San Francisco, CA
   Washington, DC-MD-VA
```

To remove these rows from the ratings matrix, enter the following.

```
rsubset = ratings;
nsubset = names;
nsubset(metro,:) = [];
rsubset(metro,:) = [];
size(rsubset)
ans =
   322     9
```

The Component Variances (Third Output)

The third output, variances, is a vector containing the variance explained by the corresponding principal component. Each column of `scores` has a sample variance equal to the corresponding element of `variances`.

```
variances
variances =
    3.4083
    1.2140
    1.1415
    0.9209
    0.7533
    0.6306
    0.4930
    0.3180
    0.1204
```

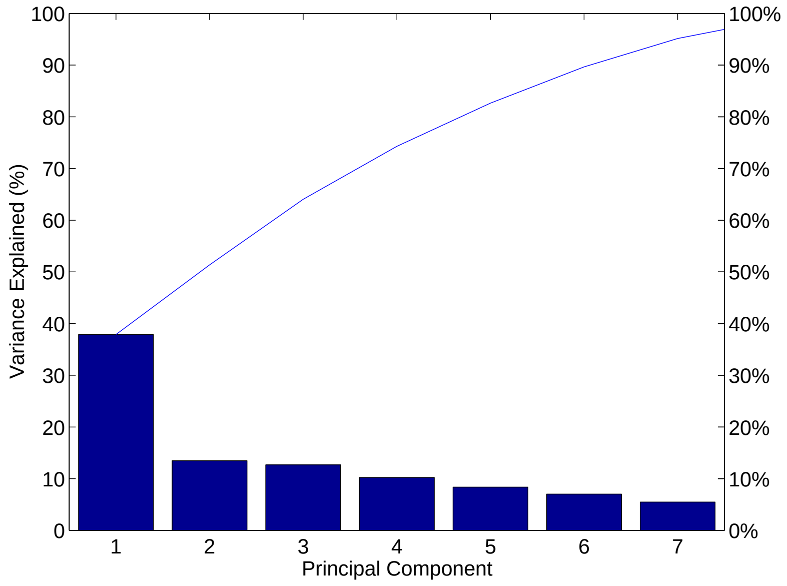
You can easily calculate the percent of the total variability explained by each principal component.

```
percent_explained = 100*variances/sum(variances)
percent_explained =
   37.8699
   13.4886
   12.6831
   10.2324
```

```
    8.3698
    7.0062
    5.4783
    3.5338
    1.3378
```

Use the `pareto` function to make a *scree plot* of the percent variability explained by each principal component.

```
pareto(percent_explained)
xlabel('Principal Component')
ylabel('Variance Explained (%)')
```



The preceding figure shows that the only clear break in the amount of variance accounted for by each component is between the first and second components. However, that component by itself explains less than 40% of the variance, so more components are probably needed. You can see that the first three principal components explain roughly two-thirds of the total variability in the standardized ratings, so that might be a reasonable way to reduce the dimensions in order to visualize the data.

Hotelling's T2 (Fourth Output)

The last output of the `princomp` function, `t2`, is Hotelling's T^2, a statistical measure of the multivariate distance of each observation from the center of the data set. This is an analytical way to find the most extreme points in the data.

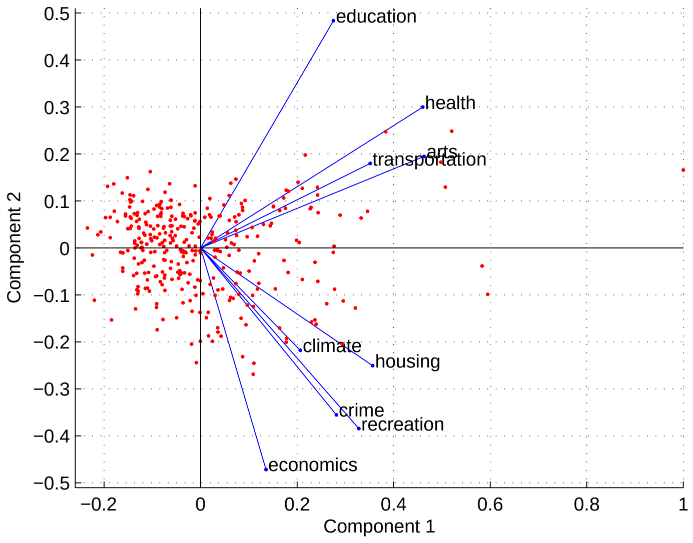
```
[st2, index] = sort(t2,'descend'); % Sort in descending order.
extreme = index(1)
extreme =
   213
names(extreme,:)
ans =
   New York, NY
```

It is not surprising that the ratings for New York are the furthest from the average U.S. town.

Visualizing the Results of a Principal Components Analysis — The Biplot

You can use the `biplot` function to help visualize both the principal component coefficients for each variable and the principal component scores for each observation in a single plot. For example, the following command plots the results from the principal components analysis on the cities and labels each of the variables.

```
biplot(coefs(:,1:2), 'scores',scores(:,1:2),...
'varlabels',categories);
axis([-.26 1 -.51 .51]);
```



Each of the nine variables is represented in this plot by a vector, and the direction and length of the vector indicates how each variable contributes to the two principal components in the plot. For example, you have seen that the first principal component, represented in this biplot by the horizontal axis, has positive coefficients for all nine variables. That corresponds to the nine vectors directed into the right half of the plot. You have also seen that the second principal component, represented by the vertical axis, has positive coefficients for the variables education, health, arts, and education, and negative coefficients for the remaining five variables. That corresponds to vectors directed into the top and bottom halves of the plot, respectively. This indicates that this component distinguishes between cities that have high values for the first set of variables and low for the second, and cities that have the opposite.

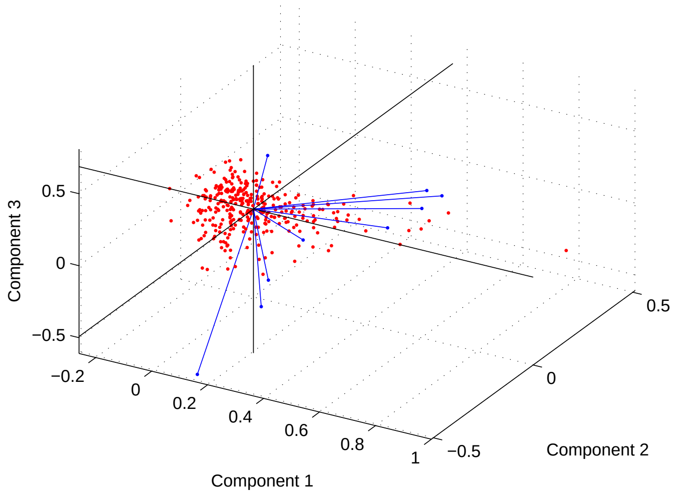
The variable labels in this figure are somewhat crowded. You could either leave out the `VarLabels` parameter when making the plot, or simply select and drag some of the labels to better positions using the Edit Plot tool from the figure window toolbar.

Each of the 329 observations is represented in this plot by a point, and their locations indicate the score of each observation for the two principal components in the plot. For example, points near the left edge of this plot have the lowest scores for the first principal component. The points are scaled to fit within the unit square, so only their relative locations may be determined from the plot.

You can use the `Data Cursor`, in the **Tools** menu in the figure window, to identify the items in this plot. By clicking on a variable (vector), you can read off that variable's coefficients for each principal component. By clicking on an observation (point), you can read off that observation's scores for each principal component.

You can also make a biplot in three dimensions. This can be useful if the first two principal coordinates do not explain enough of the variance in your data. Selecting `Rotate 3D` in the **Tools** menu enables you to rotate the figure to see it from different angles.

```
biplot(coefs(:,1:3), 'scores',scores(:,1:3),...
'obslabels',names);
axis([-.26 1 -.51 .51 -.61 .81]);
view([30 40]);
```



Factor Analysis

Multivariate data often includes a large number of measured variables, and sometimes those variables overlap, in the sense that groups of them might be dependent. For example, in a decathlon, each athlete competes in 10 events, but several of them can be thought of as speed events, while others can be thought of as strength events, etc. Thus, you can think of a competitor's 10 event scores as largely dependent on a smaller set of three or four types of athletic ability.

Factor analysis is a way to fit a model to multivariate data to estimate just this sort of interdependence. In a factor analysis model, the measured variables depend on a smaller number of unobserved (latent) factors. Because each factor might affect several variables in common, they are known as *common factors*. Each variable is assumed to be dependent on a linear combination of the common factors, and the coefficients are known as loadings. Each measured variable also includes a component due to independent random variability, known as "specific variance" because it is specific to one variable.

Specifically, factor analysis assumes that the covariance matrix of your data is of the form

$$\Sigma_x = \Lambda\Lambda^T + \Psi$$

where Λ is the matrix of loadings, and the elements of the diagonal matrix Ψ are the specific variances. The function `factoran` fits the Factor Analysis model using maximum likelihood.

This section includes these topics:

- "Example: Finding Common Factors Affecting Stock Prices" on page 9-16
- "Factor Rotation" on page 9-18
- "Predicting Factor Scores" on page 9-19
- "Visualizing the Results of a Factor Analysis — The Biplot" on page 9-21
- "Comparison of Factor Analysis and Principal Components Analysis" on page 9-22

Example: Finding Common Factors Affecting Stock Prices

Over the course of 100 weeks, the percent change in stock prices for ten companies has been recorded. Of the ten companies, the first four can be classified as primarily technology, the next three as financial, and the last three as retail. It seems reasonable that the stock prices for companies that are in the same sector might vary together as economic conditions change. Factor Analysis can provide quantitative evidence that companies within each sector do experience similar week-to-week changes in stock price.

In this example, you first load the data, and then call `factoran`, specifying a model fit with three common factors. By default, `factoran` computes rotated estimates of the loadings to try and make their interpretation simpler. But in this example, you specify an unrotated solution.

```
load stockreturns
[Loadings,specificVar,T,stats] = ...
  factoran(stocks,3,'rotate','none');
```

The first two `factoran` return arguments are the estimated loadings and the estimated specific variances. Each row of the loadings matrix represents one of the ten stocks, and each column corresponds to a common factor. With unrotated estimates, interpretation of the factors in this fit is difficult because most of the stocks contain fairly large coefficients for two or more factors.

```
Loadings
Loadings =
       0.8885    0.2367   -0.2354
       0.7126    0.3862    0.0034
       0.3351    0.2784   -0.0211
       0.3088    0.1113   -0.1905
       0.6277   -0.6643    0.1478
       0.4726   -0.6383    0.0133
       0.1133   -0.5416    0.0322
       0.6403    0.1669    0.4960
       0.2363    0.5293    0.5770
       0.1105    0.1680    0.5524
```

Note “Factor Rotation” on page 9-18 helps to simplify the structure in the `Loadings` matrix, to make it easier to assign meaningful interpretations to the factors.

From the estimated specific variances, you can see that the model indicates that a particular stock price varies quite a lot beyond the variation due to the common factors.

```
specificVar
specificVar =
       0.0991
       0.3431
       0.8097
       0.8559
       0.1429
       0.3691
       0.6928
       0.3162
       0.3311
       0.6544
```

A specific variance of 1 would indicate that there is *no* common factor component in that variable, while a specific variance of 0 would indicate that the variable is *entirely* determined by common factors. These data seem to fall somewhere in between.

The p-value returned in the `stats` structure fails to reject the null hypothesis of three common factors, suggesting that this model provides a satisfactory explanation of the covariation in these data.

```
stats.p
ans =
       0.8144
```

To determine whether fewer than three factors can provide an acceptable fit, you can try a model with two common factors. The p-value for this second fit is highly significant, and rejects the hypothesis of two factors, indicating that the simpler model is not sufficient to explain the pattern in these data.

```
[Loadings2,specificVar2,T2,stats2] = ...
  factoran(stocks, 2,'rotate','none');
stats2.p
ans =
      3.5610e-006
```

Factor Rotation

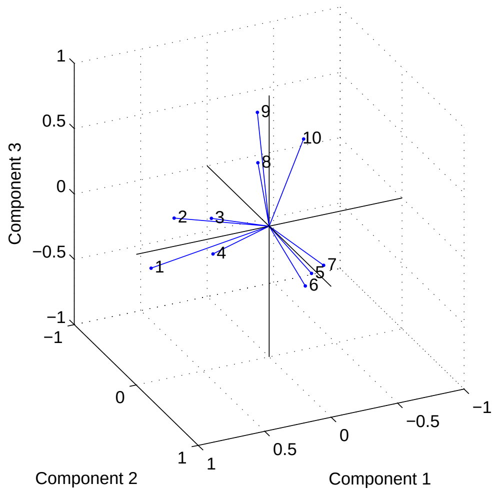
As the results in "Example: Finding Common Factors Affecting Stock Prices" on page 9-16 illustrate, the estimated loadings from an unrotated factor analysis fit can have a complicated structure. The goal of factor rotation is to find a parameterization in which each variable has only a small number of large loadings. That is, each variable is affected by a small number of factors, preferably only one. This can often make it easier to interpret what the factors represent.

If you think of each row of the loadings matrix as coordinates of a point in *M*-dimensional space, then each factor corresponds to a coordinate axis. Factor rotation is equivalent to rotating those axes and computing new loadings in the rotated coordinate system. There are various ways to do this. Some methods leave the axes orthogonal, while others are oblique methods that change the angles between them. For this example, you can rotate the estimated loadings by using the promax criterion, a common oblique method.

```
[LoadingsPM,specVarPM] = factoran(stocks,3,'rotate','promax');
LoadingsPM
LoadingsPM =
       0.9452    0.1214   -0.0617
       0.7064   -0.0178    0.2058
       0.3885   -0.0994    0.0975
       0.4162   -0.0148   -0.1298
       0.1021    0.9019    0.0768
       0.0873    0.7709   -0.0821
      -0.1616    0.5320   -0.0888
       0.2169    0.2844    0.6635
       0.0016   -0.1881    0.7849
      -0.2289    0.0636    0.6475
```

Promax rotation creates a simpler structure in the loadings, one in which most of the stocks have a large loading on only one factor. To see this structure more clearly, you can use the `biplot` function to plot each stock using its factor loadings as coordinates.

```
biplot(LoadingsPM,'varlabels',num2str((1:10)'));
axis square
view(155,27);
```



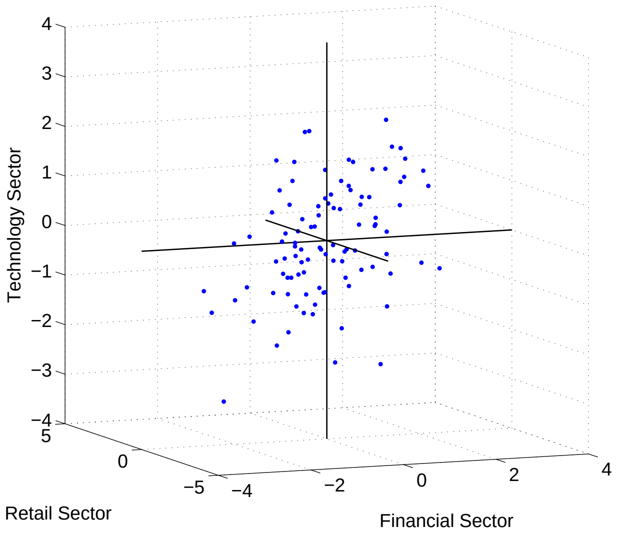
This plot shows that promax has rotated the factor loadings to a simpler structure. Each stock depends primarily on only one factor, and it is possible to describe each factor in terms of the stocks that it affects. Based on which companies are near which axes, you could reasonably conclude that the first factor axis represents the financial sector, the second retail, and the third technology. The original conjecture, that stocks vary primarily within sector, is apparently supported by the data.

Predicting Factor Scores

Sometimes, it is useful to be able to classify an observation based on its factor scores. For example, if you accepted the three-factor model and the interpretation of the rotated factors, you might want to categorize each week in terms of how favorable it was for each of the three stock sectors, based on the data from the 10 observed stocks.

Because the data in this example are the raw stock price changes, and not just their correlation matrix, you can have `factoran` return estimates of the value of each of the three rotated common factors for each week. You can then plot the estimated scores to see how the different stock sectors were affected during each week.

```
[LoadingsPM,specVarPM,TPM,stats,F] = ...
  factoran(stocks, 3,'rotate','promax');
subplot(1,1,1);
plot3(F(:,1),F(:,2),F(:,3),'b.');
line([-4 4 NaN 0 0 NaN 0 0], [0 0 NaN -4 4 NaN 0 0],...
     [0 0 NaN 0 0 NaN -4 4], 'Color','black');
xlabel('Financial Sector');
ylabel('Retail Sector');
zlabel('Technology Sector');
grid on;
axis square;
view(-22.5, 8);
```



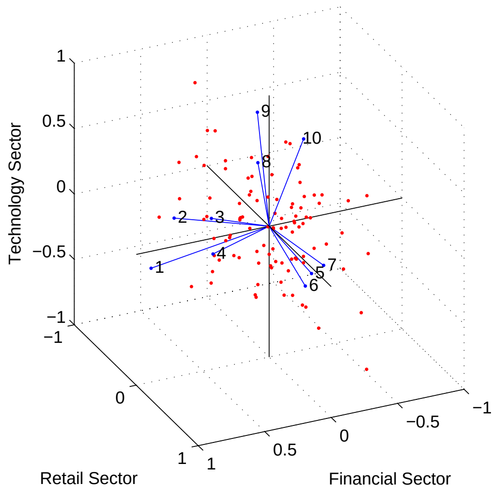
Oblique rotation often creates factors that are correlated. This plot shows some evidence of correlation between the first and third factors, and you can investigate further by computing the estimated factor correlation matrix.

```
inv(TPM'*TPM)
ans =
       1.0000    0.1559    0.4082
       0.1559    1.0000   -0.0559
       0.4082   -0.0559    1.0000
```

Visualizing the Results of a Factor Analysis — The Biplot

You can use the `biplot` function to help visualize both the factor loadings for each variable and the factor scores for each observation in a single plot. For example, the following command plots the results from the factor analysis on the stock data and labels each of the 10 stocks.

```
biplot(LoadingsPM, 'scores',F, 'varlabels',num2str((1:10)'));
xlabel('Financial Sector'); ylabel('Retail Sector');
zlabel('Technology Sector');
axis square
view(155,27);
```



In this case, the factor analysis includes three factors, and so the biplot is three-dimensional. Each of the 10 stocks is represented in this plot by a vector, and the direction and length of the vector indicates how each stock depends on the underlying factors. For example, you have seen that after promax rotation, the first four stocks have positive loadings on the first factor, and unimportant loadings on the other two factors. That first factor, interpreted as a financial sector effect, is represented in this biplot as one of the horizontal axes. The dependence of those four stocks on that factor corresponds to the four vectors directed approximately along that axis. Similarly, the dependence of stocks 5, 6, and 7 primarily on the second factor, interpreted as a retail sector effect, is represented by vectors directed approximately along that axis.

Each of the 100 observations is represented in this plot by a point, and their locations indicate the score of each observation for the three factors. For example, points near the top of this plot have the highest scores for the technology sector factor. The points are scaled to fit within the unit square, so only their relative locations can be determined from the plot.

You can use the **Data Cursor** tool from the **Tools** menu in the figure window to identify the items in this plot. By clicking a stock (vector), you can read off that stock's loadings for each factor. By clicking an observation (point), you can read off that observation's scores for each factor.

Comparison of Factor Analysis and Principal Components Analysis

There is a good deal of overlap in terminology and goals between principal components analysis (PCA) and factor analysis (FA). Much of the literature on the two methods does not distinguish between them, and some algorithms for fitting the FA model involve PCA. Both are dimension-reduction techniques, in the sense that they can be used to replace a large set of observed variables with a smaller set of new variables. However, the two methods are different in their goals and in their underlying models. Roughly speaking, you should use PCA when you simply need to summarize or approximate your data using fewer dimensions (to visualize it, for example), and you should use FA when you need an explanatory model for the correlations among your data.

Multivariate Analysis of Variance

The analysis of variance technique in “Example: One-Way ANOVA” on page 7-33 takes a set of grouped data and determine whether the mean of a variable differs significantly between groups. Often there are multiple variables, and you are interested in determining whether the entire set of means is different from one group to the next. There is a multivariate version of analysis of variance that can address that problem, as illustrated in the “Example: Multivariate Analysis of Variance” on page 9-23.

Example: Multivariate Analysis of Variance

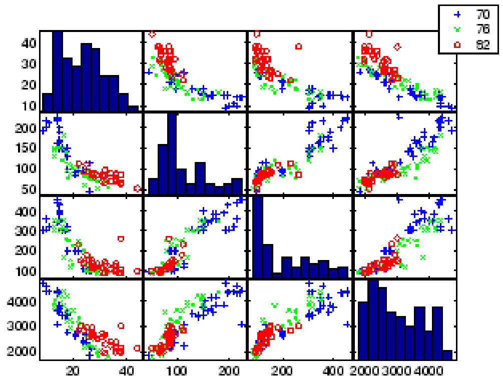
The `carsmall` data set has measurements on a variety of car models from the years 1970, 1976, and 1982. Suppose you are interested in whether the characteristics of the cars have changed over time.

First, load the data.

```
load carsmall
whos
  Name               Size           Bytes  Class
  Acceleration     100x1              800  double array
  Cylinders        100x1              800  double array
  Displacement     100x1              800  double array
  Horsepower       100x1              800  double array
  MPG              100x1              800  double array
  Model            100x36            7200  char array
  Model_Year       100x1              800  double array
  Origin           100x7             1400  char array
  Weight           100x1              800  double array
```

Four of these variables (`Acceleration`, `Displacement`, `Horsepower`, and `MPG`) are continuous measurements on individual car models. The variable `Model_Year` indicates the year in which the car was made. You can create a grouped plot matrix of these variables using the `gplotmatrix` function.

```
x = [MPG Horsepower Displacement Weight];
gplotmatrix(x,[],Model_Year,[],'+xo')
```



(When the second argument of `gplotmatrix` is empty, the function graphs the columns of the `x` argument against each other, and places histograms along the diagonals. The empty fourth argument produces a graph with the default colors. The fifth argument controls the symbols used to distinguish between groups.)

It appears the cars do differ from year to year. The upper right plot, for example, is a graph of `MPG` versus `Weight`. The 1982 cars appear to have higher mileage than the older cars, and they appear to weigh less on average. But as a group, are the three years significantly different from one another? The `manova1` function can answer that question.

```
[d,p,stats] = manova1(x,Model_Year)
d =
     2
p =
   1.0e-006 *
          0
     0.1141
stats =
              W: [4x4 double]
              B: [4x4 double]
              T: [4x4 double]
            dfW: 90
            dfB: 2
            dfT: 92
         lambda: [2x1 double]
```

```
        chisq: [2x1 double]
      chisqdf: [2x1 double]
     eigenval: [4x1 double]
     eigenvec: [4x4 double]
        canon: [100x4 double]
        mdist: [100x1 double]
       gmdist: [3x3 double]
```

The `manova1` function produces three outputs:

- The first output, `d`, is an estimate of the dimension of the group means. If the means were all the same, the dimension would be 0, indicating that the means are at the same point. If the means differed but fell along a line, the dimension would be 1. In the example the dimension is 2, indicating that the group means fall in a plane but not along a line. This is the largest possible dimension for the means of three groups.
- The second output, `p`, is a vector of p-values for a sequence of tests. The first p-value tests whether the dimension is 0, the next whether the dimension is 1, and so on. In this case both p-values are small. That's why the estimated dimension is 2.
- The third output, `stats`, is a structure containing several fields, described in the following section.

The Fields of the `stats` Structure

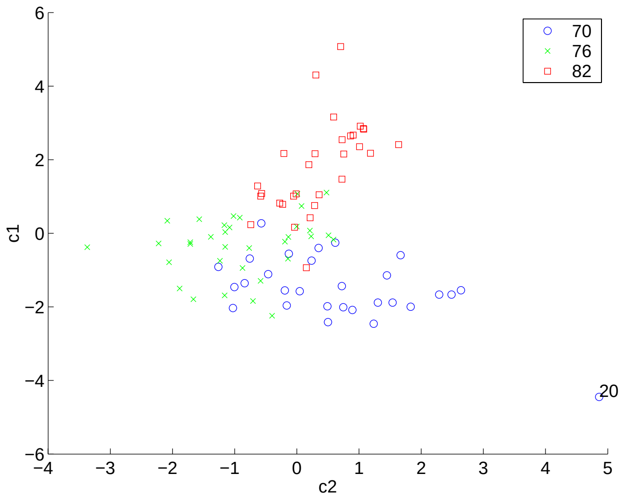
The `W`, `B`, and `T` fields are matrix analogs to the within, between, and total sums of squares in ordinary one-way analysis of variance. The next three fields are the degrees of freedom for these matrices. Fields `lambda`, `chisq`, and `chisqdf` are the ingredients of the test for the dimensionality of the group means. (The p-values for these tests are the first output argument of `manova1`.)

The next three fields are used to do a canonical analysis. Recall that in principal components analysis ("Principal Components Analysis" on page 9-2) you look for the combination of the original variables that has the largest possible variation. In multivariate analysis of variance, you instead look for the linear combination of the original variables that has the largest separation between groups. It is the single variable that would give the most significant result in a univariate one-way analysis of variance. Having found that combination, you next look for the combination with the second highest separation, and so on.

The `eigenvec` field is a matrix that defines the coefficients of the linear combinations of the original variables. The `eigenval` field is a vector measuring the ratio of the between-group variance to the within-group variance for the corresponding linear combination. The `canon` field is a matrix of the canonical variable values. Each column is a linear combination of the mean-centered original variables, using coefficients from the `eigenvec` matrix.

A grouped scatter plot of the first two canonical variables shows more separation between groups then a grouped scatter plot of any pair of original variables. In this example it shows three clouds of points, overlapping but with distinct centers. One point in the bottom right sits apart from the others. By using the `gname` function, you can see that this is the 20th point.

```
c1 = stats.canon(:,1);
c2 = stats.canon(:,2);
gscatter(c2,c1,Model_Year,[],'oxs')
gname
```



Roughly speaking, the first canonical variable, c1, separates the 1982 cars (which have high values of c1) from the older cars. The second canonical variable, c2, reveals some separation between the 1970 and 1976 cars.

The final two fields of the stats structure are Mahalanobis distances. The mdist field measures the distance from each point to its group mean. Points with large values may be outliers. In this data set, the largest outlier is the one in the scatter plot, the Buick Estate station wagon. (Note that you could have supplied the model name to the gname function above if you wanted to label the point with its model name rather than its row number.)

```
max(stats.mdist)
ans =
   31.5273
find(stats.mdist == ans)
ans =
   20
Model(20,:)
ans =
   buick_estate_wagon_(sw)
```

The gmdist field measures the distances between each pair of group means. The following commands examine the group means and their distances:

```
grpstats(x, Model_Year)
ans =
  1.0e+003 *
    0.0177    0.1489    0.2869    3.4413
    0.0216    0.1011    0.1978    3.0787
    0.0317    0.0815    0.1289    2.4535
stats.gmdist
ans =
         0    3.8277   11.1106
    3.8277         0    6.1374
   11.1106    6.1374         0
```

As might be expected, the multivariate distance between the extreme years 1970 and 1982 (11.1) is larger than the difference between more closely spaced years (3.8 and 6.1). This is consistent with the scatter plots, where the points seem to follow a progression as the year changes from 1970 through

1976 to 1982. If you had more groups, you might find it instructive to use the `manovacluster` function to draw a diagram that presents clusters of the groups, formed using the distances between their means.

Cluster Analysis

Cluster analysis, also called segmentation analysis or taxonomy analysis, is a way to create groups of objects, or *clusters*, in such a way that the profiles of objects in the same cluster are very similar and the profiles of objects in different clusters are quite distinct.

Cluster analysis can be performed on many different types of data sets. For example, a data set might contain a number of observations of subjects in a study where each observation contains a set of variables.

Many different fields of study, such as engineering, zoology, medicine, linguistics, anthropology, psychology, and marketing, have contributed to the development of clustering techniques and the application of such techniques. For example, cluster analysis can help in creating “balanced” treatment and control groups for a designed study. If you find that each cluster contains roughly equal numbers of treatment and control subjects, then statistical differences found between the groups can be attributed to the experiment and not to any initial difference between the groups.

This section explores two kinds of clustering:

- “Hierarchical Clustering” on page 9-29
- “K-Means Clustering” on page 9-46

Hierarchical Clustering

Hierarchical clustering is a way to investigate grouping in your data, simultaneously over a variety of scales, by creating a cluster tree. The tree is not a single set of clusters, but rather a multilevel hierarchy, where clusters at one level are joined as clusters at the next higher level. This allows you to decide what level or scale of clustering is most appropriate in your application.

The following sections explore the hierarchical clustering features in Statistics Toolbox:

- “Terminology and Basic Procedure” on page 9-30
- “Finding the Similarities Between Objects” on page 9-30
- “Defining the Links Between Objects” on page 9-33

- "Evaluating Cluster Formation" on page 9-35

Terminology and Basic Procedure

To perform hierarchical cluster analysis on a data set using Statistics Toolbox functions, follow this procedure:

1. **Find the similarity or dissimilarity between every pair of objects in the data set.** In this step, you calculate the *distance* between objects using the `pdist` function. The `pdist` function supports many different ways to compute this measurement. See "Finding the Similarities Between Objects" on page 9-30 for more information.

2. **Group the objects into a binary, hierarchical cluster tree.** In this step, you link pairs of objects that are in close proximity using the `linkage` function. The `linkage` function uses the distance information generated in step 1 to determine the proximity of objects to each other. As objects are paired into binary clusters, the newly formed clusters are grouped into larger clusters until a hierarchical tree is formed. See "Defining the Links Between Objects" on page 9-33 for more information.

3. **Determine where to cut the hierarchical tree into clusters.** In this step, you use the `cluster` function to prune branches off the bottom of the hierarchical tree, and assign all the objects below each cut to a single cluster. This creates a partition of the data. The `cluster` function can create these clusters by detecting natural groupings in the hierarchical tree or by cutting off the hierarchical tree at an arbitrary point. See [23] for more information.

The following sections provide more information about each of these steps.

Note Statistics Toolbox includes a convenience function, `clusterdata`, which performs all these steps for you. You do not need to execute the `pdist`, `linkage`, or `cluster` functions separately.

Finding the Similarities Between Objects

You use the `pdist` function to calculate the distance between every pair of objects in a data set. For a data set made up of *m* objects, there are

$m \cdot (m-1)/2$ pairs in the data set. The result of this computation is commonly known as a distance or dissimilarity matrix.

There are many ways to calculate this distance information. By default, the `pdist` function calculates the Euclidean distance between objects; however, you can specify one of several other options. See `pdist` for more information.

Note You can optionally normalize the values in the data set before calculating the distance information. In a real world data set, variables can be measured against different scales. For example, one variable can measure Intelligence Quotient (IQ) test scores and another variable can measure head circumference. These discrepancies can distort the proximity calculations. Using the `zscore` function, you can convert all the values in the data set to use the same proportional scale. See `zscore` for more information.

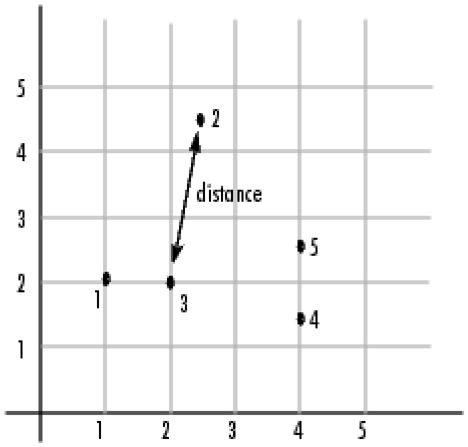
For example, consider a data set, `X`, made up of five objects where each object is a set of *x,y* coordinates.

- **Object 1**: 1, 2
- **Object 2**: 2.5, 4.5
- **Object 3**: 2, 2
- **Object 4**: 4, 1.5
- **Object 5**: 4, 2.5

You can define this data set as a matrix

```
X = [1 2;2.5 4.5;2 2;4 1.5;4 2.5]
```

and pass it to `pdist`. The `pdist` function calculates the distance between object 1 and object 2, object 1 and object 3, and so on until the distances between all the pairs have been calculated. The following figure plots these objects in a graph. The Euclidean distance between object 2 and object 3 is shown to illustrate one interpretation of distance.



Returning Distance Information. The pdist function returns this distance information in a vector, Y, where each element contains the distance between a pair of objects.

```
Y = pdist(X)
Y =
  Columns 1 through 5
    2.9155    1.0000    3.0414    3.0414    2.5495
  Columns 6 through 10
    3.3541    2.5000    2.0616    2.0616    1.0000
```

To make it easier to see the relationship between the distance information generated by pdist and the objects in the original data set, you can reformat the distance vector into a matrix using the squareform function. In this matrix, element *i,j* corresponds to the distance between object *i* and object *j* in the original data set. In the following example, element 1,1 represents the distance between object 1 and itself (which is zero). Element 1,2 represents the distance between object 1 and object 2, and so on.

```
squareform(Y)
ans =
         0    2.9155    1.0000    3.0414    3.0414
    2.9155         0    2.5495    3.3541    2.5000
    1.0000    2.5495         0    2.0616    2.0616
    3.0414    3.3541    2.0616         0    1.0000
```

```
        3.0414    2.5000    2.0616    1.0000         0
```

Defining the Links Between Objects

Once the proximity between objects in the data set has been computed, you can determine how objects in the data set should be grouped into clusters, using the `linkage` function. The `linkage` function takes the distance information generated by `pdist` and links pairs of objects that are close together into binary clusters (clusters made up of two objects). The `linkage` function then links these newly formed clusters to each other and to other objects to create bigger clusters until all the objects in the original data set are linked together in a hierarchical tree.

For example, given the distance vector `Y` generated by `pdist` from the sample data set of *x*- and *y*-coordinates, the `linkage` function generates a hierarchical cluster tree, returning the linkage information in a matrix, `Z`.

```
Z = linkage(Y)
Z =
    4.0000    5.0000    1.0000
    1.0000    3.0000    1.0000
    6.0000    7.0000    2.0616
    2.0000    8.0000    2.5000
```

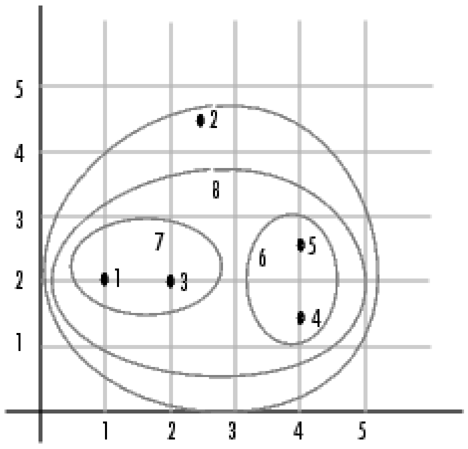
In this output, each row identifies a link between objects or clusters. The first two columns identify the objects that have been linked, that is, object 1, object 2, and so on. The third column contains the distance between these objects. For the sample data set of *x*- and *y*-coordinates, the `linkage` function begins by grouping objects 1 and 3, which have the closest proximity (distance value = 1.0000). The `linkage` function continues by grouping objects 4 and 5, which also have a distance value of 1.0000.

The third row indicates that the `linkage` function grouped objects 6 and 7. If the original sample data set contained only five objects, what are objects 6 and 7? Object 6 is the newly formed binary cluster created by the grouping of objects 1 and 3. When the `linkage` function groups two objects into a new cluster, it must assign the cluster a unique index value, starting with the value m+1, where m is the number of objects in the original data set.

(Values 1 through *m* are already used by the original data set.) Similarly, object 7 is the cluster formed by grouping objects 4 and 5.

`linkage` uses distances to determine the order in which it clusters objects. The distance vector `Y` contains the distances between the original objects 1 through 5. But linkage must also be able to determine distances involving clusters that it creates, such as objects 6 and 7. By default, `linkage` uses a method known as single linkage. However, there are a number of different methods available. See the `linkage` reference page for more information.

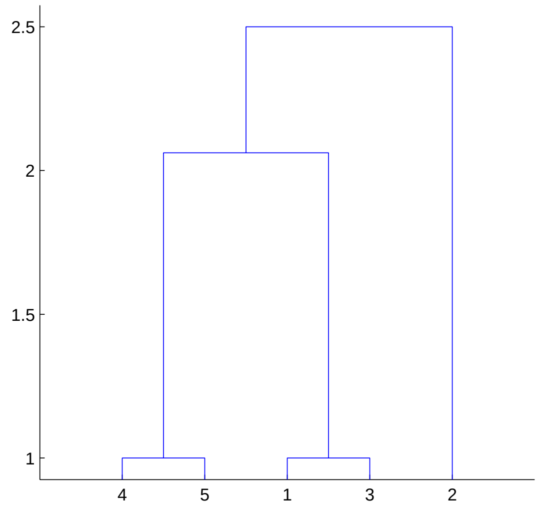
As the final cluster, the `linkage` function grouped object 8, the newly formed cluster made up of objects 6 and 7, with object 2 from the original data set. The following figure graphically illustrates the way `linkage` groups the objects into a hierarchy of clusters.



Plotting the Cluster Tree

The hierarchical, binary cluster tree created by the `linkage` function is most easily understood when viewed graphically. Statistics Toolbox includes the `dendrogram` function that plots this hierarchical tree information as a graph, as in the following example.

```
dendrogram(Z)
```



In the figure, the numbers along the horizontal axis represent the indices of the objects in the original data set. The links between objects are represented as upside-down U-shaped lines. The height of the U indicates the distance between the objects. For example, the link representing the cluster containing objects 1 and 3 has a height of 1. The link representing the cluster that groups object 2 together with objects 1, 3, 4, and 5, (which are already clustered as object 8) has a height of 2.5. The height represents the distance `linkage` computes between objects 2 and 8. For more information about creating a dendrogram diagram, see the `dendrogram` reference page.

Evaluating Cluster Formation

After linking the objects in a data set into a hierarchical cluster tree, you might want to verify that the distances (that is, heights) in the tree reflect the original distances accurately. In addition, you might want to investigate natural divisions that exist among links between objects. Statistics Toolbox provides functions to perform both these tasks, as described in the following sections:

- "Verifying the Cluster Tree" on page 9-36

- "Getting More Information About Cluster Links" on page 9-37

Verifying the Cluster Tree. In a hierarchical cluster tree, any two objects in the original data set are eventually linked together at some level. The height of the link represents the distance between the two clusters that contain those two objects. This height is known as the *cophenetic distance* between the two objects. One way to measure how well the cluster tree generated by the `linkage` function reflects your data is to compare the cophenetic distances with the original distance data generated by the `pdist` function. If the clustering is valid, the linking of objects in the cluster tree should have a strong correlation with the distances between objects in the distance vector. The `cophenet` function compares these two sets of values and computes their correlation, returning a value called the *cophenetic correlation coefficient*. The closer the value of the cophenetic correlation coefficient is to 1, the more accurately the clustering solution reflects your data.

You can use the cophenetic correlation coefficient to compare the results of clustering the same data set using different distance calculation methods or clustering algorithms. For example, you can use the `cophenet` function to evaluate the clusters created for the sample data set

```
c = cophenet(Z,Y)
c =
    0.8615
```

where `Z` is the matrix output by the `linkage` function and `Y` is the distance vector output by the `pdist` function.

Execute `pdist` again on the same data set, this time specifying the city block metric. After running the `linkage` function on this new `pdist` output using the average linkage method, call `cophenet` to evaluate the clustering solution.

```
Y = pdist(X,'cityblock');
Z = linkage(Y,'average');
c = cophenet(Z,Y)
c =
    0.9044
```

The cophenetic correlation coefficient shows that using a different distance and linkage method creates a tree that represents the original distances slightly better.

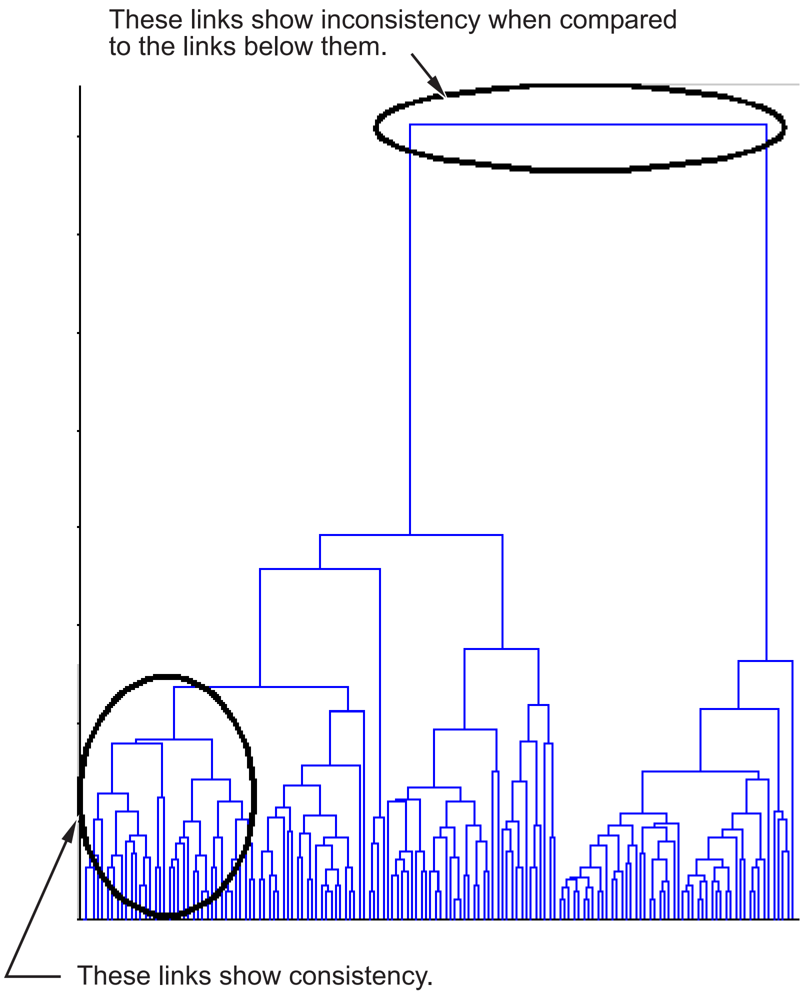
Getting More Information About Cluster Links. One way to determine the natural cluster divisions in a data set is to compare the height of each link in a cluster tree with the heights of neighboring links below it in the tree.

A link that is approximately the same height as the links below it indicates that there are no distinct divisions between the objects joined at this level of the hierarchy. These links are said to exhibit a high level of consistency, because the distance between the objects being joined is approximately the same as the distances between the objects they contain.

On the other hand, a link whose height differs noticeably from the height of the links below it indicates that the objects joined at this level in the cluster tree are much farther apart from each other than their components were when they were joined. This link is said to be inconsistent with the links below it.

In cluster analysis, inconsistent links can indicate the border of a natural division in a data set. The `cluster` function uses a quantitative measure of inconsistency to determine where to partition your data set into clusters. (See [23] for more information.)

The following dendrogram illustrates inconsistent links. Note how the objects in the dendrogram fall into two groups that are connected by links at a much higher level in the tree. These links are inconsistent when compared with the links below them in the hierarchy.



The relative consistency of each link in a hierarchical cluster tree can be quantified and expressed as the *inconsistency coefficient*. This value compares the height of a link in a cluster hierarchy with the average height of links below it. Links that join distinct clusters have a low inconsistency coefficient; links that join indistinct clusters have a high inconsistency coefficient.

To generate a listing of the inconsistency coefficient for each link in the cluster tree, use the `inconsistent` function. By default, the `inconsistent` function compares each link in the cluster hierarchy with adjacent links that are less than two levels below it in the cluster hierarchy. This is called the *depth* of the comparison. You can also specify other depths. The objects at the bottom of the cluster tree, called leaf nodes, that have no further objects below them, have an inconsistency coefficient of zero. Clusters that join two leaves also have a zero inconsistency coefficient.

For example, you can use the `inconsistent` function to calculate the inconsistency values for the links created by the `linkage` function in "Defining the Links Between Objects" on page 9-33.

```
I = inconsistent(Z)
I =
    1.0000         0    1.0000         0
    1.0000         0    1.0000         0
    1.3539    0.6129    3.0000    1.1547
    2.2808    0.3100    2.0000    0.7071
```

The `inconsistent` function returns data about the links in an (m-1)-by-4 matrix, whose columns are described in the following table.

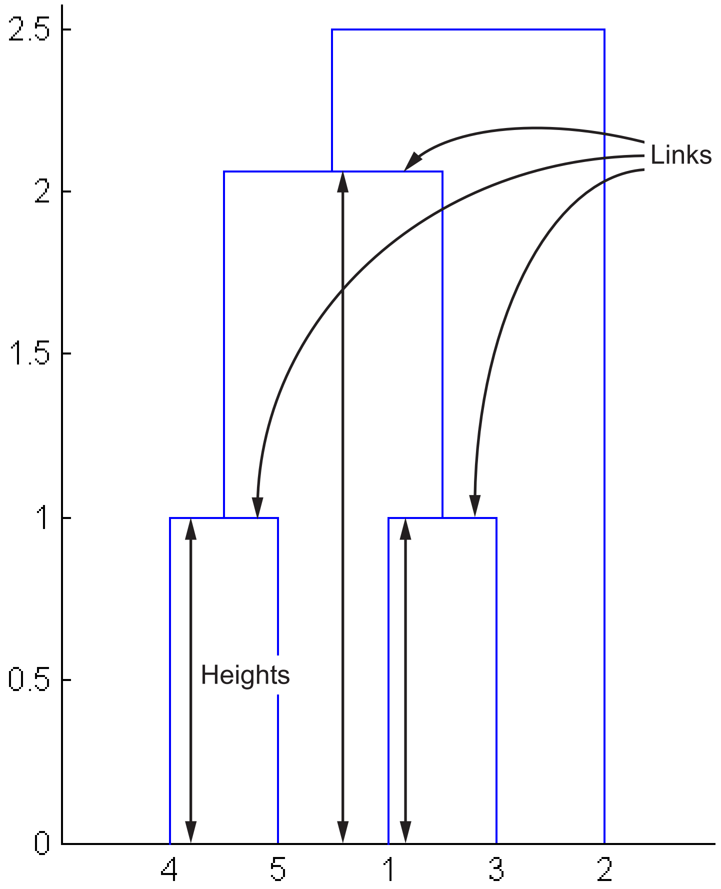
Column	Description
1	Mean of the heights of all the links included in the calculation
2	Standard deviation of all the links included in the calculation
3	Number of links included in the calculation
4	Inconsistency coefficient

In the sample output, the first row represents the link between objects 4 and 5. This cluster is assigned the index 6 by the `linkage` function. Because both 4 and 5 are leaf nodes, the inconsistency coefficient for the cluster is zero. The second row represents the link between objects 1 and 3, both of which are also leaf nodes. This cluster is assigned the index 7 by the linkage function.

The third row evaluates the link that connects these two clusters, objects 6 and 7. (This new cluster is assigned index 8 in the `linkage` output). Column 3 indicates that three links are considered in the calculation: the link itself and the two links directly below it in the hierarchy. Column 1 represents the mean of the heights of these links. The `inconsistent` function uses the height information output by the `linkage` function to calculate the mean. Column 2 represents the standard deviation between the links. The last column contains the inconsistency value for these links, 1.1547. It is the difference between the current link height and the mean, normalized by the standard deviation:

```
(2.0616 - 1.3539) / .6129
ans =
    1.1547
```

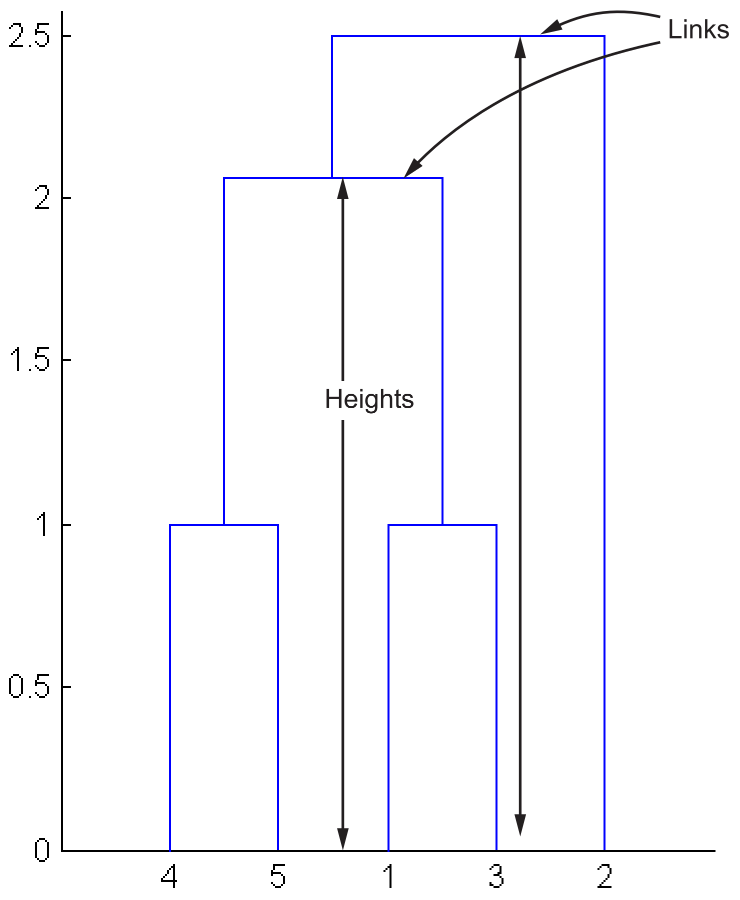
The following figure illustrates the links and heights included in this calculation.



Note In the preceding figure, the lower limit on the *y*-axis is set to `0` to show the heights of the links. To set the lower limit to `0`, select `Axes Properties` from the **Edit** menu, click the **Y Axis** tab, and enter `0` in the field immediately to the right of **Y Limits**.

Row 4 in the output matrix describes the link between object 8 and object 2. Column 3 indicates that two links are included in this calculation: the link itself and the link directly below it in the hierarchy. The inconsistency coefficient for this link is 0.7071.

The following figure illustrates the links and heights included in this calculation.



Creating Clusters

After you create the hierarchical tree of binary clusters, you can prune the tree to partition your data into clusters using the `cluster` function. The `cluster` function lets you create clusters in two ways, as discussed in the following sections:

- "Finding Natural Divisions in Data" on page 9-43
- "Specifying Arbitrary Clusters" on page 9-44

Finding Natural Divisions in Data. The hierarchical cluster tree may naturally divide the data into distinct, well-separated clusters. This can be particularly evident in a dendrogram diagram created from data where groups of objects are densely packed in certain areas and not in others. The inconsistency coefficient of the links in the cluster tree can identify these divisions where the similarities between objects change abruptly. (See "Evaluating Cluster Formation" on page 9-35 for more information about the inconsistency coefficient.) You can use this value to determine where the `cluster` function creates cluster boundaries.

For example, if you use the `cluster` function to group the sample data set into clusters, specifying an inconsistency coefficient threshold of `1.2` as the value of the `cutoff` argument, the `cluster` function groups all the objects in the sample data set into one cluster. In this case, none of the links in the cluster hierarchy had an inconsistency coefficient greater than `1.2`.

```
T = cluster(Z,'cutoff',1.2)
T =
      1
      1
      1
      1
      1
```

The `cluster` function outputs a vector, `T`, that is the same size as the original data set. Each element in this vector contains the number of the cluster into which the corresponding object from the original data set was placed.

If you lower the inconsistency coefficient threshold to `0.8`, the `cluster` function divides the sample data set into three separate clusters.

```
T = cluster(Z,'cutoff',0.8)
T =
     1
     3
     1
     2
     2
```

This output indicates that objects 1 and 3 were placed in cluster 1, objects 4 and 5 were placed in cluster 2, and object 2 was placed in cluster 3.

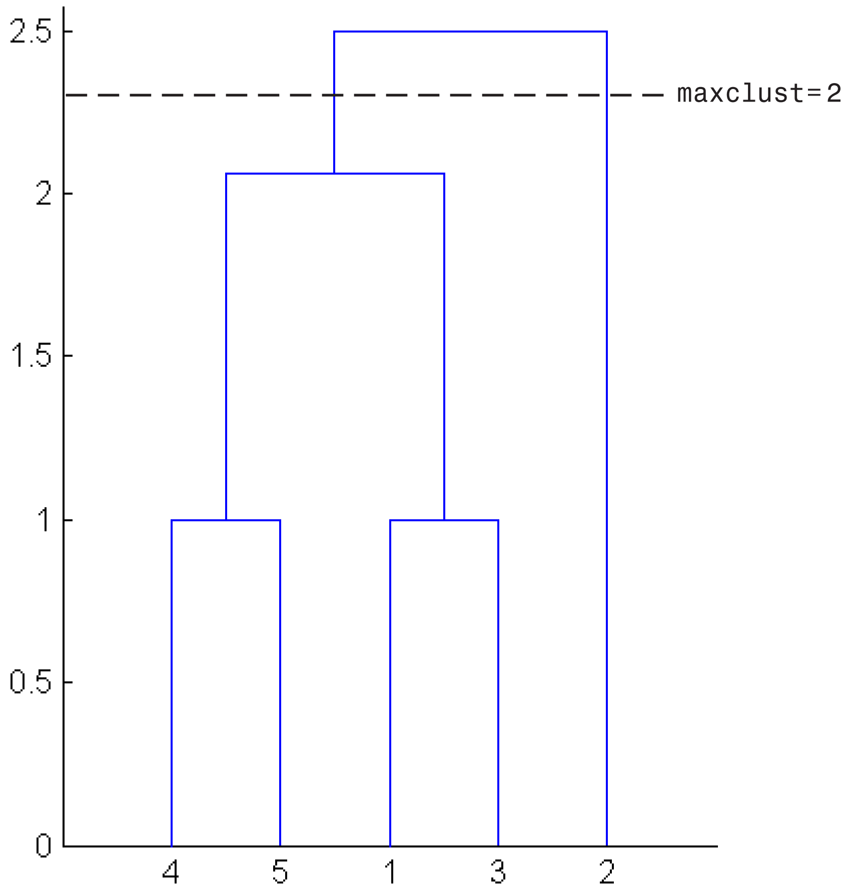
When clusters are formed in this way, the cutoff value is applied to the inconsistency coefficient. These clusters may, but do not necessarily, correspond to a horizontal slice across the dendrogram at a certain height. If you want clusters corresponding to a horizontal slice of the dendrogram, you can either use the `criterion` option to specify that the cutoff should be based on distance rather than inconsistency, or you can specify the number of clusters directly as described in the following section.

Specifying Arbitrary Clusters. Instead of letting the `cluster` function create clusters determined by the natural divisions in the data set, you can specify the number of clusters you want created.

For example, you can specify that you want the `cluster` function to partition the sample data set into two clusters. In this case, the `cluster` function creates one cluster containing objects 1, 3, 4, and 5 and another cluster containing object 2.

```
T = cluster(Z,'maxclust',2)
T =
     2
     1
     2
     2
     2
```

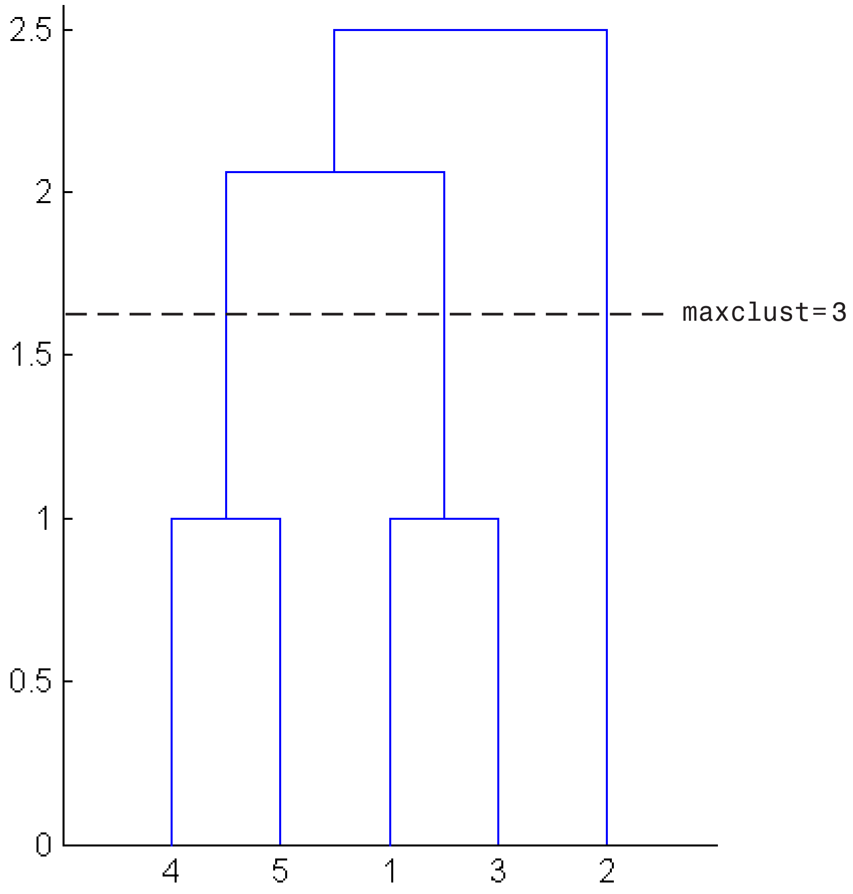
To help you visualize how the `cluster` function determines these clusters, the following figure shows the dendrogram of the hierarchical cluster tree. The horizontal dashed line intersects two lines of the dendrogram, corresponding to setting `'maxclust'` to `2`. These two lines partition the objects into two clusters: the objects below the left-hand line, namely 1, 3, 4, and 5, belong to one cluster, while the object below the right-hand line, namely 2, belongs to the other cluster.



On the other hand, if you set `'maxclust'` to `3`, the cluster function groups objects 4 and 5 in one cluster, objects 1 and 3 in a second cluster, and object 2 in a third cluster. The following command illustrates this.

```
T = cluster(Z,'maxclust',3)
T =
     1
     3
     1
     2
     2
```

This time, the `cluster` function cuts off the hierarchy at a lower point, corresponding to the horizontal line that intersects three lines of the dendrogram in the following figure.



K-Means Clustering

This section gives a description and an example of using the MATLAB function for K-means clustering, `kmeans`.

- "Example: Clustering Data in Four Dimensions" on page 9-47

Overview of K-Means Clustering

K-means clustering can best be described as a partitioning method. That is, the function `kmeans` partitions the observations in your data into K mutually exclusive clusters, and returns a vector of indices indicating to which of the `k` clusters it has assigned each observation. Unlike the hierarchical clustering methods used in `linkage` (see "Hierarchical Clustering" on page 9-29), `kmeans` does not create a tree structure to describe the groupings in your data, but rather creates a single level of clusters. Another difference is that K-means clustering uses the actual observations of objects or individuals in your data, and not just their proximities. These differences often mean that `kmeans` is more suitable for clustering large amounts of data.

`kmeans` treats each observation in your data as an object having a location in space. It finds a partition in which objects within each cluster are as close to each other as possible, and as far from objects in other clusters as possible. You can choose from five different distance measures, depending on the kind of data you are clustering.

Each cluster in the partition is defined by its member objects and by its centroid, or center. The centroid for each cluster is the point to which the sum of distances from all objects in that cluster is minimized. `kmeans` computes cluster centroids differently for each distance measure, to minimize the sum with respect to the measure that you specify.

`kmeans` uses an iterative algorithm that minimizes the sum of distances from each object to its cluster centroid, over all clusters. This algorithm moves objects between clusters until the sum cannot be decreased further. The result is a set of clusters that are as compact and well-separated as possible. You can control the details of the minimization using several optional input parameters to `kmeans`, including ones for the initial values of the cluster centroids, and for the maximum number of iterations.

Example: Clustering Data in Four Dimensions

This example explores possible clustering in four-dimensional data by analyzing the results of partitioning the points into three, four, and five clusters.

Note Because each part of this example generates random numbers sequentially, i.e., without setting a new state, you must perform all steps in sequence to duplicate the results shown. If you perform the steps out of sequence, the answers will be essentially the same, but the intermediate results, number of iterations, or ordering of the silhouette plots may differ.

Creating Clusters and Determining Separation. First, load some data.

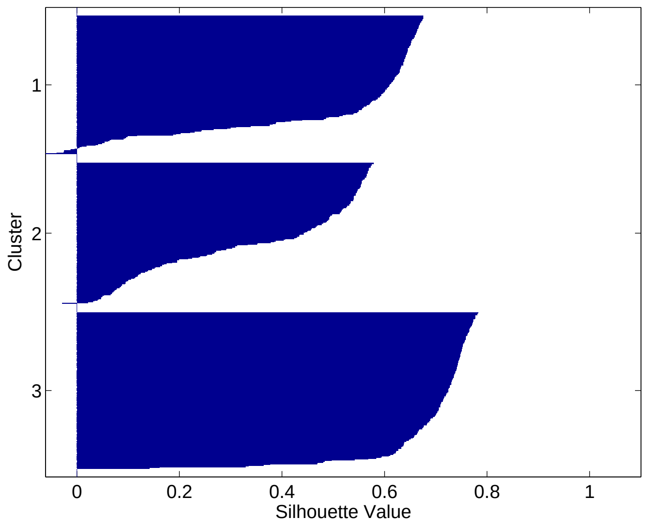
```
load kmeansdata;
size(X)
ans =
   560     4
```

Even though these data are four-dimensional, and cannot be easily visualized, `kmeans` enables you to investigate whether a group structure exists in them. Call `kmeans` with `k`, the desired number of clusters, equal to `3`. For this example, specify the city block distance measure, and use the default starting method of initializing centroids from randomly selected data points.

```
idx3 = kmeans(X,3,'distance','city');
```

To get an idea of how well-separated the resulting clusters are, you can make a silhouette plot using the cluster indices output from `kmeans`. The silhouette plot displays a measure of how close each point in one cluster is to points in the neighboring clusters. This measure ranges from +1, indicating points that are very distant from neighboring clusters, through 0, indicating points that are not distinctly in one cluster or another, to -1, indicating points that are probably assigned to the wrong cluster. `silhouette` returns these values in its first output.

```
[silh3,h] = silhouette(X,idx3,'city');
xlabel('Silhouette Value')
ylabel('Cluster')
```



From the silhouette plot, you can see that most points in the third cluster have a large silhouette value, greater than 0.6, indicating that the cluster is somewhat separated from neighboring clusters. However, the second cluster contains many points with low silhouette values, and the first contains a few points with negative values, indicating that those two clusters are not well separated.

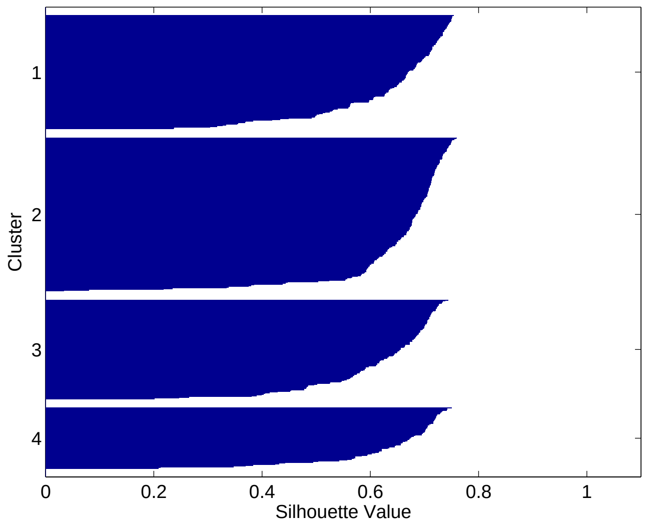
Determining the Correct Number of Clusters. Increase the number of clusters to see if `kmeans` can find a better grouping of the data. This time, use the optional `'display'` parameter to print information about each iteration.

```
idx4 = kmeans(X,4, 'dist','city', 'display','iter');
  iter  phase      num          sum
     1      1      560      2897.56
     2      1       53      2736.67
     3      1       50      2476.78
     4      1      102      1779.68
     5      1        5       1771.1
     6      2        0       1771.1
6 iterations, total sum of distances = 1771.1
```

Notice that the total sum of distances decreases at each iteration as kmeans reassigns points between clusters and recomputes cluster centroids. In this case, the second phase of the algorithm did not make any reassignments, indicating that the first phase reached a minimum after five iterations. In some problems, the first phase might not reach a minimum, but the second phase always will.

A silhouette plot for this solution indicates that these four clusters are better separated than the three in the previous solution.

```
[silh4,h] = silhouette(X,idx4,'city');
xlabel('Silhouette Value')
ylabel('Cluster')
```



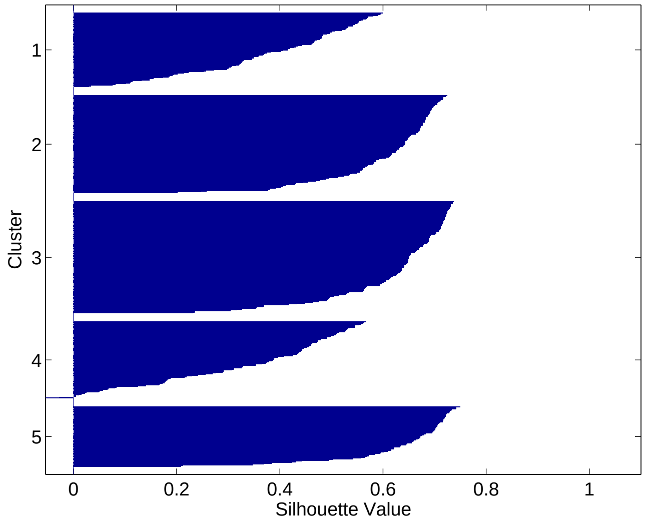
A more quantitative way to compare the two solutions is to look at the average silhouette values for the two cases.

```
mean(silh3)
ans =
      0.52594
mean(silh4)
```

```
ans =
      0.63997
```

Finally, try clustering the data using five clusters.

```
idx5 = kmeans(X,5,'dist','city','replicates',5);
[silh5,h] = silhouette(X,idx5,'city');
xlabel('Silhouette Value')
ylabel('Cluster')
mean(silh5)
ans =
      0.52657
```



This silhouette plot indicates that this is probably not the right number of clusters, since two of the clusters contain points with mostly low silhouette values. Without some knowledge of how many clusters are really in the data, it is a good idea to experiment with a range of values for `k`.

Avoiding Local Minima. Like many other types of numerical minimizations, the solution that kmeans reaches often depends on the starting points. It is possible for kmeans to reach a local minimum, where reassigning any one point to a new cluster would increase the total sum of point-to-centroid distances, but where a better solution does exist. However, you can use the optional 'replicates' parameter to overcome that problem.

For four clusters, specify five replicates, and use the 'display' parameter to print out the final sum of distances for each of the solutions.

```
[idx4,cent4,sumdist] = kmeans(X,4,'dist','city',...
                        'display','final','replicates',5);
17 iterations, total sum of distances = 2303.36
 5 iterations, total sum of distances = 1771.1
 6 iterations, total sum of distances = 1771.1
 5 iterations, total sum of distances = 1771.1
 8 iterations, total sum of distances = 2303.36
```

The output shows that, even for this relatively simple problem, non-global minima do exist. Each of these five replicates began from a different randomly selected set of initial centroids, and kmeans found two different local minima. However, the final solution that kmeans returns is the one with the lowest total sum of distances, over all replicates.

```
sum(sumdist)
ans =
       1771.1
```

Multidimensional Scaling

The following sections explain how to perform multidimensional scaling, using the functions `cmdscale` and `mdscale`:

Overview

One of the most important goals in visualizing data is to get a sense of how near or far points are from each other. Often, you can do this with a scatter plot. However, for some analyses, the data that you have might not be in the form of points at all, but rather in the form of pairwise similarities or dissimilarities between cases, observations, or subjects. Without any points, you cannot make a scatter plot.

Even if your data are in the form of points rather than pairwise distances, a scatter plot of those data might not be useful. For some kinds of data, the relevant way to measure how "near" two points are might not be their Euclidean distance. While scatter plots of the raw data make it easy to compare Euclidean distances, they are not always useful when comparing other kinds of inter-point distances, city block distance for example, or even more general dissimilarities. Also, with a large number of variables, it is very difficult to visualize distances unless the data can be represented in a small number of dimensions. Some sort of dimension reduction is usually necessary.

Multidimensional scaling (MDS) is a set of methods that address all these problems. MDS allows you to visualize how near points are to each other for many kinds of distance or dissimilarity measures and can produce a representation of your data in a small number of dimensions. MDS does not require raw data, but only a matrix of pairwise distances or dissimilarities.

Classical Multidimensional Scaling

The function `cmdscale` performs classical (metric) multidimensional scaling, also known as principal coordinates analysis. `cmdscale` takes as an input a matrix of inter-point distances and creates a configuration of points. Ideally, those points are in two or three dimensions, and the Euclidean distances between them reproduce the original distance matrix. Thus, a scatter plot of the points created by `cmdscale` provides a visual representation of the original distances.

A Simple Example

As a very simple example, you can reconstruct a set of points from only their inter-point distances. First, create some four dimensional points with a small component in their fourth coordinate, and reduce them to distances.

```
X = [ normrnd(0,1,10,3), normrnd(0,.1,10,1) ];
D = pdist(X,'euclidean');
```

Next, use `cmdscale` to find a configuration with those inter-point distances. `cmdscale` accepts distances as either a square matrix, or, as in this example, in the vector upper-triangular form produced by `pdist`.

```
[Y,eigvals] = cmdscale(D);
```

`cmdscale` produces two outputs. The first output, `Y`, is a matrix containing the reconstructed points. The second output, `eigvals`, is a vector containing the sorted eigenvalues of what is often referred to as the "scalar product matrix," which, in the simplest case, is equal to `Y*Y'`. The relative magnitudes of those eigenvalues indicate the relative contribution of the corresponding columns of `Y` in reproducing the original distance matrix `D` with the reconstructed points.

```
format short g
[eigvals eigvals/max(abs(eigvals))]
ans =
       12.623            1
       4.3699      0.34618
       1.9307      0.15295
     0.025884    0.0020505
   1.7192e-015  1.3619e-016
   6.8727e-016  5.4445e-017
   4.4367e-017  3.5147e-018
```

```
-9.2731e-016 -7.3461e-017
 -1.327e-015 -1.0513e-016
-1.9232e-015 -1.5236e-016
```

If eigvals contains only positive and zero (within round-off error) eigenvalues, the columns of Y corresponding to the positive eigenvalues provide an exact reconstruction of D, in the sense that their inter-point Euclidean distances, computed using pdist, for example, are identical (within round-off) to the values in D.

```
maxerr4 = max(abs(D - pdist(Y))) % exact reconstruction
maxerr4 =
  2.6645e-015
```

If two or three of the eigenvalues in eigvals are much larger than the rest, then the distance matrix based on the corresponding columns of Y nearly reproduces the original distance matrix D. In this sense, those columns form a lower-dimensional representation that adequately describes the data. However it is not always possible to find a good low-dimensional reconstruction.

```
% good reconstruction in 3D
maxerr3 = max(abs(D - pdist(Y(:,1:3))))
maxerr3 =
     0.029728

% poor reconstruction in 2D
maxerr2 = max(abs(D - pdist(Y(:,1:2))))
maxerr2 =
      0.91641
```

The reconstruction in three dimensions reproduces D very well, but the reconstruction in two dimensions has errors that are of the same order of magnitude as the largest values in D.

```
max(max(D))
ans =
       3.4686
```

Often, eigvals contains some negative eigenvalues, indicating that the distances in D cannot be reproduced exactly. That is, there might not be any configuration of points whose inter-point Euclidean distances are given by D. If the largest negative eigenvalue is small in magnitude relative to the largest positive eigenvalues, then the configuration returned by cmdscale might still reproduce D well. "Example: Reconstructing a Map from Intercity Distances" on page 9-60 demonstrates this.

Nonclassical Metric Multidimensional Scaling

The function cmdscale performs classical multidimensional scaling (MDS). Statistics Toolbox also includes the function mdscale to perform nonclassical MDS. As with cmdcale, you can use mdscale either to visualize dissimilarity data for which no "locations" exist, or to visualize high-dimensional data by reducing its dimensionality. Both functions take a matrix of dissimilarities as an input and produce a configuration of points. However, mdscale offers a choice of different criteria to construct the configuration, and allows missing data and weights.

For example, the cereal data include measurements on 10 variables describing breakfast cereals. You can use mdscale to visualize these data in two dimensions. First, load the data. For clarity, this example code selects a subset of 22 of the observations.

```
load cereal.mat
X = [Calories Protein Fat Sodium Fiber ...
    Carbo Sugars Shelf Potass Vitamins];
X = X(strmatch('G',Mfg),:); % take a subset from a
                            % single manufacturer
size(X)
ans =
    22 10
```

Then use pdist to transform the 10-dimensional data into dissimilarities. The output from pdist is a symmetric dissimilarity matrix, stored as a vector containing only the (23*22/2) elements in its upper triangle.

```
dissimilarities = pdist(zscore(X),'cityblock');
size(dissimilarities)
ans =
     1   231
```

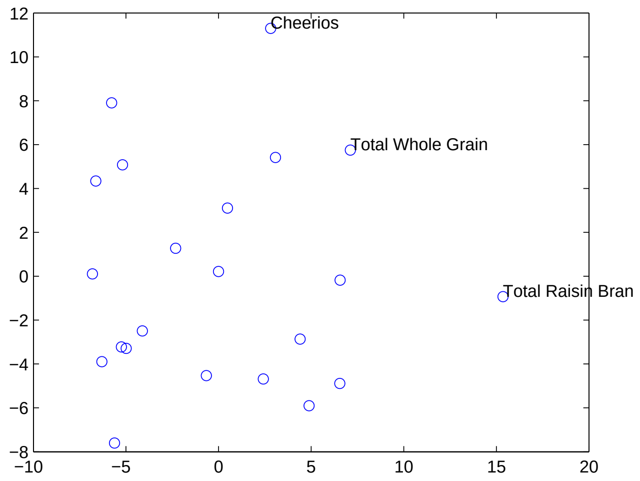
This example code first standardizes the cereal data, and then uses city block distance as a dissimilarity. The choice of transformation to dissimilarities is application-dependent, and the choice here is only for simplicity. In some applications, the original data are already in the form of dissimilarities.

Next, use mdscale to perform metric MDS. Unlike cmdscale, you must specify the desired number of dimensions, and the method to use to construct the output configuration. For this example, use two dimensions. The metric STRESS criterion is a common method for computing the output; for other choices, see the mdscale reference page in the online documentation. The second output from mdscale is the value of that criterion evaluated for the output configuration. It measures the how well the inter-point distances of the output configuration approximate the original input dissimilarities.

```
[Y,stress] =...
mdscale(dissimilarities,2,'criterion','metricstress');
stress
stress =
    0.1856
```

A scatterplot of the output from mdscale represents the original 10-dimensional data in two dimensions, and you can use the gname function to label selected points.

```
plot(Y(:,1),Y(:,2),'o');
gname(Name(strmatch('G',Mfg)))
```



Nonmetric Multidimensional Scaling

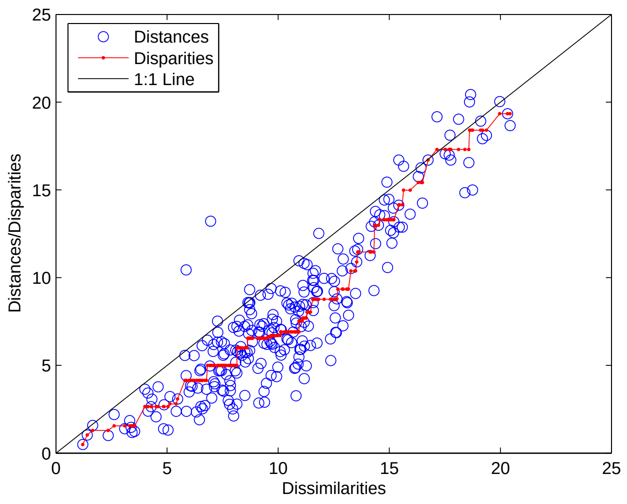
Metric multidimensional scaling creates a configuration of points whose inter-point distances approximate the given dissimilarities. This is sometimes too strict a requirement, and non-metric scaling is designed to relax it a bit. Instead of trying to approximate the dissimilarities themselves, non-metric scaling approximates a nonlinear, but monotonic, transformation of them. Because of the monotonicity, larger or smaller distances on a plot of the output will correspond to larger or smaller dissimilarities, respectively. However, the nonlinearity implies that `mdscale` only attempts to preserve the ordering of dissimilarities. Thus, there may be contractions or expansions of distances at different scales.

You use `mdscale` to perform nonmetric MDS in much the same way as for metric scaling. The nonmetric STRESS criterion is a common method for computing the output; for more choices, see the `mdscale` reference page in the online documentation. As with metric scaling, the second output from `mdscale` is the value of that criterion evaluated for the output configuration. For nonmetric scaling, however, it measures the how well the inter-point distances of the output configuration approximate the disparities. The disparities are returned in the third output. They are the transformed values of the original dissimilarities.

```
[Y,stress,disparities] = ...
mdscale(dissimilarities,2,'criterion','stress');
stress
stress =
    0.1562
```

To check the fit of the output configuration to the dissimilarities, and to understand the disparities, it helps to make a Shepard plot.

```
distances = pdist(Y);
[dum,ord] = sortrows([disparities(:) dissimilarities(:)]);
plot(dissimilarities,distances,'bo', ...
     dissimilarities(ord),disparities(ord),'r.-', ...
     [0 25],[0 25],'k-');
xlabel('Dissimilarities'); ylabel('Distances/Disparities')
legend({'Distances' 'Disparities' '1:1 Line'},...
'Location','NorthWest');
```



This plot shows that `mdscale` has found a configuration of points in two dimensions whose inter-point distances approximates the disparities, which in turn are a nonlinear transformation of the original dissimilarities. The concave shape of the disparities as a function of the dissimilarities indicates that fit tends to contract small distances relative to the corresponding dissimilarities. This may be perfectly acceptable in practice.

mdscale uses an iterative algorithm to find the output configuration, and the results can often depend on the starting point. By default, mdscale uses cmdscale to construct an initial configuration, and this choice often leads to a globally best solution. However, it is possible for mdscale to stop at a configuration that is a local minimum of the criterion. Such cases can be diagnosed and often overcome by running mdscale multiple times with different starting points. You can do this using the 'start' and 'replicates' parameters. The following code runs five replicates of MDS, each starting at a different randomly-chosen initial configuration. The criterion value is printed out for each replication; mdscale returns the configuration with the best fit.

```
opts = statset('Display','final');
[Y,stress] =...
mdscale(dissimilarities,2,'criterion','stress',...
'start','random','replicates',5,'Options',opts);
90 iterations, Final stress criterion = 0.156209
100 iterations, Final stress criterion = 0.195546
116 iterations, Final stress criterion = 0.156209
85 iterations, Final stress criterion = 0.156209
106 iterations, Final stress criterion = 0.17121
```

Notice that mdscale finds several different local solutions, some of which do not have as low a stress value as the solution found with the cmdscale starting point.

Example: Reconstructing a Map from Intercity Distances

Given only the distances between 10 US cities, cmdscale can construct a map of those cities. First, create the distance matrix and pass it to cmdscale. In this example, D is a full distance matrix: it is square and symmetric, has positive entries off the diagonal, and has zeros on the diagonal.

```
cities =
{'Atl','Chi','Den','Hou','LA','Mia','NYC','SF','Sea','WDC'};
D = [    0  587 1212  701 1936  604  748 2139 2182   543;
       587    0  920  940 1745 1188  713 1858 1737   597;
```

```
         1212  920    0  879  831 1726 1631  949 1021  1494;
          701  940  879    0 1374  968 1420 1645 1891  1220;
         1936 1745  831 1374    0 2339 2451  347  959  2300;
          604 1188 1726  968 2339    0 1092 2594 2734   923;
          748  713 1631 1420 2451 1092    0 2571 2408   205;
         2139 1858  949 1645  347 2594 2571    0  678  2442;
         2182 1737 1021 1891  959 2734 2408  678    0  2329;
          543  597 1494 1220 2300  923  205 2442 2329     0];
   [Y,eigvals] = cmdscale(D);
```

Next, look at the eigenvalues returned by `cmdscale`. Some of these are negative, indicating that the original distances are not Euclidean. This is because of the curvature of the earth.

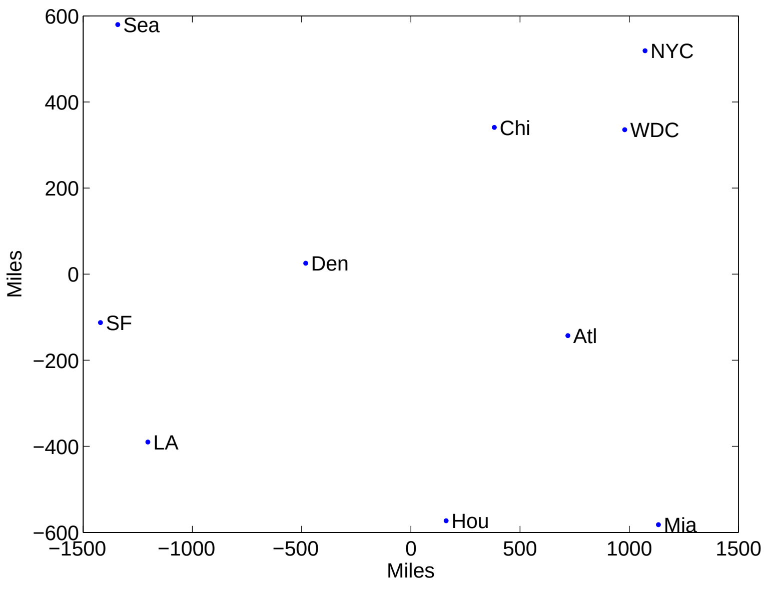
```
   format short g
   [eigvals eigvals/max(abs(eigvals))]
   ans =
     9.5821e+006            1
     1.6868e+006      0.17604
          8157.3    0.0008513
          1432.9   0.00014954
          508.67  5.3085e-005
          25.143   2.624e-006
     5.3394e-010  5.5722e-017
          -897.7 -9.3685e-005
         -5467.6   -0.0005706
          -35479   -0.0037026
```

However, in this case, the two largest positive eigenvalues are much larger in magnitude than the remaining eigenvalues. So, despite the negative eigenvalues, the first two coordinates of `Y` are sufficient for a reasonable reproduction of `D`.

```
   Dtriu = D(find(tril(ones(10),-1)))';
   maxrelerr = max(abs(Dtriu - pdist(Y(:,1:2)))) ./ max(Dtriu)
   maxrelerr =
       0.0075371
```

Here is a plot of the reconstructed city locations as a map. The orientation of the reconstruction is arbitrary. In this case, it happens to be close to, although not exactly, the correct orientation.

```
plot(Y(:,1),Y(:,2),'.');
text(Y(:,1)+25,Y(:,2),cities);
xlabel('Miles'); ylabel('Miles');
```



Statistical Process Control

Statistical process control (SPC) refers to a number of different methods for monitoring and assessing the quality of manufactured goods. Combined with methods from the Chapter 11, "Design of Experiments", SPC is used in programs that define, measure, analyze, improve, and control development and production processes. These programs are often implemented using "Design for Six Sigma" methodologies.

The following sections describe the SPC functions available in Statistics Toolbox:

Control Charts

A control chart displays measurements of process samples over time. The measurements are plotted together with user-defined *specification limits* and process-defined *control limits*. The process can then be compared with its specifications—to see if it is *in control* or *out of control*.

The chart is just a monitoring tool. Control activity might occur if the chart indicates an undesirable, systematic change in the process. The control chart is used to discover the variation, so that the process can be adjusted to reduce it.

Control charts are created with the `controlchart` function. Any of the following chart types may be specified:

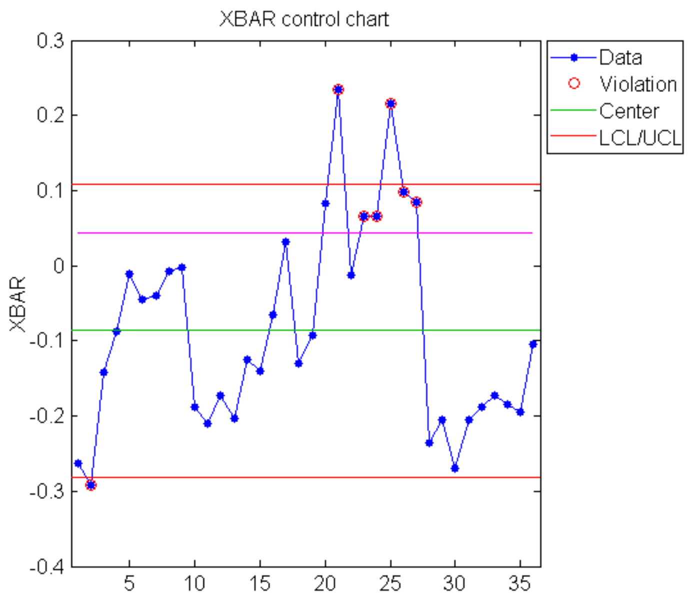
- Xbar or mean
- Standard deviation
- Range
- Exponentially weighted moving average
- Individual observation
- Moving range of individual observations
- Moving average of individual observations
- Proportion defective
- Number of defectives
- Defects per unit
- Count of defects

Control rules are specified with the `controlrules` function.

For example, the following commands create an xbar chart, using the "Western Electric 2" rule (2 of 3 points at least 2 standard errors above the center line) to mark out of control measurements:

```
load parts;
st = controlchart(runout,'rules','we2');
```

```
x = st.mean;
cl = st.mu;
se = st.sigma./sqrt(st.n);
hold on
plot(cl+2*se,'m')
```



Measurements that violate the control rule can then be identified:

```
R = controlrules('we2',x,cl,se);
I = find(R)
I =
    21
    23
    24
    25
    26
    27
```

Capability Studies

Before going into production, many manufacturers run a *capability study* to determine if their process will run within specifications enough of the time. *Capability indices* produced by such a study are used to estimate expected percentages of defective parts.

Capability studies are conducted with the `capability` function. The following capability indices are produced:

- `mu` — Sample mean
- `sigma` — Sample standard deviation
- `P` — Estimated probability of being within the lower (`L`) and upper (`U`) specification limits
- `Pl` — Estimated probability of being below `L`
- `Pu` — Estimated probability of being above `U`
- `Cp` — `(U-L)/(6*sigma)`
- `Cpl` — `(mu-L)./(3.*sigma)`
- `Cpu` — `(U-mu)./(3.*sigma)`
- `Cpk` — `min(Cpl,Cpu)`

As an example, simulate a sample from a process with a mean of 3 and a standard deviation of 0.005:

```
data = normrnd(3,0.005,100,1);
```

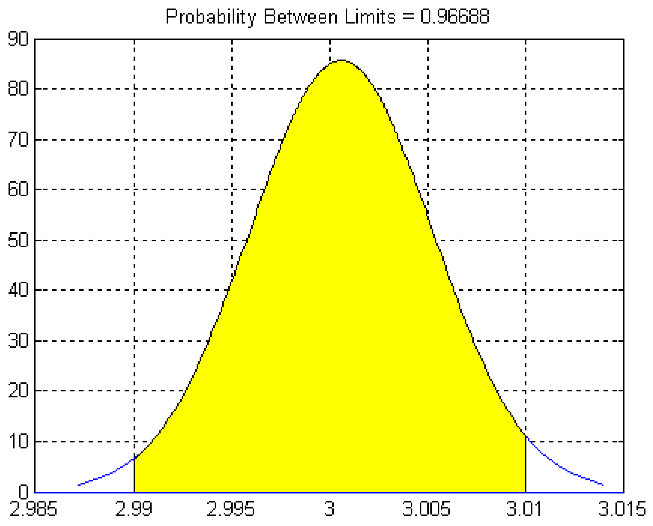
Compute capability indices if the process has an upper specification limit of 3.01 and a lower specification limit of 2.99:

```
S = capability(data,[2.99 3.01])
S =
       mu: 3.0006
    sigma: 0.0047
        P: 0.9669
       Pl: 0.0116
       Pu: 0.0215
```

```
 Cp: 0.7156
Cpl: 0.7567
Cpu: 0.6744
Cpk: 0.6744
```

Visualize the specification and process widths:

```
capaplot(data,[2.99 3.01]);
grid on
```



Design of Experiments

Introduction

There is a world of difference between data and information. To extract information from data you have to make assumptions about the system that generated the data. Using these assumptions and physical theory you may be able to develop a mathematical model of the system.

Generally, even rigorously formulated models have some unknown constants. The goal of experimentation is to acquire data that enable you to estimate these constants.

But why do you need to experiment at all? You could instrument the system you want to study and just let it run. Sooner or later you would have all the data you could use.

In fact, this is a fairly common approach. There are three characteristics of historical data that pose problems for statistical modeling:

- Suppose you observe a change in the operating variables of a system followed by a change in the outputs of the system. That does *not* necessarily mean that the change in the system *caused* the change in the outputs.
- A common assumption in statistical modeling is that the observations are independent of each other. This is not the way a system in normal operation works.
- Controlling a system in operation often means changing system variables in tandem. But if two variables change together, it is impossible to separate their effects mathematically.

Designed experiments directly address these problems. The overwhelming advantage of a designed experiment is that you actively manipulate the system you are studying. With Design of Experiments (DOE) you may generate fewer data points than by using passive instrumentation, but the quality of the information you get will be higher.

Full Factorial Designs

Suppose you want to determine whether the variability of a machining process is due to the difference in the lathes that cut the parts or the operators who run the lathes.

If the same operator always runs a given lathe then you cannot tell whether the machine or the operator is the cause of the variation in the output. By allowing every operator to run every lathe, you can separate their effects.

This is a factorial approach. `fullfact` is the function that generates the design. Suppose you have four operators and three machines. What is the factorial design?

```
d = fullfact([4 3])

d =
     1     1
     2     1
     3     1
     4     1
     1     2
     2     2
     3     2
     4     2
     1     3
     2     3
     3     3
     4     3
```

Each row of `d` represents one operator/machine combination. Note that there are 4*3 = 12 rows.

One special subclass of factorial designs is when all the variables take only two values. Suppose you want to quickly determine the sensitivity of a process to high and low values of three variables.

```
d2 = ff2n(3)

d2 =
0    0    0
0    0    1
0    1    0
0    1    1
1    0    0
1    0    1
1    1    0
1    1    1
```

There are $2^3 = 8$ combinations to check.

Fractional Factorial Designs

One difficulty with factorial designs is that the number of combinations increases exponentially with the number of variables you want to manipulate.

For example, the sensitivity study discussed above might be impractical if there were seven variables to study instead of just three. A full factorial design would require $2^7 = 128$ runs!

If you assume that the variables do not act synergistically in the system, you can assess the sensitivity with far fewer runs. The theoretical minimum number is eight. A design known as the Plackett-Burman design uses a Hadamard matrix to define this minimal number of runs. To see the design (X) matrix for the Plackett-Burman design, use the `hadamard` function.

```
X = hadamard(8)

X =
       1     1     1     1     1     1     1     1
       1    -1     1    -1     1    -1     1    -1
       1     1    -1    -1     1     1    -1    -1
       1    -1    -1     1     1    -1    -1     1
       1     1     1     1    -1    -1    -1    -1
       1    -1     1    -1    -1     1    -1     1
       1     1    -1    -1    -1    -1     1     1
       1    -1    -1     1    -1     1     1    -1
```

The last seven columns are the actual variable settings (-1 for low, 1 for high.) The first column (all ones) enables you to measure the mean effect in the linear equation, $y = X\beta + \varepsilon$.

The Plackett-Burman design enables you to study the main (linear) effects of each variable with a small number of runs. It does this by using a fraction, in this case 8/128, of the runs required for a full factorial design. A drawback of this design is that if the effect of one variable does vary with the value of another variable, the estimated effects are biased (that is, they tend to be off by a systematic amount).

At a cost of a somewhat larger design, you can find a fractional factorial that is much smaller than a full factorial, but that allows estimation of main effects independent of interactions between pairs of variables. You can do this by specifying generators that control the confounding between variables.

For example, suppose you create a design with the first four variables varying independently as in a full factorial, but with the other three variables formed by multiplying different triplets of the first four. With this design, the effects of the last three variables are confounded with three-way interactions among the first four variables. The estimated effect of any single variable, however, is not confounded with (is independent of) interaction effects between any pair of variables. Interaction effects are confounded with each other. A design like this is known as a *resolution 4 design*.

Box, Hunter, and Hunter [4] present the properties of fractional factorial designs, and provide a catalog of generators for producing designs for various numbers of factors and various resolutions.

For example, the following design uses the generators strings in this catalog to produce a resolution 4 design for 7 factors using 16 runs.

The `fracfact` function can produce this fractional factorial design using the generator strings that Box, Hunter, and Hunter provide.

```
X = fracfact('a b c d abc bcd acd')

X =
    -1    -1    -1    -1    -1    -1    -1
    -1    -1    -1     1    -1     1     1
    -1    -1     1    -1     1     1     1
    -1    -1     1     1     1    -1    -1
    -1     1    -1    -1     1     1    -1
    -1     1    -1     1     1    -1     1
    -1     1     1    -1    -1    -1     1
    -1     1     1     1    -1     1    -1
     1    -1    -1    -1     1    -1     1
     1    -1    -1     1     1     1    -1
     1    -1     1    -1    -1     1    -1
     1    -1     1     1    -1    -1     1
     1     1    -1    -1    -1     1     1
     1     1    -1     1    -1    -1    -1
     1     1     1    -1     1    -1    -1
     1     1     1     1     1     1     1
```

The fracfactgen function can find the appropriate generators to fit a model that you specify. For example, you want the generators for a design that can fit the main effectors of 7 factors (a-g), using 2^4=16 runs, and having resolution 4:

```
fracfactgen('a b c d e f g',4,4)
ans =
    'a'
    'b'
    'c'
    'd'
    'bcd'
    'acd'
    'abd'
```

These generators are not the same as the ones from the catalog in Box, Hunter, and Hunter, but they produce a design with equivalent properties. The fracfactgen uses an efficient search algorithm to find generators that meet the requirements that you specify. This search can still be time consuming, though, if the number of factors or model terms is large.

Response Surface Designs

Sometimes simple linear and interaction models are not adequate. For example, suppose that the outputs are defects or yield, and the goal is to minimize defects and maximize yield. If these optimal points are in the interior of the region in which the experiment is to be conducted, you need a mathematical model that can represent curvature so that it has a local optimum. The simplest such model has the quadratic form

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_{12} X_1 X_2 + \beta_{11} X_1^2 + \beta_{22} X_2^2$$

containing linear terms for all factors, squared terms for all factors, and products of all pairs of factors.

Designs for fitting these types of models are known as response surface designs. One such design is the full factorial design having three values for each input. Although Statistics Toolbox is capable of generating this design, it is not really a satisfactory design in most cases because it has many more runs than are necessary to fit the model.

The two most common designs generally used in response surface modeling are central composite designs and Box-Behnken designs. In these designs the inputs take on three or five distinct values (levels), but not all combinations of these values appear in the design.

The functions described here produce specific response surface designs:

- "Central Composite Designs" on page 11-8
- "Box-Behnken Designs" on page 11-11

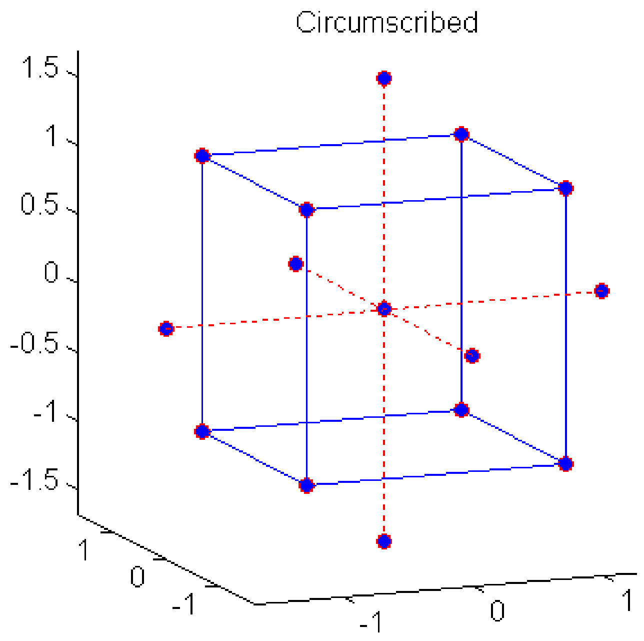
If these do not serve your purposes, consider creating a D-optimal design. "Design of Experiments Demo" on page 11-11 uses a D-optimal design to fit data that conforms to a response surface model. For more information see "D-Optimal Designs" on page 11-19.

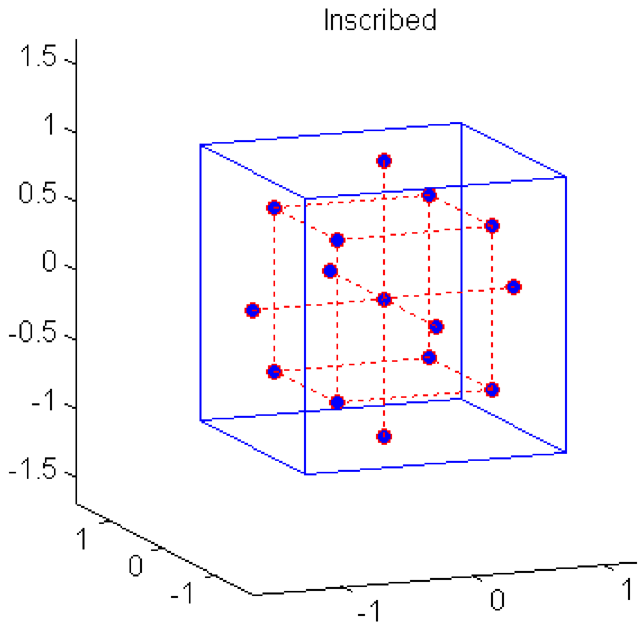
Central Composite Designs

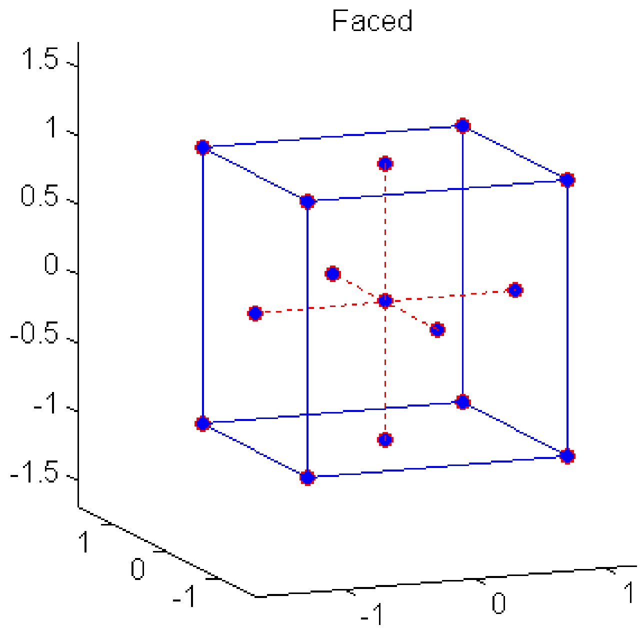
Central composite designs are response surface designs that can fit a full quadratic model. To picture a central composite design, imagine you have several factors that can vary between low and high values. For convenience, suppose each factor varies from -1 to +1.

One central composite design consists of cube points at the corners of a unit cube that is the product of the intervals [-1,1], star points along the axes at or outside the cube, and center points at the origin.

Central composite designs are of three types. Circumscribed (CCC) designs are as described above. Inscribed (CCI) designs are as described above, but scaled so the star points take the values -1 and +1, and the cube points lie in the interior of the cube. Faced (CCF) designs have the star points on the faces of the cube. Faced designs have three levels per factor, in contrast with the other types, which have five levels per factor. The following figure shows these three types of designs for three factors.




Inscribed
1.5
1
0.5
0
-0.5
-1
-1.5
1
0
-1
-1
0
1

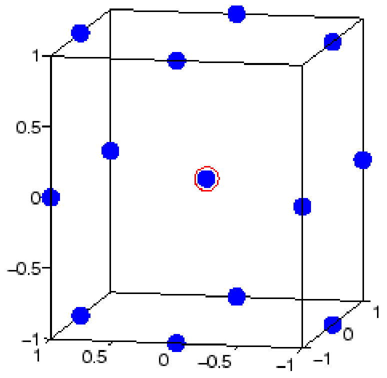

Faced
1.5
1
0.5
0
-0.5
-1
-1.5
1
0
-1
-1
0
1

Box-Behnken Designs

Like central composite designs, Box-Behnken designs are response surface designs that can fit a full quadratic model. Unlike most central composite designs, Box-Behnken designs use just three levels of each factor. This makes them appealing when the factors are quantitative but the set of achievable values is small.

Central composite faced (CCF) designs also use just three factor levels. However, they are not rotatable as Box-Behnken designs are. On the other hand, Box-Behnken designs can be expected to have poorer prediction ability in the corners of the cube that encloses the design, because unlike CCF designs they do not include points at the corners of that cube.

The following figure shows a Box-Behnken design for three factors, with the circled point appearing at the origin and possibly repeated for several runs. A repeated center point makes it possible to compute an estimate of the error term that does not depend on the fitted model. For this design all points except the center point appear at a distance $\sqrt{2}$ from the origin. That does not hold true for Box-Behnken designs with different numbers of factors.



Design of Experiments Demo

The `rsmdemo` utility is an interactive graphic environment that demonstrates the design of experiments and surface fitting through the simulation of a chemical reaction. The goal of the demo is to find the levels of the reactants needed to maximize the reaction rate.

Suitable designs for this experiment include the central composite designs and Box-Behnken designs, described in the previous two sections, and the D-optimal designs, described in "D-Optimal Designs" on page 11-19. This demo uses D-optimal designs.

There are two parts to the demo:

- "Comparing Results from Trial-and-Error Data and a Designed Experiment" on page 11-12
- "Comparing Results Using a Polynomial Model and a Nonlinear Model" on page 11-16

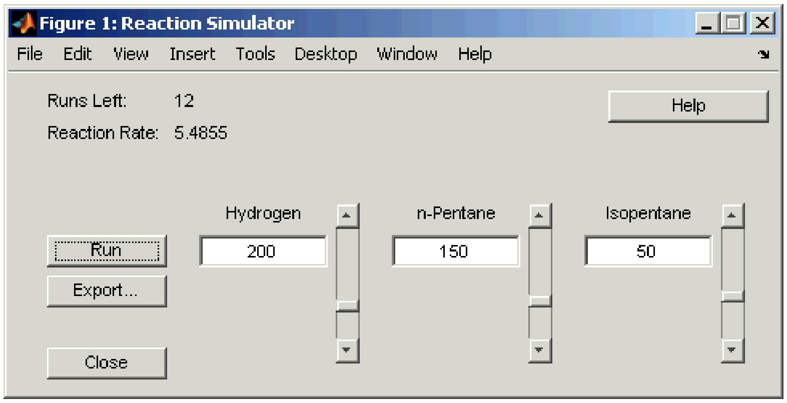
Comparing Results from Trial-and-Error Data and a Designed Experiment

This part of the experiment compares the results obtained using data gathered through trial and error and using data from a designed experiment:

1 To begin, run the `rsmdemo` function.

```
rsmdemo
```

2 Click **Run** in the Reaction Simulator window to generate a test reaction for the trial and error phase of the demo.

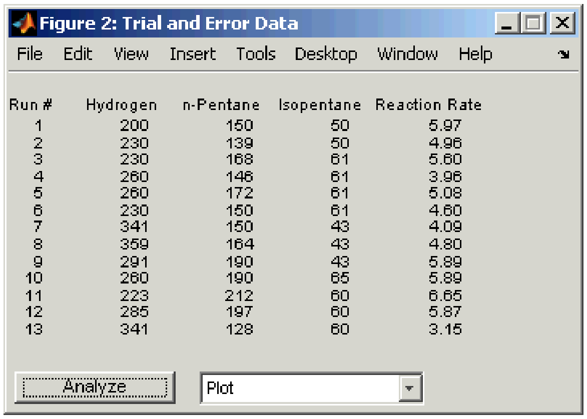


Runs Left: 12
Reaction Rate: 5.4855

Hydrogen	n-Pentane	Isopentane
200	150	50

Figure 1: Reaction Simulator
File Edit View Insert Tools Desktop Window Help
Help
Run
Export...
Close

To perform the experiment, you can click **Run** as many as 13 times. For each run, based on the results of previous runs, you can move the sliders in the Reaction Simulator window to change the levels of the reactants to increase or decrease the reaction rate. Each time you click the **Run** button, the levels for the reactants and results of the run are displayed in the Trial and Error Data window, as shown in the following figure after 13 trials.

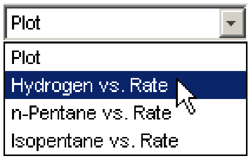


Run #	Hydrogen	n-Pentane	Isopentane	Reaction Rate
1	200	150	50	5.97
2	230	139	50	4.96
3	230	168	61	5.60
4	260	146	61	3.96
5	260	172	61	5.08
6	230	150	61	4.60
7	341	150	43	4.09
8	359	164	43	4.80
9	291	190	43	5.89
10	260	190	65	5.89
11	223	212	60	6.65
12	285	197	60	5.87
13	341	128	60	3.15

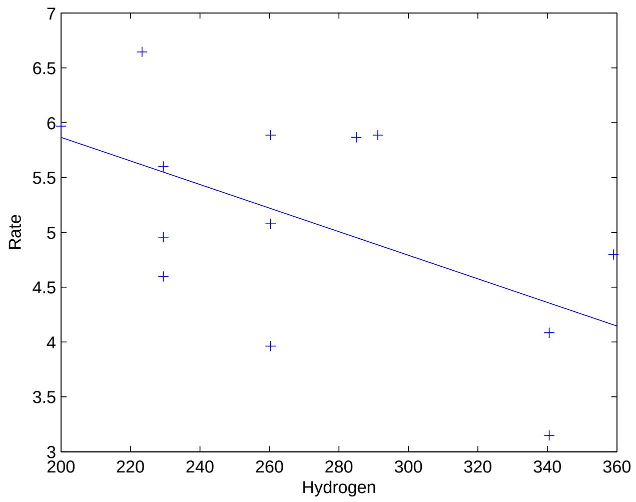
Note The results are determined using an underlying model that takes into account the noise in the process, so even if you keep all of the levels the same, the results will vary from run to run. In this case however, the **Analyze** function will not be able to generate a fit for the results.

---

3 When you have completed 13 runs, select `Hydrogen vs.  Rate`, in the field next to **Analyze**, to plot the relationships between the reactants and the reaction rate.



For this set of 13 runs, `rsmdemo` produces the following plot.



4 Click the **Analyze** button to call the `rstool` function, which you can then use to try to optimize the results. See "Exploring Graphs of Multidimensional Polynomials" on page 7-13 for more information about using the `rstool` demo.

5 Next, perform another set of 13 runs, this time from a designed experiment. In the Experimental Data window, click the **Do Experiment** button. `rsmdemo` calls the `cordexch` function to generate a D-optimal design, and then, for each run, computes the reaction rate.

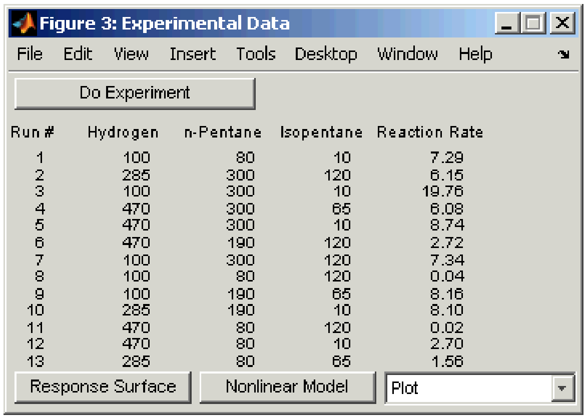
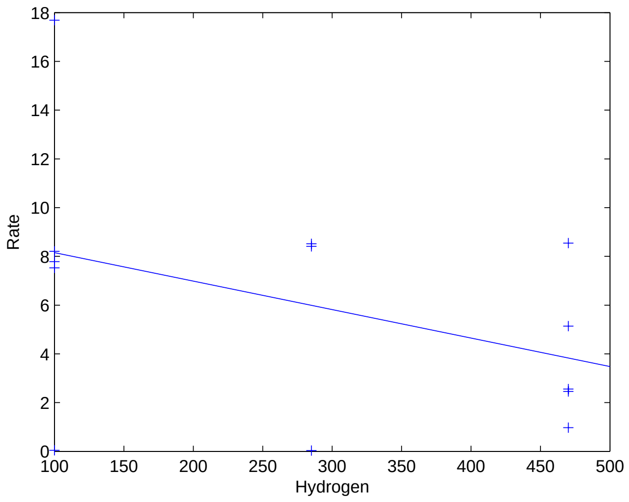


Figure 3: Experimental Data
File Edit View Insert Tools Desktop Window Help
Do Experiment
Response Surface
Nonlinear Model
Plot

Run #	Hydrogen	n-Pentane	Isopentane	Reaction Rate
1	100	80	10	7.29
2	285	300	120	6.15
3	100	300	10	19.76
4	470	300	65	6.08
5	470	300	10	8.74
6	470	190	120	2.72
7	100	300	120	7.34
8	100	80	120	0.04
9	100	190	65	8.16
10	285	190	10	8.10
11	470	80	120	0.02
12	470	80	10	2.70
13	285	80	65	1.56

6 Select `Hydrogen vs. Rate` in the field next to Nonlinear Model in the Experimental Data window. This displays the following plot.



7 You can also click the **Response Surface** button to call `rstool` to find the optimal levels of the reactants.

8 Compare the analysis results for the two sets of data. It is likely (though not certain) that you'll find some or all of these differences:

- You can fit a full quadratic model with the data from the designed experiment, but the trial and error data may be insufficient for fitting a quadratic model or interactions model.
- Using the data from the designed experiment, you are more likely to be able to find levels for the reactants that result in the maximum reaction rate. Even if you find the best settings using the trial and error data, the confidence bounds are likely to be wider than those from the designed experiment.

Comparing Results Using a Polynomial Model and a Nonlinear Model

This part of the experiment analyzes the experimental design data with a polynomial (response surface) model and a nonlinear model, and compare the results. The true model for the process, which is used to generate the data, is actually a nonlinear model. However, within the range of the data, a quadratic model approximates the true model quite well:

1 Using the results generated in the designed experiment part of "Comparing Results from Trial-and-Error Data and a Designed Experiment" on page 11-12, click the **Response Surface** button on the Experimental Data window. `rsmdemo` calls `rstool`, which fits a full quadratic model to the data. Drag the reference lines to change the levels of the reactants, and find the optimal reaction rate. Observe the width of the confidence intervals.

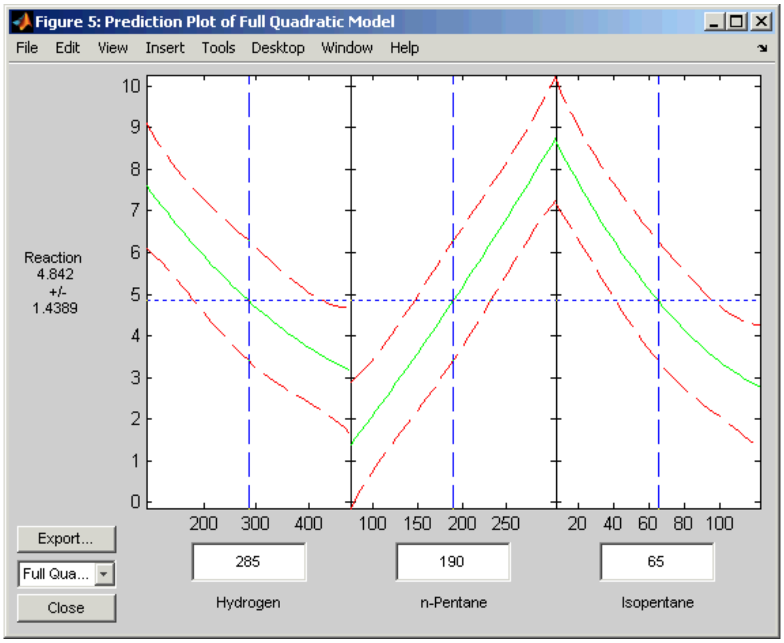
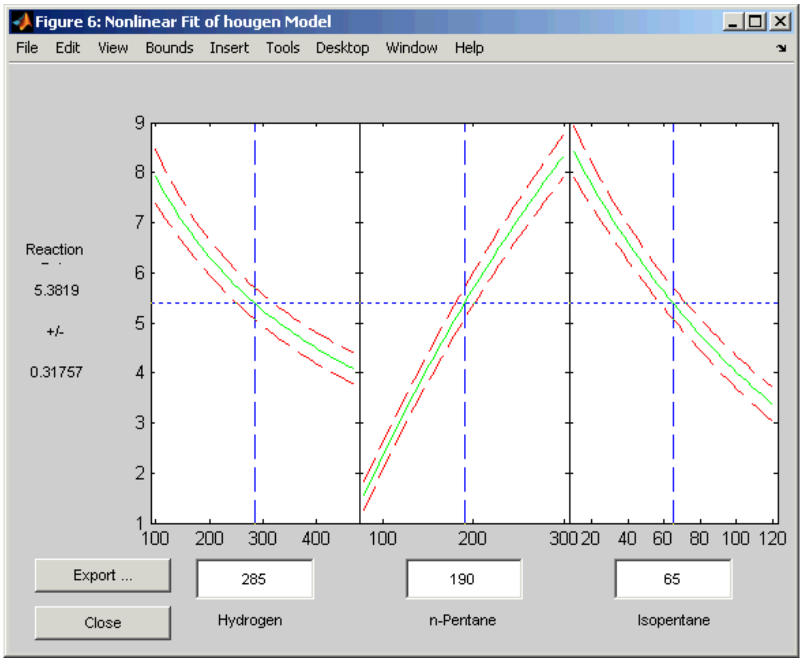


Figure 5: Prediction Plot of Full Quadratic Model
File Edit View Insert Tools Desktop Window Help
Reaction
4.842
+/-
1.4389
Export...
Full Qua...
Close
285
190
65
Hydrogen
n-Pentane
Isopentane

2 Now click the **Nonlinear Model** button on the Experimental Data window. `rsmdemo` calls `nlintool`, which fits a Hougen-Watson model to the data. As with the quadratic model, you can drag the reference lines to change the reactant levels. Observe the reaction rate and the confidence intervals.



3 Compare the analysis results for the two models. Even though the true model is nonlinear, you may find that the polynomial model provides a good fit. Because polynomial models are much easier to fit and work with than nonlinear models, a polynomial model is often preferable even when modeling a nonlinear process. Keep in mind, however, that such models are unlikely to be reliable for extrapolating outside the range of the data.

D-Optimal Designs

The designs above pre-date the computer age, and some were in use by early in the 20th century. In the 1970s statisticians started to use the computer in experimental design by recasting the design of experiments (DOE) in terms of optimization. A D-optimal design is one that maximizes the determinant of Fisher's information matrix, X^TX. This matrix is proportional to the inverse of the covariance matrix of the parameters. So maximizing $det(X^TX)$ is equivalent to minimizing the determinant of the covariance of the parameters.

A D-optimal design minimizes the volume of the confidence ellipsoid of the regression estimates of the linear model parameters, β.

There are several functions in Statistics Toolbox that generate D-optimal designs. These are `cordexch`, `daugment`, `dcovary`, and `rowexch`. The following sections explore D-optimal design in greater detail:

- "Generating D-Optimal Designs" on page 11-19
- "Augmenting D-Optimal Designs" on page 11-22
- "Designing Experiments with Uncontrolled Inputs" on page 11-24
- "Controlling Candidate Points" on page 11-25
- "Including Categorical Factors" on page 11-26

Generating D-Optimal Designs

The `cordexch` and `rowexch` functions provide two competing optimization algorithms for computing a D-optimal design given a model specification.

Both `cordexch` and `rowexch` are iterative algorithms. They operate by improving a starting design by making incremental changes to its elements. In the coordinate exchange algorithm, the increments are the individual elements of the design matrix. In row exchange, the elements are the rows of the design matrix. Atkinson and Donev [1] is a reference. The row exchange algorithm uses a candidate set of all possible design points, so this can require much more memory than the coordinate exchange algorithm. However, since the row exchange algorithm can change the coordinates of multiple factors in a single exchange, it can sometimes find better designs.

In both functions, there is randomness built into the selection of the starting design and into the choice of incremental changes. You can use this to your advantage by running the algorithm multiple times and selecting the best result as your final design. Each function has a 'tries' argument that can automate this search for you.

To generate a D-optimal design you must specify the number of inputs, the number of runs, and the order of the model you want to fit.

Both cordexch and rowexch take the following strings to specify the model:

- 'linear' or 'l' — The default model with constant and first order terms
- 'interaction' or 'i' — Includes constant, linear, and cross product terms
- 'quadratic' or 'q' — Interactions plus squared terms
- 'purequadratic' or 'p' — Includes constant, linear, and squared terms

Alternatively, you can use a matrix of integers to specify the terms. Details are in the help for the utility function x2fx.

For a simple example using the coordinate-exchange algorithm, consider the problem of quadratic modeling with two inputs. The model form is

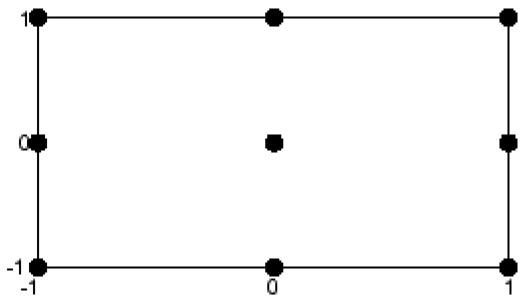
$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_{12} x_1 x_2 + \beta_{11} x_1^2 + \beta_{22} x_2^2 + \varepsilon$$

Suppose you want the D-optimal design for fitting this model with nine runs.

```
settings = cordexch(2,9,'q')
settings =
    -1     1
     1     1
     0     1
     1    -1
    -1    -1
     0    -1
     1     0
     0     0
    -1     0
```

You can plot the columns of settings against each other to get a better picture of the design.

```
h = plot(settings(:,1),settings(:,2),'.');
set(gca,'Xtick',[-1 0 1])
set(gca,'Ytick',[-1 0 1])
set(h,'Markersize',20)
```



For a simple example using the row-exchange algorithm, consider the interaction model with two inputs. The model form is

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_{12} x_1 x_2 + \varepsilon$$

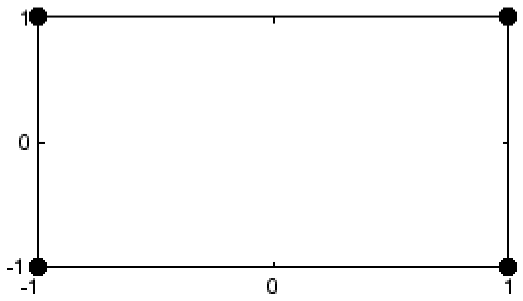
Suppose you want the D-optimal design for fitting this model with four runs.

```
[settings, X] = rowexch(2,4,'i')
settings =
    -1     1
    -1    -1
     1    -1
     1     1
X =
     1    -1     1    -1
     1    -1    -1     1
     1     1    -1    -1
     1     1     1     1
```

The settings matrix shows how to vary the inputs from run to run. The X matrix is the design matrix for fitting the above regression model. The first column of X is for fitting the constant term. The last column is the element-wise product of the second and third columns.

The associated plot is simple but elegant.

```
h = plot(settings(:,1),settings(:,2),'.');
set(gca,'Xtick',[-1 0 1])
set(gca,'Ytick',[-1 0 1])
set(h,'Markersize',20)
```



Augmenting D-Optimal Designs

In practice, experimentation is an iterative process. You often want to add runs to a completed experiment to learn more about the system. The function daugment allows you choose these extra runs optimally.

Suppose you execute the eight-run design below for fitting a linear model to four input variables.

```
settings = cordexch(4,8)
settings =
     1    -1     1     1
    -1    -1     1    -1
    -1     1     1     1
     1     1     1    -1
    -1     1    -1     1
     1    -1    -1     1
    -1    -1    -1    -1
     1     1    -1    -1
```

This design is adequate to fit the linear model for four inputs, but cannot fit the six cross-product (interaction) terms. Suppose you are willing to do eight more runs to fit these extra terms. The following code show how to do so.

```
[augmented, X] = daugment(settings,8,'i');

augmented
augmented =

     1    -1     1     1
    -1    -1     1    -1
    -1     1     1     1
     1     1     1    -1
    -1     1    -1     1
     1    -1    -1     1
    -1    -1    -1    -1
     1     1    -1    -1
    -1    -1    -1     1
     1     1     1     1
    -1    -1     1     1
    -1     1     1    -1
     1    -1     1    -1
     1    -1    -1    -1
    -1     1    -1    -1
     1     1    -1     1

info = X'*X
info =

    16     0     0     0     0     0     0     0     0     0     0
     0    16     0     0     0     0     0     0     0     0     0
     0     0    16     0     0     0     0     0     0     0     0
     0     0     0    16     0     0     0     0     0     0     0
     0     0     0     0    16     0     0     0     0     0     0
     0     0     0     0     0    16     0     0     0     0     0
     0     0     0     0     0     0    16     0     0     0     0
     0     0     0     0     0     0     0    16     0     0     0
     0     0     0     0     0     0     0     0    16     0     0
     0     0     0     0     0     0     0     0     0    16     0
     0     0     0     0     0     0     0     0     0     0    16
```

The augmented design is orthogonal, since X'*X is a multiple of the identity matrix. In fact, this design is the same as a 2^4 factorial design.

Designing Experiments with Uncontrolled Inputs

Sometimes it is impossible to control every experimental input. But you might know the values of some inputs in advance. An example is the time each run takes place. If a process is experiencing linear drift, you might want to include the time of each test run as a variable in the model.

The function dcovary enables you to choose the settings for each run in order to maximize your information despite a linear drift in the process.

Suppose you want to execute an eight-run experiment with three factors that is optimal with respect to a linear drift in the response over time. First you create the drift input variable. Note that drift is normalized to have mean zero. Its minimum is -1 and its maximum is 1.

```
drift = (linspace(-1,1,8))'
drift =

   -1.0000
   -0.7143
   -0.4286
   -0.1429
    0.1429
    0.4286
    0.7143
    1.0000

settings = dcovary(3,drift,'linear')
settings =

    1.0000    1.0000   -1.0000   -1.0000
   -1.0000   -1.0000   -1.0000   -0.7143
   -1.0000    1.0000    1.0000   -0.4286
    1.0000   -1.0000    1.0000   -0.1429
   -1.0000    1.0000   -1.0000    0.1429
    1.0000    1.0000    1.0000    0.4286
   -1.0000   -1.0000    1.0000    0.7143
```

```
    1.0000   -1.0000   -1.0000    1.0000
```

Controlling Candidate Points

The `rowexch` function generates a candidate set of possible design points, and then uses a D-optimal algorithm to select a design from those points. It does this by invoking the `candgen` and `candexch` functions. If you need to supply your own candidate set, or if you need to modify the one that the `candgen` function provides, you might prefer to call these functions separately.

This example creates a design that represents proportions of a mixture, so the sum of the proportions cannot exceed 1. A third factor is a filler factor that is not used here but that makes up the remaining proportion of the mixture.

```
% Generate a matrix of (x,y) values with x+y<=1
[x,y]=meshgrid(0:.1:1);
xy = [x(:) y(:)];
xy = xy(sum(xy,2)<=1,:);

% Compute quadratic model terms for these points.
f = x2fx(xy,'q');

% Generate a 10-point design and display it
r=candexch(f,10);
xy(r,:)
ans =
         0         0
         0    1.0000
    1.0000         0
         0    0.5000
    0.5000         0
         0    1.0000
    1.0000         0
    0.5000    0.5000
    0.5000         0
    0.5000    0.5000
```

However, both the `cordexch` and `rowexch` functions have a number of options that give you some measure of control over the points that will be considered for inclusion in the design. The following generates a design using `rowexch` with the same constraints as above:

- factors bounded between 0 and 1,
- 11 possible values for each factor,
- and factor sums must not exceed 1.

```
bnds = [0 0;1 1];
nlev = [11 11];
fn = @(x) sum(x,2)>1;
rowexch(2,10,'q','bounds',bnds,'levels',nlev,'excludefun',fn)
ans =
    0.5000         0
         0    1.0000
         0    0.5000
         0         0
    1.0000         0
    1.0000         0
         0    0.5000
    0.5000    0.5000
         0         0
    0.5000         0
```

Including Categorical Factors

So far the designs in this section have been for continuous factors. It is also possible to produce designs for categorical factors. The following produces a design for 3 factors and 9 runs, where all three are categorical and take 3 levels each:

```
x = rowexch(3,9,'l','categ',1:3,'levels',[3 3 3]);
sortrows(x)
ans =
     1     1     2
     1     2     1
     1     3     3
     2     1     1
     2     2     3
     2     3     2
     3     1     3
     3     2     2
     3     3     1
```

The resulting design has the nice property that for each pair of factors, each of the 9 possible combinations of levels appears exactly once. This property is not shared by all D-optimal designs, but it happens to be the optimal design for this problem. Also, because of some randomness built into the `rowexch` function, repeated runs of this example might give different designs.

To understand how this design is created behind the scenes, you may find it instructive to call the `candexch` function directly using a candidate set that you set up. First create a matrix `F` containing all 27 combinations of the factor levels. Create a matrix `C` containing the dummy variables for these categorical factors. Make `C` have full rank by removing one column for each factor except the first. Finally, use the `candexch` function to generate a 9-run design.

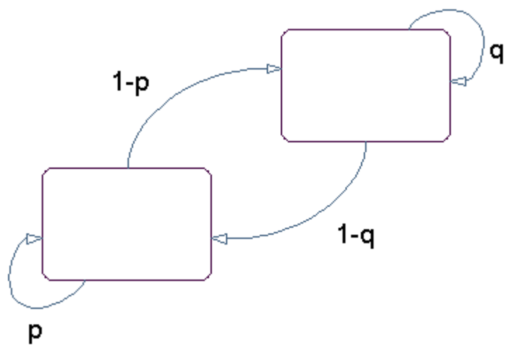
```
F = fullfact([3 3 3]);
C = dummyvar(F);
C(:,[4 7]) = [];
rows = candexch(C,9);
sortrows(F(rows,:))
ans =
     1     1     3
     1     2     1
     1     3     2
     2     1     2
     2     2     3
     2     3     1
     3     1     1
     3     2     2
     3     3     3
```

Hidden Markov Models

Introduction

Markov models are mathematical models of stochastic processes—processes that generate random sequences of outcomes according to certain probabilities. A simple example of a stochastic process is a sequence of coin tosses, the outcomes being heads or tails. People use Markov models to analyze a wide variety of stochastic processes, from daily stock prices to the positions of genes in a chromosome.

You can construct Markov models very easily using *state diagrams*, such as the one shown in this figure.



A State Diagram for a Markov Model

The rectangles in the diagram represent the possible states of the process you are trying to model, and the arrows represent transitions between states. The label on each arrow represents the probability of that transition, which depends on the process you are modeling. At each step of the process, the model generates an output, or *emission*, depending on which state it is in, and then makes a transition to another state.

For example, if you are modeling a sequence of coin tosses, the two states are heads and tails. The most recent coin toss determines the current state of the model and each subsequent toss determines the transition to the next state. If the coin is fair, the transition probabilities are all 1/2. In this simple example, the emission at any moment in time is simply the current state. However, in more complicated models, the states themselves can contain random processes that affect their emissions. For example, after each flip of the coin, you could roll a die to determine the emission at that step.

A *hidden Markov model* is one in which you observe a sequence of emissions, but you do not know the sequence of states the model went through to generate the emissions. In this case, your goal is to recover the state information from the observed data. The next section, “Example: States and Emissions” on page 12-4, provides an example.

Statistics Toolbox includes five functions for analyzing hidden Markov models:

- `hmmdecode` — Calculates the posterior state probabilities of a sequence
- `hmmgenerate` — Generates a sequence for a hidden Markov model
- `hmmestimate` — Estimates the parameters for a Markov model
- `hmmtrain` — Calculates the maximum likelihood estimate of hidden Markov model parameters
- `hmmviterbi` — Calculates the most likely state path for a hidden Markov model sequence

“Analyzing a Hidden Markov Model” on page 12-8 explains how to use these functions in detail.

Example: States and Emissions

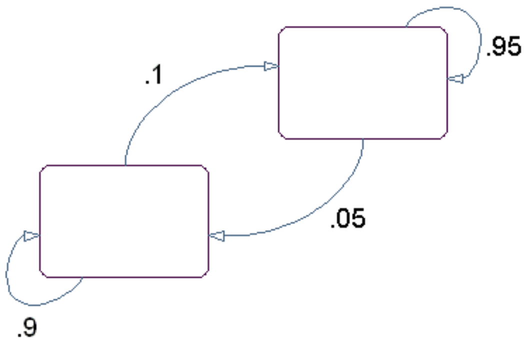
This section describes a simple example of a Markov model in which there are two states and six possible emissions. The example uses the following objects:

- A red die, having six sides, labeled 1 through 6.
- A green die, having twelve sides, five of which are labeled 2 through 6, while the remaining seven sides are labeled 1.
- A weighted red coin, for which the probability of heads is .9 and the probability of tails is .1.
- A weighted green coin, for which the probability of heads is .95 and the probability of tails is .05.

You create a sequence of numbers from the set {1, 2, 3, 4, 5, 6} using the following rules:

- Begin by rolling the red die and writing down the number that comes up, which is the emission.
- Toss the red coin and do one of the following:
  - If the result is heads, roll the red die and write down the result.
  - If the result is tails, roll the green die and write down the result.
- At each subsequent step, you flip the coin that has the same color as the die you rolled in the previous step. If the coin comes up heads, roll the same die as in the previous step. If the coin comes up tails, switch to the other die.

You can model this example with a state diagram that has two states, red and green, as shown in the following figure.



You determine the emission from a state by rolling the die with the same color as the state, and the transition to the next state by flipping the coin with the same color as the state.

So far, the model is not hidden, because you know the sequence of states from the colors of the coins and dice. But, suppose that someone else is generating the emissions without showing you the dice or coins. All you can see is the sequence of numbers. If you start seeing more 1s than other numbers, you might suspect that the model is in the green state, but you cannot be sure because you cannot see the color of the die being rolled. This is an example of a hidden Markov model: you can observe the sequence of emissions, but you do not know what state the model is in—that is, what color die is being rolled—when the emission occurs.

Not knowing the state the model is in raises the following problems:

- Given a sequence, what is the most likely state path?
- How can you estimate the parameters of the model given the state path?
- How can you estimate the parameters of the model without knowing the state path?
- What is the probability that the model generates a given sequence? This is known as the *forward probability*.
- What is the probability that the model is in a particular state at any point in the sequence? This is the *posterior probability*.

Markov Chains

This section defines *Markov chains*, which are the mathematical descriptions of Markov models. A Markov chain contains the following elements:

- A set of states {1, 2, ..., M}
- An M-by-M *transition matrix* T whose i, j entry is the probability of a transition from state i to state j. The matrix corresponds to a state diagram like the one shown in the State Diagram for a Markov Model figure. The sum of the entries in each row of T must be 1, because this is the sum of the probabilities of making a transition from a given state to each of the other states.
- A set of possible outputs, or *emissions*, $\{s_1, s_2, \ldots, s_N\}$. By default, the set of emissions is {1, 2, ... , N}, where N is the number of possible emissions, but you can choose a different set of numbers or symbols.
- An M-by-N *emission matrix* E whose i,k entry gives the probability of emitting symbol s_k given that the model is in state i.

When the model is in state i_1, it emits an output s_{k_1} with probability $E_{i_1 k_1}$. The model then makes a transition to state i_2 with probability $T_{i_1 i_2}$, and emits another symbol.

You can represent the example in "Example: States and Emissions" on page 12-4 by a Markov chain with two states, red and green. You determine transitions between states by flipping the coins. The transition matrix is

$$T = \begin{bmatrix} 0.9 & 0.1 \\ 0.05 & 0.95 \end{bmatrix}$$

You determine emissions by rolling the dice. The emissions matrix is

$$E = \begin{bmatrix} \frac{1}{6} & \frac{1}{6} & \frac{1}{6} & \frac{1}{6} & \frac{1}{6} & \frac{1}{6} \\ \frac{7}{12} & \frac{1}{12} & \frac{1}{12} & \frac{1}{12} & \frac{1}{12} & \frac{1}{12} \end{bmatrix}$$

"Analyzing a Hidden Markov Model" on page 12-8 shows how to analyze this model using functions in Statistics Toolbox.

How the Toolbox Generates Random Sequences

The hidden Markov model functions in Statistics Toolbox generate random sequences using the transition matrix, T, and the emission matrix, E, as described in the preceding section. The functions always begin with the model in the *initial state*, $i_0 = 1$, at step 0. The model then makes a transition to state i_1 with probability T_{1i_1}, and emits an output s_{k_1} with probability $E_{i_1k_1}$. Consequently, the probability of observing the sequence of states $i_1i_2\ldots i_r$ and the sequence of emissions $s_{k_1}s_{k_2}\ldots s_{k_r}$ in the first r steps, is

$$T_{1i_1}E_{i_1k_1}T_{i_1i_2}E_{i_2k_2}\ldots T_{i_{r-1}i_r}E_{i_rk}$$

Note that if the function returns a generated sequence of states, the first state in the sequence is i_1: the initial state, i_0, is not included.

In this implementation, the initial state is 1 with probability 1, and all other states have probability 0 of being the initial state. At times, you might want to change the probabilities of the initial states. You can do so by adding a new artificial state 1 that has transitions to the other states with any probabilities you want, but that never occurs after step 0. See "Changing the Probabilities of the Initial States" on page 12-13 to learn how to do this.

Analyzing a Hidden Markov Model

This section explains how to use functions in Statistics Toolbox to analyze hidden Markov models. For illustration, the section uses the example described in "Example: States and Emissions" on page 12-4. The section shows how to recover information about the model, assuming that you do not know some of the model's parameters. The section covers the following topics:

- "Setting Up the Model and Generating Data" on page 12-8
- "Computing the Most Likely Sequence of States" on page 12-9
- "Estimating the Transition and Emission Matrices" on page 12-9
- "Calculating Posterior State Probabilities" on page 12-12
- "Changing the Probabilities of the Initial States" on page 12-13
- "Example: Changing the Initial Probabilities" on page 12-14
- "Reference" on page 12-17

Setting Up the Model and Generating Data

This section shows how to set up a hidden Markov model and use it to generate data. First, create the transition and emission matrices by entering the following commands.

```
TRANS = [.9 .1; .05 .95;];

EMIS = [1/6, 1/6, 1/6, 1/6, 1/6, 1/6;...
7/12, 1/12, 1/12, 1/12, 1/12, 1/12];
```

Next, generate a random sequence of emissions from the model, `seq`, of length `1000`, using the function `hmmgenerate`. You can also return the corresponding random sequence of states in the model as the second output, `states`.

```
[seq, states] = hmmgenerate(1000, TRANS, EMIS);
```

Note In generating the sequences `seq` and `states`, `hmmgenerate` begins with the model in state $i_0 = 1$ at step 0. The model then makes a transition to state i_1 at step 1, and returns i_1 as the first entry in `states`.

Computing the Most Likely Sequence of States

Suppose you know the transition and emission matrices, TRANS and EMIS. If you observe a sequence, seq, of emissions, how can you compute the most likely sequence of states that generated the sequence? The function hmmviterbi uses the Viterbi algorithm to compute the most likely sequence of states that the model would go through to generate the given sequence of emissions.

```
likelystates = hmmviterbi(seq, TRANS, EMIS);
```

likelystates is a sequence of the same length as seq.

To test the accuracy of hmmviterbi, you can compute the percentage of the time that the actual sequence states agrees with the sequence likelystates.

```
sum(states==likelystates)/1000

ans =

0.8200
```

This shows that the most likely sequence of states agrees with the actual sequence 82% of the time. Note that your results might differ if you run the same commands, because the sequence seq is random.

Note The states at the beginning of the sequence returned by hmmviterbi are less reliable because of the computational delay in the Viterbi algorithm.

Estimating the Transition and Emission Matrices

Suppose you do not know the transition and emission matrices in the model, and you observe a sequence of emissions, seq. There are two functions that you can use to estimate the matrices:

- hmmestimate
- hmmtrain

Using hmmestimate

To use hmmestimate, you also need to know the corresponding sequence of states that the model went through to generate seq. The following command takes the emission and state sequences, seq and states, and returns estimates of the transition and emission matrices, TRANS_EST and EMIS_EST.

```
[TRANS_EST, EMIS_EST] = hmmestimate(seq, states)

TRANS_EST =

0.8989    0.1011
0.0585    0.9415

EMIS_EST =

0.1721    0.1721    0.1749    0.1612    0.1803    0.1393
0.5836    0.0741    0.0804    0.0789    0.0726    0.1104
```

You can compare these outputs with the original transition and emission matrices, TRANS and EMIS, to see how well hmmestimate estimates them.

```
TRANS

TRANS =

0.9000    0.1000
0.0500    0.9500

EMIS

EMIS =

0.1667    0.1667    0.1667    0.1667    0.1667    0.1667
0.5833    0.0833    0.0833    0.0833    0.0833    0.0833
```

Using hmmtrain

If you do not know the sequence of states, but you have an initial guess as to the values of TRANS and EMIS, you can estimate the transition and emission matrices using the function hmmtrain. For example, suppose you have the following initial guesses for TRANS and EMIS.

```
TRANS_GUESS = [.85 .15; .1 .9];
EMIS_GUESS = [.17 .16 .17 .16 .17 .17;.6 .08 .08 .08 .08 08];
```

You can estimate TRANS and EMIS with the following command.

```
[TRANS_EST2, EMIS_EST2] = hmmtrain(seq, TRANS_GUESS, EMIS_GUESS)

TRANS_EST2 =

0.2286    0.7714
0.0032    0.9968

EMIS_EST2 =

0.1436    0.2348    0.1837    0.1963    0.2350    0.0066
0.4355    0.1089    0.1144    0.1082    0.1109    0.1220
```

hmmtrain uses an iterative algorithm that alters the matrices TRANS_GUESS and EMIS_GUESS so that at each step the adjusted matrices are more likely to generate the observed sequence, seq. The algorithm halts when the matrices in two successive iterations are within a small tolerance of each other. See the reference page for hmmtrain for more information about the tolerance.

If the algorithm fails to reach this tolerance within a maximum number of iterations, whose default value is 100, the algorithm halts. In this case, hmmtrain returns the last values of TRANS_EST and EMIS_EST and issues a warning that the tolerance was not reached.

If the algorithm fails to reach the desired tolerance, you can increase the default value of the maximum number of iterations with the command

```
hmmtrain(seq, TRANS_GUESS, EMIS_GUESS, 'maxiterations', maxiter)
```

where maxiter is the maximum number of steps the algorithm executes.

You can also change default value of the tolerance with the command

```
hmmtrain(seq, TRANS_GUESS, EMIS_GUESS, 'tolerance', tol)
```

where tol is the desired value of the tolerance. Increasing the value of tol makes the algorithm halt sooner, but the results are less accurate.

Note If the sequence `seq` is long, the `hmmtrain` algorithm might take a long time to run. If so, you might want to lower the maximum number of iterations temporarily at first to find out how much time the algorithm requires.

There are two factors that can make the output matrices of `hmmtrain` less reliable:

- The algorithm might converge to a local maximum that does not represent the true transition and emission matrices. If you suspect that this is the case, try different initial guesses for the matrices `TRANS_EST` and `EMIS_EST`.
- The sequence `seq` might be too short to properly train the matrices. If you suspect this is the case, try using a longer sequence for `seq`.

Calculating Posterior State Probabilities

The posterior state probabilities of an emission sequence `seq` are the conditional probabilities that the model is in a particular state when it generates a symbol in `seq`, given that `seq` is emitted. You can compute the posterior state probabilities with the following command:

```
PSTATES = hmmdecode(seq, TRANS, EMIS)
```

The output `PSTATES` is an M-by-L matrix, where M is the number of states and L is the length of `seq`. `PSTATES(i,j)` is the conditional probability that the model is in state `i` when it generates the `j`th symbol of `seq`, given that `seq` is emitted.

Note The function `hmmdecode` begins with the model in state 1 at step 0, prior to the first emission. `PSTATES(i,1)` is the probability that the model is in state i at the following step 1.

You can also return the logarithm of the probability of the sequence `seq` as the second output argument.

```
[PSTATES, logpseq] = hmmdecode(seq, TRANS, EMIS)
```

The actual probability of a sequence tends to 0 rapidly as the length of the sequence increases, so the probability of a sufficiently long sequence is less than the smallest positive number your computer can represent. Consequently, `hmmdecode` returns the logarithm of the probability instead.

For example, the following code returns the logarithm probability of the one-element sequence `[3]`.

```
[PSTATES, logpseq] = hmmdecode([3], TRANS, EMIS);
exp(logpseq)

ans =

0.1583
```

Note that you can compute this answer directly as

$$\sum_{j=1}^{6} T_{1j}E_{j3}$$

by the commands

```
TRANS(1,:)*EMIS(:,3)

ans =

0.1583
```

Changing the Probabilities of the Initial States

By default, the hidden Markov model functions begin with the model in state 1 at step 0. In other words, with probability 1, the initial state is 1, and all other states have probability 0 of being the initial state. See "How the Toolbox Generates Random Sequences" on page 12-7.

For some models, you might want to assign different probabilities to the initial states. For example, you might want to choose initial state probabilities from a probability vector p satisfying $pT = p$. This assignment makes the Markov chain time independent: the probability of observing a given output at a specified step of the sequence is independent of the step number. This section explains how to assign any vector of probabilities for the initial states in your model.

To assign a vector of probabilities, $p = [p_1, p_2, ..., p_M]$, to the initial states, do the following:

1. Create an M+1-by-M+1 augmented transition matrix, $\hat{T}$, that has the following form:

   $$\hat{T} = \begin{bmatrix} 0 & p \\ 0 & T \end{bmatrix}$$

   where T is the true transition matrix. The first column of $\hat{T}$ contains M+1 zeros.

2. Create an M+1-by-N augmented emission matrix, $\hat{E}$, that has the following form:

   $$\hat{T} = \begin{bmatrix} 0 \\ E \end{bmatrix}$$

If the transition and emission matrices are `TRANS` and `EMIS`, respectively, you can create the augmented matrices with the following commands:

```
TRANS_HAT = [0 p; zeros(size(TRANS,1),1) TRANS];

EMIS_HAT = [zeros(1,size(EMIS,2)); EMIS];
```

Example: Changing the Initial Probabilities

For example, suppose that you have the following transition and emission matrices.

```
TRANS = [.9 .1; .05 .95;];

EMIS = [1/6, 1/6, 1/6, 1/6, 1/6, 1/6;...
7/12, 1/12, 1/12, 1/12, 1/12, 1/12];
```

You want to assign the states initial probabilities that are given by a left eigenvector, `p`, for `TRANS`, corresponding to the maximum eigenvalue 1.

These initial probabilities make the Markov model time independent. The probability of observing a given emission is the same at each step of the output sequence.

To find the vector *p*, enter the following commands.

```
[V,D] = eigs(TRANS')

V =

-0.4472   -0.7071
-0.8944    0.7071

D =

1.0000         0
     0    0.8500
```

The first column of `V` is the right eigenvector for `TRANS'` corresponding to eigenvalue 1. So the transpose of this vector is a left eigenvector for `TRANS`. You can create this vector as follows.

```
p = V(:, 1)'

p =

-0.4472   -0.8944

p*TRANS

ans =

-0.4472   -0.8944
```

This is not yet a probability vector, so divide `p` by its sum.

```
p = p/sum(p)

p =

0.3333    0.6667
```

Next, create the augmented matrices TRANS_HAT and EMIS_HAT.

```
TRANS_HAT = [0 p; zeros(size(TRANS,1),1) TRANS]

TRANS_HAT =

0    0.3333    0.6667
0    0.9000    0.1000
0    0.0500    0.9500

EMIS_HAT = [zeros(1,size(EMIS,2)); EMIS]

EMIS_HAT =

     0         0         0         0         0         0
0.1667    0.1667    0.1667    0.1667    0.1667    0.1667
0.5833    0.0833    0.0833    0.0833    0.0833    0.0833
```

This assignment of probabilities makes the Markov model time independent. For example, you can calculate the probability of seeing symbol 3 at step 1 of an emission sequence using the function hmmdecode as follows.

```
[pStates, logp]=hmmdecode([3],TRANS_HAT,EMIS_HAT);

exp(logp)

ans =
0.1111
```

Note that the second output argument, logp, is the logarithm of the probability of the sequence [3].

On the other hand, the probability of seeing symbol 3 at step 2 is the sum of the probabilities of the sequences [1 3], [2 3]. [3 3], [4 3], [5 3], and [6 3].

```
sum = 0;
for n = 1:6
 [pStates, logp] = hmmdecode([n 3],TRANS_HAT,EMIS_HAT);
 sum = sum + exp(logp);
end;
sum
```

```
sum =
0.1111
```

Reference

To learn more about hidden Markov models and their applications, see the following reference.

Durbin, R., S. Eddy, A. Krogh, and G. Mitchison, *Biological Sequence Analysis*, Cambridge Univ. Press, 1998.

Functions — By Category

File I/O

`caseread`	Read case names from file
`casewrite`	Write case names to file
`tblread`	Read tabular data from file
`tblwrite`	Write tabular data to file
`tdfread`	Read file containing tab-delimited numeric and text values

Data Organization

Categorical Arrays

`addlevels`	Add levels to categorical array
`droplevels`	Drop levels from categorical array
`getlabels`	Access labels of levels in categorical array
`islevel`	Test for categorical array levels
`ismember`	Test for categorical array membership
`isundefined`	Test for undefined elements of categorical array
`levelcounts`	Element counts by level for categorical array
`mergelevels`	Merge levels of categorical array
`nominal`	Create nominal array
`ordinal`	Create ordinal array
`reorderlevels`	Reorder levels of categorical array
`setlabels`	Define labels of levels in categorical array
`sort`	Sort ordinal array
`sortrows (ordinal)`	Sort rows of ordinal array
`summary (categorical)`	Summary statistics for categorical array

Dataset Arrays

`dataset`	Create dataset array
`datasetfun`	Apply function to variables of dataset array
`get`	Access dataset array properties
`grpstats (dataset)`	Summary statistics by group for dataset arrays
`join`	Merge observations from two dataset arrays
`replacedata`	Convert array to dataset variables
`set`	Display and define dataset array properties
`sortrows (dataset)`	Sort rows of dataset array
`summary (dataset)`	Summary statistics for dataset array

Descriptive Statistics

`bootci`	Bootstrap confidence interval
`bootstrp`	Bootstrap statistics through resampling of data
`copulaparam`	Copula parameters as function of rank correlation
`corrcoef`	Correlation coefficients
`cov`	Covariance matrix
`crosstab`	Cross-tabulation of vectors
`geomean`	Geometric mean of sample
`grp2idx`	Create index vector from grouping variable
`grpstats`	Summary statistics by group
`grpstats (dataset)`	Summary statistics by group for dataset arrays
`harmmean`	Harmonic mean of sample
`iqr`	Interquartile range of sample
`jackknife`	Jackknife statistics
`kurtosis`	Sample kurtosis
`mad`	Mean or median absolute deviation of sample
`mean`	Mean values of vectors and matrices
`median`	Median values of vectors and matrices
`moment`	Central moment of all orders
`nanmax`	Maximum, ignoring `NaN`s
`nanmean`	Mean, ignoring `NaN`s
`nanmedian`	Median, ignoring `NaN`s
`nanmin`	Minimum, ignoring `NaN`s

`nanstd`	Standard deviation, ignoring `NaN`s
`nansum`	Sum, ignoring `NaN`s
`partialcorr`	Linear or rank partial correlation coefficients
`prctile`	Percentiles of sample
`range`	Sample range
`skewness`	Sample skewness
`std`	Standard deviation of sample
`summary (categorical)`	Summary statistics for categorical array
`summary (dataset)`	Summary statistics for dataset array
`tabulate`	Frequency table
`tiedrank`	Compute ranks of sample, adjusting for ties
`trimmean`	Mean of sample, excluding extreme values
`var`	Variance of sample
`zscore`	Standardized *z*-scores

Statistical Visualization

`addedvarplot`	Create added-variable plot for stepwise regression
`aoctool`	Interactive fitting of analysis of covariance models
`boxplot`	Box plot of data sample
`cdfplot`	Plot of empirical cumulative distribution function
`dfittool`	Interactive fitting of distributions to data
`disttool`	Interactive pdf and cdf plots
`ecdfhist`	Create histogram from output of `ecdf`
`errorbar`	Plot error bars along curve
`fsurfht`	Interactive contour plot
`gline`	Interactive line plot
`glmdemo`	Demo of generalized linear models
`gname`	Label plotted points with their case names or case number
`gplotmatrix`	Plot matrix of scatter plots, by group
`gscatter`	Scatter plot, by group
`interactionplot`	Interaction plot for grouped data
`lsline`	Plot least squares lines
`maineffectsplot`	Main effects plot for grouped data
`multivarichart`	Multivari chart for grouped data
`normplot`	Normal probability plot
`pareto`	Pareto chart
`polytool`	Interactive plot of fitted polynomials and prediction intervals

`probplot`	Probability plots
`qqplot`	Quantile-quantile plot of two samples
`randtool`	Interactive random number generation
`rcoplot`	Residual case order plot
`refcurve`	Add polynomial to current plot
`refline`	Add reference line to current axes
`robustdemo`	Interactive robust regression
`rsmdemo`	Demo of design of experiments and surface fitting
`scatterhist`	2-D scatter plot with marginal histograms
`surfht`	Interactive contour plot
`wblplot`	Weibull probability plot

Probability Distributions

Probability Density Functions (p. 13-9)	Probability density functions for supported distributions
Cumulative Distribution Functions (p. 13-11)	Cumulative distribution functions for supported distributions
Inverse Cumulative Distribution Functions (p. 13-13)	Inverse cumulative distribution functions for supported distributions
Distribution Statistics Functions (p. 13-14)	Statistics for supported distributions
Distribution Fitting Functions (p. 13-16)	Fitting supported distributions to data
Piecewise Distribution Fitting (p. 13-17)	Fitting piecewise distributions to data
Negative Log-Likelihood Functions (p. 13-18)	Negative log-likelihood functions for supported distributions
Random Number Generators (p. 13-19)	Random number generators for supported distributions

Probability Density Functions

`betapdf`	Beta probability density function
`binopdf`	Binomial probability density function
`chi2pdf`	Chi-square probability density function
`copulapdf`	Copula probability density function
`disttool`	Interactive pdf and cdf plots
`evpdf`	Extreme value probability density function
`exppdf`	Exponential probability density function

`fpdf`	*F* probability density function
`gampdf`	Gamma probability density function
`geopdf`	Geometric probability density function
`gevpdf`	Generalized extreme value probability density function
`gppdf`	Generalized Pareto probability density function
`hygepdf`	Hypergeometric probability density function
`lognpdf`	Lognormal probability density function
`mnpdf`	Multinomial probability density function
`mvnpdf`	Multivariate normal probability density function
`mvtpdf`	Multivariate *t* probability density function
`nbinpdf`	Negative binomial probability density function
`ncfpdf`	Noncentral *F* probability density function
`nctpdf`	Noncentral *t* probability density function
`ncx2pdf`	Noncentral chi-square probability density function
`normpdf`	Normal probability density function
`pdf`	Probability density function for specified distribution
`poisspdf`	Poisson probability density function
`raylpdf`	Rayleigh probability density function

`tpdf`	Student's *t* probability density function
`unidpdf`	Discrete uniform probability density function
`unifpdf`	Continuous uniform probability density function
`wblpdf`	Weibull probability density function

Cumulative Distribution Functions

`betacdf`	Beta cumulative distribution function
`binocdf`	Binomial cumulative distribution function
`cdf`	Cumulative distribution function for specified distribution
`chi2cdf`	Chi-square cumulative distribution function
`copulacdf`	Copula cumulative distribution function
`disttool`	Interactive pdf and cdf plots
`ecdf`	Empirical cumulative distribution function
`evcdf`	Extreme value cumulative distribution function
`expcdf`	Exponential cumulative distribution function
`fcdf`	*F* cumulative distribution function
`gamcdf`	Gamma cumulative distribution function
`geocdf`	Geometric cumulative distribution function

`gevcdf`	Generalized extreme value cumulative distribution function
`gpcdf`	Generalized Pareto cumulative distribution function
`hygecdf`	Hypergeometric cumulative distribution function
`logncdf`	Lognormal cumulative distribution function
`mvncdf`	Multivariate normal cumulative distribution function
`mvtcdf`	Multivariate t cumulative distribution function
`ncfcdf`	Noncentral F cumulative distribution function
`nctcdf`	Noncentral t cumulative distribution function
`ncx2cdf`	Noncentral chi-square cumulative distribution function
`normcdf`	Normal cumulative distribution function
`poisscdf`	Poisson cumulative distribution function
`raylcdf`	Rayleigh cumulative distribution function
`tcdf`	Student’s t cumulative distribution function
`unidcdf`	Discrete uniform cumulative distribution function
`unifcdf`	Continuous uniform cumulative distribution function
`wblcdf`	Weibull cumulative distribution function

Inverse Cumulative Distribution Functions

`betainv`	Inverse of beta cumulative distribution function
`binoinv`	Inverse of binomial cumulative distribution function
`chi2inv`	Inverse of chi-square cumulative distribution function
`evinv`	Inverse of extreme value cumulative distribution function
`expinv`	Inverse of exponential cumulative distribution function
`finv`	Inverse of F cumulative distribution function
`gaminv`	Inverse of gamma cumulative distribution function
`geoinv`	Inverse of geometric cumulative distribution function
`gevinv`	Inverse of generalized extreme value cumulative distribution function
`gpinv`	Inverse of generalized Pareto cumulative distribution function
`hygeinv`	Inverse of hypergeometric cumulative distribution function
`icdf`	Inverse cumulative distribution function for specified distribution
`logninv`	Inverse of lognormal cumulative distribution function
`nbininv`	Inverse of negative binomial cumulative distribution function
`ncfinv`	Inverse of noncentral F cumulative distribution function

`nctinv`	Inverse of noncentral *t* cumulative distribution
`ncx2inv`	Inverse of noncentral chi-square cumulative distribution function
`norminv`	Inverse of normal cumulative distribution function
`poissinv`	Inverse of Poisson cumulative distribution function
`raylinv`	Inverse of Rayleigh cumulative distribution function
`tinv`	Inverse of Student's *t* cumulative distribution function
`unidinv`	Inverse of discrete uniform cumulative distribution function
`unifinv`	Inverse of continuous uniform cumulative distribution function
`wblinv`	Inverse of Weibull cumulative distribution function

Distribution Statistics Functions

`betastat`	Mean and variance of beta distribution
`binostat`	Mean and variance of binomial distribution
`chi2stat`	Mean and variance of chi-square distribution
`copulastat`	Rank correlation for copula
`evstat`	Mean and variance of extreme value distribution
`expstat`	Mean and variance of exponential distribution

`fstat`	Mean and variance of *F* distribution
`gamstat`	Mean and variance of gamma distribution
`geostat`	Mean and variance of geometric distribution
`gevstat`	Mean and variance of generalized extreme value distribution
`gpstat`	Mean and variance of generalized Pareto distribution
`hygestat`	Mean and variance of hypergeometric distribution
`lognstat`	Mean and variance of lognormal distribution
`nbinstat`	Mean and variance of negative binomial distribution
`ncfstat`	Mean and variance of noncentral *F* distribution
`nctstat`	Mean and variance of noncentral *t* distribution
`ncx2stat`	Mean and variance of noncentral chi-square distribution
`normstat`	Mean and variance of normal distribution
`poisstat`	Mean and variance of Poisson distribution
`raylstat`	Mean and variance of Rayleigh distribution
`tstat`	Mean and variance of Student's *t* distribution

`unifstat`	Mean and variance of continuous uniform distribution
`wblstat`	Mean and variance of Weibull distribution

Distribution Fitting Functions

`betafit`	Parameter estimates and confidence intervals for beta distributed data
`binofit`	Parameter estimates and confidence intervals for binomial distributed data
`dfittool`	Interactive fitting of distributions to data
`evfit`	Parameter estimates and confidence intervals for extreme value distributed data
`expfit`	Parameter estimates and confidence intervals for exponentially distributed data
`gamfit`	Parameter estimates and confidence intervals for gamma distributed data
`gevfit`	Parameter estimates and confidence intervals for generalized extreme value distributed data
`gpfit`	Parameter estimates and confidence intervals for generalized Pareto distributed data
`lognfit`	Parameter estimates and confidence intervals for lognormally distributed data
`mle`	Maximum likelihood estimation

`mlecov`	Asymptotic covariance matrix of maximum likelihood estimators
`nbinfit`	Parameter estimates and confidence intervals for negative binomial distributed data
`normfit`	Parameter estimates and confidence intervals for normally distributed data
`poissfit`	Parameter estimates and confidence intervals for Poisson distributed data
`raylfit`	Parameter estimates and confidence intervals for Rayleigh distributed data
`unifit`	Parameter estimates for uniformly distributed data
`wblfit`	Parameter estimates and confidence intervals for Weibull distributed data

Piecewise Distribution Fitting

`boundary`	Boundary points of piecewise distribution segments
`cdf (piecewisedistribution)`	Cumulative distribution function for piecewise distribution
`icdf (piecewisedistribution)`	Inverse cumulative distribution function for piecewise distribution
`lowerparams`	Parameters of generalized Pareto distribution lower tail
`nsegments`	Number of segments of piecewise distribution
`paretotails`	Construct Pareto tails object

`pdf (piecewisedistribution)`	Probability density function for piecewise distribution
`random (piecewisedistribution)`	Random numbers from piecewise distribution
`segment`	Segment of piecewise distribution containing input values
`upperparams`	Parameters of generalized Pareto distribution upper tail

Negative Log-Likelihood Functions

`betalike`	Negative log-likelihood for beta distribution
`evlike`	Negative log-likelihood for extreme value distribution
`explike`	Negative log-likelihood for exponential distribution
`gamlike`	Negative log-likelihood for gamma distribution
`gevlike`	Negative log-likelihood for generalized extreme value distribution
`gplike`	Negative log-likelihood for generalized Pareto distribution
`lognlike`	Negative log-likelihood for lognormal distribution
`mvregresslike`	Negative log-likelihood for multivariate regression
`normlike`	Negative log-likelihood for normal distribution
`wbllike`	Negative log-likelihood for Weibull distribution

Random Number Generators

`betarnd`	Random numbers from beta distribution
`binornd`	Random numbers from binomial distribution
`chi2rnd`	Random numbers from chi-square distribution
`copularnd`	Random numbers from copula
`evrnd`	Random numbers from extreme value distribution
`exprnd`	Random numbers from exponential distribution
`frnd`	Random numbers from *F* distribution
`gamrnd`	Random numbers from gamma distribution
`geornd`	Random numbers from geometric distribution
`gevrnd`	Random numbers from generalized extreme value distribution
`gprnd`	Random numbers from generalized Pareto distribution
`hygernd`	Random numbers from hypergeometric distribution
`iwishrnd`	Random numbers from inverse Wishart distribution
`johnsrnd`	Random numbers from Johnson system of distributions
`lhsdesign`	Generate latin hypercube sample
`lhsnorm`	Generate latin hypercube sample with normal distribution

`lognrnd`	Random numbers from lognormal distribution
`mhsample`	Markov chain Metropolis-Hastings sampler
`mnrnd`	Random numbers from multinomial distribution
`mvnrnd`	Random numbers from multivariate normal distribution
`mvtrnd`	Random numbers from multivariate *t* distribution
`nbinrnd`	Random numbers from negative binomial distribution
`ncfrnd`	Random numbers from noncentral *F* distribution
`nctrnd`	Random numbers from noncentral *t* distribution
`ncx2rnd`	Random numbers from noncentral chi-square distribution
`normrnd`	Random numbers from normal distribution
`pearsrnd`	Random numbers from Pearson system of distributions
`poissrnd`	Random numbers from Poisson distribution
`randg`	Gamma distributed random numbers and arrays (unit scale)
`random`	Random numbers from specified distribution
`randsample`	Random sample, with or without replacement
`randtool`	Interactive random number generation

`raylrnd`	Random numbers from Rayleigh distribution
`slicesample`	Markov chain slice sampler
`trnd`	Random numbers from Student's *t* distribution
`unidrnd`	Random numbers from discrete uniform distribution
`unifrnd`	Random numbers from continuous uniform distribution
`wblrnd`	Random numbers from Weibull distribution
`wishrnd`	Random numbers from Wishart distribution

Hypothesis Tests

Statistics Tests

`ansaribradley`	Ansari-Bradley test
`dwtest`	Durbin-Watson test
`linhyptest`	Linear hypothesis test on parameter estimates
`ranksum`	Wilcoxon rank sum test
`runstest`	Runs test for randomness
`signrank`	One-sample or paired-sample Wilcoxon signed rank test
`signtest`	One-sample or paired-sample sign test
`ttest`	One-sample or paired-sample *t*-test
`ttest2`	Two-sample *t*-test
`vartest`	One-sample chi-square variance test
`vartest2`	Two-sample *F*-test for equal variances
`vartestn`	Bartlett multiple-sample test for equal variances
`ztest`	One-sample *z*-test

Distribution Tests

`chi2gof`	Chi-square goodness-of-fit test
`jbtest`	Jarque-Bera test

`kstest`	One-sample Kolmogorov-Smirnov test
`kstest2`	Two-sample Kolmogorov-Smirnov test
`lillietest`	Lilliefors test

Linear Models

Linear Regression

`dummyvar`	{0,1}-valued matrix of dummy variables
`glmfit`	Generalized linear model fit
`glmval`	Values and prediction intervals for generalized linear models
`invpred`	Inverse prediction for simple linear regression
`leverage`	Leverage values for regression
`mnrfit`	Multinomial logistic regression
`mnrval`	Values and prediction intervals for multinomial logistic regression
`mvregress`	Multivariate linear regression
`mvregresslike`	Negative log-likelihood for multivariate regression
`polyconf`	Polynomial confidence intervals
`polyfit`	Polynomial fitting
`polytool`	Interactive plot of fitted polynomials and prediction intervals
`polyval`	Polynomial values and prediction intervals
`rcoplot`	Residual case order plot
`regress`	Multiple linear regression
`regstats`	Regression diagnostics for linear models

`ridge`	Ridge regression
`robustfit`	Robust linear regression
`rstool`	Interactive multidimensional response surface modeling
`stepwise`	Interactive stepwise regression
`stepwisefit`	Stepwise regression
`x2fx`	Convert predictor matrix to design matrix

Analysis of Variance

`anova1`	One-way analysis of variance
`anova2`	Two-way analysis of variance
`anovan`	*N*-way analysis of variance
`aoctool`	Interactive fitting of analysis of covariance models
`friedman`	Friedman's nonparametric two-way analysis of variance
`kruskalwallis`	Kruskal-Wallis nonparametric one-way analysis of variance
`manova1`	One-way multivariate analysis of variance
`manovacluster`	Dendrogram of group mean clusters following MANOVA
`multcompare`	Multiple comparison test

Nonlinear Models

Nonlinear Regression (p. 13-26)	Parametric models
Classification and Regression Trees (p. 13-26)	Nonparametric models

Nonlinear Regression

`coxphfit`	Cox proportional hazards regression
`nlinfit`	Nonlinear least-squares regression
`nlintool`	Interactive nonlinear fitting
`nlparci`	Confidence intervals for parameters in nonlinear regression
`nlpredci`	Confidence intervals for predictions in nonlinear regression

Classification and Regression Trees

`children`	Child nodes of tree node
`classcount`	Class counts at tree nodes
`classprob`	Class probabilities at tree nodes
`classregtree`	Construct classification and regression tree object
`cutcategories`	Categories for tree branches
`cutpoint`	Cutpoints for tree branches
`cuttype`	Cut types for tree branches
`cutvar`	Variable names for tree branches
`eval`	Predicted responses for tree
`isbranch`	Test tree node for branch
`nodeerr`	Node errors of tree

`nodeprob`	Node probabilities of tree
`nodesize`	Size of tree node
`numnodes`	Number of tree nodes
`parent`	Parent node of tree node
`prune`	Produce subtrees by pruning
`risk`	Node risks of tree
`test`	Error rate of tree
`treedisp`	Plot classification and regression trees
`treefit`	Fit tree-based model for classification or regression
`treeprune`	Produce sequence of subtrees by pruning
`treetest`	Compute error rate for tree
`treeval`	Compute fitted value for decision tree applied to data
`type`	Type of tree
`view`	View tree

Multivariate Statistics

Cluster Analysis (p. 13-28)	Cluster analysis
Dimension Reduction (p. 13-28)	Dimension reduction
Additional Multivariate Methods (p. 13-29)	Additional multivariate methods

Cluster Analysis

`cluster`	Construct clusters from `linkage` output
`clusterdata`	Construct clusters from data
`cophenet`	Cophenetic correlation coefficient
`dendrogram`	Plot dendrogram
`inconsistent`	Inconsistency coefficient of cluster tree
`kmeans`	K-means clustering
`linkage`	Create hierarchical cluster tree
`pdist`	Pairwise distance between observations
`silhouette`	Silhouette plot for clustered data
`squareform`	Reformat distance matrix

Dimension Reduction

`factoran`	Maximum likelihood common factor analysis
`pcacov`	Principal component analysis using covariance matrix

`pcares`	Residuals from principal component analysis
`princomp`	Principal component analysis

Additional Multivariate Methods

`barttest`	Bartlett's test for dimensionality
`canoncorr`	Canonical correlation analysis
`cholcov`	Cholesky-like decomposition for covariance matrix
`classify`	Discriminant analysis
`cmdscale`	Classical multidimensional scaling
`mahal`	Mahalanobis distance
`manova1`	One-way multivariate analysis of variance
`manovacluster`	Dendrogram of group mean clusters following MANOVA
`procrustes`	Procrustes analysis
`rstool`	Interactive multidimensional response surface modeling
`zscore`	Standardized z-scores

Statistical Process Control

`capability`	Process capability indices
`capaplot`	Process capability plot
`controlchart`	Shewhart control charts
`controlrules`	Western Electric and Nelson control rules
`gagerr`	Gage repeatability and reproducibility study
`histfit`	Histogram with superimposed normal density
`normspec`	Plot normal density between specification limits

Design of Experiments

`bbdesign`	Generate Box-Behnken design
`candexch`	D-optimal design from candidate set using row exchanges
`candgen`	Generate candidate set for D-optimal design
`ccdesign`	Generate central composite design
`cordexch`	D-optimal design of experiments coordinate exchange algorithm
`daugment`	D-optimal augmentation of experimental design
`dcovary`	D-optimal design with specified fixed covariates
`ff2n`	Two-level full-factorial designs
`fracfact`	Generate fractional factorial design from generators
`fracfactgen`	Fractional factorial design generators
`fullfact`	Full-factorial experimental design
`interactionplot`	Interaction plot for grouped data
`maineffectsplot`	Main effects plot for grouped data
`multivarichart`	Multivari chart for grouped data
`rowexch`	D-optimal design of experiments row exchange algorithm

Hidden Markov Models

`hmmdecode`	Posterior state probabilities of sequence
`hmmestimate`	Estimate parameters for hidden Markov model
`hmmgenerate`	Generate random sequences from Markov model
`hmmtrain`	Maximum likelihood estimate of model parameters for hidden Markov model
`hmmviterbi`	Most probable state path for hidden Markov model sequence

Nonparametric Methods

`friedman`	Friedman's nonparametric two-way analysis of variance
`kruskalwallis`	Kruskal-Wallis nonparametric one-way analysis of variance
`ksdensity`	Compute density estimate using kernel-smoothing method
`ranksum`	Wilcoxon rank sum test
`signrank`	One-sample or paired-sample Wilcoxon signed rank test
`signtest`	One-sample or paired-sample sign test

Graphical User Interfaces

`aoctool`	Interactive fitting of analysis of covariance models
`dfittool`	Interactive fitting of distributions to data
`disttool`	Interactive pdf and cdf plots
`glmdemo`	Demo of generalized linear models
`polytool`	Interactive plot of fitted polynomials and prediction intervals
`randtool`	Interactive random number generation
`regstats`	Regression diagnostics for linear models
`robustdemo`	Interactive robust regression
`rsmdemo`	Demo of design of experiments and surface fitting
`rstool`	Interactive multidimensional response surface modeling

Utility Functions

`statget`	Parameter values from statistics options structure
`statset`	Create or edit statistics options structure

Functions — Alphabetical List

addedvarplot

Purpose

Create added-variable plot for stepwise regression

Syntax

```
addedvarplot(X,y,num,inmodel)
addedvarplot(X,y,num,inmodel,stats)
```

Description

`addedvarplot(X,y,num,inmodel)` produces an added variable plot for the response `y` and the predictor in column `num` of `X`. The plot illustrates the incremental effect of a predictor in a regression model in which the columns specified by `inmodel` are used as predictors. `X` is an *n*-by-*p* matrix of *n* observations of *p* predictors. `y` is vector of *n* response values. `num` is a scalar index specifying the column of `X` to use in the plot. `inmodel` is a logical vector of *p* elements specifying the columns of `X` to use in the base model. By default, all elements of `inmodel` are `false`, which means that the model has no predictors. `addedvarplot` automatically includes a constant term in the model. Use the function `stepwisefit` to fit a regression model using stepwise regression and create the vector `inmodel`.

`addedvarplot(X,y,num,inmodel,stats)` uses the structure `stats`, which contains fitted model results created by the `stepwisefit` function. If you create the structure `stats` by calling `stepwisefit` prior to calling `addedvarplot`, you save computing time by including the `stats` argument in `addedvarplot`.

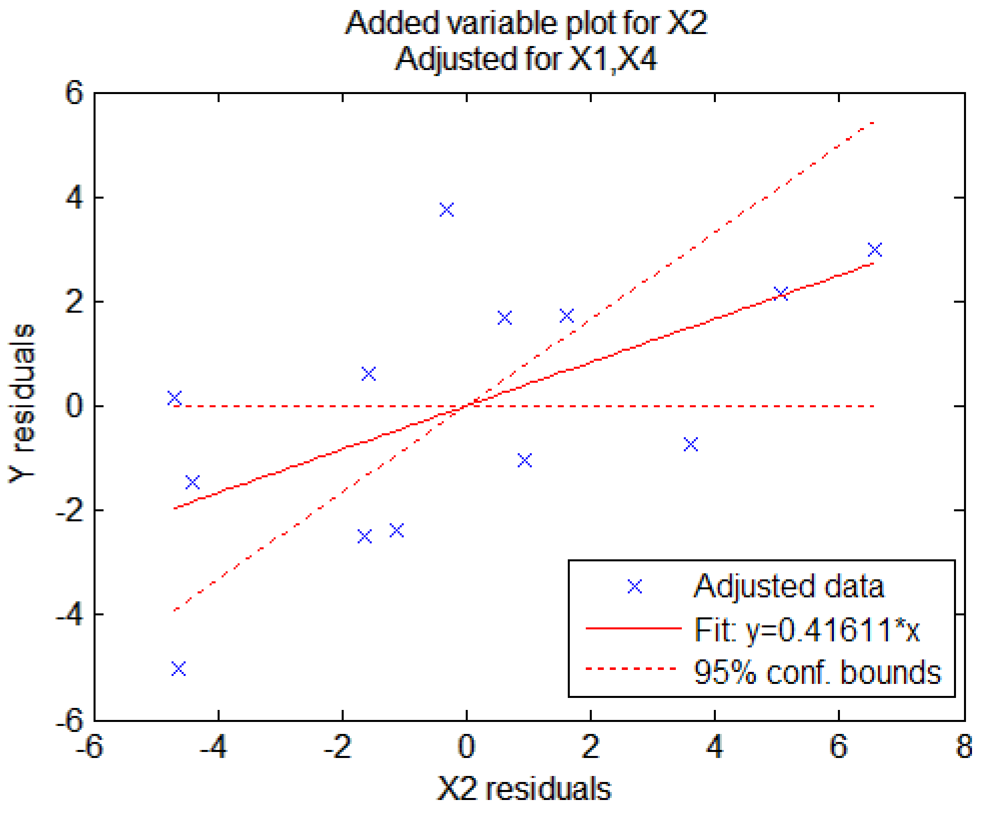
Added variable plots display data and fitted lines. If `X1` is column `num` of `X`, the data plotted are `y` versus `X1` after removing the effects of the other predictors specified by `inmodel`. The solid red line is a least squares fit of the data, and its slope is the coefficient that `X1` would have if it were included in the model. The dotted red lines are 95% confidence bounds for the fitted line, which are used to judge the significance of `X1`. If `inmodel(num)` is `true`, the plot is sometimes known as a *partial regression leverage plot*.

Example

Perform a stepwise regression on the data in `hald.mat`, and create an added-variable plot for the predictor in column 2:

```
load hald
```

```
[b,se,p,inmodel,stats] = stepwisefit(ingredients,heat);
Initial columns included:  none
Step 1, added column 4, p=0.000576232
Step 2, added column 1, p=1.10528e-006
Final columns included:  1 4
    'Coeff'      'Std.Err.'    'Status'    'P'
    [ 1.4400]    [  0.1384]    'In'        [1.1053e-006]
    [ 0.4161]    [  0.1856]    'Out'       [     0.0517]
    [-0.4100]    [  0.1992]    'Out'       [     0.0697]
    [-0.6140]    [  0.0486]    'In'        [1.8149e-007]

addedvarplot(ingredients,heat,2,inmodel,stats)
```



See Also `stepwisefit`

Purpose

Add levels to categorical array

Syntax

```
B = addlevels(A,newlevels)
```

Description

`B = addlevels(A,newlevels)` adds new levels to the categorical array `A`. `newlevels` is a cell array of strings or a two-dimensional character matrix that specifies the levels to be added. `addlevels` adds the new levels at the end of the list of possible categorical levels in `A`, but does not modify the value of any element. `B` does not contain elements at the new levels.

Examples

Example 1

Add levels for additional species in Fisher's iris data:

```
load fisheriris
species = nominal(species,...
                  {'Species1','Species2','Species3'},...
                  {'setosa','versicolor','virginica'});
species = addlevels(species,{'Species4','Species5'});
getlabels(species)
ans =
 'Species1' 'Species2' 'Species3' 'Species4' 'Species5'
```

Example 2

1 Load patient data from the CSV file `hospital.dat` and store the information in a dataset array with observation names given by the first column in the data (patient identification):

```
patients = dataset('file','hospital.dat',...
                   'delimiter',',',...
                   'ReadObsNames',true);
```

2 Make the {0,1}-valued variable `smoke` nominal, and change the labels to `'No'` and `'Yes'`:

```
patients.smoke = nominal(patients.smoke,{'No','Yes'});
```

3 Add new levels to `smoke` as placeholders for more detailed histories of smokers:

```
patients.smoke = addlevels(patients.smoke,...
                 {'0-5 Years','5-10 Years','LongTerm'});
```

4 Assuming the nonsmokers have never smoked, relabel the `'No'` level:

```
patients.smoke = setlabels(patients.smoke,'Never','No');
```

5 Drop the undifferentiated `'Yes'` level from `smoke`:

```
patients.smoke = droplevels(patients.smoke,'Yes');

Warning: OLDLEVELS contains categorical levels that
were present in A, caused some array elements to have
undefined levels.
```

Note that smokers now have an undefined level.

6 Set each smoker to one of the new levels, by observation name:

```
patients.smoke('YPL-320') = '5-10 Years';
```

See Also

`droplevels`, `islevel`, `mergelevels`, `reorderlevels`, `getlabels`

Purpose

Andrews plot for multivariate data

Syntax

```
andrewsplot(X)
andrewsplot(X,...,'Standardize','on')
andrewsplot(X,...,'Standardize','PCA')
andrewsplot(X,...,'Standardize','PCAStd')
andrewsplot(X,...,'Quantile',alpha)
andrewsplot(X,...,'Group',group)
andrewsplot(X,...,PropName,PropVal,...)
h = andrewsplot(X,...)
```

Description

`andrewsplot(X)` creates an Andrews plot of the multivariate data in the matrix `X`. The rows of `X` correspond to observations, the columns to variables. Andrews plots represent each observation by a function $f(t)$ of a continuous dummy variable t over the interval [0,1]. $f(t)$ is defined for the i th observation in `X` as

$$f(t) = X(i,1)/\sqrt{2} + X(i,2)\sin 2\pi t + X(i,2)\cos 2\pi t + \ldots$$

`andrewsplot` treats `NaN` values in `X` as missing values and ignores the corresponding rows.

`andrewsplot(X,...,'Standardize','on')` scales each column of `X` to have

mean `0` and standard deviation `1` before making the plot.

`andrewsplot(X,...,'Standardize','PCA')` creates an Andrews plot from the principal component scores of `X`, in order of decreasing eigenvalue. (See `princomp`.)

`andrewsplot(X,...,'Standardize','PCAStd')` creates an Andrews plot using the standardized principal component scores. (See `princomp`.)

`andrewsplot(X,...,'Quantile',alpha)` plots only the median and the `alpha` and (1 – `alpha`) quantiles of $f(t)$ at each value of t. This is useful if `X` contains many observations.

`andrewsplot(X,...,'Group',group)` plots the data in different groups with different colors. Groups are defined by `group`, a numeric array containing a group index for each observation. `group` can also be a categorical array, character matrix, or cell array of strings containing a group name for each observation. (See "Grouped Data" on page 2-41.)

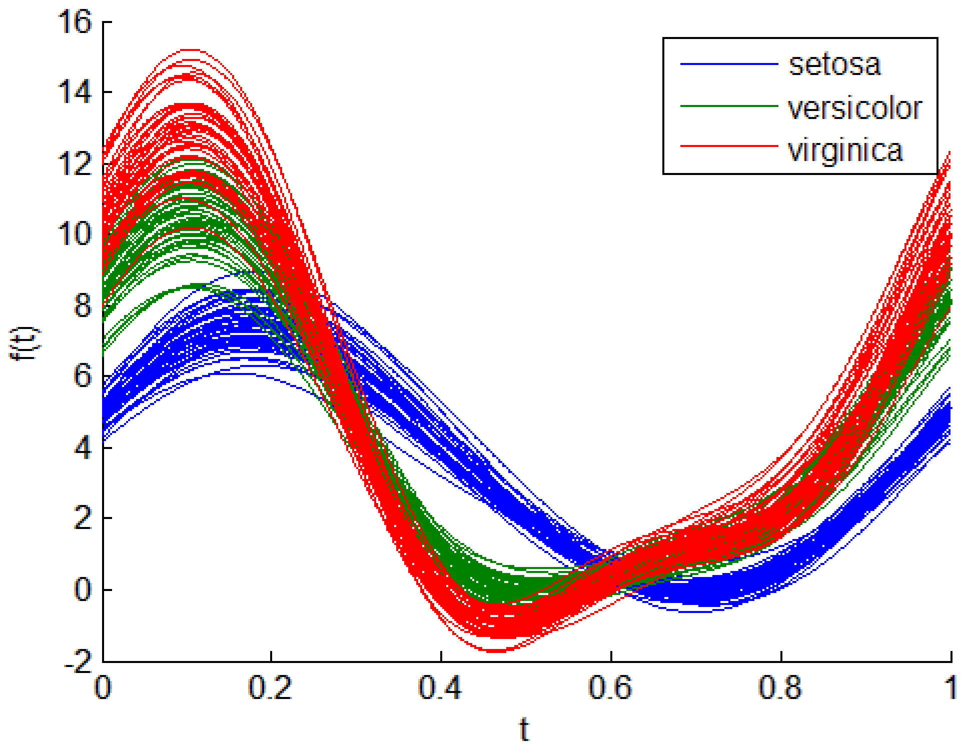
`andrewsplot(X,...,`*`PropName`*`,PropVal,...)` sets lineseries object properties to the specified values for all lineseries objects created by `andrewsplot`. (See Lineseries Properties.)

`h = andrewsplot(X,...)` returns a column vector of handles to the lineseries objects created by `andrewsplot`, one handle per row of `X`. If you use the `'Quantile'` input parameter, `h` contains one handle for each of the three lineseries objects created. If you use both the `'Quantile'` and the `'Group'` input parameters, `h` contains three handles for each group.

Examples

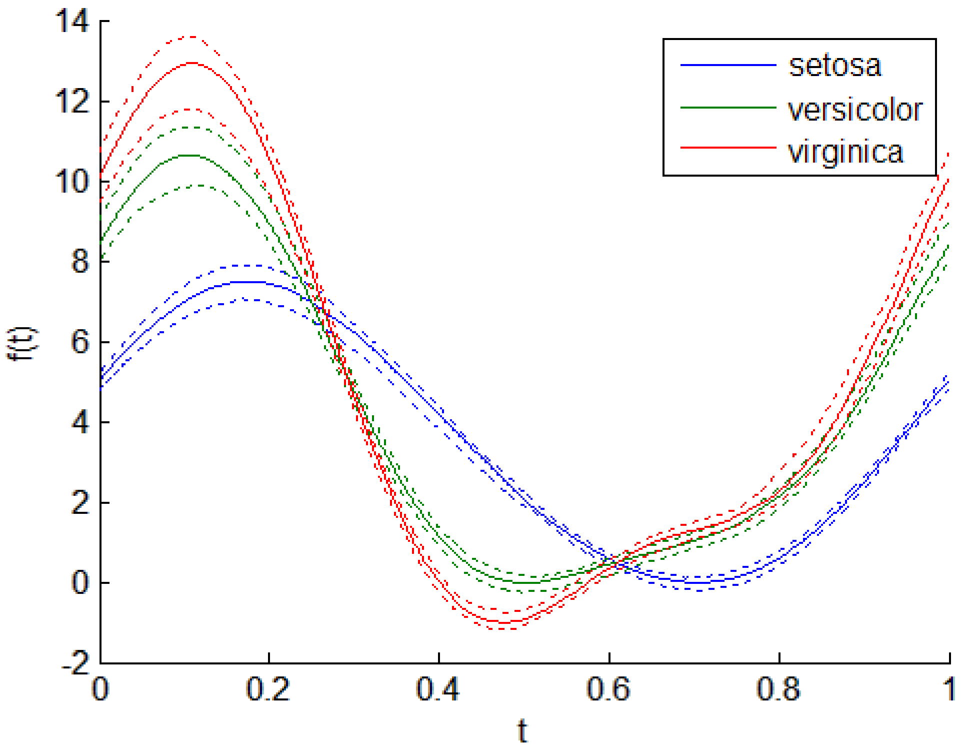
Make a grouped plot of the Fisher iris data:

```
load fisheriris
andrewsplot(meas,'group',species);
```



Plot only the median and quartiles of each group:

```
andrewsplot(meas,'group',species,'quantile',.25);
```



See Also `parallelcoords`, `glyphplot`

Purpose

One-way analysis of variance

Syntax

```
p = anova1(X)
p = anova1(X,group)
p = anova1(X,group,displayopt)
[p,table] = anova1(...)
[p,table,stats] = anova1(...)
```

Description

`p = anova1(X)` performs balanced one-way ANOVA for comparing the means of two or more columns of data in the matrix `X`, where each column represents an independent sample containing mutually independent observations. The function returns the *p*-value under the null hypothesis that all samples in `X` are drawn from populations with the same mean.

If `p` is near zero, it casts doubt on the null hypothesis and suggests that at least one sample mean is significantly different than the other sample means. Common significance levels are 0.05 or 0.01.

The `anova1` function displays two figures, the standard ANOVA table and a box plot of the columns of `X`.

The standard ANOVA table divides the variability of the data into two parts:

- Variability due to the differences among the column means (variability *between* groups)
- Variability due to the differences between the data in each column and the column mean (variability *within* groups)

The standard ANOVA table has six columns:

1. The source of the variability.
2. The sum of squares (`SS`) due to each source.
3. The degrees of freedom (`df`) associated with each source.

4. The mean squares (`MS`) for each source, which is the ratio `SS/df`.
5. The *F*-statistic, which is the ratio of the mean squares.
6. The *p*-value, which is derived from the cdf of *F*.

The box plot of the columns of `X` suggests the size of the *F*-statistic and the *p*-value. Large differences in the center lines of the boxes correspond to large values of *F* and correspondingly small values of *p*.

Columns of `X` with `NaN` values are disregarded.

`p = anova1(X,group)` performs ANOVA by group.

If X is a matrix, `anova1` treats each column as a separate group, and evaluates whether the population means of the columns are equal. This form of `anova1` is appropriate when each group has the same number of elements (balanced ANOVA). `group` can be a character array or a cell array of strings, with one row per column of `X`, containing group names. Enter an empty array (`[]`) or omit this argument if you do not want to specify group names.

If `X` is a vector, `group` must be a categorical variable, vector, string array, or cell array of strings with one name for each element of `X`. `X` values corresponding to the same value of `group` are placed in the same group. This form of `anova1` is appropriate when groups have different numbers of elements (unbalanced ANOVA).

If `group` contains empty or `NaN`-valued cells or strings, the corresponding observations in `X` are disregarded.

`p = anova1(X,group,`*`displayopt`*`)` enables the ANOVA table and box plot displays when *`displayopt`* is `'on'` (default) and suppresses the displays when *`displayopt`* is `'off'`. Notches in the boxplot provide a test of group medians (see `boxplot`) different from the *F* test for means in the ANOVA table.

`[p,table] = anova1(...)` returns the ANOVA table (including column and row labels) in the cell array `table`. Copy a text version of the ANOVA table to the clipboard using the `Copy Text` item on the **Edit** menu.

[p,table,stats] = anova1(...) returns a structure stats used to perform a follow-up multiple comparison test. anova1 evaluates the hypothesis that the samples all have the same mean against the alternative that the means are not all the same. Sometimes it is preferable to perform a test to determine which pairs of means are significantly different, and which are not. Use the multcompare function to perform such tests by supplying the stats structure as input.

Assumptions

The ANOVA test makes the following assumptions about the data in X:

- All sample populations are normally distributed.
- All sample populations have equal variance.
- All observations are mutually independent.

The ANOVA test is known to be robust with respect to modest violations of the first two assumptions.

Examples

Example 1

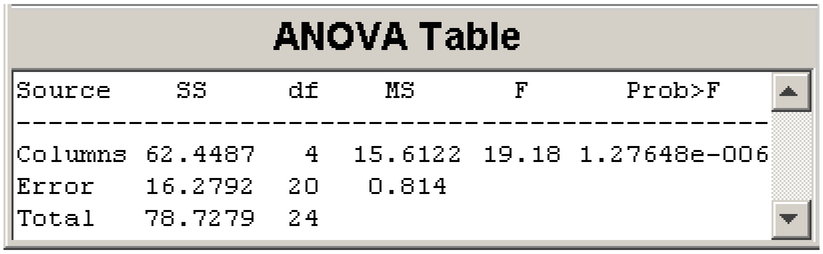
Create X with columns that are constants plus random normal disturbances with mean zero and standard deviation one:

```
X = meshgrid(1:5)
X =
   1   2   3   4   5
   1   2   3   4   5
   1   2   3   4   5
   1   2   3   4   5
   1   2   3   4   5

X = X + normrnd(0,1,5,5)
X =
   0.5674   3.1909   2.8133   4.1139   5.2944
  -0.6656   3.1892   3.7258   5.0668   3.6638
   1.1253   1.9624   2.4117   4.0593   5.7143
   1.2877   2.3273   5.1832   3.9044   6.6236
```

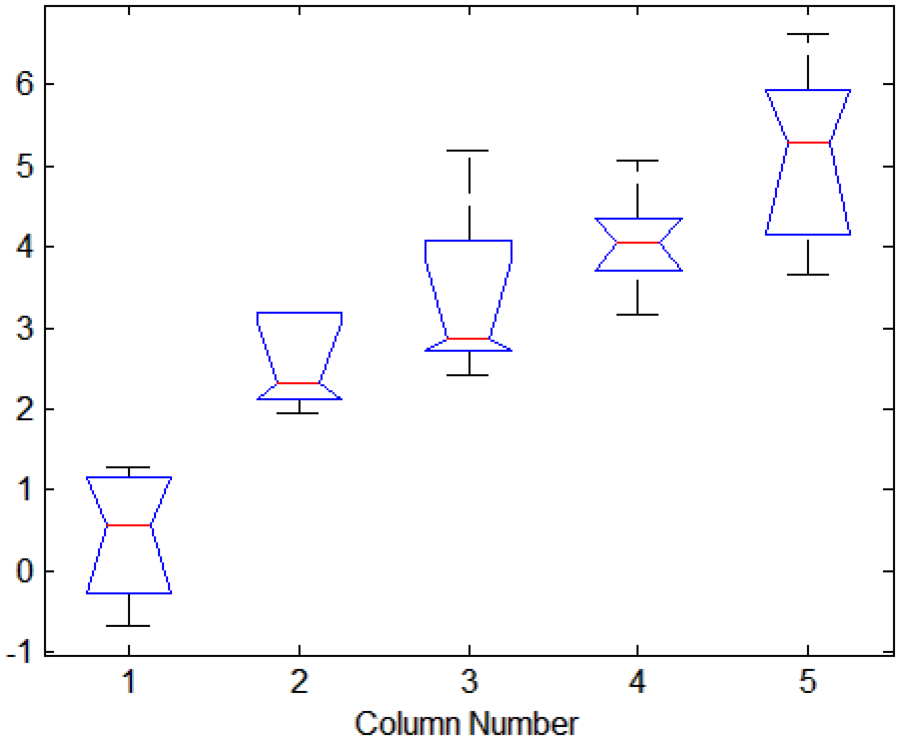
```
    -0.1465   2.1746   2.8636   3.1677   4.3082
  p = anova1(X)
  p =
   4.0889e-007
```

Perform one-way ANOVA:

```
  p = anova1(X)
  p =
    1.2765e-006
```


ANOVA Table

```
Source     SS      df     MS        F       Prob>F
------------------------------------------------------
Columns 62.4487    4   15.6122 19.18 1.27648e-006
Error   16.2792   20    0.814
Total   78.7279   24
```

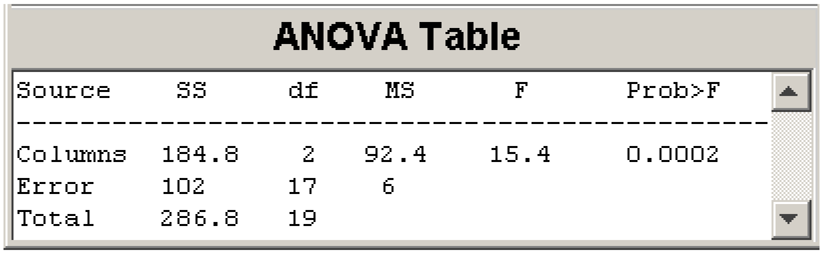


The very small *p*-value indicates that differences between column means are highly significant. The probability of this outcome under the null hypothesis (that samples drawn from the same population would have means differing by the amounts seen in `X`) is less than the *p*-value.

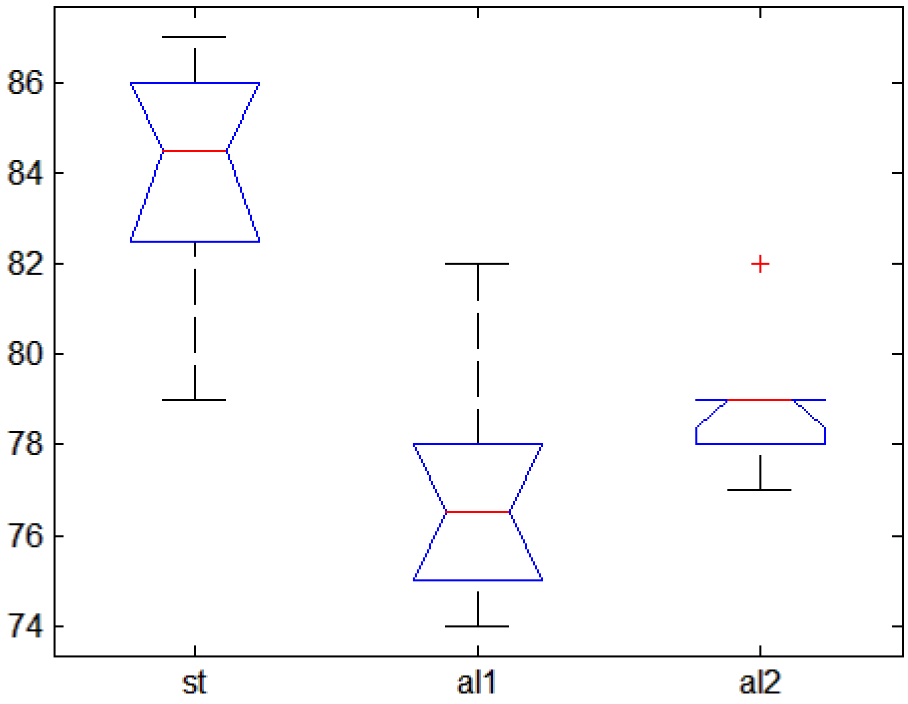
Example 2

The following example is from a study of the strength of structural beams in Hogg. The vector `strength` measures deflections of beams in thousandths of an inch under 3,000 pounds of force. The vector `alloy` identifies each beam as steel (`'st'`), alloy 1 (`'al1'`), or alloy 2 (`'al2'`). (Although `alloy` is sorted in this example, grouping variables do not need to be sorted.) The null hypothesis is that steel beams are equal in strength to beams made of the two more expensive alloys.

```
strength = [82 86 79 83 84 85 86 87 74 82 ...
            78 75 76 77 79 79 77 78 82 79];

alloy = {'st','st','st','st','st','st','st','st',...
         'al1','al1','al1','al1','al1','al1',...
         'al2','al2','al2','al2','al2','al2'};

p = anova1(strength,alloy)
p =
  1.5264e-004
```


ANOVA Table

```
Source     SS      df     MS       F        Prob>F
---------------------------------------------------
Columns   184.8     2    92.4     15.4      0.0002
Error     102      17     6
Total     286.8    19
```



The *p*-value suggests rejection of the null hypothesis. The box plot shows that steel beams deflect more than beams made of the more expensive alloys.

References

[1] Hogg, R. V., J. Ledolter, *Engineering Statistics*, MacMillan, 1987.

See Also

`anova2`, `anovan`, `boxplot`, `manova1`, `multcompare`

Purpose

Two-way analysis of variance

Syntax

```
p = anova2(X,reps)
p = anova2(X,reps,displayopt)
[p,table] = anova2(...)
[p,table,stats] = anova2(...)
```

Description

`p = anova2(X,reps)` performs a balanced two-way ANOVA for comparing the means of two or more columns and two or more rows of the observations in `X`. The data in different columns represent changes in factor *A*. The data in different rows represent changes in factor *B*. If there is more than one observation for each combination of factors, input `reps` indicates the number of replicates in each position, which much be constant. (For unbalanced designs, use `anovan`.)

The matrix below shows the format for a set-up where column factor A has two levels, row factor B has three levels, and there are two replications (`reps = 2`). The subscripts indicate row, column, and replicate, respectively.

$$\begin{array}{cc}
\begin{array}{cc} A=1 & A=2 \end{array} & \\
\left[\begin{array}{cc} x_{111} & x_{121} \\ x_{112} & x_{122} \\ x_{211} & x_{221} \\ x_{212} & x_{222} \\ x_{311} & x_{321} \\ x_{312} & x_{322} \end{array}\right] &
\begin{array}{l} \left.\begin{array}{c} \\ \\ \end{array}\right\} B=1 \\ \left.\begin{array}{c} \\ \\ \end{array}\right\} B=2 \\ \left.\begin{array}{c} \\ \\ \end{array}\right\} B=3 \end{array}
\end{array}$$

When `reps` is `1` (default), `anova2` returns two p-values in vector `p`:

1. The p-value for the null hypothesis, H_{0A}, that all samples from factor A (i.e., all column-samples in `X`) are drawn from the same population

2 The p-value for the null hypothesis, H_{0B}, that all samples from factor B (i.e., all row-samples in `X`) are drawn from the same population

When `reps` is greater than `1`, `anova2` returns a third p-value in vector `p`:

3 The p-value for the null hypothesis, H_{0AB}, that the effects due to factors A and B are *additive* (i.e., that there is no interaction between factors A and B)

If any p-value is near zero, this casts doubt on the associated null hypothesis. A sufficiently small p-value for H_{0A} suggests that at least one column-sample mean is significantly different that the other column-sample means; i.e., there is a main effect due to factor A. A sufficiently small p-value for H_{0B} suggests that at least one row-sample mean is significantly different than the other row-sample means; i.e., there is a main effect due to factor B. A sufficiently small p-value for H_{0AB} suggests that there is an interaction between factors A and B. The choice of a limit for the p-value to determine whether a result is “statistically significant” is left to the researcher. It is common to declare a result significant if the p-value is less than 0.05 or 0.01.

`anova2` also displays a figure showing the standard ANOVA table, which divides the variability of the data in `X` into three or four parts depending on the value of `reps`:

- The variability due to the differences among the column means
- The variability due to the differences among the row means
- The variability due to the interaction between rows and columns (if reps is greater than its default value of one)
- The remaining variability not explained by any systematic source

The ANOVA table has five columns:

- The first shows the source of the variability.

- The second shows the Sum of Squares (SS) due to each source.
- The third shows the degrees of freedom (df) associated with each source.
- The fourth shows the Mean Squares (MS), which is the ratio SS/df.
- The fifth shows the F statistics, which is the ratio of the mean squares.

`p = anova2(X,reps,`*`displayopt`*`)` enables the ANOVA table display when *`displayopt`* is `'on'` (default) and suppresses the display when *`displayopt`* is `'off'`.

`[p,table] = anova2(...)` returns the ANOVA table (including column and row labels) in cell array `table`. (Copy a text version of the ANOVA table to the clipboard by using the `Copy Text` item on the **Edit** menu.)

`[p,table,stats] = anova2(...)` returns a `stats` structure that you can use to perform a follow-up multiple comparison test.

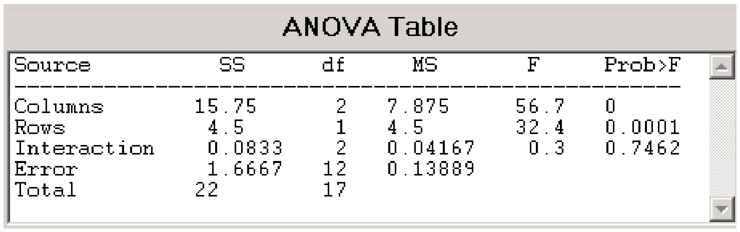
The `anova2` test evaluates the hypothesis that the row, column, and interaction effects are all the same, against the alternative that they are not all the same. Sometimes it is preferable to perform a test to determine *which pairs* of effects are significantly different, and which are not. Use the `multcompare` function to perform such tests by supplying the `stats` structure as input.

Examples

The data below come from a study of popcorn brands and popper type (Hogg 1987). The columns of the matrix `popcorn` are brands (Gourmet, National, and Generic). The rows are popper type (Oil and Air.) The study popped a batch of each brand three times with each popper. The values are the yield in cups of popped popcorn.

```
load popcorn

popcorn
popcorn =
  5.5000  4.5000  3.5000
```

```
    5.5000  4.5000  4.0000
    6.0000  4.0000  3.0000
    6.5000  5.0000  4.0000
    7.0000  5.5000  5.0000
    7.0000  5.0000  4.5000

p = anova2(popcorn,3)
p =
    0.0000  0.0001  0.7462
```



ANOVA Table

```
Source          SS      df      MS        F      Prob>F
-------------------------------------------------------
Columns       15.75      2   7.875      56.7    0
Rows           4.5       1   4.5        32.4    0.0001
Interaction    0.0833    2   0.04167     0.3    0.7462
Error          1.6667   12   0.13889
Total         22        17
```

The vector p shows the p-values for the three brands of popcorn, 0.0000, the two popper types, 0.0001, and the interaction between brand and popper type, 0.7462. These values indicate that both popcorn brand and popper type affect the yield of popcorn, but there is no evidence of a synergistic (interaction) effect of the two.

The conclusion is that you can get the greatest yield using the Gourmet brand and an Air popper (the three values `popcorn(4:6,1)`).

Reference

[1] Hogg, R. V. and J. Ledolter, *Engineering Statistics.* MacMillan, 1987.

See Also

`anova1`, `anovan`

anovan

Purpose

N-way analysis of variance

Syntax

```
p = anovan(x,group)
p = anovan(x,group,param1,val1,param2,val2,...)
[p,table] = anovan(...)
[p,table,stats] = anovan(...)
[p,table,stats,terms] = anovan(...)
```

Description

`p = anovan(x,group)` performs multiway (n-way) analysis of variance (ANOVA) for testing the effects of multiple factors (grouping variables) on the mean of the vector `x`. This test compares the variance explained by factors to the left over variance that cannot be explained. The factors and factor levels of the observations in `Y` are assigned by the cell array `group`. Each of the cells in the cell array `group` contains a list of factor levels identifying the observations in `Y` with respect to one of the factors. The list within each cell can be a categorical array, numeric vector, character matrix, or single-column cell array of strings, and must have the same number of elements as `Y`. The fitted ANOVA model includes the main effects of each grouping variable. All grouping variables are treated as fixed effects by default. The result `p` is a vector of *p*-values, one per term. For an example, see "Example of Three-Way ANOVA" on page 14-26.

`p = anovan(x,group,`*`param1`*`,val1,`*`param2`*`,val2,...)` specifies one or more of the parameter name/value pairs described in the following table.

Parameter Name	Parameter Value
`'alpha'`	A number between 0 and 1 requesting 100(1 - `alpha`)% confidence bounds (default 0.05 for 95% confidence)
`'continuous'`	A vector of indices indicating which grouping variables should be treated as continuous predictors rather than as categorical predictors.

Parameter Name	Parameter Value
`'display'`	`'on'` displays an ANOVA table (the default) `'off'` omits the display
`'model'`	The type of model used. See "Model Type" on page 14-24 for a description of this parameter.
`'nested'`	A matrix `M` of 0's and 1's specifying the nesting relationships among the grouping variables. `M`(i,j) is 1 if variable i is nested in variable j.
`'random'`	A vector of indices indicating which grouping variables are random effects (all are fixed by default). See "Example: ANOVA with Random Effects" on page 7-46 for an example of how to use `'random'`.
`'sstype'`	`1`, `2`, or `3`, to specify the type of sum of squares (default is `3`). See "Sum of Squares" on page 14-25 for a description of this parameter.
`'varnames'`	A character matrix or a cell array of strings specifying names of grouping variables, one per grouping variable. When you do not specify `'varnames'`, the default labels `'X1'`, `'X2'`, `'X3'`, ..., `'XN'` are used. See "Example: ANOVA with Random Effects" on page 7-46 for an example of how to use `'varnames'`.

`[p,table] = anovan(...)` returns the ANOVA table (including factor labels) in cell array `table`. (Copy a text version of the ANOVA table to the clipboard by using the `Copy Text` item on the **Edit** menu.)

`[p,table,stats] = anovan(...)` returns a `stats` structure that you can use to perform a follow-up multiple comparison test with the `multcompare` function. See "The stats Structure" on page 14-29 for more information.

`[p,table,stats,terms] = anovan(...)` returns the main and interaction terms used in the ANOVA computations. The terms are encoded in the output matrix `terms` using the same format described above for input `'model'`. When you specify `'model'` itself in this matrix format, the matrix returned in `terms` is identical.

Model Type

This section explains how to use the argument `'model'` with the syntax:

```
[...]  = anovan(x,group,'model',modeltype)
```

The argument *`modeltype`*, which specifies the type of model the function uses, can be any one of the following:

- `'linear'` — The default `'linear'` model computes only the p-values for the null hypotheses on the N main effects.
- `'interaction'` — The `'interaction'` model computes the p-values for null hypotheses on the N main effects and the $\binom{N}{2}$ two-factor interactions.
- `'full'` — The `'full'` model computes the p-values for null hypotheses on the N main effects and interactions at all levels.
- An integer — For an integer value of `modeltype`, `k` (`k` ≤ `N`), `anovan` computes all interaction levels through the `k`th level. For example, the value `3` means main effects plus two- and three-factor interactions. The values `k=1` and `k=2` are equivalent to the `'linear'` and `'interaction'` specifications, respectively, while the value `k=N` is equivalent to the `'full'` specification.
- A matrix of term definitions having the same form as the input to the `x2fx` function. All entries must be `0` or `1` (no higher powers).

For more precise control over the main and interaction terms that `anovan` computes, `modeltype` can specify a matrix containing one row for each main or interaction term to include in the ANOVA model. Each row defines one term using a vector of N zeros and ones. The table below illustrates the coding for a 3-factor ANOVA.

Row of Matrix	Corresponding ANOVA Term
`[1 0 0]`	Main term A
`[0 1 0]`	Main term B
`[0 0 1]`	Main term C
`[1 1 0]`	Interaction term AB
`[0 1 1]`	Interaction term BC
`[1 0 1]`	Interaction term AC
`[1 1 1]`	Interaction term ABC

For example, if *`modeltype`* is the matrix `[0 1 0;0 0 1;0 1 1]`, the output vector `p` contains the p-values for the null hypotheses on the main effects B and C and the interaction effect BC, in that order. A simple way to generate the *`modeltype`* matrix is to modify the `terms` output, which codes the terms in the current model using the format described above. If `anovan` returns `[0 1 0;0 0 1;0 1 1]` for `terms`, for example, and there is no significant result for interaction BC, you can recompute the ANOVA on just the main effects B and C by specifying `[0 1 0;0 0 1]` for *`modeltype`*.

Sum of Squares

This section explains how to use the argument `'sstype'` with the syntax:

```
[...]  = anovan(x,group,'sstype',type)
```

This syntax computes the ANOVA using the type of sum-of-squares specified by *`type`*, which can be `1`, `2`, or `3` to designate Type 1, Type 2, or Type 3 sum-of-squares, respectively. The default is `3`. The value of *`type`* only influences computations on unbalanced data.

The sum of squares for any term is determined by comparing two models. The Type 1 sum of squares for a term is the reduction in residual sum of squares obtained by adding that term to a fit that already includes the terms listed before it. The Type 2 sum of squares is the reduction in residual sum of squares obtained by adding that term to a model consisting of all other terms that do not contain the term in question. The Type 3 sum of squares is the reduction in residual sum of squares obtained by adding that term to a model containing all other terms, but with their effects constrained to obey the usual "sigma restrictions" that make models estimable.

Suppose you are fitting a model with two factors and their interaction, and that the terms appear in the order A, B, AB. Let R(·) represent the residual sum of squares for a model, so for example R(A,B,AB) is the residual sum of squares fitting the whole model, R(A) is the residual sum of squares fitting just the main effect of A, and R(1) is the residual sum of squares fitting just the mean. The three types of sums of squares are as follows:

Term	Type 1 SS	Type 2 SS	Type 3 SS
A	R(1)-R(A)	R(B)-R(A,B)	R(B,AB)-R(A,B,AB)
B	R(A)-R(A,B)	R(A)-R(A,B)	R(A,AB)-R(A,B,AB)
AB	R(A,B)-R(A,B,AB)	R(A,B)-R(A,B,AB)	R(A,B)-R(A,B,AB)

The models for Type 3 sum of squares have sigma restrictions imposed. This means, for example, that in fitting R(B,AB), the array of AB effects is constrained to sum to 0 over A for each value of B, and over B for each value of A.

Example of Three-Way ANOVA

As an example of three-way ANOVA, consider the vector `y` and `group` inputs below.

```
y = [52.7 57.5 45.9 44.5 53.0 57.0 45.9 44.0]';
g1 = [1 2 1 2 1 2 1 2];
g2 = {'hi';'hi';'lo';'lo';'hi';'hi';'lo';'lo'};
g3 = {'may';'may';'may';'may';'june';'june';'june';'june'};
```

This defines a three-way ANOVA with two levels of each factor. Every observation in y is identified by a combination of factor levels. If the factors are A, B, and C, then observation y(1) is associated with

- Level 1 of factor A
- Level 'hi' of factor B
- Level 'may' of factor C

Similarly, observation y(6) is associated with

- Level 2 of factor A
- Level 'hi' of factor B
- Level 'june' of factor C

To compute the ANOVA, enter

```
p = anovan(y, {g1 g2 g3})
p =
  0.4174
  0.0028
  0.9140
```

Output vector p contains p-values for the null hypotheses on the N main effects. Element p(1) contains the p-value for the null hypotheses, H_{0A}, that samples at all levels of factor A are drawn from the same population; element p(2) contains the p-value for the null hypotheses, H_{0B}, that samples at all levels of factor B are drawn from the same population; and so on.

If any p-value is near zero, this casts doubt on the associated null hypothesis. For example, a sufficiently small p-value for H_{0A} suggests that at least one A-sample mean is significantly different from the other A-sample means; that is, there is a main effect due to factor A. You need to choose a bound for the p-value to determine whether a result is statistically significant. It is common to declare a result significant if the p-value is less than 0.05 or 0.01.

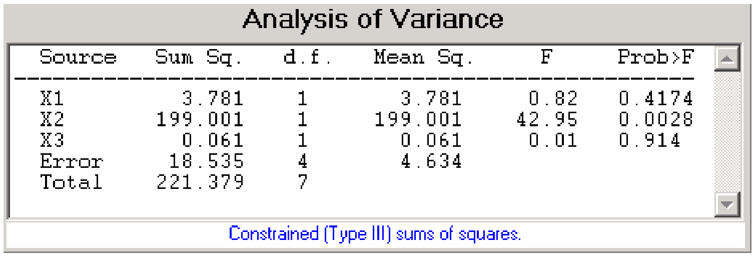
`anovan` also displays a figure showing the standard ANOVA table, which by default divides the variability of the data in `x` into

- The variability due to differences between the levels of each factor accounted for in the model (one row for each factor)
- The remaining variability not explained by any systematic source

The ANOVA table has six columns:

- The first shows the source of the variability.
- The second shows the sum of squares (SS) due to each source.
- The third shows the degrees of freedom (df) associated with each source.
- The fourth shows the mean squares (MS), which is the ratio SS/df.
- The fifth shows the F statistics, which are the ratios of the mean squares.
- The sixth shows the p-values for the F statistics.

The table is shown in the following figure:



```
  Source     Sum Sq.    d.f.    Mean Sq.      F      Prob>F
-------------------------------------------------------------
  X1           3.781     1         3.781     0.82    0.4174
  X2         199.001     1       199.001    42.95    0.0028
  X3           0.061     1         0.061     0.01    0.914
  Error       18.535     4         4.634
  Total      221.379     7
```

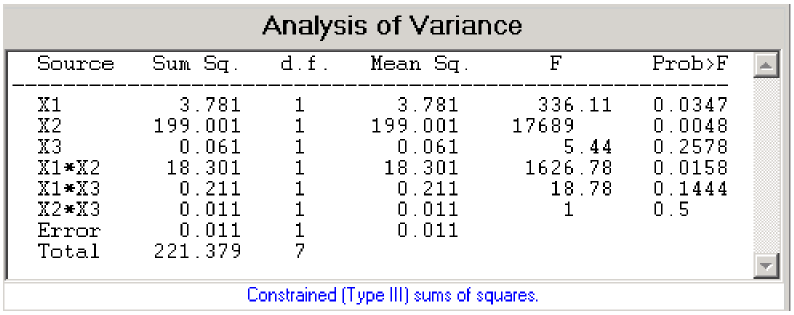
Constrained (Type III) sums of squares.

Two-Factor Interactions

By default, `anovan` computes p-values just for the three main effects. To also compute p-values for the two-factor interactions, `X1*X2`, `X1*X3`, and `X2*X3`, add the name/value pair `'model', 'interaction'` as input arguments.

```
p = anovan(y, {g1 g2 g3}, 'model', 'interaction')
p =
  0.0347
  0.0048
  0.2578
  0.0158
  0.1444
  0.5000
```

The first three entries of p are the p-values for the main effects. The last three entries are the p-values for the two-factor interactions. You can determine the order in which the two-factor interactions occur from the ANOVAN table shown in the following figure.



```
 Source    Sum Sq.   d.f.   Mean Sq.      F       Prob>F
-----------------------------------------------------------
 X1          3.781    1       3.781     336.11    0.0347
 X2        199.001    1     199.001   17689       0.0048
 X3          0.061    1       0.061       5.44    0.2578
 X1*X2      18.301    1      18.301    1626.78    0.0158
 X1*X3       0.211    1       0.211      18.78    0.1444
 X2*X3       0.011    1       0.011       1       0.5
 Error       0.011    1       0.011
 Total     221.379    7
```

Constrained (Type III) sums of squares.

The stats Structure

The `anovan` test evaluates the hypothesis that the different levels of a factor (or more generally, a term) have the same effect, against the alternative that they do not all have the same effect. Sometimes it is preferable to perform a test to determine which pairs of levels are significantly different, and which are not. Use the `multcompare` function to perform such tests by supplying the `stats` structure as input.

The `stats` structure contains the fields listed below, in addition to a number of other fields required for doing multiple comparisons using the `multcompare` function:

Stats Field	Meaning
`coeffs`	Estimated coefficients
`coeffnames`	Name of term for each coefficient
`vars`	Matrix of grouping variable values for each term
`resid`	Residuals from the fitted model

The `stats` structure also contains the following fields if there are random effects:

Stats Field	Meaning
`ems`	Expected mean squares
`denom`	Denominator definition
`rtnames`	Names of random terms
`varest`	Variance component estimates (one per random term)
`varci`	Confidence intervals for variance components

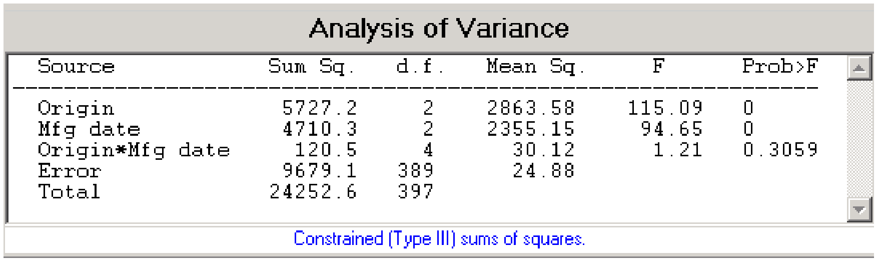
Examples

"Example: Two-Way ANOVA" on page 7-38 shows how to use `anova2` to analyze the effects of two factors on a response in a balanced design. For a design that is not balanced, use `anovan` instead.

In this example, the data set `carbig` contains a number of measurements on 406 cars. You can use `anonvan` to study how the mileage depends on where and when the cars were made.

```
load carbig

anovan(MPG,{org when},2,3,{'Origin';'Mfg date'})
ans =
      0
      0
    0.3059
```

The p-value for the interaction term is not small, indicating little evidence that the effect of the car's year or manufacture (`when`) depends on where the car was made (`org`). The linear effects of those two factors, though, are significant.



Analysis of Variance

```
Source              Sum Sq.   d.f.   Mean Sq.      F      Prob>F
-----------------------------------------------------------------
Origin               5727.2     2    2863.58    115.09    0
Mfg date             4710.3     2    2355.15     94.65    0
Origin*Mfg date       120.5     4      30.12      1.21    0.3059
Error                9679.1   389      24.88
Total               24252.6   397
```

Constrained (Type III) sums of squares.

Reference

[1] Hogg, R. V., and J. Ledolter, *Engineering Statistics*, MacMillan, 1987.

See Also

`anova1`, `anova2`, `multcompare`

ansaribradley

Purpose

Ansari-Bradley test

Syntax

```
h = ansaribradley(x,y)
h = ansaribradley(x,y,alpha)
h = ansaribradley(x,y,alpha,tail)
[h,p] = ansaribradley(...)
[h,p,stats] = ansaribradley(...)
[...] = ansaribradley(x,y,alpha,tail,exact)
[...] = ansaribradley(x,y,alpha,tail,exact,dim)
```

Description

`h = ansaribradley(x,y)` performs an Ansari-Bradley test of the hypothesis that two independent samples, in the vectors `x` and `y`, come from the same distribution, against the alternative that they come from distributions that have the same median and shape but different dispersions (e.g. variances). The result is `h = 0` if the null hypothesis of identical distributions cannot be rejected at the 5% significance level, or `h = 1` if the null hypothesis can be rejected at the 5% level. `x` and `y` can have different lengths.

`x` and `y` can also be matrices or *N*-dimensional arrays. For matrices, `ansaribradley` performs separate tests along each column, and returns a vector of results. `x` and `y` must have the same number of columns. For *N*-dimensional arrays, `ansaribradley` works along the first nonsingleton dimension. `x` and `y` must have the same size along all the remaining dimensions.

`h = ansaribradley(x,y,alpha)` performs the test at the significance level `(100*alpha)`, where `alpha` is a scalar.

`h = ansaribradley(x,y,alpha,`*`tail`*`)` performs the test against the alternative hypothesis specified by the string *`tail`*. *`tail`* is one of:

- `'both'` — Two-tailed test (dispersion parameters are not equal)
- `'right'` — Right-tailed test (dispersion of `X` is greater than dispersion of `Y`)
- `'left'` — Left-tailed test (dispersion of `X` is less than dispersion of `Y`)

`[h,p] = ansaribradley(...)` returns the p-value, i.e., the probability of observing the given result, or one more extreme, by chance if the null hypothesis is true. Small values of `p` cast doubt on the validity of the null hypothesis.

`[h,p,stats] = ansaribradley(...)` returns a structure `stats` with the following fields:

- `'W'` — Value of the test statistic `W`, which is the sum of the Ansari-Bradley ranks for the `X` sample
- `'Wstar'` — Approximate normal statistic `W*`

`[...] = ansaribradley(x,y,alpha,`*`tail`*`,`*`exact`*`)` computes `p` using an exact calculation of the distribution of `W` with `exact = 'on'`. This can be time-consuming for large samples. `exact = 'off'` computes `p` using a normal approximation for the distribution of `W*`. The default if `exact` is empty is to use the exact calculation if N, the total number of rows in `x` and `y`, is 25 or less, and to use the normal approximation if $N > 25$. Pass in `[]` for `alpha` and *`tail`* to use their default values while specifying a value for `exact`. Note that N is computed before any `NaN` values (representing missing data) are removed.

`[...] = ansaribradley(x,y,alpha,`*`tail`*`,`*`exact`*`,dim)` works along dimension `dim` of `x` and `y`.

The Ansari-Bradley test is a nonparametric alternative to the two-sample F test of equal variances. It does not require the assumption that `x` and `y` come from normal distributions. The dispersion of a distribution is generally measured by its variance or standard deviation, but the Ansari-Bradley test can be used with samples from distributions that do not have finite variances.

The theory behind the Ansari-Bradley test requires that the groups have equal medians. Under that assumption and if the distributions in each group are continuous and identical, the test does not depend on the distributions in each group. If the groups do not have the same medians, the results may be misleading. Ansari and Bradley recommend subtracting the median in that case, but the distribution of the resulting test, under the null hypothesis, is no longer independent of the common distribution of x and y. If you want to perform the tests with medians subtracted, you should subtract the medians from x and y before calling ansaribradley.

Example

Is the dispersion significantly different for two model years?

```
load carsmall
[h,p,stats] = ansaribradley(MPG(Model_Year==82),MPG(Model_Year==76))
h =
     0
p =
    0.8426
stats =
        W: 526.9000
    Wstar: 0.1986
```

See Also

vartest, vartestn, ttest2

Purpose

Interactive fitting of analysis of covariance models

Syntax

```
aoctool(x,y,group)
aoctool(x,y,group,alpha)
aoctool(x,y,group,alpha,xname,yname,gname)
aoctool(x,y,group,alpha,xname,yname,gname,displayopt)
aoctool(x,y,group,alpha,xname,yname,gname,displayopt,model)
h = aoctool(...)
[h,atab,ctab] = aoctool(...)
[h,atab,ctab,stats] = aoctool(...)
```

Description

`aoctool(x,y,group)` fits a separate line to the column vectors, `x` and `y`, for each group defined by the values in the array `group`. `group` may be a categorical variable, vector, character array, or cell array of strings. (See "Grouped Data" on page 2-41.) These types of models are known as one-way analysis of covariance (ANOCOVA) models. The output consists of three figures:

- An interactive graph of the data and prediction curves
- An ANOVA table
- A table of parameter estimates

You can use the figures to change models and to test different parts of the model. More information about interactive use of the `aoctool` function appears in "The aoctool Demo" on page 7-54.

`aoctool(x,y,group,alpha)` determines the confidence levels of the prediction intervals. The confidence level is `100(1-alpha)`%. The default value of `alpha` is 0.05.

`aoctool(x,y,group,alpha,xname,yname,gname)` specifies the name to use for the `x`, `y`, and `g` variables in the graph and tables. If you enter simple variable names for the `x`, `y`, and `g` arguments, the aoctool function uses those names. If you enter an expression for one of these arguments, you can specify a name to use in place of that expression by supplying these arguments. For example, if you enter `m(:,2)` as the `x` argument, you might choose to enter `'Col 2'` as the `xname` argument.

`aoctool(x,y,group,alpha,xname,yname,gname,`*`displayopt`*`)` enables the graph and table displays when *`displayopt`* is `'on'` (default) and suppresses those displays when *`displayopt`* is `'off'`.

`aoctool(x,y,group,alpha,xname,yname,gname,`*`displayopt`*`,`*`model`*`)` specifies the initial model to fit. The value of *`model`* can be any of the following:

- `'same mean'` — Fit a single mean, ignoring grouping
- `'separate means'` — Fit a separate mean to each group
- `'same line'` — Fit a single line, ignoring grouping
- `'parallel lines'` — Fit a separate line to each group, but constrain the lines to be parallel
- `'separate lines'` — Fit a separate line to each group, with no constraints

`h = aoctool(...)` returns a vector of handles to the line objects in the plot.

`[h,atab,ctab] = aoctool(...)` returns cell arrays containing the entries in ANOVA table (`atab`) and the table of coefficient estimates (`ctab`). (You can copy a text version of either table to the clipboard by using the `Copy Text` item on the **Edit** menu.)

`[h,atab,ctab,stats] = aoctool(...)` returns a `stats` structure that you can use to perform a follow-up multiple comparison test. The ANOVA table output includes tests of the hypotheses that the slopes or intercepts are all the same, against a general alternative that they are not all the same. Sometimes it is preferable to perform a test to determine which pairs of values are significantly different, and which are not. You can use the `multcompare` function to perform such tests by supplying the `stats` structure as input. You can test either the slopes, the intercepts, or population marginal means (the heights of the curves at the mean `x` value).

Example

This example illustrates how to fit different models non-interactively. After loading the smaller car data set and fitting a separate-slopes model, you can examine the coefficient estimates.

```
load carsmall
[h,a,c,s] = aoctool(Weight,MPG,Model_Year,0.05,...
                    '','','','off','separate lines');
c(:,1:2)
ans =
  'Term'       'Estimate'
  'Intercept'  [45.97983716833132]
  ' 70'        [-8.58050531454973]
  ' 76'        [-3.89017396094922]
  ' 82'        [12.47067927549897]
  'Slope'      [-0.00780212907455]
  ' 70'        [ 0.00195840368824]
  ' 76'        [ 0.00113831038418]
  ' 82'        [-0.00309671407243]
```

Roughly speaking, the lines relating `MPG` to `Weight` have an intercept close to 45.98 and a slope close to -0.0078. Each group's coefficients are offset from these values somewhat. For instance, the intercept for the cars made in 1970 is 45.98-8.58 = 37.40.

Next, try a fit using parallel lines. (The ANOVA table shows that the parallel-lines fit is significantly worse than the separate-lines fit.)

```
[h,a,c,s] = aoctool(Weight,MPG,Model_Year,0.05,...
                    '','','','off','parallel lines');

c(:,1:2)

ans =

  'Term'       'Estimate'
  'Intercept'  [43.38984085130596]
  ' 70'        [-3.27948192983761]
  ' 76'        [-1.35036234809006]
```

```
    ' 82'        [ 4.62984427792768]
    'Slope'      [-0.00664751826198]
```

Again, there are different intercepts for each group, but this time the slopes are constrained to be the same.

See Also anova1, multcompare, polytool

Purpose

Bartlett's test for dimensionality

Syntax

```
ndim = barttest(x,alpha)
[ndim,prob,chisquare] = barttest(x,alpha)
```

Description

`ndim = barttest(x,alpha)` returns the number of dimensions necessary to explain the nonrandom variation in the data matrix `x`, using the significance probability `alpha`. The dimension is determined by a series of hypothesis tests. The test for `ndim=1` tests the hypothesis that the variances of the data values along each principal component are equal, the test for `ndim=2` tests the hypothesis that the variances along the second through last components are equal, and so on.

`[ndim,prob,chisquare] = barttest(x,alpha)` returns the number of dimensions, the significance values for the hypothesis tests, and the χ^2 values associated with the tests.

Example

```
x = mvnrnd([0 0],[1 0.99; 0.99 1],20);
x(:,3:4) = mvnrnd([0 0],[1 0.99; 0.99 1],20);
x(:,5:6) = mvnrnd([0 0],[1 0.99; 0.99 1],20);
[ndim, prob] = barttest(x,0.05)
ndim =
   3
prob =
     0
     0
     0
   0.5081
   0.6618
```

See Also

`princomp`, `pcacov`, `pcares`

bbdesign

Purpose

Generate Box-Behnken design

Syntax

```
D = bbdesign(nfactors)
[D,blk] = bbdesign(nfactors)
[...] = bbdesign(nfactors,param1,val1,param2,val2,...)
```

Description

`D = bbdesign(nfactors)` generates a Box-Behnken design for `nfactors` factors. The output matrix `D` is `n`-by-`nfactors`, where `n` is the number of points in the design. Each row lists the settings for all factors, scaled between `-1` and `1`.

`[D,blk] = bbdesign(nfactors)` requests a blocked design. The output vector `blk` is a vector of block numbers. Blocks are groups of runs that are to be measured under similar conditions (for example, on the same day). Blocked designs minimize the effect of between-block differences on the parameter estimates.

`[...]  = bbdesign(nfactors,`*`param1`*`,`*`val1`*`,`*`param2`*`,`*`val2`*`,...)` allows you to specify additional parameters and their values. Valid parameters are:

`'center'`	Number of center points to include
`'blocksize'`	Maximum number of points allowed in a block

Remarks

Box and Behnken proposed designs when the number of factors was equal to 3-7, 9-12, or 16. This function produces those designs. For other values of `nfactors`, this function produces designs that are constructed in a similar way, even though they were not tabulated by Box and Behnken, and they may be too large to be practical.

See Also

`ccdesign`, `cordexch`, `rowexch`

Purpose

Beta cumulative distribution function

Syntax

`p = betacdf(X,A,B)`

Description

`p = betacdf(X,A,B)` computes the beta cdf at each of the values in `X` using the corresponding parameters in `A` and `B`. `X`, `A`, and `B` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs. The parameters in `A` and `B` must all be positive, and the values in `X` must lie on the interval [0,1].

The beta cdf for a given value `x` and given pair of parameters `a` and `b` is

$$p = F(x|a,b) = \frac{1}{B(a,b)}\int_0^x t^{a-1}(1-t)^{b-1}dt$$

where $B(\cdot)$ is the Beta function.

Examples

```
x = 0.1:0.2:0.9;
a = 2;
b = 2;
p = betacdf(x,a,b)
p =
    0.0280  0.2160  0.5000  0.7840  0.9720

a = [1 2 3];
p = betacdf(0.5,a,a)
p =
    0.5000  0.5000  0.5000
```

See Also

`betafit`, `betainv`, `betalike`, `betapdf`, `betarnd`, `betastat`, `cdf`

Purpose

Parameter estimates and confidence intervals for beta distributed data

Syntax

```
phat = betafit(data)
[phat,pci] = betafit(data,alpha)
```

Description

`phat = betafit(data)` computes the maximum likelihood estimates of the beta distribution parameters a and b from the data in the vector `data` and returns a column vector containing the a and b estimates, where the beta cdf is given by

$$F(x|a,b) = \frac{1}{B(a,b)}\int_0^x t^{a-1}(1-t)^{b-1}dt$$

and $B(\cdot)$ is the Beta function. The elements of `data` must lie in the interval `(0 1)`.

`[phat,pci] = betafit(data,alpha)` returns confidence intervals on the a and b parameters in the 2-by-2 matrix `pci`. The first column of the matrix contains the lower and upper confidence bounds for parameter a, and the second column contains the confidence bounds for parameter b. The optional input argument `alpha` is a value in the range `[0 1]` specifying the width of the confidence intervals. By default, `alpha` is `0.05`, which corresponds to 95% confidence intervals.

Example

This example generates 100 beta distributed observations. The true a and b parameters are 4 and 3, respectively. Compare these to the values returned in `p` by the beta fit. Note that the columns of `ci` both bracket the true parameters.

```
data = betarnd(4,3,100,1);
[p,ci] = betafit(data,0.01)
p =
  3.9010  2.6193
ci =
  2.5244  1.7488
  5.2776  3.4898
```

Reference

[1] Hahn, Gerald J., and Shapiro, Samuel S., *Statistical Models in Engineering*. John Wiley & Sons, 1994. p. 95.

See Also

`betalike`, `mle`

betainv

Purpose

Inverse of beta cumulative distribution function

Syntax

```
X = betainv(P,A,B)
```

Description

`X = betainv(P,A,B)` computes the inverse of the beta cdf with parameters specified by `A` and `B` for the corresponding probabilities in `P`. `P`, `A`, and `B` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs. The parameters in `A` and `B` must all be positive, and the values in `P` must lie on the interval [0, 1].

The inverse beta cdf for a given probability p and a given pair of parameters a and b is

$$x = F^{-1}(p|a,b) = \{x : F(x|a,b) = p\}$$

where

$$p = F(x|a,b) = \frac{1}{B(a,b)}\int_0^x t^{a-1}(1-t)^{b-1}dt$$

and $B(\cdot)$ is the Beta function. Each element of output `X` is the value whose cumulative probability under the beta cdf defined by the corresponding parameters in `A` and `B` is specified by the corresponding value in `P`.

Algorithm

The `betainv` function uses Newton's method with modifications to constrain steps to the allowable range for x, i.e., [0 1].

Examples

```
p = [0.01 0.5 0.99];
x = betainv(p,10,5)
x =
  0.3726  0.6742  0.8981
```

According to this result, for a beta cdf with a=10 and b=5, a value less than or equal to 0.3726 occurs with probability 0.01. Similarly, values less than or equal to 0.6742 and 0.8981 occur with respective probabilities 0.5 and 0.99.

See Also `betacdf`, `betafit`, `betapdf`, `betarnd`, `betastat`, `betalike`,`icdf`

betalike

Purpose

Negative log-likelihood for beta distribution

Syntax

```
nlogL = betalike(params,data)
[nlogL,AVAR] = betalike(params,data)
```

Description

`nlogL = betalike(params,data)` returns the negative of the beta log-likelihood function for the beta parameters a and b specified in vector `params` and the observations specified in the column vector `data`. The length of `nlogL` is the length of `data`.

`[nlogL,AVAR] = betalike(params,data)` also returns `AVAR`, which is the asymptotic variance-covariance matrix of the parameter estimates if the values in `params` are the maximum likelihood estimates. `AVAR` is the inverse of Fisher's information matrix. The diagonal elements of `AVAR` are the asymptotic variances of their respective parameters.

`betalike` is a utility function for maximum likelihood estimation of the beta distribution. The likelihood assumes that all the elements in the data sample are mutually independent. Since `betalike` returns the negative beta log-likelihood function, minimizing `betalike` using `fminsearch` is the same as maximizing the likelihood.

Example

This example continues the `betafit` example, which calculates estimates of the beta parameters for some randomly generated beta distributed data.

```
r = betarnd(4,3,100,1);
[nlogl,AVAR] = betalike(betafit(r),r)
nlogl =
 -39.1615
AVAR =
   0.3717  0.2644
   0.2644  0.2414
```

See Also

`betafit`, `fminsearch`, `gamlike`, `mle`, `normlike`, `wbllike`

Purpose

Beta probability density function

Syntax

```
Y = betapdf(X,A,B)
```

Description

`Y = betapdf(X,A,B)` computes the beta pdf at each of the values in `X` using the corresponding parameters in `A` and `B`. `X`, `A`, and `B` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions of the other inputs. The parameters in `A` and `B` must all be positive, and the values in `X` must lie on the interval `[0, 1]`.

The beta probability density function for a given value x and given pair of parameters a and b is

$$y = f(x|a,b) = \frac{1}{B(a,b)} x^{a-1}(1-x)^{b-1} I_{(0,1)}(x)$$

where $B(\cdot)$ is the Beta function. The indicator function $I_{(0,1)}(x)$ ensures that only values of x in the range (0 1) have nonzero probability. The uniform distribution on (0 1) is a degenerate case of the beta pdf where $a = 1$ and $b = 1$.

A *likelihood function* is the pdf viewed as a function of the parameters. Maximum likelihood estimators (MLEs) are the values of the parameters that maximize the likelihood function for a fixed value of x.

Examples

```
a = [0.5 1; 2 4]
a =
  0.5000  1.0000
  2.0000  4.0000
y = betapdf(0.5,a,a)
y =
  0.6366  1.0000
  1.5000  2.1875
```

See Also

`betacdf`, `betafit`, `betainv`, `betalike`, `betarnd`, `betastat`, `betapdf`

betarnd

Purpose

Random numbers from beta distribution

Syntax

```
R = betarnd(A,B)
R = betarnd(A,B,v)
R = betarnd(A,B,m,n)
R = betarnd(A,B,m,n,o,...)
```

Description

`R = betarnd(A,B)` generates random numbers from the beta distribution with parameters specified by `A` and `B`. `A` and `B` can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of `R`. A scalar input for `A` or `B` is expanded to a constant array with the same dimensions as the other input.

`R = betarnd(A,B,v)` generates an array `R` of size `v` containing random numbers from the beta distribution with parameters `A` and `B`, where `v` is a row vector. If v is a 1-by-2 vector, `R` is a matrix with `v(1)` rows and `v(2)` columns. If `v` is 1-by-n, `R` is an n-dimensional array.

`R = betarnd(A,B,m,n)` generates an `m`-by-`n` matrix containing random numbers from the beta distribution with parameters `A` and `B`.

`R = betarnd(A,B,m,n,o,...)` generates an m-by-n-by-o-by-... multidimensional array containing random numbers from the beta distribution with parameters `A` and `B`.

Example

```
a = [1 1;2 2];
b = [1 2;1 2];

r = betarnd(a,b)
r =
  0.6987  0.6139
  0.9102  0.8067

r = betarnd(10,10,[1 5])
r =
  0.5974  0.4777  0.5538  0.5465  0.6327

r = betarnd(4,2,2,3)
```

```
r =
  0.3943  0.6101  0.5768
  0.5990  0.2760  0.5474
```

See Also `betacdf`, `betafit`, `betainv`, `betalike`, `betapdf`, `betastat`, `rand`, `randn`, `randtool`

betastat

Purpose

Mean and variance of beta distribution

Syntax

```
[M,V] = betastat(A,B)
```

Description

`[M,V] = betastat(A,B)`, with `A>0` and `B>0`, returns the mean of and variance for the beta distribution with parameters specified by `A` and `B`. `A` and `B` can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of `M` and `V`. A scalar input for `A` or `B` is expanded to a constant array with the same dimensions as the other input.

The mean of the beta distribution with parameters a and b is $a/(a+b)$ and the variance is

$$\frac{ab}{(a+b+1)(a+b)^2}$$

Examples

If parameters a and b are equal, the mean is 1/2.

```
a = 1:6;
[m,v] = betastat(a,a)
m =
  0.5000   0.5000   0.5000   0.5000   0.5000   0.5000
v =
  0.0833   0.0500   0.0357   0.0278   0.0227   0.0192
```

See Also

`betacdf`, `betafit`, `betainv`, `betalike`, `betapdf`, `betarnd`

Purpose

Binomial cumulative distribution function

Syntax

```
Y = binocdf(X,N,P)
```

Description

`Y = binocdf(X,N,P)` computes a binomial cdf at each of the values in `X` using the corresponding parameters in `N` and `P`. `X`, `N`, and `P` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions of the other inputs. The values in `N` must all be positive integers, the values in `X` must lie on the interval `[0,N]`, and the values in `P` must lie on the interval `[0 1]`.

The binomial cdf for a given value x and given pair of parameters n and p is

$$y = F(x|n,p) = \sum_{i=0}^{x} \binom{n}{i} p^i q^{(n-i)} I_{(0,1,\ldots,n)}(i)$$

The result, y, is the probability of observing up to x successes in n independent trials, where the probability of success in any given trial is p. The indicator function $I_{(0,1,\ldots,n)}(i)$ ensures that x only adopts values of $0,1,\ldots,n$.

Examples

If a baseball team plays 162 games in a season and has a 50-50 chance of winning any game, then the probability of that team winning more than 100 games in a season is:

```
1 - binocdf(100,162,0.5)
```

The result is `0.001` (i.e., `1-0.999`). If a team wins 100 or more games in a season, this result suggests that it is likely that the team's true probability of winning any game is greater than 0.5.

See Also

`binofit`, `binoinv`, `binopdf`, `binornd`, `binostat`, `cdf`

Purpose

Parameter estimates and confidence intervals for binomial distributed data

Syntax

```
phat = binofit(x,n)
[phat,pci] = binofit(x,n)
[phat,pci] = binofit(x,n,alpha)
```

Description

`phat = binofit(x,n)` returns a maximum likelihood estimate of the probability of success in a given binomial trial based on the number of successes, `x`, observed in `n` independent trials. If `x = (x(1), x(2), ... x(k))` is a vector, `binofit` returns a vector of the same size as `x` whose ith entry is the parameter estimate for `x(i)`. All `k` estimates are independent of each other. If `n = (n(1), n(2), ..., n(k))` is a vector of the same size as `x`, the binomial fit, `binofit`, returns a vector whose ith entry is the parameter estimate based on the number of successes `x(i)` in `n(i)` independent trials. A scalar value for `x` or `n` is expanded to the same size as the other input.

`[phat,pci] = binofit(x,n)` returns the probability estimate, `phat`, and the 95% confidence intervals, `pci`.

`[phat,pci] = binofit(x,n,alpha)` returns the `100(1 - alpha)`% confidence intervals. For example, `alpha` = `0.01` yields 99% confidence intervals.

Note `binofit` behaves differently than other functions in Statistics Toolbox that compute parameter estimates, in that it returns independent estimates for each entry of `x`. By comparison, `expfit` returns a single parameter estimate based on all the entries of `x`.

Unlike most other distribution fitting functions, the `binofit` function treats its input `x` vector as a collection of measurements from separate samples. If you want to treat `x` as a single sample and compute a single parameter estimate for it, you can use `binofit(sum(x),sum(n))` when `n` is a vector, and `binofit(sum(X),N*length(X))` when `n` is a scalar.

Example

This example generates a binomial sample of 100 elements, where the probability of success in a given trial is 0.6, and then estimates this probability from the outcomes in the sample.

```
r = binornd(100,0.6);
[phat,pci] = binofit(r,100)
phat =
  0.5800
pci =
  0.4771  0.6780
```

The 95% confidence interval, `pci`, contains the true value, 0.6.

Reference

[1] Johnson, N. L., S. Kotz, and A. W. Kemp, *Univariate Discrete Distributions, 2nd edition,* Wiley, 1992, pp. 124-130.

See Also

`binocdf`, `binoinv`, `binopdf`, `binornd`, `binostat`, `mle`

binoinv

Purpose

Inverse of binomial cumulative distribution function

Syntax

```
X = binoinv(Y,N,P)
```

Description

`X = binoinv(Y,N,P)` returns the smallest integer `X` such that the binomial cdf evaluated at `X` is equal to or exceeds `Y`. You can think of `Y` as the probability of observing `X` successes in `N` independent trials where `P` is the probability of success in each trial. Each `X` is a positive integer less than or equal to `N`.

`Y`, `N`, and `P` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs. The parameters in `N` must be positive integers, and the values in both P and Y must lie on the interval [0 1].

Examples

If a baseball team has a 50-50 chance of winning any game, what is a reasonable range of games this team might win over a season of 162 games?

```
binoinv([0.05 0.95],162,0.5)
ans =
    71    91
```

This result means that in 90% of baseball seasons, a .500 team should win between 71 and 91 games.

See Also

`binocdf`, `binofit`, `binopdf`, `binornd`, `binostat`, `icdf`

Purpose

Binomial probability density function

Syntax

```
Y = binopdf(X,N,P)
```

Description

`Y = binopdf(X,N,P)` computes the binomial pdf at each of the values in `X` using the corresponding parameters in `N` and `P`. `Y`, `N`, and `P` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions of the other inputs.

The parameters in `N` must be positive integers, and the values in `P` must lie on the interval `[0 1]`.

The binomial probability density function for a given value *x* and given pair of parameters *n* and *p* is

$$y = f(x|n,p) = \binom{n}{x} p^x q^{(n-x)} I_{(0,1,\ldots,n)}(x)$$

where $q = 1\text{-}p$. The result, *y*, is the probability of observing *x* successes in *n* independent trials, where the probability of success in any *given* trial is *p*. The indicator function $I_{(0,1,\ldots,n)}(x)$ ensures that *x* only adopts values of 0, 1, ..., *n*.

Examples

A Quality Assurance inspector tests 200 circuit boards a day. If 2% of the boards have defects, what is the probability that the inspector will find no defective boards on any given day?

```
binopdf(0,200,0.02)
ans =
  0.0176
```

What is the most likely number of defective boards the inspector will find?

```
defects=0:200;
y = binopdf(defects,200,.02);
[x,i]=max(y);
defects(i)
```

```
ans =
  4
```

See Also binocdf, binofit, binoinv, binornd, binostat, pdf

Purpose

Random numbers from binomial distribution

Syntax

```
R = binornd(N,P)
R = binornd(N,P,v)
R = binornd(N,p,m,n)
```

Description

`R = binornd(N,P)` generates random numbers from the binomial distribution with parameters specified by `N` and `P`. `N` and `P` can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of `R`. A scalar input for `N` or `P` is expanded to a constant array with the same dimensions as the other input.

`R = binornd(N,P,v)` generates an array `R` of size `v` containing random numbers from the binomial distribution with parameters `N` and `P`, where `v` is a row vector. If v is a 1-by-2 vector, `R` is a matrix with `v(1)` rows and `v(2)` columns. If `v` is 1-by-n, `R` is an n-dimensional array.

`R = binornd(N,p,m,n)` generates an `m`-by-`n` matrix containing random numbers from the binomial distribution with parameters `N` and `P`.

Algorithm

The `binornd` function uses the direct method using the definition of the binomial distribution as a sum of Bernoulli random variables.

Example

```
n = 10:10:60;

r1 = binornd(n,1./n)
r1 =
   2   1   0   1   1   2

r2 = binornd(n,1./n,[1 6])
r2 =
   0   1   2   1   3   1

r3 = binornd(n,1./n,1,6)
r3 =
   0   1   1   1   0   3
```

See Also binocdf, binofit, binoinv, binopdf, binostat, rand, randtool

Purpose

Mean and variance of binomial distribution

Syntax

```
[M,V] = binostat(N,P)
```

Description

`[M,V] = binostat(N,P)` returns the mean of and variance for the binomial distribution with parameters specified by `N` and `P`. `N` and `P` can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of `M` and `V`. A scalar input for `N` or `P` is expanded to a constant array with the same dimensions as the other input.

The mean of the binomial distribution with parameters n and p is np. The variance is npq, where q = 1-p.

Examples

```
n = logspace(1,5,5)
n =
    10     100    1000    10000   100000

[m,v] = binostat(n,1./n)
m =
   1    1    1    1    1
v =
  0.9000  0.9900  0.9990  0.9999  1.0000

[m,v] = binostat(n,1/2)
m =
     5     50     500    5000    50000
v =
  1.0e+04 *
  0.0003  0.0025  0.0250  0.2500  2.5000
```

See Also

`binocdf`, `binofit`, `binoinv`, `binopdf`, `binornd`

biplot

Purpose Biplot of variable/factor coefficients and scores

Syntax

```
biplot(coefs)
biplot(coefs,...,'Scores',scores)
biplot(coefs,...,'VarLabels',varlabels)
biplot(coefs,...,'Scores',scores,'ObsLabels',obslabels)
biplot(coeffs,...,PropertyName,PropertyValue,...)
h = biplot(coefs,...)
```

Description

`biplot(coefs)` creates a biplot of the coefficients in the matrix `coefs`. The biplot is two-dimensional if `coefs` has two columns or three-dimensional if it has three columns. `coefs` usually contains principal component coefficients created with `princomp`, `pcacov`, or factor loadings estimated with `factoran`. The axes in the biplot represent the principal components or latent factors (columns of `coefs`), and the observed variables (rows of `coefs`) are represented as vectors.

`biplot(coefs,...,'Scores',scores)` plots both `coefs` and the `scores` in the matrix `scores` in the biplot. `scores` usually contains principal component scores created with `princomp` or factor scores estimated with `factoran`. Each observation (row of scores) is represented as a point in the biplot.

A biplot allows you to visualize the magnitude and sign of each variable's contribution to the first two or three principal components, and how each observation is represented in terms of those components.

`biplot` imposes a sign convention, forcing the element with largest magnitude in each column of `coefs` to be positive.

`biplot(coefs,...,'VarLabels',varlabels)` labels each vector (variable) with the text in the character array or cell array `varlabels`.

`biplot(coefs,...,'Scores',scores,'ObsLabels',obslabels)` uses the text in the character array or cell array `obslabels` as observation names when displaying data cursors.

`biplot(coeffs,...,`*`PropertyName`*`,`*`PropertyValue`*`,...)` sets properties to the specified property values for all line graphics objects created by `biplot`.

`h = biplot(coefs,...)` returns a column vector of handles to the graphics objects created by `biplot`. The `h` contains, in order, handles corresponding to variables (line handles, followed by marker handles, followed by text handles), to observations (if present, marker handles followed by text handles), and to the axis lines.

Example

```
load carsmall
x = [Acceleration Displacement Horsepower MPG Weight];
x = x(all(~isnan(x),2),:);
[coefs,score] = princomp(zscore(x));
vlabs = {'Accel','Disp','HP','MPG','Wgt'};
biplot(coefs(:,1:3),'scores',score(:,1:3),...
                    'varlabels',vlabs);
```

See Also

`factoran`, `princomp`, `pcacov`, `rotatefactors`

Purpose

Bootstrap confidence interval

Syntax

```
ci = bootci(nboot,bootfun,...)
ci = bootci(nboot,{bootfun,...},'alpha',alpha)
ci = bootci(nboot,{bootfun,...},...,'type',type)
ci = bootci(nboot,{bootfun,...},...,'type','stud','nbootstd',
   nbootstd)
ci = bootci(nboot,{bootfun,...},...,'type','stud','stderr',
   stderr)
```

Description

`ci = bootci(nboot,bootfun,...)` computes the 95% BCa bootstrap confidence interval of the statistic defined by the function `bootfun`. `nboot` is a positive integer indicating the number of bootstrap data samples used in the computation. `bootfun` is a function handle specified with `@`. The third and later input arguments to `bootci` are data (scalars, column vectors, or matrices) that are used to create inputs to `bootfun`. `bootci` creates each bootstrap sample by sampling with replacement from the rows of the non-scalar data arguments (these must have the same number of rows). Scalar data are passed to `bootfun` unchanged. `ci` is a vector containing the lower and upper bounds of the confidence interval.

`ci = bootci(nboot,{bootfun,...},'alpha',alpha)` computes the `100*(1-alpha)`% BCa bootstrap confidence interval of the statistic defined by the function `bootfun`. `alpha` is a scalar between `0` and `1`. The default value of `alpha` is `0.05`.

`ci = bootci(nboot,{bootfun,...},...,'type',type)` computes the bootstrap confidence interval of the statistic defined by the function `bootfun`. `type` is the confidence interval type, specifying different methods of computing the confidence interval. `type` is a string chosen from the following:

`'normal'` or `'norm'`	Normal approximated interval with bootstrapped bias and standard error
`'per'` or `'percentile'`	Basic percentile method

`'cper'` or `'corrected percentile'`	Bias corrected percentile method
`'bca'`	Bias corrected and accelerated percentile method
`'stud'` or `'student'`	Studentized confidence interval

The default value of `type` is `'bca'`.

`ci = bootci(nboot,{bootfun,...},...,'type','stud','nbootstd',nbootstd)` computes the studentized bootstrap confidence interval of the statistic defined by the function `bootfun`. The standard error of the bootstrap statistics is estimated using bootstrap, with `nbootstd` bootstrap data samples. `nbootstd` is a positive integer value. The default value of `nbootstd` is `100`.

`ci = bootci(nboot,{bootfun,...},...,'type','stud','stderr',stderr)` computes the studentized bootstrap confidence interval of statistics defined by the function `bootfun`. The standard error of the bootstrap statistics is evaluated by the function `stderr`. `stderr` is a function handle created using `@`. `stderr` takes the same arguments as `bootfun` and returns the standard error of the statistic computed by `bootfun`.

Examples

Computing the Bootstrap Confidence Interval for Statistical Process Control

Compute the confidence interval for the capability index in statistical process control.

```
y = normrnd(1,1,30,1);                  % Simulated process data
LSL = -3; USL = 3;                      % Process specifications
capable = @(x) (USL-LSL)./(6* std(x));  % Process capability
bootci(2000,capable, y)                 % BCa confidence interval
ans =
    0.8122
    1.2657
```

```
bootci(2000,{capable, y},'type','stud') % Studentized ci
ans =
    0.7739
    1.2707
```

See Also bootstrp, jackknife

Purpose

Bootstrap statistics through resampling of data

Syntax

```
bootstat = bootstrp(nboot,bootfun,d1,...)
[bootstat,bootsam] = bootstrp(...)
```

Description

`bootstat = bootstrp(nboot,bootfun,d1,...)` draws `nboot` bootstrap data samples, computes statistics on each sample using `bootfun`, and returns the results in the matrix `bootstat`. `nboot` must be a positive integer. `bootfun` is a function handle specified with `@`. Each row of `bootstat` contains the results of applying `bootfun` to one bootstrap sample. If `bootfun` returns a matrix or array, then this output is converted to a row vector for storage in `bootstat`.

The third and later input arguments (`d1,...`) are data (scalars, column vectors, or matrices) used to create inputs to `bootfun`. `bootstrp` creates each bootstrap sample by sampling with replacement from the rows of the non-scalar data arguments (these must have the same number of rows). `bootfun` accepts scalar data unchanged.

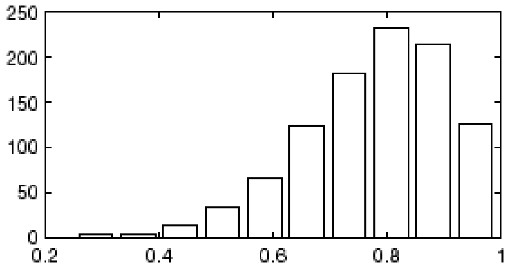
`[bootstat,bootsam] = bootstrp(...)` returns an `n`-by-`nboot` matrix of bootstrap indices, `bootsam`. Each column in `bootsam` contains indices of the values that were drawn from the original data sets to constitute the corresponding bootstrap sample. For example, if `d1,...` each contain 16 values, and `nboot = 4`, then `bootsam` is a 16-by-4 matrix. The first column contains the indices of the 16 values drawn from `d1,...`, for the first of the four bootstrap samples, the second column contains the indices for the second of the four bootstrap samples, and so on. (The bootstrap indices are the same for all input data sets.) To get the output samples `bootsam` without applying a function, set `bootfun` to empty (`[]`).

Examples

Bootstrapping a Correlation Coefficient Standard Error

Load a data set containing the LSAT scores and law-school GPA for 15 students. These 15 data points are resampled to create 1000 different data sets, and the correlation between the two variables is computed for each data set.

```
load lawdata
[bootstat,bootsam] = bootstrp(1000,@corr,lsat,gpa);
```



The histogram shows the variation of the correlation coefficient across all the bootstrap samples. The sample minimum is positive, indicating that the relationship between LSAT score and GPA is not accidental.

Finally, compute a bootstrap standard of error for the estimated correlation coefficient.

```
se = std(bootstat)
se =
    0.1327
```

Display the first 5 bootstrapped correlation coefficients.

```
bootstat(1:5,:)
ans =
    0.6600
    0.7969
    0.5807
    0.8766
    0.9197
```

Display the indices of the data selected for the first 5 bootstrap samples.

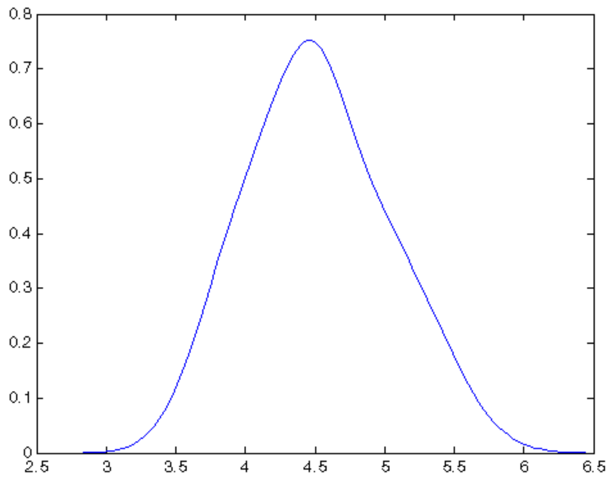
```
bootsam(:,1:5)
ans =
     9     8    15    11    15
    14     7     6     7    14
```

```
         4     6    10     3    11
         3    10    11     9     2
        15     4    13     4    14
         9     4     5     2    10
         8     5     4     3    13
         1     9     1    15    11
        10     8     6    12     3
         1     4     5     2     8
         1     1    10     6     2
         3    10    15    10     8
        14     6    10     3     8
        13    12     1     2     4
        12     6     4     9     8
hist(bootstat)
```

Estimating the Density of Bootstrapped Statistic

Compute a sample of 100 bootstrapped means of random samples taken from the vector Y, and plot an estimate of the density of these bootstrapped means:

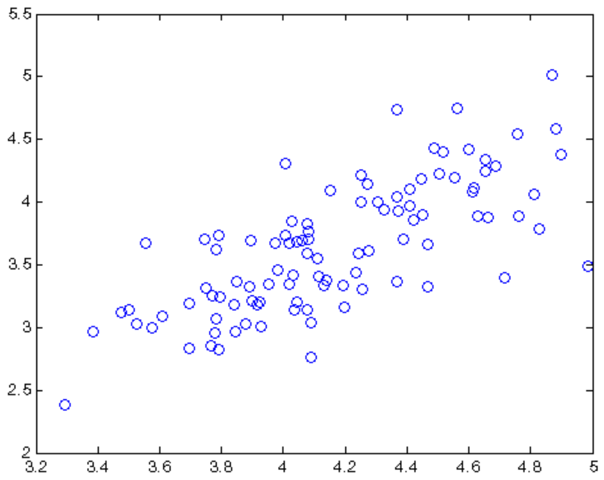
```
y = exprnd(5,100,1);
m = bootstrp(100,@mean,y);
[fi,xi] = ksdensity(m);
plot(xi,fi);
```



Bootstrapping More Than One Statistic

Compute a sample of 100 bootstrapped means and standard deviations of random samples taken from the vector Y, and plot the bootstrap estimate pairs:

```
y = exprnd(5,100,1);
stats = bootstrp(100,@(x)[mean(x) std(x)],y);
plot(stats(:,1),stats(:,2),'o')
```



Bootstrapping a Regression Model

Estimate the standard errors for a coefficient vector in a linear regression by bootstrapping residuals:

```
load hald
x = [ones(size(heat)),ingredients];
y = heat;
b = regress(y,x);
yfit = x*b;
resid = y - yfit;
se = std(bootstrp(...
         1000,@(bootr)regress(yfit+bootr,x),resid));
```

boundary

Purpose

Boundary points of piecewise distribution segments

Syntax

```
[p,q] = boundary(obj)
[p,q] = boundary(obj,i)
```

Description

`[p,q] = boundary(obj)` returns the boundary points between segments of the piecewise distribution object `obj`. `p` is a vector of cumulative probabilities at each boundary. `q` is a vector of quantiles at each boundary.

`[p,q] = boundary(obj,i)` returns `p` and `q` for the *i*th boundary.

Example

Fit Pareto tails to a *t* distribution at cumulative probabilities 0.1 and 0.9:

```
t = trnd(3,100,1);
obj = paretotails(t,0.1,0.9);
[p,q] = boundary(obj)
p =
    0.1000
    0.9000
q =
   -1.7766
    1.8432
```

See Also

`paretotails`, `cdf (piecewisedistribution)`, `icdf (piecewisedistribution)`, `nsegments`

Purpose

Box plot of data sample

Syntax

```
boxplot(X)
boxplot(x,group)
boxplot(...,param1,val1,param2,val2,...)
boxplot(h,...)
H = boxplot(...)
```

Description

`boxplot(X)` produces a box and whisker plot for each column of the matrix `X`. The box has lines at the lower quartile, median, and upper quartile values. Whiskers extend from each end of the box to the adjacent values in the data—by default, the most extreme values within 1.5 times the interquartile range from the ends of the box. Outliers are data with values beyond the ends of the whiskers. Outliers are displayed with a red + sign.

`boxplot(x,group)` produces a box and whisker plot for the vector `x` grouped by the grouping variable `group`. `group` is a categorical variable, vector, string matrix, or cell array of strings. (See "Grouped Data" on page 2-41.) `group` can also be a cell array of several grouping variables (such as `{g1 g2 g3}`) to group the values in `x` by each unique combination of grouping variable values.

`boxplot(...,`*`param1`*`,`*`val1`*`,`*`param2`*`,`*`val2`*`,...)` specifies optional parameter name/value pairs, as described in the following table.

Name	Value
`'notch'`	`'on'` to include notches (default is `'off'`)
`'symbol'`	Symbol to use for outliers (default is `'r+'`). See `LineSpec` for a description of symbols.
`'orientation'`	Box orientation, `'vertical'` (default) or `'horizontal'`
`'whisker'`	Maximum whisker length in units of interquartile range (default `1.5`)

Name	Value
`'labels'`	Character array or cell array of strings containing column labels (used only if `X` is a matrix, and the default label is the column number)
`'colors'`	A string, such as `'bgry'`, or a three-column matrix of box colors. Letters in the string specify colors, as described in `LineSpec`. Each box (outline, median line, and whiskers) is drawn in the corresponding color. The default is to draw all boxes with blue outline, red median, and black whiskers. Colors are reused in the same order if necessary.
`'widths'`	A numeric vector of box widths. The default is `0.5`, or slightly smaller for fewer than three boxes. Widths are reused if necessary.
`'positions'`	A numeric vector of box positions. The default is `1:n`.
`'grouporder'`	When the grouping variable `G` is given, a character array or cell array of group names, specifying the ordering of the groups in `G`. Ignored when `G` is not given.

Notches display the variability of the median between samples. The width of a notch is computed so that box plots whose notches do not overlap have different medians at the 5% significance level. The significance level is based on a normal distribution assumption, but comparisons of medians are reasonably robust for other distributions. Comparing boxplot medians is like a visual hypothesis test, analogous to the *t* test used for means.

Whiskers extend from the box out to the most extreme data value within `whis*iqr`, where `whis` is the value of the `'whisker'` parameter and `iqr` is the interquartile range of the sample.

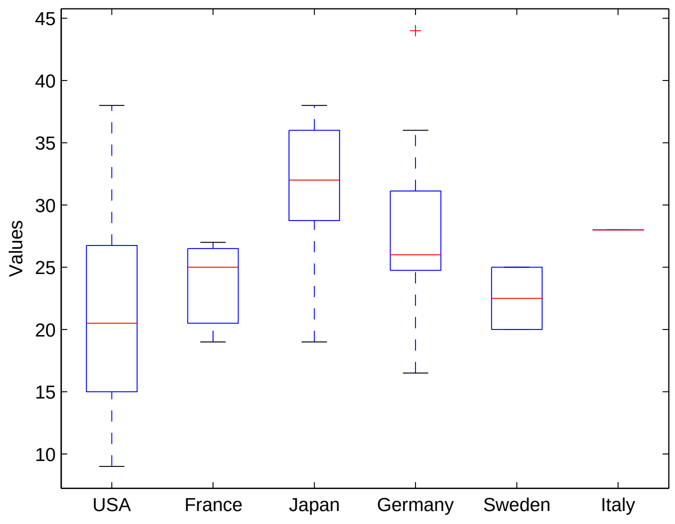
`boxplot(h,...)` plots into the axes with handle `h`.

`H = boxplot(...)` returns a matrix `H` of handles to the lines in the box plot. `H` contains one column for each box. Each column contains seven handles corresponding to the upper whisker, lower whisker, upper adjacent value, lower adjacent value, box, median, and outliers.

Examples

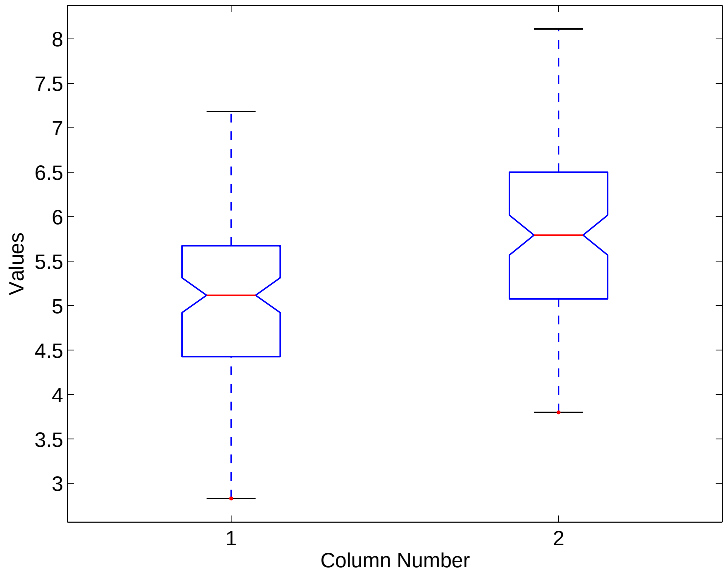
The following commands create a box plot of car mileage grouped by country.

```
load carsmall
boxplot(MPG,Origin)
```



The following example produces notched box plots for two groups of sample data.

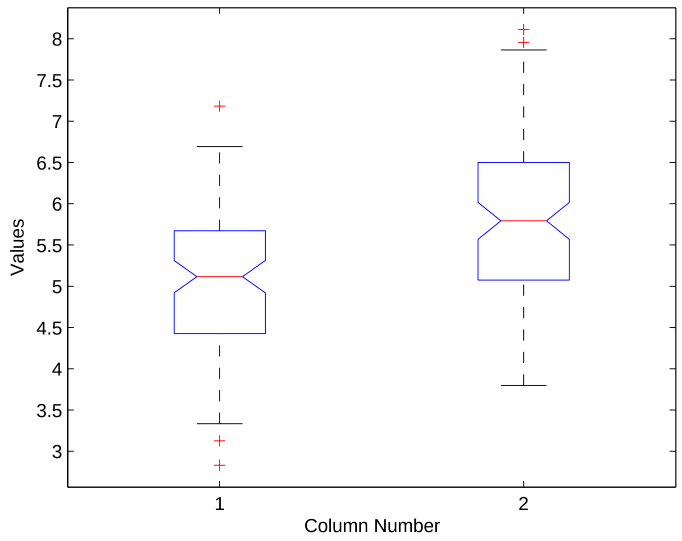
```
x1 = normrnd(5,1,100,1);
x2 = normrnd(6,1,100,1);
boxplot([x1,x2],'notch','on')
```



The difference between the medians of the two groups is approximately 1. Since the notches in the boxplot do not overlap, you can conclude, with 95% confidence, that the true medians do differ.

The following figure shows the boxplot for same data with the length of the whiskers specified as 1.0 times the interquartile range. Points beyond the whiskers are displayed using +.

```
boxplot([x1,x2],'notch','on','whisker',1)
```



References

[1] McGill, R., J. W. Tukey, and W. A. Larsen, "Variations of Boxplots," *The American Statistician*, Vol. 32, 1978, pp.12-16.

[2] Velleman, P.F., and D.C. Hoaglin, *Applications, Basics, and Computing of Exploratory Data Analysis*, Duxbury Press, 1981.

[3] Nelson, L. S., "Evaluating Overlapping Confidence Intervals," *Journal of Quality Technology*, Vol. 21, 1989, pp. 140-141.

See Also

`anova1`, `kruskalwallis`, `multcompare`

Purpose

D-optimal design from candidate set using row exchanges

Syntax

```
rlist = candexch(C,nrows)
rlist = candexch(C,nrows,param1,value1,param2,value2,...)
```

Description

`rlist = candexch(C,nrows)` uses a row-exchange algorithm to select a D-optimal design from the candidate set `C`. `C` is an `n`-by-`p` matrix containing the values of `p` model terms at each of `n` points. `nrows` is the desired number of rows in the design. `rlist` is a vector of length `nrows` listing the selected rows.

The `candexch` function selects a starting design `X` at random, and uses a row-exchange algorithm to iteratively replace rows of `X` by rows of `C` in an attempt to improve the determinant of `X'*X`.

`rlist = candexch(C,nrows,`*`param1`*`,value1,`*`param2`*`,value2,...)` provides more control over the design generation through a set of parameter/value pairs. Valid parameters are the following:

Parameter	Value
`'display'`	Either `'on'` or `'off'` to control display of iteration number The default is `'on'`.
`'init'`	Initial design as an `nrows`-by-`p` matrix. The default is a random subset of the rows of `C`.
`'maxiter'`	Maximum number of iterations. The default is `10`.

Note The `rowexch` function also generates D-optimal designs using a row-exchange algorithm, but it accepts a model type and automatically selects a candidate set that is appropriate for such a model.

Examples

Generate a D-optimal design when there is a restriction on the candidate set. In this case, the `rowexch` function isn't appropriate.

```
F = (fullfact([5 5 5])-1)/4;    % Factor settings in unit cube
```

```
T = sum(F,2)<=1.51;            % Find rows matching a restriction
F = F(T,:);                    % Take only those rows
C = [ones(size(F,1),1) F F.^2]; % Compute model terms including a
                               %  constant and all squared terms
R = candexch(C,12);            % Find a D-optimal 12-point subset
X = F(R,:);                    % Get factor settings
```

See Also

`candgen`, `cordexch`, `rowexch`, `x2fx`

candgen

Purpose

Generate candidate set for D-optimal design

Syntax

```
xcand = candgen(nfactors,model)
[xcand,fxcand] = candgen(nfactors,model)
```

Description

`xcand = candgen(nfactors,`*`model`*`)` generates a candidate set appropriate for a D-optimal design with `nfactors` factors and the model `model`. The output matrix `xcand` has `nfactors` columns, with each row representing the coordinates of a candidate point. `model` is one of:

`'linear'`	Constant and linear terms (the default)
`'interaction'`	Constant, linear, and cross product terms
`'quadratic'`	Interactions plus squared terms
`'purequadratic'`	Constant, linear, and squared terms

Alternatively, `model` can be a matrix of term definitions as accepted by the `x2fx` function.

`[xcand,fxcand] = candgen(nfactors,model)` returns both the matrix of factor values `xcand` and the matrix of term values `fxcand`. You can input the latter to `candexch` to generate the D-optimal design.

Note The `rowexch` function automatically generates a candidate set using `candgen`, and creates a D-optimal design from that candidate set using `candexch`. Call these functions separately if you want to modify the default candidate set.

See Also

`candexch`, `rowexch`, `x2fx`

Purpose

Canonical correlation analysis

Syntax

```
[A,B] = canoncorr(X,Y)
[A,B,r] = canoncorr(X,Y)
[A,B,r,U,V] = canoncorr(X,Y)
[A,B,r,U,V,stats] = canoncorr(X,Y)
```

Description

`[A,B] = canoncorr(X,Y)` computes the sample canonical coefficients for the `n`-by-`d1` and `n`-by-`d2` data matrices `X` and `Y`. `X` and `Y` must have the same number of observations (rows) but can have different numbers of variables (columns). `A` and `B` are `d1`-by-`d` and `d2`-by-`d` matrices, where `d = min(rank(X),rank(Y))`. The `j`th columns of `A` and `B` contain the canonical coefficients, i.e., the linear combination of variables making up the `j`th canonical variable for `X` and `Y`, respectively. Columns of `A` and `B` are scaled to make the covariance matrices of the canonical variables the identity matrix (see `U` and `V` below). If `X` or `Y` is less than full rank, `canoncorr` gives a warning and returns zeros in the rows of `A` or `B` corresponding to dependent columns of `X` or `Y`.

`[A,B,r] = canoncorr(X,Y)` also returns a 1-by-`d` vector containing the sample canonical correlations. The `j`th element of `r` is the correlation between the *j*th columns of `U` and `V` (see below).

`[A,B,r,U,V] = canoncorr(X,Y)` also returns the canonical variables, scores. `U` and `V` are `n`-by-`d` matrices computed as

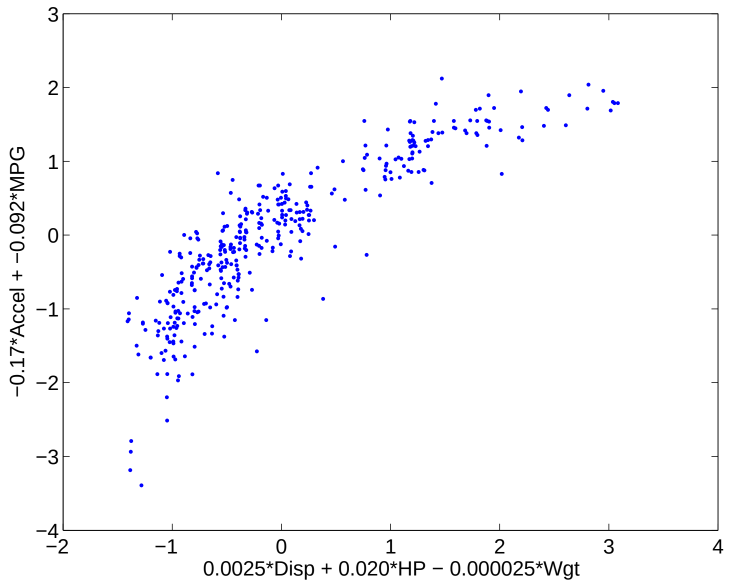
```
U = (X-repmat(mean(X),N,1))*A
V = (Y-repmat(mean(Y),N,1))*B
```

`[A,B,r,U,V,stats] = canoncorr(X,Y)` also returns a structure `stats` containing information relating to the sequence of `d` null hypotheses $H_0^{(k)}$, that the `(k+1)`st through `d`th correlations are all zero, for `k = 0:(d-1)`. `stats` contains seven fields, each a `1`-by-`d` vector with elements corresponding to the values of `k`, as described in the following table:

`Wilks`	Wilks’ lambda (likelihood ratio) statistic
`chisq`	Bartlett’s approximate chi-squared statistic for $H_0^{(k)}$ with Lawley’s modification
`pChisq`	Right-tail significance level for `chisq`
`F`	Rao’s approximate F statistic for $H_0^{(k)}$
`pF`	Right-tail significance level for `F`
`df1`	Degrees of freedom for the chi-squared statistic, and the numerator degrees of freedom for the F statistic
`df2`	Denominator degrees of freedom for the F statistic

Examples

```
load carbig;
X = [Displacement Horsepower Weight Acceleration MPG];
nans = sum(isnan(X),2) > 0;
[A B r U V] = canoncorr(X(~nans,1:3),X(~nans,4:5));

plot(U(:,1),V(:,1),'.');
xlabel('0.0025*Disp+0.020*HP-0.000025*Wgt');
ylabel('-0.17*Accel-0.092*MPG')
```



References

[1] Krzanowski, W. J., *Principles of Multivariate Analysis*, Oxford University Press, 1988.

[2] Seber, G. A. F., *Multivariate Observations*, Wiley, 1984.

See Also

`manova1`, `princomp`

Purpose

Process capability indices

Syntax

```
S = capability(data,specs)
```

Description

`S = capability(data,specs)` estimates capability indices for measurements in `data` given the specifications in `specs`. `data` can be either a vector or a matrix of measurements. If `data` is a matrix, indices are computed for the columns. `specs` can be either a two-element vector of the form `[L,U]` containing lower and upper specification limits, or (if `data` is a matrix) a two-row matrix with the same number of columns as `data`. If there is no lower bound, use `-Inf` as the first element of `specs`. If there is no upper bound, use `Inf` as the second element of `specs`.

The output `S` is a structure with the following fields:

- `mu` — Sample mean
- `sigma` — Sample standard deviation
- `P` — Estimated probability of being within limits
- `Pl` — Estimated probability of being below `L`
- `Pu` — Estimated probability of being above `U`
- `Cp` — `(U-L)/(6*sigma)`
- `Cpl` — `(mu-L)./(3.*sigma)`
- `Cpu` — `(U-mu)./(3.*sigma)`
- `Cpk` — `min(Cpl,Cpu)`

Indices are computed under the assumption that data values are independent samples from a normal population with constant mean and variance.

Indices divide a "specification width" (between specification limits) by a "process width" (between control limits). Higher ratios indicate processes with less measurements outside of specification.

Example

Simulate a sample from a process with a mean of 3 and a standard deviation of 0.005:

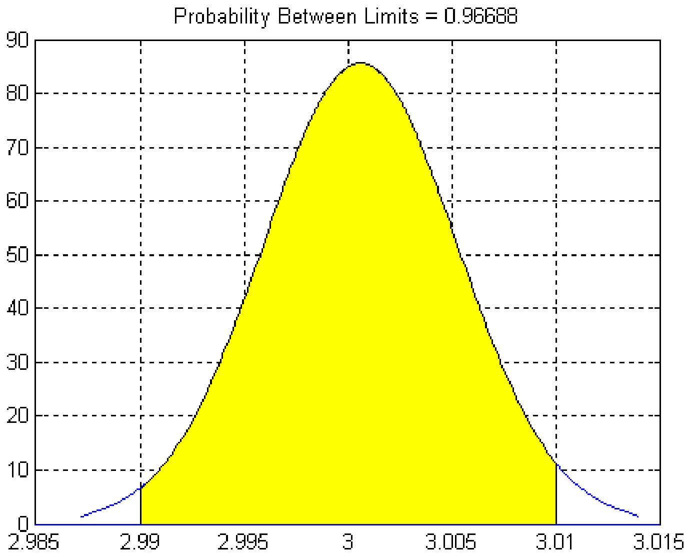
```
data = normrnd(3,0.005,100,1);
```

Compute capability indices if the process has an upper specification limit of 3.01 and a lower specification limit of 2.99:

```
S = capability(data,[2.99 3.01])
S =
       mu: 3.0006
    sigma: 0.0047
        P: 0.9669
       Pl: 0.0116
       Pu: 0.0215
       Cp: 0.7156
      Cpl: 0.7567
      Cpu: 0.6744
      Cpk: 0.6744
```

Visualize the specification and process widths:

```
capaplot(data,[2.99 3.01]);
grid on
```



Reference

[1] Montgomery, D., *Introduction to Statistical Quality Control*, John Wiley & Sons, 1991, pp. 369–374.

See Also

`capaplot`, `histfit`

Purpose

Process capability plot

Syntax

```
p = capaplot(data,specs)
[p,h] = capaplot(data,specs)
```

Description

`p = capaplot(data,specs)` estimates the mean of and variance for the observations in input vector `data`, and plots the pdf of the resulting T distribution. The observations in `data` are assumed to be normally distributed. The output, `p`, is the probability that a new observation from the estimated distribution will fall within the range specified by the two-element vector `specs`. The portion of the distribution between the lower and upper bounds specified in `specs` is shaded in the plot.

`[p,h] = capaplot(data,specs)` additionally returns handles to the plot elements in `h`.

Example

Simulate a sample from a process with a mean of 3 and a standard deviation of 0.005:

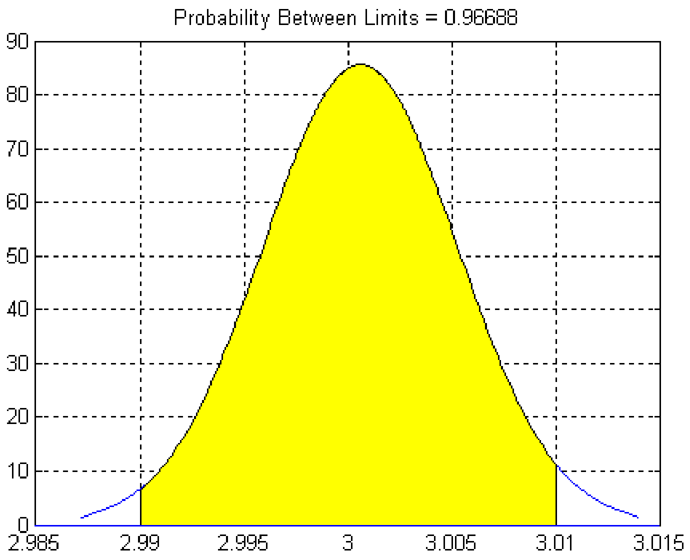
```
data = normrnd(3,0.005,100,1);
```

Compute capability indices if the process has an upper specification limit of 3.01 and a lower specification limit of 2.99:

```
S = capability(data,[2.99 3.01])
S =
       mu: 3.0006
    sigma: 0.0047
        P: 0.9669
       Pl: 0.0116
       Pu: 0.0215
       Cp: 0.7156
      Cpl: 0.7567
      Cpu: 0.6744
      Cpk: 0.6744
```

Visualize the specification and process widths:

```
capaplot(data,[2.99 3.01]);
grid on
```



See Also `capability`, `histfit`

Purpose

Read case names from file

Syntax

```
names = caseread(filename)
names = caseread
```

Description

`names = caseread(`*`filename`*`)` reads the contents of *`filename`* and returns a string matrix of `names`. *`filename`* is the name of a file in the current directory, or the complete path name of any file elsewhere. `caseread` treats each line as a separate case.

`names = caseread` displays the **Select File to Open** dialog box for interactive selection of the input file.

Example

Read the file `months.dat` created using the function `casewrite` on the next page.

```
type months.dat

January
February
March
April
May

names = caseread('months.dat')
names =
January
February
March
April
May
```

See Also

`tblread`, `gname`, `casewrite`, `tdfread`

Purpose

Write case names to file

Syntax

```
casewrite(strmat,filename)
casewrite(strmat)
```

Description

`casewrite(strmat,`*`filename`*`)` writes the contents of string matrix `strmat` to *`filename`*. Each row of `strmat` represents one case name. *`filename`* is the name of a file in the current directory, or the complete path name of any file elsewhere. `casewrite` writes each name to a separate line in *`filename`*.

`casewrite(strmat)` displays the **Select File to Write** dialog box for interactive specification of the output file.

Example

```
strmat = str2mat('January','February',...
                 'March','April','May')
strmat =
January
February
March
April
May

casewrite(strmat,'months.dat')
type months.dat

January
February
March
April
May
```

See Also

`gname`, `caseread`, `tblwrite`, `tdfread`

Purpose

Generate central composite design

Syntax

```
D = ccdesign(nfactors)
[D,blk] = ccdesign(nfactors)
[...] = ccdesign(nfactors,param1,val1,param2,val2,...)
```

Description

`D = ccdesign(nfactors)` generates a central composite design for `nfactors` factors. The output matrix `D` is `n`-by-`nfactors`, where `n` is the number of points in the design. Each row represents one run of the design, and it has the settings of all factors for that run. Factor values are normalized so that the cube points take values between `-1` and `1`.

`[D,blk] = ccdesign(nfactors)` requests a blocked design. The output vector `blk` is a vector of block numbers. Blocks are groups of runs that are to be measured under similar conditions (for example, on the same day). Blocked designs minimize the effect of between-block differences on the parameter estimates.

`[...] = ccdesign(nfactors,`*`param1`*`,`*`val1`*`,`*`param2`*`,`*`val2`*`,...)` enables you to specify additional parameters and their values. Valid parameters are:

`'center'`	Number of center points:	
	Integer	Specific number of center points to include
	`'uniform'`	Number of center points is selected to give uniform precision
	`'orthogonal'`	Number of center points is selected to give an orthogonal design (default)

'fraction'	Fraction of full factorial for cube portion expressed as an exponent of 1/2. For example:	
	0	Whole design
	1	1/2 fraction
	2	1/4 fraction
'type'	Either 'inscribed', 'circumscribed', or 'faced'	
'blocksize'	Maximum number of points allowed in a block.	

See Also

bbdesign, cordexch, rowexch

Purpose

Cumulative distribution function for specified distribution

Syntax

```
Y = cdf(name,X,A)
Y = cdf(name,X,A,B)
Y = cdf(name,X,A,B,C)
```

Description

`Y = cdf(name,X,A)` computes the cumulative distribution function for the one-parameter family of distributions specified by `name`. Parameter values for the distribution are given in `A`. The cumulative distribution function is evaluated at the values in `X` and its values are returned in `Y`.

If `X` and `A` are arrays, they must be the same size. If `X` is a scalar, it is expanded to a constant matrix the same size as `A`. If `A` is a scalar, it is expanded to a constant matrix the same size as `X`.

`Y` is the common size of `X` and `A` after any necessary scalar expansion.

`Y = cdf(name,X,A,B)` computes the cumulative distribution function for two-parameter families of distributions, where parameter values are given in `A` and `B`.

If `X`, `A`, and `B` are arrays, they must be the same size. If `X` is a scalar, it is expanded to a constant matrix the same size as `A` and `B`. If either `A` or `B` are scalars, they are expanded to constant matrices the same size as `X`.

`Y` is the common size of `X`, `A`, and `B` after any necessary scalar expansion.

`Y = cdf(name,X,A,B,C)` computes the cumulative distribution function for three-parameter families of distributions, where parameter values are given in `A`, `B`, and `C`.

If `X`, `A`, `B`, and `C` are arrays, they must be the same size. If `X` is a scalar, it is expanded to a constant matrix the same size as `A`, `B`, and `C`. If any of `A`, `B` or `C` are scalars, they are expanded to constant matrices the same size as `X`.

`Y` is the common size of `X`, `A`, `B` and `C` after any necessary scalar expansion.

Acceptable strings for `name` are:

- `'beta'` (Beta distribution)
- `'bino'` (Binomial distribution)
- `'chi2'` (Chi-square distribution)
- `'exp'` (Exponential distribution)
- `'ev'` (Extreme value distribution)
- `'f'` (*F* distribution)
- `'gam'` (Gamma distribution)
- `'gev'` (Generalized extreme value distribution)
- `'gp'` (Generalized Pareto distribution)
- `'geo'` (Geometric distribution)
- `'hyge'` (Hypergeometric distribution)
- `'logn'` (Lognormal distribution)
- `'nbin'` (Negative binomial distribution)
- `'ncf'` (Noncentral *F* distribution)
- `'nct'` (Noncentral *t*distribution)
- `'ncx2'` (Noncentral chi-square distribution)
- `'norm'` (Normal distribution)
- `'poiss'` (Poisson distribution)
- `'rayl'` (Rayleigh distribution)
- `'t'` (*t* distribution)
- `'unif'` (Uniform distribution)
- `'unid'` (Discrete uniform distribution)
- `'wbl'` (Weibull distribution)

Examples

```
p = cdf('Normal',-2:2,0,1)
p =
  0.0228  0.1587  0.5000  0.8413  0.9772

p = cdf('Poisson',0:5,1:6)
p =
  0.3679  0.4060  0.4232  0.4335  0.4405  0.4457
```

See Also

`mle`, `random`, `icdf`, `pdf`

cdf (piecewisedistribution)

Purpose

Cumulative distribution function for piecewise distribution

Syntax

```
P = cdf(obj,X)
```

Description

`P = cdf(obj,X)` returns an array `P` of values of the cumulative distribution function for the piecewise distribution object `obj`, evaluated at the values in the array `X`.

Example

Fit Pareto tails to a *t* distribution at cumulative probabilities 0.1 and 0.9:

```
t = trnd(3,100,1);
obj = paretotails(t,0.1,0.9);
[p,q] = boundary(obj)
p =
    0.1000
    0.9000
q =
   -1.7766
    1.8432

cdf(obj,q)
ans =
    0.1000
    0.9000
```

See Also

`paretotails`, `pdf (piecewisedistribution)`, `icdf (piecewisedistribution)`

Purpose

Plot of empirical cumulative distribution function

Syntax

```
cdfplot(X)
H = cdfplot(X)
[h,stats] = cdfplot(X)
```

Description

`cdfplot(X)` displays a plot of the empirical cumulative distribution function (cdf) for the data in the vector `X`. The empirical cdf $F(x)$ is defined as the proportion of `X` values less than or equal to x.

This plot, like those produced by `hist` and `normplot`, is useful for examining the distribution of a sample of data. You can overlay a theoretical cdf on the same plot to compare the empirical distribution of the sample to the theoretical distribution.

The `kstest`, `kstest2`, and `lillietest` functions compute test statistics that are derived from the empirical cdf. You may find the empirical cdf plot produced by `cdfplot` useful in helping you to understand the output from those functions.

`H = cdfplot(X)` returns a handle to the cdf curve.

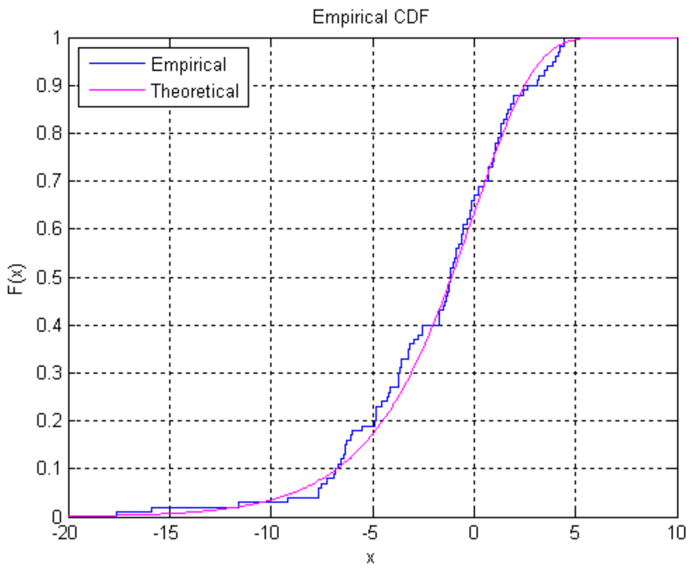
`[h,stats] = cdfplot(X)` also returns a `stats` structure with the following fields.

Field	Contents
`stats.min`	Minimum value
`stats.max`	Maximum value
`stats.mean`	Sample mean
`stats.median`	Sample median (50th percentile)
`stats.std`	Sample standard deviation

Example

The following example compares the empirical cdf for a sample from an extreme value distribution with a plot of the cdf for the sampling distribution. In practice, the sampling distribution would be unknown, and would be chosen to match the empirical cdf.

```
y = evrnd(0,3,100,1);
cdfplot(y)
hold on
x = -20:0.1:10;
f = evcdf(x,0,3);
plot(x,f,'m')
legend('Empirical','Theoretical','Location','NW')
```



See Also

ecdf, hist, kstest, kstest2, lillietest, normplot

Purpose

Chi-square cumulative distribution function

Syntax

```
P = chi2cdf(X,V)
```

Description

`P = chi2cdf(X,V)` computes the chi-square cdf at each of the values in `X` using the corresponding parameters in `V`. `X` and `V` can be vectors, matrices, or multidimensional arrays that have the same size. A scalar input is expanded to a constant array with the same dimensions as the other input.

The degrees of freedom parameters in `V` must be positive integers, and the values in `X` must lie on the interval `[0 Inf]`.

The χ^2 cdf for a given value x and degrees-of-freedom ν is

$$p = F(x|\nu) = \int_0^x \frac{t^{(\nu-2)/2} e^{-t/2}}{2^{\nu/2}\,\Gamma(\nu/2)}\,dt$$

where $\Gamma(\cdot)$ is the Gamma function.

The chi-square density function with ν degrees-of-freedom is the same as the gamma density function with parameters $\nu/2$ and 2.

Examples

```
probability = chi2cdf(5,1:5)
probability =
  0.9747  0.9179  0.8282  0.7127  0.5841

probability = chi2cdf(1:5,1:5)
probability =
  0.6827  0.6321  0.6084  0.5940  0.5841
```

See Also

`cdf`, `chi2inv`, `chi2pdf`, `chi2rnd`, `chi2stat`

Purpose

Chi-square goodness-of-fit test

Syntax

```
h = chi2gof(x)
[h,p] = chi2gof(...)
[h,p,stats] = chi2gof(...)
[...] = chi2gof(X,name1,val1,name2,val2,...)
```

Description

`h = chi2gof(x)` performs a chi-square goodness-of-fit test of the default null hypothesis that the data in vector `x` are a random sample from a normal distribution with mean and variance estimated from `x`, against the alternative that the data are not normally distributed with the estimated mean and variance. The result `h` is `1` if the null hypothesis can be rejected at the 5% significance level. The result `h` is `0` if the null hypothesis cannot be rejected at the 5% significance level.

The null distribution can be changed from a normal distribution to an arbitrary discrete or continuous distribution. See the syntax for specifying optional argument name/value pairs below.

The test is performed by grouping the data into bins, calculating the observed and expected counts for those bins, and computing the chi-square test statistic

$$\chi^2 = \sum_{i=1}^{N} (O_i - E_i)^2 / E_i$$

where O_i are the observed counts and E_i are the expected counts. The statistic has an approximate chi-square distribution when the counts are sufficiently large. Bins in either tail with an expected count less than 5 are pooled with neighboring bins until the count in each extreme bin is at least 5. If bins remain in the interior with counts less than 5, `chi2gof` displays a warning. In this case, you should use fewer bins, or provide bin centers or edges, to increase the expected counts in all bins. (See the syntax for specifying optional argument name/value pairs below.) `chi2gof` sets the number of bins, `nbins`, to 10 by default, and compares the test statistic to a chi-square distribution with `nbins – 3` degrees of freedom to take into account the two estimated parameters.

`[h,p] = chi2gof(...)` also returns the *p*-value of the test, `p`. The *p*-value is the probability, under assumption of the null hypothesis, of observing the given statistic or one more extreme.

`[h,p,stats] = chi2gof(...)` also returns a structure `stats` with the following fields:

- `chi2stat` — The chi-square statistic
- `df` — Degrees of freedom
- `edges` — Vector of bin edges after pooling
- `O` — Observed count in each bin
- `E` — Expected count in each bin

`[...] = chi2gof(X,`*`name1`*`,`*`val1`*`,`*`name2`*`,`*`val2`*`,...)` specifies optional argument name/value pairs chosen from the following lists. Argument names are case insensitive and partial matches are allowed.

The following name/value pairs control the initial binning of the data before pooling. You should not specify more than one of these options.

- `'nbins'` — The number of bins to use. Default is 10.
- `'ctrs'` — A vector of bin centers
- `'edges'` — A vector of bin edges

The following name/value pairs determine the null distribution for the test. You should not specify both `'cdf'` and `'expected'`.

- `'cdf'` — A fully specified cumulative distribution function. This can be a function name or function handle, and the function must take `x` as its only argument. Alternately, you can provide a cell array whose first element is a function name or handle, and whose later elements are parameter values, one per cell. The function must take `x` as its first argument, and other parameters as later arguments.

- `'expected'` — A vector with one element per bin specifying the expected counts for each bin
- `'nparams'` — The number of estimated parameters; used to adjust the degrees of freedom to be `nbins` – `1` – `nparams`, where `nbins` is the number of bins

If your `'cdf'` or `'expected'` input depends on estimated parameters, you should use `'nparams'` to ensure that the degrees of freedom for the test is correct. If `'cdf'`is a cell array, the default value of `'nparams'` is the number of parameters in the array; otherwise the default is 0.

The following name/value pairs control other aspects of the test.

- `'emin'` — The minimum allowed expected value for a bin; any bin in either tail having an expected value less than this amount is pooled with a neighboring bin. Use the value 0 to prevent pooling. The default is 5.
- `'frequency'` — A vector the same length as `x` containing the frequency of the corresponding xvalues
- `'alpha'` — Significance level for the test. The default is 0.05.

Examples

Example 1

Equivalent ways to test against an unspecified normal distribution with estimated parameters:

```
x = normrnd(50,5,100,1);

[h,p] = chi2gof(x)
h =
     0
p =
    0.7532

[h,p] = chi2gof(x,'cdf',@(z)normcdf(z,mean(x),std(x)),'nparams',2)
h =
     0
```

```
p =
    0.7532

[h,p] = chi2gof(x,'cdf',{@normcdf,mean(x),std(x)})
h =
     0
p =
    0.7532
```

Example 2

Test against the standard normal:

```
x = randn(100,1);

[h,p] = chi2gof(x,'cdf',@normcdf)
h =
     0
p =
    0.9443
```

Example 3

Test against the standard uniform:

```
x = rand(100,1);

n = length(x);
edges = linspace(0,1,11);
expectedCounts = n * diff(edges);
[h,p,st] = chi2gof(x,'edges',edges,...
                     'expected',expectedCounts)
h =
     0
p =
    0.3191
st =
    chi2stat: 10.4000
          df: 9
```

```
        edges: [1x11 double]
            O: [6 11 4 12 15 8 14 9 11 10]
            E: [1x10 double]
```

Example 4

Test against the Poisson distribution by specifying observed and expected counts:

```
bins = 0:5;
obsCounts = [6 16 10 12 4 2];
n = sum(obsCounts);
lambdaHat = sum(bins.*obsCounts)/n;
expCounts = n*poisspdf(bins,lambdaHat);

[h,p,st] = chi2gof(bins,'ctrs',bins,...
                        'frequency',obsCounts, ...
                        'expected',expCounts,...
                        'nparams',1)
h =
     0
p =
    0.4654
st =
    chi2stat: 2.5550
          df: 3
       edges: [1x6 double]
           O: [6 16 10 12 6]
           E: [7.0429 13.8041 13.5280 8.8383 6.0284]
```

See Also `crosstab`, `chi2cdf`, `kstest`, `lillietest`

Purpose

Inverse of chi-square cumulative distribution function

Syntax

```
X = chi2inv(P,V)
```

Description

`X = chi2inv(P,V)` computes the inverse of the chi-square cdf with parameters specified by `V` for the corresponding probabilities in `P`. `P` and `V` can be vectors, matrices, or multidimensional arrays that have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs.

The degrees of freedom parameters in `V` must be positive integers, and the values in `P` must lie in the interval [0 1].

The inverse chi-square cdf for a given probability *p* and ν degrees of freedom is

$$x = F^{-1}(p|\nu) = \{x:F(x|\nu) = p\}$$

where

$$p = F(x|\nu) = \int_0^x \frac{t^{(\nu-2)/2}e^{-t/2}}{2^{\nu/2}\,\Gamma(\nu/2)}dt$$

and $\Gamma(\cdot)$ is the Gamma function. Each element of output `X` is the value whose cumulative probability under the chi-square cdf defined by the corresponding degrees of freedom parameter in `V` is specified by the corresponding value in `P`.

Examples

Find a value that exceeds 95% of the samples from a chi-square distribution with 10 degrees of freedom.

```
x = chi2inv(0.95,10)
x =
  18.3070
```

You would observe values greater than 18.3 only 5% of the time by chance.

See Also `chi2gof`, `chi2pdf`, `chi2rnd`, `chi2stat`, `icdf`

Purpose

Chi-square probability density function

Syntax

```
Y = chi2pdf(X,V)
```

Description

`Y = chi2pdf(X,V)` computes the chi-square pdf at each of the values in `X` using the corresponding parameters in `V`. `X` and `V` can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of the output `Y`. A scalar input is expanded to a constant array with the same dimensions as the other input.

The degrees of freedom parameters in `V` must be positive integers, and the values in `X` must lie on the interval `[0 Inf]`.

The chi-square pdf for a given value x and ν degrees of freedom is

$$y = f(x|\nu) = \frac{x^{(\nu-2)/2}e^{-x/2}}{2^{\nu/2}\Gamma(\nu/2)}$$

where $\Gamma(\cdot)$ is the Gamma function.

If x is standard normal, then x^2 is distributed chi-square with one degree of freedom. If x_1, x_2, ..., x_n are n independent standard normal observations, then the sum of the squares of the x's is distributed chi-square with n degrees of freedom (and is equivalent to the gamma density function with parameters ν/2 and 2).

Examples

```
nu = 1:6;
x = nu;
y = chi2pdf(x,nu)
y =
  0.2420  0.1839  0.1542  0.1353  0.1220  0.1120
```

The mean of the chi-square distribution is the value of the degrees of freedom parameter, `nu`. The above example shows that the probability density of the mean falls as `nu` increases.

See Also

`chi2gof`, `chi2inv`, `chi2rnd`, `chi2stat`, `pdf`

Purpose

Random numbers from chi-square distribution

Syntax

```
R = chi2rnd(V)
R = chi2rnd(V,u)
R = chi2rnd(V,m,n)
```

Description

`R = chi2rnd(V)` generates random numbers from the chi-square distribution with degrees of freedom parameters specified by `V`. `V` can be a vector, a matrix, or a multidimensional array. `R` is the same size as `V`.

`R = chi2rnd(V,u)` generates an array `R` of size `u` containing random numbers from the chi-square distribution with degrees of freedom parameters specified by `V`, where `u` is a row vector. If `u` is a 1-by-2 vector, `R` is a matrix with `u(1)` rows and `u(2)` columns. If `u` is 1-by-n, `R` is an n-dimensional array.

`R = chi2rnd(V,m,n)` generates an `m`-by-`n` matrix containing random numbers from the chi-square distribution with degrees of freedom parameter `V`.

Example

Note that the first and third commands are the same, but are different from the second command.

```
r = chi2rnd(1:6)
r =
    0.0037   3.0377   7.8142   0.9021   3.2019   9.0729

r = chi2rnd(6,[1 6])
r =
    6.5249   2.6226   12.2497   3.0388   6.3133   5.0388

r = chi2rnd(1:6,1,6)
r =
  0.7638   6.0955   0.8273   3.2506   1.5469   10.9197
```

See Also

`chi2gof`, `chi2inv`, `chi2pdf`, `chi2stat`

Purpose

Mean and variance of chi-square distribution

Syntax

```
[M,V] = chi2stat(NU)
```

Description

`[M,V] = chi2stat(NU)` returns the mean of and variance for the chi-square distribution with degrees of freedom parameters specified by `NU`.

The mean of the chi-square distribution is ν, the degrees of freedom parameter, and the variance is 2ν.

Example

```
nu = 1:10;
nu = nu'*nu;
[m,v] = chi2stat(nu)
m =
 1   2   3   4   5   6   7   8   9  10
 2   4   6   8  10  12  14  16  18  20
 3   6   9  12  15  18  21  24  27  30
 4   8  12  16  20  24  28  32  36  40
 5  10  15  20  25  30  35  40  45  50
 6  12  18  24  30  36  42  48  54  60
 7  14  21  28  35  42  49  56  63  70
 8  16  24  32  40  48  56  64  72  80
 9  18  27  36  45  54  63  72  81  90
10  20  30  40  50  60  70  80  90 100

v =
 2   4   6   8  10  12  14  16  18  20
 4   8  12  16  20  24  28  32  36  40
 6  12  18  24  30  36  42  48  54  60
 8  16  24  32  40  48  56  64  72  80
10  20  30  40  50  60  70  80  90 100
12  24  36  48  60  72  84  96 108 120
14  28  42  56  70  84  98 112 126 140
16  32  48  64  80  96 112 128 144 160
18  36  54  72  90 108 126 144 162 180
20  40  60  80 100 120 140 160 180 200
```

See Also `chi2gof`, `chi2inv`, `chi2pdf`, `chi2rnd`

Purpose

Child nodes of tree node

Syntax

```
C = children(t)
C = children(t,nodes)
```

Description

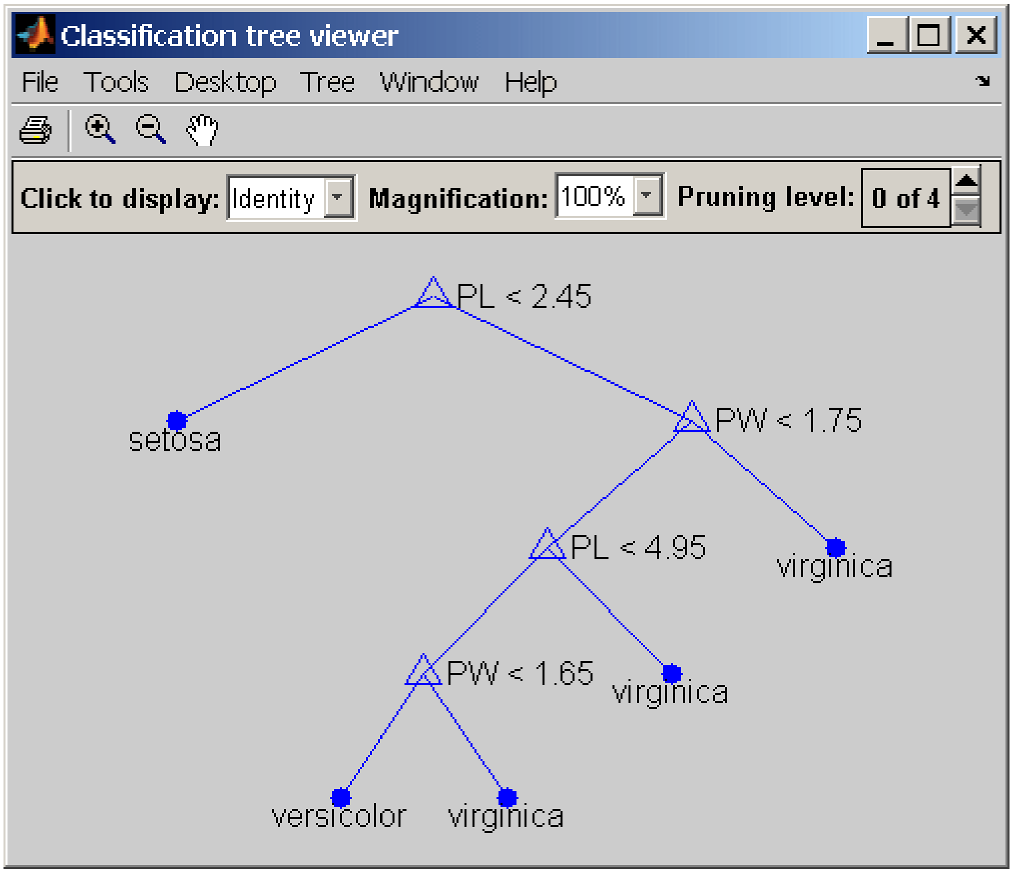
`C = children(t)` returns an n-by-2 array `C` containing the numbers of the child nodes for each node in the tree `t`, where n is the number of nodes. Leaf nodes have child node `0`.

`C = children(t,nodes)` takes a vector `nodes` of node numbers and returns the children for the specified nodes.

Example

Create a classification tree for Fisher's iris data:

```
load fisheriris;

t = classregtree(meas,species,'names',{'SL' 'SW' 'PL' 'PW'})
t =
Decision tree for classification
1  if PL<2.45 then node 2 else node 3
2  class = setosa
3  if PW<1.75 then node 4 else node 5
4  if PL<4.95 then node 6 else node 7
5  class = virginica
6  if PW<1.65 then node 8 else node 9
7  class = virginica
8  class = versicolor
9  class = virginica

view(t)
```


Classification tree viewer
File Tools Desktop Tree Window Help

Click to display: Identity Magnification: 100% Pruning level: 0 of 4

```
C = children(t)
C =
     2     3
     0     0
     4     5
     6     7
     0     0
     8     9
     0     0
     0     0
     0     0
```

Reference

[1] Breiman, L., et al., *Classification and Regression Trees*, Chapman & Hall, Boca Raton, 1993.

See Also

`classregtree`, `numnodes`, `parent`

Purpose

Cholesky-like decomposition for covariance matrix

Syntax

```
T = cholcov(SIGMA)
[T,num] = cholcov(SIGMA)
[T,num] = cholcov(SIGMA,0)
```

Description

`T = cholcov(SIGMA)` computes `T` such that `SIGMA = T'*T`. `SIGMA` must be square, symmetric, and positive semi-definite. If `SIGMA` is positive definite, then `T` is the square, upper triangular Cholesky factor. If `SIGMA` is not positive definite, `T` is computed from an eigenvalue decomposition of `SIGMA`. `T` is not necessarily triangular or square in this case. Any eigenvectors whose corresponding eigenvalue is close to zero (within a small tolerance) are omitted. If any remaining eigenvectors are negative, `T` is empty.

`[T,num] = cholcov(SIGMA)` returns the number `num` of negative eigenvalues of `SIGMA`, and `T` is empty if `num` is positive. If `num` is zero, `SIGMA` is positive semi-definite. If `SIGMA` is not square and symmetric, `num` is `NaN` and `T` is empty.

`[T,num] = cholcov(SIGMA,0)` returns `num` equal to zero if `SIGMA` is positive definite, and `T` is the Cholesky factor. If `SIGMA` is not positive definite, `num` is a positive integer and `T` is empty. `[...] = cholcov(SIGMA,1)` is equivalent to `[...] = cholcov(SIGMA)`.

Example

The following 4-by-4 covariance matrix is rank-deficient:

```
C1 = [2 1 1 2;1 2 1 2;1 1 2 2;2 2 2 3]
C1 =
     2     1     1     2
     1     2     1     2
     1     1     2     2
     2     2     2     3
rank(C1)
ans =
     3
```

Use `cholcov` to factor `C1`:

```
T = cholcov(C1)
T =
   -0.2113    0.7887   -0.5774         0
    0.7887   -0.2113   -0.5774         0
    1.1547    1.1547    1.1547    1.7321

C2 = T'*T
C2 =
    2.0000   1.0000   1.0000   2.0000
    1.0000   2.0000   1.0000   2.0000
    1.0000   1.0000   2.0000   2.0000
    2.0000   2.0000   2.0000   3.0000
```

Use T to generate random data with the specified covariance:

```
C3 = cov(randn(1e6,3)*T)
C3 =
    1.9973    0.9982    0.9995    1.9975
    0.9982    1.9962    0.9969    1.9956
    0.9995    0.9969    1.9980    1.9972
    1.9975    1.9956    1.9972    2.9951
```

See Also

chol

classcount

Purpose

Class counts at tree nodes

Syntax

```
P = classcount(t)
P = classcount(t,nodes)
```

Description

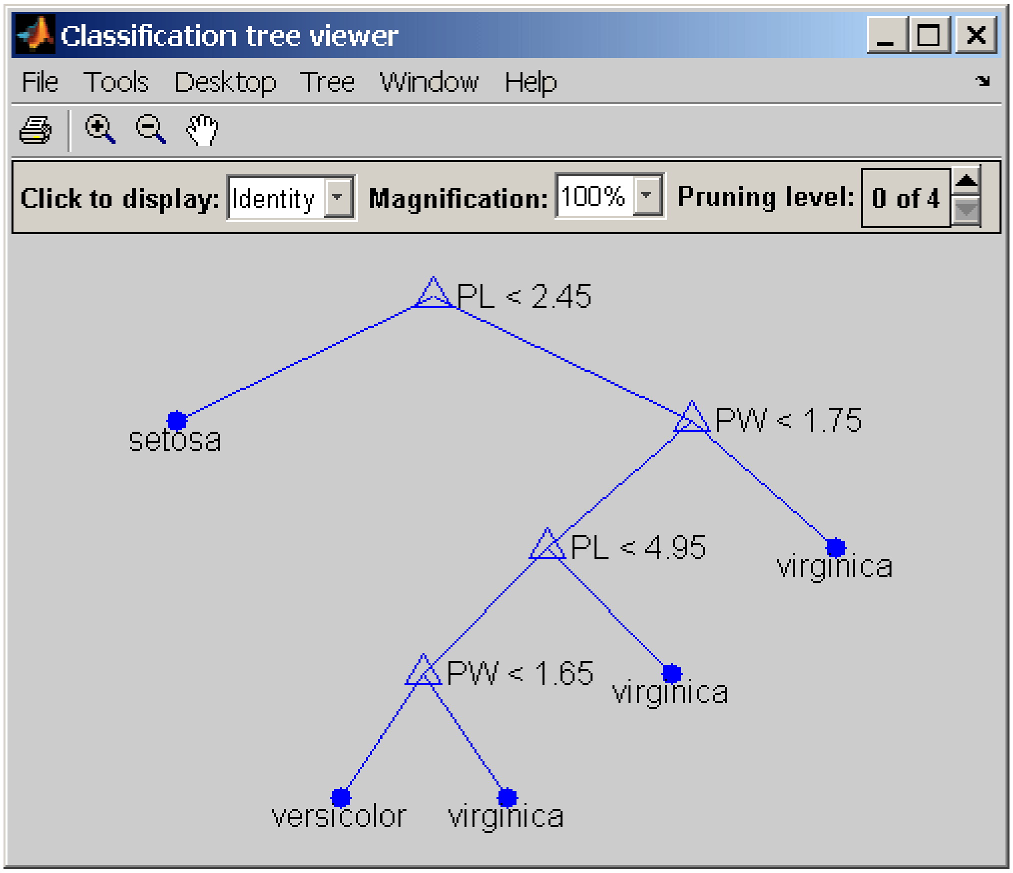
`P = classcount(t)` returns an *n*-by-*m* array `P` of class counts for the nodes in the classification tree `t`, where *n* is the number of nodes and *m* is the number of classes. For any node number `i`, the class counts `P(i,:)` are counts of observations (from the data used in fitting the tree) from each class satisfying the conditions for node `i`.

`P = classcount(t,nodes)` takes a vector `nodes` of node numbers and returns the class counts for the specified nodes.

Example

Create a classification tree for Fisher's iris data:

```
load fisheriris;

t = classregtree(meas,species,'names',{'SL' 'SW' 'PL' 'PW'})
t =
Decision tree for classification
1  if PL<2.45 then node 2 else node 3
2  class = setosa
3  if PW<1.75 then node 4 else node 5
4  if PL<4.95 then node 6 else node 7
5  class = virginica
6  if PW<1.65 then node 8 else node 9
7  class = virginica
8  class = versicolor
9  class = virginica

view(t)
```



```
P = classcount(t)
P =
    50    50    50
    50     0     0
     0    50    50
     0    49     5
     0     1    45
     0    47     1
     0     2     4
     0    47     0
     0     0     1
```

Reference

[1] Breiman, L., et al., *Classification and Regression Trees*, Chapman & Hall, Boca Raton, 1993.

See Also

`classregtree`, `numnodes`

Purpose

Discriminant analysis

Syntax

```
class = classify(sample,training,group)
class = classify(sample,training,group,type)
class = classify(sample,training,group,type,prior)
[class,err] = classify(...)
[class,err,POSTERIOR] = classify(...)
[class,err,POSTERIOR,logp] = classify(...)
[class,err,POSTERIOR,logp,coeff] = classify(...)
```

Description

`class = classify(sample,training,group)` classifies each row of the data in `sample` into one of the groups in `training`. `sample` and `training` must be matrices with the same number of columns. `group` is a grouping variable for `training`. Its unique values define groups; each element defines the group to which the corresponding row of `training` belongs. `group` can be a categorical variable, a numeric vector, a string array, or a cell array of strings. `training` and `group` must have the same number of rows. (See "Grouped Data" on page 2-41.) `classify` treats `NaN`s or empty strings in `group` as missing values, and ignores the corresponding rows of `training`. The output `class` indicates the group to which each row of `sample` has been assigned, and is of the same type as `group`.

`class = classify(sample,training,group,`*`type`*`)` allows you to specify the type of discriminant function. *`type`* is one of:

- `'linear'` — Fits a multivariate normal density to each group, with a pooled estimate of covariance. This is the default.
- `'diaglinear'` — Similar to `'linear'`, but with a diagonal covariance matrix estimate (naive Bayes classifiers).
- `'quadratic'` — Fits multivariate normal densities with covariance estimates stratified by group.
- `'diagquadratic'` — Similar to `'quadratic'`, but with a diagonal covariance matrix estimate (naive Bayes classifiers).

- `'mahalanobis'` — Uses Mahalanobis distances with stratified covariance estimates.

`class = classify(sample,training,group,`*`type`*`,`*`prior`*`)` allows you to specify prior probabilities for the groups. *`prior`* is one of:

- A numeric vector the same length as the number of unique values in `group` (or the number of levels defined for `group`, if `group` is categorical). If `group` is numeric or categorical, the order of *`prior`* must correspond to the ordered values in `group`, or, if `group` contains strings, to the order of first occurrence of the values in `group`.
- A 1-by-1 structure with fields:
  - `prob` — A numeric vector.
  - `group` — Of the same type as `group`, containing unique values indicating the groups to which the elements of `prob` correspond.

  As a structure, *`prior`* can contain groups that do not appear in `group`. This can be useful if `training` is a subset a larger training set. `classify` ignores any groups that appear in the structure but not in the `group` array.
- The string `'empirical'`, indicating that group prior probabilities should be estimated from the group relative frequencies in `training`.

`prior` defaults to a numeric vector of equal probabilities, i.e., a uniform distribution. *`prior`* is not used for discrimination by Mahalanobis distance, except for error rate calculation.

`[class,err] = classify(...)` also returns an estimate `err` of the misclassification error rate based on the `training` data. `classify` returns the apparent error rate, i.e., the percentage of observations in `training` that are misclassified, weighted by the prior probabilities for the groups.

`[class,err,POSTERIOR] = classify(...)` also returns a matrix `POSTERIOR` of estimates of the posterior probabilities that the jth training group was the source of the ith sample observation, i.e., *Pr*(*group j*|*obs i*). `POSTERIOR` is not computed for Mahalanobis discrimination.

`[class,err,POSTERIOR,logp] = classify(...)` also returns a vector `logp` containing estimates of the logarithms of the unconditional predictive probability density of the sample observations, *p*(*obs i*) = sum of *p*(*obs i*|*group j*)*Pr*(*group j*) over all groups. `logp` is not computed for Mahalanobis discrimination.

`[class,err,POSTERIOR,logp,coeff] = classify(...)` also returns a structure array `coeff` containing coefficients of the boundary curves between pairs of groups. Each element `coeff(I,J)` contains information for comparing group `I` to group `J` in the following fields:

- `type` — Type of discriminant function, from the *`type`* input.
- `name1` — Name of the first group.
- `name2` — Name of the second group.
- `const` — Constant term of the boundary equation (K)
- `linear` — Linear coefficients of the boundary equation (L)
- `quadratic` — Quadratic coefficient matrix of the boundary equation (Q)

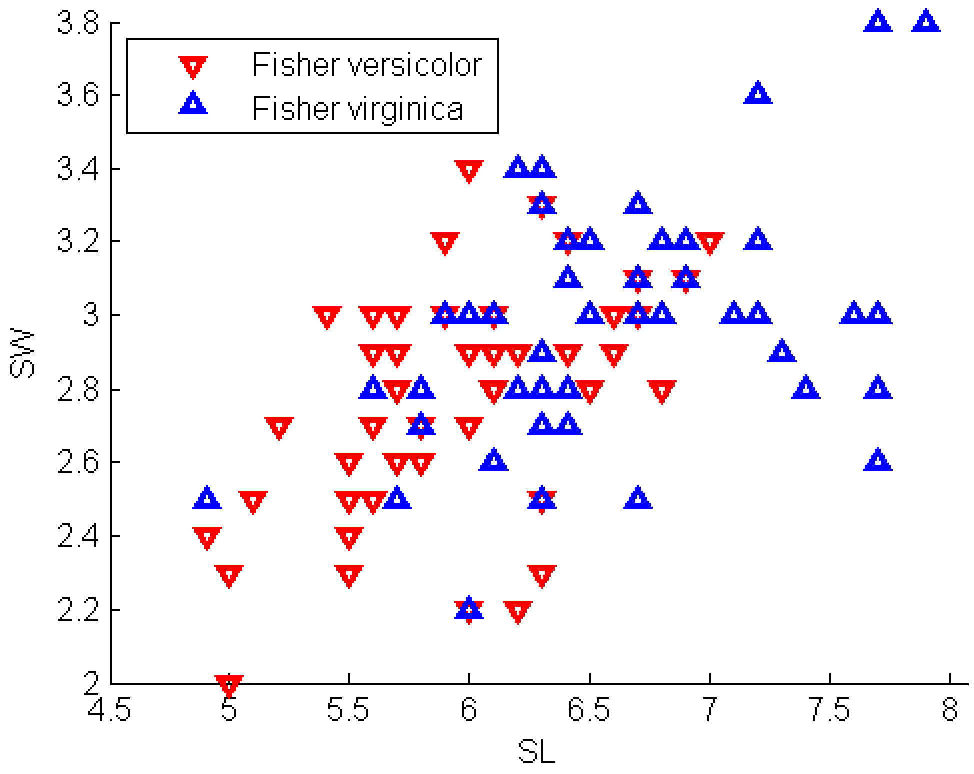
For the `'linear'` and `'diaglinear'` types, the `quadratic` field is absent, and a row `x` from the `sample` array is classified into group `I` rather than group `J` if `0 < K+x*L`. For the other types, `x` is classified into group `I` if `0 < K+x*L+x*Q*x'`.

Example

For training data, use Fisher's sepal measurements for iris versicolor and virginica:

```
load fisheriris
SL = meas(51:end,1);
SW = meas(51:end,2);
group = species(51:end);
h1 = gscatter(SL,SW,group,'rb','v^',[],'off');
```

```
set(h1,'LineWidth',2)
legend('Fisher versicolor','Fisher virginica',...
       'Location','NW')
```



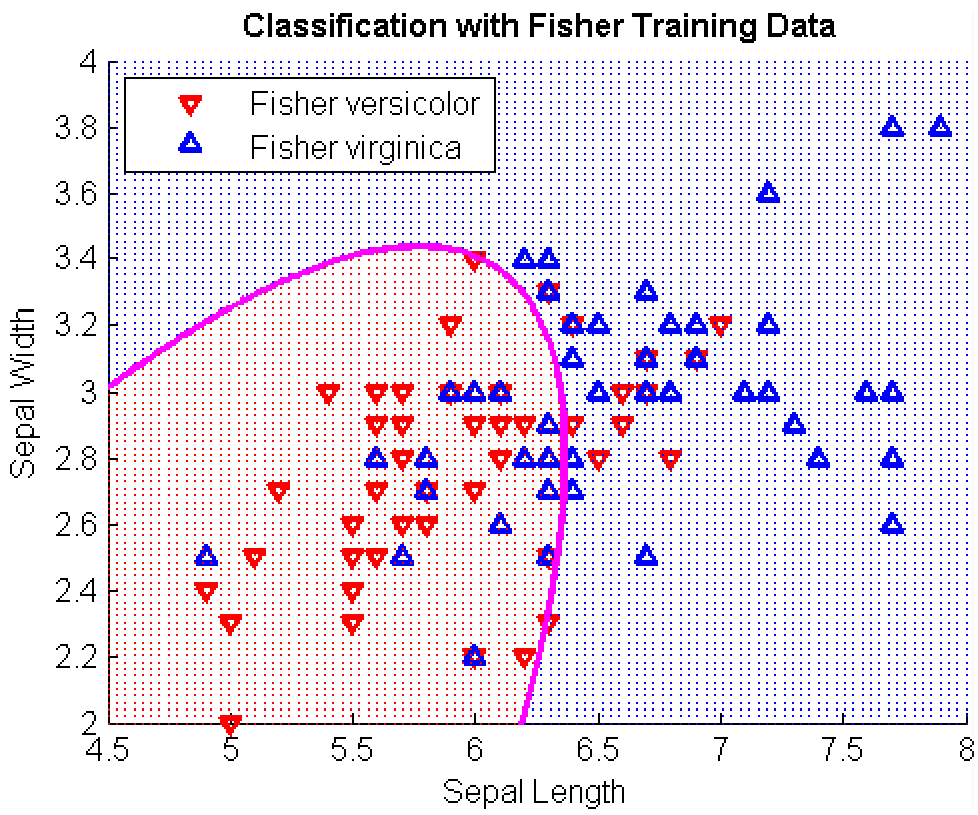
Classify a grid of measurements on the same scale:

```
[X,Y] = meshgrid(linspace(4.5,8),linspace(2,4));
X = X(:); Y = Y(:);
[C,err,P,logp,coeff] = classify([X Y],[SL SW],group,'quadratic');
```

Visualize the classification:

```
hold on;
```

```
gscatter(X,Y,C,'rb','.',1,'off');
K = coeff(1,2).const;
L = coeff(1,2).linear;
Q = coeff(1,2).quadratic;
f = sprintf('0 = %g+%g*x+%g*y+%g*x^2+%g*x.*y+%g*y.^2',...
            K,L,Q(1,1),Q(1,2)+Q(2,1),Q(2,2));
h2 = ezplot(f,[4.5 8 2 4]);
set(h2,'Color','m','LineWidth',2)
axis([4.5 8 2 4])
xlabel('Sepal Length')
ylabel('Sepal Width')
title('{\bf Classification with Fisher Training Data}')
```



See Also `mahal`, `treefit`

References

[1] Krzanowski, W. J., *Principles of Multivariate Analysis*, Oxford University Press, 1988.

[2] Seber, G.A.F., *Multivariate Observations*, Wiley, 1984.

Purpose

Class probabilities at tree nodes

Syntax

```
P = classprob(t)
P = classprob(t,nodes)
```

Description

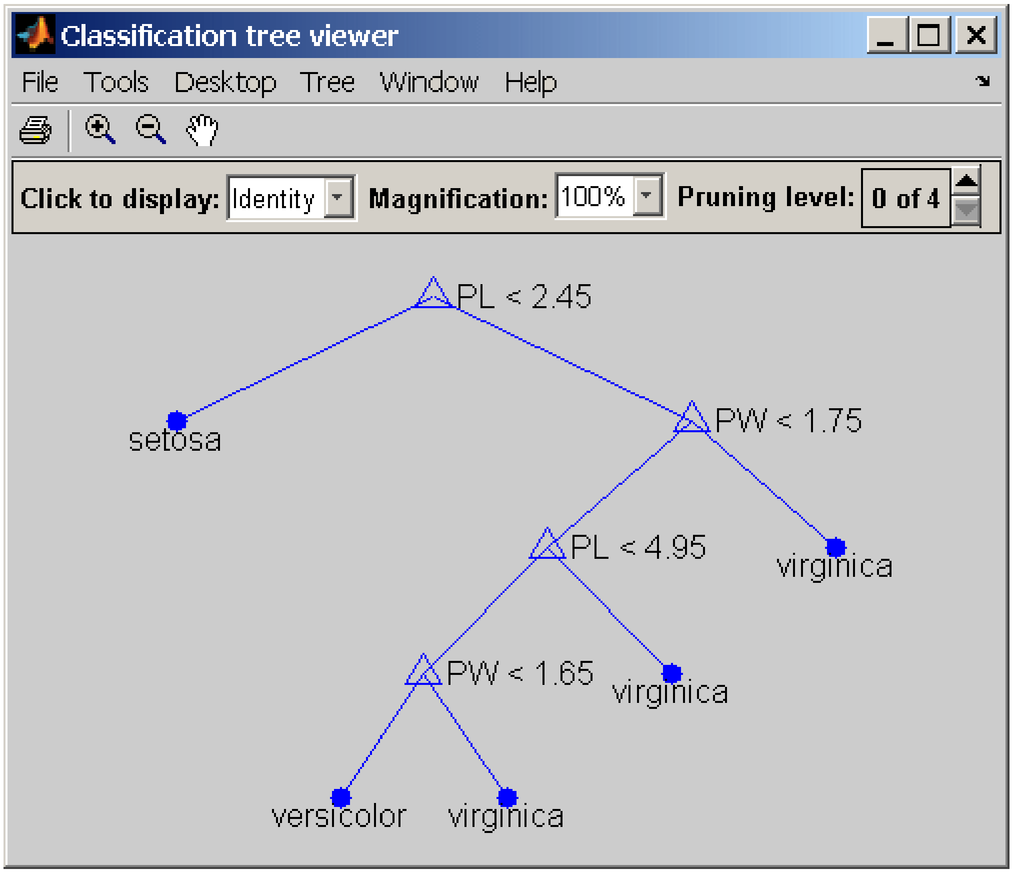
`P = classprob(t)` returns an *n*-by-*m* array `P` of class probabilities for the nodes in the classification tree `t`, where *n* is the number of nodes and *m* is the number of classes. For any node number `i`, the class probabilities `P(i,:)` are the estimated probabilities for each class for a point satisfying the conditions for node `i`.

`P = classprob(t,nodes)` takes a vector `nodes` of node numbers and returns the class probabilities for the specified nodes.

Example

Create a classification tree for Fisher's iris data:

```
load fisheriris;

t = classregtree(meas,species,'names',{'SL' 'SW' 'PL' 'PW'})
t =
Decision tree for classification
1  if PL<2.45 then node 2 else node 3
2  class = setosa
3  if PW<1.75 then node 4 else node 5
4  if PL<4.95 then node 6 else node 7
5  class = virginica
6  if PW<1.65 then node 8 else node 9
7  class = virginica
8  class = versicolor
9  class = virginica

view(t)
```



```
P = classprob(t)
P =
    0.3333    0.3333    0.3333
    1.0000         0         0
         0    0.5000    0.5000
         0    0.9074    0.0926
         0    0.0217    0.9783
         0    0.9792    0.0208
         0    0.3333    0.6667
         0    1.0000         0
         0         0    1.0000
```

Reference

[1] Breiman, L., et al., *Classification and Regression Trees*, Chapman & Hall, Boca Raton, 1993.

See Also

`classregtree`, `numnodes`

classregtree

Purpose

Construct classification and regression tree object

Syntax

```
t = classregtree(X,y)
t = classregtree(X,y,param1,val1,param2,val2)
```

Description

`t = classregtree(X,y)` creates a decision tree `t` for predicting the response `y` as a function of the predictors in the columns of `X`. `X` is an *n*-by-*m* matrix of predictor values. If `y` is a vector of *n* response values, `classregtree` performs regression. If `y` is a categorical variable, character array, or cell array of strings, `classregtree` performs classification. Either way, `t` is a binary tree where each branching node is split based on the values of a column of `X`. `NaN` values in `X` or `y` are taken as missing values, and observations with any missing values are not used in the fit.

`t = classregtree(X,y,`*`param1`*`,`*`val1`*`,`*`param2`*`,`*`val2`*`)` specifies optional parameter name/value pairs, as follows.

For all trees:

- `'categorical'` — Vector of indices of the columns of `X` that are to be treated as unordered (nominal) categorical variables.
- `'method'` — Either `'classification'` (default if `y` is text or a categorical variable) or `'regression'` (default if `y` is numeric).
- `'names'` — A cell array of names for the predictor variables, in the order in which they appear in the `X` from which the tree was created. (See `treefit`.)
- `'prune'` — `'on'` (default) to compute the full tree and the optimal sequence of pruned subtrees, or `'off'` for the full tree without pruning.
- `'splitmin'` — A number *k* such that impure nodes must have *k* or more observations to be split (default is `10`).

For classification trees only:

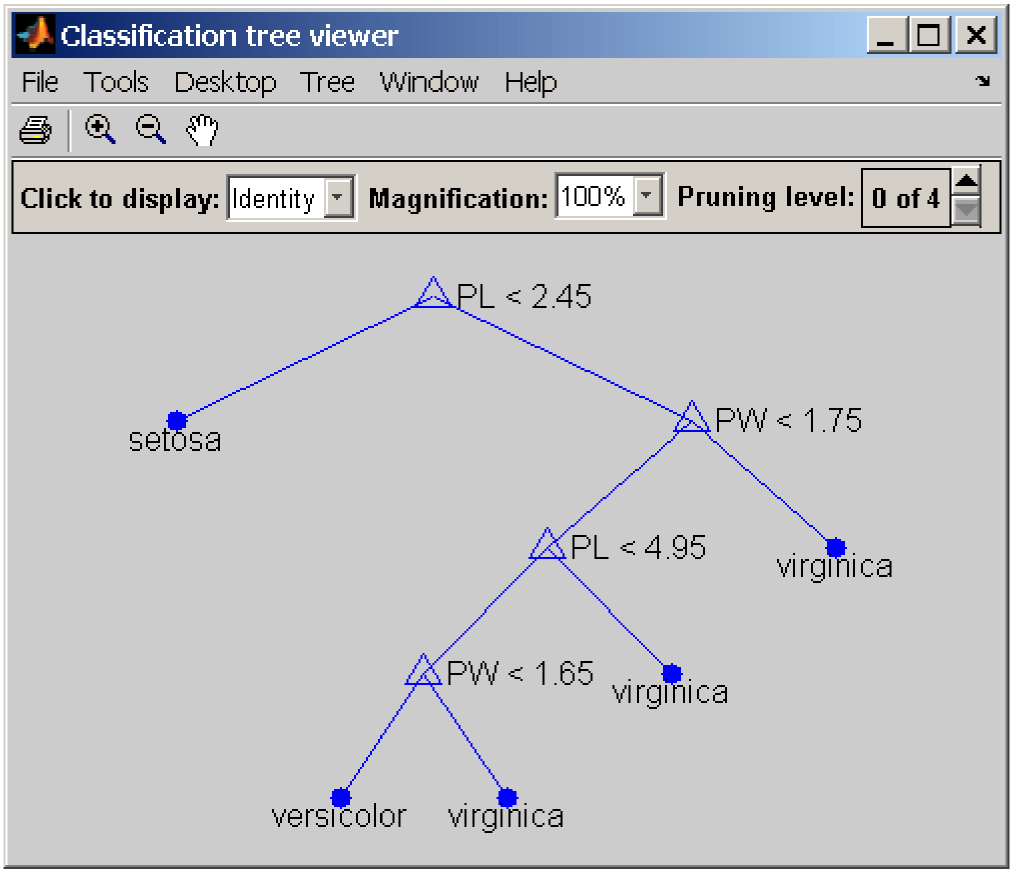
- 'cost' — Square matrix C, where C(i,j) is the cost of classifying a point into class j if its true class is i. (The default has C(i,j) = 1 if i ~= j, and C(i,j) = 0 if i = j.) Alternatively, this value can be a structure with two fields:
  - group — Containing the group names as a character array or cell array of strings
  - cost — Containing the cost matrix C
- 'splitcriterion' — Criterion for choosing a split. One of:
  - 'gdi' — For Gini's diversity index (default)
  - 'twoing' — For the twoing rule
  - 'deviance' — For maximum deviance reduction
- 'priorprob' — Prior probabilities for each class, specified as a vector (one value for each distinct group name) or as a structure with two fields:
  - group — Containing the group names as a character array or cell array of strings
  - prob — Containing a vector of corresponding probabilities

Example

Create a classification tree for Fisher's iris data:

```
load fisheriris;

t = classregtree(meas,species,'names',{'SL' 'SW' 'PL' 'PW'})
t =
Decision tree for classification
1  if PL<2.45 then node 2 else node 3
2  class = setosa
3  if PW<1.75 then node 4 else node 5
4  if PL<4.95 then node 6 else node 7
5  class = virginica
6  if PW<1.65 then node 8 else node 9
7  class = virginica
```

```
8  class = versicolor
9  class = virginica

view(t)
```



Reference

[1] Breiman, L., et al., *Classification and Regression Trees*, Chapman & Hall, Boca Raton, 1993.

See Also

`eval`, `test`, `view`, `prune`

cluster

Purpose

Construct clusters from `linkage` output

Syntax

```
T = cluster(Z,'cutoff',c)
T = cluster(Z,'maxclust',n)
T = cluster(...,'criterion',crit)
T = cluster(...,'depth',d)
```

Description

`T = cluster(Z,'cutoff',c)` constructs clusters from the hierarchical cluster tree, `Z`, generated by the `linkage` function. `Z` is a matrix of size (m-1)-by-3, where m is the number of observations in the original data. `c` is a threshold for cutting `Z` into clusters. Clusters are formed when inconsistent values are less than `c`. See the `inconsistent` function for more information. The output `T` is a vector of size m that contains the cluster number for each observation in the original data.

`T = cluster(Z,'maxclust',n)` specifies `n` as the maximum number of clusters to form from the hierarchical tree in `Z`.

`T = cluster(...,'criterion',`*`crit`*`)` uses the specified criterion for forming clusters, where `crit` is either `'inconsistent'` or `'distance'`.

`T = cluster(...,'depth',d)` evaluates inconsistent values to a depth of `d` in the tree. The default is `d = 2`. An inconsistency coefficient computation compares a link between two objects in the cluster tree with neighboring links up to the specified depth. See the `inconsistent` function for more information.

Example

The example uses the `pdist` function to calculate the distance between items in a matrix of random numbers and then uses the `linkage` function to compute the hierarchical cluster tree based on the matrix. The example passes the output of the `linkage` function to the `cluster` function. The `'maxclust'` value `3` indicates that you want to group the items into three clusters. The `find` function lists all the items grouped into cluster 1.

```
rand('state', 7)
X = [rand(10,3); rand(10,3)+1; rand(10,3)+2];
Y = pdist(X);
```

```
Z = linkage(Y);
T = cluster(Z,'maxclust',3);
find(T==1)
ans =
    11
    12
    13
    14
    15
    16
    17
    18
    19
    20
```

See Also

`clusterdata`, `cophenet`, `inconsistent`, `linkage`, `pdist`

clusterdata

Purpose

Construct clusters from data

Syntax

```
T = clusterdata(X,cutoff)
T = clusterdata(X,param1,val1,param2,val2,...)
```

Description

`T = clusterdata(X,cutoff)` uses the `pdist`, `linkage`, and `cluster` functions to construct clusters from data `X`. `X` is an *m*-by-*n* matrix, treated as *m* observations of *n* variables. `cutoff` is a threshold for cutting the hierarchical tree generated by `linkage` into clusters. When `0 < cutoff < 2`, `clusterdata` forms clusters when inconsistent values are greater than `cutoff` (see the `inconsistent` function). When `cutoff` is an integer and `cutoff` ≥ `2`, then `clusterdata` interprets `cutoff` as the maximum number of clusters to keep in the hierarchical tree generated by `linkage`. The output `T` is a vector of size *m* containing a cluster number for each observation.

`T = clusterdata(X,cutoff)` is the same as

```
Y = pdist(X,'euclid');
Z = linkage(Y,'single');
T = cluster(Z,'cutoff',cutoff);
```

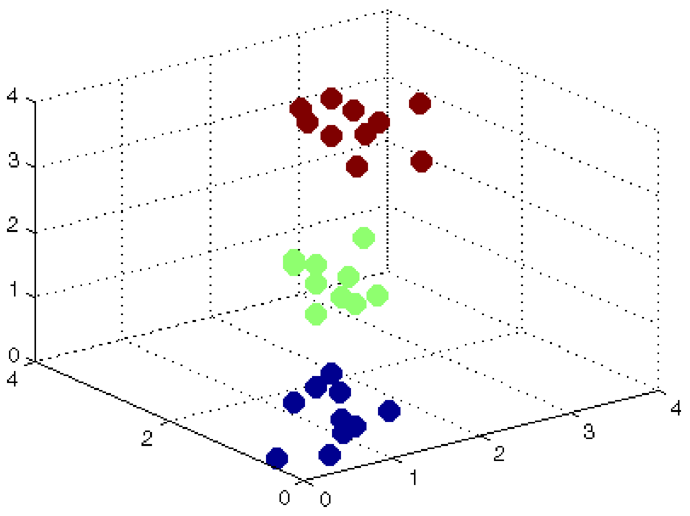
`T = clusterdata(X,param1,val1,param2,val2,...)` provides more control over the clustering through a set of parameter/value pairs. Valid parameters are

`'distance'`	Any of the distance metric names allowed by `pdist` (follow the `'minkowski'` option by the value of the exponent `p`)
`'linkage'`	Any of the linkage methods allowed by the `linkage` function
`'cutoff'`	Cutoff for inconsistent or distance measure
`'maxclust'`	Maximum number of clusters to form
`'criterion'`	Either `'inconsistent'` or `'distance'`
`’depth'`	Depth for computing inconsistent values

Example

The example first creates a sample data set of random numbers. It then uses `clusterdata` to compute the distances between items in the data set and create a hierarchical cluster tree from the data set. Finally, the `clusterdata` function groups the items in the data set into three clusters. The example uses the `find` function to list all the items in cluster 2, and the `scatter3` function to plot the data with each cluster shown in a different color.

```
rand('state',12);
X = [rand(10,3); rand(10,3)+1.2; rand(10,3)+2.5];
T = clusterdata(X,'maxclust',3);
find(T==2)
ans =
  11
  11
  13
  14
  15
  16
  17
  18
  19
  20
scatter3(X(:,1),X(:,2),X(:,3),100,T,'filled')
```



See Also `cluster`, `inconsistent`, `kmeans`, `linkage`, `pdist`

Purpose

Classical multidimensional scaling

Syntax

```
Y = cmdscale(D)
[Y,e] = cmdscale(D)
```

Description

`Y = cmdscale(D)` takes an `n`-by-`n` distance matrix `D`, and returns an `n`-by-`p` configuration matrix `Y`. Rows of `Y` are the coordinates of `n` points in `p`-dimensional space for some `p < n`. When `D` is a Euclidean distance matrix, the distances between those points are given by `D`. `p` is the dimension of the smallest space in which the `n` points whose inter-point distances are given by `D` can be embedded.

`[Y,e] = cmdscale(D)` also returns the eigenvalues of `Y*Y'`. When `D` is Euclidean, the first `p` elements of `e` are positive, the rest zero. If the first `k` elements of `e` are much larger than the remaining `(n-k)`, then you can use the first `k` columns of `Y` as `k`-dimensional points whose inter-point distances approximate `D`. This can provide a useful dimension reduction for visualization, e.g., for `k = 2`.

`D` need not be a Euclidean distance matrix. If it is non-Euclidean or a more general dissimilarity matrix, then some elements of `e` are negative, and `cmdscale` chooses `p` as the number of positive eigenvalues. In this case, the reduction to `p` or fewer dimensions provides a reasonable approximation to `D` only if the negative elements of `e` are small in magnitude.

You can specify `D` as either a full dissimilarity matrix, or in upper triangle vector form such as is output by `pdist`. A full dissimilarity matrix must be real and symmetric, and have zeros along the diagonal and positive elements everywhere else. A dissimilarity matrix in upper triangle form must have real, positive entries. You can also specify `D` as a full similarity matrix, with ones along the diagonal and all other elements less than one. `cmdscale` transforms a similarity matrix to a dissimilarity matrix in such a way that distances between the points returned in `Y` equal or approximate `sqrt(1-D)`. To use a different transformation, you must transform the similarities prior to calling `cmdscale`.

Examples

Generate some points in 4-dimensional space, but close to 3-dimensional space, then reduce them to distances only.

```
X = [normrnd(0,1,10,3) normrnd(0,.1,10,1)];
D = pdist(X,'euclidean');
```

Find a configuration with those inter-point distances.

```
[Y,e] = cmdscale(D);

% Four, but fourth one small
dim = sum(e > eps^(3/4))

% Poor reconstruction
maxerr2 = max(abs(pdist(X)-pdist(Y(:,1:2))))

% Good reconstruction
maxerr3 = max(abs(pdist(X)-pdist(Y(:,1:3))))

% Exact reconstruction
maxerr4 = max(abs(pdist(X)-pdist(Y)))

% D is now non-Euclidean
D = pdist(X,'cityblock');
[Y,e] = cmdscale(D);

% One is large negative
min(e)

% Poor reconstruction
maxerr = max(abs(pdist(X)-pdist(Y)))
```

References

[1] Seber, G. A. F., *Multivariate Observations*, Wiley, 1984

See Also

`mdscale`, `pdist`, `procrustes`

Purpose

Enumeration of all combinations of n objects k at a time

Syntax

```
C = combnk(v,k)
```

Description

`C = combnk(v,k)` returns all combinations of the n elements in `v` taken `k` at a time.

`C = combnk(v,k)` produces a matrix `C` with `k` columns and $n! / k!(n-k)!$ rows, where each row contains `k` of the elements in the vector `v`.

It is not practical to use this function if `v` has more than about 15 elements.

Example

Combinations of characters from a string.

```
C = combnk('tendril',4);
last5 = C(31:35,:)
last5 =
tedr
tenl
teni
tenr
tend
```

Combinations of elements from a numeric vector.

```
c = combnk(1:4,2)
c =
   3   4
   2   4
   2   3
   1   4
   1   3
   1   2
```

Purpose

Shewhart control charts

Syntax

```
controlchart(X)
controlchart(x,group)
controlchart(X,group)
[stats,plotdata] = controlchart(...)
controlchart(...,param1,val1,param2,val2,...)
```

Description

`controlchart(X)` produces an xbar chart of the measurements in matrix `X`. Each row of `X` is considered to be a subgroup of measurements containing replicate observations taken at the same time. The rows should be in time order. If `X` is a time series object, the time samples should contain replicate observations.

The chart plots the means of the subgroups in time order, a center line (`CL`) at the average of the means, and upper and lower control limits (`UCL`, `LCL`) at three standard deviations from the center line. Process standard deviation is estimated from the average of the subgroup standard deviations. Out of control measurements are marked as violations and drawn with a red circle. Data cursor mode is enabled, so clicking any data point displays information about that point.

`controlchart(x,group)` accepts a grouping variable `group` for a vector of measurements `x`. (See "Grouped Data" on page 2-41.) `group` is a categorical variable, vector, string array, or cell array of strings the same length as `x`. Consecutive measurements `x(n)` sharing the same value of `group(n)` for 1 ≤ `n` ≤ `length(x)` are defined to be a subgroup. Subgroups can have different numbers of observations.

Control limits are shown at three subgroup standard deviations from the subgroup means.

`controlchart(X,group)` accepts a grouping variable `group` for a matrix of measurements in `X`. In this case, `group` is only used to label the time axis; it does not change the default grouping by rows.

`[stats,plotdata] = controlchart(...)` returns a structure `stats` of subgroup statistics and parameter estimates, and a structure `plotdata` of plotted values. `plotdata` contains one record for each chart.

The fields in `stats` and `plotdata` depend on the chart type.

The fields in `stats` are selected from the following:

- `mean` — Subgroup means
- `std` — Subgroup standard deviations
- `range` — Subgroup ranges
- `n` — Subgroup size, or total inspection size or area
- `i` — Individual data values
- `ma` — Moving averages
- `mr` — Moving ranges
- `count` — Count of defects or defective items
- `mu` — Estimated process mean
- `sigma` — Estimated process standard deviation
- `p` — Estimated proportion defective
- `m` — Estimated mean defects per unit

The fields in `plotdata` are the following:

- `pts` — Plotted point values
- `cl` — Center line
- `lcl` — Lower control limit
- `ucl` — Upper control limit
- `se` — Standard error of plotted point
- `n` — Subgroup size
- `ooc` — Logical that is true for points that are out of control

`controlchart(...,`*`param1`*`,`*`val1`*`,`*`param2`*`,`*`val2`*`,...)` specifies one or more of the following parameter name/value pairs:

- `'charttype'` — The name of a chart type chosen from among the following:
  - `'xbar'` — Xbar or mean
  - `'s'` — Standard deviation
  - `'r'` — Range
  - `'ewma'` — Exponentially weighted moving average
  - `'i'` — Individual observation
  - `'mr'` — Moving range of individual observations
  - `'ma'` — Moving average of individual observations
  - `'p'` — Proportion defective
  - `'np'` — Number of defectives
  - `'u'` — Defects per unit
  - `'c'` — Count of defects

  Alternatively, a parameter can be a cell array listing multiple compatible chart types. There are four sets of compatible types:
  - `'xbar'`, `'s'`, `'r'`, and `'ewma'`
  - `'i'`, `'mr'`, and `'ma'`
  - `'p'` and `'np'`
  - `'u'` and `'c'`
- `'display'` — Either `'on'` (default) to display the control chart, or `'off'` to omit the display
- `'label'` — A string array or cell array of strings, one per subgroup. This label is displayed as part of the data cursor for a point on the plot.
- `'lambda'` — A parameter between 0 and 1 controlling how much the current prediction is influenced by past observations in an EWMA

plot. Higher values of `'lambda'` give less weight to past observations and more weight to the current observation. The default is 0.4.

- `'limits'` — A three-element vector specifying the values of the lower control limit, center line, and upper control limits. Default is to estimate the center line and to compute control limits based on the estimated value of sigma. Not permitted if there are multiple chart types.
- `'mean'` — Value for the process mean, or an empty value (default) to estimate the mean from `X`. This is the `p` parameter for `p` and `np` charts, the mean defects per unit for `u` and `c` charts, and the normal `mu` parameter for other charts.
- `'nsigma'` — The number of sigma multiples from the center line to a control limit. Default is 3.
- `'parent'` — The handle of the axes to receive the control chart plot. Default is to create axes in a new figure. Not permitted if there are multiple chart types.
- `'rules'` — The name of a control rule, or a cell array containing multiple control rule names. These rules, together with the control limits, determine if a point is marked as out of control. The default is to apply no control rules, and to use only the control limits to decide if a point is out of control. See `controlrules` for more information. Control rules are applied to charts that measure the process level (`xbar`, `i`, `c`, `u`, `p`, and `np`) rather than the variability (`r`, `s`), and they are not applied to charts based on moving statistics (`ma`, `mr`, `ewma`).
- `'sigma'` — Either a value for sigma, or a method of estimating sigma chosen from among `'std'` (the default) to use the average within-subgroup standard deviation, `'range'` to use the average subgroup range, and `'variance'` to use the square root of the pooled variance. When creating `i`, `mr`, or `ma` charts for data not in subgroups, the estimate is always based on a moving range.
- `'specs'` — A vector specifying specification limits. Typically this is a two-element vector of lower and upper specification limits. Since specification limits typically apply to individual measurements, this

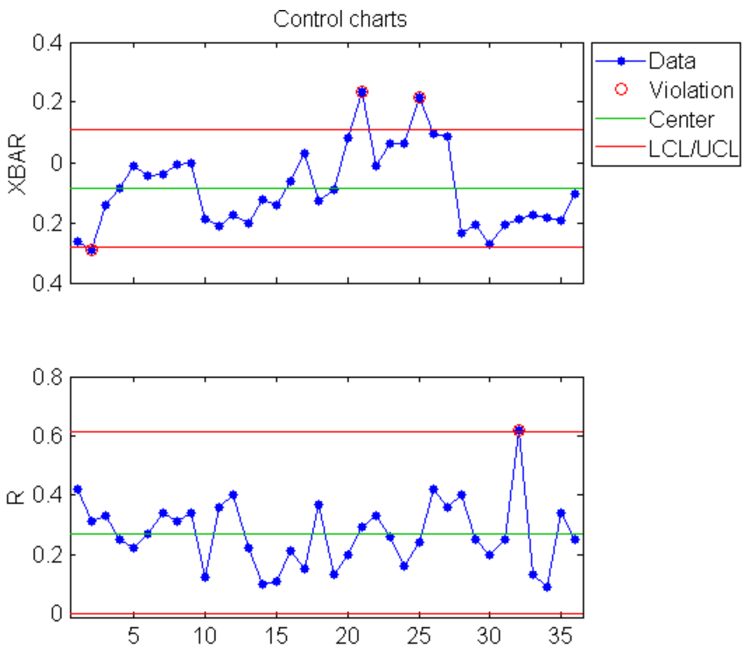
parameter is primarily suitable for `i` charts. These limits are not plotted on `r`, `s`, or `mr` charts.

- `'unit'` — The total number of inspected items for `p` and `np` charts, and the size of the inspected unit for `u` and `c` charts. In both cases `X` must be the count of the number of defects or defectives found. Default is 1 for `u` and `c` charts. This argument is required (no default) for `p` and `np` charts.
- `'width'` — The width of the window used for computing the moving ranges and averages in `mr` and `ma` charts, and for computing the sigma estimate in `i`, `mr`, and `ma` charts. Default is 5.

Example

Create `xbar` and `r` control charts for the data in `parts.mat`:

```
load parts
st = controlchart(runout,'chart',{'xbar' 'r'});
```



Display the process mean and standard deviation:

```
fprintf('Parameter estimates:  mu = %g, sigma = %g\n',st.mu,st.sigma);
Parameter estimates:  mu = -0.0863889, sigma = 0.130215
```

See Also

`controlrules`

controlrules

Purpose Western Electric and Nelson control rules

Syntax

```
R = controlrules(rules,x,cl,se)
[R,RULES] = controlrules(...)
```

Description `R = controlrules(`*`rules`*`,x,cl,se)` determines which points in the vector `x` violate the control rules in *`rules`*. `cl` is a vector of center-line values. `se` is a vector of standard errors. (Typically, control limits on a control chart are at the values `cl` – `3*se` and `cl` + `3*se`.) `rules` is the name of a control rule, or a cell array containing multiple control rule names, from the list below. If `x` has *n* values and *`rules`* contains *m* rules, then `R` is an *n*-by-*m* logical array, with `R(i,j)` assigned the value `1` if point `i` violates rule `j`, `0` if it does not.

The following are accepted values for *`rules`*:

- `'we1'` — 1 point above `cl` + `3*se`
- `'we2'` — 2 of 3 above `cl` + `2*se`
- `'we3'` — 4 of 5 above `cl` + `se`
- `'we4'` — 8 of 8 above `cl`
- `'we5'` — 1 below `cl` `3*se`
- `'we6'` — 2 of 3 below `cl` `2*se`
- `'we7'` — 4 of 5 below `cl` `se`
- `'we8'` — 8 of 8 below `cl`
- `'we9'` — 15 of 15 between `cl` `se` and `cl` + `se`
- `'we10'` — 8 of 8 below `cl` `se` or above `cl` + `se`
- `'n1'` — 1 point below `cl` `3*se` or above `cl` + `3*se`
- `'n2'` — 9 of 9 on the same side of `cl`
- `'n3'` — 6 of 6 increasing or decreasing
- `'n4'` — 14 alternating up/down

- `'n5'` — 2 of 3 below `cl   2*se` or above `cl + 2*se`, same side
- `'n6'` — 4 of 5 below `cl   se` or above `cl + se`, same side
- `'n7'` — 15 of 15 between `cl   se` and `cl + se`
- `'n8'` — 8 of 8 below `cl   se` or above `cl + se`, either side
- `'we'` — All Western Electric rules
- `'n'` — All Nelson rules

For multi-point rules, a rule violation at point `i` indicates that the set of points ending at point `i` triggered the rule. Point `i` is considered to have violated the rule only if it is one of the points violating the rule's condition.

Any points with `NaN` as their `x`, `cl`, or `se` values are not considered to have violated rules, and are not counted in the rules for other points.

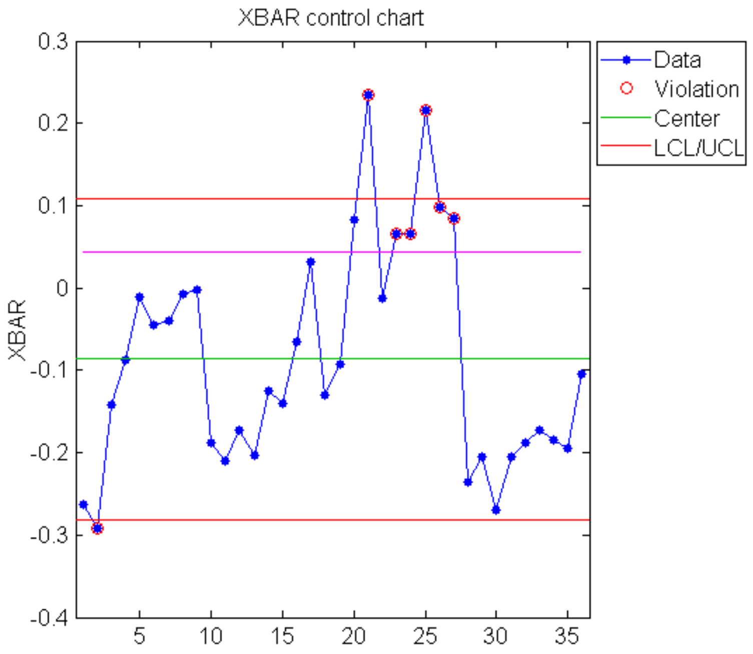
Control rules can be specified in the `controlchart` function as values for the `'rules'` parameter.

`[R,RULES] = controlrules(...)` returns a cell array of text strings `RULES` listing the rules applied.

Example

Create an xbar chart using the `we2` rule to mark out of control measurements:

```
load parts;
st = controlchart(runout,'rules','we2');
x = st.mean;
cl = st.mu;
se = st.sigma./sqrt(st.n);
hold on
plot(cl+2*se,'m')
```



Use `controlrules` to identify the measurements that violate the control rule:

```
R = controlrules('we2',x,cl,se);
I = find(R)
I =
    21
    23
    24
    25
    26
    27
```

See Also

`controlchart`

Purpose

Cophenetic correlation coefficient

Syntax

```
c = cophenet(Z,Y)
[c,d] = cophenet(Z,Y)
```

Description

`c = cophenet(Z,Y)` computes the cophenetic correlation coefficient for the hierarchical cluster tree represented by `Z`. `Z` is the output of the `linkage` function. `Y` contains the distances or dissimilarities used to construct `Z`, as output by the `pdist` function. `Z` is a matrix of size (m-1)-by-3, with distance information in the third column. `Y` is a vector of size $m \cdot (m-1)/2$.

`[c,d] = cophenet(Z,Y)` returns the cophenetic distances `d` in the same lower triangular distance vector format as `Y`.

The cophenetic correlation for a cluster tree is defined as the linear correlation coefficient between the cophenetic distances obtained from the tree, and the original distances (or dissimilarities) used to construct the tree. Thus, it is a measure of how faithfully the tree represents the dissimilarities among observations.

The cophenetic distance between two observations is represented in a dendrogram by the height of the link at which those two observations are first joined. That height is the distance between the two subclusters that are merged by that link.

The output value, `c`, is the cophenetic correlation coefficient. The magnitude of this value should be very close to 1 for a high-quality solution. This measure can be used to compare alternative cluster solutions obtained using different algorithms.

The cophenetic correlation between `Z(:,3)` and `Y` is defined as

$$c = \frac{\Sigma_{i<j}(Y_{ij}-y)(Z_{ij}-z)}{\sqrt{\Sigma_{i<j}(Y_{ij}-y)^2\Sigma_{i<j}(Z_{ij}-z)^2}}$$

where:

- Y_{ij} is the distance between objects i and j in `Y`.
- Z_{ij} is the cophenetic distance between objects i and j, from `Z(:,3)`.
- y and z are the average of `Y` and `Z(:,3)`, respectively.

Example

```
X = [rand(10,3); rand(10,3)+1; rand(10,3)+2];
Y = pdist(X);
Z = linkage(Y,'average');

% Compute Spearman's rank correlation between the
% dissimilarities and the cophenetic distances
[c,D] = cophenet(Z,Y);
r = corr(Y',D','type','spearman')
r =
   0.8279
```

See Also

`cluster`, `dendrogram`, `inconsistent`, `linkage`, `pdist`, `squareform`

Purpose

Copula cumulative distribution function

Syntax

```
Y = copulacdf('Gaussian',U,rho)
Y = copulacdf('t',U,rho,NU)
Y = copulacdf(family,U,alpha)
```

Description

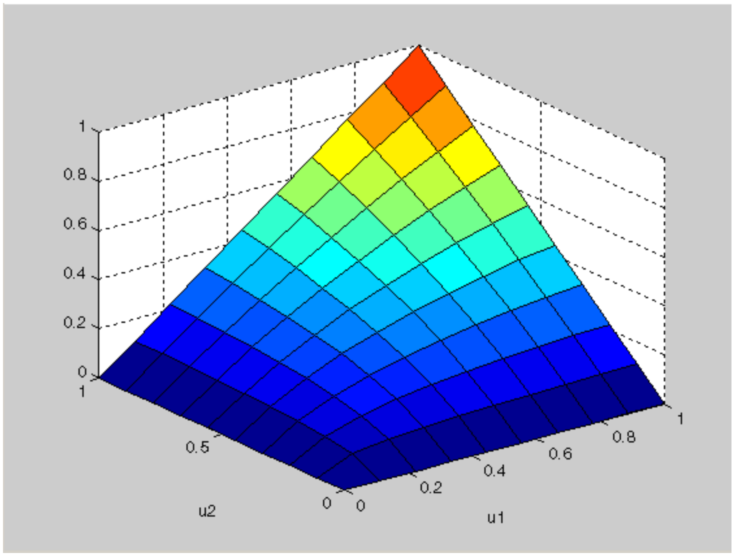
`Y = copulacdf('Gaussian',U,rho)` returns the cumulative probability of the Gaussian copula with linear correlation parameters `rho`, evaluated at the points in `U`. `U` is an `n`-by-`p` matrix of values in `[0,1]`, representing `n` points in the `p`-dimensional unit hypercube. `rho` is a `p`-by-`p` correlation matrix. If `U` is an `n`-by-`2` matrix, `rho` may also be a scalar correlation coefficient.

`Y = copulacdf('t',U,rho,NU)` returns the cumulative probability of the `t` copula with linear correlation parameters `rho` and degrees of freedom parameter `NU`, evaluated at the points in `U`. `U` is an `n`-by-`p` matrix of values in `[0,1]`. `rho` is a `p`-by-`p` correlation matrix. If `U` is an `n`-by-`2` matrix, `rho` may also be a scalar correlation coefficient.

`Y = copulacdf(`*`family`*`,U,alpha)` returns the cumulative probability of the bivariate Archimedean copula determined by *`family`*, with scalar parameter `alpha`, evaluated at the points in `U`. *`family`* is `'Clayton'`, `'Frank'`, or `'Gumbel'`. `U` is an `n`-by-`2` matrix of values in `[0,1]`.

Example

```
u = linspace(0,1,10);
[U1,U2] = meshgrid(u,u);
F = copulacdf('Clayton',[U1(:) U2(:)],1);
surf(U1,U2,reshape(F,10,10));
xlabel('u1'); ylabel('u2');
```



See Also copulapdf, copularnd, copulastat, copulaparam

Purpose

Copula parameters as function of rank correlation

Syntax

```
rho = copulaparam('Gaussian',R)
rho = copulaparam('t',R,NU)
alpha = copulaparam(family,R)
[...] = copulaparam(...,'type',type)
```

Description

`rho = copulaparam('Gaussian',R)` returns the linear correlation parameters `rho` corresponding to a Gaussian copula having Kendall's rank correlation `R`. If `R` is a scalar correlation coefficient, `rho` is a scalar correlation coefficient corresponding to a bivariate copula. If `R` is a `p`-by-`p` correlation matrix, `rho` is a `p`-by-`p` correlation matrix.

`rho = copulaparam('t',R,NU)` returns the linear correlation parameters `rho` corresponding to a `t` copula having Kendall's rank correlation `R` and degrees of freedom `NU`. If `R` is a scalar correlation coefficient, `rho` is a scalar correlation coefficient corresponding to a bivariate copula. If `R` is a `p`-by-`p` correlation matrix, `rho` is a `p`-by-`p` correlation matrix.

`alpha = copulaparam(`*`family`*`,R)` returns the copula parameter `alpha` corresponding to a bivariate Archimedean copula having Kendall's rank correlation `R`. `R` is a scalar. *`family`* is one of `'Clayton'`, `'Frank'`, or `'Gumbel'`.

`[...]  = copulaparam(...,'type',type)` assumes `R` is the specified type of rank correlation. `type` is `'Kendall'` for Kendall's tau or `'Spearman'` for Spearman's rho.

`copulaparam` uses an approximation to Spearman's rank correlation for some copula families when no analytic formula exists. The approximation is based on a smooth fit to values computed using Monte Carlo simulations.

Example

Get the linear correlation coefficient corresponding to a bivariate Gaussian copula having a rank correlation of `-0.5`.

```
tau = -0.5
rho = copulaparam('gaussian',tau)
rho =
   -0.7071

% Generate dependent beta random values using that copula
u = copularnd('gaussian',rho,100);
b = betainv(u,2,2);

% Verify that the sample has a rank correlation
% approximately equal to tau
tau_sample = corr(b,'type','k')
tau_sample =
    1.0000   -0.4638
   -0.4638    1.0000
```

See Also

copulacdf, copulapdf, copularnd, copulastat

Purpose

Copula probability density function

Syntax

```
Y = copulapdf('Gaussian',U,rho)
Y = copulapdf('t',U,rho,NU)
Y = copulapdf(family,U,alpha)
```

Description

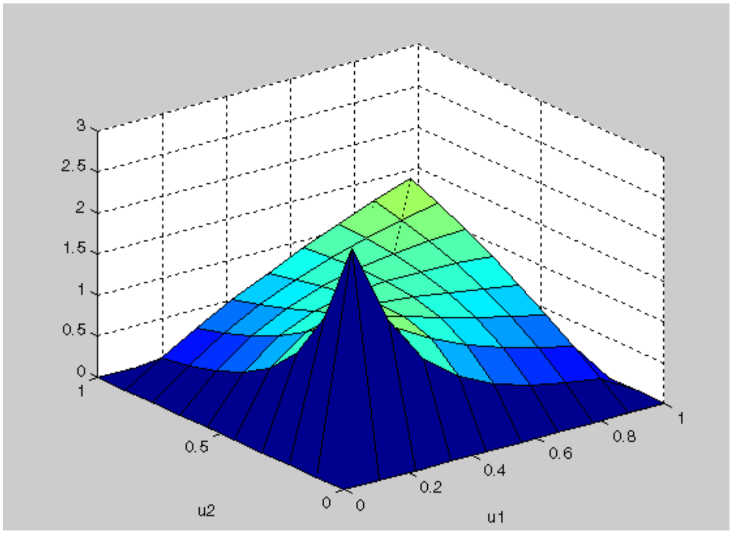
`Y = copulapdf('Gaussian',U,rho)` returns the probability density of the Gaussian copula with linear correlation parameters `rho`, evaluated at the points in `U`. `U` is an `n`-by-`p` matrix of values in `[0,1]`, representing `n` points in the `p`-dimensional unit hypercube. `rho` is a `p`-by-`p` correlation matrix. If `U` is an `n`-by-`2` matrix, `rho` may also be a scalar correlation coefficient.

`Y = copulapdf('t',U,rho,NU)` returns the probability density of the `t` copula with linear correlation parameters `rho` and degrees of freedom parameter `NU`, evaluated at the points in `U`. `U` is an `n`-by-`p` matrix of values in `[0,1]`. `rho` is a `p`-by-`p` correlation matrix. If `U` is an `n`-by-`2` matrix, `rho` may also be a scalar correlation coefficient.

`Y = copulapdf(`*`family`*`,U,alpha)` returns the probability density of the bivariate Archimedean copula determined by *`family`*, with scalar parameter `alpha`, evaluated at the points in `U`. *`family`* is `'Clayton'`, `'Frank'`, or `'Gumbel'`. `U` is an `n`-by-`2` matrix of values in `[0,1]`.

Example

```
u = linspace(0,1,10);
[U1,U2] = meshgrid(u,u);
F = copulapdf('Clayton',[U1(:) U2(:)],1);
surf(U1,U2,reshape(F,10,10));
xlabel('u1'); ylabel('u2');
```



See Also `copulacdf`, `copulaparam`, `copularnd`, `copulastat`

Purpose

Rank correlation for copula

Syntax

```
R = copulastat('Gaussian',rho)
R = copulastat('t',rho,NU)
R = copulastat(family,alpha)
R = copulastat(...,'type',type)
```

Description

`R = copulastat('Gaussian',rho)` returns the Kendall's rank correlation `R` that corresponds to a Gaussian copula having linear correlation parameters `rho`. If `rho` is a scalar correlation coefficient, `R` is a scalar correlation coefficient corresponding to a bivariate copula. If `rho` is a `p`-by-`p` correlation matrix, `R` is a `p`-by-`p` correlation matrix.

`R = copulastat('t',rho,NU)` returns the Kendall's rank correlation R that corresponds to a `t` copula having linear correlation parameters `rho` and degrees of freedom `NU`. If `rho` is a scalar correlation coefficient, `R` is a scalar correlation coefficient corresponding to a bivariate copula. If `rho` is a `p`-by-`p` correlation matrix, `R` is a `p`-by-`p` correlation matrix.

`R = copulastat(`*`family`*`,alpha)` returns the Kendall's rank correlation `R` that corresponds to a bivariate Archimedean copula with scalar parameter `alpha`. *`family`* is one of `'Clayton'`, `'Frank'`, or `'Gumbel'`.

`R = copulastat(...,'type',type)` returns the specified type of rank correlation. `type` is `'Kendall'` to compute Kendall's tau, or `'Spearman'` to compute Spearman's rho.

`copulastat` uses an approximation to Spearman's rank correlation for some copula families when no analytic formula exists. The approximation is based on a smooth fit to values computed using Monte-Carlo simulations.

Example

Get the theoretical rank correlation coefficient for a bivariate.

```
% Gaussian copula with linear correlation parameter rho
rho = -.7071;
tau = copulastat('gaussian',rho)
tau =
   -0.5000
```

```
% Generate dependent beta random values using that copula
u = copularnd('gaussian',rho,100);
b = betainv(u,2,2);

% Verify that the sample has a rank correlation
% approximately equal to tau
tau_sample = corr(b,'type','k')
tau_sample =
    1.0000   -0.5265
   -0.5265    1.0000
```

See Also

copulacdf, copulaparam, copulapdf, copularnd

Purpose

Random numbers from copula

Syntax

```
U = copularnd('Gaussian',rho,N)
U = copularnd('t',rho,NU,N)
U = copularnd(family,alpha,N)
```

Description

`U = copularnd('Gaussian',rho,N)` returns `N` random vectors generated from a Gaussian copula with linear correlation parameters `rho`. If `rho` is a `p`-by-`p` correlation matrix, `U` is an `n`-by-`p` matrix. If `rho` is a scalar correlation coefficient, `copularnd` generates `U` from a bivariate Gaussian copula. Each column of `U` is a sample from a `Uniform(0,1)` marginal distribution.

`U = copularnd('t',rho,NU,N)` returns `N` random vectors generated from a `t` copula with linear correlation parameters `rho` and degrees of freedom `NU`. If `rho` is a `p`-by-`p` correlation matrix, `U` is an `n`-by-`p` matrix. If `rho` is a scalar correlation coefficient, `copularnd` generates `U` from a bivariate `t` copula. Each column of `U` is a sample from a `Uniform(0,1)` marginal distribution.

`U = copularnd(`*`family`*`,alpha,N)` returns `N` random vectors generated from the bivariate Archimedean copula determined by *`family`*, with scalar parameter `alpha`. *`family`* is `'Clayton'`, `'Frank'`, or `'Gumbel'`. `U` is an `n`-by-2 matrix. Each column of `U` is a sample from a `Uniform(0,1)` marginal distribution.

Example

Determine the linear correlation parameter corresponding to a bivariate Gaussian copula having a rank correlation of `-0.5`.

```
tau = -0.5
rho = copulaparam('gaussian',tau)
 rho =
   -0.7071

% Generate dependent beta random values using that copula
u = copularnd('gaussian',rho,100);
b = betainv(u,2,2);
```

```
% Verify that the sample has a rank correlation
% approximately equal to tau
tau_sample = corr(b,'type','kendall')
tau_sample =
    1.0000   -0.4537
   -0.4537    1.0000
```

See Also `copulacdf`, `copulaparam`, `copulapdf`, `copulastat`

Purpose

D-optimal design of experiments coordinate exchange algorithm

Syntax

```
settings = cordexch(nfactors,nruns)
[settings,X] = cordexch(nfactors,nruns)
[settings,X] = cordexch(nfactors,nruns,model)
[settings,X] = cordexch(...,param1,val1,param2,val2,...)
```

Description

`settings = cordexch(nfactors,nruns)` generates the factor settings matrix, `settings`, for a D-optimal design using a linear additive model with a constant term. `settings` has `nruns` rows and `nfactors` columns.

`[settings,X] = cordexch(nfactors,nruns)` also generates the associated design matrix `X`.

`[settings,X] = cordexch(nfactors,nruns,`*`model`*`)` produces a design for fitting a specified regression model. The input, *`model`*, can be one of the following strings:

`'linear'`	Includes constant and linear terms (the default)
`'interaction'`	Includes constant, linear, and cross-product terms.
`'quadratic'`	Includes interactions and squared terms.
`'purequadratic'`	Includes constant, linear and squared terms.

Alternatively `model` can be a matrix of term definitions as accepted by the `x2fx` function.

`[settings,X] = cordexch(...,`*`param1`*`,`*`val1`*`,`*`param2`*`,`*`val2`*`,...)` provides more control over the design generation through a set of parameter/value pairs. Valid parameters are:

`'bounds'`	Lower and upper bounds for each factor, specified as a 2-by-`nfactors` matrix. Alternatively, this value can be a cell array containing `nfactors` elements, each element specifying the vector of allowable values for the corresponding factor.
`'categorical'`	Indices of categorical predictors.

`'display'`	Either `'on'` or `'off'` to control display of iteration counter. The default is `'on'`.
`'excludefun'`	Function to exclude undesirable runs.
`'init'`	Initial design as an `nruns`-by-`nfactors` matrix. The default is a randomly selected set of points.
`'levels'`	Vector of number of levels for each factor.
`'tries'`	Number of times to try to generate a design from a new starting point, using random points for each try except possibly the first. The default is 1.
`'maxiter'`	Maximum number of iterations. The default is 10.

Examples

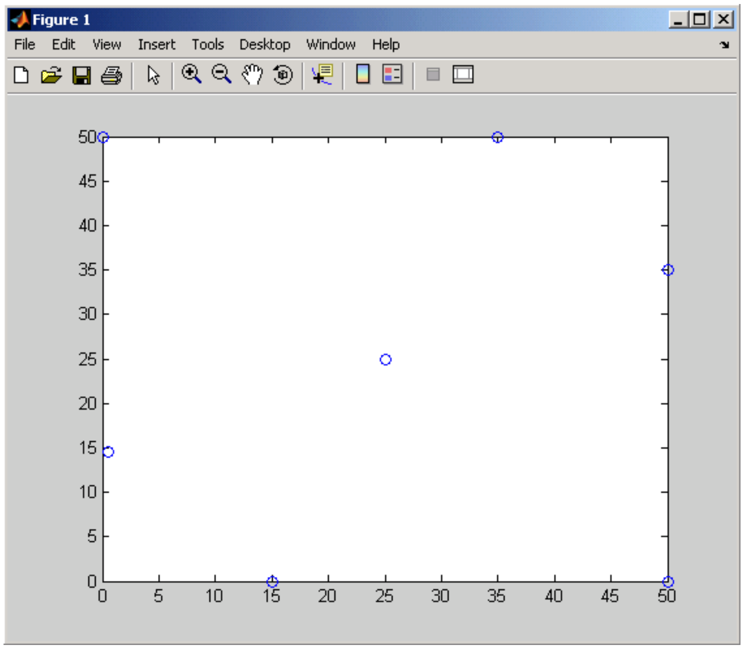
The D-optimal design for two factors in nine runs using a quadratic model is the 3^2 factorial as shown below:

```
settings = cordexch(2,9,'quadratic')
settings =
  -1   1
   1   1
   0   1
   1  -1
  -1  -1
   0  -1
   1   0
   0   0
  -1   0
```

The D-optimal design for 2 of 3 factors making up a mixture, where factor values are up to 50%, and the two factors must not make up less than 15% or greater than 85% of the whole mixture is shown as:

```
f = @(x) sum(x,2)>85 | sum(x,2)<15;
```

```
bnds = [0 0;50 50];
x = sortrows(cordexch(2,9,'q','bounds',bnds,...
                      'levels',101,'excl',f))
x =
         0   50.0000
         0   50.0000
    0.5000   14.5000
   15.0000         0
   25.0000   25.0000
   25.0000   25.0000
   35.0000   50.0000
   50.0000         0
   50.0000   35.0000

plot(x(:,1),x(:,2),'bo')
```



Algorithm

The `cordexch` function searches for a D-optimal design using a coordinate exchange algorithm. It creates a starting design, and then iterates by changing each coordinate of each design point in an attempt to reduce the variance of the coefficients that would be estimated using this design.

See Also

`bbdesign`, `candexch`, `candgen`, `ccdesign`, `daugment`, `dcovary`, `rowexch`, `x2fx`

Purpose

Linear or rank correlation

Syntax

```
RHO = corr(X)
RHO = corr(X,Y,...)
[RHO,PVAL] = corr(...)
[...] = corr(...,param1,val1,param2,val2,...)
```

Description

`RHO = corr(X)` returns a p-by-p matrix containing the pairwise linear correlation coefficient between each pair of columns in the n-by-p matrix `X`.

`RHO = corr(X,Y,...)` returns a $p1$-by-$p2$ matrix containing the pairwise correlation coefficient between each pair of columns in the n-by-$p1$ and n-by-$p2$ matrices `X` and `Y`.

`[RHO,PVAL] = corr(...)` also returns `PVAL`, a matrix of p-values for testing the hypothesis of no correlation against the alternative that there is a nonzero correlation. Each element of `PVAL` is the p-value for the corresponding element of `RHO`. If `PVAL(i, j)` is small, say less than `0.05`, then the correlation `RHO(i, j)` is significantly different from zero.

`[...]  = corr(...,`*`param1`*`,`*`val1`*`,`*`param2`*`,`*`val2`*`,...)` specifies additional parameters and their values. The following table lists the valid parameters and their values.

Parameter	Values
`'type'`	• `'Pearson'` (the default) computes Pearson's linear correlation coefficient • `'Kendall'` computes Kendall's tau • `'Spearman'` computes Spearman's rho

Parameter	Values
`'rows'`	• `'all'` (the default) uses all rows regardless of missing values (`NaN`s) • `'complete'` uses only rows with no missing values • `'pairwise'`computes `RHO(i,j)` using rows with no missing values in column `i` or `j`
`'tail'` — The alternative hypothesis against which to compute p-values for testing the hypothesis of no correlation	• `'ne'` — Correlation is not zero (the default) • `'gt'` — Correlation is greater than zero • `'lt'` — Correlation is less than zero

Using the `'pairwise'` option for the `'rows'` parameter might return a matrix that is not positive definite. The `'complete'` option always returns a positive definite matrix, but in general the estimates will be based on fewer observations.

`corr` computes p-values for Pearson's correlation using a Student's `t` distribution for a transformation of the correlation. This is exact when `X` and `Y` are normal. `corr` computes p-values for Kendall's tau and Spearman's rho using either the exact permutation distributions (for small sample sizes), or large-sample approximations.

`corr` computes p-values for the two-tailed test by doubling the more significant of the two one-tailed p-values.

See Also

`corrcoef`, `partialcorr`, `tiedrank`

Purpose

Correlation coefficients

Syntax

```
R = corrcoef(X)
R = corrcoef(x,y)
[R,P]=corrcoef(...)
[R,P,RLO,RUP]=corrcoef(...)
[...]=corrcoef(...,param1,val1,param2,val2,...)
```

Description

`R = corrcoef(X)` returns a matrix `R` of correlation coefficients calculated from an input matrix `X` whose rows are observations and whose columns are variables. The `(i,j)`th element of the matrix `R` is related to the covariance matrix `C = cov(X)` by

$$R(i,j) = \frac{C(i,j)}{\sqrt{C(i,i)C(j,j)}}$$

`corrcoef(X)` is the zeroth lag of the covariance function, that is, the zeroth lag of `xcov(x,'coeff')` packed into a square array.

`R = corrcoef(x,y)` where `x` and `y` are column vectors is the same as `corrcoef([x y])`.

`[R,P]=corrcoef(...)` also returns `P`, a matrix of p-values for testing the hypothesis of no correlation. Each p-value is the probability of getting a correlation as large as the observed value by random chance, when the true correlation is zero. If `P(i,j)` is small, say less than `0.05`, then the correlation `R(i,j)` is significant.

`[R,P,RLO,RUP]=corrcoef(...)` also returns matrices `RLO` and `RUP`, of the same size as `R`, containing lower and upper bounds for a 95% confidence interval for each coefficient.

`[...]=corrcoef(...,`*`param1`*`,`*`val1`*`,`*`param2`*`,`*`val2`*`,...)` specifies additional parameters and their values. Valid parameters are the following.

`'alpha'`	A number between 0 and 1 to specify a confidence level of 100(1 - `alpha`)%. Default is 0.05 for 95% confidence intervals.
`'rows'`	Either `'all'` (default) to use all rows, `'complete'` to use rows with no `NaN` values, or `'pairwise'` to compute `R(i,j)` using rows with no `NaN` values in either column `i` or `j`.

The p-value is computed by transforming the correlation to create a t statistic having `n`-2 degrees of freedom, where `n` is the number of rows of `X`. The confidence bounds are based on an asymptotic normal distribution of `0.5*log((1+R)/(1-R))`, with an approximate variance equal to `1/(n-3)`. These bounds are accurate for large samples when `X` has a multivariate normal distribution. The `'pairwise'` option can produce an `R` matrix that is not positive definite.

The `corrcoef` function is part of the standard MATLAB language.

Examples

Generate random data having correlation between column 4 and the other columns.

```
x = randn(30,4);       % Uncorrelated data
x(:,4) = sum(x,2);     % Introduce correlation
[r,p] = corrcoef(x)    % Sample correlation and p-values
r =
  1.0000  -0.3566  0.1929  0.3457
  -0.3566  1.0000  -0.1429  0.4461
  0.1929  -0.1429  1.0000  0.5183
  0.3457  0.4461  0.5183  1.0000
p =
  1.0000  0.0531  0.3072  0.0613
  0.0531  1.0000  0.4511  0.0135
  0.3072  0.4511  1.0000  0.0033
  0.0613  0.0135  0.0033  1.0000

[i,j] = find(p<0.05); % Find significant correlations
[i,j]                 % Display their (row,col) indices
```

```
ans =
     4     2
     4     3
     2     4
     3     4
```

See Also

`cov`, `mean`, `std`, `var`, `partialcorr`

Purpose

Covariance matrix

Syntax

```
C = cov(X)
cov(x,y)
```

Description

`C = cov(X)` computes the covariance matrix. For a single vector, `cov(x)` returns a scalar containing the variance. For matrices, where each row is an observation, and each column a variable, `cov(X)` is the covariance matrix.

The variance function, `var(X)` is the same as `diag(cov(X))`.

The standard deviation function, `std(X)` is equivalent to `sqrt(diag(cov(X)))`.

`cov(x,y)`, where `x` and `y` are column vectors of equal length, gives the same result as `cov([x y])`.

The `cov` function is part of the standard MATLAB language.

Algorithm

The algorithm for `cov` is

```
[n,p] = size(X);
X = X-ones(n,1)*mean(X);
Y = X'*X/(n-1);
```

See Also

`corrcoef`, `mean`, `std`, `var`

Purpose

Cox proportional hazards regression

Syntax

```
b = coxphfit(X,y)
[...] = coxphfit(X,Y,param1,val1,param2,val2,...)
[b,logl,H,stats] = coxphfit(...)
```

Description

`b = coxphfit(X,y)` returns a p-by-1 vector `b` of coefficient estimates for a Cox proportional hazards regression of the responses in `y` on the predictors in `X`. `X` is an n-by-p matrix of p predictors at each of n observations. `y` is an n-by-1 vector of observed responses.

The hazard rate for the distribution of `y` is modeled by `h(t)*exp(X*b)`, where `h(t)` is a common baseline hazard function. The model does not include a constant term, and `X` should not contain a column of ones.

`[...] = coxphfit(X,Y,param1,val1,param2,val2,...)` specifies additional parameter name/value pairs chosen from the following:

Name	Value
`'baseline'`	The `X` values at which the baseline hazard is to be computed. Default is `mean(X)`, so the hazard at `X` is `h(t)*exp((X-mean(X))*b)`. Enter `0` to compute the baseline relative to 0, so the hazard at `X` is `h(t)*exp(X*b)`.
`'censoring'`	A boolean array of the same size as `y` that is 1 for observations that are right-censored and 0 for observations that are observed exactly. Default is all observations observed exactly.
`'frequency'`	An array of the same size as `y` containing nonnegative integer counts. The j^{th} element of this vector gives the number of times the j^{th} element of `y` and the j^{th} row of `X` were observed. Default is one observation per row of `X` and `y`.

Name	Value
'init'	A vector containing initial values for the estimated coefficients b.
'options'	A structure specifying control parameters for the iterative algorithm used to estimate b. This argument can be created by a call to statset. For parameter names and default values, type statset('coxphfit').

[b,logl,H,stats] = coxphfit(...) returns additional results. logl is the log likelihood. H is a two-column matrix containing y values in the first column and the estimated baseline cumulative hazard evaluated at those values in the second column. stats is a structure that contains the fields:

- beta — Coefficient estimates (same as b)
- se — Standard errors of coefficient estimates b
- z — *z* statistics for b (b divided by standard error)
- p — *p*-values for b
- covb — Estimated covariance matrix for b

Example

Generate Weibull data depending on predictor x:

```
x = 4*rand(100,1);
A = 50*exp(-0.5*x); B = 2;
y = wblrnd(A,B);
```

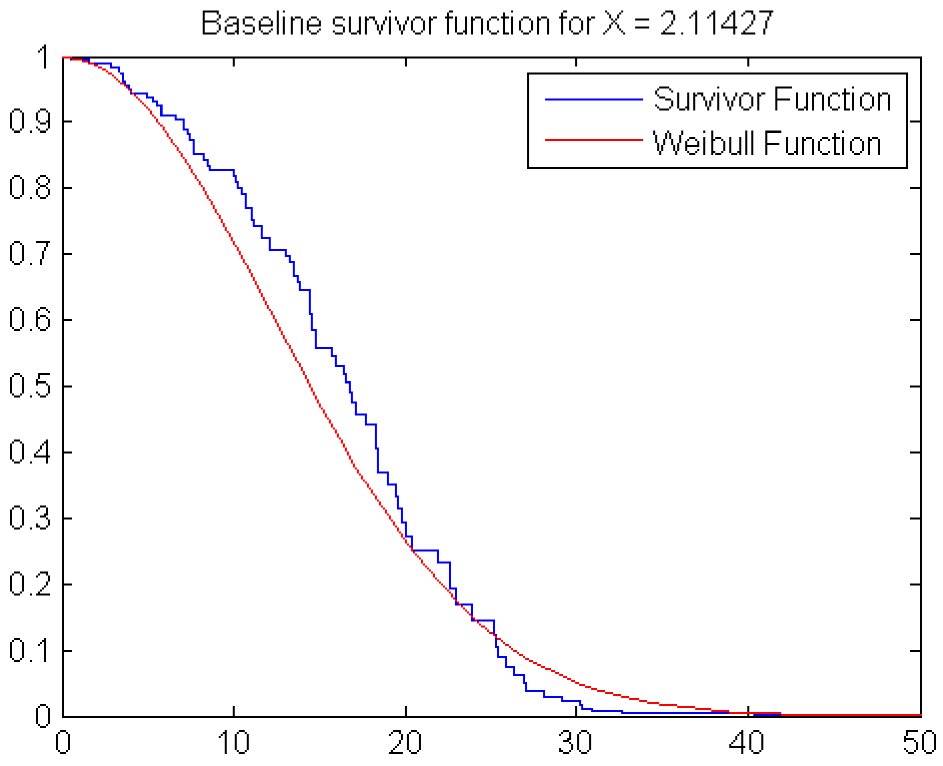
Fit a Cox model :

```
[b,logL,H,stats] = coxphfit(x,y);
```

Show the Cox estimate of the baseline survivor function together with the known Weibull function:

```
stairs(H(:,1),exp(-H(:,2)))
```

```
xx = linspace(0,100);
line(xx,1-wblcdf(xx,50*exp(-0.5*mean(x)),B),'color','r')
xlim([0,50])
title(sprintf('Baseline survivor function ...
              for X = %g',mean(x)));
legend('Survivor Function','Weibull Function')
```



Reference

[1] Cox, D.R., and D. Oakes, *Analysis of Survival Data*, Chapman & Hall, Boca Raton, 1984.

[2] Lawless, J.F., *Statistical Models and Methods for Lifetime Data*, Wiley, New York, 2003.

See Also `ecdf`, `statset`, `wblfit`

Purpose

Cross-tabulation of vectors

Syntax

```
table = crosstab(col1,col2)
table = crosstab(col1,col2,col3,...)
[table,chi2,p] = crosstab(col1,col2)
[table,chi2,p,label] = crosstab(col1,col2)
```

Description

`table = crosstab(col1,col2)` takes two vectors of positive integers and returns a matrix, `table`, of cross-tabulations. The *ij*th element of `table` contains the count of all instances where `col1` = *i* and `col2` = *j*.

Alternatively, `col1` and `col2` can be vectors containing non-integer values, categorical variables, character arrays, or cell arrays of strings. `crosstab` implicitly assigns a positive integer group number to each distinct value in `col1` and `col2`, and creates a cross-tabulation using those numbers.

`table = crosstab(col1,col2,col3,...)` returns `table` as an *n*-dimensional array, where *n* is the number of arguments you supply. The value of `table(i,j,k,...)` is the count of all instances where `col1` = *i*, `col2` = *j*, `col3` = *k*, and so on.

`[table,chi2,p] = crosstab(col1,col2)` also returns the chi-square statistic, `chi2`, for testing the independence of the rows and columns of `table`. The scalar `p` is the significance level of the test. Values of `p` near zero cast doubt on the assumption of independence of the rows and columns of table.

`[table,chi2,p,label] = crosstab(col1,col2)` also returns a cell array `label` that has one column for each input argument. The value in `label(`*i*`,`*j*`)` is the value of `col`*j* that defines group *i* in the *j*th dimension.

Example

Example 1

This example generates 2 columns of 50 discrete uniform random numbers. The first column has numbers from 1 to 3. The second has only the numbers 1 and 2. The two columns are independent so it would be surprising if `p` were near zero.

```
r1 = unidrnd(3,50,1);
r2 = unidrnd(2,50,1);
[table,chi2,p] = crosstab(r1,r2)
table =

  10   5
   8   8
   6  13
chi2 =
  4.1723
p =
  0.1242
```

The result, 0.1242, is not a surprise. A very small value of `p` would make you suspect the “randomness” of the random number generator.

Example 2

Suppose you have data collected on several cars over a period of time. How many four-cylinder cars were made in the USA during the late part of this period?

```
[t,c,p,l] = crosstab(cyl4,when,org);
l
l =
  'Other'  'Early'  'USA'
  'Four'   'Mid'    'Europe'
    []     'Late'   'Japan'
t(2,3,1)
ans =
  38
```

See Also

`tabulate`

Purpose

Categories for tree branches

Syntax

```
C = cutcategories(t)
C = cutcategories(t,nodes)
```

Description

`C = cutcategories(t)` returns an *n*-by-2 cell array `C` of the categories used at branches in the decision tree `t`, where *n* is the number of nodes. For each branch node `i` based on a categorical predictor variable `x`, the left child is chosen if `x` is among the categories listed in `C{i,1}`, and the right child is chosen if `x` is among those listed in `C{i,2}`. Both columns of `C` are empty for branch nodes based on continuous predictors and for leaf nodes.

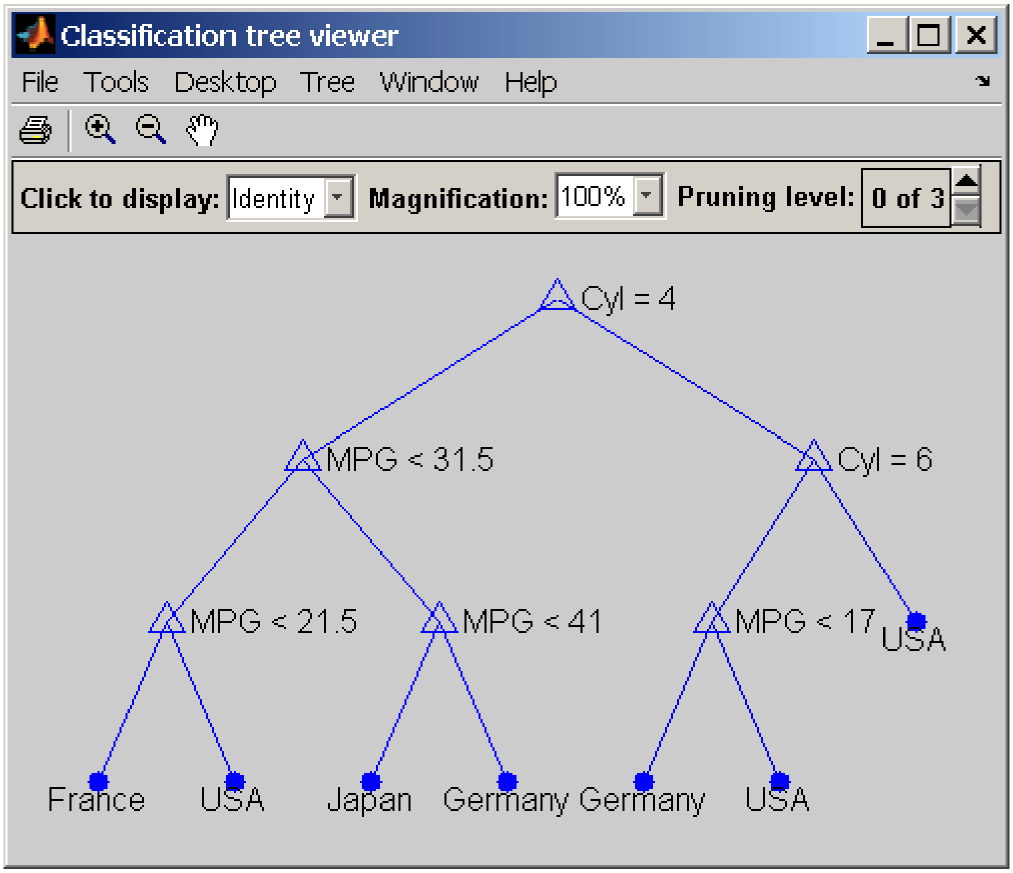
`C = cutcategories(t,nodes)` takes a vector `nodes` of node numbers and returns the categories for the specified nodes.

Example

Create a classification tree for car data:

```
load carsmall

t = classregtree([MPG Cylinders],Origin,...
                 'names',{'MPG' 'Cyl'},'cat',2)
t =
Decision tree for classification
 1  if Cyl=4 then node 2 else node 3
 2  if MPG<31.5 then node 4 else node 5
 3  if Cyl=6 then node 6 else node 7
 4  if MPG<21.5 then node 8 else node 9
 5  if MPG<41 then node 10 else node 11
 6  if MPG<17 then node 12 else node 13
 7  class = USA
 8  class = France
 9  class = USA
10  class = Japan
11  class = Germany
12  class = Germany
13  class = USA
```

```
view(t)
```



```
C = cutcategories(t)
C =
    [4]    [1x2 double]
     []              []
    [6]    [          8]
     []              []
```

```
          []              []
          []              []
          []              []
          []              []
          []              []
          []              []
          []              []
          []              []
          []              []
     C{1,2}
     ans =
          6     8
```

Reference

[1] Breiman, L., et al., *Classification and Regression Trees*, Chapman & Hall, Boca Raton, 1993.

See Also

`classregtree`, `cutvar`, `cutpoint`, `cuttype`

cutpoint

Purpose

Cutpoints for tree branches

Syntax

```
v = cutpoint(t)
v = cutpoint(t,nodes)
```

Description

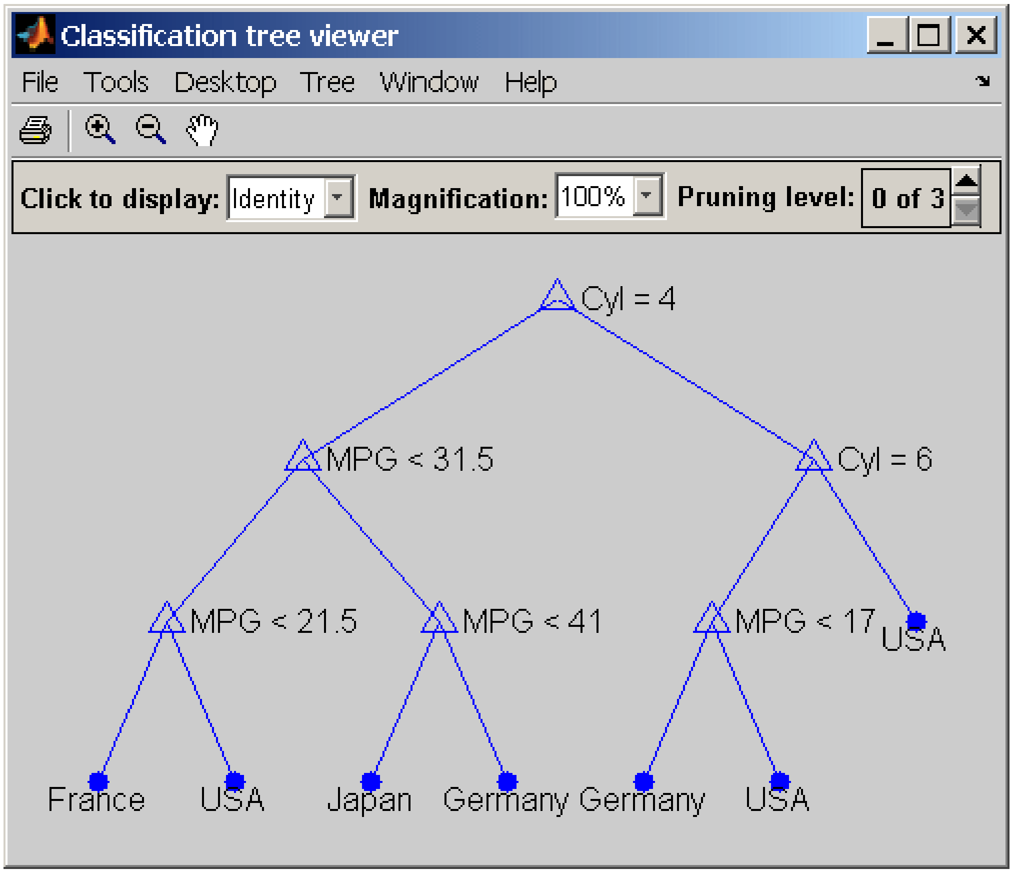
`v = cutpoint(t)` returns an *n*-element vector `v` of the values used as cutpoints in the decision tree `t`, where *n* is the number of nodes. For each branch node `i` based on a continuous predictor variable `x`, the left child is chosen if `x < v(i)` and the right child is chosen if `x >= v(i)`. `v` is `NaN` for branch nodes based on categorical predictors and for leaf nodes.

`v = cutpoint(t,nodes)` takes a vector `nodes` of node numbers and returns the cutpoints for the specified nodes.

Example

Create a classification tree for car data:

```
load carsmall

t = classregtree([MPG Cylinders],Origin,...
                 'names',{'MPG' 'Cyl'},'cat',2)
t =
Decision tree for classification
 1  if Cyl=4 then node 2 else node 3
 2  if MPG<31.5 then node 4 else node 5
 3  if Cyl=6 then node 6 else node 7
 4  if MPG<21.5 then node 8 else node 9
 5  if MPG<41 then node 10 else node 11
 6  if MPG<17 then node 12 else node 13
 7  class = USA
 8  class = France
 9  class = USA
10  class = Japan
11  class = Germany
12  class = Germany
13  class = USA

view(t)
```



```
v = cutpoint(t)
v =
          NaN
      31.5000
          NaN
      21.5000
      41.0000
      17.0000
          NaN
```

```
NaN
NaN
NaN
NaN
NaN
NaN
```

Reference

[1] Breiman, L., et al., *Classification and Regression Trees*, Chapman & Hall, Boca Raton, 1993.

See Also

`classregtree`, `cutvar`, `cutcategories`, `cuttype`

Purpose

Cut types for tree branches

Syntax

```
c = cuttype(t)
c = cuttype(t,nodes)
```

Description

`c = cuttype(t)` returns an *n*-element cell array `c` indicating the type of cut at each node in the tree `t`, where *n* is the number of nodes. For each node `i`, `c{i}` is:

- `'continuous'` — If the cut is defined in the form `x < v` for a variable `x` and cutpoint `v`.
- `'categorical'` — If the cut is defined by whether a variable `x` takes a value in a set of categories.
- `''` — If `i` is a leaf node.

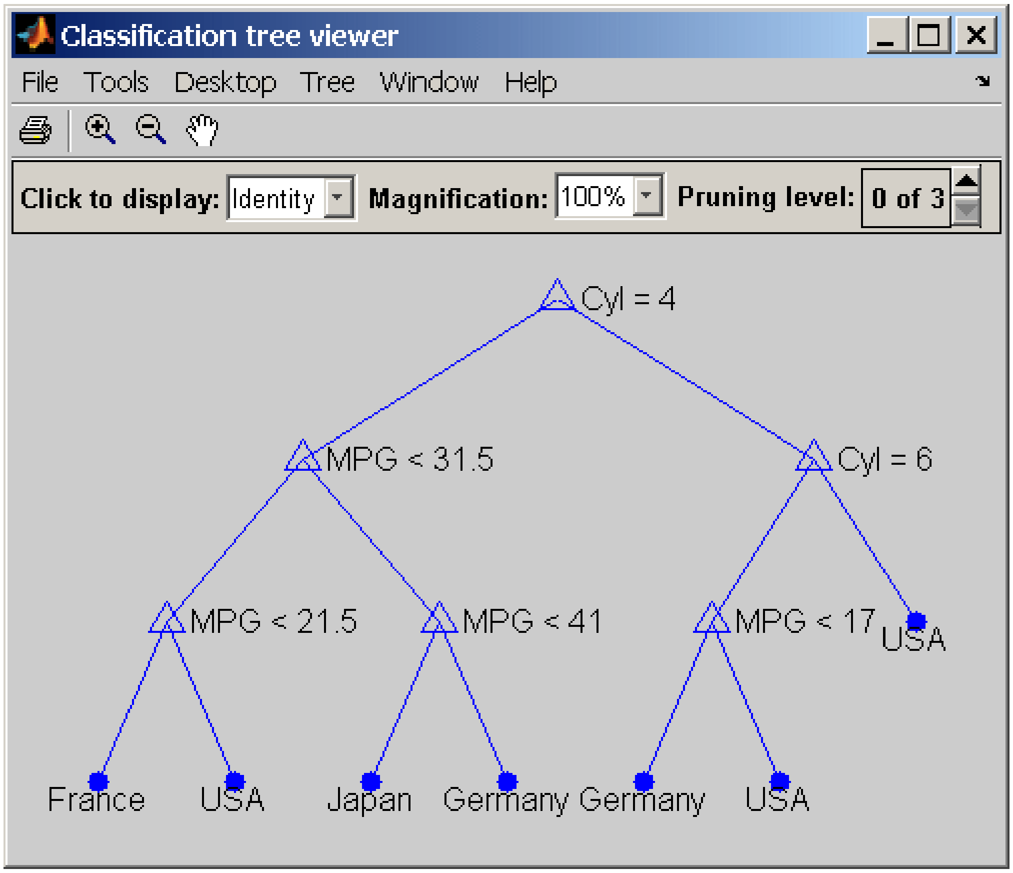
`cutvar` returns the cutpoints for `'continuous'` cuts, and `cutcategories` returns the set of categories.

`c = cuttype(t,nodes)` takes a vector `nodes` of node numbers and returns the cut types for the specified nodes.

Example

Create a classification tree for car data:

```
load carsmall

t = classregtree([MPG Cylinders],Origin,...
                 'names',{'MPG' 'Cyl'},'cat',2)
t =
Decision tree for classification
 1  if Cyl=4 then node 2 else node 3
 2  if MPG<31.5 then node 4 else node 5
 3  if Cyl=6 then node 6 else node 7
 4  if MPG<21.5 then node 8 else node 9
 5  if MPG<41 then node 10 else node 11
 6  if MPG<17 then node 12 else node 13
 7  class = USA
 8   class = France
 9   class = USA
10   class = Japan
11   class = Germany
12   class = Germany
13   class = USA

view(t)
```



```
c = cuttype(t)
c = 
    'categorical'
    'continuous'
    'categorical'
    'continuous'
    'continuous'
    'continuous'
    ''
    ''
    ''
    ''
    ''
    ''
    ''
```

Reference

[1] Breiman, L., et al., *Classification and Regression Trees*, Chapman & Hall, Boca Raton, 1993.

See Also

`classregtree`, `numnodes`, `cutvar`, `cutcategories`

Purpose

Variable names for tree branches

Syntax

```
v = cutvar(t)
v = cutvar(t,nodes)
[v,num] = cutvar(...)
```

Description

`v = cutvar(t)` returns an *n*-element cell array `v` of the names of the variables used for branching in each node of the tree `t`, where *n* is the number of nodes. These variables are sometimes known as *cut variables*. For leaf nodes, `v` contains an empty string.

`v = cutvar(t,nodes)` takes a vector `nodes` of node numbers and returns the cut variables for the specified nodes.

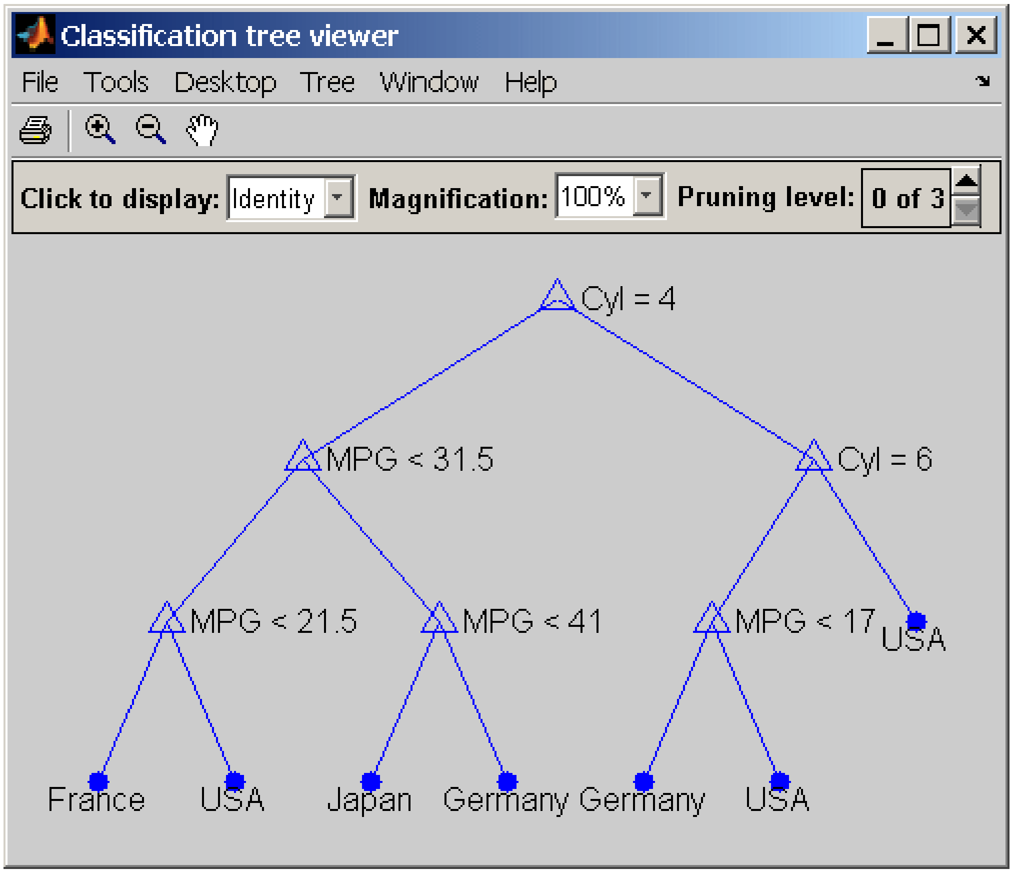
`[v,num] = cutvar(...)` also returns a vector `num` containing the number of each variable.

Example

Create a classification tree for car data:

```
load carsmall

t = classregtree([MPG Cylinders],Origin,...
                 'names',{'MPG' 'Cyl'},'cat',2)
t =
Decision tree for classification
 1  if Cyl=4 then node 2 else node 3
 2  if MPG<31.5 then node 4 else node 5
 3  if Cyl=6 then node 6 else node 7
 4  if MPG<21.5 then node 8 else node 9
 5  if MPG<41 then node 10 else node 11
 6  if MPG<17 then node 12 else node 13
 7  class = USA
 8  class = France
 9  class = USA
10  class = Japan
11  class = Germany
12  class = Germany
13  class = USA
```

```
view(t)
```


Classification tree viewer
File Tools Desktop Tree Window Help
Click to display: Identity Magnification: 100% Pruning level: 0 of 3

```
[v,num] = cutvar(t)
v =
    'Cyl'
    'MPG'
    'Cyl'
    'MPG'
```

```
          'MPG'
          'MPG'
          ''
          ''
          ''
          ''
          ''
          ''
          ''
num =
            2
            1
            2
            1
            1
            1
            0
            0
            0
            0
            0
            0
            0
```

Reference

[1] Breiman, L., et al., *Classification and Regression Trees*, Chapman & Hall, Boca Raton, 1993.

See Also

`classregtree`, `numnodes`, `children`

Purpose

Create dataset array

Syntax

```
A = dataset(var1,var2,...)
A = dataset(...,{var,name},...)
A = dataset(...,{var,name_1,...,name_m},...)
A = dataset(...,'varnames',{name_1,...,name_m},...)
A = dataset(...,'obsnames',{name_1,...,name_n},...)
A = dataset('File',filename,...)
A = dataset('File',filename,'Format',format,...)
A = dataset('XLSFile',filename,...)
```

Description

`A = dataset(var1,var2,...)` creates a dataset array `A` from the workspace variables `var1`, `var2`, ... All variables must have the same number of rows.

`A = dataset(...,{var,name},...)` creates the variable `var` in `A` and assigns the variable name *`name`*. Names must be valid, unique MATLAB identifier strings.

`A = dataset(...,{var,name_1,...,name_m},...)`, where `var` is an `n`-by-`m`-by-`p`-by-... array, creates `m` variables in `A`, each of size `n`-by-`p`-by-..., with names *`name_1`*, ..., *`name_m`*.

`A = dataset(...,'varnames',{name_1,...,name_m},...)` creates variables in `A` with the specified variable names. Names must be valid, unique MATLAB identifier strings. You may not use the `'varnames'` parameter and provide names for individual variables.

`A = dataset(...,'obsnames',{name_1,...,name_n},...)` creates observations in `A` with the specified observation names. The names need not be valid MATLAB identifier strings, but must be unique.

Note Dataset arrays may contain built-in types or array objects as variables. Array objects must implement each of the following:

- Standard MATLAB parenthesis indexing of the form `var(i,...)`, where `i` is a numeric or logical vector corresponding to rows of the variable
- A `size` method with a `dim` argument
- A `vertcat` method

`A = dataset('File',`*`filename,`*`...)` creates a dataset array `A` from column-oriented data in a text file specified by the string *`filename`*. You can indicate a delimiter character using a *`delimiter`* argument, as for `tdfread`.

`A = dataset('File',`*`filename,`*`'Format',`*`format`*`,...)` creates a dataset array `A` from column-oriented data in a text file specified by the string *`filename`* and format string *`format`*, as for `textscan`. You can also use any of the other parameter name/value pairs allowed by `textscan`.

`A = dataset('XLSFile',`*`filename,`*`...)` creates a dataset array `A` from column-oriented data in an Excel spreadsheet file specified by the string *`filename`*. You can also indicate a sheet number and a rectangular range of cells in the spreadsheet, as for `xlsread`.

When reading from a text file or spreadsheet, by default, variable names are taken from the first row of the file. You can use a `'ReadVarNames'` parameter and an associated logical value to indicate whether (`true`) or not (`false`) to respect the default behavior. Similarly, a `'ReadObsNames'` parameter and an associated logical value can be used to determine whether or not the first column of the file is treated as observation names (the default value is `false`). If `'ReadVarNames'` and `'ReadObsNames'` are both `true`, the name in the first column of the first row of the file is saved as the first dimension name for the dataset array.

Reading from a text file or spreadsheet creates scalar-valued variables in the dataset array, i.e., one variable from each column in the file. The variables that are created are either `double`-valued, if the entire column is numeric, or string-valued (i.e., a cell array of strings), if any element in a column is nonnumeric.

Examples

Example 1

Create a dataset array to contain Fisher's iris data:

```
load fisheriris
NumObs = size(meas,1);
ObsNames = strcat({'Obs'},num2str((1:NumObs)','%d'));
iris = dataset({nominal(species),'species'},...
               {meas,'SL','SW','PL','PW'},...
               'obsnames',ObsNames);
iris(1:5,:)
ans =
            species    SL     SW     PL     PW
    Obs1    setosa     5.1    3.5    1.4    0.2
    Obs2    setosa     4.9      3    1.4    0.2
    Obs3    setosa     4.7    3.2    1.3    0.2
    Obs4    setosa     4.6    3.1    1.5    0.2
    Obs5    setosa       5    3.6    1.4    0.2
```

Example 2

1 Load patient data from the CSV file `hospital.dat` and store the information in a dataset array with observation names given by the first column in the data (patient identification):

```
patients = dataset('file','hospital.dat',...
                   'delimiter',',',...
                   'ReadObsNames',true);
```

2 Make the {0,1}-valued variable `smoke` nominal, and change the labels to `'No'` and `'Yes'`:

```
patients.smoke = nominal(patients.smoke,{'No','Yes'});
```

3 Add new levels to `smoke` as placeholders for more detailed histories of smokers:

```
patients.smoke = addlevels(patients.smoke,...
                 {'0-5 Years','5-10 Years','LongTerm'});
```

4 Assuming the nonsmokers have never smoked, relabel the `'No'` level:

```
patients.smoke = setlabels(patients.smoke,'Never','No');
```

5 Drop the undifferentiated `'Yes'` level from `smoke`:

```
patients.smoke = droplevels(patients.smoke,'Yes');

Warning: OLDLEVELS contains categorical levels that
were present in A, caused some array elements to have
undefined levels.
```

Note that smokers now have an undefined level.

6 Set each smoker to one of the new levels, by observation name:

```
patients.smoke('YPL-320') = '5-10 Years';
```

See Also

`tdfread`, `textscan`, `xlsread`

Purpose

Apply function to variables of dataset array

Syntax

```
b = datasetfun(fun,A)
[b,c,...] = datasetfun(fun,A)
[b,...] = datasetfun(fun,A,...,'UniformOutput',false)
[b,...] = datasetfun(fun,A,...,'DatasetOutput',true)
[b,...] = datasetfun(fun,A,...,'DataVars',vars)
[b,...] = datasetfun(fun,A,...,'ObsNames',obsnames)
[b,...] = datasetfun(fun,A,...,'ErrorHandler',efun)
```

Description

`b = datasetfun(fun,A)` applies the function specified by `fun` to each variable of the dataset array `A`, and returns the results in the vector `b`. The *i*th element of `b` is equal to `fun` applied to the *i*th dataset variable of `A`. `fun` is a function handle to a function that takes one input argument and returns a scalar value. `fun` must return values of the same class each time it is called, and `datasetfun` concatenates them into the vector `b`. The outputs from `fun` must be one of the following types: numeric, logical, character, structure, or cell.

To apply functions that return results that are nonscalar or of different sizes and types, use the `'UniformOutput'` or `'DatasetOutput'` parameters described below.

Do not rely on the order in which `datasetfun` computes the elements of `b`, which is unspecified.

If `fun` is bound to more than one built-in function or M-file, (that is, if it represents a set of overloaded functions), `datasetfun` follows MATLAB dispatching rules in calling the function. (See “How MATLAB Determines Which Method to Call”.)

`[b,c,...]  = datasetfun(fun,A)`, where `fun` is a function handle to a function that returns multiple outputs, returns vectors `b`, `c`, ..., each corresponding to one of the output arguments of `fun`. `datasetfun` calls `fun` each time with as many outputs as there are in the call to `datasetfun`. `fun` may return output arguments having different classes, but the class of each output must be the same each time `fun` is called.

`[b,...] = datasetfun(fun,A,...,'UniformOutput',false)` allows you to specify a function `fun` that returns values of different sizes or types. `datasetfun` returns a cell array (or multiple cell arrays), where the *i*th cell contains the value of `fun` applied to the *i*th dataset variable of `A`. Setting `'UniformOutput'` to `true` is equivalent to the default behavior.

`[b,...] = datasetfun(fun,A,...,'DatasetOutput',true)` specifies that the output(s) of `fun` are returned as variables in a dataset array (or multiple dataset arrays). `fun` must return values with the same number of rows each time it is called, but it may return values of any type. The variables in the output dataset array(s) have the same names as the variables in the input. Setting `'DatasetOutput'` to `false` specifies that the type of the output(s) from `datasetfun` is determined by `'UniformOutput'`.

`[b,...] = datasetfun(fun,A,...,'DataVars',vars)` allows you to apply `fun` only to the dataset variables in `A` specified by `vars`. `vars` is a positive integer, a vector of positive integers, a variable name, a cell array containing one or more variable names, or a logical vector.

`[b,...] = datasetfun(fun,A,...,'ObsNames',obsnames)` specifies observation names for the dataset output when `'DatasetOutput'` is `true`.

`[b,...] = datasetfun(fun,A,...,'ErrorHandler',efun)`, where `efun` is a function handle, specifies the function for MATLAB to call if the call to `fun` fails. The error-handling function is called with the following input arguments:

- A structure with the fields `identifier`, `message`, and `index`, respectively containing the identifier of the error that occurred, the text of the error message, and the linear index into the input array(s) at which the error occurred
- The set of input arguments at which the call to the function failed

The error-handling function should either re-throw an error, or return the same number of outputs as `fun`. These outputs are then returned as the outputs of `datasetfun`. If `'UniformOutput'` is true, the outputs of the error handler must also be scalars of the same type as the outputs of `fun`. For example, the following code could be saved in an M-file as the error-handling function:

```
function [A,B] = errorFunc(S,varargin)

warning(S.identifier,S.message);
A = NaN;
B = NaN;
```

If an error-handling function is not specified, the error from the call to `fun` is rethrown.

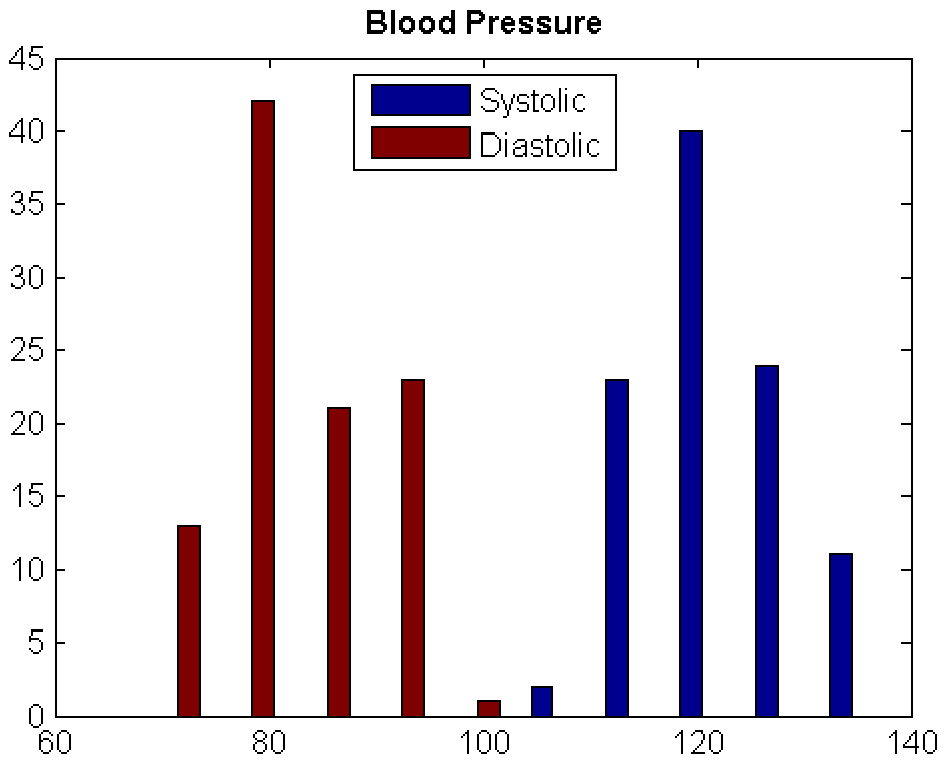
Example

Compute statistics on selected variables in the `hospital` dataset array:

```
load hospital

stats = datasetfun(@mean,hospital,...
                   'DataVars',{'Weight','BloodPressure'},...
                   'UniformOutput',false)
stats =
    [154]    [1x2 double]
stats{2}
ans =
  122.7800   82.9600
```

Display the blood pressure variable:

```
datasetfun(@hist,hospital,...
           'DataVars','BloodPressure',...
           'UniformOutput',false);
title('{\bf Blood Pressure}')
legend('Systolic','Diastolic','Location','N')
```



See Also `grpstats (dataset)`

Purpose

D-optimal augmentation of experimental design

Syntax

```
settings = daugment(startdes,nruns)
[settings,X] = daugment(startdes,nruns)
[settings,X] = daugment(startdes,nruns,model)
[settings,X] = daugment(...,param1,val1,param2,val2,...)
```

Description

`settings = daugment(startdes,nruns)` adds `nruns` runs to an experimental design using the coordinate exchange D-optimal algorithm. `startdes` is a matrix of factor settings in the original design. The output matrix `settings` is the matrix of factor settings for the design.

`[settings,X] = daugment(startdes,nruns)` also generates the associated design matrix, `X`.

`[settings,X] = daugment(startdes,nruns,`*`model`*`)` also controls the order of the regression model. The input *`model`* can be one of the following strings:

`'linear'`	Includes constant and linear terms (the default)
`'interaction'`	Includes constant, linear, and cross-product terms.
`'quadratic'`	Includes interactions and squared terms.
`'purequadratic'`	Includes constant, linear and squared terms.

Alternatively *`model`* can be a matrix of term definitions as accepted by the `x2fx` function.

`[settings,X] = daugment(...,`*`param1`*`,`*`val1`*`,`*`param2`*`,`*`val2`*`,...)` provides more control over the design generation through a set of parameter/value pairs. Valid parameter/value pairs are the following:

Parameter	Value
'display'	Either 'on' or 'off' to control display of iteration counter. The default is 'on'.
'init'	Initial design as an nruns-by-nfactors matrix. The default is a randomly selected set of points.
'maxiter'	Maximum number of iterations. The default is 10.

Example

This example adds 5 runs to a 2^2 factorial design to fit a quadratic model.

```
startdes = [-1 -1; 1 -1; -1 1; 1 1];
settings = daugment(startdes,5,'quadratic')
settings =
  -1   -1
   1   -1
  -1    1
   1    1
   1    0
  -1    0
   0    1
   0    0
   0   -1
```

The result is a 3^2 factorial design.

See Also

cordexch, x2fx

Purpose

D-optimal design with specified fixed covariates

Syntax

```
settings = dcovary(nfactors,covariates)
[settings,X] = dcovary(nfactors,covariates)
[settings,X] = dcovary(nfactors,covariates,model)
[settings,X] = dcovary(...,param1,val1,param2,val2,...)
```

Description

`settings = dcovary(nfactors,covariates)` uses a coordinate exchange algorithm to generate a D-optimal design for `nfactors` factors, subject to the constraint that it also include the fixed covariate values in the input matrix `covariates`. The number of runs in the design is taken to be the number of rows in the `covariates` matrix. The output matrix `settings` is the matrix of factor settings for the design, including the fixed covariates.

`[settings,X] = dcovary(nfactors,covariates)` also generates the associated design matrix, `X`.

`[settings,X] = dcovary(nfactors,covariates,`*`model`*`)` also controls the order of the regression model. The input *`model`* can be one of the following strings:

`'linear'`	Includes constant and linear terms (the default)
`'interaction'`	Includes constant, linear, and cross-product terms.
`'quadratic'`	Includes interactions and squared terms.
`'purequadratic'`	Includes constant, linear and squared terms.

Alternatively *`model`* can be a matrix of term definitions as accepted by the `x2fx` function. The model is applied to the fixed covariates as well as the regular factors. If you want to treat the fixed covariates specially, for example by including linear terms for them but quadratic terms for the regular factors, you can do this by creating the proper *`model`* matrix.

`[settings,X] = dcovary(...,`*`param1`*`,`*`val1`*`,`*`param2`*`,`*`val2`*`,...)` provides more control over the design generation through a set of parameter/value pairs. Valid parameters are:

'display'	Either 'on' or 'off' to control display of iteration counter. The default is 'on'.
'init'	Initial design as an nruns-by-nfactors matrix. The default is a randomly selected set of points.
'maxiter'	Maximum number of iterations. The default is 10.

Example

Example 1

Generate a design for three factors in 2 blocks of 4 runs.

```
blk = [-1 -1 -1 -1 1 1 1 1]';
dsgn = dcovary(3,blk)
dsgn =
   -1    1    1   -1
    1   -1   -1   -1
   -1    1   -1   -1
    1   -1    1   -1
    1    1   -1    1
    1    1    1    1
   -1   -1    1    1
   -1   -1   -1    1
```

Example 2

Suppose you want to block an eight-run experiment into 4 blocks of size 2 to fit a linear model on two factors.

```
covariates = dummyvar([1 1 2 2 3 3 4 4]);
settings = dcovary(2,covariates(:,1:3),'linear')
settings =
    1    1    1    0    0
   -1   -1    1    0    0
   -1    1    0    1    0
    1   -1    0    1    0
    1    1    0    0    1
   -1   -1    0    0    1
   -1    1    0    0    0
```

```
     1    -1     0     0     0
```

The first two columns of the output matrix contain the settings for the two factors. The last three columns are *dummy variable* codings for the four blocks.

Algorithm

The `dcovary` function creates a starting design that includes the fixed covariate values, and then iterates by changing the non-fixed coordinates of each design point in an attempt to reduce the variance of the coefficients that would be estimated using this design.

See Also

`cordexch`, `daugment`, `rowexch`, `x2fx`

dendrogram

Purpose

Plot dendrogram

Syntax

```
H = dendrogram(Z)
H = dendrogram(Z,p)
[H,T] = dendrogram(...)
[H,T,perm] = dendrogram(...)
[...] = dendrogram(...,'colorthreshold',t)
[...] = dendrogram(...,'orientation','orient')
[...] = dendrogram(...,'labels',S)
```

Description

`H = dendrogram(Z)` generates a dendrogram plot of the hierarchical, binary cluster tree represented by `Z`. `Z` is an (`m-1`)-by-3 matrix, generated by the `linkage` function, where `m` s the number of objects in the original data set. The output, `H`, is a vector of handles to the lines in the dendrogram.

A dendrogram consists of many U-shaped lines connecting objects in a hierarchical tree. The height of each U represents the distance between the two objects being connected. If there were 30 or fewer data points in the original dataset, each leaf in the dendrogram corresponds to one data point. If there were more than 30 data points, the complete tree can look crowded, and `dendrogram` collapses lower branches as necessary, so that some leaves in the plot correspond to more than one data point.

`H = dendrogram(Z,p)` generates a dendrogram with no more than `p` leaf nodes, by collapsing lower branches of the tree. To display the complete tree, set `p = 0`.

`[H,T] = dendrogram(...)` generates a dendrogram and returns `T`, a vector of length `m` that contains the leaf node number for each object in the original data set. `T` is useful when `p` is less than the total number of objects, so some leaf nodes in the display correspond to multiple objects. For example, to find out which objects are contained in leaf node `k` of the dendrogram, use `find(T==k)`. When there are fewer than `p` objects in the original data, all objects are displayed in the dendrogram. In this case, `T` is the identity map, i.e., `T = (1:m)'`, where each node contains only a single object.

`[H,T,perm] = dendrogram(...)` generates a dendrogram and returns the permutation vector of the node labels of the leaves of the dendrogram. `perm` is ordered from left to right on a horizontal dendrogram and bottom to top for a vertical dendrogram.

`[...] = dendrogram(...,'colorthreshold',t)` assigns a unique color to each group of nodes in the dendrogram where the linkage is less than the threshold `t`. `t` is a value in the interval `[0,max(Z(:,3))]`. Setting `t` to the string `'default'` is the same as `t` = .7(`max(Z(:,3))`). `0` is the same as not specifying `'colorthreshold'`. The value `max(Z(:,3))` treats the entire tree as one group and colors it all one color.

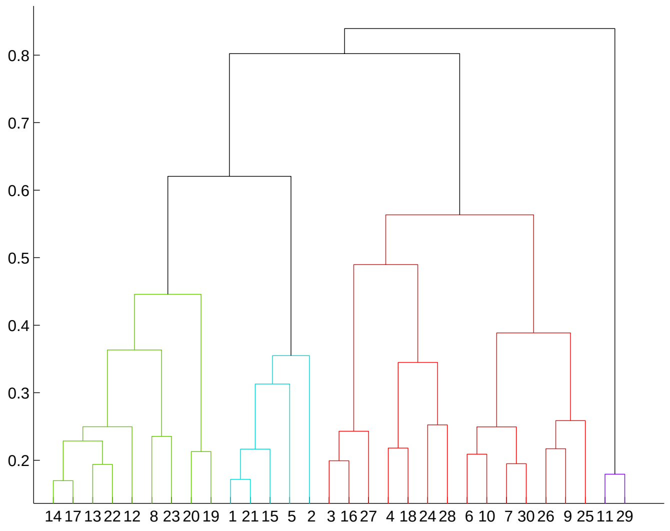
`[...] = dendrogram(...,'orientation','`*`orient`*`')` orients the dendrogram within the figure window. The options for '*`orient`*' are

`'top'`	Top to bottom (default)
`'bottom'`	Bottom to top
`'left'`	Left to right
`'right'`	Right to left

`[...] = dendrogram(...,'labels',S)` accepts a character array or cell array of strings `S` with one label for each observation. Any leaves in the tree containing a single observation are labeled with that observation's label.

Example

```
X= rand(100,2);
Y= pdist(X,'cityblock');
Z= linkage(Y,'average');
[H,T] = dendrogram(Z,'colorthreshold','default');
```



```
find(T==20)
ans =
    20
    49
    62
    65
    73
    96
```

This output indicates that leaf node 20 in the dendrogram contains the original data points 20, 49, 62, 65, 73, and 96.

See Also

`cluster`, `clusterdata`, `cophenet`, `inconsistent`, `linkage`, `silhouette`

Purpose

Interactive fitting of distributions to data

Syntax

```
dfittool
dfittool(y)
dfittool(y,cens)
dfittool(y,cens,freq)
dfittool(y,cens,freq,dsname)
```

Description

`dfittool` opens a graphical user interface for displaying fit distributions to data. To fit distributions to your data and display them over plots over plots of the empirical distributions, you can import data from the workspace.

`dfittool(y)` displays the Distribution Fitting Tool and creates a data set with data specified by the vector `y`.

`dfittool(y,cens)` uses the vector `cens` to specify whether the observation `y(j)` is censored, `(cens(j)==1)` and/or observed, exactly `(cens(j)==0)`. If `cens` is omitted or empty, no `y` values are censored.

`dfittool(y,cens,freq)` uses the vector `freq` to specify the frequency of each element of `y`. If `freq` is omitted or empty, all `y` values have a frequency of 1.

`dfittool(y,cens,freq,dsname)` creates a data set with the name `dsname` using the data vector `y`, censoring indicator `cens`, and frequency vector `freq`.

For more information, see “Distribution Fitting Tool” on page 5-122.

See Also

`mle`,`randtool`,`disttool`

disttool

Purpose Interactive pdf and cdf plots

Syntax `disttool`

Description `disttool` is a graphical interface for exploring the effects of changing parameters on the plot of a cdf or pdf.

See Also `randtool`, `dfittool`

Purpose

Drop levels from categorical array

Syntax

```
B = droplevels(A)
B = droplevels(A,oldlevels)
```

Description

`B = droplevels(A)` removes unused levels from the categorical array `A`. `B` is a categorical array with the same size and values as `A`, but with a list of potential levels that includes only those present in some element of `A`.

`B = droplevels(A,oldlevels)` removes specified levels from the categorical array `A`. `oldlevels` is a cell array of strings or a two-dimensional character matrix specifying the levels to be removed.

`droplevels` removes levels, but does not remove elements. Elements of `B` that correspond to elements of `A` having levels in `oldlevels` all have an undefined level.

Examples

Example 1

Drop unused age levels from the data in `hospital.mat`:

```
load hospital
edges = 0:10:100;
labels = strcat(num2str((0:10:90)','%d'),{'s'});
AgeGroup = ordinal(hospital.Age,labels,[],edges);
AgeGroup = droplevels(AgeGroup);
getlabels(AgeGroup)
ans =
    '20s'    '30s'    '40s'    '50s'
```

Example 2

1. Load patient data from the CSV file `hospital.dat` and store the information in a dataset array with observation names given by the first column in the data (patient identification):

```
patients = dataset('file','hospital.dat',...
                   'delimiter',',',...
```

```
                           'ReadObsNames',true);
```

2 Make the {0,1}-valued variable `smoke` nominal, and change the labels to `'No'` and `'Yes'`:

```
patients.smoke = nominal(patients.smoke,{'No','Yes'});
```

3 Add new levels to `smoke` as placeholders for more detailed histories of smokers:

```
patients.smoke = addlevels(patients.smoke,...
                 {'0-5 Years','5-10 Years','LongTerm'});
```

4 Assuming the nonsmokers have never smoked, relabel the `'No'` level:

```
patients.smoke = setlabels(patients.smoke,'Never','No');
```

5 Drop the undifferentiated `'Yes'` level from `smoke`:

```
patients.smoke = droplevels(patients.smoke,'Yes');

Warning: OLDLEVELS contains categorical levels that
were present in A, caused some array elements to have
undefined levels.
```

Note that smokers now have an undefined level.

6 Set each smoker to one of the new levels, by observation name:

```
patients.smoke('YPL-320') = '5-10 Years';
```

See Also

`addlevels`, `islevel`, `mergelevels`, `reorderlevels`, `getlabels`

Purpose

{0,1}-valued matrix of dummy variables

Syntax

```
D = dummyvar(group)
```

Description

`D = dummyvar(group)` generates a matrix, `D`, of {`0, 1`}-valued columns. `D` has one column for each unique value in each column of the matrix `group`. `group` can be a categorical variable, a cell array of multiple categorical variables, or a matrix of grouping variable values. (See "Grouped Data" on page 2-41.) If `group` is a matrix, the values of the elements in any column go from `1` to the number of members in the group defined by that column.

Example

Suppose you are studying the effects of two machines and three operators on a process. The first column of `group` would have the values `1` or `2` depending on which machine was used. The second column of `group` would have the values `1`, `2`, or `3` depending on which operator ran the machine.

```
group = [1 1;1 2;1 3;2 1;2 2;2 3];
D = dummyvar(group)
D =
   1   0   1   0   0
   1   0   0   1   0
   1   0   0   0   1
   0   1   1   0   0
   0   1   0   1   0
   0   1   0   0   1
```

See Also

`pinv`, `regress`

Purpose

Durbin-Watson test

Syntax

```
[P,DW] = dwtest(R,X)
[...] = dwtest(R,X,method)
[...] = dwtest(R,X,method,tail)
```

Description

`[P,DW] = dwtest(R,X)` performs a Durbin-Watson test on the vector `R` of residuals from a linear regression, where `X` is the design matrix from that linear regression. `P` is the computed p-value for the test, and `DW` is the Durbin-Watson statistic. The Durbin-Watson test is used to test if the residuals are independent, against the alternative that there is autocorrelation among them.

`[...]  = dwtest(R,X,method)` specifies the method to be used in computing the p-value. *`method`* can be either of the following:

- `'exact'` — Calculates an exact p-value using the PAN algorithm (the default if the sample size is less than 400).
- `'approximate'` — Calculates the p-value using a normal approximation (the default if the sample size is 400 or larger).

`[...]  = dwtest(R,X,method,tail)` performs the test against the alternative hypothesis specified by *`tail`*:

`'both'`	Serial correlation is not 0.
`'right'`	Serial correlation is greater than 0 (right-tailed test).
`'left'`	Serial correlation is less than 0 (left-tailed test).

See Also

`regress`

Purpose

Empirical cumulative distribution function

Syntax

```
[f,x] = ecdf(y)
[f,x,flo,fup] = ecdf(y)
ecdf(...)
ecdf(ax,...)
[...] = ecdf(y,param1,val1,param2,val2,...)
```

Description

`[f,x] = ecdf(y)` calculates the Kaplan-Meier estimate of the cumulative distribution function (cdf), also known as the empirical cdf. `y` is a vector of data values. `f` is a vector of values of the empirical cdf evaluated at `x`.

`[f,x,flo,fup] = ecdf(y)` also returns lower and upper confidence bounds for the cdf. These bounds are calculated using Greenwood's formula, and are not simultaneous confidence bounds.

`ecdf(...)` without output arguments produces a plot of the empirical cdf.

`ecdf(ax,...)` plots into axes `ax` instead of `gca`.

`[...] = ecdf(y,param1,val1,param2,val2,...)` specifies additional parameter/value pairs chosen from the following:

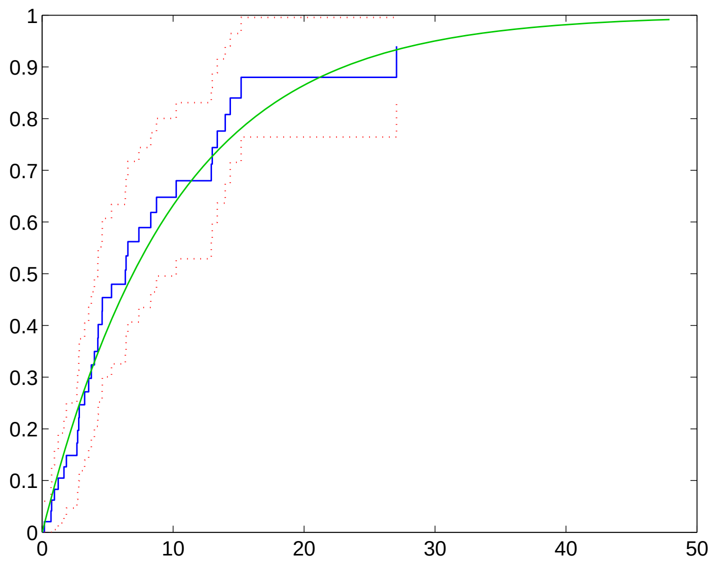
`'censoring'`	Boolean vector of the same size as `x`. Elements are `1` for observations that are right-censored and `0` for observations that are observed exactly. Default is all observations observed exactly.
`'frequency'`	Vector of the same size as `x` containing nonnegative integer counts. The `j`th element of this vector gives the number of times the `j`th element of `x` was observed. Default is `1` observation per element of `x`.
`'alpha'`	Value between `0` and `1` for a confidence level of `100(1-alpha)`%. Default is `alpha=0.05` for 95% confidence.

`'function'`	Type of function returned as the `f` output argument, chosen from `'cdf'` (default), `'survivor'`, or `'cumulative hazard'`.
`'bounds'`	Either `'on'` to include bounds, or `'off'` (the default) to omit them. Used only for plotting.

Examples

Generate random failure times and random censoring times, and compare the empirical cdf with the known true cdf.

```
y = exprnd(10,50,1); % Random failure times exponential(10)
d = exprnd(20,50,1); % Drop-out times exponential(20)
t = min(y,d);    % Observe the minimum of these times
censored = (y>d);  % Observe whether the subject failed

% Calculate and plot empirical cdf and confidence bounds
[f,x,flo,fup] = ecdf(t,'censoring',censored);
stairs(x,f);
hold on;
stairs(x,flo,'r:'); stairs(x,fup,'r:');

% Superimpose a plot of the known true cdf
xx = 0:.1:max(t); yy = 1-exp(-xx/10); plot(xx,yy,'g-')
hold off;
```



References

[1] Cox, D. R., and D. Oakes, *Analysis of Survival Data*, Chapman & Hall, London, 1984.

See Also

`cdfplot`, `ecdfhist`

Purpose

Create histogram from output of `ecdf`

Syntax

```
n = ecdfhist(f,x)
n = ecdfhist(f,x,m)
n = ecdfhist(f,x,c)
[n,c] = ecdfhist(...)
ecdfhist(...)
```

Description

`n = ecdfhist(f,x)` takes a vector `f` of empirical cumulative distribution function (cdf) values and a vector `x` of evaluation points, and returns a vector `n` containing the heights of histogram bars for 10 equally spaced bins. The function computes the bar heights from the increases in the empirical cdf, and normalizes them so that the area of the histogram is equal to 1. In contrast, `hist` produces bars whose heights represent bin counts.

`n = ecdfhist(f,x,m)`, where `m` is a scalar, uses `m` bins.

`n = ecdfhist(f,x,c)`, where `c` is a vector, uses bins with centers specified by `c`.

`[n,c] = ecdfhist(...)` also returns the position of the bin centers in `c`.

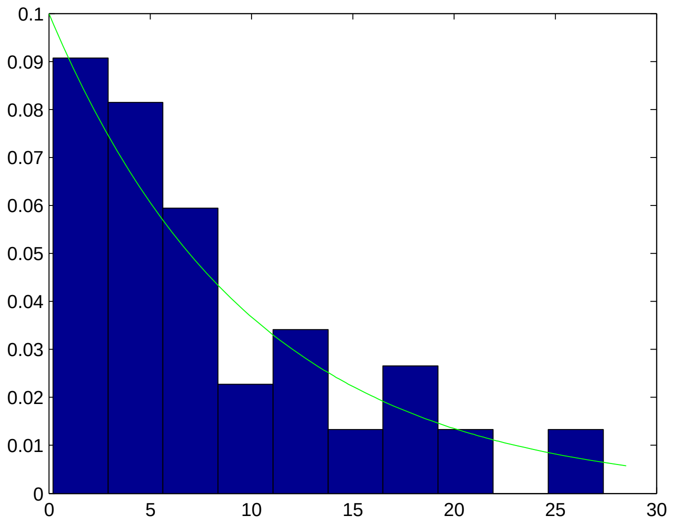
`ecdfhist(...)` without output arguments produces a histogram bar plot of the results.

Example

The following code generates random failure times and random censoring times, and compares the empirical pdf with the known true pdf.

```
y = exprnd(10,50,1); % Random failure times
d = exprnd(20,50,1); % Drop-out times
t = min(y,d);        % Observe the minimum of these times
censored = (y>d);    % Observe whether the subject failed

% Calculate the empirical cdf and plot a histogram from it
[f,x] = ecdf(t,'censoring',censored);
ecdfhist(f,x);
```

```
% Superimpose a plot of the known true pdf
hold on;
xx = 0:.1:max(t); yy = exp(-xx/10)/10; plot(xx,yy,'g-');
hold off;
```



See Also

`ecdf`, `hist`, `histc`

errorbar

Purpose

Plot error bars along curve

Syntax

```
errorbar(X,Y,L,U,symbol)
errorbar(X,Y,L)
errorbar(Y,L)
```

Description

`errorbar(X,Y,L,U,`*`symbol`*`)` plots `X` versus `Y` with error bars specified by `L` and `U`. `X`, `Y`, `L`, and `U` must be the same length. If `X`, `Y`, `L`, and `U` are matrices, then each column produces a separate line. The error bars are each drawn a distance of `U(i)` above and `L(i)` below the points in `(X,Y)`. *`symbol`* is a string that controls the line type, plotting symbol, and color of the error bars.

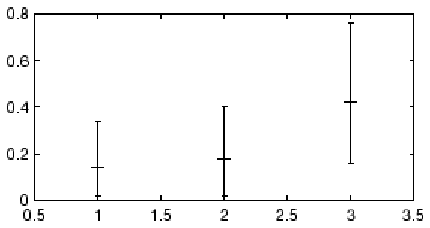
`errorbar(X,Y,L)` plots `X` versus `Y` with symmetric error bars about `Y`.

`errorbar(Y,L)` plots `Y` with error bars `[Y-L Y+L]`.

The `errorbar` function is a part of the standard MATLAB language.

Example

```
  lambda = (0.1:0.2:0.5);
  r = poissrnd(lambda(ones(50,1),:));
  [p,pci] = poissfit(r,0.001);
  L = p - pci(1,:)
  L =
    0.1200  0.1600  0.2600
  U = pci(2,:)-p
  U =
    0.2000  0.2200  0.3400
  errorbar(1:3,p,L,U,'+')
```



Purpose

Predicted responses for tree

Syntax

```
yfit = eval(t,X)
yfit = eval(t,X,s)
[yfit,nodes] = eval(...)
[yfit,nodes,cnums] = eval(...)
[...] = t(X)
[...] = t(X,s)
```

Description

`yfit = eval(t,X)` takes a classification or regression tree `t` and a matrix `X` of predictors, and produces a vector `yfit` of predicted response values. For a regression tree, `yfit(i)` is the fitted response value for a point having the predictor values `X(i,:)`. For a classification tree, `yfit(i)` is the class into which the tree assigns the point with data `X(i,:)`.

`yfit = eval(t,X,s)` takes an additional vector `s` of pruning levels, with 0 representing the full, unpruned tree. `t` must include a pruning sequence as created by `classregtree` or by `prune`. If `s` has *k* elements and `X` has *n* rows, the output `yfit` is an *n*-by-*k* matrix, with the `j`th column containing the fitted values produced by the `s(j)` subtree. `s` must be sorted in ascending order.

To compute fitted values for a tree that is not part of the optimal pruning sequence, first use `prune` to prune the tree.

`[yfit,nodes] = eval(...)` also returns a vector `nodes` the same size as `yfit` containing the node number assigned to each row of `X`. Use `view` to display the node numbers for any node you select.

`[yfit,nodes,cnums] = eval(...)` is valid only for classification trees. It returns a vector `cnum` containing the predicted class numbers.

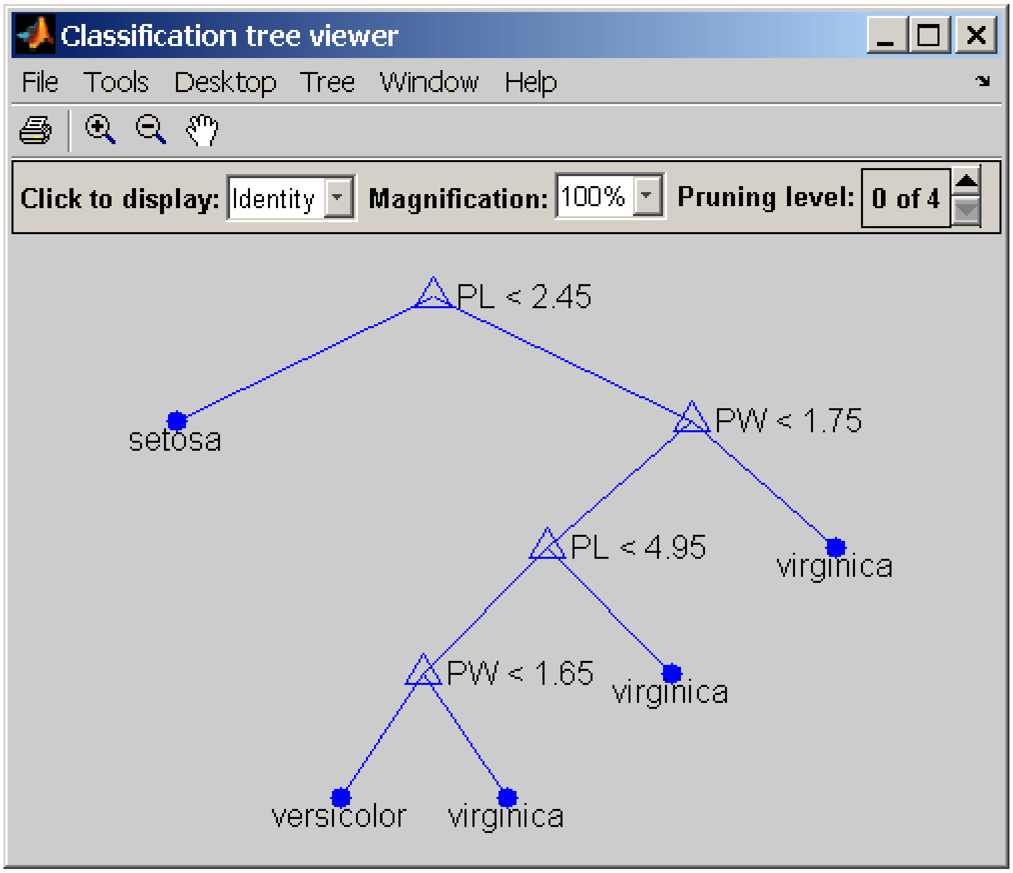
`NaN` values in `X` are treated as missing. If `eval` encounters a missing value when it attempts to evaluate the split rule at a branch node, it cannot determine whether to proceed to the left or right child node. Instead, it sets the corresponding fitted value equal to the fitted value assigned to the branch node.

`[...]  = t(X)` or `[...]  = t(X,s)` also invoke `eval`.

Example

Create a classification tree for Fisher's iris data:

```
load fisheriris;

t = classregtree(meas,species,'names',{'SL' 'SW' 'PL' 'PW'})
t =
Decision tree for classification
1  if PL<2.45 then node 2 else node 3
2  class = setosa
3  if PW<1.75 then node 4 else node 5
4  if PL<4.95 then node 6 else node 7
5  class = virginica
6  if PW<1.65 then node 8 else node 9
7  class = virginica
8  class = versicolor
9  class = virginica

view(t)
```



Find assigned class names:

```
sfit = eval(t,meas);
```

Compute proportion correctly classified:

```
pct = mean(strcmp(sfit,species))
pct =
    0.9800
```

Reference

[1] Breiman, L., et al., *Classification and Regression Trees*, Chapman & Hall, Boca Raton, 1993.

See Also

`classregtree`, `prune`, `view`, `test`

Purpose

Extreme value cumulative distribution function

Syntax

```
P = evcdf(X,mu,sigma)
[P,PLO,PUP] = evcdf(X,mu,sigma,pcov,alpha)
```

Description

`P = evcdf(X,mu,sigma)` computes the cumulative distribution function (cdf) for the type 1 extreme value distribution, with location parameter `mu` and scale parameter `sigma`, at each of the values in `X`. `X`, `mu`, and `sigma` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array of the same size as the other inputs. The default values for `mu` and `sigma` are `0` and `1`, respectively.

`[P,PLO,PUP] = evcdf(X,mu,sigma,pcov,alpha)` produces confidence bounds for `P` when the input parameters `mu` and `sigma` are estimates. `pcov` is a 2-by-2 covariance matrix of the estimated parameters. `alpha` has a default value of `0.05`, and specifies `100(1 - alpha)%` confidence bounds. `PLO` and `PUP` are arrays of the same size as `P`, containing the lower and upper confidence bounds.

The function `evcdf` computes confidence bounds for `P` using a normal approximation to the distribution of the estimate

$$\frac{X - \hat{\mu}}{\hat{\sigma}}$$

and then transforming those bounds to the scale of the output `P`. The computed bounds give approximately the desired confidence level when you estimate `mu`, `sigma`, and `pcov` from large samples, but in smaller samples other methods of computing the confidence bounds might be more accurate.

The type 1 extreme value distribution is also known as the Gumbel distribution. If x has a Weibull distribution, then $X = \log(x)$ has the type 1 extreme value distribution.

See Also

`cdf`, `evfit`, `evinv`, `evlike`, `evpdf`, `evrnd`, `evstat`

evfit

Purpose

Parameter estimates and confidence intervals for extreme value distributed data

Syntax

```
parmhat = evfit(data)
[parmhat,parmci] = evfit(data)
[parmhat,parmci] = evfit(data,alpha)
[...] = evfit(data,alpha,censoring)
[...] = evfit(data,alpha,censoring,freq)
[...] = evfit(data,alpha,censoring,freq,options)
```

Description

`parmhat = evfit(data)` returns maximum likelihood estimates of the parameters of the type 1 extreme value distribution given the data in the vector `data`. `parmhat(1)` is the location parameter, , and `parmhat(2)` is the scale parameter, σ.

`[parmhat,parmci] = evfit(data)` returns 95% confidence intervals for the parameter estimates on the   and σ parameters in the 2-by-2 matrix `parmci`. The first column of the matrix of the extreme value fit contains the lower and upper confidence bounds for the parameter , and the second column contains the confidence bounds for the parameter σ.

`[parmhat,parmci] = evfit(data,alpha)` returns 100(1 - `alpha`)% confidence intervals for the parameter estimates, where `alpha` is a value in the range `[0 1]` specifying the width of the confidence intervals. By default, `alpha` is `0.05`, which corresponds to 95% confidence intervals.

`[...]  = evfit(data,alpha,censoring)` accepts a Boolean vector, `censoring`, of the same size as `data`, which is `1` for observations that are right-censored and `0` for observations that are observed exactly.

`[...]  = evfit(data,alpha,censoring,freq)` accepts a frequency vector, `freq` of the same size as `data`. Typically, `freq` contains integer frequencies for the corresponding elements in `data`, but can contain any nonnegative values. Pass in `[]` for `alpha`, `censoring`, or `freq` to use their default values.

`[...]  = evfit(data,alpha,censoring,freq,options)` accepts a structure, `options`, that specifies control parameters for the iterative algorithm the function uses to compute maximum likelihood estimates. You can create `options` using the function `statset`. Enter `statset('evfit')` to see the names and default values of the parameters that `evfit` accepts in the `options` structure. See the reference page for `statset` for more information about these options.

The type 1 extreme value distribution is also known as the Gumbel distribution. If x has a Weibull distribution, then $X = \log(x)$ has the type 1 extreme value distribution.

See Also

`evcdf`, `evinv`, `evlike`, `evpdf`, `evrnd`, `evstat`, `mle`, `statset`

Purpose

Inverse of extreme value cumulative distribution function

Syntax

```
X = evinv(P,mu,sigma)
[X,XLO,XUP] = evinv(P,mu,sigma,pcov,alpha)
```

Description

`X = evinv(P,mu,sigma)` returns the inverse cumulative distribution function (cdf) for a type 1 extreme value distribution with location parameter `mu` and scale parameter `sigma`, evaluated at the values in `P`. `P`, `mu`, and `sigma` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array of the same size as the other inputs. The default values for `mu` and `sigma` are `0` and `1`, respectively.

`[X,XLO,XUP] = evinv(P,mu,sigma,pcov,alpha)` produces confidence bounds for `X` when the input parameters `mu` and `sigma` are estimates. `pcov` is the covariance matrix of the estimated parameters. `alpha` is a scalar that specifies 100(1 - `alpha`)% confidence bounds for the estimated parameters, and has a default value of 0.05. `XLO` and `XUP` are arrays of the same size as `X` containing the lower and upper confidence bounds.

The function `evinv` computes confidence bounds for `P` using a normal approximation to the distribution of the estimate

$$\hat{\mu} + \hat{\sigma}q$$

where q is the `P`th quantile from an extreme value distribution with parameters $\mu = 0$ and $\sigma = 1$. The computed bounds give approximately the desired confidence level when you estimate `mu`, `sigma`, and `pcov` from large samples, but in smaller samples other methods of computing the confidence bounds might be more accurate.

The type 1 extreme value distribution is also known as the Gumbel distribution. If x has a Weibull distribution, then $X = \log(x)$ has the type 1 extreme value distribution.

See Also

`evcdf`, `evfit`, `evlike`, `evpdf`, `evrnd`, `evstat`, `icdf`

Purpose

Negative log-likelihood for extreme value distribution

Syntax

```
nlogL = evlike(params,data)
[nlogL,AVAR] = evlike(params,data)
[...] = evlike(params,data,censoring)
[...] = evlike(params,data,censoring,freq)
```

Description

`nlogL = evlike(params,data)` returns the negative of the log-likelihood for the type 1 extreme value distribution, evaluated at parameters `params(1) = mu` and `params(2) = sigma`, given `data`. `nlogL` is a scalar.

`[nlogL,AVAR] = evlike(params,data)` returns the inverse of Fisher's information matrix, `AVAR`. If the input parameter values in `params` are the maximum likelihood estimates, the diagonal elements of `AVAR` are their asymptotic variances. `AVAR` is based on the observed Fisher's information, not the expected information.

`[...] = evlike(params,data,censoring)` accepts a Boolean vector of the same size as `data`, which is 1 for observations that are right-censored and 0 for observations that are observed exactly.

`[...] = evlike(params,data,censoring,freq)` accepts a frequency vector of the same size as `data`. `freq` typically contains integer frequencies for the corresponding elements in `data`, but can contain any nonnegative values. Pass in `[]` for `censoring` to use its default value.

The type 1 extreme value distribution is also known as the Gumbel distribution. If x has a Weibull distribution, then $X = \log(x)$ has the type 1 extreme value distribution.

See Also

`evcdf`, `evfit`, `evinv`, `evpdf`, `evrnd`, `evstat`

Purpose

Extreme value probability density function

Syntax

```
Y = evpdf(X,mu,sigma)
```

Description

`Y = evpdf(X,mu,sigma)` returns the pdf of the type 1 extreme value distribution with location parameter `mu` and scale parameter `sigma`, evaluated at the values in `X`. `X`, `mu`, and `sigma` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array of the same size as the other inputs. The default values for `mu` and `sigma` are `0` and `1`, respectively.

The type 1 extreme value distribution is also known as the Gumbel distribution. If x has a Weibull distribution, then $X = \log(x)$ has the type 1 extreme value distribution.

See Also

`evcdf`, `evfit`, `evinv`, `evlike`, `evrnd`, `evstat`, `pdf`

Purpose

Random numbers from extreme value distribution

Syntax

```
R = evrnd(mu,sigma)
R = evrnd(mu,sigma,v)
R = evrnd(mu,sigma,m,n)
```

Description

`R = evrnd(mu,sigma)` generates random numbers from the extreme value distribution with parameters specified by `mu` and `sigma`. `mu` and `sigma` can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of R. A scalar input for `mu` or `sigma` is expanded to a constant array with the same dimensions as the other input.

`R = evrnd(mu,sigma,v)` generates an array `R` of size `v` containing random numbers from the extreme value distribution with parameters `mu` and `sigma`, where `v` is a row vector. If `v` is a 1-by-2 vector, `R` is a matrix with `v(1)` rows and `v(2)` columns. If `v` is 1-by-n, `R` is an n-dimensional array.

If `mu` and `sigma` are both scalars, `R = evrnd(mu,sigma,m,n)` returns an `m`-by-`n` matrix.

The type 1 extreme value distribution is also known as the Gumbel distribution. If x has a Weibull distribution, then $X = \log(x)$ has the type 1 extreme value distribution.

See Also

`evcdf`, `evfit`, `evinv`, `evlike`, `evpdf`, `evstat`

Purpose Mean and variance of extreme value distribution

Syntax

```
[M,V] = evstat(mu,sigma)
```

Description `[M,V] = evstat(mu,sigma)` returns the mean of and variance for the type 1 extreme value distribution with location parameter `mu` and scale parameter `sigma`. `mu` and `sigma` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array of the same size as the other input. The default values for `mu` and `sigma` are `0` and `1`, respectively.

The type 1 extreme value distribution is also known as the Gumbel distribution. If x has a Weibull distribution, then $X = \log(x)$ has the type 1 extreme value distribution.

See Also `evcdf`, `evfit`, `evinv`, `evlike`, `evpdf`, `evrnd`

Purpose

Exponential cumulative distribution function

Syntax

```
P = expcdf(X,mu)
[P,PLO,PUP] = expcdf(X,mu,pcov,alpha)
```

Description

`P = expcdf(X,mu)` computes the exponential cdf at each of the values in `X` using the corresponding parameters in `mu`. `X` and `mu` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other input. The parameters in `mu` must be positive.

The exponential cdf is

$$p = F(x|\mu) = \int_0^x \frac{1}{\mu} e^{-\frac{t}{\mu}} dt = 1 - e^{-\frac{x}{\mu}}$$

The result, p, is the probability that a single observation from an exponential distribution will fall in the interval [0 x].

`[P,PLO,PUP] = expcdf(X,mu,pcov,alpha)` produces confidence bounds for `P` when the input parameter `mu` is an estimate. `pcov` is the variance of the estimated `mu`. `alpha` specifies 100(1 - `alpha`)% confidence bounds. The default value of `alpha` is 0.05. `PLO` and `PUP` are arrays of the same size as `P` containing the lower and upper confidence bounds. The bounds are based on a normal approximation for the distribution of the log of the estimate of `mu`. If you estimate `mu` from a set of data, you can get a more accurate set of bounds by applying `expfit` to the data to get a confidence interval for `mu`, and then evaluating `expinv` at the lower and upper endpoints of that interval.

Examples

The following code shows that the median of the exponential distribution is μ*log(2).

```
mu = 10:10:60;
p = expcdf(log(2)*mu,mu)
p =
  0.5000  0.5000  0.5000  0.5000  0.5000  0.5000
```

What is the probability that an exponential random variable is less than or equal to the mean, µ?

```
mu = 1:6;
x = mu;
p = expcdf(x,mu)
p =
  0.6321  0.6321  0.6321  0.6321  0.6321  0.6321
```

See Also

`cdf`, `expfit`, `expinv`, `exppdf`, `exprnd`, `expstat`

Purpose

Parameter estimates and confidence intervals for exponentially distributed data

Syntax

```
parmhat = expfit(data)
[parmhat,parmci] = expfit(data)
[parmhat,parmci] = expfit(data,alpha)
[...] = expfit(data,alpha,censoring)
[...] = expfit(data,alpha,censoring,freq)
```

Description

`parmhat = expfit(data)` returns estimates of the parameter, μ, of the exponential distribution, given the data in `data`. Each entry of `parmhat` corresponds to the data in a column of `data`.

`[parmhat,parmci] = expfit(data)` returns 95% confidence intervals for the parameter estimates in matrix `parmci`. The first row of `parmci` contains the lower bounds of the confidence intervals, and the second row contains the upper bounds.

`[parmhat,parmci] = expfit(data,alpha)` returns 100(1 - `alpha`)% confidence intervals for the parameter estimates, where `alpha` is a value in the range `[0 1]` specifying the width of the confidence intervals. By default, `alpha` is `0.05`, which corresponds to 95% confidence intervals.

`[...]  = expfit(data,alpha,censoring)` accepts a Boolean vector, `censoring`, of the same size as `data`, which is 1 for observations that are right-censored and 0 for observations that are observed exactly. `data` must be a vector in order to pass in the argument `censoring`.

`[...]  = expfit(data,alpha,censoring,freq)` accepts a frequency vector, `freq` of the same size as `data`. Typically, `freq` contains integer frequencies for the corresponding elements in `data`, but can contain any nonnegative values. Pass in `[]` for `alpha`, `censoring`, or `freq` to use their default values.

Example

This example generates 100 independent samples of exponential data with μ = 3. `muhat` is an estimate of μ and `muci` is a 99% confidence interval around `muhat`. Notice that `muci` contains the true value of μ.

```
  data = exprnd(3, 100, 1);
```

```
[parmhat, parmci] = expfit(data, 0.01)
parmhat =
  2.7292
parmci =
  2.1384
  3.5854
```

See Also `expcdf`, `expinv`, `explike`, `exppdf`, `exprnd`, `expstat`, `mle`, `statset`

Purpose

Inverse of exponential cumulative distribution function

Syntax

```
X = expinv(P,mu)
[X,XLO,XUP] = expinv(X,mu,pcov,alpha)
```

Description

`X = expinv(P,mu)` computes the inverse of the exponential cdf with parameters specified by `mu` for the corresponding probabilities in `P`. `P` and `mu` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other input. The parameters in `mu` must be positive and the values in `P` must lie on the interval [0 1].

`[X,XLO,XUP] = expinv(X,mu,pcov,alpha)` produces confidence bounds for `X` when the input parameter `mu` is an estimate. `pcov` is the variance of the estimated `mu`. `alpha` specifies 100(1 - `alpha`)% confidence bounds. The default value of `alpha` is 0.05. `XLO` and `XUP` are arrays of the same size as `X` containing the lower and upper confidence bounds. The bounds are based on a normal approximation for the distribution of the log of the estimate of `mu`. If you estimate `mu` from a set of data, you can get a more accurate set of bounds by applying `expfit` to the data to get a confidence interval for `mu`, and then evaluating `expinv` at the lower and upper end points of that interval.

The inverse of the exponential cdf is

$$x = F^{-1}(p|\mu) = -\mu\ln(1-p)$$

The result, x, is the value such that an observation from an exponential distribution with parameter μ will fall in the range [0 x] with probability p.

Examples

Let the lifetime of light bulbs be exponentially distributed with μ = 700 hours. What is the median lifetime of a bulb?

```
expinv(0.50,700)
ans =
 485.2030
```

Suppose you buy a box of “700 hour” light bulbs. If 700 hours is the mean life of the bulbs, half of them will burn out in less than 500 hours.

See Also

`expcdf`, `expfit`, `exppdf`, `exprnd`, `expstat`, `icdf`

Purpose

Negative log-likelihood for exponential distribution

Syntax

```
nlogL = explike(param,data)
[nlogL,avar] = explike(param,data)
[...] = explike(param,data,censoring)
[...] = explike(param,data,censoring,freq)
```

Description

`nlogL = explike(param,data)` returns the negative of the log-likelihood for the exponential distribution, evaluated at the parameter `param = mu`, given `data`. `nlogL` is a scalar.

`[nlogL,avar] = explike(param,data)` returns the inverse of Fisher's information, `avar`, a scalar. If the input parameter value in `param` is the maximum likelihood estimate, `avar` is its asymptotic variance. `avar` is based on the observed Fisher's information, not the expected information.

`[...]  = explike(param,data,censoring)` accepts a Boolean vector, `censoring`, of the same size as `data`, which is `1` for observations that are right-censored and `0` for observations that are observed exactly.

`[...]  = explike(param,data,censoring,freq)` accepts a frequency vector, `freq`, of the same size as `data`. The vector `freq` typically contains integer frequencies for the corresponding elements in `data`, but can contain any nonnegative values. Pass in `[]` for `censoring` to use its default value.

See Also

`expcdf`, `expfit`, `expinv`, `exppdf`, `exprnd`

exppdf

Purpose

Exponential probability density function

Syntax

```
Y = exppdf(X,mu)
```

Description

`Y = exppdf(X,mu)` computes the exponential pdf at each of the values in `X` using the corresponding parameters in `mu`. `X` and `mu` can be vectors, matrices, or multidimensional arrays that have the same size. A scalar input is expanded to a constant array with the same dimensions as the other input. The parameters in `mu` must be positive.

The exponential pdf is

$$y = f(x|\mu) = \frac{1}{\mu}e^{-\frac{x}{\mu}}$$

The exponential pdf is the gamma pdf with its first parameter equal to 1.

The exponential distribution is appropriate for modeling waiting times when the probability of waiting an additional period of time is independent of how long you have already waited. For example, the probability that a light bulb will burn out in its next minute of use is relatively independent of how many minutes it has already burned.

Examples

```
y = exppdf(5,1:5)
y =
  0.0067  0.0410  0.0630  0.0716  0.0736

y = exppdf(1:5,1:5)
y =
  0.3679  0.1839  0.1226  0.0920  0.0736
```

See Also

`expcdf`, `expfit`, `expinv`, `exprnd`, `expstat`, `pdf`

Purpose

Random numbers from exponential distribution

Syntax

```
R = exprnd(mu)
R = exprnd(mu,v)
R = exprnd(mu,m,n)
```

Description

`R = exprnd(mu)` generates random numbers from the exponential distribution with mean parameter `mu`. `mu` can be a vector, a matrix, or a multidimensional array. The size of `R` is the size of `mu`.

`R = exprnd(mu,v)` generates an array `R` of size `v` containing random numbers from the exponential distribution with mean `mu`, where `v` is a row vector. If `v` is a 1-by-2 vector, `R` is a matrix with `v(1)` rows and `v(2)` columns. If `v` is 1-by-n, `R` is an n-dimensional array.

`R = exprnd(mu,m,n)` generates random numbers from the exponential distribution with mean parameter `mu`, where scalars `m` and `n` are the row and column dimensions of `R`.

Examples

```
n1 = exprnd(5:10)
n1 =
  7.5943  18.3400  2.7113  3.0936  0.6078  9.5841

n2 = exprnd(5:10,[1 6])
n2 =
  3.2752  1.1110  23.5530  23.4303  5.7190  3.9876

n3 = exprnd(5,2,3)
n3 =
  24.3339  13.5271  1.8788
   4.7932   4.3675  2.6468
```

See Also

`expcdf`, `expfit`, `expinv`, `exppdf`, `expstat`

Purpose

Mean and variance of exponential distribution

Syntax

```
[m,v] = expstat(mu)
```

Description

`[m,v] = expstat(mu)` returns the mean of and variance for the exponential distribution with parameters `mu`. `mu` can be a vectors, matrix, or multidimensional array. The mean of the exponential distribution is μ, and the variance is μ^2.

Examples

```
[m,v] = expstat([1 10 100 1000])
m =
      1      10      100      1000
v =
      1      100     10000    1000000
```

See Also

`expcdf`, `expfit`, `expinv`, `exppdf`, `exprnd`

Purpose

Maximum likelihood common factor analysis

Syntax

```
lambda = factoran(X,m)
[lambda,psi] = factoran(X,m)
[lambda,psi,T] = factoran(X,m)
[lambda,psi,T,stats] = factoran(X,m)
[lambda,psi,T,stats,F] = factoran(X,m)
[...] = factoran(...,param1,val1,param2,val2,...)
```

Definition

`factoran` computes the maximum likelihood estimate (MLE) of the factor loadings matrix Λ in the factor analysis model

$$x = \mu + \Lambda f + e$$

where x is a vector of observed variables, μ is a constant vector of means, Λ is a constant d-by-m matrix of factor loadings, f is a vector of independent, standardized common factors, and e is a vector of independent specific factors. x, μ, and e are of length d. f is of length m.

Alternatively, the factor analysis model can be specified as

$$\operatorname{cov}(x) = \Lambda\Lambda^T + \Psi$$

where $\Psi = \operatorname{cov}(e)$ is a d-by-d diagonal matrix of specific variances.

Description

`lambda = factoran(X,m)` returns the maximum likelihood estimate, `lambda`, of the factor loadings matrix, in a common factor analysis model with `m` common factors. `X` is an `n`-by-`d` matrix where each row is an observation of `d` variables. The `(i,j)`th element of the `d`-by-`m` matrix `lambda` is the coefficient, or loading, of the `j`th factor for the `i`th variable. By default, `factoran` calls the function `rotatefactors` to rotate the estimated factor loadings using the `'varimax'` option.

`[lambda,psi] = factoran(X,m)` also returns maximum likelihood estimates of the specific variances as a column vector `psi` of length `d`.

`[lambda,psi,T] = factoran(X,m)` also returns the `m`-by-`m` factor loadings rotation matrix `T`.

`[lambda,psi,T,stats] = factoran(X,m)` also returns a structure `stats` containing information relating to the null hypothesis, H_0, that the number of common factors is `m`. `stats` includes the following fields:

`loglike`	Maximized log-likelihood value
`dfe`	Error degrees of freedom = `((d-m)^2 - (d+m))/2`
`chisq`	Approximate chi-squared statistic for the null hypothesis
`p`	Right-tail significance level for the null hypothesis

`factoran` does not compute the `chisq` and `p` fields unless `dfe` is positive and all the specific variance estimates in `psi` are positive (see "Heywood Case" on page 14-243 below). If `X` is a covariance matrix, then you must also specify the `'nobs'` parameter if you want `factoran` to compute the `chisq` and `p` fields.

`[lambda,psi,T,stats,F] = factoran(X,m)` also returns, in `F`, predictions of the common factors, known as factor scores. `F` is an `n`-by-`m` matrix where each row is a prediction of `m` common factors. If `X` is a covariance matrix, `factoran` cannot compute `F`. `factoran` rotates `F` using the same criterion as for `lambda`.

`[...]  = factoran(...,`*`param1`*`,`*`val1`*`,`*`param2`*`,`*`val2`*`,...)` enables you to specify optional parameter name/value pairs to control the model fit and the outputs. The following are the valid parameter/value pairs.

Parameter	**Value**	
`'xtype'`	Type of input in the matrix `X`. `'xtype'` can be one of:	
	`'data'`	Raw data (default)
	`'covariance'`	Positive definite covariance or correlation matrix

Parameter	Value	
'scores'	Method for predicting factor scores. 'scores' is ignored if X is not raw data.	
	’wls' 'Bartlett'	Synonyms for a weighted least squares estimate that treats F as fixed (default)
	'regression' 'Thomson'	Synonyms for a minimum mean squared error prediction that is equivalent to a ridge regression
'start'	Starting point for the specific variances psi in the maximum likelihood optimization. Can be specified as:	
	'random'	Chooses d uniformly distributed values on the interval [0,1].
	’Rsquared'	Chooses the starting vector as a scale factor times diag(inv(corrcoef(X))) (default). For examples, see Jöreskog [2].
	Positive integer	Performs the given number of maximum likelihood fits, each initialized as with 'random'. factoran returns the fit with the highest likelihood.
	Matrix	Performs one maximum likelihood fit for each column of the specified matrix. The ith optimization is initialized with the values from the ith column. The matrix must have d rows.
'rotate'	Method used to rotate factor loadings and scores. 'rotate' can have the same values as the 'Method' parameter of rotatefactors. See the reference page for rotatefactors for a full description of the available methods.	
	’none'	Performs no rotation.

Parameter	Value	
	`'equamax'`	Special case of the orthomax rotation. Use the `'normalize'`, `'reltol'`, and `'maxit'` parameters to control the details of the rotation.
	`'orthomax'`	Orthogonal rotation that maximizes a criterion based on the variance of the loadings. Use the `'coeff'`, `'normalize'`, `'reltol'`, and `'maxit`’ parameters to control the details of the rotation.
	`'parsimax'`	Special case of the orthomax rotation (default). Use the `'normalize'`, `'reltol'`, and ’maxit’ parameters to control the details of the rotation.
	`'pattern'`	Performs either an oblique rotation (the default) or an orthogonal rotation to best match a specified pattern matrix. Use the `'type'` parameter to choose the type of rotation. Use the `'target'` parameter to specify the pattern matrix.
	`’procrustes'`	Performs either an oblique (the default) or an orthogonal rotation to best match a specified target matrix in the least squares sense. Use the `'type'` parameter to choose the type of rotation. Use `'target'` to specify the target matrix.

Parameter	Value	
	`'promax'`	Performs an oblique procrustes rotation to a target matrix determined by `factoran` as a function of an orthomax solution. Use the `'power'` parameter to specify the exponent for creating the target matrix. Because `'promax'` uses `'orthomax'` internally, you can also specify the parameters that apply to `'orthomax'`.
	`'quartimax'`	Special case of the orthomax rotation (default). Use the `'normalize'`, `'reltol'`, and ’maxit’ parameters to control the details of the rotation.
	`'varimax'`	Special case of the orthomax rotation (default). Use the `'normalize'`, `'reltol'`, and `'maxit'` parameters to control the details of the rotation.

	Function	Function handle to rotation function of the form
		
		`[B,T] =`
		`myrotation(A,...)`
		
		where `A` is a `d`-by-`m` matrix of unrotated factor loadings, `B` is a `d`-by-`m` matrix of rotated loadings, and `T` is the corresponding `m`-by-`m` rotation matrix.
		
		Use the `factoran` parameter `'userargs'` to pass additional arguments to this rotation function. See Example 4.

Parameter	Value
'coeff'	Coefficient, often denoted as γ, defining the specific 'orthomax' criterion. Must be between 0 and 1. The value 0 corresponds to quartimax, and 1 corresponds to varimax. Default is 1.
'normalize'	Flag indicating whether the loading matrix should be row-normalized (1) or left unnormalized (0) for 'orthomax' or 'varimax' rotation. Default is 1.
'reltol'	Relative convergence tolerance for 'orthomax' or 'varimax' rotation. Default is sqrt(eps).
'maxit'	Iteration limit for 'orthomax' or 'varimax' rotation. Default is 250.
'target'	Target factor loading matrix for 'procrustes' rotation. Required for 'procrustes' rotation. No default value.
'type'	Type of 'procrustes' rotation. Can be 'oblique' (default) or 'orthogonal'.
'power'	Exponent for creating the target matrix in the 'promax' rotation. Must be $\geq$ 1. Default is 4.
'userargs'	Denotes the beginning of additional input values for a user-defined rotation function. factoran appends all subsequent values, in order and without processing, to the rotation function argument list, following the unrotated factor loadings matrix A. See Example 4.
'nobs'	If X is a covariance or correlation matrix, indicates the number of observations that were used in its estimation. This allows calculation of significance for the null hypothesis even when the original data are not available. There is no default. 'nobs' is ignored if X is raw data.

Parameter	Value
'delta'	Lower bound for the specific variances psi during the maximum likelihood optimization. Default is 0.005.
'optimopts'	Structure that specifies control parameters for the iterative algorithm the function uses to compute maximum likelihood estimates. Create this structure with the function statset. Enter statset('factoran') to see the names and default values of the parameters that factoran accepts in the options structure. See the reference page for statset for more information about these options.

Remarks

Observed Data Variables

The variables in the observed data matrix X must be linearly independent, i.e., cov(X) must have full rank, for maximum likelihood estimation to succeed. factoran reduces both raw data and a covariance matrix to a correlation matrix before performing the fit.

factoran standardizes the observed data X to zero mean and unit variance before estimating the loadings lambda. This does not affect the model fit, because MLEs in this model are invariant to scale. However, lambda and psi are returned in terms of the standardized variables, i.e., lambda*lambda'+diag(psi) is an estimate of the correlation matrix of the original data X (although not after an oblique rotation). See Examples 1 and 3.

Heywood Case

If elements of psi are equal to the value of the 'delta' parameter (i.e., they are essentially zero), the fit is known as a Heywood case, and interpretation of the resulting estimates is problematic. In particular, there can be multiple local maxima of the likelihood, each with different estimates of the loadings and the specific variances. Heywood cases can indicate overfitting (i.e., m is too large), but can also be the result of underfitting.

Rotation of Factor Loadings and Scores

Unless you explicitly specify no rotation using the `'rotate'` parameter, `factoran` rotates the estimated factor loadings, `lambda`, and the factor scores, `F`. The output matrix `T` is used to rotate the loadings, i.e., `lambda = lambda0*T`, where `lambda0` is the initial (unrotated) MLE of the loadings. `T` is an orthogonal matrix for orthogonal rotations, and the identity matrix for no rotation. The inverse of `T` is known as the primary axis rotation matrix, while `T` itself is related to the reference axis rotation matrix. For orthogonal rotations, the two are identical.

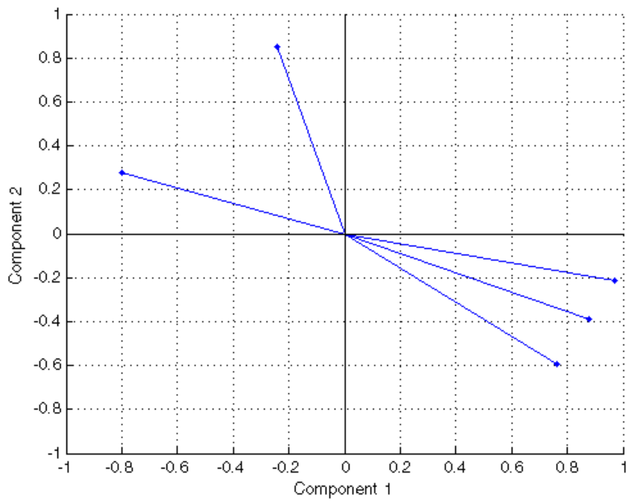
`factoran` computes factor scores that have been rotated by `inv(T')`, i.e., `F = F0 * inv(T')`, where `F0` contains the unrotated predictions. The estimated covariance of `F` is `inv(T'*T)`, which, for orthogonal or no rotation, is the identity matrix. Rotation of factor loadings and scores is an attempt to create a more easily interpretable structure in the loadings matrix after maximum likelihood estimation.

Examples

Example 1

Load the `carbig` data, and fit the default model with two factors.

```
load carbig
X = [Acceleration Displacement Horsepower MPG Weight];
X = X(all(~isnan(X),2),:);
[Lambda,Psi,T,stats,F] = factoran(X,2,...
                                'scores','regression')
inv(T'*T) % Estimated correlation matrix of F, == eye(2)
Lambda*Lambda'+diag(Psi) % Estimated correlation matrix
Lambda*inv(T)            % Unrotate the loadings
F*T'                     % Unrotate the factor scores
biplot(Lambda)           % Create biplot of two factors
```



Example 2

Although the estimates are the same, the use of a covariance matrix rather than raw data doesn't let you request scores or significance level.

```
[Lambda,Psi,T] = factoran(cov(X),2,'xtype','cov')
[Lambda,Psi,T] = factoran(corrcoef(X),2,'xtype','cov')
```

Example 3

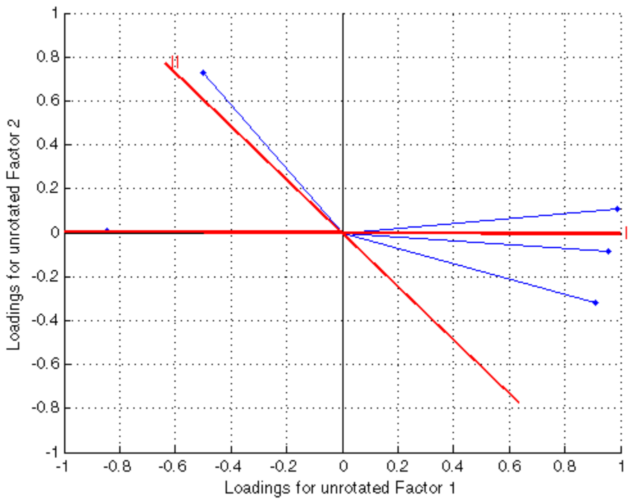
Use promax rotation.

```
[Lambda,Psi,T,stats,F] = factoran(X,2,'rotate','promax',...
                                   'powerpm',4)
inv(T'*T)                         % Est'd corr of F,
                                  % no longer eye(2)
Lambda*inv(T'*T)*Lambda'+diag(Psi)   % Est'd corr of X
```

Plot the unrotated variables with oblique axes superimposed.

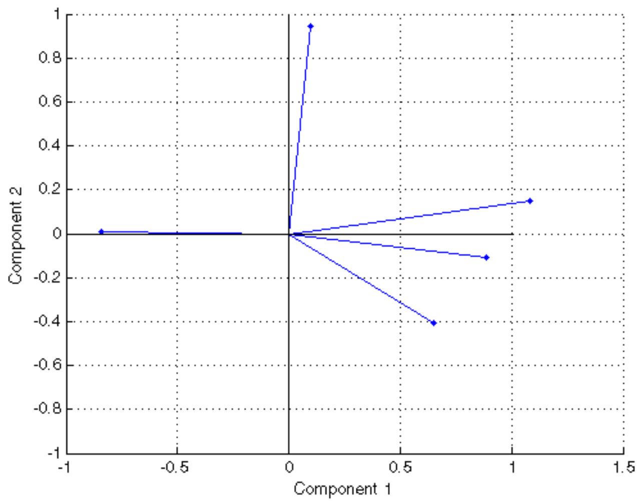
```
invT = inv(T)
```

```
Lambda0 = Lambda*invT
biplot(Lambda0);
line([-invT(1,1) invT(1,1) NaN -invT(2,1) invT(2,1)], ...
     [-invT(1,2) invT(1,2) NaN -invT(2,2) invT(2,2)], ...
     'color','r','linewidth',2);
text(invT(:,1), invT(:,2),['I';'II'],'color','r');
xlabel('Loadings for unrotated Factor 1')
ylabel('Loadings for unrotated Factor 2')
```



Plot the rotated variables against the oblique axes.

```
biplot(Lambda)
```



Example 4

Syntax for passing additional arguments to a user-defined rotation function.

```
[Lambda,Psi,T] = ...
     factoran(X,2,'rotate',@myrotation,'userargs',1,'two')
```

References

[1] Harman, H. H., *Modern Factor Analysis*, 3rd Ed., University of Chicago Press, Chicago, 1976.

[2] Jöreskog, K. G., "Some Contributions to Maximum Likelihood Factor Analysis," *Psychometrika*, Vol. 32, 1967, pp. 443-482.

[3] Lawley, D. N. and A. E. Maxwell, *Factor Analysis as a Statistical Method*, 2nd Edition, American Elsevier Pub. Co., New York, 1971.

See Also

`biplot`, `princomp`, `procrustes`, `pcacov`, `rotatefactors`, `statset`

Purpose

F cumulative distribution function

Syntax

```
P = fcdf(X,V1,V2)
```

Description

`P = fcdf(X,V1,V2)` computes the F cdf at each of the values in `X` using the corresponding parameters in `V1` and `V2`. `X`, `V1`, and `V2` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant matrix with the same dimensions as the other inputs. The parameters in `V1` and `V2` must be positive integers.

The F cdf is

$$p = F(x|\nu_1,\nu_2) = \int_0^x \frac{\Gamma\left[\frac{(\nu_1+\nu_2)}{2}\right]}{\Gamma\left(\frac{\nu_1}{2}\right)\Gamma\left(\frac{\nu_2}{2}\right)}\left(\frac{\nu_1}{\nu_2}\right)^{\frac{\nu_1}{2}} \frac{t^{\frac{\nu_1-2}{2}}}{\left[1+\left(\frac{\nu_1}{\nu_2}\right)t\right]^{\frac{\nu_1+\nu_2}{2}}}dt$$

The result, p, is the probability that a single observation from an F distribution with parameters ν_1 and ν_2 will fall in the interval $[0\ x]$.

Examples

This example illustrates an important and useful mathematical identity for the F distribution.

```
nu1 = 1:5;
nu2 = 6:10;
x = 2:6;

F1 = fcdf(x,nu1,nu2)
F1 =
  0.7930  0.8854  0.9481  0.9788  0.9919

F2 = 1 - fcdf(1./x,nu2,nu1)
F2 =
  0.7930  0.8854  0.9481  0.9788  0.9919
```

See Also

`cdf`, `finv`, `fpdf`, `frnd`, `fstat`

Purpose

Two-level full-factorial designs

Syntax

```
X = ff2n(n)
```

Description

`X = ff2n(n)` creates a two-level full-factorial design, `X`, where `n` is the desired number of columns of `X`. The number of rows in `X` is 2^n.

Example

```
X = ff2n(3)

X =
     0     0     0
     0     0     1
     0     1     0
     0     1     1
     1     0     0
     1     0     1
     1     1     0
     1     1     1
```

`X` is the binary representation of the numbers from 0 to 2^n-1.

See Also

`fracfact`, `fullfact`

Purpose

Inverse of *F* cumulative distribution function

Syntax

```
X = finv(P,V1,V2)
```

Description

`X = finv(P,V1,V2)` computes the inverse of the *F* cdf with numerator degrees of freedom `V1` and denominator degrees of freedom `V2` for the corresponding probabilities in `P`. `P`, `V1`, and `V2` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs.

The parameters in `V1` and `V2` must all be positive integers, and the values in `P` must lie on the interval [0 1].

The *F* inverse function is defined in terms of the *F* cdf as

$$x = F^{-1}(p|\nu_1,\nu_2) = \{x : F(x|\nu_1,\nu_2) = p\}$$

where

$$p = F(x|\nu_1,\nu_2) = \int_0^x \frac{\Gamma\left[\frac{(\nu_1+\nu_2)}{2}\right]}{\Gamma\left(\frac{\nu_1}{2}\right)\Gamma\left(\frac{\nu_2}{2}\right)}\left(\frac{\nu_1}{\nu_2}\right)^{\frac{\nu_1}{2}} \frac{t^{\frac{\nu_1-2}{2}}}{\left[1+\left(\frac{\nu_1}{\nu_2}\right)t\right]^{\frac{\nu_1+\nu_2}{2}}}dt$$

Examples

Find a value that should exceed 95% of the samples from an *F* distribution with 5 degrees of freedom in the numerator and 10 degrees of freedom in the denominator.

```
x = finv(0.95,5,10)
x =
  3.3258
```

You would observe values greater than 3.3258 only 5% of the time by chance.

See Also

`fcdf`, `fpdf`, `frnd`, `fstat`, `icdf`

Purpose

F probability density function

Syntax

```
Y = fpdf(X,V1,V2)
```

Description

`Y = fpdf(X,V1,V2)` computes the *F* pdf at each of the values in `X` using the corresponding parameters in `V1` and `V2`. `X`, `V1`, and `V2` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs. The parameters in `V1` and `V2` must all be positive integers, and the values in `X` must lie on the interval [0 ∞).

The probability density function for the *F* distribution is

$$y = f(x|\nu_1,\nu_2) = \frac{\Gamma\left[\frac{(\nu_1+\nu_2)}{2}\right]}{\Gamma\left(\frac{\nu_1}{2}\right)\Gamma\left(\frac{\nu_2}{2}\right)}\left(\frac{\nu_1}{\nu_2}\right)^{\frac{\nu_1}{2}}\frac{x^{\frac{\nu_1-2}{2}}}{\left[1+\left(\frac{\nu_1}{\nu_2}\right)x\right]^{\frac{\nu_1+\nu_2}{2}}}$$

Examples

```
y = fpdf(1:6,2,2)
y =
  0.2500   0.1111   0.0625   0.0400   0.0278   0.0204

z = fpdf(3,5:10,5:10)
z =
  0.0689   0.0659   0.0620   0.0577   0.0532   0.0487
```

See Also

`fcdf`, `finv`, `frnd`, `fstat`, `pdf`

Purpose

Generate fractional factorial design from generators

Syntax

```
x = fracfact(gen)
[x,conf] = fracfact(gen)
```

Description

`x = fracfact(gen)` generates a fractional factorial design as specified by the generator string `gen`, and returns a matrix `x` of design points. The input string `gen` is a generator string consisting of “words” separated by spaces. Each word describes how a column of the output design should be formed from columns of a full factorial. Typically `gen` will include single-letter words for the first few factors, plus additional multiple-letter words describing how the remaining factors are confounded with the first few.

The output matrix `x` is a fraction of a two-level full-factorial design. Suppose there are *m* words in `gen`, and that each word is formed from a subset of the first *n* letters of the alphabet. The output matrix `x` has 2^n rows and *m* columns. Let `F` represent the two-level full-factorial design as produced by `ff2n(n)`. The values in column *j* of `x` are computed by multiplying together the columns of `F` corresponding to letters that appear in the *j*th word of the generator string.

`[x,conf] = fracfact(gen)` also returns a cell array, `conf`, that describes the confounding pattern among the main effects and all two-factor interactions.

Examples

Example 1

You want to run an experiment to study the effects of four factors on a response, but you can only afford eight runs. (A run is a single repetition of the experiment at a specified combination of factor values.) Your goal is to determine which factors affect the response. There may be interactions between some pairs of factors.

A total of sixteen runs would be required to test all factor combinations. However, if you are willing to assume there are no three-factor interactions, you can estimate the main factor effects in just eight runs.

```
[x,conf] = fracfact('a b c abc')
```

```
x =
 -1  -1  -1  -1
 -1  -1   1   1
 -1   1  -1   1
 -1   1   1  -1
  1  -1  -1   1
  1  -1   1  -1
  1   1  -1  -1
  1   1   1   1
conf =
 'Term'   'Generator' 'Confounding'
 'X1'     'a'         'X1'
 'X2'     'b'         'X2'
 'X3'     'c'         'X3'
 'X4'     'abc'       'X4'
 'X1*X2'  'ab'        'X1*X2+X3*X4'
 'X1*X3'  'ac'        'X1*X3+X2*X4'
 'X1*X4'  'bc'        'X1*X4+X2*X3'
 'X2*X3'  'bc'        'X1*X4+X2*X3'
 'X2*X4'  'ac'        'X1*X3+X2*X4'
 'X3*X4'  'ab'        'X1*X2+X3*X4'
```

The first three columns of the x matrix form a full-factorial design. The final column is formed by multiplying the other three. The confounding pattern shows that the main effects for all four factors are estimable, but the two-factor interactions are not. For example, the X1*X2 and X3*X4 interactions are confounded, so it is not possible to estimate their effects separately.

After conducting the experiment, you may find out that the 'ab' effect is significant. In order to determine whether this effect comes from X1*X2 or X3*X4 you would have to run the remaining eight runs. You can obtain those runs by reversing the sign of the final generator.

```
fracfact('a b c -abc')
ans =
 -1  -1  -1   1
 -1  -1   1  -1
```

```
    -1     1    -1    -1
    -1     1     1     1
     1    -1    -1    -1
     1    -1     1     1
     1     1    -1     1
     1     1     1    -1
```

Example 2

Suppose now you need to study the effects of eight factors. A full factorial would require 256 runs. By clever choice of generators, you can find a sixteen-run design that can estimate those eight effects with no confounding from two-factor interactions.

```
[x,c] = fracfact('a b c d abc acd abd bcd');
c(1:10,:)
ans =
 'Term'   'Generator'  'Confounding'
 'X1'     'a'          'X1'
 'X2'     'b'          'X2'
 'X3'     'c'          'X3'
 'X4'     'd'          'X4'
 'X5'     'abc'        'X5'
 'X6'     'acd'        'X6'
 'X7'     'abd'        'X7'
 'X8'     'bcd'        'X8'
 'X1*X2'  'ab'         'X1*X2+X3*X5+X4*X7+X6*X8'
```

This confounding pattern shows that the main effects are not confounded with two-factor interactions. The final row shown reveals that a group of four two-factor interactions is confounded. Other choices of generators would not have the same desirable property.

```
[x,c] = fracfact('a b c d ab cd ad bc');
c(1:10,:)
ans =
 'Term'   'Generator'  'Confounding'
 'X1'     'a'          'X1+X2*X5+X4*X7'
 'X2'     'b'          'X2+X1*X5+X3*X8'
```

```
'X3'    'c'          'X3+X2*X8+X4*X6'
'X4'    'd'          'X4+X1*X7+X3*X6'
'X5'    'ab'         'X5+X1*X2'
'X6'    'cd'         'X6+X3*X4'
'X7'    'ad'         'X7+X1*X4'
'X8'    'bc'         'X8+X2*X3'
'X1*X2' 'ab'         'X5+X1*X2'
```

Here all the main effects are confounded with one or more two-factor interactions.

References

[1] Box, G. A. F., W. G. Hunter, and J. S. Hunter, *Statistics for Experimenters*, Wiley, 1978.

See Also

`ff2n`, `fullfact`, `hadamard`

fracfactgen

Purpose

Fractional factorial design generators

Syntax

```
gens = fracfactgen(model,K)
gens = fracfactgen(model,K,res)
gens = fracfactgen(model,K,res,basic)
```

Description

`gens = fracfactgen(`*`model`*`,K)` finds a set of fractional factorial design generators suitable for fitting a specified model. *`model`* specifies the model, and is either a text string or a matrix of `0`s and `1`s as accepted by the `x2fx` function. The design has `2^K` runs. The output `gens` is a cell array that specifies the confounding of the design, and that is suitable for use as input to the `fracfact` function. The `fracfact` function can generate the design and display the confounding pattern for the generators. If `K` is not given, `fracfactgen` tries to find the smallest possible value.

If *`model`* is a text string, *`model`* must consist of a sequence of words separated by spaces, each word representing a term that must be estimable in the design. The j^{th} letter of the alphabet represents the j^{th} factor. For example, `'a b c d ac'` defines a model that includes the main effects for factors `a` through `d`, and the interaction between factors `a` and `c`.

`fracfactgen` uses the Franklin-Bailey algorithm to find the generators of a design that is capable of fitting the specified model.

`gens = fracfactgen(`*`model`*`,K,res)` tries to find a design with resolution `res` (the default value is `3`). If `fracfactgen` is unable to find the requested resolution, it either displays an error, or if it locates a lower-resolution design capable of fitting the model, it returns the generators for that design along with a warning. If the result is an error, it may still be possible to call `fracfactgen` with a lower value of `res` and find a set of design generators.

`gens = fracfactgen(`*`model`*`,K,res,basic)` also accepts a vector `basic` with `K` elements specifying the numbers of the factors that are to be treated as basic factors. These factors receive single-letter generators, and other factors are confounded with interactions among the basic factors. The default is chosen to include factors that are part of the highest-order interaction in *`model`*.

Examples

Find the generators for a design with four factors and 2^3=8 runs so that you can estimate the interaction between the first and third factors.

```
fracfactgen('a b c d ac',3)
ans =
    'a'
    'b'
    'c'
    'abc'

m = [1 0 0 0;
     0 1 0 0;
     0 0 1 0;
     0 0 0 1;
     1 0 1 0];
fracfactgen(m,3)
ans =
    'a'
    'b'
    'c'
    'abc'
```

See Also

`fracfact`

friedman

Purpose

Friedman's nonparametric two-way analysis of variance

Syntax

```
p = friedman(X,reps)
p = friedman(X,reps,displayopt)
[p,table] = friedman(...)
[p,table,stats] = friedman(...)
```

Description

`p = friedman(X,reps)` performs the nonparametric Friedman's test to compare column effects in a two-way layout. Friedman's test is similar to classical balanced two-way ANOVA, but it tests only for column effects after adjusting for possible row effects. It does not test for row effects or interaction effects. Friedman's test is appropriate when columns represent treatments that are under study, and rows represent nuisance effects (blocks) that need to be taken into account but are not of any interest.

The different columns of `X` represent changes in a factor A. The different rows represent changes in a blocking factor B. If there is more than one observation for each combination of factors, input `reps` indicates the number of replicates in each "cell," which must be constant.

The matrix below illustrates the format for a set-up where column factor A has three levels, row factor B has two levels, and there are two replicates (`reps=2`). The subscripts indicate row, column, and replicate, respectively.

$$\begin{bmatrix} x_{111} & x_{121} & x_{131} \\ x_{112} & x_{122} & x_{132} \\ x_{211} & x_{221} & x_{231} \\ x_{212} & x_{222} & x_{232} \end{bmatrix}$$

Friedman's test assumes a model of the form

$$x_{ijk} = \mu + \alpha_i + \beta_j + \varepsilon_{ijk}$$

where μ is an overall location parameter, α_i represents the column effect, β_j represents the row effect, and ε_{ijk} represents the error. This test ranks the data within each level of B, and tests for a difference across levels of A. The `p` that `friedman` returns is the p-value for the null hypothesis that $\alpha_i = 0$. If the p-value is near zero, this casts doubt on the null hypothesis. A sufficiently small p-value suggests that at least one column-sample median is significantly different than the others; i.e., there is a main effect due to factor A. The choice of a critical p-value to determine whether a result is "statistically significant" is left to the researcher. It is common to declare a result significant if the p-value is less than 0.05 or 0.01.

`friedman` also displays a figure showing an ANOVA table, which divides the variability of the ranks into two or three parts:

- The variability due to the differences among the column effects
- The variability due to the interaction between rows and columns (if reps is greater than its default value of 1)
- The remaining variability not explained by any systematic source

The ANOVA table has six columns:

- The first shows the source of the variability.
- The second shows the Sum of Squares (SS) due to each source.
- The third shows the degrees of freedom (df) associated with each source.
- The fourth shows the Mean Squares (MS), which is the ratio SS/df.
- The fifth shows Friedman's chi-square statistic.
- The sixth shows the p-value for the chi-square statistic.

`p = friedman(X,reps,`*`displayopt`*`)` enables the ANOVA table display when *`displayopt`* is `'on'` (default) and suppresses the display when *`displayopt`* is `'off'`.

`[p,table] = friedman(...)` returns the ANOVA table (including column and row labels) in cell array `table`. (You can copy a text version of the ANOVA table to the clipboard by selecting `Copy Text` from the **Edit** menu.

`[p,table,stats] = friedman(...)` returns a `stats` structure that you can use to perform a follow-up multiple comparison test. The `friedman` test evaluates the hypothesis that the column effects are all the same against the alternative that they are not all the same. Sometimes it is preferable to perform a test to determine which pairs of column effects are significantly different, and which are not. You can use the `multcompare` function to perform such tests by supplying the `stats` structure as input.

Assumptions

Friedman's test makes the following assumptions about the data in `X`:

- All data come from populations having the same continuous distribution, apart from possibly different locations due to column and row effects.
- All observations are mutually independent.

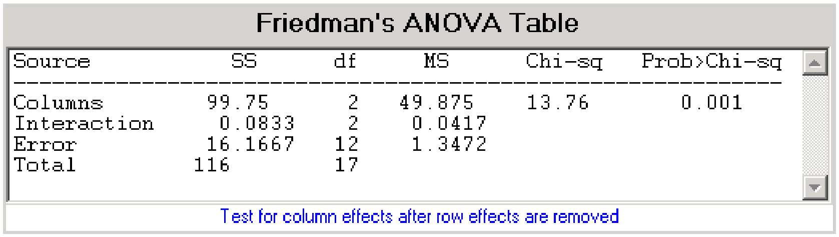
The classical two-way ANOVA replaces the first assumption with the stronger assumption that data come from normal distributions.

Examples

Let's repeat the example from the `anova2` function, this time applying Friedman's test. Recall that the data below come from a study of popcorn brands and popper type (Hogg 1987). The columns of the matrix `popcorn` are brands (Gourmet, National, and Generic). The rows are popper type (Oil and Air). The study popped a batch of each brand three times with each popper. The values are the yield in cups of popped popcorn.

```
load popcorn
popcorn
popcorn =
  5.5000  4.5000  3.5000
```

```
    5.5000    4.5000    4.0000
    6.0000    4.0000    3.0000
    6.5000    5.0000    4.0000
    7.0000    5.5000    5.0000
    7.0000    5.0000    4.5000

p = friedman(popcorn,3)
p =
    0.0010
```



```
Source              SS      df      MS      Chi-sq   Prob>Chi-sq
-----------------------------------------------------------------
Columns           99.75      2    49.875    13.76       0.001
Interaction        0.0833    2     0.0417
Error             16.1667   12     1.3472
Total            116        17
```

The small p-value of `0.001` indicates the popcorn brand affects the yield of popcorn. This is consistent with the results from `anova2`.

You could also test popper type by permuting the `popcorn` array as described in "Friedman's Test" on page 7-30 and repeating the test.

References

[1] Hogg, R. V. and J. Ledolter, *Engineering Statistics,* MacMillan, 1987.

[2] Hollander, M., and D. A. Wolfe, *Nonparametric Statistical Methods*, Wiley, 1973.

See Also

`anova2`, `multcompare`, `kruskalwallis`

Purpose

Random numbers from *F* distribution

Syntax

```
R = frnd(V1,V2)
R = frnd(V1,V2,v)
R = frnd(V1,V2,m,n)
```

Description

`R = frnd(V1,V2)` generates random numbers from the *F* distribution with numerator degrees of freedom `V1` and denominator degrees of freedom `V2`. `V1` and `V2` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input for `V1` or `V2` is expanded to a constant array with the same dimensions as the other input.

`R = frnd(V1,V2,v)` generates random numbers from the *F* distribution with parameters `V1` and `V2`, where `v` is a row vector. If `v` is a 1-by-2 vector, `R` is a matrix with `v(1)` rows and `v(2)` columns. If `v` is 1-by-n, `R` is an n-dimensional array.

`R = frnd(V1,V2,m,n)` generates random numbers from the *F* distribution with parameters `V1` and `V2`, where scalars m and n are the row and column dimensions of `R`.

Example

```
  n1 = frnd(1:6,1:6)
  n1 =
    0.0022  0.3121  3.0528  0.3189  0.2715  0.9539

  n2 = frnd(2,2,[2 3])
  n2 =
    0.3186  0.9727  3.0268
    0.2052 148.5816  0.2191

  n3 = frnd([1 2 3;4 5 6],1,2,3)
  n3 =
    0.6233  0.2322  31.5458
    2.5848  0.2121  4.4955
```

See Also

`fcdf`, `finv`, `fpdf`, `fstat`

Purpose

Mean and variance of *F* distribution

Syntax

```
[M,V] = fstat(V1,V2)
```

Description

`[M,V] = fstat(V1,V2)` returns the mean of and variance for the *F* distribution with parameters specified by `V1` and `V2`. `V1` and `V2` can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of `M` and `V`. A scalar input for `V1` or `V2` is expanded to a constant arrays with the same dimensions as the other input.

The mean of the *F* distribution for values of ν_2 greater than 2 is

$$\frac{\nu_2}{\nu_2 - 2}$$

The variance of the *F* distribution for values of ν_2 greater than 4 is

$$\frac{2\nu_2^{\,2}(\nu_1 + \nu_2 - 2)}{\nu_1(\nu_2 - 2)^2(\nu_2 - 4)}$$

The mean of the *F* distribution is undefined if ν_2 is less than 3. The variance is undefined for ν_2 less than 5.

Examples

`fstat` returns `NaN` when the mean and variance are undefined.

```
[m,v] = fstat(1:5,1:5)
m =
    NaN  NaN  3.0000  2.0000  1.6667
v =
    NaN  NaN  NaN  NaN  8.8889
```

See Also

`fcdf`, `finv`, `frnd`, `frnd`

Purpose

Interactive contour plot

Syntax

```
fsurfht(fun,xlims,ylims)
fsurfht(fun,xlims,ylims,p1,p2,p3,p4,p5)
```

Description

`fsurfht(fun,xlims,ylims)` is an interactive contour plot of the function specified by the text variable `fun`. The *x*-axis limits are specified by `xlims` in the form `[xmin xmax]`, and the *y*-axis limits are specified by `ylims` in the form `[ymin ymax]`.

`fsurfht(fun,xlims,ylims,p1,p2,p3,p4,p5)` allows for five optional parameters that you can supply to the function `fun`.

The intersection of the vertical and horizontal reference lines on the plot defines the current *x*-value and *y*-value. You can drag these reference lines and watch the calculated *z*-values (at the top of the plot) update simultaneously. Alternatively, you can type the *x*-value and *y*-value into editable text fields on the *x*-axis and *y*-axis.

Example

Plot the Gaussian likelihood function for the `gas.mat` data.

```
load gas
```

Create a function containing the following commands, and name it `gauslike.m`.

```
function z = gauslike(mu,sigma,p1)
n = length(p1);
z = ones(size(mu));
for i = 1:n
z = z .* (normpdf(p1(i),mu,sigma));
end
```

The `gauslike` function calls `normpdf`, treating the data sample as fixed and the parameters μ and σ as variables. Assume that the gas prices are normally distributed, and plot the likelihood surface of the sample.

```
fsurfht('gauslike',[112 118],[3 5],price1)
```

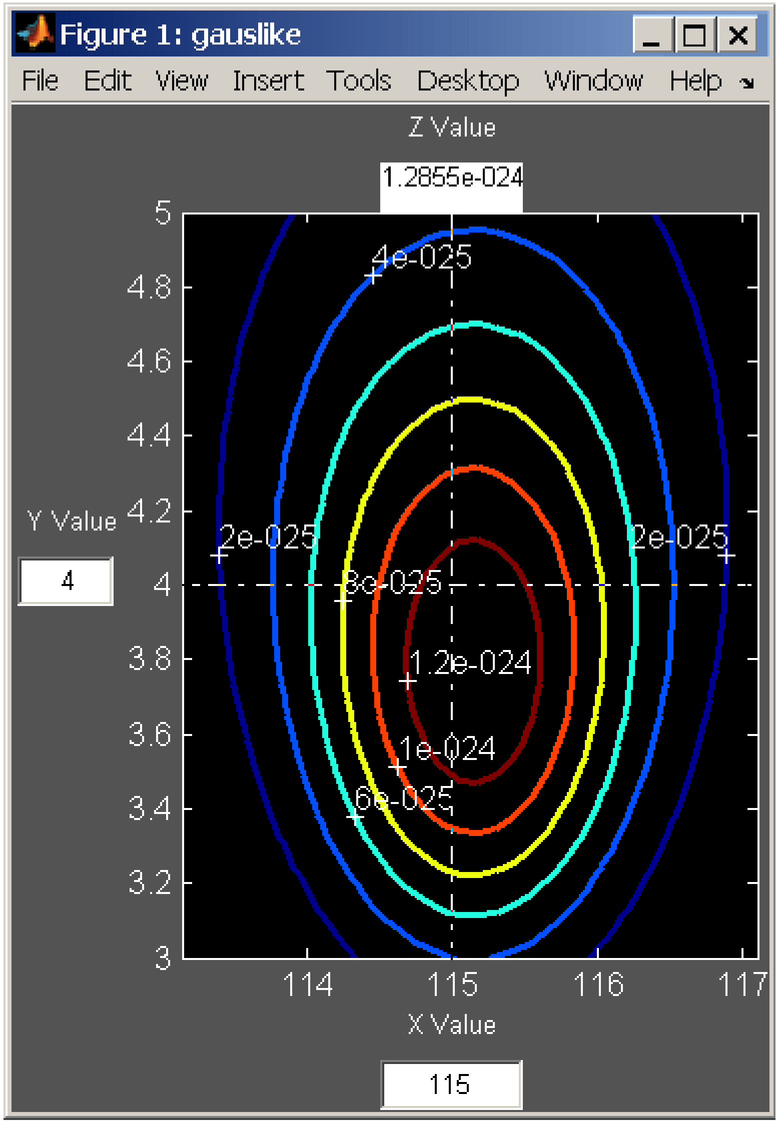


Figure 1: gauslike
File
Edit
View
Insert
Tools
Desktop
Window
Help
Z Value
1.2855e-024
Y Value
4
X Value
115
5
4.8
4.6
4.4
4.2
4
3.8
3.6
3.4
3.2
3
114
115
116
117
4e-025
2e-025
2e-025
3e-025
1.2e-024
1e-024
6e-025

The sample mean is the *x*-value at the maximum, but the sample standard deviation is *not* the *y*-value at the maximum.

```
mumax = mean(price1)
mumax =
 115.1500
sigmamax = std(price1)*sqrt(19/20)
sigmamax =
  3.7719
```

Purpose

Full-factorial experimental design

Syntax

```
design = fullfact(levels)
```

Description

`design = fullfact(levels)` give the factor settings for a full factorial design. Each element in the vector `levels` specifies the number of unique values in the corresponding column of `design`.

For example, if the first element of `levels` is `3`, then the first column of `design` contains only integers from `1` to `3`.

Example

If `levels = [2 4]`, `fullfact` generates an eight-run design with two levels in the first column and four in the second column.

```
d = fullfact([2 4])
d =
    1   1
    2   1
    1   2
    2   2
    1   3
    2   3
    1   4
    2   4
```

See Also

`ff2n`, `dcovary`, `daugment`, `cordexch`

Purpose

Gage repeatability and reproducibility study

Syntax

```
gagerr(y,{part,operator})
gagerr(y,GROUP)
gagerr(y,part)
gagerr(...,param1,val1,param2,val2,...)
[TABLE, stats] = gagerr(...)
```

Description

`gagerr(y,{part,operator})` performs a gage repeatability and reproducibility study on measurements in `y` collected by `operator` on `part`. `y` is a column vector containing the measurements on different parts. `part` and `operator` are categorical variables, numeric vectors, character matrices, or cell arrays of strings. The number of elements in `part` and `operator` should be the same as in `y`.

`gagerr` prints a table in the command window in which the decomposition of variance, standard deviation, study var (5.15 x standard deviation) are listed with respective percentages for different sources. Summary statistics are printed below the table giving the number of distinct categories (NDC) and the percentage of Gage R&R of total variations (PRR).

`gagerr` also plots a bar graph showing the percentage of different components of variations. Gage R&R, repeatability, reproducibility, and part-to-part variations are plotted as four vertical bars. Variance and study var are plotted as two groups.

To determine the capability of a measurement system using NDC, use the following guidelines:

- If NDC > 5, the measurement system is capable.
- If NDC < 2, the measurement system is not capable.
- Otherwise, the measurement system may be acceptable.

To determine the capability of a measurement system using PRR, use the following guidelines:

- If PRR < 10%, the measurement system is capable.
- If PRR > 30%, the measurement system is not capable.
- Otherwise, the measurement system may be acceptable.

`gagerr(y,GROUP)` performs a gage R&R study on measurements in `y` with `part` and `operator` represented in `GROUP`. `GROUP` is a numeric matrix whose first and second columns specify different parts and operators, respectively. The number of rows in `GROUP` should be the same as the number of elements in `y`. (See "Grouped Data" on page 2-41.)

`gagerr(y,part)` performs a gage R&R study on measurements in `y` without operator information. The assumption is that all variability is contributed by `part`.

`gagerr(...,`*`param1`*`,`*`val1`*`,`*`param2`*`,`*`val2`*`,...)` performs a gage R&R study using one or more of the following parameter name/value pairs:

- `'spec'` — A two-element vector that defines the lower and upper limit of the process, respectively. In this case, summary statistics printed in the command window include Precision-to-Tolerance Ratio (PTR). Also, the bar graph includes an additional group, the percentage of tolerance.

  To determine the capability of a measurement system using PTR, use the following guidelines:

  - If PTR < 0.1, the measurement system is capable.
  - If PTR > 0.3, the measurement system is not capable.
  - Otherwise, the measurement system may be acceptable.
- `'printtable'` — A string with a value `'on'` or `'off'` that indicates whether the tabular output should be printed in the command window or not. The default value is `'on'`.
- `'printgraph'` — A string with a value `'on'` or `'off'` that indicates whether the bar graph should be plotted or not. The default value is `'on'`.

- `'randomoperator'` — A logical value, `true` or `false`, that indicates whether the effect of `operator` is random or not. The default value is `true`.
- `'model'` — The model to use, specified by one of:
  - `'linear'` — Main effects only (default)
  - `'interaction'` — Main effects plus two-factor interactions
  - `'nested'` — Nest `operator` in `part`

  The default value is `'linear'`.

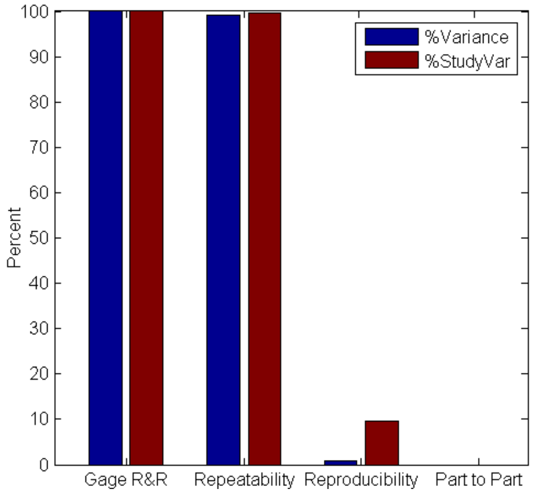
`[TABLE, stats] = gagerr(...)` returns a 6-by-5 matrix `TABLE` and a structure `stats`. The columns of `TABLE`, from left to right, represent variance, percentage of variance, standard deviations, study var, and percentage of study var. The rows of `TABLE`, from top to bottom, represent different sources of variations: gage R&R, repeatability, reproducibility, operator, operator and part interactions, and part. `stats` is a structure containing summary statistics for the performance of the measurement system. The fields of `stats` are:

- `ndc` — Number of distinct categories
- `prr` — Percentage of gage R&R of total variations
- `ptr` — Precision-to-tolerance ratio. The value is `NaN` if the parameter `'spec'` is not given.

Example

Conduct a gage R&R study for a simulated measurement system using a mixed ANOVA model without interactions:

```
y = randn(100,1);                               % measurements
part = ceil(3*rand(100,1));                     % parts
operator = ceil(4*rand(100,1));                 % operators
gagerr(y,{part, operator},'randomoperator',true) % analysis

Source           Variance     % Variance      sigma      5.15*sigma    % 5.15*sigma
===================================================================================
Gage R&R            0.77       100.00         0.88         4.51       100.00
  Repeatability     0.76        99.08         0.87         4.49        99.54
  Reproducibility   0.01         0.92         0.08         0.43         9.61
  Operator          0.01         0.92         0.08         0.43         9.61
Part                0.00         0.00         0.00         0.00         0.00
Total               0.77       100.00         0.88         4.51
-----------------------------------------------------------------------------------

               Number of distinct categories (NDC):  0
               % of Gage R&R of total variations (PRR): 100.00

Note: The last column of the above table does not have to sum to 100%
```



gamcdf

Purpose

Gamma cumulative distribution function

Syntax

```
gamcdf(X,A,B)
[P,PLO,PUP] = gamcdf(X,A,B,pcov,alpha)
```

Description

`gamcdf(X,A,B)` computes the gamma cdf at each of the values in `X` using the corresponding parameters in `A` and `B`. `X`, `A`, and `B` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs. The parameters in `A` and `B` must be positive.

The gamma cdf is

$$p = F(x|a,b) = \frac{1}{b^a \Gamma(a)} \int_0^x t^{a-1} e^{-\frac{t}{b}} dt$$

The result, p, is the probability that a single observation from a gamma distribution with parameters a and b will fall in the interval [0 x].

`[P,PLO,PUP] = gamcdf(X,A,B,pcov,alpha)` produces confidence bounds for `P` when the input parameters `A` and `B` are estimates. `pcov` is a 2-by-2 matrix containing the covariance matrix of the estimated parameters. `alpha` has a default value of 0.05, and specifies `100(1-alpha)%` confidence bounds. `PLO` and `PUP` are arrays of the same size as `P` containing the lower and upper confidence bounds.

`gammainc` is the gamma distribution with b fixed at 1.

Examples

```
a = 1:6;
b = 5:10;
prob = gamcdf(a.*b,a,b)
prob =
  0.6321  0.5940  0.5768  0.5665  0.5595  0.5543
```

The mean of the gamma distribution is the product of the parameters, ab. In this example, the mean approaches the median as it increases (i.e., the distribution becomes more symmetric).

See Also

`cdf`, `gamfit`, `gaminv`, `gamlike`, `gampdf`, `gamrnd`, `gamstat`, `gammainc`

Purpose

Parameter estimates and confidence intervals for gamma distributed data

Syntax

```
phat = gamfit(data)
[phat,pci] = gamfit(data)
[phat,pci] = gamfit(data,alpha)
[...] = gamfit(data,alpha,censoring,freq,options)
```

Description

`phat = gamfit(data)` returns the maximum likelihood estimates (MLEs) for the parameters of the gamma distribution given the data in vector `data`.

`[phat,pci] = gamfit(data)` returns MLEs and 95% percent confidence intervals. The first row of `pci` is the lower bound of the confidence intervals; the last row is the upper bound.

`[phat,pci] = gamfit(data,alpha)` returns `100(1 - alpha)%` confidence intervals. For example, `alpha = 0.01` yields 99% confidence intervals.

`[...]  = gamfit(data,alpha,censoring)` accepts a boolean vector of the same size as `data` that is 1 for observations that are right-censored and 0 for observations that are observed exactly.

`[...]  = gamfit(data,alpha,censoring,freq)` accepts a frequency vector of the same size as `data`. `freq` typically contains integer frequencies for the corresponding elements in `data`, but may contain any nonnegative values.

`[...]  = gamfit(data,alpha,censoring,freq,options)` accepts a structure, `options`, that specifies control parameters for the iterative algorithm the function uses to compute maximum likelihood estimates. The gamma fit function accepts an `options` structure which can be created using the function `statset`. Enter `statset('gamfit')` to see the names and default values of the parameters that `gamfit` accepts in the `options` structure.

Example

Note that the 95% confidence intervals in the example below bracket the true parameter values of 2 and 4.

```
a = 2; b = 4;
data = gamrnd(a,b,100,1);
[p,ci] = gamfit(data)
p =
  2.1990  3.7426
ci =
  1.6840  2.8298
  2.7141  4.6554
```

Reference

[1] Hahn, G. J., and S. S. Shapiro, *Statistical Models in Engineering.* John Wiley & Sons, 1994. p. 88.

See Also

`gamcdf`, `gaminv`, `gamlike`, `gampdf`, `gamrnd`, `gamstat`, `mle`, `statset`

gaminv

Purpose

Inverse of gamma cumulative distribution function

Syntax

```
X = gaminv(P,A,B)
[X,XLO,XUP] = gamcdf(P,A,B,pcov,alpha)
```

Description

`X = gaminv(P,A,B)` computes the inverse of the gamma cdf with parameters `A` and `B` for the corresponding probabilities in `P`. `P`, `A`, and `B` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs. The parameters in `A` and `B` must all be positive, and the values in `P` must lie on the interval [0 1].

The gamma inverse function in terms of the gamma cdf is

$$x = F^{-1}(p|a,b) = \{x : F(x|a,b) = p\}$$

where

$$p = F(x|a,b) = \frac{1}{b^a\Gamma(a)}\int_0^x t^{a-1}e^{-\frac{t}{b}}dt$$

`[X,XLO,XUP] = gamcdf(P,A,B,pcov,alpha)` produces confidence bounds for `P` when the input parameters `A` and `B` are estimates. `pcov` is a 2-by-2 matrix containing the covariance matrix of the estimated parameters. `alpha` has a default value of 0.05, and specifies `100(1-alpha)`% confidence bounds. `PLO` and `PUP` are arrays of the same size as `P` containing the lower and upper confidence bounds.

Algorithm

There is no known analytical solution to the integral equation above. `gaminv` uses an iterative approach (Newton's method) to converge on the solution.

Examples

This example shows the relationship between the gamma cdf and its inverse function.

```
a = 1:5;
b = 6:10;
```

```
x = gaminv(gamcdf(1:5,a,b),a,b)
x =
  1.0000  2.0000  3.0000  4.0000  5.0000
```

See Also `gamcdf`, `gamfit`, `gamlike`, `gampdf`, `gamrnd`, `gamstat`, `icdf`

gamlike

Purpose Negative log-likelihood for gamma distribution

Syntax

```
nlogL = gamlike(params,data)
[nlogL,AVAR] = gamlike(params,data)
```

Description

`nlogL = gamlike(params,data)` returns the negative of the gamma log-likelihood function for the parameters, `params`, given data. The length of output vector `nlogL` is the length of vector data.

`[nlogL,AVAR] = gamlike(params,data)` also returns `AVAR`, which is the asymptotic variance-covariance matrix of the parameter estimates when the values in `params` are the maximum likelihood estimates. `AVAR` is the inverse of Fisher's information matrix. The diagonal elements of `AVAR` are the asymptotic variances of their respective parameters.

`[...] = gamlike(params,data,censoring)` accepts a boolean vector of the same size as `data` that is 1 for observations that are right-censored and 0 for observations that are observed exactly.

`[...] = gamfit(params,data,censoring,freq)` accepts a frequency vector of the same size as `data`. `freq` typically contains integer frequencies for the corresponding elements in `data`, but may contain any non-negative values.

`gamlike` is a utility function for maximum likelihood estimation of the gamma distribution. Since `gamlike` returns the negative gamma log-likelihood function, minimizing `gamlike` using `fminsearch` is the same as maximizing the likelihood.

Example

This example continues the example for `gamfit`.

```
a = 2; b = 3;
r = gamrnd(a,b,100,1);
[nlogL,info] = gamlike(gamfit(r),r)
nlogL =
  267.5648
info =
  0.0788  -0.1104
 -0.1104  0.1955
```

See Also betalike, gamcdf, gamfit, gaminv, gampdf, gamrnd, gamstat, mle, normlike, wbllike

gampdf

Purpose

Gamma probability density function

Syntax

```
Y = gampdf(X,A,B)
```

Description

`Y = gampdf(X,A,B)` computes the gamma pdf at each of the values in `X` using the corresponding parameters in `A` and `B`. `X`, `A`, and `B` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs. The parameters in `A` and `B` must all be positive, and the values in `X` must lie on the interval [0 ∞).

The gamma pdf is

$$y = f(x|a,b) = \frac{1}{b^a\Gamma(a)}x^{a-1}e^{-\frac{x}{b}}$$

The gamma probability density function is useful in reliability models of lifetimes. The gamma distribution is more flexible than the exponential distribution in that the probability of a product surviving an additional period may depend on its current age. The exponential and χ^2 functions are special cases of the gamma function.

Examples

The exponential distribution is a special case of the gamma distribution.

```
mu = 1:5;

y = gampdf(1,1,mu)
y =
  0.3679  0.3033  0.2388  0.1947  0.1637

y1 = exppdf(1,mu)
y1 =
  0.3679  0.3033  0.2388  0.1947  0.1637
```

See Also

`gamcdf`, `gamfit`, `gaminv`, `gamlike`, `gamrnd`, `gamstat`, `pdf`, `gamma`, `gammaln`

Purpose

Random numbers from gamma distribution

Syntax

```
R = gamrnd(A,B)
R = gamrnd(A,B,v)
R = gamrnd(A,B,m,n)
```

Description

`R = gamrnd(A,B)` generates random numbers from the gamma distribution with parameters `A` and `B`. `A` and `B` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input for `A` or `B` is expanded to a constant array with the same dimensions as the other input.

`R = gamrnd(A,B,v)` generates random numbers from the gamma distribution with parameters `A` and `B`, where `v` is a row vector. If v is a 1-by-2 vector, `R` is a matrix with `v(1)` rows and `v(2)` columns. If `v` is 1-by-n, `R` is an n-dimensional array.

`R = gamrnd(A,B,m,n)` generates gamma random numbers with parameters `A` and `B`, where scalars m and n are the row and column dimensions of `R`.

Example

```
  n1 = gamrnd(1:5,6:10)
  n1 =
    9.1132  12.8431  24.8025  38.5960 106.4164

  n2 = gamrnd(5,10,[1 5])
  n2 =
    30.9486  33.5667  33.6837  55.2014  46.8265

  n3 = gamrnd(2:6,3,1,5)
  n3 =
    12.8715  11.3068  3.0982  15.6012  21.6739
```

See Also

`gamcdf`, `gamfit`, `gaminv`, `gamlike`, `gampdf`, `gamstat`, `randg`

gamstat

Purpose

Mean and variance of gamma distribution

Syntax

```
[M,V] = gamstat(A,B)
```

Description

`[M,V] = gamstat(A,B)` returns the mean of and variance for the gamma distribution with parameters specified by `A` and `B`. `A` and `B` can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of `M` and `V`. A scalar input for `A` or `B` is expanded to a constant array with the same dimensions as the other input.

The mean of the gamma distribution with parameters a and b is ab. The variance is ab^2.

Examples

```
  [m,v] = gamstat(1:5,1:5)
  m =
     1    4    9   16   25
  v =
     1    8   27   64   125

  [m,v] = gamstat(1:5,1./(1:5))
  m =
     1    1    1    1    1
  v =
    1.0000   0.5000   0.3333   0.2500   0.2000
```

See Also

`gamcdf`, `gamfit`, `gaminv`, `gamlike`, `gampdf`, `gamrnd`

Purpose

Access dataset array properties

Syntax

```
get(A)
s = get(A)
p = get(A,PropertyName)
p = get(A,{PropertyName1,PropertyName2,...})
```

Description

`get(A)` displays a list of property/value pairs for the dataset array `A`.

`s = get(A)` returns the values in a scalar structure `s` with field names given by the properties.

`p = get(A,`*`PropertyName`*`)` returns the value of the property specified by the string *`PropertyName`*.

`p = get(A,{`*`PropertyName1`*`,`*`PropertyName2`*`,...})` allows multiple property names to be specified and returns their values in a cell array.

Example

Create a dataset array from Fisher's iris data and access the information:

```
load fisheriris
NumObs = size(meas,1);
ObsNames = strcat({'Obs'},num2str((1:NumObs)','%d'));
iris = dataset({nominal(species),'species'},...
               {meas,'SL','SW','PL','PW'},...
               'obsnames',ObsNames);

get(iris)
   Description: ''
   Units: {}
   DimNames: {'Observations' 'Variables'}
   UserData: []
   ObsNames: {150x1 cell}
   VarNames: {'species' 'SL' 'SW' 'PL' 'PW'}

ON = get(iris,'ObsNames');
ON(1:3)
```

```
ans =
    'Obs1'
    'Obs2'
    'Obs3'
```

See Also set, summary (dataset)

Purpose

Access labels of levels in categorical array

Syntax

`labels = getlabels(A)`

Description

`labels = getlabels(A)` returns the labels of the levels in the categorical array `A` as a cell array of strings `labels`. For ordinal `A`, the labels are returned in the order of the levels.

Examples

Example 1

Display levels in a nominal and an ordinal array:

```
standings = nominal({'Leafs','Canadiens','Bruins'});
getlabels(standings)
ans =
    'Bruins'  'Canadiens'  'Leafs'

standings = ordinal(1:3,{'Leafs','Canadiens','Bruins'});
getlabels(standings)
ans =
    'Leafs'  'Canadiens'  'Bruins'
```

Example 2

Display age groups containing data in `hospital.mat`:

```
load hospital
edges = 0:10:100;
labels = strcat(num2str((0:10:90)','%d'),{'s'});
AgeGroup = ordinal(hospital.Age,labels,[],edges);
AgeGroup = droplevels(AgeGroup);
getlabels(AgeGroup)
ans =
    '20s'    '30s'    '40s'    '50s'
```

See Also

`setlabels`

Purpose

Geometric cumulative distribution function

Syntax

```
Y = geocdf(X,P)
```

Description

`Y = geocdf(X,P)` computes the geometric cdf at each of the values in `X` using the corresponding probabilities in `P`. `X` and `P` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other input. The parameters in `P` must lie on the interval [0 1].

The geometric cdf is

$$y = F(x|p) = \sum_{i=0}^{floor(x)} pq^{i}$$

where $q = 1-p$.

The result, y, is the probability of observing up to x trials before a success, when the probability of success in any given trial is p.

Examples

Suppose you toss a fair coin repeatedly. If the coin lands face up (heads), that is a success. What is the probability of observing three or fewer tails before getting a heads?

```
p = geocdf(3,0.5)
p =
  0.9375
```

See Also

`cdf`, `geoinv`, `geopdf`, `geornd`, `geostat`

Purpose

Inverse of geometric cumulative distribution function

Syntax

```
X = geoinv(Y,P)
```

Description

`X = geoinv(Y,P)` returns the smallest positive integer `X` such that the geometric cdf evaluated at `X` is equal to or exceeds `Y`. You can think of `Y` as the probability of observing `X` successes in a row in independent trials where `P` is the probability of success in each trial.

`Y` and `P` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input for `P` or `Y` is expanded to a constant array with the same dimensions as the other input. The values in `P` and `Y` must lie on the interval [0 1].

Examples

The probability of correctly guessing the result of 10 coin tosses in a row is less than 0.001 (unless the coin is not fair).

```
psychic = geoinv(0.999,0.5)
psychic =
   9
```

The example below shows the inverse method for generating random numbers from the geometric distribution.

```
rndgeo = geoinv(rand(2,5),0.5)
rndgeo =
   0    1    3    1    0
   0    1    0    2    0
```

See Also

`geocdf`, `geopdf`, `geornd`, `geostat`, `icdf`

geomean

Purpose

Geometric mean of sample

Syntax

```
m = geomean(x)
geomean(X,dim)
```

Description

`m = geomean(x)` calculates the geometric mean of a sample. For vectors, `geomean(x)` is the geometric mean of the elements in `x`. For matrices, `geomean(X)` is a row vector containing the geometric means of each column. For N-dimensional arrays, `geomean` operates along the first nonsingleton dimension of `X`.

`geomean(X,dim)` takes the geometric mean along the dimension `dim` of `X`.

The geometric mean is

$$m = \left[\prod_{i=1}^{n} x_i\right]^{\frac{1}{n}}$$

Examples

The arithmetic mean is greater than or equal to the geometric mean.

```
x = exprnd(1,10,6);

geometric = geomean(x)
geometric =
  0.7466  0.6061  0.6038  0.2569  0.7539  0.3478

average = mean(x)
average =
  1.3509  1.1583  0.9741  0.5319  1.0088  0.8122
```

See Also

`mean`, `median`, `harmmean`, `trimmean`

Purpose

Geometric probability density function

Syntax

```
Y = geopdf(X,P)
```

Description

`Y = geopdf(X,P)` computes the geometric pdf at each of the values in `X` using the corresponding probabilities in `P`. `X` and `P` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other input. The parameters in `P` must lie on the interval [0 1].

The geometric pdf is

$$y = f(x|p) = pq^x I_{(0,1,K)}(x)$$

where $q = 1 - p$.

Examples

Suppose you toss a fair coin repeatedly. If the coin lands face up (heads), that is a success. What is the probability of observing exactly three tails before getting a heads?

```
p = geopdf(3,0.5)
p =
  0.0625
```

See Also

`geocdf`, `geoinv`, `geornd`, `geostat`, `pdf`

geornd

Purpose

Random numbers from geometric distribution

Syntax

```
R = geornd(P)
R = geornd(P,v)
R = geornd(P,m,n)
```

Description

The geometric distribution is useful when you want to model the number of successive failures preceding a success, where the probability of success in any given trial is the constant `P`.

`R = geornd(P)` generates geometric random numbers with probability parameter `P`. `P` can be a vector, a matrix, or a multidimensional array. The size of `R` is the size of `P`.

`R = geornd(P,v)` generates geometric random numbers with probability parameter `P`, where `v` is a row vector. If `v` is a 1-by-2 vector, `R` is a matrix with `v(1)` rows and `v(2)` columns. If `v` is 1-by-n, `R` is an n-dimensional array.

`R = geornd(P,m,n)` generates geometric random numbers with probability parameter `P`, where scalars m and n are the row and column dimensions of `R`.

The parameters in `P` must lie on the interval [0 1].

Example

```
r1 = geornd(1 ./ 2.^(1:6))
r1 =
   2  10   2   5   2  60

r2 = geornd(0.01,[1 5])
r2 =
  65  18  334  291  63

r3 = geornd(0.5,1,6)
r3 =
   0   7   1   3   1   0
```

See Also

`geocdf`, `geoinv`, `geopdf`, `geostat`

Purpose

Mean and variance of geometric distribution

Syntax

```
[M,V] = geostat(P)
```

Description

`[M,V] = geostat(P)` returns the mean of and variance for the geometric distribution with parameters specified by `P`.

The mean of the geometric distribution with parameter p is q/p, where $q = 1\text{-}p$. The variance is q/p^2.

Examples

```
[m,v] = geostat(1./(1:6))
m =
     0   1.0000   2.0000   3.0000   4.0000   5.0000
v =
     0   2.0000   6.0000  12.0000  20.0000  30.0000
```

See Also

`geocdf`, `geoinv`, `geopdf`, `geornd`

Purpose

Generalized extreme value cumulative distribution function

Syntax

```
P = gevcdf(X,K,sigma,mu)
```

Description

`P = gevcdf(X,K,sigma,mu)` returns the cdf of the generalized extreme value (GEV) distribution with shape parameter `K`, scale parameter `sigma`, and location parameter, `mu`, evaluated at the values in `X`. The size of `P` is the common size of the input arguments. A scalar input functions as a constant matrix of the same size as the other inputs.

Default values for `K`, `sigma`, and `mu` are 0, 1, and 0, respectively.

When `K < 0`, the GEV is the type III extreme value distribution. When `K > 0`, the GEV distribution is the type II, or Frechet, extreme value distribution. If `w` has a Weibull distribution as computed by the `wblcdf` function, then `-w` has a type III extreme value distribution and `1/w` has a type II extreme value distribution. In the limit as `K` approaches 0, the GEV is the mirror image of the type I extreme value distribution as computed by the `evcdf` function.

The mean of the GEV distribution is not finite when `K` ≥ `1`, and the variance is not finite when `K` ≥ `1/2`. The GEV distribution has positive density only for values of `X` such that `K*(X-mu)/sigma > -1`.

References

[1] Embrechts, P., C. Klüppelberg, and T. Mikosch (1997) *Modelling Extremal Events for Insurance and Finance*, Springer.

[2] Kotz, S. and S. Nadarajah (2001) *Extreme Value Distributions: Theory and Applications*, World Scientific Publishing Company.

See Also

`evcdf`, `gevfit`, `gevinv`, `gevlike`, `gevpdf`, `gevrnd`, `gevstat`, `cdf`

Purpose

Parameter estimates and confidence intervals for generalized extreme value distributed data

Syntax

```
parmhat = gevfit(X)
[parmhat,parmci] = gevfit(X)
[parmhat,parmci] = gevfit(X,alpha)
[...] = gevfit(X,alpha,options)
```

Description

`parmhat = gevfit(X)` returns maximum likelihood estimates of the parameters for the generalized extreme value (GEV) distribution given the data in X. `parmhat(1)` is the shape parameter, `K`, `parmhat(2)` is the scale parameter, `sigma`, and `parmhat(3)` is the location parameter, `mu`.

`[parmhat,parmci] = gevfit(X)` returns 95% confidence intervals for the parameter estimates.

`[parmhat,parmci] = gevfit(X,alpha)` returns `100(1-alpha)`% confidence intervals for the parameter estimates.

`[...]  = gevfit(X,alpha,options)` specifies control parameters for the iterative algorithm used to compute ML estimates. This argument can be created by a call to `statset`. See `statset('gevfit')` for parameter names and default values. Pass in `[]` for `alpha` to use the default values.

When `K < 0`, the GEV is the type III extreme value distribution. When `K > 0`, the GEV distribution is the type II, or Frechet, extreme value distribution. If `w` has a Weibull distribution as computed by the `wblfit` function, then `-w` has a type III extreme value distribution and `1/w` has a type II extreme value distribution. In the limit as `K` approaches 0, the GEV is the mirror image of the type I extreme value distribution as computed by the `evfit` function.

The mean of the GEV distribution is not finite when `K` ≥ `1`, and the variance is not finite when `K` ≥ `1/2`. The GEV distribution is defined for `K*(X-mu)/sigma > -1`.

References

[1] Embrechts, P., C. Klüppelberg, and T. Mikosch (1997) *Modelling Extremal Events for Insurance and Finance*, Springer.

[2] Kotz, S. and S. Nadarajah (2001) *Extreme Value Distributions: Theory and Applications*, World Scientific Publishing Company.

See Also

`evfit`, `gevcdf`, `gevinv`, `gevlike`, `gevpdf`, `gevrnd`, `gevstat`, `mle`, `statset`

Purpose

Inverse of generalized extreme value cumulative distribution function

Syntax

```
X = gevinv(P,K,sigma,mu)
```

Description

`X = gevinv(P,K,sigma,mu)` returns the inverse cdf of the generalized extreme value (GEV) distribution with shape parameter `K`, scale parameter `sigma`, and location parameter `mu`, evaluated at the values in `P`. The size of `X` is the common size of the input arguments. A scalar input functions as a constant matrix of the same size as the other inputs.

Default values for `K`, `sigma`, and `mu` are 0, 1, and 0, respectively.

When `K < 0`, the GEV is the type III extreme value distribution. When `K > 0`, the GEV distribution is the type II, or Frechet, extreme value distribution. If `w` has a Weibull distribution as computed by the `wblinv` function, then `-w` has a type III extreme value distribution and `1/w` has a type II extreme value distribution. In the limit as `K` approaches 0, the GEV is the mirror image of the type I extreme value distribution as computed by the `evinv` function.

The mean of the GEV distribution is not finite when `K` ≥ `1`, and the variance is not finite when `K` ≥ `1/2`. The GEV distribution has positive density only for values of `X` such that `K*(X-mu)/sigma > -1`.

References

[1] Embrechts, P., C. Klüppelberg, and T. Mikosch (1997) *Modelling Extremal Events for Insurance and Finance*, Springer.

[2] Kotz, S. and S. Nadarajah (2001) *Extreme Value Distributions: Theory and Applications*, World Scientific Publishing Company.

See Also

`evinv`, `gevfit`, `gevcdf`, `gevlike`, `gevpdf`, `gevrnd`, `gevstat`, `icdf`

gevlike

Purpose

Negative log-likelihood for generalized extreme value distribution

Syntax

```
nlogL = gevlike(params,data)
[nlogL,ACOV] = gevlike(params,data)
```

Description

`nlogL = gevlike(params,data)` returns the negative of the log-likelihood `nlogL` for the generalized extreme value (GEV) distribution, evaluated at parameters `params(1) = K`, `params(2) = sigma`, and `params(3) = mu`, given `data`.

`[nlogL,ACOV] = gevlike(params,data)` returns the inverse of Fisher's information matrix, `ACOV`. If the input parameter values in `params` are the maximum likelihood estimates, the diagonal elements of `ACOV` are their asymptotic variances. `ACOV` is based on the observed Fisher's information, not the expected information.

When `K < 0`, the GEV is the type III extreme value distribution. When `K > 0`, the GEV distribution is the type II, or Frechet, extreme value distribution. If `w` has a Weibull distribution as computed by the `wbllike` function, then `-w` has a type III extreme value distribution and `1/w` has a type II extreme value distribution. In the limit as `K` approaches 0, the GEV is the mirror image of the type I extreme value distribution as computed by the `evlike` function.

The mean of the GEV distribution is not finite when `K` ≥ `1`, and the variance is not finite when `K` ≥ `1/2`. The GEV distribution has positive density only for values of `X` such that `K*(X-mu)/sigma > -1`.

References

[1] Embrechts, P., C. Klüppelberg, and T. Mikosch (1997) *Modelling Extremal Events for Insurance and Finance*, Springer.

[2] Kotz, S. and S. Nadarajah (2001) *Extreme Value Distributions: Theory and Applications*, World Scientific Publishing Company.

See Also

`evlike`, `gevfit`, `gevinv`, `gevcdf`, `gevpdf`, `gevrnd`, `gevstat`

Purpose

Generalized extreme value probability density function

Syntax

```
Y = gevpdf(X,K,sigma,mu)
```

Description

`Y = gevpdf(X,K,sigma,mu)` returns the pdf of the generalized extreme value (GEV) distribution with shape parameter `K`, scale parameter `sigma`, and location parameter, `mu`, evaluated at the values in `X`. The size of `Y` is the common size of the input arguments. A scalar input functions as a constant matrix of the same size as the other inputs.

Default values for `K`, `sigma`, and `mu` are 0, 1, and 0, respectively.

When `K < 0`, the GEV is the type III extreme value distribution. When `K > 0`, the GEV distribution is the type II, or Frechet, extreme value distribution. If `w` has a Weibull distribution as computed by the `wblpdf` function, then `-w` has a type III extreme value distribution and `1/w` has a type II extreme value distribution. In the limit as `K` approaches 0, the GEV is the mirror image of the type I extreme value distribution as computed by the `evcdf` function.

The mean of the GEV distribution is not finite when `K` ≥ `1`, and the variance is not finite when `K` ≥ `1/2`. The GEV distribution has positive density only for values of `X` such that `K*(X-mu)/sigma > -1`.

References

[1] Embrechts, P., C. Klüppelberg, and T. Mikosch (1997) *Modelling Extremal Events for Insurance and Finance*, Springer.

[2] Kotz, S. and S. Nadarajah (2001) *Extreme Value Distributions: Theory and Applications*, World Scientific Publishing Company.

See Also

`evpdf`, `gevfit`, `gevinv`, `gevlike`, `gevcdf`, `gevrnd`, `gevstat`, `pdf`

gevrnd

Purpose

Random numbers from generalized extreme value distribution

Syntax

```
R = gevrnd(K,sigma,mu)
R = gevrnd(K,sigma,mu,M,N,...)
R = gevrnd(K,sigma,mu,[M,N,...])
```

Description

`R = gevrnd(K,sigma,mu)` returns an array of random numbers chosen from the generalized extreme value (GEV) distribution with shape parameter `K`, scale parameter `sigma`, and location parameter, `mu`. The size of `R` is the common size of the input arguments if all are arrays. If any parameter is a scalar, the size of `R` is the size of the other parameters.

`R = gevrnd(K,sigma,mu,M,N,...)` or

`R = gevrnd(K,sigma,mu,[M,N,...])` returns an m-by-n-by-... array.

When `K < 0`, the GEV is the type III extreme value distribution. When `K > 0`, the GEV distribution is the type II, or Frechet, extreme value distribution. If `w` has a Weibull distribution as computed by the `wblrnd` function, then `-w` has a type III extreme value distribution and `1/w` has a type II extreme value distribution. In the limit as `K` approaches 0, the GEV is the mirror image of the type I extreme value distribution as computed by the `evrnd` function.

The mean of the GEV distribution is not finite when `K` ≥ `1`, and the variance is not finite when `K` ≥ `1/2`. The GEV distribution has positive density only for values of `X` such that `K*(X-mu)/sigma > -1`.

References

[1] Embrechts, P., C. Klüppelberg, and T. Mikosch (1997) *Modelling Extremal Events for Insurance and Finance*, Springer.

[2] Kotz, S. and S. Nadarajah (2001) *Extreme Value Distributions: Theory and Applications*, World Scientific Publishing Company.

See Also

`evrnd`, `gevfit`, `gevinv`, `gevlike`, `gevpdf`, `gevcdf`, `gevstat`, `random`

Purpose

Mean and variance of generalized extreme value distribution

Syntax

```
[M,V] = gevstat(K,sigma,mu)
```

Description

`[M,V] = gevstat(K,sigma,mu)` returns the mean of and variance for the generalized extreme value (GEV) distribution with shape parameter `K`, scale parameter `sigma`, and location parameter, `mu`. The sizes of `M` and `V` are the common size of the input arguments. A scalar input functions as a constant matrix of the same size as the other inputs.

Default values for `K`, `sigma`, and `mu` are 0, 1, and 0, respectively.

When `K < 0`, the GEV is the type III extreme value distribution. When `K > 0`, the GEV distribution is the type II, or Frechet, extreme value distribution. If `w` has a Weibull distribution as computed by the `wblstat` function, then `-w` has a type III extreme value distribution and `1/w` has a type II extreme value distribution. In the limit as `K` approaches 0, the GEV is the mirror image of the type I extreme value distribution as computed by the `evstat` function.

The mean of the GEV distribution is not finite when `K` ≥ `1`, and the variance is not finite when `K` ≥ `1/2`. The GEV distribution has positive density only for values of `X` such that `K*(X-mu)/sigma > -1`.

References

[1] Embrechts, P., C. Klüppelberg, and T. Mikosch (1997) *Modelling Extremal Events for Insurance and Finance*, Springer.

[2] Kotz, S. and S. Nadarajah (2001) *Extreme Value Distributions: Theory and Applications*, World Scientific Publishing Company.

See Also

`evstat`, `gevfit`, `gevinv`, `gevlike`, `gevpdf`, `gevrnd`, `gevcdf`

Purpose

Interactive line plot

Syntax

```
gline(fig)
h = gline(fig)
gline
```

Description

`gline(fig)` allows you to draw a line segment in the figure `fig` by clicking the pointer at the two endpoints. A rubber band line tracks the pointer movement.

`h = gline(fig)` returns the handle to the line in `h`.

`gline` with no input arguments draws in the current figure.

See Also

`refline`, `gname`

Purpose

Demo of generalized linear models

Syntax

```
glmdemo
```

Description

`glmdemo` begins a slide show demonstration of generalized linear models. The slides indicate when generalized linear models are useful, how to fit generalized linear models using the `glmfit` function, and how to make predictions using the `glmval` function.

Note To run this demo from the command line, type `playshow glmdemo`.

See Also

`glmfit`, `glmval`

Purpose

Generalized linear model fit

Syntax

```
b = glmfit(X,y,distr)
b = glmfit(X,y,distr,param1,val1,param2,val2,...)
[b,dev] = glmfit(...)
[b,dev,stats] = glmfit(...)
```

Description

`b = glmfit(X,y,distr)` returns a p-by-1 vector `b` of coefficient estimates for a generalized linear regression of the responses in `y` on the predictors in `X`, using the distribution *`distr`*. `X` is an n-by-p matrix of p predictors at each of n observations. *`distr`* can be any of the following strings: `'binomial'`, `'gamma'`, `'inverse gaussian'`, `'normal'` (the default), and `'poisson'`.

In most cases, `y` is an n-by-1 vector of observed responses. For the binomial distribution, `y` can be a binary vector indicating success or failure at each observation, or a two column matrix with the first column indicating the number of successes for each observation and the second column indicating the number of trials for each observation.

This syntax uses the canonical link (see below) to relate the distribution to the predictors.

Note By default, `glmfit` adds a first column of ones to `X`, corresponding to a constant term in the model. Do not enter a column of ones directly into `X`. You can change the default behavior of `glmfit` using the `'constant'` parameter, below.

`glmfit` treats `NaN`s in either `X` or `y` as missing values, and ignores them.

`b = glmfit(X,y,distr,param1,val1,param2,val2,...)` additionally allows you to specify optional parameter name/value pairs to control the model fit. Acceptable parameters are as follows:

Parameter	Value	Meaning
`'link'`	`'identity'`, default for the distribution `'normal'`	$\mu = Xb$
	`'log'`, default for the distribution `'poisson'`	$\log(\mu) = Xb$
	`'logit'`, default for the distribution `'binomial'`	$\log(\mu/(1–\mu)) = Xb$
	`'probit'`	$\text{norminv}(\mu) = Xb$
	`'comploglog'`	$\log(-\log(1–\mu)) = Xb$
	`'reciprocal'`	$1/\mu = Xb$
	`'loglog'`, default for the distribution `'gamma'`	$\log(-\log(\mu)) = Xb$
	`p` (a number), default for the distribution `'inverse gaussian'` (with $p = -2$)	$\mu^p = Xb$
	cell array of the form `{FL FD FI}`, containing three function handles, created using `@`, that define the link (`FL`), the derivative of the link (`FD`), and the inverse link (`FI`).	User-specified link function

Parameter	Value	Meaning
'estdisp'	'on'	Estimates a dispersion parameter for the binomial or Poisson distribution
	'off' (Default for binomial or Poisson distribution)	Uses the theoretical value of 1.0 for those distributions
'offset'	Vector	Used as an additional predictor variable, but with a coefficient value fixed at 1.0
'weights'	Vector of prior weights, such as the inverses of the relative variance of each observation	
'constant'	'on' (default)	Includes a constant term in the model. The coefficient of the constant term is the first element of b.
	'off'	Omit the constant term

[b,dev] = glmfit(...)returns dev, the deviance of the fit at the solution vector. The deviance is a generalization of the residual sum of squares. It is possible to perform an analysis of deviance to compare several models, each a subset of the other, and to test whether the model with more terms is significantly better than the model with fewer terms.

[b,dev,stats] = glmfit(...) returns dev and stats.

stats is a structure with the following fields:

- beta — Coefficient estimates b
- dfe — Degrees of freedom for error

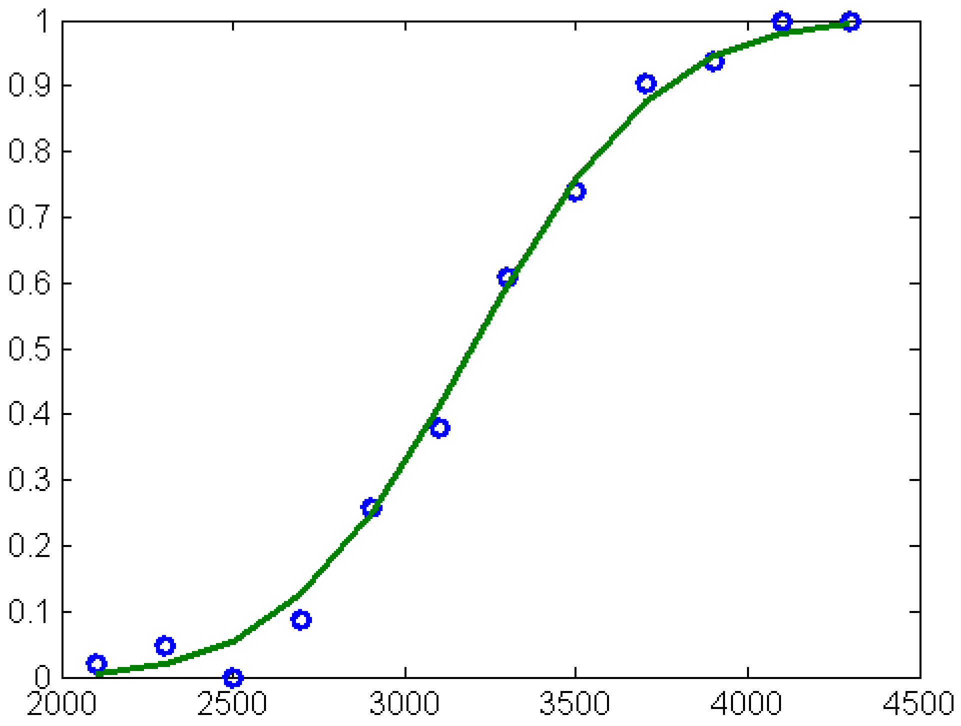
- `s` — Theoretical or estimated dispersion parameter
- `sfit` — Estimated dispersion parameter
- `se` — Vector of standard errors of the coefficient estimates `b`
- `coeffcorr` — Correlation matrix for `b`
- `covb` — Estimated covariance matrix for B
- `t` — *t* statistics for `b`
- `p` — *p*-values for `b`
- `resid` — Vector of residuals
- `residp` — Vector of Pearson residuals
- `residd` — Vector of deviance residuals
- `resida` — Vector of Anscombe residuals

If you estimate a dispersion parameter for the binomial or Poisson distribution, then `stats.s` is set equal to `stats.sfit`. Also, the elements of `stats.se` differ by the factor `stats.s` from their theoretical values.

Example

Fit a probit regression model for `y` on `x`. Each `y(i)` is the number of successes in `n(i)` trials.

```
x = [2100 2300 2500 2700 2900 3100 ...
     3300 3500 3700 3900 4100 4300]';
n = [48 42 31 34 31 21 23 23 21 16 17 21]';
y = [1 2 0 3 8 8 14 17 19 15 17 21]';
b = glmfit(x,[y n],'binomial','link','probit');
yfit = glmval(b, x,'probit','size', n);
plot(x, y./n,'o',x,yfit./n,'-','LineWidth',2)
```



References

[1] Dobson, A. J., *An Introduction to Generalized Linear Models*, CRC Press, 1990.

[2] MuCullagh, P., and J. A. Nelder, *Generalized Linear Models*, 2nd edition, Chapman & Hall, 1990.

[3] Collett, D., *Modelling Binary Data*, 2nd edition, Chapman & Hall/CRC Press, 2002.

See Also

`glmval`, `regress`, `regstats`

Purpose

Values and prediction intervals for generalized linear models

Syntax

```
yhat = glmval(b,X,link)
[yhat,dylo,dyhi] = glmval(b,X,link,stats)
[...] = glmval(...,param1,val1,param2,val2,...)
```

Description

`yhat = glmval(b,X,`*`link`*`)` computes predicted values for the generalized linear model with link function `link` and predictors `X`. Distinct predictor variables should appear in different columns of `X`. `b` is a vector of coefficient estimates as returned by the `glmfit` function. `link` can be any of the strings used as values for the `link` parameter in the `glmfit` function.

Note By default, `glmval` adds a first column of ones to `X`, corresponding to a constant term in the model. Do not enter a column of ones directly into `X`. You can change the default behavior of `glmval` using the `'constant'` parameter, below.

`[yhat,dylo,dyhi] = glmval(b,X,`*`link`*`,stats)` also computes 95% confidence bounds for the predicted values. When the `stats` structure output of the `glmfit` function is specified, `dylo` and `dyhi` are also returned. `dylo` and `dyhi` define a lower confidence bound of `yhat-dylo`, and an upper confidence bound of `yhat+dyhi`. Confidence bounds are nonsimultaneous, and apply to the fitted curve, not to a new observation.

`[...]  = glmval(...,`*`param1`*`,`*`val1`*`,`*`param2`*`,`*`val2`*`,...)` specifies optional parameter name/value pairs to control the predicted values. Acceptable parameters are:

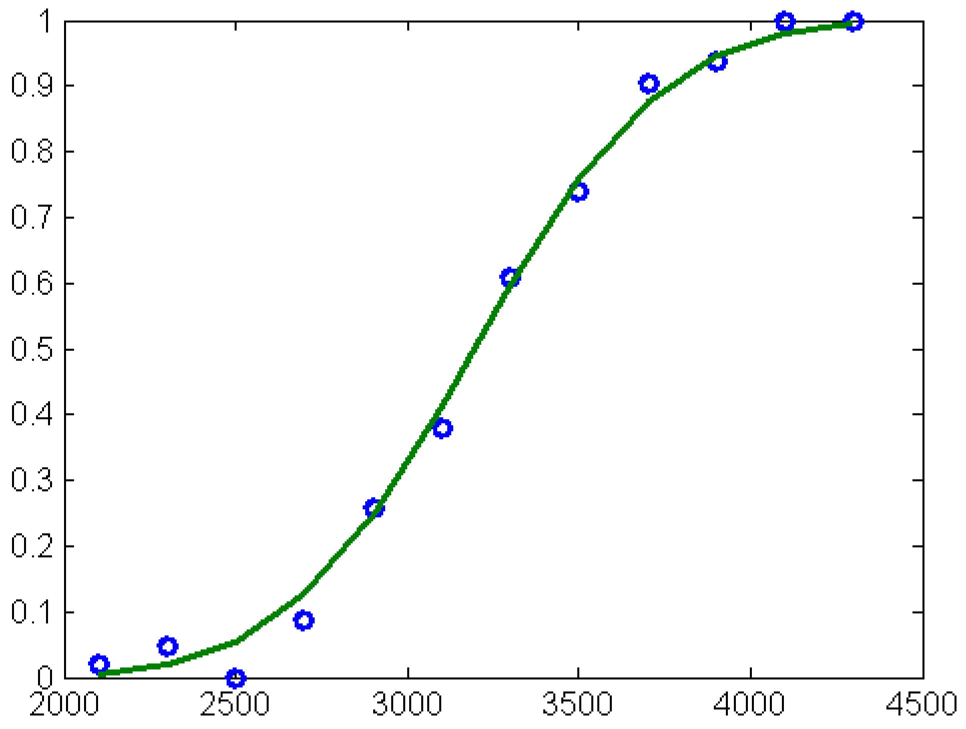
Parameter	Value
`'confidence'` — the confidence level for the confidence bounds	A scalar between 0 and 1
`'size'` — the size parameter (N) for a binomial model	A scalar, or a vector with one value for each row of X

Parameter	Value
`'offset'` — used as an additional predictor variable, but with a coefficient value fixed at 1.0	A vector
`'constant'`	• `'on'` — Includes a constant term in the model. The coefficient of the constant term is the first element of `b`. • `'off'` — Omit the constant term

Example

Fit a probit regression model for `y` on `x`. Each `y(i)` is the number of successes in `n(i)` trials.

```
x = [2100 2300 2500 2700 2900 3100 ...
     3300 3500 3700 3900 4100 4300]';
n = [48 42 31 34 31 21 23 23 21 16 17 21]';
y = [1 2 0 3 8 8 14 17 19 15 17 21]';
b = glmfit(x,[y n],'binomial','link','probit');
yfit = glmval(b,x,'probit','size',n);
plot(x, y./n,'o',x,yfit./n,'-')
```



References

[1] Dobson, A. J., *An Introduction to Generalized Linear Models*, CRC Press, 1990.

[2] MuCullagh, P., and J. A. Nelder, *Generalized Linear Models*, 2nd edition, Chapman & Hall, 1990.

[3] Collett, D., *Modelling Binary Data*, 2nd edition, Chapman & Hall/CRC Press, 2002.

See Also

`glmfit`

Purpose

Plot stars or Chernoff faces for multivariate data

Syntax

```
glyphplot(X)
glyphplot(X,'Glyph','face')
glyphplot(X,'Glyph','face','Features',F)
glyphplot(X,...,'Grid',[rows,cols])
glyphplot(X,...,'Grid',[rows,cols],'Page',page)
glyphplot(X,...,'Centers',C)
glyphplot,...,'Centers',C,'Radius',r)
glyphplot(X,...,'ObsLabels',labels)
glyphplot(X,...,'Standardize',method)
glyphplot(X,...,PropertyName,PropertyValue,...)
h = glyphplot(X,...)
```

Description

`glyphplot(X)` creates a star plot from the multivariate data in the n-by-p matrix `X`. Rows of `X` correspond to observations, columns to variables. A star plot represents each observation as a “star” whose ith spoke is proportional in length to the ith coordinate of that observation. `glyphplot` standardizes `X` by shifting and scaling each column separately onto the interval `[0,1]` before making the plot, and centers the glyphs on a rectangular grid that is as close to square as possible. `glyphplot` treats `NaN`s in `X` as missing values, and does not plot the corresponding rows of `X`. `glyphplot(X,'Glyph','star')` is a synonym for `glyphplot(X)`.

`glyphplot(X,'Glyph','face')` creates a face plot from `X`. A face plot represents each observation as a “face,” whose ith facial feature is drawn with a characteristic proportional to the i-th coordinate of that observation. The features are described in “Face Features” on page 14-312.

`glyphplot(X,'Glyph','face','Features',F)` creates a face plot where the i-th element of the index vector `F` defines which facial feature will represent the ith column of `X`. `F` must contain integers from `0` to `17`, where zeros indicate that the corresponding column of `X` should not be plotted. See “Face Features” on page 14-312 for more information.

`glyphplot(X,...,'Grid',[rows,cols])` organizes the glyphs into a `rows`-by-`cols` grid.

`glyphplot(X,...,'Grid',[rows,cols],'Page',page)` organizes the glyph into one or more pages of a `rows`-by-`cols` grid, and displays the `page`'th page. If `page` is a vector, `glyphplot` displays multiple pages in succession. If `page` is `'all'`, `glyphplot` displays all pages. If `page` is `'scroll'`, `glyphplot` displays a single plot with a scrollbar.

`glyphplot(X,...,'Centers',C)` creates a plot with each glyph centered at the locations in the n-by-2 matrix `C`.

`glyphplot,...,'Centers',C,'Radius',r)` creates a plot with glyphs positioned using `C`, and scale the glyphs so the largest has radius `r`.

`glyphplot(X,...,'ObsLabels',labels)` labels each glyph with the text in the character array or cell array of strings `labels`. By default, the glyphs are labelled `1:N`. Pass in `''` for no labels.

`glyphplot(X,...,'Standardize',`*`method`*`)` standardizes `X` before making the plot. Choices for *`method`* are

- `'column'` — Maps each column of `X` separately onto the interval [0,1]. This is the default.
- `'matrix'` — Maps the entire matrix `X` onto the interval `[0,1]`.
- `'PCA'` — Transforms `X` to its principal component scores, in order of decreasing eigenvalue, and maps each one onto the interval `[0,1]`.
- `'off'` — No standardization. Negative values in `X` may make a star plot uninterpretable.

`glyphplot(X,...,`*`PropertyName`*`,`*`PropertyValue`*`,...)` sets properties to the specified property values for all line graphics objects created by `glyphplot`.

`h = glyphplot(X,...)` returns a matrix of handles to the graphics objects created by `glyphplot`. For a star plot, `h(:,1)` and `h(:,2)` contain handles to the line objects for each star's perimeter and spokes, respectively. For a face plot, `h(:,1)` and `h(:,2)` contain object handles to the lines making up each face and to the pupils, respectively. `h(:,3)` contains handles to the text objects for the labels, if present.

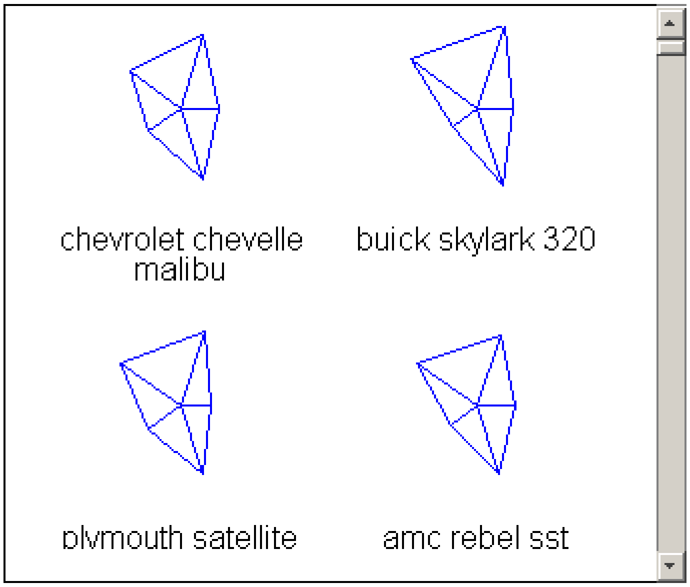
Face Features

The following table describes the correspondence between the columns of the vector `F`, the value of the `'Features'` input parameter, and the facial features of the glyph plot. If `X` has fewer than 17 columns, unused features are displayed at their default value.

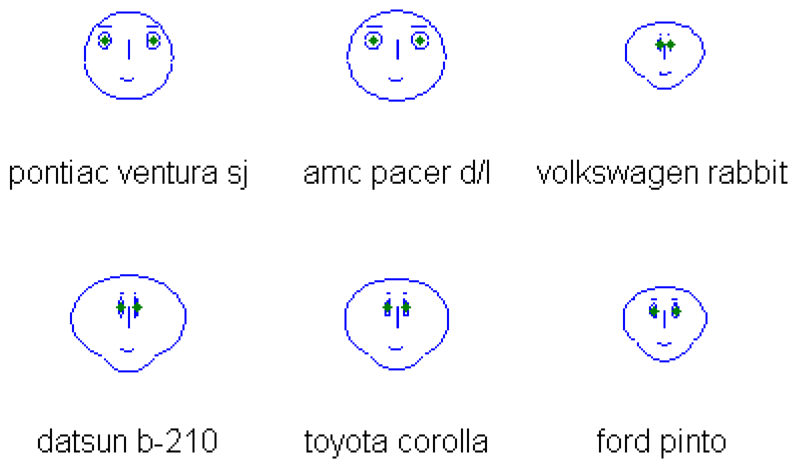
Column	Facial Feature
1	Size of face
2	Forehead/jaw relative arc length
3	Shape of forehead
4	Shape of jaw
5	Width between eyes
6	Vertical position of eyes
7	Height of eyes
8	Width of eyes (this also affects eyebrow width)
9	Angle of eyes (this also affects eyebrow angle)
10	Vertical position of eyebrows
11	Width of eyebrows (relative to eyes)
12	Angle of eyebrows (relative to eyes)
13	Direction of pupils
14	Length of nose
15	Vertical position of mouth
16	Shape of mouth
17	Mouth arc length

Examples

```
load carsmall
X = [Acceleration Displacement Horsepower MPG Weight];

glyphplot(X,'Standardize','column','ObsLabels',Model,...
          'grid',[2 2],'page','scroll');
```



```
glyphplot(X,'Glyph','face','ObsLabels',Model,...
          'grid',[2 3],'page',9);
```



See Also `andrewsplot`, `parallelcoords`

Purpose

Label plotted points with their case names or case number

Syntax

```
gname(cases)
gname
h = gname(cases,line_handle)
```

Description

`gname(cases)` displays a figure window and waits for you to press a mouse button or a keyboard key. The input argument `cases` is a character array or a cell array of strings, in which each row of the character array or each element of the cell array contains the case name of a point. Moving the mouse over the graph displays a pair of cross-hairs. If you position the cross-hairs near a point with the mouse and click once, the graph displays the name of the city corresponding to that point. Alternatively, you can click and drag the mouse to create a rectangle around several points. When you release the mouse button, the graph displays the labels for all points in the rectangle. Right-click a point to remove its label. When you are done labelling points, press the **Enter** or **Escape** key to stop labeling.

`gname` with no arguments labels each case with its case number.

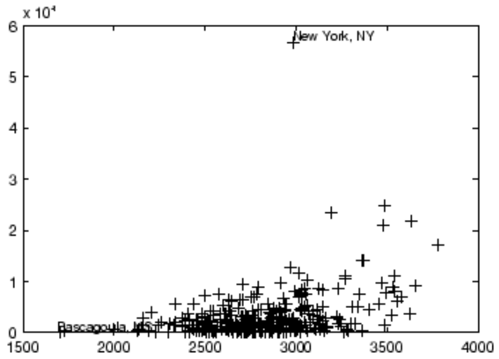
`h = gname(cases,line_handle)` returns a vector of handles to the text objects on the plot. Use the scalar `line_handle` to identify the correct line if there is more than one line object on the plot.

You can use `gname` to label plots created by the `plot`, `scatter`, `gscatter`, `plotmatrix`, and `gplotmatrix` functions.

Example

This example uses the city ratings data sets to find out which cities are the best and worst for education and the arts.

```
load cities
education = ratings(:,6);
arts = ratings(:,7);
plot(education,arts,'+')
gname(names)
```



Click the point at the top of the graph to display its label, “New York.”

See Also

`gplotmatrix`, `gscatter`, `gtext`, `plot`, `plotmatrix`, `scatter`

Purpose

Generalized Pareto cumulative distribution function

Syntax

```
P = gpcdf(X,K,sigma,theta)
```

Description

`P = gpcdf(X,K,sigma,theta)` returns the cdf of the generalized Pareto (GP) distribution with the tail index (shape) parameter `K`, scale parameter `sigma`, and threshold (location) parameter, `theta`, evaluated at the values in `X`. The size of `P` is the common size of the input arguments. A scalar input functions as a constant matrix of the same size as the other inputs.

Default values for `K`, `sigma`, and `theta` are 0, 1, and 0, respectively.

When `K = 0` and `theta = 0`, the GP is equivalent to the exponential distribution. When `K > 0` and `theta = sigma`, the GP is equivalent to the Pareto distribution. The mean of the GP is not finite when `K` $\geq$ `1`, and the variance is not finite when `K` $\geq$ `1/2`. When `K` $\geq$ `0`, the GP has positive density for

`X > theta`, or, when

`K < 0`, $0 \leq \frac{X-\theta}{\sigma} \leq -\frac{1}{K}$.

References

[1] Embrechts, P., C. Klüppelberg, and T. Mikosch (1997) *Modelling Extremal Events for Insurance and Finance*, Springer.

[2] Kotz, S. and S. Nadarajah (2001) *Extreme Value Distributions: Theory and Applications*, World Scientific Publishing Company.

See Also

`gpfit`, `gpinv`, `gplike`, `gppdf`, `gprnd`, `gpstat`, `cdf`

Purpose

Parameter estimates and confidence intervals for generalized Pareto distributed data

Syntax

```
parmhat = gpfit(X)
[parmhat,parmci] = gpfit(X)
[parmhat,parmci] = gpfit(X,alpha)
[...] = gpfit(X,alpha,options)
```

Description

`parmhat = gpfit(X)` returns maximum likelihood estimates of the parameters for the two-parameter generalized Pareto (GP) distribution given the data in `X`. `parmhat(1)` is the tail index (shape) parameter, `K` and `parmhat(2)` is the scale parameter, `sigma`. `gpfit` does not fit a threshold (location) parameter.

`[parmhat,parmci] = gpfit(X)` returns 95% confidence intervals for the parameter estimates.

`[parmhat,parmci] = gpfit(X,alpha)` returns `100(1-alpha)`% confidence intervals for the parameter estimates.

`[...]  = gpfit(X,alpha,options)` specifies control parameters for the iterative algorithm used to compute ML estimates. This argument can be created by a call to `statset`. See `statset('gpfit')` for parameter names and default values.

Other functions for the generalized Pareto, such as `gpcdf` allow a threshold parameter, `theta`. However, `gpfit` does not estimate theta. It is assumed to be known, and subtracted from `X` before calling `gpfit`.

When `K = 0` and `theta = 0`, the GP is equivalent to the exponential distribution. When `K > 0` and `theta = sigma`, the GP is equivalent to the Pareto distribution. The mean of the GP is not finite when `K` ≥ `1`, and the variance is not finite when `K` ≥ `1/2`. When `K` ≥ `0`, the GP has positive density for

`X > theta`, or, when

$$0 \le \frac{X-\theta}{\sigma} \le -\frac{1}{K}$$

References

[1] Embrechts, P., C. Klüppelberg, and T. Mikosch (1997) *Modelling Extremal Events for Insurance and Finance*, Springer.

[2] Kotz, S. and S. Nadarajah (2001) *Extreme Value Distributions: Theory and Applications*, World Scientific Publishing Company.

See Also

`gpcdf`, `gpinv`, `gplike`, `gppdf`, `gprnd`, `gpstat`, `mle`, `statset`

Purpose

Inverse of generalized Pareto cumulative distribution function

Syntax

```
X = gpinv(P,K,sigma,theta)
```

Description

`X = gpinv(P,K,sigma,theta)` returns the inverse cdf for a generalized Pareto (GP) distribution with tail index (shape) parameter `K`, scale parameter `sigma`, and threshold (location) parameter `theta`, evaluated at the values in `P`. The size of `X` is the common size of the input arguments. A scalar input functions as a constant matrix of the same size as the other inputs.

Default values for `K`, `sigma`, and `theta` are 0, 1, and 0, respectively.

When `K = 0`, the GP is equivalent to the exponential distribution. When `K > 0`, the GP is equivalent to the Pareto distribution shifted to the origin. The mean of the GP is not finite when $K \geq 1$, and the variance is not finite when $K \geq 1/2$. When $K \geq 0$, the GP has positive density for

`X > theta`, or, when

`K < 0`, $0 \leq \frac{X-\theta}{\sigma} \leq -\frac{1}{K}$.

References

[1] Embrechts, P., C. Klüppelberg, and T. Mikosch (1997) *Modelling Extremal Events for Insurance and Finance*, Springer.

[2] Kotz, S. and S. Nadarajah (2001) *Extreme Value Distributions: Theory and Applications*, World Scientific Publishing Company.

See Also

`gpfit`, `gpcdf`, `gplike`, `gppdf`, `gprnd`, `gpstat`, `icdf`

Purpose

Negative log-likelihood for generalized Pareto distribution

Syntax

```
nlogL = gplike(params,data)
[nlogL,ACOV] = gplike(params,data)
```

Description

`nlogL = gplike(params,data)` returns the negative of the log-likelihood `nlogL` for the two-parameter generalized Pareto (GP) distribution, evaluated at parameters `params(1) = K`, `params(2) = sigma`, and `params(3) = mu`, given `data`.

`[nlogL,ACOV] = gplike(params,data)` returns the inverse of Fisher's information matrix, `ACOV`. If the input parameter values in `params` are the maximum likelihood estimates, the diagonal elements of `ACOV` are their asymptotic variances. `ACOV` is based on the observed Fisher's information, not the expected information.

When `K = 0`, the GP is equivalent to the exponential distribution. When `K > 0`, the GP is equivalent to the Pareto distribution shifted to the origin. The mean of the GP is not finite when `K` ≥ 1, and the variance is not finite when `K` ≥ `1/2`. When `K` ≥ `0`, the GP has positive density for

`X > theta`, or, when

`K < 0`, $0 \le \frac{X-\theta}{\sigma} \le -\frac{1}{K}$.

References

[1] Embrechts, P., C. Klüppelberg, and T. Mikosch (1997) *Modelling Extremal Events for Insurance and Finance*, Springer.

[2] Kotz, S. and S. Nadarajah (2001) *Extreme Value Distributions: Theory and Applications*, World Scientific Publishing Company.

See Also

`gpfit`, `gpinv`, `gpcdf`, `gppdf`, `gprnd`, `gpstat`

Purpose

Generalized Pareto probability density function

Syntax

```
P = gppdf(X,K,sigma,theta)
```

Description

`P = gppdf(X,K,sigma,theta)` returns the pdf of the generalized Pareto (GP) distribution with the tail index (shape) parameter `K`, scale parameter `sigma`, and threshold (location) parameter, `theta`, evaluated at the values in `X`. The size of `P` is the common size of the input arguments. A scalar input functions as a constant matrix of the same size as the other inputs.

Default values for `K`, `sigma`, and `theta` are 0, 1, and 0, respectively.

When `K = 0` and `theta = 0`, the GP is equivalent to the exponential distribution. When `K > 0` and `theta = sigma`, the GP is equivalent to the Pareto distribution. The mean of the GP is not finite when `K` ≥ `1`, and the variance is not finite when `K` ≥ `1/2`. When `K` ≥ `0`, the GP has positive density for

`X > theta`, or, when

`K < 0`, $0 \le \frac{X-\theta}{\sigma} \le -\frac{1}{K}$.

References

[1] Embrechts, P., C. Klüppelberg, and T. Mikosch (1997) *Modelling Extremal Events for Insurance and Finance*, Springer.

[2] Kotz, S. and S. Nadarajah (2001) *Extreme Value Distributions: Theory and Applications*, World Scientific Publishing Company.

See Also

`gpfit`, `gpinv`, `gplike`, `gpcdf`, `gprnd`, `gpstat`, `pdf`

Purpose

Plot matrix of scatter plots, by group

Syntax

```
gplotmatrix(x,y,group)
gplotmatrix(x,y,group,clr,sym,siz)
gplotmatrix(x,y,group,clr,sym,siz,doleg)
gplotmatrix(x,y,group,clr,sym,siz,doleg,dispopt)
gplotmatrix(x,y,group,clr,sym,siz,doleg,dispopt,xnam,ynam)
[h,ax,bigax] = gplotmatrix(...)
```

Description

`gplotmatrix(x,y,group)` creates a matrix of scatter plots. Each individual set of axes in the resulting figure contains a scatter plot of a column of `x` against a column of `y`. All plots are grouped by the grouping variable `group`. (See "Grouped Data" on page 2-41.)

`x` and `y` are matrices with the same number of rows. If `x` has *p* columns and `y` has *q* columns, the figure contains a *p*-by-*q* matrix of scatter plots. If you omit `y` or specify it as the empty matrix, `[]`, `gplotmatrix` creates a square matrix of scatter plots of columns of `x` against each other.

`group` is a grouping variable that can be a categorical variable, vector, string array, or cell array of strings. `group` must have the same number of rows as `x` and `y`. Points with the same value of `group` are placed in the same group, and appear on the graph with the same marker and color. Alternatively, `group` can be a cell array containing several grouping variables (such as `{g1 g2 g3}`); in that case, observations are in the same group if they have common values of all grouping variables.

`gplotmatrix(x,y,group,`*`clr`*`,`*`sym`*`,siz)` specifies the color, marker type, and size for each group. *`clr`* is a string array of colors recognized by the `plot` function. The default for *`clr`* is `'bgrcmyk'`. *`sym`* is a string array of symbols recognized by the `plot` command, with the default value `'.'`. `siz` is a vector of sizes, with the default determined by the `DefaultLineMarkerSize` property. If you do not specify enough values for all groups, `gplotmatrix` cycles through the specified values as needed.

`gplotmatrix(x,y,group,`*`clr`*`,`*`sym`*`,siz,`*`doleg`*`)` controls whether a legend is displayed on the graph (*`doleg`* is `'on'`, the default) or not (*`doleg`* is `'off'`).

`gplotmatrix(x,y,group,`*`clr`*`,`*`sym`*`,siz,`*`doleg`*`,`*`dispopt`*`)` controls what appears along the diagonal of a plot matrix of `y` versus `x`. Allowable values are `'none'`, to leave the diagonals blank, `'hist'` (the default), to plot histograms, or `'variable'`, to write the variable names.

`gplotmatrix(x,y,group,`*`clr`*`,`*`sym`*`,siz,`*`doleg`*`,`*`dispopt`*`,xnam,ynam)` specifies the names of the columns in the `x` and `y` arrays. These names are used to label the *x*- and *y*-axes. `xnam` and `ynam` must be character arrays or cell arrays of strings, with one name for each column of `x` and `y`, respectively.

`[h,ax,bigax] = gplotmatrix(...)` returns three arrays of handles. `h` is an array of handles to the lines on the graphs. The array's third dimension corresponds to groups in G. `ax` is a matrix of handles to the axes of the individual plots. If *`dispopt`* is `'hist'`, `ax` contains one extra row of handles to invisible axes in which the histograms are plotted. `bigax` is a handle to big (invisible) axes framing the entire plot matrix. `bigax` is fixed to point to the current axes, so a subsequent `title`, `xlabel`, or `ylabel` command will produce labels that are centered with respect to the entire plot matrix.

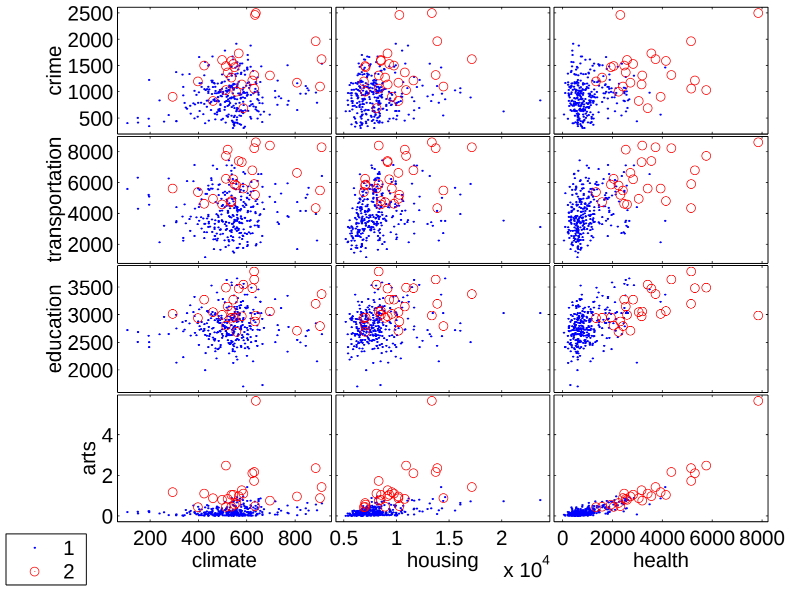
Example

Load the `cities` data. The `ratings` array has ratings of the cities in nine categories (category names are in the array `categories`). `group` is a code whose value is 2 for the largest cities. You can make scatter plots of the first three categories against the other four, grouped by the city size code.

```
load discrim
gplotmatrix(ratings(:,1:3),ratings(:,4:7),group)
```

The output figure (not shown) has an array of graphs with each city group represented by a different color. The graphs are a little easier to read if you specify colors and plotting symbols, label the axes with the rating categories, and move the legend off the graphs.

```
gplotmatrix(ratings(:,1:3),ratings(:,4:7),group,...
          'br','.o',[],'on','',categories(1:3,:),...
           categories(4:7,:))
```



See Also

`grpstats`, `gscatter`, `plotmatrix`

gprnd

Purpose Random numbers from generalized Pareto distribution

Syntax

```
R = gprnd(K,sigma,theta)
R = gprnd(K,sigma,theta,M,N,...)
R = gprnd(K,sigma,theta,[M,N,...])
```

Description `R = gprnd(K,sigma,theta)` returns an array of random numbers chosen from the generalized Pareto (GP) distribution with tail index (shape) parameter `K`, scale parameter `sigma`, and threshold (location) parameter, `theta`. The size of `R` is the common size of the input arguments if all are arrays. If any parameter is a scalar, the size of `R` is the size of the other parameters.

Default values for `K`, `sigma`, and `theta` are 0, 1, and 0, respectively.

`R = gprnd(K,sigma,theta,M,N,...)` or `R = gprnd(K,sigma,theta,[M,N,...])` returns an m-by-n-by-... array.

When `K = 0` and `theta = 0`, the GP is equivalent to the exponential distribution. When `K > 0` and `theta = sigma`, the GP is equivalent to the Pareto distribution. The mean of the GP is not finite when `K` ≥ `1`, and the variance is not finite when `K` ≥ `1/2`. When `K` ≥ `0`, the GP has positive density for

`X > theta`, or, when

$$0 \le \frac{X-\theta}{\sigma} \le -\frac{1}{K}$$

References

[1] Embrechts, P., C. Klüppelberg, and T. Mikosch (1997) *Modelling Extremal Events for Insurance and Finance*, Springer.

[2] Kotz, S. and S. Nadarajah (2001) *Extreme Value Distributions: Theory and Applications*, World Scientific Publishing Company.

See Also `gpfit`, `gpinv`, `gplike`, `gppdf`, `gpcdf`, `gpstat`, `random`

gpstat

Purpose Mean and variance of generalized Pareto distribution

Syntax

```
[M,V] = gpstat(X,K,sigma,theta)
```

Description

`[M,V] = gpstat(X,K,sigma,theta)` returns the mean of and variance for the generalized Pareto (GP) distribution with the tail index (shape) parameter `K`, scale parameter `sigma`, and threshold (location) parameter, `theta`.

The default value for `theta` is 0.

When `K = 0` and `theta = 0`, the GP is equivalent to the exponential distribution. When `K > 0` and `theta = sigma`, the GP is equivalent to the Pareto distribution. The mean of the GP is not finite when `K` ≥ `1`, and the variance is not finite when `K` ≥ `1/2`. When `K` ≥ `0`, the GP has positive density for `X > theta`, or when

`K < 0`, $0 \le \frac{X-\theta}{\sigma} \le -\frac{1}{K}$.

References

[1] Embrechts, P., C. Klüppelberg, and T. Mikosch (1997) *Modelling Extremal Events for Insurance and Finance*, Springer.

[2] Kotz, S. and S. Nadarajah (2001) *Extreme Value Distributions: Theory and Applications*, World Scientific Publishing Company.

See Also

`gpfit`, `gpinv`, `gplike`, `gppdf`, `gprnd`, `gpcdf`

Purpose Create index vector from grouping variable

Syntax `[G,GN]=grp2idx(group)`

Description `[G,GN]=grp2idx(group)` creates an index vector `G` from the grouping variable `group`. (See "Grouped Data" on page 2-41.) The variable `group` can be a categorical variable, a numeric vector, a character matrix (with each row representing a group name), or a cell array of strings stored as a column vector. The result `G` is a vector taking integer values from 1 up to the number of unique entries in `group`. `GN` is a cell array of names such that `GN(G)` reproduces `s` (with the exception of any differences in type).

See Also `gscatter`, `grpstats`

Purpose

Summary statistics by group

Syntax

```
means = grpstats(X)
means = grpstats(X,group)
grpstats(x,group,alpha)
[A,B,...] = grpstats(x,group,whichstats)
[...] = grpstats(X,group,whichstats,alpha)
```

Description

`means = grpstats(X)` computes the mean of the entire sample without grouping, where `X` is a matrix of observations.

`means = grpstats(X,group)` returns the means of each column of X by `group`. The array, `group` defines the grouping such that two elements of X are in the same group if their corresponding `group` values are the same. (See "Grouped Data" on page 2-41.) The grouping variable `group` can be a categorical variable, vector, string array, or cell array of strings. It can also be a cell array containing several grouping variables (such as `{g1 g2 g3}`) to group the values in `X` by each unique combination of grouping variable values.

`grpstats(x,group,alpha)` displays a plot of the means versus index with `100(1-alpha)`% confidence intervals around each mean.

`[A,B,...]  = grpstats(x,group,`*`whichstats`*`)` returns the statistics specified in *`whichstats`*. The input *`whichstats`* can be a single function handle or name, or a cell array containing multiple function handles or names. The number of outputs (`A`,`B`, ...) must match the number function handles and names in *`whichstats`*. The names can be chosen from among the following:

- `'mean'` — mean
- `'sem'` — standard error of the mean
- `'numel'` — count, or number of non-`NaN` elements
- `'gname'` — group name
- `'std'` — standard deviation
- `'var'` — variance

- `'meanci'` — 95% confidence interval for the mean
- `'predci'` — 95% prediction interval for a new observation

Each function included in *`whichstats`* must accept a vector of data and compute a descriptive statistic for it. For example, `@median` and `@skewness` are suitable functions. The size of the output is *Ngroups*-by-*Nvals*, where *Ngroups* is the number of groups and *Nvals* is the number of values returned by the function for a single group. The function may also accept a matrix of data and compute column statistics. In this case the output size is *Ngroups*-by-*Ncols*-by-*Nvals*, where *Ncols* is the number of columns of `X`.

`[...] = grpstats(X,group,`*`whichstats`*`,alpha)` specifies the confidence level as `100(1-alpha)`% for the `'meanci'` and `'predci'` options. It does not display a plot.

Example

```
load carsmall
[m,p,g] = grpstats(Weight,Model_Year,...
                   {'mean','predci','gname'})
n = length(m)
errorbar((1:n)',m,p(:,2)-m)
set(gca,'xtick',1:n,'xticklabel',g)
title('95% prediction intervals for mean weight by year')
```

See Also

`gscatter`, `grp2idx`, `grpstats (dataset)`

Purpose

Summary statistics by group for dataset arrays

Syntax

```
B = grpstats(A,groupvars)
B = grpstats(A,groupvars,whichstats)
B = grpstats(A,groupvars,whichstats,...,'DataVars',vars)
B = grpstats(A,groupvars,whichstats,...,'VarNames',names)
```

Description

`B = grpstats(A,groupvars)` returns a dataset array `B` that contains the means, computed by group, for variables in the dataset array `A`. The optional input `groupvars` specifies the variables in `A` that define the groups. `groupvars` can be a positive integer, a vector of positive integers, a variable name, a cell array containing one or more variable names, or a logical vector. `groupvars` can also be `[]` or omitted to compute the means of the variables in `A` without grouping. Grouping variables can be vectors of categorical, logical, or numeric values, a character array of strings, or a cell vector of strings. (See "Grouped Data" on page 2-41.)

`B` contains the grouping variables, plus a variable giving the number of observations in `A` for each group, plus a variable for each of the remaining variables in `A`. `B` contains one observation for each group of observations in `A`.

`grpstats` treats `NaN`s as missing values, and removes them.

`B = grpstats(A,groupvars,`*`whichstats`*`)` returns a dataset array `B` with variables for each of the statistics specified in *`whichstats`*, applied to each of the nongrouping variables in `A`. *`whichstats`* can be a single function handle or name, or a cell array containing multiple function handles or names. The names can be chosen from among the following:

- `'mean'` — mean
- `'sem'` — standard error of the mean
- `'numel'` — count, or number of non-`NaN` elements
- `'gname'` — group name
- `'std'` — standard deviation
- `'var'` — variance

- `'meanci'` — 95% confidence interval for the mean
- `'predci'` — 95% prediction interval for a new observation

Each function included in *`whichstats`* must accept a subset of the rows of a dataset variable, and compute column-wise descriptive statistics for it. A function should typically return a value that has one row but is otherwise the same size as its input data. For example, `@median` and `@skewness` are suitable functions to apply to a numeric dataset variable.

A summary statistic function may also return values with more than one row, provided the return values have the same number of rows each time `grpstats` applies the function to different subsets of data from a given dataset variable. For a dataset variable that is `nobs`-by-`m`-by-... if a summary statistic function returns values that are `nvals`-by-`m`-by-... then the corresponding summary statistic variable in `B` is `ngroups`-by-`m`-by-...-by-`nvals`, where `ngroups` is the number of groups in `A`.

`B = grpstats(A,groupvars,`*`whichstats`*`,...,'DataVars',vars)` specifies the variables in `A` to which the functions in *`whichstats`* should be applied. The output dataset arrays contain one summary statistic variable for each of the specified variables. `vars` is a positive integer, a vector of positive integers, a variable name, a cell array containing one or more variable names, or a logical vector.

`B = grpstats(A,groupvars,`*`whichstats`*`,...,'VarNames',names)` specifies the names of the variables in `B`. By default, `grpstats` uses the names from `A` for the grouping variables, and constructs names for the summary statistic variables based on the function name and the data variable names. The number of variables in `B` is `ngroupvars + 1 + ndatavars*nfuns`, where `ngroupvars` is the number of variables specified in `groupvars`, `ndatavars` is the number of variables specified in `vars`, and `nfuns` is the number of summary statistics specified in *`whichstats`*.

Example

Compute blood pressure statistics for the data in `hospital.mat`, by sex and smoker status:

```
load hospital
grpstats(hospital,...
         {'Sex','Smoker'},...
         {@median,@iqr},...
         'DataVars','BloodPressure')
ans =
             Sex       Smoker    GroupCount
 Female_0    Female    false     40
 Female_1    Female    true      13
 Male_0      Male      false     26
 Male_1      Male      true      21

             median_BloodPressure
 Female_0    119.5             79
 Female_1      129             91
 Male_0        119             79
 Male_1        129             92

             iqr_BloodPressure
 Female_0     6.5           5.5
 Female_1       8           5.5
 Male_0         7             6
 Male_1      10.5           4.5
```

See Also

grpstats, summary (dataset)

Purpose

Scatter plot, by group

Syntax

```
gscatter(x,y,group)
gscatter(x,y,group,clr,sym,siz)
gscatter(x,y,group,clr,sym,siz,doleg)
gscatter(x,y,group,clr,sym,siz,doleg,xnam,ynam)
h = gscatter(...)
```

Description

`gscatter(x,y,group)` creates a scatter plot of `x` and `y`, grouped by `group`. `x` and `y` are vectors of the same size. `group` is a grouping variable in the form of a categorical variable, vector, string array, or cell array of strings. (See "Grouped Data" on page 2-41.) Alternatively, `group` can be a cell array containing several grouping variables (such as `{g1 g2 g3}`), in which case observations are in the same group if they have common values of all grouping variables. Points in the same group and appear on the graph with the same marker and color.

`gscatter(x,y,group,`*`clr`*`,`*`sym`*`,siz)` specifies the color, marker type, and size for each group. *`clr`* is a string array of colors recognized by the `plot` function. The default for *`clr`* is `'bgrcmyk'`. *`sym`* is a string array of symbols recognized by the `plot` command, with the default value `'.'`. `siz` is a vector of sizes, with the default determined by the `'DefaultLineMarkerSize'` property. If you do not specify enough values for all groups, `gscatter` cycles through the specified values as needed.

`gscatter(x,y,group,`*`clr`*`,`*`sym`*`,siz,`*`doleg`*`)` controls whether a legend is displayed on the graph (*`doleg`* is `'on'`, the default) or not (*`doleg`* is `'off'`).

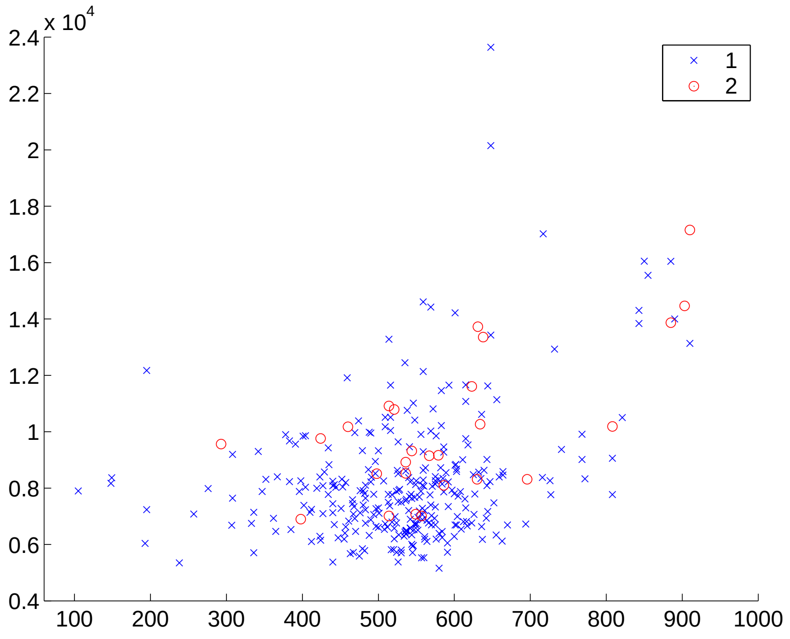
`gscatter(x,y,group,`*`clr`*`,`*`sym`*`,siz,`*`doleg`*`,xnam,ynam)` specifies the name to use for the *x*-axis and *y*-axis labels. If the `x` and `y` inputs are simple variable names and `xnam` and `ynam` are omitted, `gscatter` labels the axes with the variable names.

`h = gscatter(...)` returns an array of handles to the lines on the graph.

Example

Load the `cities` data and look at the relationship between the ratings for climate (first column) and housing (second column) grouped by city size. We'll also specify the colors and plotting symbols.

```
load discrim
gscatter(ratings(:,1),ratings(:,2),group,'br','xo')
```



See Also

`gplotmatrix`, `grpstats`, `scatter`

harmmean

Purpose

Harmonic mean of sample

Syntax

```
m = harmmean(X)
harmmean(X,dim)
```

Description

`m = harmmean(X)` calculates the harmonic mean of a sample. For vectors, `harmmean(x)` is the harmonic mean of the elements in `x`. For matrices, `harmmean(X)` is a row vector containing the harmonic means of each column. For *N*-dimensional arrays, `harmmean` operates along the first nonsingleton dimension of `X`.

`harmmean(X,dim)` takes the harmonic mean along dimension `dim` of `X`.

The harmonic mean is

$$m = \frac{n}{\sum_{i=1}^{n} \frac{1}{x_i}}$$

Examples

The arithmetic mean is greater than or equal to the harmonic mean.

```
x = exprnd(1,10,6);

harmonic = harmmean(x)
harmonic =
  0.3382  0.3200  0.3710  0.0540  0.4936  0.0907

average = mean(x)
average =
  1.3509  1.1583  0.9741  0.5319  1.0088  0.8122
```

See Also

`mean`, `median`, `geomean`, `trimmean`

Purpose

Plot histograms

Syntax

```
hist(y)
hist(y,nb)
hist(y,x)
[n,x] = hist(y,...)
```

Description

`hist(y)` draws a 10-bin histogram for the data in vector `y`. The bins are equally spaced between the minimum and maximum values in `y`.

`hist(y,nb)` draws a histogram with `nb` bins.

`hist(y,x)` draws a histogram using the bins in the vector `x`.

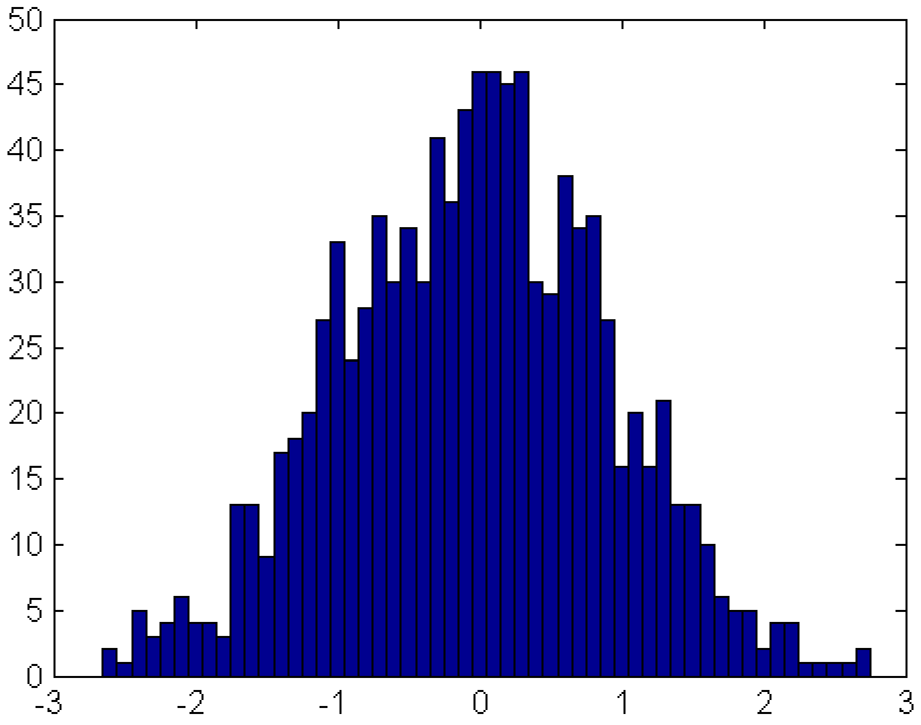
`[n,x] = hist(y,...)` do not draw graphs, but return vectors `n` and `x` containing the frequency counts and the bin locations such that `bar(x,n)` plots the histogram. This is useful in situations where more control is needed over the appearance of a graph, for example, to combine a histogram into a more elaborate plot statement.

The `hist` function is a part of the standard MATLAB language.

Examples

Generate bell-curve histograms from Gaussian data.

```
x = -2.9:0.1:2.9;
y = normrnd(0,1,1000,1);
hist(y,x)
```



See Also `hist3`, `histc`

Purpose

Three-dimensional histogram of bivariate data

Syntax

```
hist3(X)
hist3(X,nbins)
hist3(X,ctrs)
hist3(X,'Edges',edges)
N = hist3(X,...)
[N,C] = hist3(X,...)
hist3(...,param1,val1,param2,val2,...)
```

Description

`hist3(X)` bins the elements of the m-by-2 matrix `X` into a 10-by-10 grid of equally spaced containers, and plots a histogram. Each column of **`X`** corresponds to one dimension in the bin grid.

`hist3(X,nbins)` plots a histogram using an `nbins(1)`-by-`nbins(2)` grid of bins. `hist3(X,'Nbins',nbins)` is equivalent to `hist3(X,nbins)`.

`hist3(X,ctrs)`, where `ctrs` is a two-element cell array of numeric vectors with monotonically non-decreasing values, uses a two-dimensional grid of bins centered on `ctrs{1}` in the first dimension and on `ctrs{2}` in the second. `hist3` assigns rows of `X` falling outside the range of that grid to the bins along the outer edges of the grid, and ignores rows of `X` containing `NaN`s. `hist3(X,'Ctrs',ctrs)` is equivalent to `hist3(X,ctrs)`.

`hist3(X,'Edges',edges)`, where `edges` is a two-element cell array of numeric vectors with monotonically non-decreasing values, uses a two-dimensional grid of bins with edges at `edges{1}` in the first dimension and at `edges{2}` in the second. The (i, j)th bin includes the value `X(k,:)` if

```
edges{1}(i) <= X(k,1) < edges{1}(i+1)
edges{2}(j) <= X(k,2) < edges{2}(j+1)
```

Rows of `X` that fall on the upper edges of the grid, `edges{1}(end)` or `edges{2}(end)`, are counted in the `(I,j)`th or `(i,J)`th bins, where `I` and `J` are the lengths of `edges{1}` and `edges{2}`. `hist3` does not count rows of X falling outside the range of the grid. Use -Inf and Inf in edges to include all non-NaN values.

N = hist3(X,...) returns a matrix containing the number of elements of X that fall in each bin of the grid, and does not plot the histogram.

[N,C] = hist3(X,...) returns the positions of the bin centers in a 1-by-2 cell array of numeric vectors, and does not plot the histogram. hist3(ax,X,...) plots onto an axes with handle ax instead of the current axes. See the axes reference page for more information about handles to plots.

hist3(...,*param1*,*val1*,*param2*,*val2*,...) allows you to specify graphics parameter name/value pairs to fine-tune the plot.

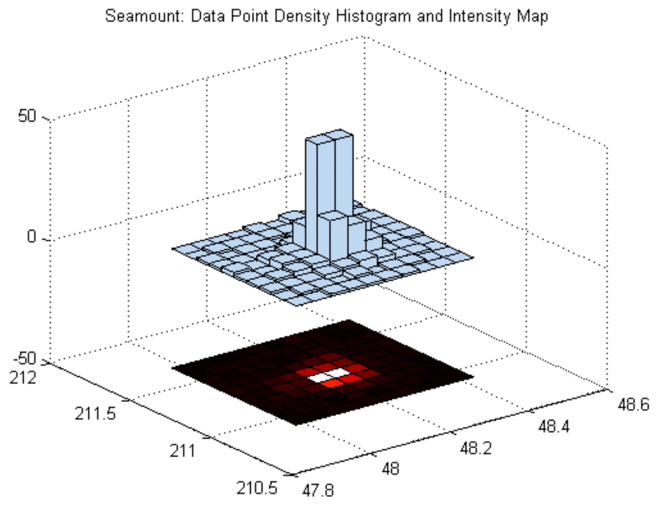
Example

Example 1

Make a 3-D figure using a histogram with a density plot underneath:

```
figure;
load seamount
dat = [-y,x]; % Grid corrected for negative y-values
hold on
hist3(dat) % Draw histogram in 2D

n = hist3(dat); % Extract histogram data;
                % default to 10x10 bins
n1 = n';
n1( size(n,1) + 1 ,size(n,2) + 1 ) = 0;

% Generate grid for 2-D projected view of intensities
xb = linspace(min(dat(:,1)),max(dat(:,1)),size(n,1)+1);
yb = linspace(min(dat(:,2)),max(dat(:,2)),size(n,1)+1);

% Make a pseudocolor plot on this grid
h = pcolor(xb,yb,n1);

% Set the z-level and colormap of the displayed grid
set(h, 'zdata', ones(size(n1)) * -max(max(n)))
```

```
colormap(hot) % heat map
title('Seamount: ...
      Data Point Density Histogram and Intensity Map');
grid on

% Display the default 3-D perspective view
view(3);
```

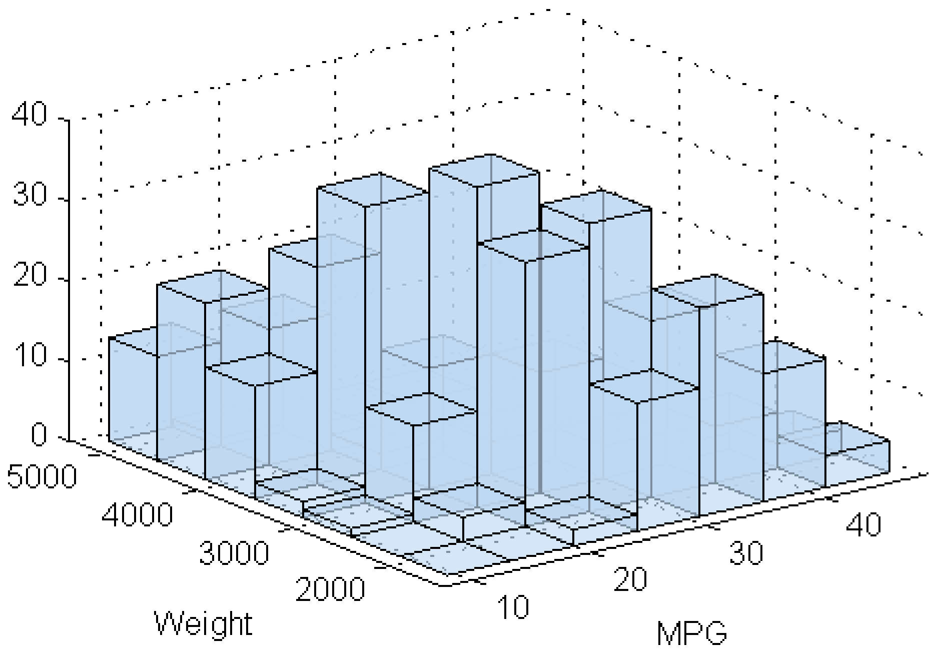


Example 2

Use the car data to make a histogram on a 7-by-7 grid of bins.

```
load carbig
X = [MPG,Weight];
hist3(X,[7 7]);
xlabel('MPG'); ylabel('Weight');

% Make a histogram with semi-transparent bars
hist3(X,[7 7],'FaceAlpha',.65);
xlabel('MPG'); ylabel('Weight');
```

```
set(gcf,'renderer','opengl');
```



```
% Specify bin centers, different in each direction.
% Get back counts, but don't make the plot.
cnt = hist3(X, {0:10:50 2000:500:5000});
```

See Also

`accumarray`, `bar`, `bar3`, `hist`, `histc`

Purpose

Histogram with superimposed normal density

Syntax

```
histfit(data,nbins)
histfit(data)
h = histfit(data,nbins)
```

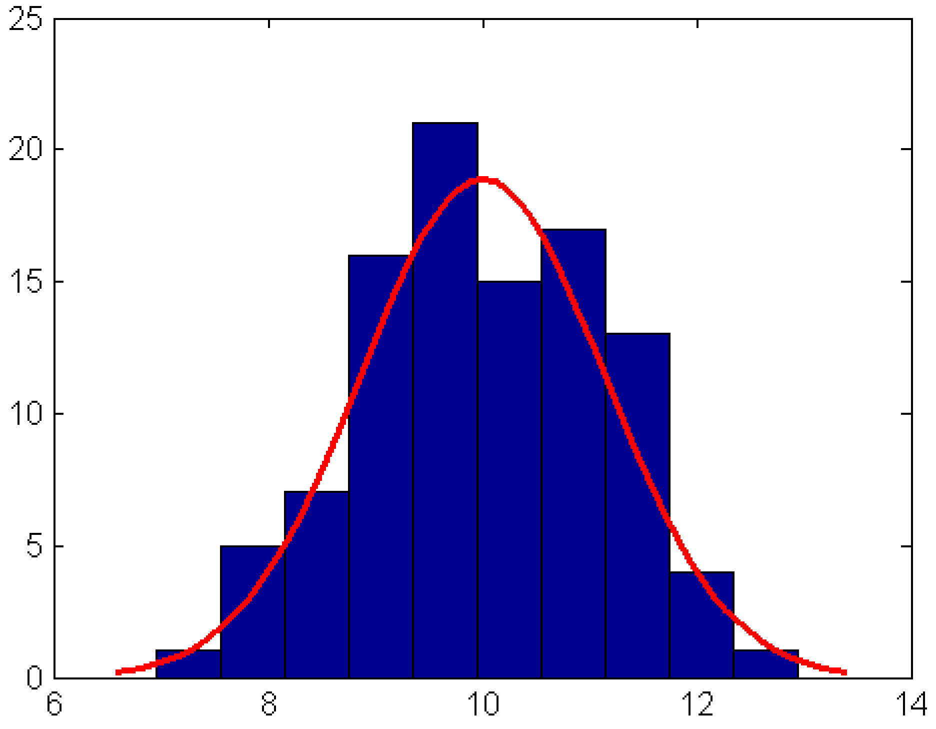
Description

`histfit(data,nbins)` plots a histogram of the values in the vector `data` using `nbins` bars in the histogram. `histfit(data)` with `nbins` is omitted, its value is set to the square root of the number of elements in `data`.

`h = histfit(data,nbins)` returns a vector of handles to the plotted lines, where `h(1)` is the handle to the histogram, `h(2)` is the handle to the density curve.

Example

```
r = normrnd(10,1,100,1);
histfit(r)
```



See Also `hist`, `hist3`, `normfit`

Purpose

Posterior state probabilities of sequence

Syntax

```
PSTATES = hmmdecode(seq,TRANS,EMIS)
[PSTATES,logpseq] = hmmdecode(...)
[PSTATES,logpseq,FORWARD,BACKWARD,S] = hmmdecode(...)
hmmdecode(...,'Symbols',SYMBOLS)
```

Description

`PSTATES = hmmdecode(seq,TRANS,EMIS)` calculates the posterior state probabilities, `PSTATES`, of the sequence `seq`, from a hidden Markov model. The posterior state probabilities are the conditional probabilities of being at state *k* at step *i*, given the observed sequence of symbols, `sym`. You specify the model by a transition probability matrix, `TRANS`, and an emissions probability matrix, `EMIS`. `TRANS(i,j)` is the probability of transition from state `i` to state `j`. `EMIS(k,sym)` is the probability that symbol `sym` is emitted from state `k`.

`PSTATES` is an array with the same length as `seq` and one row for each state in the model. The (i, j)th element of `PSTATES` gives the probability that the model is in state *i* at the *j*th step, given the sequence `seq`.

Note The function `hmmdecode` begins with the model in state 1 at step 0, prior to the first emission. `hmmdecode` computes the probabilities in `PSTATES` based on the fact that the model begins in state 1. See "How the Toolbox Generates Random Sequences" on page 12-7 for more information

---

`[PSTATES,logpseq] = hmmdecode(...)` returns `logpseq`, the logarithm of the probability of sequence `seq`, given transition matrix `TRANS` and emission matrix `EMIS`.

`[PSTATES,logpseq,FORWARD,BACKWARD,S] = hmmdecode(...)` returns the forward and backward probabilities of the sequence scaled by `S`. See "Reference" on page 14-346 for a reference that explains the forward and backward probabilities.

`hmmdecode(...,'Symbols',SYMBOLS)` specifies the symbols that are emitted. `SYMBOLS` can be a numeric array or a cell array of the names of the symbols. The default symbols are integers `1` through `N`, where `N` is the number of possible emissions.

See "Calculating Posterior State Probabilities" on page 12-12 for an example of using `hmmdecode`.

Examples

```
trans = [0.95,0.05;
         0.10,0.90];
emis = [1/6 1/6 1/6 1/6 1/6 1/6;
   1/10 1/10 1/10 1/10 1/10 1/2];

[seq,states] = hmmgenerate(100,trans,emis);
pStates = hmmdecode(seq,tr,e);
[seq,states] = hmmgenerate(100,trans,emis,...
   'Symbols',{'one','two','three','four','five','six'})
pStates = hmmdecode(seq,trans,emis,...
   'Symbols',{'one','two','three','four','five','six'});
```

Reference

[1] Durbin, R., S. Eddy, A. Krogh, and G. Mitchison, *Biological Sequence Analysis*, Cambridge University Press, 1998.

See Also

`hmmgenerate`, `hmmestimate`, `hmmviterbi`, `hmmtrain`

Purpose

Estimate parameters for hidden Markov model

Syntax

```
[TRANS,EMIS] = hmmestimate(seq,states)
hmmestimate(...,'Symbols',SYMBOLS)
hmmestimate(...,'Statenames',STATENAMES)
hmmestimate(...,'Pseudoemissions',PSEUDOE)
hmmestimate(...,'Pseudotransitions',PSEUDOTR)
```

Description

`[TRANS,EMIS] = hmmestimate(seq,states)` calculates the maximum likelihood estimate of the transition, `TRANS`, and emission, `EMIS`, probabilities of a hidden Markov model for sequence, `seq`, with known states, `states`.

`hmmestimate(...,'Symbols',SYMBOLS)` specifies the symbols that are emitted. `SYMBOLS` can be a numeric array or a cell array of the names of the symbols. The default symbols are integers 1 through N, where N is the number of possible emissions.

`hmmestimate(...,'Statenames',STATENAMES)` specifies the names of the states. `STATENAMES` can be a numeric array or a cell array of the names of the states. The default state names are 1 through M, where M is the number of states.

`hmmestimate(...,'Pseudoemissions',PSEUDOE)` specifies pseudocount emission values in the matrix `PSEUDO`. Use this argument to avoid zero probability estimates for emissions with very low probability that might not be represented in the sample sequence. `PSEUDOE` should be a matrix of size *m*-by-*n*, where *m* is the number of states in the hidden Markov model and *n* is the number of possible emissions. If the $i \rightarrow k$ emission does not occur in `seq`, you can set `PSEUDOE(i,k)` to be a positive number representing an estimate of the expected number of such emissions in the sequence `seq`.

`hmmestimate(...,'Pseudotransitions',PSEUDOTR)` specifies pseudocount transition values. You can use this argument to avoid zero probability estimates for transitions with very low probability that might not be represented in the sample sequence. `PSEUDOTR` should be a matrix of size *m*-by-*m*, where *m* is the number of states in the hidden Markov model. If the $i \rightarrow j$ transition does not occur in `states`, you can set PSEUDOTR(i,j) to be a positive number representing an estimate of the expected number of such transitions in the sequence states.

See "Using hmmestimate" on page 12-10 for an example of using hmmestimate.

Pseudotransitions and Pseudoemissions

If the probability of a specific transition or emission is very low, the transition might never occur in the sequence states, or the emission might never occur in the sequence seq. In either case, the algorithm returns a probability of 0 for the given transition or emission in TRANS or EMIS. You can compensate for the absence of transition with the 'Pseudotransitions' and 'Pseudoemissions' arguments. The simplest way to do this is to set the corresponding entry of PSEUDO or PSEUDOTR to 1. For example, if the transition $i \rightarrow j$ does not occur in states, set PSEUOTR(i,j) = 1. This forces TRANS(i,j) to be positive. If you have an estimate for the expected number of transitions $i \rightarrow j$ in a sequence of the same length as states, and the actual number of transitions $i \rightarrow j$ that occur in seq is substantially less than what you expect, you can set PSEUOTR(i,j) to the expected number. This increases the value of TRANS(i,j). For transitions that do occur in states with the frequency you expect, set the corresponding entry of PSEUDOTR to 0, which does not increase the corresponding entry of TRANS.

If you do not know the sequence of states, use hmmtrain to estimate the model parameters.

Examples:

```
trans = [0.95,0.05; 0.10,0.90];
emis = [1/6 1/6 1/6 1/6 1/6 1/6;
   1/10 1/10 1/10 1/10 1/10 1/2];

[seq,states] = hmmgenerate(1000,trans,emis);
[estimateTR,estimateE] = hmmestimate(seq,states);
```

See Also

hmmgenerate, hmmdecode, hmmviterbi, hmmtrain

Purpose

Generate random sequences from Markov model

Syntax

```
[seq,states] = hmmgenerate(len,TRANS,EMIS)
hmmgenerate(...,'Symbols',SYMBOLS)
hmmgenerate(...,'Statenames',STATENAMES)
```

Description

`[seq,states] = hmmgenerate(len,TRANS,EMIS)` takes a known Markov model, specified by transition probability matrix `TRANS` and emission probability matrix `EMIS`, and uses it to generate

- A random sequence `seq` of emission symbols
- A random sequence `states` of states

The length of both `seq` and `states` is `len`. `TRANS(i,j)` is the probability of transition from state `i` to state `j`. `EMIS(k,l)` is the probability that symbol `l` is emitted from state `k`.

Note The function `hmmgenerate` begins with the model in state 1 at step 0, prior to the first emission. The model then makes a transition to state i_1, with probability T_{1i_1}, and generates an emissiona_{k_1} with probability $E_{i_1 k_{1_1}}$. `hmmgenerate` returns i_1 as the first entry of `states`, anda_{k_1} as the first entry of `seq`. See "How the Toolbox Generates Random Sequences" on page 12-7 for more information

`hmmgenerate(...,'Symbols',SYMBOLS)` specifies the symbols that are emitted. `SYMBOLS` can be a numeric array or a cell array of the names of the symbols. The default symbols are integers `1` through `N`, where `N` is the number of possible emissions.

`hmmgenerate(...,'Statenames',STATENAMES)` specifies the names of the states. `STATENAMES` can be a numeric array or a cell array of the names of the states. The default state names are `1` through `M`, where `M` is the number of states.

Since the model always begins at state 1, whose transition probabilities are in the first row of `TRANS`, in the following example, the first entry of the output `states` is be 1 with probability 0.95 and 2 with probability 0.05.

See "Setting Up the Model and Generating Data" on page 12-8 for an example of using `hmmgenerate`.

Examples

```
   trans = [0.95,0.05;
            0.10,0.90];
   emis = [1/6 1/6 1/6 1/6 1/6 1/6;
      1/10 1/10 1/10 1/10 1/10 1/2];

  [seq,states] = hmmgenerate(100,trans,emis)
  [seq,states] = hmmgenerate(100,trans,emis,...
      'Symbols',{'one','two','three','four','five','six'},...
      'Statenames',{'fair';'loaded'})
```

See Also

`hmmviterbi`, `hmmdecode`, `hmmestimate`, `hmmtrain`

Purpose

Maximum likelihood estimate of model parameters for hidden Markov model

Syntax

```
[ESTTR,ESTEMIT] = hmmtrain(seq,TRGUESS,EMITGUESS)
hmmtrain(...,'Algorithm',algorithm)
hmmtrain(...,'Symbols',SYMBOLS)
hmmtrain(...,'Tolerance',tol)
hmmtrain(...,'Maxiterations',maxiter)
hmmtrain(...,'Verbose',true)
hmmtrain(...,'Pseudoemissions',PSEUDOE)
hmmtrain(...,'Pseudotransitions',PSEUDOTR)
```

Description

`[ESTTR,ESTEMIT] = hmmtrain(seq,TRGUESS,EMITGUESS)` estimates the transition and emission probabilities for a hidden Markov model using the Baum-Welch algorithm. `seq` can be a row vector containing a single sequence, a matrix with one row per sequence, or a cell array with each cell containing a sequence. `TRGUESS` and `EMITGUESS` are initial estimates of the transition and emission probability matrices. `TRGUESS(i,j)` is the estimated probability of transition from state `i` to state `j`. `EMITGUESS(i,k)` is the estimated probability that symbol `k` is emitted from state `i`.

`hmmtrain(...,'Algorithm',`*`algorithm`*`)` specifies the training algorithm. *`algorithm`* can be either `'BaumWelch'` or `'Viterbi'`. The default algorithm is `'BaumWelch'`.

`hmmtrain(...,'Symbols',SYMBOLS)` specifies the symbols that are emitted. `SYMBOLS` can be a numeric array or a cell array of the names of the symbols. The default symbols are integers `1` through `N`, where `N` is the number of possible emissions.

`hmmtrain(...,'Tolerance',tol)` specifies the tolerance used for testing convergence of the iterative estimation process. The default tolerance is `1e-4`.

`hmmtrain(...,'Maxiterations',maxiter)` specifies the maximum number of iterations for the estimation process. The default maximum is `100`.

`hmmtrain(...,'Verbose',true)` returns the status of the algorithm at each iteration.

`hmmtrain(...,'Pseudoemissions',PSEUDOE)` specifies pseudocount emission values for the Viterbi training algorithm. Use this argument to avoid zero probability estimates for emissions with very low probability that might not be represented in the sample sequence. `PSEUDOE` should be a matrix of size *m*-by-*n*, where *m* is the number of states in the hidden Markov model and *n* is the number of possible emissions. If the $i \rightarrow k$ emission does not occur in `seq`, you can set `PSEUDOE(i,k)` to be a positive number representing an estimate of the expected number of such emissions in the sequence `seq`.

`hmmtrain(...,'Pseudotransitions',PSEUDOTR)` specifies pseudocount transition values for the Viterbi training algorithm. Use this argument to avoid zero probability estimates for transitions with very low probability that might not be represented in the sample sequence. `PSEUDOTR` should be a matrix of size *m*-by-*m*, where *m* is the number of states in the hidden Markov model. If the $i \rightarrow j$ transition does not occur in `states`, you can set `PSEUDOTR(i,j)` to be a positive number representing an estimate of the expected number of such transitions in the sequence `states`.

See "Pseudotransitions and Pseudoemissions" on page 14-348 for more information.

If you know the states corresponding to the sequences, use `hmmestimate` to estimate the model parameters.

Tolerance

The input argument ’`tolerance'` controls how many steps the `hmmtrain` algorithm executes before the function returns an answer. The algorithm terminates when all of the following three quantities are less than the value that you specify for `tolerance`:

- The log likelihood that the input sequence `seq` is generated by the currently estimated values of the transition and emission matrices

- The change in the norm of the transition matrix, normalized by the size of the matrix
- The change in the norm of the emission matrix, normalized by the size of the matrix

The default value of `'tolerance'` is `.0001`. Increasing the tolerance decreases the number of steps the `hmmtrain` algorithm executes before it terminates.

`maxiterations`

The maximum number of iterations, `'maxiterations'`, controls the maximum number of steps the algorithm executes before it terminates. If the algorithm executes `maxiter` iterations before reaching the specified tolerance, the algorithm terminates and the function returns a warning. If this occurs, you can increase the value of `'maxiterations'` to make the algorithm reach the desired tolerance before terminating.

See "Using hmmtrain" on page 12-10 for an example of using `hmmtrain`.

Examples:

```
trans = [0.95,0.05;
      0.10,0.90];
emis = [1/6, 1/6, 1/6, 1/6, 1/6, 1/6;
   1/10, 1/10, 1/10, 1/10, 1/10, 1/2];

seq1 = hmmgenerate(100,trans,emis);
seq2 = hmmgenerate(200,trans,emis);
seqs = {seq1,seq2};
[estTR,estE] = hmmtrain(seqs,trans,emis);
```

See Also

`hmmgenerate`, `hmmdecode`, `hmmestimate`, `hmmviterbi`

hmmviterbi

Purpose Most probable state path for hidden Markov model sequence

Syntax

```
STATES = hmmvitervi(seq,TRANS,EMIS)
hmmviterbi(...,'Symbols',SYMBOLS)
hmmviterbi(...,'Statenames',STATENAMES)
```

Description

`STATES = hmmvitervi(seq,TRANS,EMIS)` given a sequence, `seq`, calculates the most likely path through the hidden Markov model specified by transition probability matrix, `TRANS`, and emission probability matrix `EMIS`. `TRANS(i,j)` is the probability of transition from state `i` to state `j`. `EMIS(i,k)` is the probability that symbol `k` is emitted from state `i`.

Note The function `hmmviterbi` begins with the model in state 1 at step 0, prior to the first emission. `hmmviterbi` computes the most likely path based on the fact that the model begins in state 1. See "How the Toolbox Generates Random Sequences" on page 12-7 for more information.

`hmmviterbi(...,'Symbols',SYMBOLS)` specifies the symbols that are emitted. `SYMBOLS` can be a numeric array or a cell array of the names of the symbols. The default symbols are integers `1` through `N`, where `N` is the number of possible emissions.

`hmmviterbi(...,'Statenames',STATENAMES)` specifies the names of the states. `STATENAMES` can be a numeric array or a cell array of the names of the states. The default state names are 1 through M, where M is the number of states.

See "Computing the Most Likely Sequence of States" on page 12-9 for an example of using `hmmviterbi`.

Examples

```
trans = [0.95,0.05;
         0.10,0.90];
emis = [1/6 1/6 1/6 1/6 1/6 1/6;
   1/10 1/10 1/10 1/10 1/10 1/2];
```

```
[seq,states] = hmmgenerate(100,trans,emis);
estimatedStates = hmmviterbi(seq,trans,emis);
[seq,states] = ...
   hmmgenerate(100,trans,emis,'Statenames',{'fair';'loaded'});
estimatesStates = ...
   hmmviterbi(seq,trans,eemis,'Statenames',{'fair';'loaded'});
```

See Also

`hmmgenerate`, `hmmdecode`, `hmmestimate`, `hmmtrain`

hougen

Purpose

Hougen-Watson model for reaction kinetics

Syntax

```
yhat = hougen(beta,x)
```

Description

`yhat = hougen(beta,x)` returns the predicted values of the reaction rate, `yhat`, as a function of the vector of parameters, `beta`, and the matrix of data, `X`. `beta` must have 5 elements and `X` must have three columns.

`hougen` is a utility function for `rsmdemo`.

The model form is:

$$\hat{y} = \frac{\beta_1 x_2 - x_3/\beta_5}{1 + \beta_2 x_1 + \beta_3 x_2 + \beta_4 x_3}$$

Reference

[1] Bates, D., and D. Watts, *Nonlinear Regression Analysis and Its Applications*. Wiley, 1988, pp. 271-272.

See Also

`rsmdemo`

Purpose

Hypergeometric cumulative distribution function

Syntax

```
hygecdf(X,M,K,N)
```

Description

`hygecdf(X,M,K,N)` computes the hypergeometric cdf at each of the values in `X` using the corresponding parameters in `M`, `K`, and `N`. Vector or matrix inputs for `X`, `M`, `K`, and `N` must all have the same size. A scalar input is expanded to a constant matrix with the same dimensions as the other inputs.

The hypergeometric cdf is

$$p = F(x|M,K,N) = \sum_{i=0}^{x} \frac{\binom{K}{i}\binom{M-K}{N-i}}{\binom{M}{N}}$$

The result, p, is the probability of drawing up to x of a possible K items in N drawings without replacement from a group of M objects.

Examples

Suppose you have a lot of 100 floppy disks and you know that 20 of them are defective. What is the probability of drawing zero to two defective floppies if you select 10 at random?

```
p = hygecdf(2,100,20,10)
p =
  0.6812
```

See Also

`hygepdf`, `hygeinv`

hygeinv

Purpose

Inverse of hypergeometric cumulative distribution function

Syntax

`hygeinv(P,M,K,N)`

Description

`hygeinv(P,M,K,N)` returns the smallest integer `X` such that the hypergeometric cdf evaluated at `X` equals or exceeds `P`. You can think of `P` as the probability of observing `X` defective items in `N` drawings without replacement from a group of `M` items where `K` are defective.

Examples

Suppose you are the Quality Assurance manager for a floppy disk manufacturer. The production line turns out floppy disks in batches of 1,000. You want to sample 50 disks from each batch to see if they have defects. You want to accept 99% of the batches if there are no more than 10 defective disks in the batch. What is the maximum number of defective disks should you allow in your sample of 50?

```
x = hygeinv(0.99,1000,10,50)
x =
    3
```

What is the median number of defective floppy disks in samples of 50 disks from batches with 10 defective disks?

```
x = hygeinv(0.50,1000,10,50)
x =
    0
```

See Also

`hygecdf`

Purpose

Hypergeometric probability density function

Syntax

```
Y = hygepdf(X,M,K,N)
```

Description

`Y = hygepdf(X,M,K,N)` computes the hypergeometric pdf at each of the values in `X` using the corresponding parameters in `M`, `K`, and `N`. `X`, `M`, `K`, and `N` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs.

The parameters in `M`, `K`, and `N` must all be positive integers, with `N` ≤ `M`. The values in `X` must be less than or equal to all the parameter values.

The hypergeometric pdf is

$$y = f(x|M, K, N) = \frac{\binom{K}{x}\binom{M-K}{N-x}}{\binom{M}{N}}$$

The result, y, is the probability of drawing exactly x of a possible K items in n drawings without replacement from a group of M objects.

Examples

Suppose you have a lot of 100 floppy disks and you know that 20 of them are defective. What is the probability of drawing 0 through 5 defective floppy disks if you select 10 at random?

```
p = hygepdf(0:5,100,20,10)
p =
  0.0951  0.2679  0.3182  0.2092  0.0841  0.0215
```

See Also

`hygecdf`, `hygernd`

Purpose

Random numbers from hypergeometric distribution

Syntax

```
R = hygernd(M,K,N)
R = hygernd(M,K,N,v)
R = hygernd(M,K,N,m,n)
```

Description

`R = hygernd(M,K,N)` generates random numbers from the hypergeometric distribution with parameters `M`, `K`, and `N`. `M`, `K`, and `N` can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of `R`. A scalar input for `M`, `K`, or `N` is expanded to a constant array with the same dimensions as the other inputs.

`R = hygernd(M,K,N,v)` generates random numbers from the hypergeometric distribution with parameters `M`, `K`, and `N`, where `v` is a row vector. If `v` is a 1-by-2 vector, `R` is a matrix with `v(1)` rows and `v(2)` columns. If `v` is 1-by-n, `R` is an n-dimensional array.

`R = hygernd(M,K,N,m,n)` generates random numbers from the hypergeometric distribution with parameters `M`, `K`, and `N`, where scalars `m` and `n` are the row and column dimensions of `R`.

Example

```
  numbers = hygernd(1000,40,50)
  numbers =
     1
```

See Also

`hygepdf`

Purpose

Mean and variance of hypergeometric distribution

Syntax

```
[MN,V] = hygestat(M,K,N)
```

Description

`[MN,V] = hygestat(M,K,N)` returns the mean of and variance for the hypergeometric distribution with parameters specified by `M`, `K`, and `N`. Vector or matrix inputs for `M`, `K`, and `N` must have the same size, which is also the size of `MN` and `V`. A scalar input for `M`, `K`, or `N` is expanded to a constant matrix with the same dimensions as the other inputs.

The mean of the hypergeometric distribution with parameters `M`, `K`, and `N` is `NK/M`, and the variance is

$$N\frac{K}{M}\frac{M-K}{M}\frac{M-N}{M-1}$$

Examples

The hypergeometric distribution approaches the binomial distribution, where $p = K / M$ as M goes to infinity.

```
[m,v] = hygestat(10.^(1:4),10.^(0:3),9)
m =
  0.9000  0.9000  0.9000  0.9000
v =
  0.0900  0.7445  0.8035  0.8094

[m,v] = binostat(9,0.1)
m =
  0.9000
v =
  0.8100
```

See Also

`hygepdf`

Purpose

Inverse cumulative distribution function for specified distribution

Syntax

```
Y = icdf(name,X,A)
Y = icdf(name,X,A,B)
Y = icdf(name,X,A,B,C)
```

Description

`Y = icdf(name,X,A)` computes the inverse cumulative distribution function for the one-parameter family of distributions specified by `name`. Parameter values for the distribution are given in `A`. The inverse cumulative distribution function is evaluated at the values in `X` and its values are returned in `Y`.

If `X` and `A` are arrays, they must be the same size. If `X` is a scalar, it is expanded to a constant matrix the same size as `A`. If `A` is a scalar, it is expanded to a constant matrix the same size as `X`.

`Y` is the common size of `X` and `A` after any necessary scalar expansion.

`Y = icdf(name,X,A,B)` computes the inverse cumulative distribution function for two-parameter families of distributions, where parameter values are given in `A` and `B`.

If `X`, `A`, and `B` are arrays, they must be the same size. If `X` is a scalar, it is expanded to a constant matrix the same size as `A` and `B`. If either `A` or `B` are scalars, they are expanded to constant matrices the same size as `X`.

`Y` is the common size of `X`, `A`, and `B` after any necessary scalar expansion.

`Y = icdf(name,X,A,B,C)` computes the inverse cumulative distribution function for three-parameter families of distributions, where parameter values are given in `A`, `B`, and `C`.

If `X`, `A`, `B`, and `C` are arrays, they must be the same size. If `X` is a scalar, it is expanded to a constant matrix the same size as `A`, `B`, and `C`. If any of `A`, `B` or `C` are scalars, they are expanded to constant matrices the same size as `X`.

`Y` is the common size of `X`, `A`, `B` and `C` after any necessary scalar expansion.

Acceptable strings for `name` are:

- `'beta'` (Beta distribution)
- `'bino'` (Binomial distribution)
- `'chi2'` (Chi-square distribution)
- `'exp'` (Exponential distribution)
- `'ev'` (Extreme value distribution)
- `'f'` (*F* distribution)
- `'gam'` (Gamma distribution)
- `'gev'` (Generalized extreme value distribution)
- `'gp'` (Generalized Pareto distribution)
- `'geo'` (Geometric distribution)
- `'hyge'` (Hypergeometric distribution)
- `'logn'` (Lognormal distribution)
- `'nbin'` (Negative binomial distribution)
- `'ncf'` (Noncentral *F* distribution)
- `'nct'` (Noncentral *t*distribution)
- `'ncx2'` (Noncentral chi-square distribution)
- `'norm'` (Normal distribution)
- `'poiss'` (Poisson distribution)
- `'rayl'` (Rayleigh distribution)
- `'t'` (*t* distribution)
- `'unif'` (Uniform distribution)
- `'unid'` (Discrete uniform distribution)
- `'wbl'` (Weibull distribution)

Examples

```
x = icdf('Normal',0.1:0.2:0.9,0,1)
x =
  -1.2816  -0.5244      0  0.5244  1.2816

x = icdf('Poisson',0.1:0.2:0.9,1:5)
x =
   0   1   3   5   8
```

See Also cdf, mle, pdf, random

Purpose

Inverse cumulative distribution function for piecewise distribution

Syntax

```
X = icdf(obj,P)
```

Description

`X = icdf(obj,P)` returns an array `X` of values of the inverse cumulative distribution function for the piecewise distribution object `obj`, evaluated at the values in the array `P`.

Example

Fit Pareto tails to a *t* distribution at cumulative probabilities 0.1 and 0.9:

```
t = trnd(3,100,1);
obj = paretotails(t,0.1,0.9);
[p,q] = boundary(obj)
p =
    0.1000
    0.9000
q =
   -1.7766
    1.8432

icdf(obj,p)
ans =
   -1.7766
    1.8432
```

See Also

`paretotails`, `cdf (piecewisedistribution)`

Purpose

Inconsistency coefficient of cluster tree

Syntax

```
Y = inconsistent(Z)
Y = inconsistent(Z,d)
```

Description

`Y = inconsistent(Z)` computes the inconsistency coefficient for each link of the hierarchical cluster tree `Z`, where `Z` is an (m-1)-by-3 matrix generated by the `linkage` function. The inconsistency coefficient characterizes each link in a cluster tree by comparing its length with the average length of other links at the same level of the hierarchy. The higher the value of this coefficient, the less similar the objects connected by the link.

`Y = inconsistent(Z,d)` computes the inconsistency coefficient for each link in the hierarchical cluster tree `Z` to depth `d`, where `d` is an integer denoting the number of levels of the cluster tree that are included in the calculation. By default, `d=2`.

The output, `Y`, is an (m-1)-by-4 matrix formatted as follows.

Column	Description
1	Mean of the lengths of all the links included in the calculation.
2	Standard deviation of all the links included in the calculation.
3	Number of links included in the calculation.
4	Inconsistency coefficient.

For each link, k, the inconsistency coefficient is calculated as:

$$Y(k,4) = (z(k,3)-Y(k,1))/Y(k,2)$$

For leaf nodes, nodes that have no further nodes under them, the inconsistency coefficient is set to 0.

Example

```
rand('state',12);
X = rand(10,2);
Y = pdist(X);
Z = linkage(Y,'centroid');
W = inconsistent(Z,3)
W =
  0.1313       0  1.0000       0
  0.1386       0  1.0000       0
  0.1727  0.0482  2.0000  0.7071
  0.2391       0  1.0000       0
  0.2242  0.0955  3.0000  1.0788
  0.2357  0.1027  3.0000  0.9831
  0.3222  0.1131  3.0000  0.9772
  0.3376  0.1485  6.0000  1.4883
  0.4920  0.1341  4.0000  1.1031
```

References

[1] Jain, A., and R. Dubes, *Algorithms for Clustering Data*, Prentice-Hall, Englewood Cliffs, NJ, 1988.

[2] Zahn, C.T., "Graph-theoretical methods for detecting and describing Gestalt clusters," *IEEE Transactions on Computers*, C 20, pp. 68-86, 1971.

See Also

`cluster`, `cophenet`, `clusterdata`, `dendrogram`, `linkage`, `pdist`, `squareform`

interactionplot

Purpose

Interaction plot for grouped data

Syntax

```
interactionplot(Y,GROUP)
interactionplot(Y,GROUP,'varnames',VARNAMES)
[h,AX,bigax] = interactionplot(...)
```

Description

`interactionplot(Y,GROUP)` displays the two-factor interaction plot for the group means of matrix `Y` with groups defined by entries in the cell array `GROUP`. `Y` is a numeric matrix or vector. If `Y` is a matrix, the rows represent different observations and the columns represent replications of each observation. If `Y` is a vector, the rows give the means of each entry in the cell array `GROUP`. Each cell of `GROUP` must contain a grouping variable that can be a categorical variable, numeric vector, character matrix, or a single-column cell array of strings. (See "Grouped Data" on page 2-41.) `GROUP` can also be a matrix whose columns represent different grouping variables. Each grouping variable must have the same number of rows as `Y`. The number of grouping variables must be greater than 1.

The interaction plot is a matrix plot, with the number of rows and columns both equal to the number of grouping variables. The grouping variable names are printed on the diagonal of the plot matrix. The plot at off-diagonal position (*i*,*j*) is the interaction of the two variables whose names are given at row diagonal (*i*,*i*) and column diagonal (*j*,*j*), respectively.

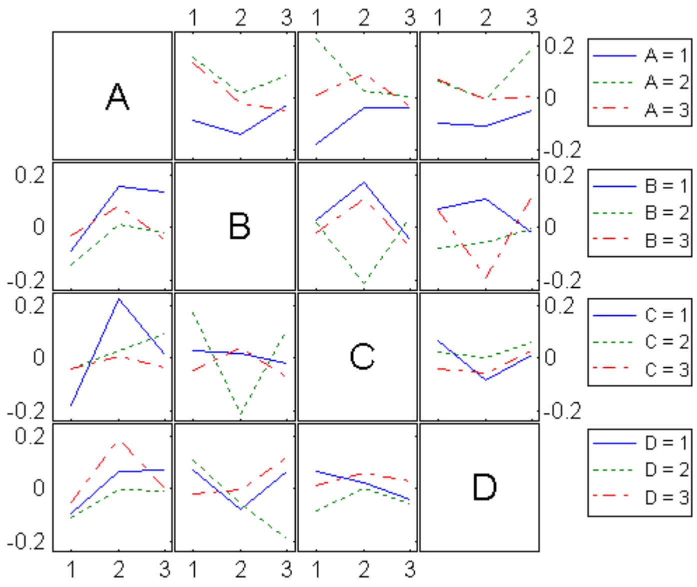
`interactionplot(Y,GROUP,'varnames',VARNAMES)` displays the interaction plot with user-specified grouping variable names `VARNAMES`. `VARNAMES` is a character matrix or a cell array of strings, one per grouping variable. Default names are `'X1'`, `'X2'`, ...

`[h,AX,bigax] = interactionplot(...)` returns a handle `h` to the figure window, a matrix `AX` of handles to the subplot axes, and a handle `bigax` to the big (invisible) axes framing the subplots.

Example

Display interaction plots for data with four 3-level factors named `'A'`, `'B'`,`'C'`, and `'D'`:

```
y = randn(1000,1); % response
group = ceil(3*rand(1000,4)); % four 3-level factors
interactionplot(y,group,'varnames',{'A','B','C','D'})
```



See Also

`maineffectsplot`, `multivarichart`

invpred

Purpose Inverse prediction for simple linear regression

Syntax

```
X0 = invpred(X,Y,Y0)
[X0,DXLO,DXUP] = invpred(X,Y,Y0)
[X0,DXLO,DXUP] = invpred(X,Y,Y0,name1,val1,name2,val2,...)
```

Description

`X0 = invpred(X,Y,Y0)` accepts vectors `X` and `Y` of the same length, fits a simple regression, and returns the estimated value `X0` for which the height of the line is equal to `Y0`. The output, `X0`, has the same size as `Y0`, and `Y0` can be an array of any size.

`[X0,DXLO,DXUP] = invpred(X,Y,Y0)` also computes 95% inverse prediction intervals. `DXLO` and `DXUP` define intervals with lower bound `X0 DXLO` and upper bound `X0+DXUP`. Both `DXLO` and `DXUP` have the same size as `Y0`.

The intervals are not simultaneous and are not necessarily finite. Some intervals may extend from a finite value to `-Inf` or `+Inf`, and some may extend over the entire real line.

`[X0,DXLO,DXUP] = invpred(X,Y,Y0,`*`name1`*`,`*`val1`*`,`*`name2`*`,`*`val2`*`,...)` specifies optional argument name/value pairs chosen from the following list. Argument names are case insensitive and partial matches are allowed.

Name	Value
`'alpha'`	A value between 0 and 1 specifying a confidence level of `100*(1-alpha)`%. Default is `alpha=0.05` for 95% confidence.
`'predopt'`	Either `'observation'`, the default value to compute the intervals for `X0` at which a new observation could equal `Y0`, or `'curve'` to compute intervals for the `X0` value at which the curve is equal to `Y0`.

Example

```
x = 4*rand(25,1);
y = 10 + 5*x + randn(size(x));
scatter(x,y)
x0 = invpred(x,y,20)
```

See Also

`polyfit`, `polytool`, `polyconf`

iqr

Purpose

Interquartile range of sample

Syntax

```
y = iqr(X)
iqr(X,dim)
```

Description

`y = iqr(X)` returns the interquartile range of the values in X. For vector input, `y` is the difference between the 75th and the 25th percentiles of the sample in X. For matrix input, `y` is a row vector containing the interquartile range of each column of `X`. For N-dimensional arrays, `iqr` operates along the first nonsingleton dimension of `X`.

`iqr(X,dim)` calculates the interquartile range along the dimension `dim` of `X`.

Remarks

The IQR is a robust estimate of the spread of the data, since changes in the upper and lower 25% of the data do not affect it. If there are outliers in the data, then the IQR is more representative than the standard deviation as an estimate of the spread of the body of the data. The IQR is less efficient than the standard deviation as an estimate of the spread when the data is all from the normal distribution.

Multiply the IQR by 0.7413 to estimate σ (the second parameter of the normal distribution.)

Examples

This Monte Carlo simulation shows the relative efficiency of the IQR to the sample standard deviation for normal data.

```
x = normrnd(0,1,100,100);
s = std(x);
s_IQR = 0.7413*iqr(x);
efficiency = (norm(s-1)./norm(s_IQR-1)).^2
efficiency =
  0.3297
```

See Also

`std`, `mad`, `range`

Purpose

Test tree node for branch

Syntax

```
ib = isbranch(t)
ib = isbranch(t,nodes)
```

Description

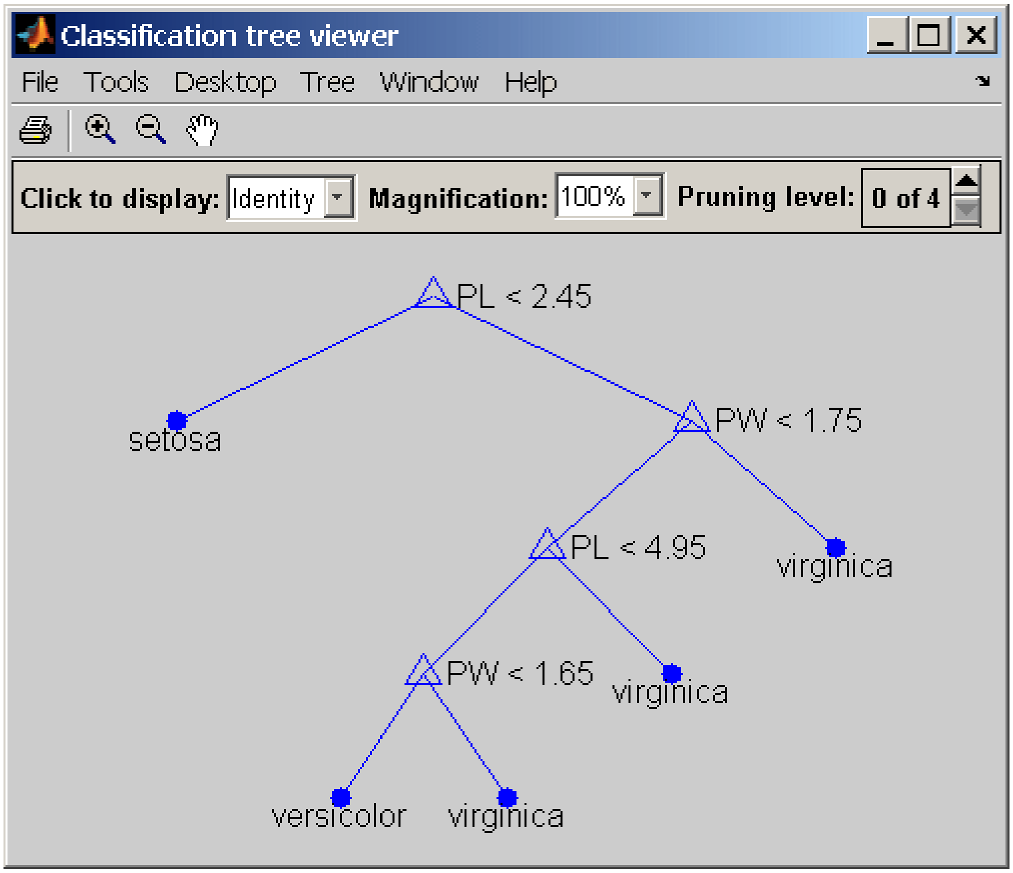
`ib = isbranch(t)` returns an *n*-element logical vector `ib` that is `true` for each branch node and `false` for each leaf node.

`ib = isbranch(t,nodes)` takes a vector `nodes` of node numbers and returns a vector of logical values for the specified nodes.

Example

Create a classification tree for Fisher's iris data:

```
load fisheriris;

t = classregtree(meas,species,'names',{'SL' 'SW' 'PL' 'PW'})
t =
Decision tree for classification
1  if PL<2.45 then node 2 else node 3
2  class = setosa
3  if PW<1.75 then node 4 else node 5
4  if PL<4.95 then node 6 else node 7
5  class = virginica
6  if PW<1.65 then node 8 else node 9
7  class = virginica
8  class = versicolor
9  class = virginica

view(t)
```



```
ib = isbranch(t)
ib =
     1
     0
     1
     1
     0
     1
     0
     0
     0
```

Reference

[1] Breiman, L., et al., *Classification and Regression Trees*, Chapman & Hall, Boca Raton, 1993.

See Also

`classregtree`, `numnodes`, `cutvar`

islevel

Purpose

Test for categorical array levels

Syntax

```
I = islevel(levels,A)
```

Description

`I = islevel(levels,A)` returns a logical array `I` the same size as the string, cell array of strings, or two-dimensional character matrix `levels`. `I` is `true` (`1`) where the corresponding element of `levels` is the label of a level in the categorical array `A`, even if the level contains no elements. `I` is `false` (`0`) otherwise.

Example

Display age levels in the data in `hospitl.mat`, before and after dropping occupied levels:

```
load hospital
edges = 0:10:100;
labels = strcat(num2str((0:10:90)','%d'),{'s'});
disp(labels')
 '0s' '10s' '20s' '30s' '40s' '50s' '60s' '70s' '80s' '90s'

AgeGroup = ordinal(hospital.Age,labels,[],edges);
I = islevel(labels,AgeGroup);
disp(I')
 1  1  1  1  1  1  1  1  1  1

AgeGroup = droplevels(AgeGroup);
I = islevel(labels,AgeGroup);
disp(I')
 0  0  1  1  1  1  0  0  0  0
```

See Also

`ismember`, `isundefined`

Purpose

Test for categorical array membership

Syntax

```
I = ismember(A,levels)
[I,IDX] = ismember(A,levels)
```

Description

`I = ismember(A,levels)` returns a logical array `I` the same size as the categorical array `A`. `I` is `true` (`1`) where the corresponding element of `A` is one of the levels specified by the labels in the categorical array, cell array of strings, or two-dimensional character array `levels`. `I` is false (`0`) otherwise.

`[I,IDX] = ismember(A,levels)` also returns an array of indices `IDX` containing the highest absolute index in `levels` for each element in `A` whose level is a member of `levels`, and `0` if there is no such index.

Examples

Example 1

For nominal data:

```
load hospital
sex = hospital.Sex; % Nominal
smokers = hospital.Smoker; % Logical
I = ismember(sex(smokers),'Female');
I(1:5)
ans =
     0
     1
     0
     0
     0
```

The use of `ismember` above is equivalent to:

```
I = (sex(smokers) == 'Female');
```

Example 2

For ordinal data:

```
load hospital
```

```
edges = 0:10:100;
labels = strcat(num2str((0:10:90)','%d'),{'s'});
AgeGroup = ordinal(hospital.Age,labels,[],edges);
I = ismember(AgeGroup(1:5),{'20s','30s'})
I =
     1
     0
     1
     0
     0
```

See Also

`islevel`, `isundefined`

Purpose

Test for undefined elements of categorical array

Syntax

```
I = isundefined(A)
```

Description

`I = isundefined(A)` returns a logical array `I` the same size as the categorical array `A`. `I` is `true` (`1`) where the corresponding element of `A` is not assigned to any level. `I` is `false` (`0`) where the corresponding element of `A` is assigned to a level.

Example

Create and display undefined levels in an ordinal array:

```
A = ordinal([1 2 3 2 1],{'lo','med','hi'})
A =
     lo      med      hi      med      lo

A = droplevels(A,{'med','hi'})
Warning: OLDLEVELS contains categorical levels that
were present in A, caused some array elements to
have undefined levels.
A =
     lo  <undefined>  <undefined>  <undefined>  lo

I = isundefined(A)
I =
     0     1     1     1     0
```

See Also

`islevel`, `ismember`

iwishrnd

Purpose

Random numbers from inverse Wishart distribution

Syntax

```
W = iwishrnd(sigma,df)
W = iwishrnd(sigma,df,DI)
[W,DI] = iwishrnd(sigma,df)
```

Description

`W = iwishrnd(sigma,df)` generates a random matrix `W` from the inverse Wishart distribution with parameters `sigma` and `df`. The inverse of `W` has the Wishart distribution with covariance matrix `inv(sigma)` and with `df` degrees of freedom. `sigma` can be a vector, a matrix, or a multidimensional array.

`W = iwishrnd(sigma,df,DI)` expects `DI` to be the Cholesky factor of the inverse of `sigma`. `DI` is an array of the same size as `sigma`. If you call `iwishrnd` multiple times using the same value of `sigma`, it is more efficient to supply `DI` instead of computing it each time.

`[W,DI] = iwishrnd(sigma,df)` returns `DI` so you can use it as input in future calls to `iwishrnd`.

Note that different sources use different parameterizations for the inverse Wishart distribution. This function defines the parameter `sigma` so that the mean of the output matrix is `sigma`/(`df`−`k`−1), where `k` is the number of rows and columns in `sigma`.

See Also

`wishrnd`

Purpose

Jackknife statistics

Syntax

`jackstat = jackknife(jackfun,...)`

Description

`jackstat = jackknife(jackfun,...)` draws jackknife data samples, computes statistics on each sample using the function `jackfun`, and returns the results in the matrix `jackstat`. `jackfun` is a function handle specified with `@`. Each row of `jackstat` contains the results of applying `jackfun` to one jackknife sample. If `jackfun` returns a matrix or array, this output is converted to a row vector for storage in `jackstat`.

The third and later input arguments to `jackknife` are scalars, column vectors, or matrices that are used to create inputs to `jackfun`. `jackknife` creates each jackknife sample by sampling with replacement from the rows of the nonscalar data arguments (these must have the same number of rows). Scalar data are passed to `jackfun` unchanged.

Example

Estimate the bias of the MLE variance estimator of random samples taken from the vector `y` using `jackknife`. The bias has a known formula in this problem, so you can compare the `jackknife` value to this formula.

```
y = exprnd(5,100,1);
m = jackknife(@var,y,1);
n = length(y);

bias = var(y,1)-var(y,0) % Bias formula
bias =
   -0.2069

jbias = (n-1)*(mean(m)-var(y,1)) % Jackknife estimate
jbias =
   -0.2069
```

See Also

`bootstrp`, `random`, `randsample`, `hist`, `ksdensity`

Purpose

Jarque-Bera test

Syntax

```
h = jbtest(x)
h = jbtest(x,alpha)
[h,p] = jbtest(...)
[h,p,jbstat] = jbtest(...)
[h,p,jbstat,critval] = jbtest(...)
[h,p,...] = jbtest(x,alpha,mctol)
```

Description

`h = jbtest(x)` performs a Jarque-Bera test of the null hypothesis that the sample in vector `x` comes from a normal distribution with unknown mean and variance, against the alternative that it does not come from a normal distribution. The test is specifically designed for alternatives in the Pearson system of distributions. The test returns the logical value `h = 1` if it rejects the null hypothesis at the 5% significance level, and `h = 0` if it cannot. The test treats `NaN` values in `x` as missing values, and ignores them.

The Jarque-Bera test is a two-sided goodness-of-fit test suitable when a fully-specified null distribution is unknown and its parameters must be estimated. The test statistic is

$$JB = \frac{n}{6}\left(s^2 + \frac{(k-3)^2}{4}\right)$$

where n is the sample size, s is the sample skewness, and k is the sample kurtosis. For large sample sizes, the test statistic has a chi-square distribution with two degrees of freedom.

Jarque-Bera tests often use the chi-square distribution to estimate critical values for large samples, deferring to the Lilliefors test (see `lillietest`) for small samples. `jbtest`, by contrast, uses a table of critical values computed using Monte-Carlo simulation for sample sizes less than 2000 and significance levels between 0.001 and 0.50. Critical values for a test are computed by interpolating into the table, using the analytic chi-square approximation only when extrapolating for larger sample sizes.

`h = jbtest(x,alpha)` performs the test at significance level `alpha`. `alpha` is a scalar in the range [0.001, 0.50]. To perform the test at a significance level outside of this range, use the `mctol` input argument.

`[h,p] = jbtest(...)` returns the *p*-value `p`, computed using inverse interpolation into the table of critical values. Small values of `p` cast doubt on the validity of the null hypothesis. `jbtest` warns when `p` is not found within the tabulated range of [0.001, 0.50], and returns either the smallest or largest tabulated value. In this case, you can use the `mctol` input argument to compute a more accurate *p*-value.

`[h,p,jbstat] = jbtest(...)` returns the test statistic `jbstat`.

`[h,p,jbstat,critval] = jbtest(...)` returns the critical value `critval` for the test. When `jbstat` > `critval`, the null hypothesis is rejected at significance level `alpha`.

`[h,p,...]  = jbtest(x,alpha,mctol)` computes a Monte-Carlo approximation for `p` directly, rather than interpolating into the table of pre-computed values. This is useful when `alpha` or `p` lie outside the range of the table. `jbtest` chooses the number of Monte Carlo replications, `mcreps`, large enough to make the Monte Carlo standard error for `p`, `sqrt(p*(1-p)/mcreps)`, less than `mctol`.

Example

Use `jbtest` to determine if car mileage, in miles per gallon (MPG), follows a normal distribution across different makes of cars:

```
load carbig
[h,p] = jbtest(MPG)
h =
     1
p =
    0.0022
```

The *p*-value is below the default significance level of 5%, and the test rejects the null hypothesis that the distribution is normal.

With a log transformation, the distribution becomes closer to normal, but the *p*-value is still well below 5%:

```
[h,p] = jbtest(log(MPG))
h =
     1
p =
    0.0078
```

Decreasing the significance level makes it harder to reject the null hypothesis:

```
[h,p] = jbtest(log(MPG),0.0075)
h =
     0
p =
    0.0078
```

References

[1] Jarque, C.M., and A.K. Bera, A test for normality of observations and regression residuals, *International Statistical Review*, Vol. 55, No. 2, 1987, pp. 1-10. This paper proposed the original test.

[2] Deb, P., and M. Sefton, The distribution of a Lagrange multiplier test of normality, *Economics Letters*, Vol. 51, 1996, pp. 123-130. This paper proposed a Monte Carlo simulation for determining the distribution of the test statistic. The results of this function are based on an independent Monte Carlo simulation, not the results in this paper.

Purpose

Random numbers from Johnson system of distributions

Syntax

```
r = johnsrnd(quantiles,m,n)
r = johnsrnd(quantiles)
[r,type] = johnsrnd(...)
[r,type,coefs] = johnsrnd(...)
```

Description

`r = johnsrnd(quantiles,m,n)` returns an `m`-by-`n` matrix of random numbers drawn from the distribution in the Johnson system that satisfies the quantile specification given by `quantiles`. `quantiles` is a four-element vector of quantiles for the desired distribution that correspond to the standard normal quantiles [-1.5 -0.5 0.5 1.5]. In other words, you specify a distribution from which to draw random values by designating quantiles that correspond to the cumulative probabilities [0.067 0.309 0.691 0.933]. `quantiles` may also be a `2`-by-`4` matrix whose first row contains four standard normal quantiles, and whose second row contains the corresponding quantiles of the desired distribution. The standard normal quantiles must be spaced evenly.

Note Because `r` is a random sample, its sample quantiles typically differ somewhat from the specified distribution quantiles.

`r = johnsrnd(quantiles)` returns a scalar value.

`r = johnsrnd(quantiles,m,n,...)` or `r = johnsrnd(quantiles,[m,n,...])` returns an `m`-by-`n`-by-... array.

`[r,type] = johnsrnd(...)` returns the type of the specified distribution within the Johnson system. `type` is `'SN'`, `'SL'`, `'SB'`, or `'SU'`. Set `m` and `n` to zero to identify the distribution type without generating any random values.

The four distribution types in the Johnson system correspond to the following transformations of a normal random variate:

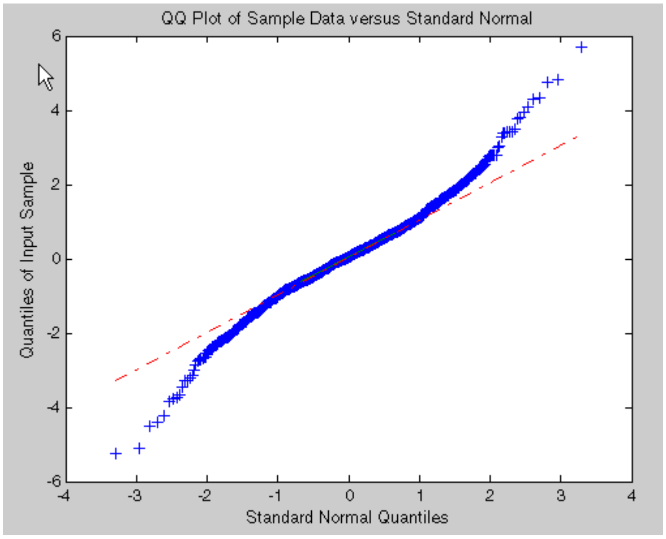
`'SN'`	Identity transformation (normal distribution)
`'SL'`	Exponential transformation (lognormal distribution)
`'SB'`	Logistic transformation (bounded)
`'SU'`	Hyperbolic sine transformation (unbounded)

`[r,type,coefs] = johnsrnd(...)` returns coefficients of the transformation that defines the distribution. `coefs` is `[gamma, eta, epsilon, lambda]`. If `z` is a standard normal random variable and `h` is one of the transformations defined above, `r = lambda*h((z-gamma)/eta)+epsilon` is a random variate from the distribution type corresponding to `h`.

Example

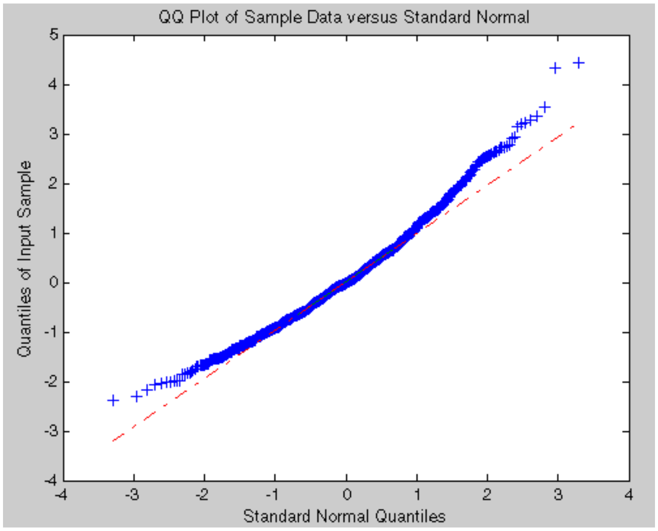
Generate random values with longer tails than a standard normal.

```
r = johnsrnd([-1.7 -.5 .5 1.7],1000,1);
qqplot(r);
```



Generate random values skewed to the right.

```
r = johnsrnd([-1.3 -.5 .5 1.7],1000,1);
qqplot(r);
```



Generate random values that match some sample data well in the right-hand tail.

```
load carbig;
qnorm = [.5 1 1.5 2];
q = quantile(Acceleration, normcdf(qnorm));
r = johnsrnd([qnorm;q],1000,1);
[q;quantile(r,normcdf(qnorm))]
ans =
   16.7000   18.2086   19.5376   21.7263
   16.8190   18.2474   19.4492   22.4156
```

Determine the distribution type and the coefficients.

```
[r,type,coefs] = johnsrnd([qnorm;q],0)
r =
    []
type =
    SU
coefs =
    1.0920    0.5829   18.4382    1.4494
```

See Also random, pearsrnd

Purpose

Merge observations from two dataset arrays

Syntax

```
C = join(A,B)
C = join(A,B,key)
C = join(A,B,param1,val1,param2,val2,...)
[C,idx] = join(...)
```

Description

`C = join(A,B)` creates a dataset array `C` by merging observations from the two dataset arrays `A` and `B`. `join` performs the merge by first finding *key variables*, that is, a pair of dataset variables, one in `A` and one in `B`, that share the same name. The key from `B` must contain unique values, and must contain all the values that are present in the key from `A`. `join` then uses these key variables to define a many-to-one correspondence between observations in `A` and those in `B`. `join` uses this correspondence to replicate the observations in `B` and combine them with the observations in `A` to create `C`.

`C` contains one observation for each observation in `A`. Variables in `C` include all of the variables from `A`, as well as one variable corresponding to each variable in `B` (except for the key from `B`).

`C = join(A,B,key)` performs the merge using the variable specified by `key` as the key variable in both `A` and `B`. `key` is a positive integer, a variable name, a cell array containing a variable name, or a logical vector with one `true` entry.

`C = join(A,B,`*`param1`*`,`*`val1`*`,`*`param2`*`,`*`val2`*`,...)` specifies optional parameter name/value pairs to control how the dataset variables in `A` and `B` are used in the merge. Parameters are:

- `'Key'` — Specifies the variable to use as a key in both `A` and `B`.
- `'LeftKey'` — Specifies the variable to use as a key in `A`.
- `'RightKey'` — Specifies the variable to use as a key in `B`.

You may provide either the `'Key'` parameter, or both the `'LeftKey'` and `'RightKey'` parameters. The value for these parameters is a positive integer, a variable name, a cell array containing a variable name, or a logical vector with one true entry.

- `'LeftVars'` — Specifies the variables from `A` to include in `C`. By default, `join` includes all variables from `A`.
- `'RightVars'` — Specifies the variables from `B` to include in `C`. By default, `join` includes all variables from `B` except the key variable.

The value for these parameters is a positive integer, a vector of positive integers, a variable name, a cell array containing one or more variable names, or a logical vector.

`[C,idx] = join(...)` returns an index vector `idx`, where the observations in `C` are constructed by horizontally concatenating `A(:,leftvars)` and `B(idx,rightvars)`.

Example

Create a dataset array from Fisher's iris data:

```
load fisheriris
NumObs = size(meas,1);
ObsNames = strcat({'Obs'},num2str((1:NumObs)','%d'));
iris = dataset({nominal(species),'species'},...
               {meas,'SL','SW','PL','PW'},...
               'obsnames',ObsNames);
```

Create a separate dataset array with the diploid chromosome counts for each species of iris:

```
snames = nominal({'setosa';'versicolor';'virginica'});
CC = dataset({snames,'species'},{[38;108;70],'cc'})
CC =
    species       cc
    setosa         38
    versicolor    108
    virginica      70
```

Broadcast the data in `CC` to the rows of `iris` using the key variable `species` in each dataset:

```
iris2 = join(iris,CC);
iris2([1 2 51 52 101 102],:)
ans =
           species       SL     SW     PL     PW     cc
 Obs1      setosa        5.1    3.5    1.4    0.2     38
 Obs2      setosa        4.9      3    1.4    0.2     38
 Obs51     versicolor      7    3.2    4.7    1.4    108
 Obs52     versicolor    6.4    3.2    4.5    1.5    108
 Obs101    virginica     6.3    3.3      6    2.5     70
 Obs102    virginica     5.8    2.7    5.1    1.9     70
```

See Also

`sortrows (dataset)`

Purpose

K-means clustering

Syntax

```
IDX = kmeans(X,k)
[IDX,C] = kmeans(X,k)
[IDX,C,sumd] = kmeans(X,k)
[IDX,C,sumd,D] = kmeans(X,k)
[...] = kmeans(...,param1,val1,param2,val2,...)
```

Description

`IDX = kmeans(X,k)` partitions the points in the `n`-by-`p` data matrix `X` into `k` clusters. This iterative partitioning minimizes the sum, over all clusters, of the within-cluster sums of point-to-cluster-centroid distances. Rows of `X` correspond to points, columns correspond to variables. `kmeans` returns an `n`-by-`1` vector `IDX` containing the cluster indices of each point. By default, `kmeans` uses squared Euclidean distances.

`[IDX,C] = kmeans(X,k)` returns the `k` cluster centroid locations in the `k`-by-`p` matrix `C`.

`[IDX,C,sumd] = kmeans(X,k)` returns the within-cluster sums of point-to-centroid distances in the `1`-by-`k` vector `sumd`.

`[IDX,C,sumd,D] = kmeans(X,k)` returns distances from each point to every centroid in the `n`-by-`k` matrix `D`.

`[...] = kmeans(...,`*`param1`*`,`*`val1`*`,`*`param2`*`,`*`val2`*`,...)` enables you to specify optional parameter/value pairs to control the iterative algorithm used by `kmeans`. Valid parameter strings are the following.

Parameter	Value	
`'distance'`	Distance measure, in `p`-dimensional space. `kmeans` minimizes with respect to this parameter. `kmeans` computes centroid clusters differently for the different supported distance measures:	
	`'sqEuclidean'`	Squared Euclidean distance (default). Each centroid is the mean of the points in that cluster.

Parameter	Value	
	’cityblock'	Sum of absolute differences, i.e., the L1 distance. Each centroid is the component-wise median of the points in that cluster.
	'cosine'	One minus the cosine of the included angle between points (treated as vectors). Each centroid is the mean of the points in that cluster, after normalizing those points to unit Euclidean length.
	'correlation'	One minus the sample correlation between points (treated as sequences of values). Each centroid is the component-wise mean of the points in that cluster, after centering and normalizing those points to zero mean and unit standard deviation.
	'Hamming'	Percentage of bits that differ (only suitable for binary data). Each centroid is the component-wise median of points in that cluster.

Parameter	Value	
`'start'`	Method used to choose the initial cluster centroid positions, sometimes known as *seeds*. Valid starting values are:	
	`'sample'`	Select `k` observations from `X` at random (default).
	`'uniform'`	Select `k` points uniformly at random from the range of `X`. Not valid with Hamming distance.
	`'cluster'`	Perform a preliminary clustering phase on a random 10% subsample of `X`. This preliminary phase is itself initialized using 'sample'.
	Matrix	`k`-by-`p` matrix of centroid starting locations. In this case, you can pass in `[]` for `k`, and `kmeans` infers `k` from the first dimension of the matrix. You can also supply a 3-dimensional array, implying a value for the `'replicates'` parameter from the array's third dimension.
`'replicates'`	Number of times to repeat the clustering, each with a new set of initial cluster centroid positions. `kmeans` returns the solution with the lowest value for `sumd`. You can supply `'replicates'` implicitly by supplying a 3-dimensional array as the value for the `'start'` parameter.	
`'maxiter'`	Maximum number of iterations. Default is 100.	

Parameter	Value	
'emptyaction'	Action to take if a cluster loses all its member observations. Can be one of:	
	'error'	Treat an empty cluster as an error. (default)
	'drop'	Remove any clusters that become empty. kmeans sets the corresponding return values in C and D to NaN.
	'singleton'	Create a new cluster consisting of the one point furthest from its centroid.
'display'	Controls display of output.	
	'off'	Display no output.
	'iter'	Display information about each iteration during minimization, including the iteration number, the optimization phase (see "Algorithm" on page 14-395), the number of points moved, and the total sum of distances.
	'final'	Display a summary of each replication.
	'notify'	Display only warning and error messages. (default)

Algorithm

kmeans uses a two-phase iterative algorithm to minimize the sum of point-to-centroid distances, summed over all k clusters:

- The first phase uses what the literature often describes as *batch updates*, where each iteration consists of reassigning points to their nearest cluster centroid, all at once, followed by recalculation of cluster centroids. You can think of this phase as providing a fast but potentially only approximate solution as a starting point for the second phase.
- The second phase uses what the literature often describes as *online updates*, where points are individually reassigned if doing so will reduce the sum of distances, and cluster centroids are recomputed after each reassignment. Each iteration during this second phase consists of one pass though all the points.

  `kmeans` can converge to a local optimum, in this case, a partition of points in which moving any single point to a different cluster increases the total sum of distances. This problem can only be solved by a clever (or lucky, or exhaustive) choice of starting points.

See Also

`clusterdata`, `linkage`, `silhouette`

References

[1] Seber, G. A. F., *Multivariate Observations*, Wiley, 1984.

[2] Spath, H., *Cluster Dissection and Analysis: Theory, FORTRAN Programs, Examples*, translated by J. Goldschmidt, Halsted Press, 1985, 226 pp.

Purpose

Kruskal-Wallis nonparametric one-way analysis of variance

Syntax

```
p = kruskalwallis(X)
p = kruskalwallis(X,group)
p = kruskalwallis(X,group,displayopt)
[p,table] = kruskalwallis(...)
[p,table,stats] = kruskalwallis(...)
```

Description

`p = kruskalwallis(X)` performs a Kruskal-Wallis test to compare samples from two or more groups. Each column of the *m*-by-*n* matrix `X` represents an independent sample containing *m* mutually independent observations. The function compares the medians of the samples in `X`, and returns the p-value for the null hypothesis that all samples are drawn from the same population (or equivalently, from different populations with the same distribution). Note that the Kruskal-Wallis test is a nonparametric version of the classical one-way ANOVA, and an extension of the Wilcoxon rank sum test to more than two groups.

If the *p*-value is near zero, this casts doubt on the null hypothesis and suggests that at least one sample median is significantly different from the others. The choice of a critical *p*-value to determine whether the result is judged statistically significant is left to the researcher. It is common to declare a result significant if the *p*-value is less than 0.05 or 0.01.

The `kruskalwallis` function displays two figures. The first figure is a standard ANOVA table, calculated using the ranks of the data rather than their numeric values. Ranks are found by ordering the data from smallest to largest across all groups, and taking the numeric index of this ordering. The rank for a tied observation is equal to the average rank of all observations tied with it. For example, the following table shows the ranks for a small sample.

X value	1.4	2.7	1.6	1.6	3.3	0.9	1.1
Rank	3	6	4.5	4.5	7	1	2

The entries in the ANOVA table are the usual sums of squares, degrees of freedom, and other quantities calculated on the ranks. The usual *F* statistic is replaced by a chi-square statistic. The p-value measures the significance of the chi-square statistic.

The second figure displays box plots of each column of `X` (not the ranks of `X`).

`p = kruskalwallis(X,group)` uses the values in `group` (a character array or cell array) as labels for the box plot of the samples in `X`, when `X` is a matrix. Each row of `group` contains the label for the data in the corresponding column of `X`, so `group` must have length equal to the number of columns in `X`. (See "Grouped Data" on page 2-41.)

When `X` is a vector, `kruskalwallis` performs a Kruskal-Wallis test on the samples contained in `X`, as indexed by input `group` (a categorical variable, vector, character array, or cell array). Each element in `group` identifies the group (i.e., sample) to which the corresponding element in vector `X` belongs, so `group` must have the same length as `X`. The labels contained in `group` are also used to annotate the box plot.

It is not necessary to label samples sequentially (`1`, `2`, `3`, ...). For example, if `X` contains measurements taken at three different temperatures, -27°, 65°, and 110°, you could use these numbers as the sample labels in `group`. If a row of `group` contains an empty cell or empty string, that row and the corresponding observation in `X` are disregarded. `NaN`s in either input are similarly ignored.

`p = kruskalwallis(X,group,`*`displayopt`*`)` enables the table and box plot displays when *`displayopt`* is `'on'` (default) and suppresses the displays when *`displayopt`* is `'off'`.

`[p,table] = kruskalwallis(...)` returns the ANOVA table (including column and row labels) in cell array `table`.

`[p,table,stats] = kruskalwallis(...)` returns a `stats` structure that you can use to perform a follow-up multiple comparison test. The `kruskalwallis` test evaluates the hypothesis that all samples come from populations that have the same median, against the alternative that the medians are not all the same. Sometimes it is preferable to perform a test to determine which pairs are significantly different, and which are not. You can use the `multcompare` function to perform such tests by supplying the `stats` structure as input.

Assumptions

The Kruskal-Wallis test makes the following assumptions about the data in `X`:

- All samples come from populations having the same continuous distribution, apart from possibly different locations due to group effects.
- All observations are mutually independent.

The classical one-way ANOVA test replaces the first assumption with the stronger assumption that the populations have normal distributions.

Example

This example compares the material strength study used with the `anova1` function, to see if the nonparametric Kruskal-Wallis procedure leads to the same conclusion. The example studies the strength of beams made from three alloys:

```
strength = [82 86 79 83 84 85 86 87 74 82 ...
            78 75 76 77 79 79 77 78 82 79];

alloy = {'st','st','st','st','st','st','st','st',...
         'al1','al1','al1','al1','al1','al1',...
         'al2','al2','al2','al2','al2','al2'};
```

This example uses both classical and Kruskal-Wallis ANOVA, omitting displays:

```
anova1(strength,alloy,'off')
ans =
 1.5264e-004

kruskalwallis(strength,alloy,'off')
ans =
    0.0018
```

Both tests find that the three alloys are significantly different, though the result is less significant according to the Kruskal-Wallis test. It is typical that when a data set has a reasonable fit to the normal distribution, the classical ANOVA test is more sensitive to differences between groups.

To understand when a nonparametric test may be more appropriate, let's see how the tests behave when the distribution is not normal. You can simulate this by replacing one of the values by an extreme value (an outlier).

```
strength(20)=120;
anova1(strength,alloy,'off')
ans =
  0.2501

kruskalwallis(strength,alloy,'off')
ans =
  0.0060
```

Now the classical ANOVA test does not find a significant difference, but the nonparametric procedure does. This illustrates one of the properties of nonparametric procedures - they are often not severely affected by changes in a small portion of the data.

Reference

[1] Gibbons, J. D., *Nonparametric Statistical Inference*, 2nd edition, M. Dekker, 1985.

[2] Hollander, M., and D. A. Wolfe, *Nonparametric Statistical Methods,* Wiley, 1973.

See Also

`anova1`, `boxplot`, `friedman`, `multcompare`, `ranksum`

Purpose

Compute density estimate using kernel-smoothing method

Syntax

```
[f,xi] = ksdensity(x)
f = ksdensity(x,xi)
ksdensity(...)
ksdensity(ax,...)
[f,xi,u] = ksdensity(...)
[...] = ksdensity(...,param1,val1,param2,val2,...)
```

Description

`[f,xi] = ksdensity(x)` computes a probability density estimate of the sample in the vector `x`. `f` is the vector of density values evaluated at the points in `xi`. The estimate is based on a normal kernel function, using a window parameter (`'width'`) that is a function of the number of points in `x`. The density is evaluated at 100 equally spaced points that cover the range of the data in `x`.

`f = ksdensity(x,xi)` specifies the vector `xi` of values, where the density estimate is to be evaluated.

`ksdensity(...)` without output arguments produces a plot of the results.

`ksdensity(ax,...)` plots into axes `ax` instead of `gca`.

`[f,xi,u] = ksdensity(...)` also returns the width of the kernel-smoothing window.

`[...] = ksdensity(...,`*`param1`*`,`*`val1`*`,`*`param2`*`,`*`val2`*`,...)` specifies parameter/value pairs to control the density estimation. Valid parameter strings and their possible values are as follows:

`'censoring'`	A logical vector of the same length as `x`, indicating which entries are censoring times. Default is no censoring.
`'kernel'`	The type of kernel smoother to use. Choose the value as `'normal'` (default), `'box'`, `'triangle'`, or `'epanechnikov'`. Alternatively, you can specify some other function, as a function handle or as a string, e.g., `@normpdf` or `'normpdf'`. The function must take a single argument that is an array of distances between data values and places where the density is evaluated. It must return an array of the same size containing corresponding values of the kernel function.
`'npoints'`	The number of equally spaced points in `xi`. Default is `100`.
`'support'`	• `'unbounded'` allows the density to extend over the whole real line (default). • `'positive'` restricts the density to positive values. • A two-element vector gives finite lower and upper bounds for the support of the density.
`'weights'`	Vector of the same length as `x`, assigning weight to each `x` value.
`'width'`	The bandwidth of the kernel-smoothing window. The default is optimal for estimating normal densities, but you may want to choose a smaller value to reveal features such as multiple modes.

`'width'`	The bandwidth of the kernel-smoothing window. The default is optimal for estimating normal densities, but you may want to choose a smaller value to reveal features such as multiple modes.
`'function'`	The function type to estimate, chosen from among `'pdf'`, `'cdf'`, `'icdf'`, `'survivor'`, or `'cumhazard'` for the density, cumulative probability, inverse cumulative probability, survivor, or cumulative hazard functions, respectively.

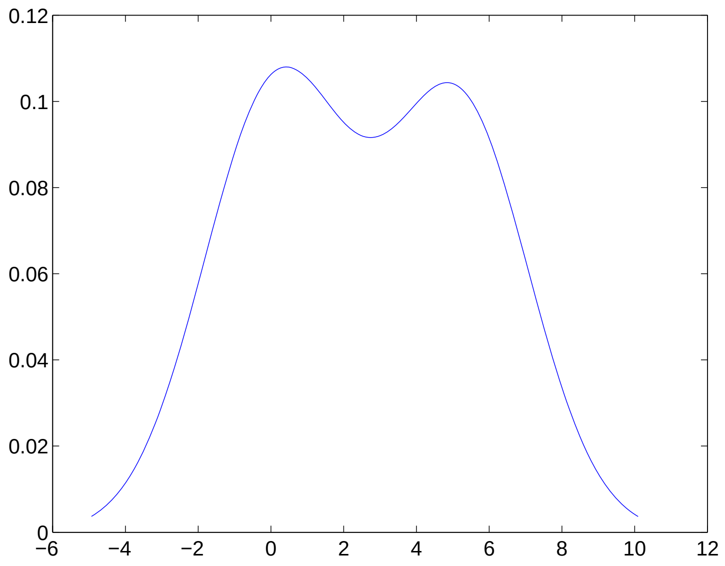
In place of the kernel functions listed above, you can specify another kernel function by using `@` (such as `@normpdf`) or quotes (such as `'normpdf'`). The function must take a single argument that is an array of distances between data values and places where the density is evaluated, and return an array of the same size containing corresponding values of the kernel function. When the `'function'` parameter value is `'pdf'`, this kernel function should return density values; otherwise, it should return cumulative probability values. Specifying a custom kernel when the `'function'` parameter value is `'icdf'` is an error.

If the `'support'` parameter is `'positive'`, `ksdensity` transforms `x` using a log function, estimates the density of the transformed values, and transforms back to the original scale. If `'support'` is a vector `[L U]`, `ksdensity` uses the transformation `log((X-L)/(U-X))`. The `'width'` parameter and `u` outputs are on the scale of the transformed values.

Examples

This example generates a mixture of two normal distributions and plots the estimated density.

```
x = [randn(30,1); 5+randn(30,1)];
[f,xi] = ksdensity(x);
plot(xi,f);
```



References

[1] Bowman, A. W., and A. Azzalini, *Applied Smoothing Techniques for Data Analysis*, Oxford University Press, 1997.

See Also

`hist`, `@` (function handle)

Purpose

One-sample Kolmogorov-Smirnov test

Syntax

```
H = kstest(X)
H = kstest(X,cdf)
H = kstest(X,cdf,alpha)
H = kstest(X,cdf,alpha,tail)
[H,P,KSSTAT,CV] = kstest(X,cdf,alpha,tail)
```

Description

`H = kstest(X)` performs a Kolmogorov-Smirnov test to compare the values in the data vector `X` with a standard normal distribution (that is, a normal distribution having mean 0 and variance 1). The null hypothesis for the Kolmogorov-Smirnov test is that `X` has a standard normal distribution. The alternative hypothesis that `X` does not have that distribution. The result `H` is `1` if you can reject the hypothesis that `X` has a standard normal distribution, or `0` if you cannot reject that hypothesis. You reject the hypothesis if the test is significant at the 5% level.

For each potential value x, the Kolmogorov-Smirnov test compares the proportion of values less than x with the expected number predicted by the standard normal distribution. The `kstest` function uses the maximum difference over all x values is its test statistic. Mathematically, this can be written as

$$\max(|F(x) - G(x)|)$$

where $F(x)$ is the proportion of `X` values less than or equal to x and $G(x)$ is the standard normal cumulative distribution function evaluated at x.

`H = kstest(X,cdf)` compares the distribution of `X` to the hypothesized continuous distribution defined by the two-column matrix `cdf`. Column one contains a set of possible x values, and column two contains the corresponding hypothesized cumulative distribution function values $G(x)$. If possible, you should define `cdf` so that column one contains the values in `X`. If there are values in `X` not found in column one of `cdf`, `kstest` will approximate $G(X)$ by interpolation. All values in `X` must lie in the interval between the smallest and largest values in the first column of `cdf`. If the second argument is empty (`cdf = []`), `kstest` uses the standard normal distribution as if there were no second argument.

The Kolmogorov-Smirnov test requires that `cdf` be predetermined. It is not accurate if `cdf` is estimated from the data. To test `X` against a normal distribution without specifying the parameters, use `lillietest` instead.

`H = kstest(X,cdf,alpha)` specifies the significance level `alpha` for the test. The default is `0.05`.

`H = kstest(X,cdf,alpha,tail)` specifies the type of test in the string `tail`. `tail` can have one of the following values:

- `'unequal'`
- `'larger'`
- `'smaller'`

The tests specified by these values are described in "Tests Specified by tail" on page 14-406.

`[H,P,KSSTAT,CV] = kstest(X,cdf,alpha,tail)` also returns the observed p-value `P`, the observed Kolmogorov-Smirnov statistic `KSSTAT`, and the cutoff value `CV` for determining if `KSSTAT` is significant. If the return value of `CV` is `NaN`, then `kstest` determined the significance calculating a p-value according to an asymptotic formula rather than by comparing `KSSTAT` to a critical value.

Tests Specified by tail

Let $S(x)$ be the empirical c.d.f. estimated from the sample vector `X`, let $F(x)$ be the corresponding true (but unknown) population c.d.f., and let CDF be the known input c.d.f. specified under the null hypothesis. The one-sample Kolmogorov-Smirnov test tests the null hypothesis that $F(x) = \text{CDF}$ for all x against the alternative specified by one of the following possible values of `tail`:

tail	Alternative Hypothesis	Test Statistic
'unequal'	$F(x)$ does not equal CDF (two-sided test)	max|$S(x)$ – CDF|
'larger'	$F(x)$ > CDF (one-sided test)	max[$S(x)$ – CDF]
'smaller'	$F(x)$ < CDF (one-sided test)	max[$S(x)$ – CDF]

Examples

Example 1

Let's generate some evenly spaced numbers and perform a Kolmogorov-Smirnov test to see how well they fit to a standard normal distribution:

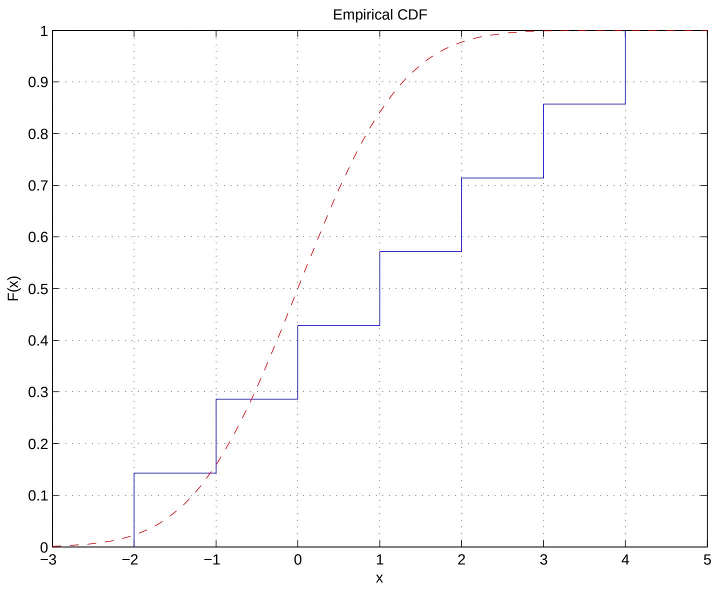
```
x = -2:1:4
x =
  -2  -1   0   1   2   3   4

[h,p,k,c] = kstest(x,[],0.05,0)
h =
   0
p =
   0.13632
k =
   0.41277
c =
   0.48342
```

You cannot reject the null hypothesis that the values come from a standard normal distribution. Although intuitively it seems that these evenly-spaced integers could not follow a normal distribution, this example illustrates the difficulty in testing normality in small samples.

To understand the test, it is helpful to generate an empirical cumulative distribution plot and overlay the theoretical normal distribution.

```
xx = -3:.1:5;
cdfplot(x)
hold on
```

```
plot(xx,normcdf(xx),'r-')
```



The Kolmogorov-Smirnov test statistic is the maximum difference between these curves. It appears that this maximum of 0.41277 occurs as the data approaches `x = 1.0` from below. You can see that the empirical curve has the value 3/7 here, and you can easily verify that the difference between the curves is 0.41277.

```
normcdf(1) - 3/7
ans =
   0.41277
```

You can also perform a one-sided test. Setting *`tail`* `= -1` indicates that the alternative is $F < G$, so the test statistic counts only points where this inequality is true.

```
[h,p,k] = kstest(x,[],.05,-1)
```

```
h =
   0
p =
   0.068181
k =
   0.41277
```

The test statistic is the same as before because in fact $F < G$ at `x = 1.0`. However, the p-value is smaller for the one-sided test. If you carry out the other one-sided test, you see that the test statistic changes, and is the difference between the two curves near `x = -1.0`.

```
[h,p,k] = kstest(x,[],0.05,1)
h =
   0
p =
   0.77533
k =
   0.12706

2/7-normcdf(-1)
ans =
   0.12706
```

Example 2

Now let's generate random numbers from a Weibull distribution, and test against that Weibull distribution and an exponential distribution.

```
x = wblrnd(1,2,100,1);

kstest(x,[x wblcdf(x,1,2)])
ans =
   0

kstest(x,[x expcdf(x,1)])
ans =
   1
```

See Also `kstest2`, `lillietest`

Purpose

Two-sample Kolmogorov-Smirnov test

Syntax

```
H = kstest2(X1,X2)
H = kstest2(X1,X2,alpha,tail)
[H,P] = kstest2(...)
[H,P,ksstat] = KSTEST2(...)
```

Description

`H = kstest2(X1,X2)` performs a two-sample Kolmogorov-Smirnov test to compare the distributions of values in the two data vectors `X1` and `X2` of length `n1` and `n2`, respectively, representing random samples from some underlying distribution(s). The null hypothesis for this test is that `X1` and `X2` are drawn from the same continuous distribution. The alternative hypothesis is that they are drawn from different continuous distributions. The result `H` is `1` if you can reject the hypothesis that the distributions are the same, or `0` if you cannot reject that hypothesis. You reject the hypothesis if the test is significant at the 5% level.

For each potential value x, the Kolmogorov-Smirnov test compares the proportion of `X1` values less than x with proportion of `X2` values less than x. The `kstest2` function uses the maximum difference over all x values is its test statistic. Mathematically, this can be written as

$$\max(|F1(x) - F2(x)|)$$

where $F1(x)$ is the proportion of `X1` values less than or equal to x and $F2(x)$ is the proportion of `X2` values less than or equal to x. Missing observations, indicated by `NaN`s are ignored.

`H = kstest2(X1,X2,alpha)` performs the test at the `(100*alpha)`% significance level.

The decision to reject the null hypothesis occurs when the significance level, `alpha`, equals or exceeds the P-value.

`H = kstest2(X1,X2,alpha,tail)` accepts a string `tail` that specifies the type of test. `tail` can have one of the following values:

- `'unequal'`

- `'larger'`
- `'smaller'`

The tests specified by these values are described in "Tests Specified by tail" on page 14-412

`[H,P] = kstest2(...)` also returns the asymptotic p-value `P`. The asymptotic `p`-value becomes very accurate for large sample sizes, and is believed to be reasonably accurate for sample sizes `n1` and `n2` such that `(n1*n2)/(n1 + n2) >= 4`.

`[H,P,ksstat] = KSTEST2(...)` also returns the Kolmogorov-Smirnov test statistic `KSSTAT` defined above for the test type indicated by *`tail`*.

Tests Specified by tail

Let $S1(x)$ and $S2(x)$ be the empirical distribution functions from the sample vectors `X1` and `X2`, respectively, and $F1(x)$ and $F2(x)$ be the corresponding true (but unknown) population CDFs. The two-sample Kolmogorov-Smirnov test tests the null hypothesis that F$F1(x)$ = $F2(x)$, for all x, against the alternative hypothesis specified by *`tail`*, as described in the following table.

tail	Alternative Hypothesis	Test Statistic
`'unequal'`	$F1(x)$ does not equal $F2(x)$ (two-sided test)	$\max\lvert S1(x) - S2(x)\rvert$
`'larger'`	$F1(x) > F2(x)$ (one-sided test)	$\max[S1(x) - S2(x)]$
`'smaller'`	$F1(x) < F2(x)$ (one-sided test)	$\max[S2(x) - S1(x))]$

Examples

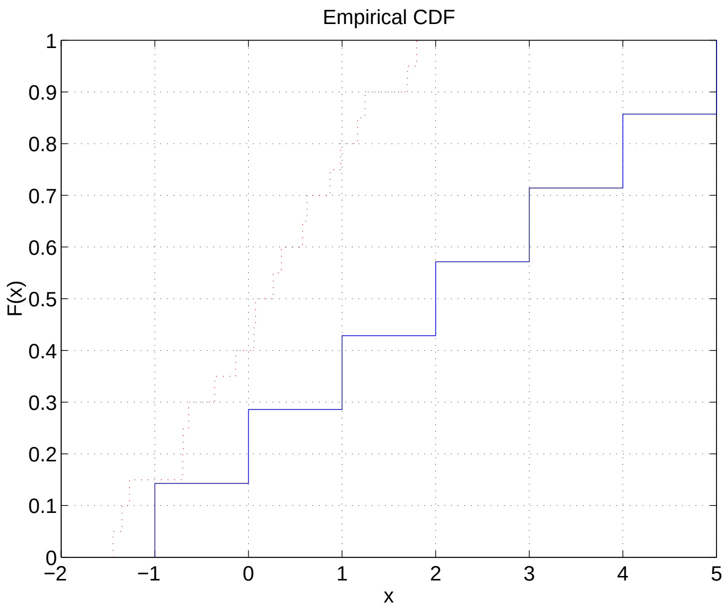
The following commands compare the distributions of a small evenly-spaced sample and a larger normal sample:

```
x = -1:1:5
y = randn(20,1);
[h,p,k] = kstest2(x,y)
h =
     1
```

```
p =
  0.0403
k =
  0.5714
```

The difference between their distributions is significant at the 5% level (p = 4%). To visualize the difference, you can overlay plots of the two empirical cumulative distribution functions. The Kolmogorov-Smirnov statistic is the maximum difference between these functions. After changing the color and line style of one of the two curves, you can see that the maximum difference appears to be near `x = 1.9`. You can also verify that the difference equals the `k` value that `kstest2` reports:

```
cdfplot(x)
hold on
cdfplot(y)
h = findobj(gca,'type','line');
set(h(1),'linestyle',':','color','r')

1 - 3/7
ans =
   0.5714
```



See Also

`kstest`, `lillietest`

Purpose

Sample kurtosis

Syntax

```
k = kurtosis(X)
k = kurtosis(X,flag)
k = kurtosis(X,flag,dim)
```

Description

`k = kurtosis(X)` returns the sample kurtosis of `X`. For vectors, `kurtosis(x)` is the kurtosis of the elements in the vector `x`. For matrices `kurtosis(X)` returns the sample kurtosis for each column of `X`. For N-dimensional arrays, `kurtosis` operates along the first nonsingleton dimension of `X`.

`k = kurtosis(X,flag)` specifies whether to correct for bias (`flag` is `0`) or not (`flag` is `1`, the default). When `X` represents a sample from a population, the kurtosis of `X` is biased, that is, it will tend to differ from the population kurtosis by a systematic amount that depends on the size of the sample. You can set `flag` to `0` to correct for this systematic bias.

`k = kurtosis(X,flag,dim)` takes the kurtosis along dimension `dim` of `X`.

`kurtosis` treats `NaN`s as missing values and removes them.

Remarks

Kurtosis is a measure of how outlier-prone a distribution is. The kurtosis of the normal distribution is 3. Distributions that are more outlier-prone than the normal distribution have kurtosis greater than 3; distributions that are less outlier-prone have kurtosis less than 3.

The kurtosis of a distribution is defined as

$$k = \frac{E(x-\mu)^4}{\sigma^4}$$

where μ is the mean of x, σ is the standard deviation of x, and $E(t)$ represents the expected value of the quantity t.

Note Some definitions of kurtosis subtract 3 from the computed value, so that the normal distribution has kurtosis of 0. The `kurtosis` function does not use this convention.

Example

```
X = randn([5 4])
X =
  1.1650  1.6961  -1.4462  -0.3600
  0.6268  0.0591  -0.7012  -0.1356
  0.0751  1.7971  1.2460  -1.3493
  0.3516  0.2641  -0.6390  -1.2704
  -0.6965  0.8717  0.5774  0.9846

k = kurtosis(X)
k =
  2.1658  1.2967  1.6378  1.9589
```

See Also

`mean`, `moment`, `skewness`, `std`, `var`

Purpose

Element counts by level for categorical array

Syntax

```
C = levelcounts(A)
C = levelcounts(A,dim)
```

Description

`C = levelcounts(A)` for a categorical vector `A` counts the number of elements in `A` equal to each of the possible levels in `A`. The output is a vector `C` containing those counts, and has as many elements as `A` has levels. For matrix `A`, `C` is a matrix of column counts. For *N*-dimensional arrays, `levelcounts` operates along the first nonsingleton dimension.

`C = levelcounts(A,dim)` operates along the dimension `dim`.

Example

Count the number of patients in each age group in the data in `hospital.mat`:

```
load hospital
edges = 0:10:100;
labels = strcat(num2str((0:10:90)','%d'),{'s'});
disp(labels')
 '0s' '10s' '20s' '30s' '40s' '50s' '60s' '70s' '80s' '90s'

AgeGroup = ordinal(hospital.Age,labels,[],edges);
I = islevel(labels,AgeGroup);
disp(I')
 1   1   1   1   1   1   1   1   1   1
c = levelcounts(AgeGroup);
disp(c')
 0   0 15  41  42   2   0   0   0   0

AgeGroup = droplevels(AgeGroup);
I = islevel(labels,AgeGroup);
disp(I')
 0   0   1   1   1   1   0   0   0   0
c = levelcounts(AgeGroup);
disp(c')
 15  41  42   2
```

See Also `islevel`, `ismember`, `summary (categorical)`

Purpose

Leverage values for regression

Syntax

```
h = leverage(data)
h = leverage(data,model)
```

Description

`h = leverage(data)` finds the leverage of each row (point) in the matrix `data` for a linear additive regression model.

`h = leverage(data,`*`model`*`)` finds the leverage on a regression, using a specified model type, where *`model`* can be one of these strings:

- `'linear'` - includes constant and linear terms
- `'interaction'` - includes constant, linear, and cross product terms
- `'quadratic'` - includes interactions and squared terms
- `'purequadratic'` - includes constant, linear, and squared terms

Leverage is a measure of the influence of a given observation on a regression due to its location in the space of the inputs.

Example

One rule of thumb is to compare the leverage to $2p/n$ where n is the number of observations and p is the number of parameters in the model. For the Hald data set this value is 0.7692.

```
load hald
h = max(leverage(ingredients,'linear'))
h =
  0.7004
```

Since 0.7004 < 0.7692, there are no high leverage points using this rule.

Algorithm

```
[Q,R] = qr(x2fx(data,'model'));

leverage = (sum(Q'.*Q'))'
```

Reference

[1] Goodall, C. R., “Computation Using the QR Decomposition,” *Handbook in Statistics,* Volume 9. Elsevier/North-Holland, 1993.

See Also `regstats`

Purpose

Generate latin hypercube sample

Syntax

```
X = lhsdesign(n,p)
X = lhsdesign(...,'smooth','off')
X = lhsdesign(...,'criterion',criterion)
X = lhsdesign(...,'iterations',k)
```

Description

`X = lhsdesign(n,p)` generates a latin hypercube sample `X` containing `n` values on each of `p` variables. For each column, the `n` values are randomly distributed with one from each interval `(0,1/n)`, `(1/n,2/n)`, ..., `(1-1/n,1)`, and they are randomly permuted.

`X = lhsdesign(...,'smooth','off')` produces points at the midpoints of the above intervals: `0.5/n`, `1.5/n`, ..., `1-0.5/n`. The default is `'on'`.

`X = lhsdesign(...,'criterion',`*`criterion`*`)` iteratively generates latin hypercube samples to find the best one according to the criterion *`criterion`*, which can be one of the following strings:

`'none'`	No iteration
`'maximin'`	Maximize minimum distance between points
`'correlation'`	Reduce correlation

`X = lhsdesign(...,'iterations',k)` iterates up to `k` times in an attempt to improve the design according to the specified criterion. Default is `K = 5`.

Latin hypercube designs are useful when you need a sample that is random but that is guaranteed to be relatively uniformly distributed over each dimension.

See Also

`lhsnorm`, `unifrnd`

lhsnorm

Purpose

Generate latin hypercube sample with normal distribution

Syntax

```
X = lhsnorm(mu,sigma,n)
X = lhsnorm(mu,sigma,n,flag)
```

Description

`X = lhsnorm(mu,sigma,n)` generates a latin hypercube sample `X` of size `n` from the multivariate normal distribution with mean vector `mu` and covariance matrix `sigma`. `X` is similar to a random sample from the multivariate normal distribution, but the marginal distribution of each column is adjusted so that its sample marginal distribution is close to its theoretical normal distribution.

`X = lhsnorm(mu,sigma,n,`*`flag`*`)` controls the amount of smoothing in the sample. If *`flag`* is `'off'`, each column has points equally spaced on the probability scale. In other words, each column is a permutation of the values G`(0.5/n), `G`(1.5/n), ..., `G`(1-0.5/n)` where G is the inverse normal cumulative distribution for that column's marginal distribution. If *`flag`* is `'on'` (the default), each column has points uniformly distributed on the probability scale. For example, in place of `0.5/n` you use a value having a uniform distribution on the interval `(0/n,1/n)`.

See Also

`lhsdesign`, `mvnrnd`

Purpose

Lilliefors test

Syntax

```
h = lillietest(x)
h = lillietest(x,alpha)
h = lillietest(x,alpha,distr)
[h,p] = lillietest(...)
[h,p,kstat] = lillietest(...)
[h,p,kstat,critval] = lillietest(...)
[h,p,...] = lillietest(x,alpha,distr,mctol)
```

Description

`h = lillietest(x)` performs a Lillilifors test of the default null hypothesis that the sample in vector `x` comes from a distribution in the normal family, against the alternative that it does not come from a normal distribution. The test returns the logical value `h = 1` if it rejects the null hypothesis at the 5% significance level, and `h = 0` if it cannot. The test treats `NaN` values in `x` as missing values, and ignores them.

The Lilliefors test is a 2-sided goodness-of-fit test suitable when a fully-specified null distribution is unknown and its parameters must be estimated. This is in contrast to the one-sample Kolmogorov-Smirnov test (see `kstest`), which requires that the null distribution be completely specified. The Lilliefors test statistic is the same as for the Kolmogorov-Smirnov test:

$$KS = \max_x \left| SCDF(x) - CDF(x) \right|$$

where *SCDF* is the empirical cdf estimated from the sample and *CDF* is the normal cdf with mean and standard deviation equal to the mean and standard deviation of the sample.

`lillietest` uses a table of critical values computed using Monte-Carlo simulation for sample sizes less than 1000 and significance levels between 0.001 and 0.50. The table is larger and more accurate than the table introduced by Lilliefors. Critical values for a test are computed by interpolating into the table, using an analytic approximation when extrapolating for larger sample sizes.

h = lillietest(x,alpha) performs the test at significance level alpha. alpha is a scalar in the range [0.001, 0.50]. To perform the test at a significance level outside of this range, use the mctol input argument.

h = lillietest(x,alpha,distr) performs the test of the null hypothesis that x came from the location-scale family of distributions specified by distr. Acceptable values for distr are 'norm' (normal, the default), 'exp' (exponential), and 'ev' (extreme value). The Lilliefors test can not be used when the null hypothesis is not a location-scale family of distributions.

[h,p] = lillietest(...) returns the *p*-value p, computed using inverse interpolation into the table of critical values. Small values of p cast doubt on the validity of the null hypothesis. lillietest warns when p is not found within the tabulated range of [0.001, 0.50], and returns either the smallest or largest tabulated value. In this case, you can use the mctol input argument to compute a more accurate *p*-value.

[h,p,kstat] = lillietest(...) returns the test statistic kstat.

[h,p,kstat,critval] = lillietest(...) returns the critical value critval for the test. When kstat > critval, the null hypothesis is rejected at significance level alpha

[h,p,...] = lillietest(x,alpha,distr,mctol) computes a Monte-Carlo approximation for p directly, rather than interpolating into the table of pre-computed values. This is useful when alpha or p lie outside the range of the table. lillietest chooses the number of Monte Carlo replications, mcreps, large enough to make the Monte Carlo standard error for p, sqrt(p*(1-p)/mcreps), less than mctol.

Example

Use lillietest to determine if car mileage, in miles per gallon (MPG), follows a normal distribution across different makes of cars:

```
[h,p] = lillietest(MPG)
Warning: P is less than the smallest tabulated value, returning 0.001.
h =
     1
p =
     1.0000e-003
```

This is clear evidence for rejecting the null hypothesis of normality, but the *p*-value returned is just the smallest value in the table of pre-computed values. To find a more accurate *p*-value for the test, run a Monte Carlo approximation using the `mctol` input argument:

```
[h,p] = lillietest(MPG,0.05,'norm',1e-4)
h =
     1
p =
  8.3333e-006
```

References

[1] Conover, W.J., *Practical Nonparametric Statistics*, Wiley, 1980

[2] Lilliefors, H.W., "On the Komogorov-Smirnov test for normality with mean and variance unknown," *Journal of the American Statistical Association*, vol. 62, 1967, pp. 399-402.

[3] Lilliefors, H.W., "On the Kolmogorov-Smirnov test for the exponential distribution with mean unknown," *Journal of the American Statistical Association*, vol. 64, 1969, pp. 387-389.

See Also

`jbtest`, `kstest`, `kstest2`, `cdfplot`

linhyptest

Purpose

Linear hypothesis test on parameter estimates

Syntax

```
p = linhyptest(beta,SIGMA,c,H,dfe)
[p,t,r] = linhyptest(...)
```

Description

`p = linhyptest(beta,SIGMA,c,H,dfe)` returns the *p*-value `p` of a hypothesis test on a vector of parameters. `beta` is a vector of *k* parameter estimates. `SIGMA` is the *k*-by-*k* estimated covariance matrix of the parameter estimates. `c` and `H` specify the null hypothesis in the form `H*b = c`, where `b` is the vector of unknown parameters estimated by `beta`. `dfe` is the degrees of freedom for the `SIGMA` estimate, or `Inf` if `SIGMA` is known rather than estimated.

`beta` is required. The remaining arguments have default values:

- `SIGMA = eye(k)`
- `c = zeros(k,1)`
- `H = eye(K)`
- `dfe = Inf`

If `H` is omitted, `c` must have *k* elements and it specifies the null hypothesis values for the entire parameter vector.

Note The `nlinfit` function returns a `SIGMA` output argument suitable for use as the `SIGMA` input argument to `linhyptest`. The following functions return `covb`, which can also be used as the `SIGMA` input to `linhyptest`: `coxphfit`, `glmfit`, `mnrfit`, `regstats`, `robustfit`. Except for `regstats`, `covb` is returned as a field in a `stats` output structure.

`[p,t,r] = linhyptest(...)` also returns the test statistic `t` and the rank `r` of the hypothesis matrix `H`. If `dfe` is `Inf` or is not given, `t` is a chi-square statistic with `r` degrees of freedom . If `dfe` is specified as a finite value, `t` is an *F* statistic with `r` and `dfe` degrees of freedom.

`linhyptest` performs a test based on an asymptotic normal distribution for the parameter estimates. It can be used after any estimation procedure for which the parameter covariances are available, such as `regstats` or `glmfit`. For linear regression, the *p*-values are exact. For other procedures, the *p*-values are approximate, and may be less accurate than other procedures such as those based on a likelihood ratio.

Example

Fit a multiple linear model to the data in `hald.mat`:

```
load hald
stats = regstats(heat,ingredients,'linear');
beta = stats.beta
beta =
   62.4054
    1.5511
    0.5102
    0.1019
   -0.1441
```

Perform an *F*-test that the last two coefficients are both 0:

```
SIGMA = stats.covb;
dfe = stats.fstat.dfe;
H = [0 0 0 1 0;0 0 0 0 1];
c = [0;0];
[p,F] = linhyptest(beta,SIGMA,c,H,dfe)
p =
    0.4668
F =
    0.8391
```

See Also

`regstats`, `glmfit`, `robustfit`, `mnrfit`, `nlinfit`, `coxphfit`

linkage

Purpose

Create hierarchical cluster tree

Syntax

```
Z = linkage(Y)
Z = linkage(Y,method)
```

Description

`Z = linkage(Y)` creates a hierarchical cluster tree, using the Single Linkage algorithm. The input `Y` is a distance vector of length $((m-1)\cdot m/2)$-by-1, where m is the number of objects in the original data set. You can generate such a vector with the `pdist` function. `Y` can also be a more general dissimilarity matrix conforming to the output format of `pdist`.

`Z = linkage(Y,`*`method`*`)` computes a hierarchical cluster tree using the algorithm specified by *`method`*, where *`method`* can be any of the following character strings, whose definitions are explained in "Mathematical Definitions" on page 14-429.

`'single'`	Shortest distance (default)
`'complete'`	Furthest distance
`'average'`	Unweighted average distance (UPGMA) (also known as group average)
`'weighted'`	Weighted average distance (WPGMA)
`'centroid'`	Centroid distance (UPGMC)
`'median'`	Weighted center of mass distance (WPGMC)
`'ward'`	Inner squared distance (minimum variance algorithm)

Note When `'method'` is `'centroid'`, `'median'`, or `'ward'`, the output of `linkage` is meaningful only if the input `Y` contains Euclidean distances.

The output, Z, is an (*m*-1)-by-3 matrix containing cluster tree information. The leaf nodes in the cluster hierarchy are the objects in the original data set, numbered from 1 to *m*. They are the singleton clusters from which all higher clusters are built. Each newly formed cluster, corresponding to row *i* in Z, is assigned the index *m*+*i*, where *m* is the total number of initial leaves.

Columns 1 and 2, `Z(i,1:2)`, contain the indices of the objects that were linked in pairs to form a new cluster. This new cluster is assigned the index value *m*+*i*. There are *m*-1 higher clusters that correspond to the interior nodes of the hierarchical cluster tree.

Column 3, `Z(i,3)`, contains the corresponding linkage distances between the objects paired in the clusters at each row *i*.

For example, consider a case with 30 initial nodes. If the tenth cluster formed by the `linkage` function combines object 5 and object 7 and their distance is 1.5, then row 10 of `Z` will contain the values (`5`, `7`, `1.5`). This newly formed cluster will have the index 10+30=40. If cluster 40 shows up in a later row, that means this newly formed cluster is being combined again into some bigger cluster.

Mathematical Definitions

The *`method`* argument is a character string that specifies the algorithm used to generate the hierarchical cluster tree information. These linkage algorithms are based on different ways of measuring the distance between two clusters of objects. If n_r is the number of objects in cluster *r* and n_s is the number of objects in cluster *s*, and x_{ri} is the *i*th object in cluster *r*, the definitions of these various measurements are as follows:

- *Single linkage*, also called *nearest neighbor*, uses the smallest distance between objects in the two clusters.

$$d(r,s) = min(dist(x_{ri}, x_{sj})), i \in (i, \ldots, n_r), j \in (1, \ldots, n_s)$$

- *Complete linkage*, also called *furthest neighbor*, uses the largest distance between objects in the two clusters.

$$d(r,s) = max(dist(x_{ri}, x_{sj})), i \in (1, \ldots, n_r), j \in (1, \ldots, n_s)$$

- *Average linkage* uses the average distance between all pairs of objects in cluster *r* and cluster *s*.

  $$d(r,s) = \frac{1}{n_r n_s} \sum_{i=1}^{n_r} \sum_{j=1}^{n_s} dist(x_{ri}, x_{sj})$$

- *Centroid linkage* uses the Euclidean distance between the centroids of the two clusters,

  $$d(r,s) = \| \bar{x}_r - \bar{x}_s \|_2$$

  where

  $$\bar{x}_r = \frac{1}{n_r} \sum_{i=1}^{n_r} x_{ri}$$

  $\bar{x}_s$ is defined similarly. The input Y should contain Euclidean distances.

- Median linkage uses the Euclidean distance between weighted centroids of the two clusters,

  $$d(r,s) = \| \tilde{x}_r - \tilde{x}_s \|_2$$

  where $\tilde{x}_r$ and $\tilde{x}_s$ are weighted centroids for the clusters *r* and *s*. If cluster *r* was created by combining clusters *p* and *q*, $\tilde{x}_r$ is defined recursively as

  $$\tilde{x}_r = \frac{1}{2}(\tilde{x}_p + \tilde{x}_q)$$

  $\tilde{x}_s$ is defined similarly. The input Y should contain Euclidean distances.

- *Ward's linkage* uses the incremental sum of squares; that is, the increase in the total within-cluster sum of squares as a result of joining clusters *r* and *s*. The within-cluster sum of squares is defined as the sum of the squares of the distances between all objects in the cluster and the centroid of the cluster. The equivalent distance is given by

$$d^2(r,s) = n_r n_s \frac{\|\bar{x}_r - \bar{x}_s\|_2^2}{(n_r + n_s)}$$

where $\| \|_2$ is Euclidean distance, and $\bar{x}_r$ and $\bar{x}_s$ are the centroids of clusters *r* and *s*, as defined in the Centroid linkage, respectively. The input `Y` should contain Euclidean distances.

The `centroid` and `median` methods can produce a cluster tree that is not monotonic. This occurs when the distance from the union of two clusters, *r* and *s*, to a third cluster is less than the distance from either *r* or *s* to that third cluster. In this case, sections of the dendrogram change direction. This is an indication that you should use another method.

Example

```
X = [3 1.7; 1 1; 2 3; 2 2.5; 1.2 1; 1.1 1.5; 3 1];
Y = pdist(X);
Z = linkage(Y)
Z =
    2.0000  5.0000  0.2000
    3.0000  4.0000  0.5000
    8.0000  6.0000  0.5099
    1.0000  7.0000  0.7000
   11.0000  9.0000  1.2806
   12.0000 10.0000  1.3454
```

See Also

`cluster`, `clusterdata`, `cophenet`, `dendrogram`, `inconsistent`, `kmeans`, `pdist`, `silhouette`, `squareform`

logncdf

Purpose Lognormal cumulative distribution function

Syntax

```
P = logncdf(X,mu,sigma)
[P,PLO,PUP] = logncdf(X,mu,sigma,pcov,alpha)
```

Description

`P = logncdf(X,mu,sigma)` returns values at X of the lognormal cdf with distribution parameters `mu` and `sigma`. `mu` and `sigma` are the mean and standard deviation, respectively, of the associated normal distribution. `X`, `mu`, and `sigma` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input for `X`, `mu`, or `sigma` is expanded to a constant array with the same dimensions as the other inputs.

`[P,PLO,PUP] = logncdf(X,mu,sigma,pcov,alpha)` returns confidence bounds for `P` when the input parameters `mu` and `sigma` are estimates. `pcov` is the covariance matrix of the estimated `parameters`. `alpha` specifies 100(1 - `alpha`)% confidence bounds. The default value of `alpha` is 0.05. `PLO` and `PUP` are arrays of the same size as `P` containing the lower and upper confidence bounds.

`logncdf` computes confidence bounds for `P` using a normal approximation to the distribution of the estimate

$$\frac{X-\hat{\mu}}{\hat{\sigma}}$$

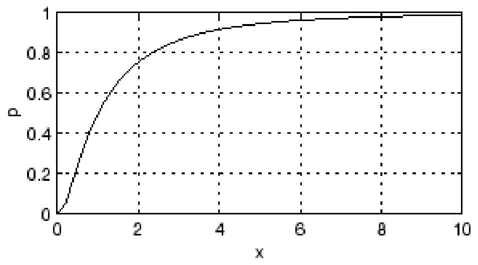
and then transforming those bounds to the scale of the output `P`. The computed bounds give approximately the desired confidence level when you estimate `mu`, `sigma`, and `pcov` from large samples, but in smaller samples other methods of computing the confidence bounds might be more accurate.

The lognormal cdf is

$$p = F(x|\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}}\int_0^x \frac{e^{\frac{-(\ln(t)-\mu)^2}{2\sigma^2}}}{t}\,dt$$

Example

```
x = (0:0.2:10);
y = logncdf(x,0,1);
plot(x,y); grid;
xlabel('x'); ylabel('p');
```



Reference

[1] Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions, 2nd Edition*, John Wiley and Sons, 1993, p. 102-105.

See Also

`cdf`, `logninv`, `lognpdf`, `lognrnd`, `lognstat`

Purpose

Parameter estimates and confidence intervals for lognormally distributed data

Syntax

```
parmhat = lognfit(data)
[parmhat,parmci] = lognfit(data)
[parmhat,parmci] = lognfit(data,alpha)
[...] = lognfit(data,alpha,censoring)
[...] = lognfit(data,alpha,censoring,freq)
[...] = lognfit(data,alpha,censoring,freq,options)
```

Description

`parmhat = lognfit(data)` returns a vector of maximum likelihood estimates `parmhat(1) = mu` and `parmhat(2) = sigma` of parameters for a lognormal distribution fitting `data`. `mu` and `sigma` are the mean and standard deviation, respectively, of the associated normal distribution.

`[parmhat,parmci] = lognfit(data)` returns 95% confidence intervals for the parameter estimates `mu` and `sigma` in the 2-by-2 matrix `parmci`. The first column of the matrix contains the lower and upper confidence bounds for parameter `mu`, and the second column contains the confidence bounds for parameter `sigma`.

`[parmhat,parmci] = lognfit(data,alpha)` returns 100(1 - `alpha`) % confidence intervals for the parameter estimates, where `alpha` is a value in the range `(0 1)` specifying the width of the confidence intervals. By default, `alpha` is `0.05`, which corresponds to 95% confidence intervals.

`[...] = lognfit(data,alpha,censoring)` accepts a Boolean vector `censoring`, of the same size as `data`, which is 1 for observations that are right-censored and 0 for observations that are observed exactly.

`[...] = lognfit(data,alpha,censoring,freq)` accepts a frequency vector, `freq`, of the same size as `data`. Typically, `freq` contains integer frequencies for the corresponding elements in `data`, but can contain any nonnegative values. Pass in `[]` for `alpha`, `censoring`, or `freq` to use their default values.

`[...] = lognfit(data,alpha,censoring,freq,options)` accepts a structure, `options`, that specifies control parameters for the iterative algorithm the function uses to compute maximum likelihood estimates when there is censoring. The lognormal fit function accepts an `options` structure which can be created using the function `statset`. Enter `statset('lognfit')` to see the names and default values of the parameters that `lognfit` accepts in the `options` structure. See the reference page for `statset` for more information about these options.

Example

This example generates 100 independent samples of lognormally distributed data with $\mu = 0$ and $\sigma = 3$. `parmhat` estimates μ and σ and `parmci` gives 99% confidence intervals around `parmhat`. Notice that `parmci` contains the true values of μ and σ.

```
data = lognrnd(0,3,100,1);
[parmhat,parmci] = lognfit(data,0.01)
parmhat =
  -0.2480  2.8902
parmci =
  -1.0071  2.4393
   0.5111  3.5262
```

See Also

`logncdf`, `logninv`, `lognlike`, `lognpdf`, `lognrnd`, `lognstat`, `mle`, `statset`

logninv

Purpose

Inverse of lognormal cumulative distribution function

Syntax

```
X = logninv(P,mu,sigma)
[X,XLO,XUP] = logninv(P,mu,sigma,pcov,alpha)
```

Description

`X = logninv(P,mu,sigma)` returns values at `P` of the inverse lognormal cdf with distribution parameters `mu` and `sigma`. `mu` and `sigma` are the mean and standard deviation, respectively, of the associated normal distribution. `mu` and `sigma` can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of `X`. A scalar input for `P`, `mu`, or `sigma` is expanded to a constant array with the same dimensions as the other inputs.

`[X,XLO,XUP] = logninv(P,mu,sigma,pcov,alpha)` returns confidence bounds for `X` when the input parameters `mu` and `sigma` are estimates. `pcov` is the covariance matrix of the estimated parameters. `alpha` specifies 100(1 - `alpha`)% confidence bounds. The default value of `alpha` is 0.05. `XLO` and `XUP` are arrays of the same size as `X` containing the lower and upper confidence bounds.

`logninv` computes confidence bounds for `P` using a normal approximation to the distribution of the estimate

$$\hat{\mu} + \hat{\sigma}q$$

where q is the `P`th quantile from a normal distribution with mean 0 and standard deviation 1. The computed bounds give approximately the desired confidence level when you estimate `mu`, `sigma`, and `pcov` from large samples, but in smaller samples other methods of computing the confidence bounds might be more accurate.

The lognormal inverse function is defined in terms of the lognormal cdf as

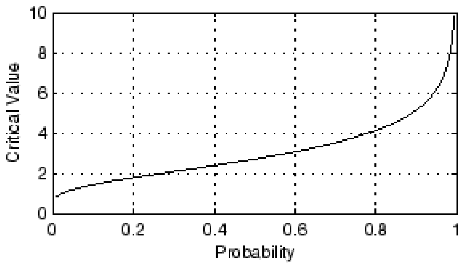
$$x = F^{-1}(p|\mu,\sigma) = \{x{:}F(x|\mu,\sigma) = p\}$$

where

$$p = F(x|\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}}\int_0^x \frac{e^{\frac{-(\ln(t)-\mu)^2}{2\sigma^2}}}{t}dt$$

Example

```
p = (0.005:0.01:0.995);
crit = logninv(p,1,0.5);
plot(p,crit)
xlabel('Probability'); ylabel('Critical Value'); grid
```



Reference

[1] Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions, 2nd edition*, John Wiley and Sons, 1993, pp. 102-105.

See Also

`icdf`, `logncdf`, `lognpdf`, `lognrnd`, `lognstat`

lognlike

Purpose

Negative log-likelihood for lognormal distribution

Syntax

```
nlogL = lognlike(params,data)
[nlogL,avar] = lognlike(params,data)
[...] = lognlike(params,data,censoring)
[...] = lognlike(params,data,censoring,freq)
```

Description

`nlogL = lognlike(params,data)` returns the negative log-likelihood of `data` for the lognormal distribution with parameters `params(1) = mu` and `params(2) = sigma`. `mu` and `sigma` are the mean and standard deviation, respectively, of the associated normal distribution. The values of `mu` and `sigma` are scalars, and the output `nlogL` is a scalar.

`[nlogL,avar] = lognlike(params,data)` returns the inverse of Fisher's information matrix. If the input parameter value in `params` is the maximum likelihood estimate, `avar` is its asymptotic variance. `avar` is based on the observed Fisher's information, not the expected information.

`[...]  = lognlike(params,data,censoring)` accepts a Boolean vector, `censoring`, of the same size as `data`, which is 1 for observations that are right-censored and 0 for observations that are observed exactly.

`[...]  = lognlike(params,data,censoring,freq)` accepts a frequency vector, `freq`, of the same size as `data`. The vector `freq` typically contains integer frequencies for the corresponding elements in `data`, but can contain any nonnegative values. Pass in `[]` for `censoring` to use its default value.

See Also

`logncdf`, `lognfit`, `logninv`, `lognpdf`, `lognrnd`

Purpose

Lognormal probability density function

Syntax

```
Y = lognpdf(X,mu,sigma)
```

Description

`Y = lognpdf(X,mu,sigma)` returns values at X of the lognormal pdf with distribution parameters `mu` and `sigma`. `mu` and `sigma` are the mean and standard deviation, respectively, of the associated normal distribution. `X`, `mu`, and `sigma` can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of `Y`. A scalar input for `X`, `mu`, or `sigma` is expanded to a constant array with the same dimensions as the other inputs.

The lognormal pdf is

$$y = f(x|\mu,\sigma) = \frac{1}{x\sigma\sqrt{2\pi}}e^{\frac{-(\ln(x)-\mu)^2}{2\sigma^2}}$$

The normal and lognormal distributions are closely related. If X is distributed lognormally with parameters μ and σ, then log(X) is distributed normally with mean μ and standard deviation σ.

The mean m and variance v of a lognormal random variable are functions of μ and σ that can be calculated with the `lognstat` function. They are:

$$m = \exp(\mu + \sigma^2/2)$$
$$v = \exp(2\mu + \sigma^2)\exp(\sigma^2 - 1)$$

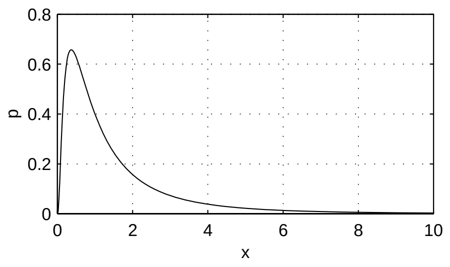
A lognormal distribution with mean m and variance v has parameters

$$\mu = \log(m^2/\sqrt{v+m^2})$$
$$\sigma = \sqrt{\log(v/m^2+1)}$$

The lognormal distribution is applicable when the quantity of interest must be positive, since log(X) exists only when X is positive.

Example

```
x = (0:0.02:10);
y = lognpdf(x,0,1);
plot(x,y); grid;
xlabel('x'); ylabel('p')
```



Reference

[1] Mood, A. M., F. A. Graybill, and D. C. Boes, *Introduction to the Theory of Statistics, 3rd edition,* McGraw-Hill, 1974, pp. 540-541.

See Also

`logncdf`, `logninv`, `lognrnd`, `lognstat`, `pdf`

Purpose

Random numbers from lognormal distribution

Syntax

```
R = lognrnd(mu,sigma)
R = lognrnd(mu,sigma,v)
R = lognrnd(mu,sigma,m,n)
```

Description

`R = lognrnd(mu,sigma)` returns an array of random numbers generated from the lognormal distribution with parameters `mu` and `sigma`. `mu` and `sigma` are the mean and standard deviation, respectively, of the associated normal distribution. `mu` and `sigma` can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of `R`. A scalar input for `mu` or `sigma` is expanded to a constant array with the same dimensions as the other input.

`R = lognrnd(mu,sigma,v)` returns an array of random numbers generated from the lognormal distribution with parameters `mu` and `sigma`, where `v` is a row vector. If `v` is a 1-by-2 vector, `R` is a matrix with `v(1)` rows and `v(2)` columns. If `v` is 1-by-n, `R` is an n-dimensional array.

`R = lognrnd(mu,sigma,m,n)` returns an array of random numbers generated from the lognormal distribution with parameters `mu` and `sigma`, where scalars `m` and `n` are the row and column dimensions of `R`.

The normal and lognormal distributions are closely related. If X is distributed lognormally with parameters μ and σ, then log(X) is distributed normally with mean μ and standard deviation σ.

The mean m and variance v of a lognormal random variable are functions of μ and σ that can be calculated with the `lognstat` function. They are:

$$m = \exp(\mu + \sigma^2/2)$$

$$v = \exp(2\mu + \sigma^2)\exp(\sigma^2 - 1)$$

A lognormal distribution with mean m and variance v has parameters

$$\mu = \log(m^2 / \sqrt{v + m^2})$$
$$\sigma = \sqrt{\log(v / m^2 + 1)}$$

Example

Generate one million lognormally distributed random numbers with mean 1 and variance 2:

```
m = 1;
v = 2;
mu = log((m^2)/sqrt(v+m^2));
sigma = sqrt(log(v/(m^2)+1));

[M,V]= lognstat(mu,sigma)
M =
     1
V =
    2.0000

X = lognrnd(mu,sigma,1,1e6);

MX = mean(X)
MX =
    0.9974
VX = var(X)
VX =
    1.9776
```

Reference

[1] Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions, 2nd edition*, John Wiley and Sons, 1993, pp. 102-105.

See Also

`random`, `logncdf`, `logninv`, `lognpdf`, `lognstat`

Purpose

Mean and variance of lognormal distribution

Syntax

```
[M,V] = lognstat(mu,sigma)
```

Description

`[M,V] = lognstat(mu,sigma)` returns the mean of and variance of the lognormal distribution with parameters `mu` and `sigma`. `mu` and `sigma` are the mean and standard deviation, respectively, of the associated normal distribution. `mu` and `sigma` can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of `M` and `V`. A scalar input for `mu` or `sigma` is expanded to a constant array with the same dimensions as the other input.

The normal and lognormal distributions are closely related. If X is distributed lognormally with parameters μ and σ, then $\log(X)$ is distributed normally with mean μ and standard deviation σ.

The mean m and variance v of a lognormal random variable are functions of μ and σ that can be calculated with the `lognstat` function. They are:

$$m = \exp(\mu + \sigma^2/2)$$
$$v = \exp(2\mu + \sigma^2)\exp(\sigma^2 - 1)$$

A lognormal distribution with mean m and variance v has parameters

$$\mu = \log(m^2/\sqrt{v + m^2})$$
$$\sigma = \sqrt{\log(v/m^2 + 1)}$$

Example

Generate one million lognormally distributed random numbers with mean 1 and variance 2:

```
m = 1;
v = 2;
mu = log((m^2)/sqrt(v+m^2));
sigma = sqrt(log(v/(m^2)+1));
```

```
[M,V]= lognstat(mu,sigma)
M =
     1
V =
    2.0000

X = lognrnd(mu,sigma,1,1e6);

MX = mean(X)
MX =
    0.9974
VX = var(X)
VX =
    1.9776
```

Reference

[1] Mood, A. M., F. A. Graybill, and D.C. Boes, *Introduction to the Theory of Statistics, 3rd edition,* McGraw-Hill 1974, pp. 540-541.

See Also

`logncdf`, `logninv`, `lognrnd`, `lognrnd`

Purpose

Parameters of generalized Pareto distribution lower tail

Syntax

`params = lowerparams(obj)`

Description

`params = lowerparams(obj)` returns the 2-element vector `params` of shape and scale parameters, respectively, of the lower tail of the Pareto tails object `obj`. `lowerparams` does not return a location parameter.

Example

Fit Pareto tails to a *t* distribution at cumulative probabilities 0.1 and 0.9:

```
t = trnd(3,100,1);
obj = paretotails(t,0.1,0.9);

lowerparams(obj)
ans =
   -0.1901    1.1898
upperparams(obj)
ans =
    0.3646    0.5103
```

See Also

`paretotails`, `upperparams`

Purpose

Plot least squares lines

Syntax

```
lsline
h = lsline
```

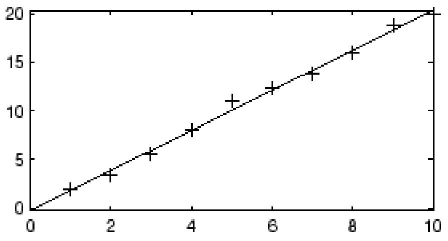
Description

`lsline` superimposes the least squares line on each line object in the current axes (except `LineStyles` `'-'`, `'--'`, `'.-'`).

`h = lsline` returns the handles to the line objects.

Example

```
y = [2 3.4 5.6 8 11 12.3 13.8 16 18.8 19.9]';
plot(y,'+');
lsline;
```



See Also

`polyfit`, `polyval`

Purpose

Mean or median absolute deviation of sample

Syntax

```
y = mad(X)
Y = mad(X,1)
Y = mad(X,0)
```

Description

`y = mad(X)` returns the mean absolute deviation of the values in `X`. For vector input, `y` is `mean(abs(X - mean(X))`. For a matrix input, `y` is a row vector containing the mean absolute deviation of each column of `X`. For N-dimensional arrays, `mad` operates along the first nonsingleton dimension of `X`.

`Y = mad(X,1)` computes Y based on medians, that is, `median(abs(X-median(X))`.

`Y = mad(X,0)` is the same as `mad(X)`, and uses means.

`mad(X,flag,dim)` takes the MAD along dimension `dim` of `X`.

`mad` treats `NaN`s as missing values and removes them.

Remarks

The MAD is less efficient than the standard deviation as an estimate of the spread when all the data is from the normal distribution.

For normal data, multiply the MAD by `1.3` as a robust estimate of σ (the scale parameter of the normal distribution).

Note The default version of MAD, based on means, is also commonly referred to as the average absolute deviation (AAD).

Examples

This example shows a Monte Carlo simulation of the relative efficiency of the MAD to the sample standard deviation for normal data.

```
x = normrnd(0,1,100,100);
s = std(x);
s_MAD = 1.3 * mad(x);
efficiency = (norm(s - 1)./norm(s_MAD - 1)).^2
efficiency =
  0.5972
```

Reference [1] Sachs, L., *Applied Statistics: A Handbook of Techniques*, Springer-Verlag, 1984, p. 253.

See Also `std`, `range`, `iqr`

Purpose

Mahalanobis distance

Syntax

```
mahal(Y,X)
```

Description

`mahal(Y,X)` computes the Mahalanobis distance (in squared units) of each point (row) of the matrix `Y` from the sample in the matrix `X`.

The number of columns of `Y` must equal the number of columns in `X`, but the number of rows may differ. The number of rows in `X` must exceed the number of columns.

The Mahalanobis distance is a multivariate measure of the separation of a data set from a point in space. It is the criterion minimized in linear discriminant analysis.

Example

The Mahalanobis distance of a matrix `r` when applied to itself is a way to find outliers.

```
r = mvnrnd([0 0],[1 0.9;0.9 1],100);
r = [r;10 10];
d = mahal(r,r);
last6 = d(96:101)
last6 =
  1.1036
  2.2353
  2.0219
  0.3876
  1.5571
  52.7381
```

The last element is clearly an outlier.

See Also

`classify`

Purpose

Main effects plot for grouped data

Syntax

```
maineffectsplot(Y,GROUP)
maineffectsplot(Y,GROUP,param1,val1,param2,val2,...)
[figh,AXESH] = maineffectsplot(...)
```

Description

`maineffectsplot(Y,GROUP)` displays main effects plots for the group means of matrix `Y` with groups defined by entries in the cell array `GROUP`. `Y` is a numeric matrix or vector. If `Y` is a matrix, the rows represent different observations and the columns represent replications of each observation. Each cell of `GROUP` must contain a grouping variable that can be a categorical variable, numeric vector, character matrix, or single-column cell array of strings. (See "Grouped Data" on page 2-41.) `GROUP` can also be a matrix whose columns represent different grouping variables. Each grouping variable must have the same number of rows as `Y`. The number of grouping variables must be greater than 1.

The display has one subplot per grouping variable, with each subplot showing the group means of `Y` as a function of one grouping variable.

`maineffectsplot(Y,GROUP,`*`param1`*`,`*`val1`*`,`*`param2`*`,`*`val2`*`,...)` specifies one or more of the following name/value pairs:

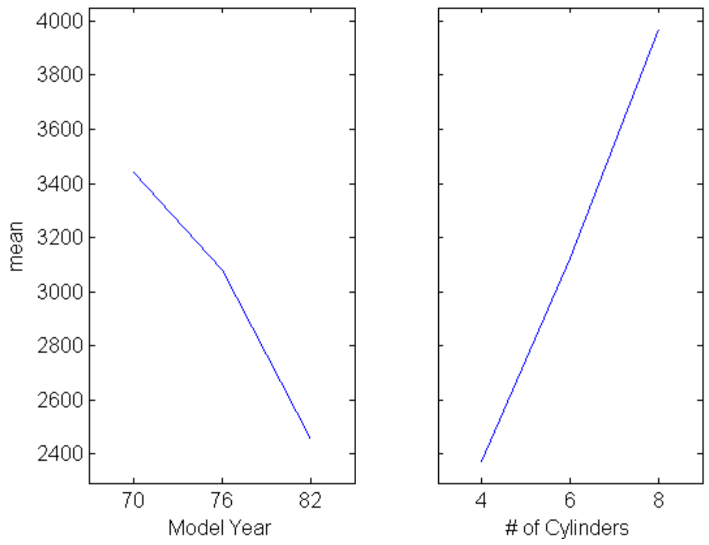
- `'varnames'` — Grouping variable names in a character matrix or a cell array of strings, one per grouping variable. Default names are `'X1'`, `'X2'`, ...
- `'statistic'` — String values that indicate whether the group mean or the group standard deviation should be plotted. Use `'mean'` or `'std'`. The default is `'mean'`. If the value is `'std'`, `Y` is required to have multiple columns.
- `'parent'` — A handle to the figure window for the plots. The default is the current figure window.

`[figh,AXESH] = maineffectsplot(...)` returns the handle `figh` to the figure window and an array of handles `AXESH` to the subplot axes.

Example

Display main effects plots for car weight with two grouping variables, model year and number of cylinders:

```
load carsmall;
maineffectsplot(Weight,{Model_Year,Cylinders}, ...
                'varnames',{'Model Year','# of Cylinders'})
```



See Also

`interactionplot`, `multivarichart`

Purpose

One-way multivariate analysis of variance

Syntax

```
d = manova1(X,group)
d = manova1(X,group,alpha)
[d,p] = manova1(...)
[d,p,stats] = manova1(...)
```

Description

`d = manova1(X,group)` performs a one-way Multivariate Analysis of Variance (MANOVA) for comparing the multivariate means of the columns of `X`, grouped by `group`. `X` is an m-by-n matrix of data values, and each row is a vector of measurements on n variables for a single observation. `group` is a grouping variable defined as a categorical variable, vector, string array, or cell array of strings. Two observations are in the same group if they have the same value in the `group` array. (See "Grouped Data" on page 2-41.) The observations in each group represent a sample from a population.

The function returns `d`, an estimate of the dimension of the space containing the group means. `manova1` tests the null hypothesis that the means of each group are the same n-dimensional multivariate vector, and that any difference observed in the sample `X` is due to random chance. If `d = 0`, there is no evidence to reject that hypothesis. If `d = 1`, then you can reject the null hypothesis at the 5% level, but you cannot reject the hypothesis that the multivariate means lie on the same line. Similarly, if `d = 2` the multivariate means may lie on the same plane in n-dimensional space, but not on the same line.

`d = manova1(X,group,alpha)` gives control of the significance level, `alpha`. The return value `d` will be the smallest dimension having `p > alpha`, where `p` is a p-value for testing whether the means lie in a space of that dimension.

`[d,p] = manova1(...)` also returns a `p`, a vector of p-values for testing whether the means lie in a space of dimension 0, 1, and so on. The largest possible dimension is either the dimension of the space, or one less than the number of groups. There is one element of `p` for each dimension up to, but not including, the largest.

If the *i*th p-value is near zero, this casts doubt on the hypothesis that the group means lie on a space of *i*-1 dimensions. The choice of a critical p-value to determine whether the result is judged statistically significant is left to the researcher and is specified by the value of the input argument `alpha`. It is common to declare a result significant if the p-value is less than 0.05 or 0.01.

`[d,p,stats] = manova1(...)` also returns `stats`, a structure containing additional MANOVA results. The structure contains the following fields.

Field	**Contents**
`W`	Within-groups sum of squares and cross-products matrix
`B`	Between-groups sum of squares and cross-products matrix
`T`	Total sum of squares and cross-products matrix
`dfW`	Degrees of freedom for `W`
`dfB`	Degrees of freedom for `B`
`dfT`	Degrees of freedom for `T`
`lambda`	Vector of values of Wilk's lambda test statistic for testing whether the means have dimension 0, 1, etc.
`chisq`	Transformation of `lambda` to an approximate chi-square distribution
`chisqdf`	Degrees of freedom for `chisq`
`eigenval`	Eigenvalues of $W^{-1}B$
`eigenvec`	Eigenvectors of $W^{-1}B$; these are the coefficients for the canonical variables `C`, and they are scaled so the within-group variance of the canonical variables is 1

Field	Contents
canon	Canonical variables C, equal to XC*eigenvec, where XC is X with columns centered by subtracting their means
mdist	A vector of Mahalanobis distances from each point to the mean of its group
gmdist	A matrix of Mahalanobis distances between each pair of group means

The canonical variables C are linear combinations of the original variables, chosen to maximize the separation between groups. Specifically, C(:,1) is the linear combination of the X columns that has the maximum separation between groups. This means that among all possible linear combinations, it is the one with the most significant F statistic in a one-way analysis of variance. C(:,2) has the maximum separation subject to it being orthogonal to C(:,1), and so on.

You may find it useful to use the outputs from manova1 along with other functions to supplement your analysis. For example, you may want to start with a grouped scatter plot matrix of the original variables using gplotmatrix. You can use gscatter to visualize the group separation using the first two canonical variables. You can use manovacluster to graph a dendrogram showing the clusters among the group means.

Assumptions

The MANOVA test makes the following assumptions about the data in X:

- The populations for each group are normally distributed.
- The variance-covariance matrix is the same for each population.
- All observations are mutually independent.

Example

you can use manova1 to determine whether there are differences in the averages of four car characteristics, among groups defined by the country where the cars were made.

```
load carbig
[d,p] = manova1([MPG Acceleration Weight Displacement],...
                Origin)
d =
   3
p =
     0
  0.0000
  0.0075
  0.1934
```

There are four dimensions in the input matrix, so the group means must lie in a four-dimensional space. `manova1` shows that you cannot reject the hypothesis that the means lie in a three-dimensional subspace.

References

[1] Krzanowski, W. J., *Principles of Multivariate Analysis*. Oxford University Press, 1988.

See Also

`anova1`, `canoncorr`, `gscatter`, `gplotmatrix`, `manovacluster`

manovacluster

Purpose

Dendrogram of group mean clusters following MANOVA

Syntax

```
manovacluster(stats)
manovacluster(stats,method)
H = manovacluster(stats,method)
```

Description

`manovacluster(stats)` generates a dendrogram plot of the group means after a multivariate analysis of variance (MANOVA). `stats` is the output `stats` structure from `manova1`. The clusters are computed by applying the single linkage method to the matrix of Mahalanobis distances between group means.

See `dendrogram` for more information on the graphical output from this function. The dendrogram is most useful when the number of groups is large.

`manovacluster(stats,`*`method`*`)` uses the specified method in place of single linkage. *`method`* can be any of the following character strings that identify ways to create the cluster hierarchy. See `linkage` for further explanation.

`'single'`	Shortest distance (default)
`'complete'`	Largest distance
`'average'`	Average distance
`'centroid'`	Centroid distance
`'ward'`	Incremental sum of squares

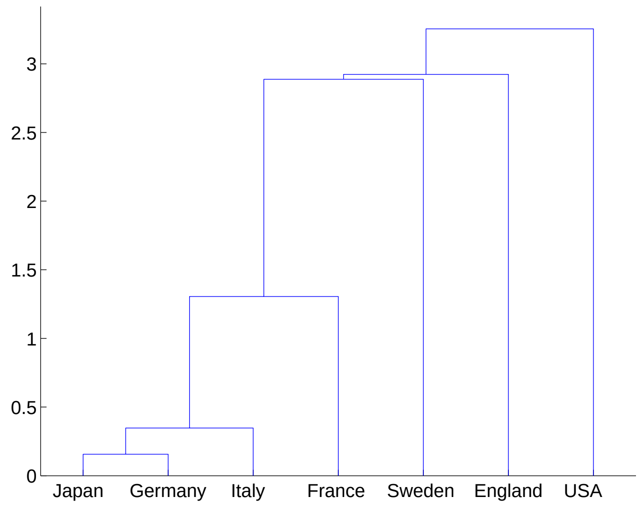
`H = manovacluster(stats,`*`method`*`)` returns a vector of handles to the lines in the figure.

Example

Let's analyze the larger car data set to determine which countries produce cars with the most similar characteristics.

```
load carbig
X = [MPG Acceleration Weight Displacement];
[d,p,stats] = manova1(X,Origin);
```

```
manovacluster(stats)
```



See Also

`cluster`, `dendrogram`, `linkage`, `manova1`

mdscale

Purpose

Nonmetric and metric multidimensional scaling

Syntax

```
Y = mdscale(D,p)
[Y,stress] = mdscale(D,p)
[Y,stress,disparities] = mdscale(D,p)
[...] = mdscale(...,param1,val1,param2,val2,...)
```

Description

`Y = mdscale(D,p)` performs nonmetric multidimensional scaling on the n-by-n dissimilarity matrix `D`, and returns `Y`, a configuration of n points (rows) in `p` dimensions (columns). The Euclidean distances between points in `Y` approximate a monotonic transformation of the corresponding dissimilarities in **`D`**. By default, `mdscale` uses Kruskal's normalized stress1 criterion.

You can specify `D` as either a full n-by-n matrix, or in upper triangle form such as is output by `pdist`. A full dissimilarity matrix must be real and symmetric, and have zeros along the diagonal and non-negative elements everywhere else. A dissimilarity matrix in upper triangle form must have real, non-negative entries. `mdscale` treats `NaN`s in `D` as missing values, and ignores those elements. `Inf` is not accepted.

You can also specify `D` as a full similarity matrix, with ones along the diagonal and all other elements less than one. `mdscale` transforms a similarity matrix to a dissimilarity matrix in such a way that distances between the points returned in `Y` approximate `sqrt(1-D)`. To use a different transformation, transform the similarities prior to calling `mdscale`.

`[Y,stress] = mdscale(D,p)` returns the minimized stress, i.e., the stress evaluated at `Y`.

`[Y,stress,disparities] = mdscale(D,p)` returns the disparities, that is, the monotonic transformation of the dissimilarities `D`.

`[...] = mdscale(...,param1,val1,param2,val2,...)` enables you to specify optional parameter name/value pairs that control further details of `mdscale`. The parameters are

- `'Criterion'`— The goodness-of-fit criterion to minimize. This also determines the type of scaling, either non-metric or metric, that `mdscale` performs. Choices for non-metric scaling are:
  - `'stress'` — Stress normalized by the sum of squares of the inter-point distances, also known as stress1. This is the default.
  - `'sstress'` — Squared stress, normalized with the sum of 4th powers of the inter-point distances.

  Choices for metric scaling are:
  - `'metricstress'` — Stress, normalized with the sum of squares of the dissimilarities.
  - `'metricsstress'` — Squared stress, normalized with the sum of 4th powers of the dissimilarities.
  - `'sammon'` — Sammon's nonlinear mapping criterion. Off-diagonal dissimilarities must be strictly positive with this criterion.
  - `'strain'` — A criterion equivalent to that used in classical multidimensional scaling.
- `'Weights'` — A matrix or vector the same size as `D`, containing nonnegative dissimilarity weights. You can use these to weight the contribution of the corresponding elements of `D` in computing and minimizing stress. Elements of `D` corresponding to zero weights are effectively ignored.
- `'Start'` — Method used to choose the initial configuration of points for Y. The choices are
  - `'cmdscale'` — Use the classical multidimensional scaling solution. This is the default. `'cmdscale'` is not valid when there are zero weights.
  - `'random'` — Choose locations randomly from an appropriately scaled p-dimensional normal distribution with uncorrelated coordinates.
  - An *n*-by-`p` matrix of initial locations, where n is the size of the matrix `D` and `p` is the number of columns of the output matrix

Y. In this case, you can pass in [] for p and mdscale infers p from the second dimension of the matrix. You can also supply a three-dimensional array, implying a value for 'Replicates' from the array's third dimension.

- 'Replicates' — Number of times to repeat the scaling, each with a new initial configuration. The default is 1.
- 'Options' — Options for the iterative algorithm used to minimize the fitting criterion. Pass in an options structure created by statset. For example,

  ```
  opts = statset(param1,val1,param2,val2, ...);
  [...] = mdscale(...,'Options',opts)
  ```

  The choices of statset parameters are

  - 'Display' — Level of display output. The choices are 'off' (the default), 'iter', and 'final'.
  - 'MaxIter' — Maximum number of iterations allowed. The default is 200.
  - 'TolFun' — Termination tolerance for the stress criterion and its gradient. The default is 1e-4.
  - 'TolX'— Termination tolerance for the configuration location step size. The default is 1e-4.

Example

```
load cereal.mat
X = [Calories Protein Fat Sodium Fiber ...
     Carbo Sugars Shelf Potass Vitamins];

% Take a subset from a single manufacturer.
X = X(strmatch('K',Mfg),:);

% Create a dissimilarity matrix.
dissimilarities = pdist(X);

% Use non-metric scaling to recreate the data in 2D,
```

```
% and make a Shepard plot of the results.
[Y,stress,disparities] = mdscale(dissimilarities,2);
distances = pdist(Y);
[dum,ord] = sortrows([disparities(:) dissimilarities(:)]);
plot(dissimilarities,distances,'bo', ...
dissimilarities(ord),disparities(ord),'r.-');
xlabel('Dissimilarities'); ylabel('Distances/Disparities')
legend({'Distances' 'Disparities'},'Location','NW');

% Do metric scaling on the same dissimilarities.
[Y,stress] = ...
mdscale(dissimilarities,2,'criterion','metricsstress');
distances = pdist(Y);
plot(dissimilarities,distances,'bo', ...
[0 max(dissimilarities)],[0 max(dissimilarities)],'k:');
xlabel('Dissimilarities'); ylabel('Distances')
```

See Also

`cmdscale`, `pdist`, `statset`

Purpose

Mean values of vectors and matrices

Syntax

```
m = mean(X)
m = mean(X,dim)
```

Description

`m = mean(X)` calculates the sample mean

$$\overline{x_j} = \frac{1}{n}\sum_{i=1}^{n} x_{ij}$$

For vectors, `mean(x)` is the mean value of the elements in vector `x`. For matrices, `mean(X)` is a row vector containing the mean value of each column.

`m = mean(X,dim)` returns the mean values for elements along the dimension of `X` specified by scalar `dim`. For matrices, `mean(X,2)` is a column vector containing the mean value of each row. The default of `dim` is 1.

The `mean` function is part of the standard MATLAB language.

Example

These commands generate five samples of 100 normal random numbers with mean, zero, and standard deviation, one. The sample means in `xbar` are much less variable (0.00 ± 0.10).

```
x = normrnd(0,1,100,5);
xbar = mean(x)
xbar =
  0.0727  0.0264  0.0351  0.0424  0.0752
```

See Also

`median`, `std`, `cov`, `corrcoef`, `var`

Purpose

Median values of vectors and matrices

Syntax

```
m = median(X)
```

Description

`m = median(X)` calculates the median value, which is the 50th percentile of a sample. The median is a robust estimate of the center of a sample of data, since outliers have little effect on it.

For vectors, `median(x)` is the median value of the elements in vector `x`. For matrices, `median(X)` is a row vector containing the median value of each column. Since `median` is implemented using `sort`, it can be costly for large matrices.

The `median` function is part of the standard MATLAB language.

Examples

```
xodd = 1:5;
modd = median(xodd)
modd =
   3

xeven = 1:4;
meven = median(xeven)
meven =
  2.5000
```

This example shows robustness of the median to outliers.

```
xoutlier = [(1:4) 10000];
moutlier = median(xoutlier)
moutlier =
   3
```

See Also

`mean`, `std`, `cov`, `corrcoef`

mergelevels

Purpose

Merge levels of categorical array

Syntax

```
B = mergelevels(A,oldlevels,newlevel)
B = mergelevels(A,oldlevels)
```

Description

`B = mergelevels(A,oldlevels,newlevel)` merges two or more levels of the categorical array `A` into a single new level. `oldlevels` is a cell array of strings or a two-dimensional character matrix that specifies the levels to be merged. Any elements of `A` that have levels in `oldlevels` are assigned the new level in the corresponding elements of `B`. `newlevel` is a character string that specifies the label for the new level. For ordinal arrays, the levels of `A` specified by `oldlevels` must be consecutive, and `mergelevels` inserts the new level to preserve the order of the levels.

`B = mergelevels(A,oldlevels)` merges two or more levels of `A`. For nominal arrays, `mergelevels` uses the first label in `oldlevels` as the label for the new level. For ordinal arrays, `mergelevels` uses the label corresponding to the lowest level in `oldlevels` as the label for the new level.

Examples

Example 1

For nominal data:

```
load fisheriris
species = nominal(species);
species = mergelevels(species,...
                      {'setosa','virginica'},'parent');
species = setlabels(species,'hybrid','versicolor');
getlabels(species)
ans =
    'hybrid'   'parent'
```

Example 2

For ordinal data:

```
A = ordinal([1 2 3 2 1],{'lo','med','hi'})
A =
     lo      med      hi      med      lo

A = mergelevels(A,{'lo','med'},'bad')
A =
     bad      bad      hi      bad      bad
```

See Also

`addlevels`, `droplevels`, `islevel`, `reorderlevels`, `getlabels`

mhsample

Purpose

Markov chain Metropolis-Hastings sampler

Syntax

```
smpl = mhsample(start,nsamples,'pdf',pdf,'proppdf',proppdf,
   'proprnd',proprnd)
smpl = mhsample(...,'symmetric',sym)
smpl = mhsample(...,'burnin',K)
smpl = mhsample(...,'thin',m)
smpl = mhsample(...,'nchain',n)
[smpl,accept] = mhsample(...)
```

Description

`smpl = mhsample(start,nsamples,'pdf',pdf,'proppdf',proppdf,'proprnd',proprnd)` draws `nsamples` random samples from a target stationary distribution `pdf` using the Metropolis-Hastings algorithm.

`start` is a row vector containing the start value of the Markov Chain, `nsamples` is an integer specifying the number of samples to be generated, and `pdf`, `proppdf`, and `proprnd` are function handles created using `@`. `proppdf` defines the proposal distribution density, and `proprnd` defines the random number generator for the proposal distribution. `pdf` and `proprnd` take one argument as an input with the same type and size as `start`. `proppdf` takes two arguments as inputs with the same type and size as `start`.

`smpl` is a column vector or matrix containing the samples. If the log density function is preferred, `'pdf'` and `'proppdf'` can be replaced with `'logpdf'` and `'proppdf'`. The density functions used in Metropolis-Hastings algorithm are not necessarily normalized. If the `proppdf` or `logpdf` q(x,y) satisfies q(x,y) = q(y,x), for example, the proposal distribution is symmetric, `mhsample` implements the Random Walk Metropolis-Hastings sampling. If the `proppdf` or `logpdf` q(x,y) satisfies q(x,y) = q(x), for example, the proposal distribution is independent of current values, `mhsample` implements the Independent Metropolis-Hastings sampling.

`smpl = mhsample(...,'symmetric',sym)` draws `nsamples` random samples from a target stationary distribution `pdf` using the Metropolis-Hastings algorithm. `sym` is a logical value that indicates whether the proposal distribution is symmetric. The default value is false, which corresponds to the asymmetric proposal distribution. If `sym` is true, for example, the proposal distribution is symmetric, `proppdf` and `logproppdf` are optional.

`smpl = mhsample(...,'burnin',K)` generates a Markov chain with values between the starting point and the `k`th point omitted in the generated sequence. Values beyond the `k`th point are kept. `k` is a nonnegative integer with default value of `0`.

`smpl = mhsample(...,'thin',m)` generates a Markov chain with `m-1` out of `m` values omitted in the generated sequence. `m` is a positive integer with default value of `1`.

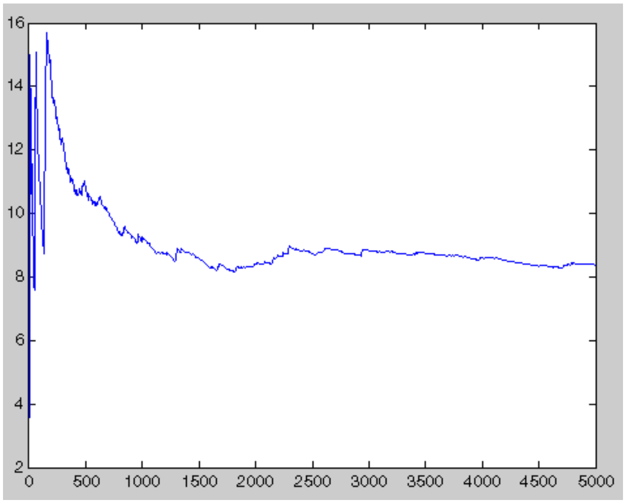
`smpl = mhsample(...,'nchain',n)` generates `n` Markov chains using the Metropolis-Hastings algorithm. `n` is a positive integer with a default value of 1. `smpl` is a matrix containing the samples. The last dimension contains the indices for individual chains.

`[smpl,accept] = mhsample(...)` also returns `accept`, the acceptance rate of the proposed distribution. `accept` is a scalar if a single chain is generated and is a vector if multiple chains are generated.

Examples

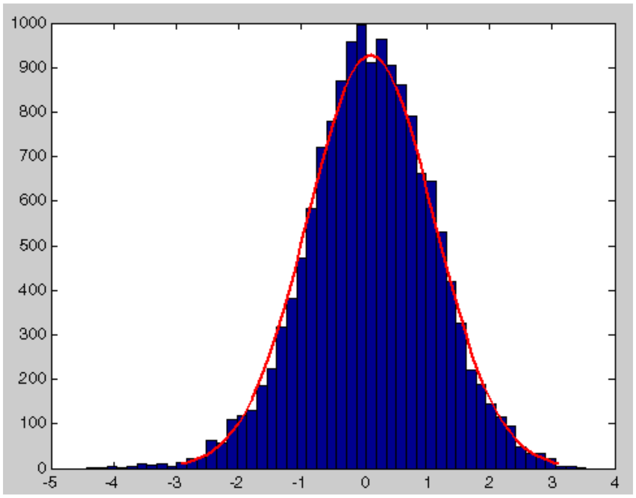
Estimate the second order moment of a Gamma distribution using the Independent Metropolis-Hastings sampling.

```
alpha = 2.43;
beta = 1;
pdf = @(x)gampdf(x,alpha,beta); %target distribution
proppdf = @(x,y)gampdf(x,floor(alpha),floor(alpha)/alpha);
proprnd = @(x)sum(...
              exprnd(floor(alpha)/alpha,floor(alpha),1));
nsamples = 5000;
smpl = mhsample(1,nsamples,'pdf',pdf,'proprnd',proprnd,...
                'proppdf',proppdf);
xxhat = cumsum(smpl.^2)./(1:nsamples)';
plot(1:nsamples,xxhat)
```



Generate random samples from `N(0,1)` using the Random Walk Metropolis-Hastings sampling.

```
delta = .5;
pdf = @(x) normpdf(x);
proppdf = @(x,y) unifpdf(y-x,-delta,delta);
proprnd = @(x) x + rand*2*delta - delta;
nsamples = 15000;
x = mhsample(1,nsamples,'pdf',pdf,'proprnd',proprnd,'symmetric',1);
histfit(x,50)
```



See Also

`slicesample`, `rand`

Purpose

Maximum likelihood estimation

Syntax

```
phat = mle(data)
[phat,pci] = mle(data)
[...] = mle(data,'distribution',dist)
[...] = mle(data,...,name1,val1,name2,val2,...)
[...] = mle(data,'pdf',pdf,'cdf',cdf,'start',start,...)
[...] = mle(data,'logpdf',logpdf,'logsf',logsf,'start',start,
   ...)
[...] = mle(data,'nloglf',nloglf,'start',start,...)
```

Description

`phat = mle(data)` returns maximum likelihood estimates (MLEs) for the parameters of a normal distribution, computed using the sample data in the vector `data`.

`[phat,pci] = mle(data)` returns MLEs and 95% confidence intervals for the parameters.

`[...]  = mle(data,'distribution',`*`dist`*`)` computes parameter estimates for the distribution specified by *`dist`*. The following table lists acceptable values for *`dist`*.

Distribution	Value of *`dist`*
Beta	`'beta'`
Bernoulli	`'bernoulli'`
Binomial	`'binomial'`
Discrete uniform	`'discrete uniform'` or `'unid'`
Exponential	`'exponential'`
Extreme value	`'extreme value'` or `'ev'`
Gamma	`'gamma'`
Generalized extreme value	`'generalized extreme value'` or `'gev'`
Generalized Pareto	`'generalized pareto'` or `'gp'`

Distribution	Value of *dist*
Geometric	`'geometric'`
Lognormal	`'lognormal'`
Negative binomial	`'negative binomial'` or `'nbin'`
Normal	`'normal'`
Poisson	`'poisson'`
Rayleigh	`'rayleigh'`
Uniform	`'uniform'`
Weibull	`'weibull'` or `'wbl'`

`[...] = mle(data,...,`*`name1`*`,`*`val1`*`,`*`name2`*`,`*`val2`*`,...)` specifies optional argument name/value pairs chosen from the following list.

Name	Value
`'censoring'`	A boolean vector of the same size as `data`, containing ones when the corresponding elements of data are right-censored observations and zeros when the corresponding elements are exact observations. The default is that all observations are observed exactly. Censoring is not supported for all distributions.
`'frequency'`	A vector of the same size as `data`, containing non-negative integer frequencies for the corresponding elements in `data`. The default is one observation per element of `data`.
`'alpha'`	A value between 0 and 1 specifying a confidence level of `100(1-alpha)`% for `pci`. The default is `0.05` for 95% confidence.

Name	Value
`'ntrials'`	A scalar, or a vector of the same size as data, containing the total number of trials for the corresponding element of data. Applies only to the binomial distribution.
`'options'`	A structure created by a call to `statset`, containing numerical options for the fitting algorithm. Not applicable to all distributions.

`mle` can also fit custom distributions that you define using distribution functions, in one of three ways.

`[...] = mle(data,'pdf',pdf,'cdf',cdf,'start',start,...)` returns MLEs for the parameters of the distribution defined by the probability density and cumulative distribution functions `pdf` and `cdf`. `pdf` and `cdf` are function handles created using the `@` sign. They accept as inputs a vector `data` and one or more individual distribution parameters, and return vectors of probability density values and cumulative probability values, respectively. If the `'censoring'` name/value pair is not present, you can omit the `'cdf'` name/value pair. `mle` computes the estimates by numerically maximizing the distribution's log-likelihood, and `start` is a vector containing initial values for the parameters.

`[...] =`
`mle(data,'logpdf',logpdf,'logsf',logsf,'start',start,...)` returns MLEs for the parameters of the distribution defined by the log probability density and log survival functions `logpdf` and `logsf`. `logpdf` and `logsf` are function handles created using the `@` sign. They accept as inputs a vector `data` and one or more individual distribution parameters, and return vectors of logged probability density values and logged survival function values, respectively. This form is sometimes more robust to the choice of starting point than using `pdf` and `cdf` functions. If the `'censoring'` name/value pair is not present, you can omit the `'logsf'` name/value pair. `start` is a vector containing initial values for the distribution's parameters.

`[...] = mle(data,'nloglf',nloglf,'start',start,...)` returns MLEs for the parameters of the distribution whose negative log-likelihood is given by `nloglf`. `nloglf` is a function handle, specified using the `@` sign, that accepts the four input arguments:

- `params` - a vector of distribution parameter values
- `data` - a vector of data
- `cens` - a boolean vector of censoring values
- `freq` - a vector of integer data frequencies

`nloglf` must accept all four arguments even if you do not supply the `'censoring'` or `'frequency'` name/value pairs (see above). However, `nloglf` can safely ignore its `cens` and `freq` arguments in that case. `nloglf` returns a scalar negative log-likelihood value and, optionally, a negative log-likelihood gradient vector (see the `'GradObj'` `statset` parameter below). `start` is a vector containing initial values for the distribution's parameters.

`pdf`, `cdf`, `logpdf`, `logsf`, or `nloglf` can also be cell arrays whose first element is a function handle as defined above, and whose remaining elements are additional arguments to the function. `mle` places these arguments at the end of the argument list in the function call.

The following optional argument name/value pairs are valid only when `'pdf'` and `'cdf'`, `'logpdf'` and `'logcdf'`, or `'nloglf'` are given:

- `'lowerbound'` — A vector the same size as `start` containing lower bounds for the distribution parameters. The default is `-Inf`.
- `'upperbound'` — A vector the same size as `start` containing upper bounds for the distribution parameters. The default is `Inf`.
- `'optimfun'` — A string, either `'fminsearch'` or `'fmincon'`, naming the optimization function to be used in maximizing the likelihood. The default is `'fminsearch'`. You can only specify `'fmincon'` if the Optimization Toolbox is available.

When fitting a custom distribution, use the `'options'` parameter to control details of the maximum likelihood optimization. See `statset('mlecustom')` for parameter names and default values. `mle` interprets the following `statset` parameters for custom distribution fitting as follows:

Parameter	Value
`'GradObj'`	`'on'` or `'off'`, indicating whether or not `fmincon` can expect the function provided with the `'nloglf'` name/value pair to return the gradient vector of the negative log-likelihood as a second output. The default is `'off'`. Ignored when using `fminsearch`.
`'DerivStep'`	The relative difference used in finite difference derivative approximations when using `fmincon`, and `'GradObj'` is `'off'`. `'DerivStep'` can be a scalar, or the same size as `'start'`. The default is `eps^(1/3)`. Ignored when using `fminsearch`.
`'FunValCheck'`	`'on'` or `'off'`, indicating whether or not `mle` should check the values returned by the custom distribution functions for validity. The default is `'on'`. A poor choice of starting point can sometimes cause these functions to return `NaN`s, infinite values, or out of range values if they are written without suitable error-checking.
`'TolBnd'`	An offset for upper and lower bounds when using `fmincon`. `mle` treats upper and lower bounds as strict inequalities (i.e., open bounds). With `fmincon`, this is approximated by creating closed bounds inset from the specified upper and lower bounds by `TolBnd`. The default is `1e-6`.

Example

```
rv = binornd(20,0.75)
rv =
  16
```

```
[p,pci] = mle('binomial',rv,0.05,20)
p =
  0.8000
pci =
  0.5634
  0.9427
```

See Also

`betafit`, `binofit`, `evfit`, `expfit`, `gamfit`, `gevfit`, `gpfit`, `lognfit`, `nbinfit`, `normfit`, `mlecov`, `poissfit`, `raylfit`, `statset`, `unifit`, `wblfit`

Purpose

Asymptotic covariance matrix of maximum likelihood estimators

Syntax

```
ACOV = mlecov(params,data,...)
ACOV = mlecov(params,data,'pdf',pdf,'cdf',cdf)
ACOV = mlecov(params,data,'logpdf',logpdf,'logsf',logsf)
ACOV = mlecov(params,data,'nloglf',nloglf)
[...] = mlecov(params,data,...,param1,val1,param2,val2,...)
```

Description

`ACOV = mlecov(params,data,...)` returns an approximation to the asymptotic covariance matrix of the maximum likelihood estimators of the parameters for a specified distribution. The following paragraphs describe how to specify the distribution. `mlecov` computes a finite difference approximation to the Hessian of the log-likelihood at the maximum likelihood estimates `params`, given the observed `data`, and returns the negative inverse of that Hessian. `ACOV` is a *p*-by-*p* matrix, where *p* is the number of elements in `params`.

You must specify a distribution after the input argument `data`, as follows.

`ACOV = mlecov(params,data,'pdf',pdf,'cdf',cdf)` enables you to define a distribution by its probability density and cumulative distribution functions, `pdf` and `cdf`, respectively. `pdf` and `cdf` are function handles that you create using the `@` sign. They accept a vector of data and one or more individual distribution parameters as inputs and return vectors of probability density function values and cumulative distribution values, respectively. If the `'censoring'` name/value pair (see below) is not present, you can omit the `'cdf'` name/value pair.

`ACOV = mlecov(params,data,'logpdf',logpdf,'logsf',logsf)` enables you to define a distribution by its log probability density and log survival functions, `logpdf` and `logsf`, respectively. `logpdf` and `logsf` are function handles that you create using the `@` sign. They accept as inputs a vector of data and one or more individual distribution parameters, and return vectors of logged probability density values and logged survival function values, respectively. If the `'censoring'` name/value pair (see below) is not present, you can omit the `'logsf'` name/value pair.

`ACOV = mlecov(params,data,'nloglf',nloglf)` enables you to define a distribution by its log-likelihood function. `nloglf` is a function handle, specified using the `@` sign, that accepts the following four input arguments:

- `params` — Vector of distribution parameter values
- `data` — Vector of data
- `cens` — Boolean vector of censoring values
- `freq` — Vector of integer data frequencies

`nloglf` must accept all four arguments even if you do not supply the `'censoring'` or `'frequency'` name/value pairs (see below). However, `nloglf` can safely ignore its `cens` and `freq` arguments in that case. `nloglf` returns a scalar negative log-likelihood value and, optionally, the negative log-likelihood gradient vector (see the `'gradient'` name/value pair below).

`pdf`, `cdf`, `logpdf`, `logsf`, and `nloglf` can also be cell arrays whose first element is a function handle, as defined above, and whose remaining elements are additional arguments to the function. The `mle` function places these arguments at the end of the argument list in the function call.

`[...] = mlecov(params,data,...,`*`param1`*`,`*`val1`*`,`*`param2`*`,`*`val2`*`,...)` specifies optional parameter name/value pairs chosen from the following:

Name	**Value**
`'censoring'`	Boolean vector of the same size as `data`, containing 1's when the corresponding elements of `data` are right-censored observations and 0's when the corresponding elements are exact observations. The default is that all observations are observed exactly. Censoring is not supported for all distributions.

Name	Value
'frequency'	A vector of the same size as data containing nonnegative frequencies for the corresponding elements in data. The default is one observation per element of data.
'options'	A structure opts containing numerical options for the finite difference Hessian calculation. You create opts by calling statset. The applicable statset parameters are: • 'GradObj' — 'on' or 'off', indicating whether or not the function provided with the 'nloglf' name/value pair can return the gradient vector of the negative log-likelihood as its second output. The default is 'off'. • 'DerivStep' — Relative step size used in finite difference for Hessian calculations. Can be a scalar, or the same size as params. The default is eps^(1/4). A smaller value might be appropriate if 'GradObj' is 'on'.

Example

Create the following M-file function:

```
function logpdf = betalogpdf(x,a,b)
logpdf = (a-1)*log(x)+(b-1)*log(1-x)-betaln(a,b);
```

Fit a beta distribution to some simulated data, and compute the approximate covariance matrix of the parameter estimates:

```
x = betarnd(1.23,3.45,25,1);
phat = mle(x,'dist','beta')
acov = mlecov(phat,x,'logpdf',@betalogpdf)
```

See Also

mle

Purpose

Multinomial probability density function

Syntax

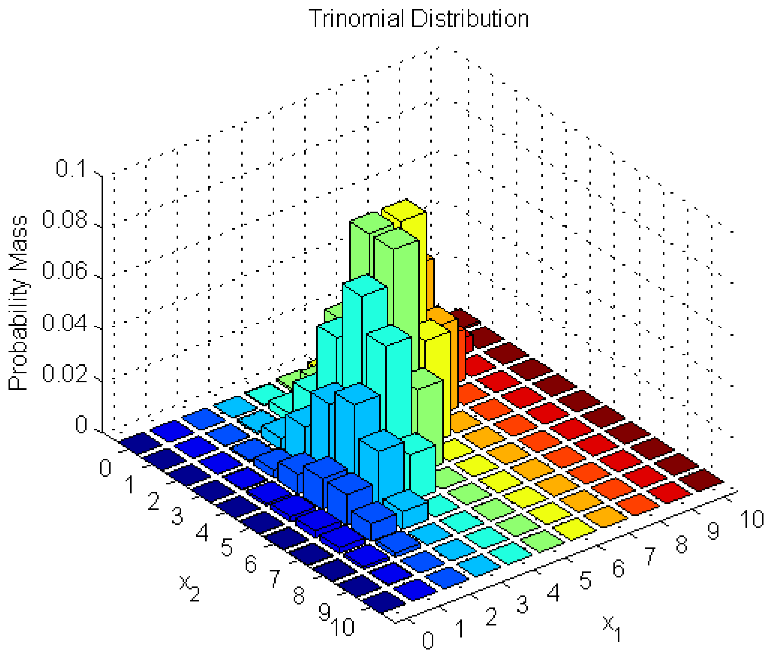
```
Y = mnpdf(X,PROB)
```

Description

`Y = mnpdf(X,PROB)` returns the pdf for the multinomial distribution with probabilities `PROB`, evaluated at each row of `X`. `X` and `PROB` are *m*-by-*k* matrices or 1-by-*k* vectors, where *k* is the number of multinomial bins or categories. Each row of `PROB` must sum to one, and the sample sizes for each observation (rows of `X`) are given by the row sums `sum(X,2)`. `Y` is an *m*-by-*k* matrix, and `mnpdf` computes each row of `Y` using the corresponding rows of the inputs, or replicates them if needed.

Example

```
% Compute the distribution
p = [1/2 1/3 1/6]; % Outcome probabilities
n = 10; % Sample size
x1 = 0:n;
x2 = 0:n;
[X1,X2] = meshgrid(x1,x2);
X3 = n-(X1+X2);
Y = mnpdf([X1(:),X2(:),X3(:)],repmat(p,(n+1)^2,1));

% Plot the distribution
Y = reshape(Y,n+1,n+1);
bar3(Y)
set(gca,'XTickLabel',0:n)
set(gca,'YTickLabel',0:n)
xlabel('x_1')
ylabel('x_2')
zlabel('Probability Mass')
```



Note that the visualization does not show x_3, which is determined by the constraint $x_1 + x_2 + x_3 = n$.

See Also

`mnrfit`, `mnrval`, `mnrnd`

Purpose

Multinomial logistic regression

Syntax

```
B = mnrfit(X,Y)
B = mnrfit(X,Y,param1,val1,param2,val2,...)
[B,dev] = mnrfit(...)
[B,dev,stats] = mnrfit(...)
```

Description

`B = mnrfit(X,Y)` returns a matrix `B` of coefficient estimates for a multinomial logistic regression of the responses in `Y` on the predictors in `X`. `X` is an n-by-p matrix of p predictors at each of n observations. `Y` is an n-by-k matrix, where `Y(i,j)` is the number of outcomes of the multinomial category `j` for the predictor combinations given by `X(i,:)`. The sample sizes for each observation are given by the row sums `sum(Y,2)`.

Alternatively, `Y` can be an n-by-1 column vector of scalar integers from 1 to k indicating the value of the response for each observation, and all sample sizes are taken to be 1.

The result `B` is a (p+1)-by-(k–1) matrix of estimates, where each column corresponds to the estimated intercept term and predictor coefficients, one for each of the first k–1 multinomial categories. The estimates for the k^{th} category are taken to be zero.

Note `mnrfit` automatically includes a constant term in all models. Do not enter a column of ones directly into `X`.

`mnrfit` treats `NaN`s in either `X` or `Y` as missing values, and ignores them.

`B = mnrfit(X,Y,param1,val1,param2,val2,...)` allows you to specify optional parameter name/value pairs to control the model fit. Parameters are:

- `'model'` — The type of model to fit; one of the text strings `'nominal'` (the default), `'ordinal'`, or `'hierarchical'`

- `'interactions'` — Determines whether the model includes an interaction between the multinomial categories and the coefficients. Specify as `'off'` to fit a model with a common set of coefficients for the predictor variables, across all multinomial categories. This is often described as *parallel regression*. Specify as `'on'` to fit a model with different coefficients across categories. In all cases, the model has different intercepts across categories. Thus, `B` is a vector containing k–1+p coefficient estimates when `'interaction'` is `'off'`, and a (p+1)-by-(k–1) matrix when it is `'on'`. The default is `'off'` for ordinal models, and `'on'` for nominal and hierarchical models.
- `'link'` — The link function to use for ordinal and hierarchical models. The link function defines the relationship $g(\mu_{ij}) = x_i b_j$ between the mean response for the i^{th} observation in the j^{th} category, μ_{ij} , and the linear combination of predictors $x_i b_j$. Specify the link parameter value as one of the text strings `'logit'`(the default), `'probit'`, `'comploglog'`, or `'loglog'`. You may not specify the `'link'` parameter for nominal models; these always use a multivariate logistic link.
- `'estdisp'` — Specify as `'on'` to estimate a dispersion parameter for the multinomial distribution in computing standard errors, or `'off'` (the default) to use the theoretical dispersion value of 1.

`[B,dev] = mnrfit(...)` returns the deviance of the fit `dev`.

`[B,dev,stats] = mnrfit(...)` returns a structure `stats` that contains the following fields:

- `dfe` — Degrees of freedom for error
- `s` — Theoretical or estimated dispersion parameter
- `sfit` — Estimated dispersion parameter
- `se` — Standard errors of coefficient estimates `B`
- `coeffcorr` — Estimated correlation matrix for `B`
- `covb` — Estimated covariance matrix for `B`

- `t` — *t* statistics for `B`
- `p` — *p*-values for `B`
- `resid` — Residuals
- `residp` — Pearson residuals
- `residd` — Deviance residuals

References

[1] McCullagh, P., J.A. Nelder, *Generalized Linear Models*, 2nd edition, Chapman & Hall/CRC Press, 1990.

See Also

`mnrval`, `glmfit`, `glmval`, `regress`, `regstats`

Purpose

Random numbers from multinomial distribution

Syntax

```
R = mnrnd(n,PROB)
R = mnrnd(n,PROB,m)
```

Description

`R = mnrnd(n,PROB)` returns random vectors chosen from the multinomial distribution with sample sizes `n` and probabilities `PROB`. `PROB` is an *m*-by-*k* matrix or a 1-by-*k* vector of multinomial probabilities, where *k* is the number of multinomial bins or categories. Each row of `PROB` must sum to one. `n` is an *m*-by-1 vector of positive integers or a positive scalar integer. `R` is an *m*-by-*k* matrix, and `mnrnd` generates each row of Y using the corresponding rows of the inputs, or replicates them if needed.

`R = mnrnd(n,PROB,m)` returns an *m*-by-*k* matrix of random vectors.

See Also

`mnpdf`, `mnrfit`, `mnrval`, `randsample`

Purpose

Values and prediction intervals for multinomial logistic regression

Syntax

```
PHAT = mnrval(B,X)
YHAT = mnrval(B,X,ssize)
[...,DLO,DHI] = mnrval(B,X,...,stats)
[...] = mnrval(...,param1,val1,param2,val2,...)
```

Description

`PHAT = mnrval(B,X)` computes predicted probabilities for the multinomial logistic regression model with predictors `X`. `B` contains intercept and coefficient estimates as returned by the `mnrfit` function. `X` is an *n*-by-*p* matrix of *p* predictors at each of *n* observations. `PHAT` is an *n*-by-*k* matrix of predicted probabilities for each multinomial category.

Note `mnrval` automatically includes a constant term in all models. Do not enter a column of ones directly into `X`.

`YHAT = mnrval(B,X,ssize)` computes predicted category counts for sample sizes `ssize`. `ssize` is an *n*-by-1 column vector of positive integers.

`[...,DLO,DHI] = mnrval(B,X,...,stats)` also computes 95% confidence bounds on the predicted probabilities `PHAT` or counts `YHAT`. `stats` is the structure returned by the `mnrfit` function. `DLO` and `DHI` define a lower confidence bound of `PHAT` or `YHAT` minus `DLO` and an upper confidence bound of `PHAT` or `YHAT` plus `DHI`. Confidence bounds are nonsimultaneous and they apply to the fitted curve, not to new observations.

`[...]  = mnrval(...,param1,val1,param2,val2,...)` allows you to specify optional parameter name/value pairs to control the predicted values. These parameters must be set to the corresponding values used with the `mnrfit` function to compute `B`. Parameters are:

- `'model'` — The type of model that was fit by `mnrfit`; one of the text strings `'nominal'` (the default), `'ordinal'`, or `'hierarchical'`.

- `'interactions'` — Determines whether the model fit by `mnrfit` included an interaction between the multinomial categories and the coefficients. The default is `'off'` for ordinal models, and `'on'` for nominal and hierarchical models.
- `'link'` — The link function that was used by `mnrfit` for ordinal and hierarchical models. Specify the link parameter value as one of the text strings `'logit'`(the default), `'probit'`, `'comploglog'`, or `'loglog'`. You may not specify the `'link'` parameter for nominal models; these always use a multivariate logistic link.
- `'type'` — Set to `'category'` (the default) to return predictions and confidence bounds for the probabilities (or counts) of the k multinomial categories. Set to `'cumulative'` to return predictions and confidence bounds for the cumulative probabilities (or counts) of the first k–1 multinomial categories, as an n-by-(k–1) matrix. The predicted cumulative probability for the kth category is 1. Set to `'conditional'` to return predictions and confidence bounds in terms of the first k–1 conditional category probabilities, i.e., the probability for category j, given an outcome in category j or higher. When `'type'` is `'conditional'`, and you supply the sample size argument `ssize`, the predicted counts at each row of `X` are conditioned on the corresponding element of `ssize`, across all categories.
- `'confidence'` — The confidence level for the confidence bounds; a value between 0 and 1. The default is 0.95.

References

[1] McCullagh, P., J.A. Nelder, *Generalized Linear Models*, 2nd edition, Chapman & Hall/CRC Press, 1990.

See Also

`mnrfit`, `glmfit`, `glmval`

Purpose

Central moment of all orders

Syntax

```
m = moment(X,order)
moment(X,order,dim)
```

Description

`m = moment(X,order)` returns the central sample moment of `X` specified by the positive integer `order`. For vectors, `moment(x,order)` returns the central moment of the specified order for the elements of `x`. For matrices, `moment(X,order)` returns central moment of the specified order for each column. For N-dimensional arrays, `moment` operates along the first nonsingleton dimension of `X`.

`moment(X,order,dim)` takes the moment along dimension `dim` of `X`.

Remarks

Note that the central first moment is zero, and the second central moment is the variance computed using a divisor of n rather than n-1, where n is the length of the vector `x` or the number of rows in the matrix `X`.

The central moment of order k of a distribution is defined as

$$m_k = E(x-\mu)^k$$

where $E(x)$ is the expected value of x.

Example

```
X = randn([6 5])
X =
  1.1650  0.0591  1.2460 -1.2704 -0.0562
  0.6268  1.7971 -0.6390  0.9846  0.5135
  0.0751  0.2641  0.5774 -0.0449  0.3967
  0.3516  0.8717 -0.3600 -0.7989  0.7562
 -0.6965 -1.4462 -0.1356 -0.7652  0.4005
  1.6961 -0.7012 -1.3493  0.8617 -1.3414

m = moment(X,3)
m =
  -0.0282  0.0571  0.1253  0.1460  -0.4486
```

See Also `kurtosis`, `mean`, `skewness`, `std`, `var`

Purpose

Multiple comparison test

Syntax

```
c = multcompare(stats)
c = multcompare(stats,param1,val1,param2,val2,...)
[c,m] = multcompare(...)
[c,m,h] = multcompare(...)
[c,m,h,gnames] = multcompare(...)
```

Description

`c = multcompare(stats)` performs a multiple comparison test using the information in the `stats` structure, and returns a matrix `c` of pairwise comparison results. It also displays an interactive graph of the estimates with comparison intervals around them. See "Examples" on page 14-494.

In a one-way analysis of variance, you compare the means of several groups to test the hypothesis that they are all the same, against the general alternative that they are not all the same. Sometimes this alternative may be too general. You may need information about which pairs of means are significantly different, and which are not. A test that can provide such information is called a *multiple comparison procedure*.

When you perform a simple t-test of one group mean against another, you specify a significance level that determines the cutoff value of the t statistic. For example, you can specify the value `alpha = 0.05` to insure that when there is no real difference, you will incorrectly find a significant difference no more than 5% of the time. When there are many group means, there are also many pairs to compare. If you applied an ordinary t-test in this situation, the `alpha` value would apply to each comparison, so the chance of incorrectly finding a significant difference would increase with the number of comparisons. Multiple comparison procedures are designed to provide an upper bound on the probability that *any* comparison will be incorrectly found significant.

The output `c` contains the results of the test in the form of a five-column matrix. Each row of the matrix represents one test, and there is one row for each pair of groups. The entries in the row indicate the means being compared, the estimated difference in means, and a confidence interval for the difference.

For example, suppose one row contains the following entries.

```
2.0000  5.0000  1.9442  8.2206  14.4971
```

These numbers indicate that the mean of group 2 minus the mean of group 5 is estimated to be 8.2206, and a 95% confidence interval for the true mean is [1.9442, 14.4971].

In this example the confidence interval does not contain 0.0, so the difference is significant at the 0.05 level. If the confidence interval did contain 0.0, the difference would not be significant at the 0.05 level.

The `multcompare` function also displays a graph with each group mean represented by a symbol and an interval around the symbol. Two means are significantly different if their intervals are disjoint, and are not significantly different if their intervals overlap. You can use the mouse to select any group, and the graph will highlight any other groups that are significantly different from it.

`c = multcompare(stats,`*`param1`*`,`*`val1`*`,`*`param2`*`,`*`val2`*`,...)` specifies one or more of the parameter name/value pairs described in the following table.

Parameter Name	Parameter Values
`'alpha'`	Scalar between `0` and `1` that determines the confidence levels of the intervals in the matrix `c` and in the figure (default is `0.05`). The confidence level is `100(1-alpha)`%.
`'displayopt'`	Either `'on'` (the default) to display a graph of the estimates with comparison intervals around them, or `'off'` to omit the graph. See "Examples" on page 14-494.
`ctype`	Specifies the type of critical value to use for the multiple comparison. "Values of `ctype`" on page 14-491 describes the allowed values for `ctype`.

Parameter Name	Parameter Values
`'dimension'`	A vector specifying the dimension or dimensions over which the population marginal means are to be calculated. Use only if you create `stats` with the function `anovan`. The default is `1` to compute over the first dimension. See “Dimension Parameter” on page 14-493 for more information.
`'estimate'`	Specifies the estimate to be compared. The allowable values of estimate depend on the function that was the source of the stats structure, as described in “Values of estimate” on page 14-493

`[c,m] = multcompare(...)` returns an additional matrix `m`. The first column of `m` contains the estimated values of the means (or whatever statistics are being compared) for each group, and the second column contains their standard errors.

`[c,m,h] = multcompare(...)` returns a handle `h` to the comparison graph. Note that the title of this graph contains instructions for interacting with the graph, and the *x*-axis label contains information about which means are significantly different from the selected mean. If you plan to use this graph for presentation, you may want to omit the title and the *x*-axis label. You can remove them using interactive features of the graph window, or you can use the following commands.

```
title('')
xlabel('')
```

`[c,m,h,gnames] = multcompare(...)` returns `gnames`, a cell array with one row for each group, containing the names of the groups.

Values of `ctype`

The following table describes the allowed values for the parameter `ctype`.

Values `ctype`	Meaning
`'hsd'` or `'tukey-kramer'`	Use Tukey's honestly significant difference criterion. This is the default, and it is based on the Studentized range distribution. It is optimal for balanced one-way ANOVA and similar procedures with equal sample sizes. It has been proven to be conservative for one-way ANOVA with different sample sizes. According to the unproven Tukey-Kramer conjecture, it is also accurate for problems where the quantities being compared are correlated, as in analysis of covariance with unbalanced covariate values.
`'lsd'`	Use Tukey's least significant difference procedure. This procedure is a simple t-test. It is reasonable if the preliminary test (say, the one-way ANOVA F statistic) shows a significant difference. If it is used unconditionally, it provides no protection against multiple comparisons.
`'bonferroni'`	Use critical values from the t distribution, after a Bonferroni adjustment to compensate for multiple comparisons. This procedure is conservative, but usually less so than the Scheffé procedure.
`'dunn-sidak'`	Use critical values from the t distribution, after an adjustment for multiple comparisons that was proposed by Dunn and proved accurate by Sidák. This procedure is similar to, but less conservative than, the Bonferroni procedure.
`'scheffe'`	Use critical values from Scheffé's S procedure, derived from the F distribution. This procedure provides a simultaneous confidence level for comparisons of all linear combinations of the means, and it is conservative for comparisons of simple differences of pairs.

Values of estimate

The allowable values of the parameter `'estimate'` depend on the function that was the source of the `stats` structure, according to the following table.

Source	Allowable Values of 'estimate'
`'anova1'`	Ignored. Always compare the group means.
`'anova2'`	Either `'column'` (the default) or `'row'` to compare column or row means.
`'anovan'`	Ignored. Always compare the population marginal means as specified by the `dim` argument.
`'aoctool'`	Either `'slope'`, `'intercept'`, or `'pmm'` to compare slopes, intercepts, or population marginal means. If the analysis of covariance model did not include separate slopes, then `'slope'` is not allowed. If it did not include separate intercepts, then no comparisons are possible.
`'friedman'`	Ignored. Always compare average column ranks.
`'kruskalwallis'`	Ignored. Always compare average group ranks.

Dimension Parameter

The dimension parameter is a vector specifying the dimension or dimensions over which the population marginal means are to be calculated. For example, if `dim = 1`, the estimates that are compared are the means for each value of the first grouping variable, adjusted by removing effects of the other grouping variables as if the design were balanced. If `dim = [1 3]`, population marginal means are computed for each combination of the first and third grouping variables, removing effects of the second grouping variable. If you fit a singular model, some cell means may not be estimable and any population marginal means that depend on those cell means will have the value `NaN`.

Population marginal means are described by Milliken and Johnson (1992) and by Searle, Speed, and Milliken (1980). The idea behind population marginal means is to remove any effect of an unbalanced design by fixing the values of the factors specified by `dim`, and averaging out the effects of other factors as if each factor combination occurred the same number of times. The definition of population marginal means does not depend on the number of observations at each factor combination. For designed experiments where the number of observations at each factor combination has no meaning, population marginal means can be easier to interpret than simple means ignoring other factors. For surveys and other studies where the number of observations at each combination does have meaning, population marginal means may be harder to interpret.

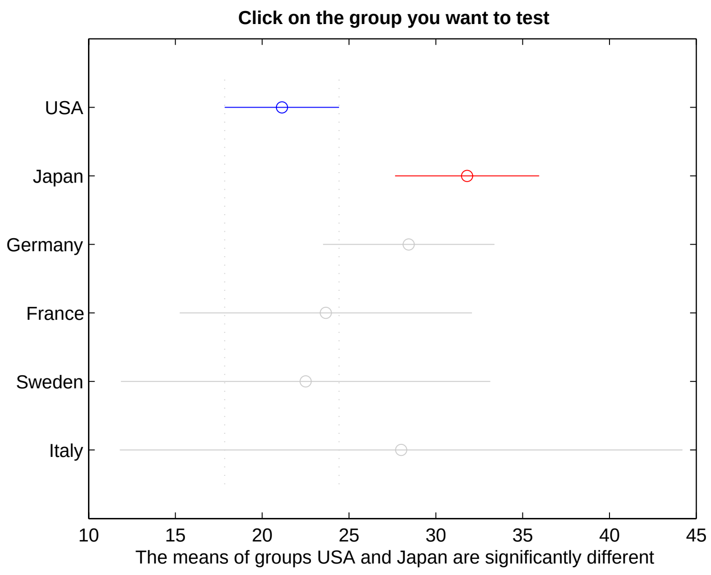
Examples

Example 1

The following example performs a 1-way analysis of variance (ANOVA) and displays group means with their names.

```
load carsmall
[p,t,st] = anova1(MPG,Origin,'off');
[c,m,h,nms] = multcompare(st,'display','off');
[nms num2cell(m)]
ans =
  'USA'     [21.1328]  [0.8814]
  'Japan'   [31.8000]  [1.8206]
  'Germany' [28.4444]  [2.3504]
  'France'  [23.6667]  [4.0711]
  'Sweden'  [22.5000]  [4.9860]
  'Italy'   [28.0000]  [7.0513]
```

`multcompare` also displays the following graph of the estimates with comparison intervals around them.



You can click the graphs of each country to compare its mean to those of other countries.

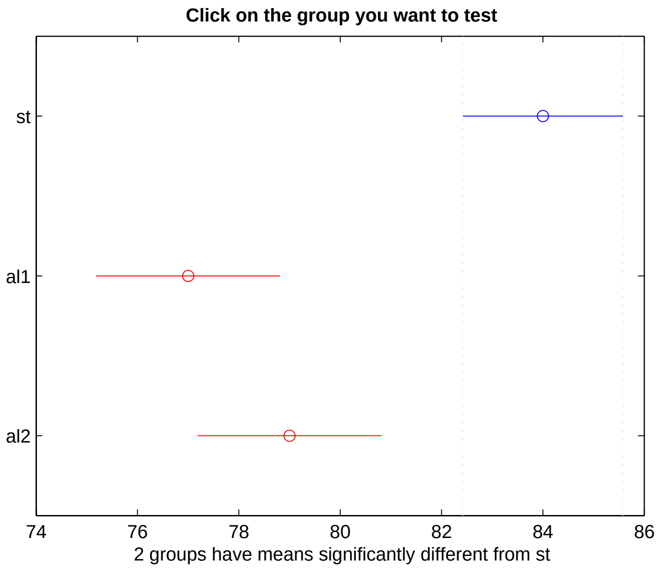
Example 2

The following continues the example described in the `anova1` reference page, which is related to testing the material strength in structural beams. From the `anova1` output you found significant evidence that the three types of beams are not equivalent in strength. Now you can determine where those differences lie. First you create the data arrays and you perform one-way ANOVA.

```
strength = [82 86 79 83 84 85 86 87 74 82 ...
            78 75 76 77 79 79 77 78 82 79];
alloy = {'st','st','st','st','st','st','st','st',...
         'al1','al1','al1','al1','al1','al1',...
         'al2','al2','al2','al2','al2','al2'};
[p,a,s] = anova1(strength,alloy);
```

Among the outputs is a structure that you can use as input to `multcompare`.

```
[c,m,h,nms] = multcompare(s);
[nms num2cell(c)]
ans =
  'st'   [1]  [2]  [ 3.6064]  [ 7]  [10.3936]
  'al1'  [1]  [3]  [ 1.6064]  [ 5]  [ 8.3936]
  'al2'  [2]  [3]  [-5.6280]  [-2]  [ 1.6280]
```



The third row of the output matrix shows that the differences in strength between the two alloys is not significant. A 95% confidence interval for the difference is [-5.6, 1.6], so you cannot reject the hypothesis that the true difference is zero.

The first two rows show that both comparisons involving the first group (steel) have confidence intervals that do not include zero. In other words, those differences are significant. The graph shows the same information.

See Also

`anova1`, `anova2`, `anovan`, `aoctool`, `friedman`, `kruskalwallis`

References

[1] Hochberg, Y., and A. C. Tamhane, *Multiple Comparison Procedures*, Wiley, 1987.

[2] Milliken, G. A., and D. E. Johnson, *Analysis of Messy Data, Volume 1: Designed Experiments*, Chapman & Hall, 1992.

[3] Searle, S. R., F. M. Speed, and G. A. Milliken, “Population marginal means in the linear model: an alternative to least squares means,” *American Statistician*, 1980, pp. 216-221.

multivarichart

Purpose

Multivari chart for grouped data

Syntax

```
multivarichart(y,GROUP)
multivarichart(Y)
multivarichart(...,param1,val1,param2,val2,...)
[charthandle,AXESH] = multivarichart(...)
```

Description

`multivarichart(y,GROUP)` displays the multivari chart for the vector `y` grouped by entries in the cell array `GROUP`. Each cell of `GROUP` must contain a grouping variable that can be a categorical variable, numeric vector, character matrix, or single-column cell array of strings. (See "Grouped Data" on page 2-41.) `GROUP` can also be a matrix whose columns represent different grouping variables. Each grouping variable must have the same number of elements as `y`. The number of grouping variables must be 2, 3, or 4.

Each subplot of the plot matrix contains a multivari chart for the first and second grouping variables. The *x*-axis in each subplot indicates values of the first grouping variable. The legend at the bottom of the figure window indicates values of the second grouping variable. The subplot at position (*i*,*j*) is the multivari chart for the subset of `y` at the *i*th level of the third grouping variable and the *j*th level of the fourth grouping variable. If the third or fourth grouping variable is absent, it is considered to have only one level.

`multivarichart(Y)` displays the multivari chart for a matrix `Y`. The data in different columns represent changes in one factor. The data in different rows represent changes in another factor.

`multivarichart(...,param1,val1,param2,val2,...)` specifies one or more of the following name/value pairs:

- `'varnames'` — Grouping variable names in a character matrix or a cell array of strings, one per grouping variable. Default names are `'X1'`, `'X2'`, ...
- `'plotorder'` — A string with the value `'sorted'` or a vector containing a permutation of the integers from 1 to the number of grouping variables.

If `'plotorder'` is a string with value `'sorted'`, the grouping variables are rearranged in descending order according to the number of levels in each variable.

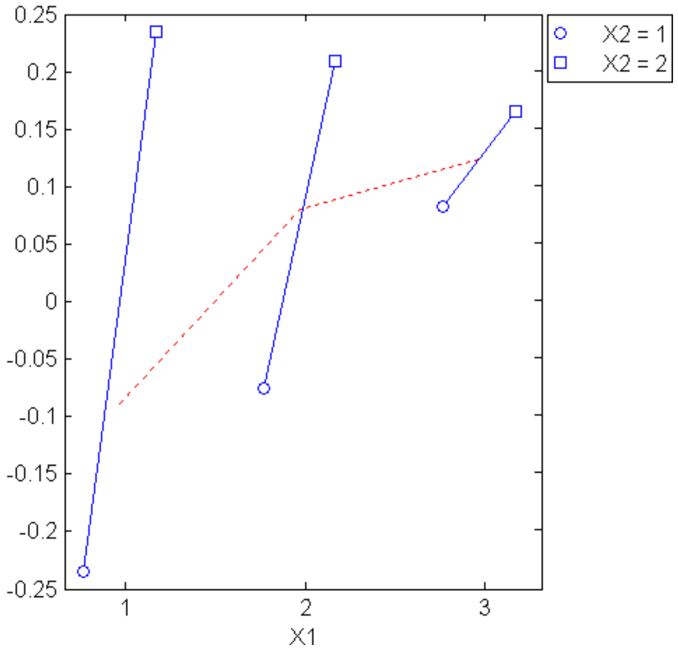
If `'plotorder'` is a vector, it indicates the order in which each grouping variable should be plotted. For example, `[2,3,1,4]` indicates that the second grouping variable should be used as the *x*-axis of each subplot, the third grouping variable should be used as the legend, the first grouping variable should be used as the columns of the plot, and the fourth grouping variable should be used as the rows of the plot.

`[charthandle,AXESH] = multivarichart(...)` returns a handle `charthandle` to the figure window and a matrix `AXESH` of handles to the subplot axes.

Example

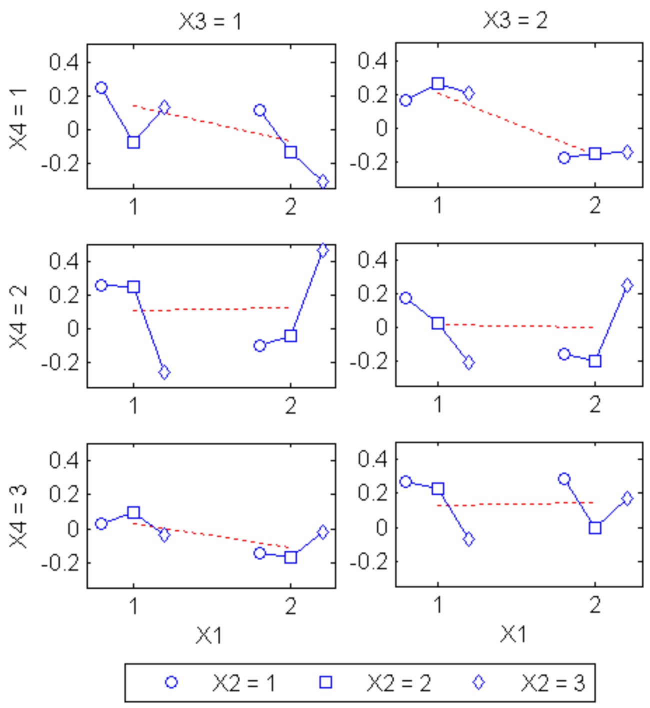
Display a multivari chart for data with two grouping variables:

```
y = randn(100,1); % response
group = [ceil(3*rand(100,1)) ceil(2*rand(100,1))];
multivarichart(y,group)
```



Display a multivari chart for data with four grouping variables:

```
y = randn(1000,1); % response
group = {ceil(2*rand(1000,1)),ceil(3*rand(1000,1)), ...
         ceil(2*rand(1000,1)),ceil(3*rand(1000,1))};
multivarichart(y,group)
```



See Also `maineffectsplot`, `interactionplot`

Purpose

Multivariate normal cumulative distribution function

Syntax

```
y = mvncdf(X)
y = mvncdf(X,mu,SIGMA)
y = mvncdf(xl,xu,mu,SIGMA)
[y,err] = mvncdf(...)
[...] = mvncdf(...,options)
```

Description

`y = mvncdf(X)` returns the cumulative probability of the multivariate normal distribution with zero mean and identity covariance matrix, evaluated at each row of `X`. Rows of the n-by-d matrix `X` correspond to observations or points, and columns correspond to variables or coordinates. `y` is an n-by-`1` vector.

`y = mvncdf(X,mu,SIGMA)` returns the cumulative probability of the multivariate normal distribution with mean `mu` and covariance `SIGMA`, evaluated at each row of `X`. `mu` is a `1`-by-d vector, and `SIGMA` is a d-by-d symmetric, positive definite matrix. `mu` can also be a scalar value, which `mvncdf` replicates to match the size of `X`. Pass in the empty matrix `[]` for `mu` to use as its default value when you want to only specify `SIGMA`.

The multivariate normal cumulative probability at `X` is defined as the probability that a random vector `V`, distributed as multivariate normal, will fall within the semi-infinite rectangle with upper limits defined by `X`, for example, `Pr{V(1)≤X(1),V(2)≤X(2),...,V(`*d*`)≤X(`*d*`)}`.

`y = mvncdf(xl,xu,mu,SIGMA)` returns the multivariate normal cumulative probability evaluated over the rectangle with lower and upper limits defined by `xl` and `xu`, respectively.

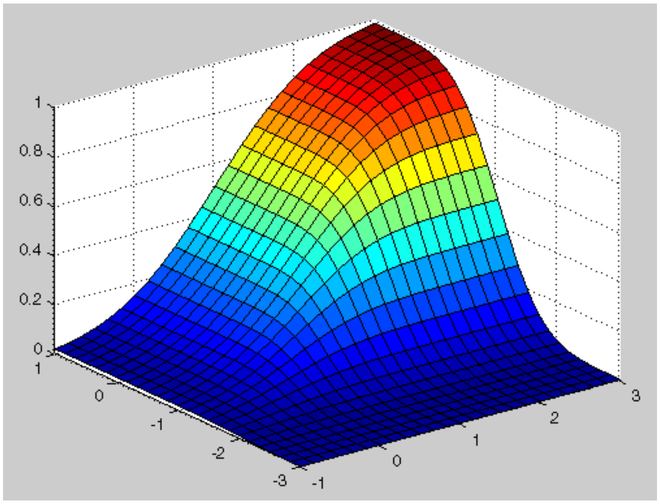
`[y,err] = mvncdf(...)` returns an estimate of the error in `y`. For bivariate and trivariate distributions, `mvncdf` uses adaptive quadrature on a transformation of the t density, based on methods developed by Drezner and Wesolowsky and by Genz, as described in the references. The default absolute error tolerance for these cases is `1e-8`. For four or more dimensions, `mvncdf` uses a quasi-Monte Carlo integration algorithm based on methods developed by Genz and Bretz, as described in the references. The default absolute error tolerance for these cases is `1e-4`.

`[...] = mvncdf(...,options)` specifies control parameters for the numerical integration used to compute `y`. This argument can be created by a call to `statset`. Choices of `statset` parameters:

- `'TolFun'` — Maximum absolute error tolerance. Default is `1e-8` when $d < 4$, or `1e-4` when $d \geq 4$.
- `'MaxFunEvals'` — Maximum number of integrand evaluations allowed when $d \geq 4$. Default is `1e7`. `'MaxFunEvals'` is ignored when $d < 4$.
- `'Display'` — Level of display output. Choices are `'off'` (the default), `'iter'`, and `'final'`. `'Display'` is ignored when $d < 4$.

Example

```
mu = [1 -1]; SIGMA = [.9 .4; .4 .3];
[X1,X2] = meshgrid(linspace(-1,3,25)',linspace(-3,1,25)');
X = [X1(:) X2(:)];
p = mvncdf(X,mu,SIGMA);
surf(X1,X2,reshape(p,25,25));
```



See Also

`mvtcdf`, `mvnpdf`, `mvnrnd`, `normcdf`

Purpose

Multivariate normal probability density function

Syntax

```
y = mvnpdf(X)
y = mvnpdf(X,MU)
y = mvnpdf(X,MU,SIGMA)
```

Description

`y = mvnpdf(X)` returns the n-by-1 vector `y`, containing the probability density of the multivariate normal distribution with zero mean and identity covariance matrix, evaluated at each row of the n-by-d matrix `X`. Rows of `X` correspond to observations and columns correspond to variables or coordinates.

`y = mvnpdf(X,MU)` returns the density of the multivariate normal distribution with mean `mu` and identity covariance matrix, evaluated at each row of `X`. `MU` is a 1-by-d vector, or an n-by-d matrix. If `MU` is a matrix, the density is evaluated for each row of `X` with the corresponding row of `MU`. `MU` can also be a scalar value, which `mvnpdf` replicates to match the size of `X`.

`y = mvnpdf(X,MU,SIGMA)` returns the density of the multivariate normal distribution with mean `MU` and covariance `SIGMA`, evaluated at each row of `X`. `SIGMA` is a d-by-d matrix, or an d-by-d-by-n array, in which case the density is evaluated for each row of `X` with the corresponding page of `SIGMA`, i.e., `mvnpdf` computes `y(i)` using `X(i,:)` and `SIGMA(:,:,i)`. Specify `[]` for `MU` to use its default value when you want to specify only `SIGMA`.

If `X` is a 1-by-d vector, `mvnpdf` replicates it to match the leading dimension of `mu` or the trailing dimension of `SIGMA`.

Example

```
mu = [1 -1];
SIGMA = [.9 .4; .4 .3];
X = mvnrnd(mu,SIGMA,10);
p = mvnpdf(X,mu,SIGMA);
```

See Also

`mvncdf`, `mvnrnd`, `normpdf`

Purpose

Multivariate linear regression

Syntax

```
beta = mvregress(X,Y)
[beta,SIGMA] = mvregress(X,Y)
[beta,SIGMA,RESID] = mvregress(X,Y)
[beta,SIGMA,RESID,COVBETA] = mvregress(...)
[beta,SIGMA,RESID,objective] = mvregress(...)
[...] = mvregress(X,Y,param1,val1,param2,val2,...)
```

Description

`beta = mvregress(X,Y)` returns a p-by-1 vector `beta` of coefficient estimates for a multivariate regression of the responses in `Y` on the predictors in `X`. `X` is an n-by-p matrix of p predictors at each of n observations. `Y` is an n-by-d vector of d-dimensional multivariate responses.

Note To include a constant term in a model, `X` should contain a column of ones.

`X` can also be a cell array of length n, with each cell containing a d-by-p design matrix for one multivariate observation. If all observations have the same d-by-p design matrix, `X` can be a single cell.

`mvregress` treats `NaN`s in `X` or `Y` as missing values. Missing values in `X` are ignored. Missing values in `Y` are handled according to the value of the `'algorithm'` parameter described below.

`[beta,SIGMA] = mvregress(X,Y)` also returns a d-by-d matrix `SIGMA` for the estimated covariance of `Y`.

`[beta,SIGMA,RESID] = mvregress(X,Y)` also returns an n-by-d matrix `RESID` of residuals.

The `RESID` values corresponding to missing values in `Y` are the differences between the conditionally imputed values for `Y` and the fitted values. The `SIGMA` estimate is not the sample covariance matrix of the `RESID` matrix.

`[beta,SIGMA,RESID,COVBETA] = mvregress(...)` also returns a matrix `COVBETA` for the estimated covariance the coefficients. By default, or if the `'varformat'` parameter is `'beta'` (see below), `COVBETA` is the estimated covariance matrix of `beta`. If the `'varformat'` parameter is `'full'`, `COVBETA` is the combined estimated covariance matrix for `beta` and `SIGMA`.

`[beta,SIGMA,RESID,objective] = mvregress(...)` also returns the value of the objective function, or log likelihood, `objective`, after the last iteration.

`[...]  = mvregress(X,Y,`*`param1`*`,`*`val1`*`,`*`param2`*`,`*`val2`*`,...)` specifies additional parameter name/value pairs chosen from the following:

- `'algorithm'` — Either `'ecm'` to compute the maximum likelihood estimates via the ECM algorithm, `'cwls'` to perform least squares (optionally conditionally weighted by an input covariance matrix), or `'mvn'` to omit observations with missing data and compute the ordinary multivariate normal estimates. Default is `'mvn'` for complete data, `'ecm'` for missing data when the sample size is sufficient to estimate all parameters, and `'cwls'` otherwise.
- `'maxiter'` — Maximum number of iterations. Default is 100.
- `'tolbeta'` — Convergence tolerance for `beta`. Default is `sqrt(eps)`. Iterations continue until the `tolbeta` and `tolobj` conditions are met. The test for convergence at iteration `k` is

  ```
  norm(beta(k)-beta(k-1)) <
  sqrt(p)*tolbeta*(1+norm(beta(k)))
  ```

  where `p = length(beta)`.
- `'tolobj'` — Convergence tolerance for changes in the objective function. Default is `eps^(3/4)`. The test is

  ```
  abs(obj(k)-obj(k-1)) < tolobj*(1+abs(obj(k)))
  ```

  where `obj` is the objective function. If both `tolobj` and `tolbeta` are `0`, the function performs `maxiter` iterations with no convergence test.

- `'param0'` — A vector of p elements to be used as the initial estimate for `beta`. Default is a zero vector. Not used for the `'mvn'` algorithm.
- `'covar0'` — A d-by-d matrix to be used as the initial estimate for `SIGMA`. Default is the identity matrix. For the `'cwls'` algorithm, this matrix is usually a diagonal matrix used as a weighting at each iteration. The `'cwls'` algorithm uses the initial value of SIGMA at each iteration, without changing it.
- `'outputfcn'` — An output function called with three arguments:

  1. A vector of current parameter estimates.

  2. A structure with fields `'Covar'` for the current value of the covariance matrix, `'iteration'` for the current iteration number, and `'fval'` for the current value of the objective function.

  3. A text string that is `'init'` when called during initialization, `'iter'` when called after an iteration, and `'done'` when called after completion.
- `'varformat'` — Either `'beta'` to compute `COVBETA` for `beta` only (default), or `'full'` to compute `COVBETA` for both `b` and `SIGMA`.
- `'vartype'` — Either `'hessian'` to compute `COVBETA` using the Hessian or observed information (default), or `'fisher'` to compute `COVBETA` using the complete-data Fisher or expected information. The `'hessian'` method takes into account the increased uncertainties due to missing data, while the `'fisher'` method does not.

References

[1] Little, R. J. A., D. B. Rubin, *Statistical Analysis with Missing Data*, 2nd ed., John Wiley & Sons, Inc., 2002.

[2] Meng, X., D. B. Rubin, “Maximum Likelihood Estimation via the ECM Algorithm,” *Biometrika*, Vol. 80, No. 2, 1993, pp. 267–278.

[3] Sexton, J., A. R. Swensen, “ECM Algorithms that Converge at the Rate of EM,” *Biometrika*, Vol. 87, No. 3, 2000, pp. 651–662.

[4] Dempster, A. P., N.M. Laird, D. B. Rubin, "Maximum Likelihood from Incomplete Data via the EM Algorithm," *Journal of the Royal Statistical Society*, Series B, Vol. 39, No. 1, 1977, pp. 1–37.

See Also

`mvregresslike`, `regstats`, `manova1`

Purpose

Negative log-likelihood for multivariate regression

Syntax

```
nlogL = mvregresslike(X,Y,beta,SIGMA,alg)
[nlogL,COVBETA] = mvregresslike(...)
[nlogL,COVBETA] = mvregresslike(...,type,format)
```

Description

`nlogL = mvregresslike(X,Y,beta,SIGMA,alg)` computes the negative log-likelihood `nlogL` for a multivariate regression of the d-dimensional multivariate observations in the n-by-d matrix `Y` on the predictor variables in the matrix or cell array `X`, evaluated for the p-by-1 column vector `beta` of coefficient estimates and the d-by-d matrix `SIGMA` specifying the covariance of a row of `Y`. If $d = 1$, `X` can be an n-by-p design matrix of predictor variables. For any value of d, `X` can also be a cell array of length n, with each cell containing a d-by-p design matrix for one multivariate observation. If all observations have the same d-by-p design matrix, `X` can be a single cell.

`NaN` values in `X` or `Y` are taken as missing. Observations with missing values in `X` are ignored. Treatment of missing values in `Y` depends on the algorithm specified by *`alg`*.

`alg` should match the algorithm used by `mvregress` to obtain the coefficient estimates `beta`, and must be one of the following:

- `'ecm'` — ECM algorithm
- `'cwls'` — Least squares conditionally weighted by `SIGMA`
- `'mvn'` — Multivariate normal estimates computed after omitting rows with any missing values in `Y`

`[nlogL,COVBETA] = mvregresslike(...)` also returns an estimated covariance matrix `COVBETA` of the parameter estimates `beta`.

`[nlogL,COVBETA] = mvregresslike(...,type,format)` specifies the type and format of `COVBETA`.

`type` is either:

- `'hessian'` — To use the Hessian or observed information. This method takes into account the increased uncertainties due to missing data. This is the default.
- `'fisher'` — To use the Fisher or expected information. This method uses the complete data expected information, and does not include uncertainty due to missing data.

`format` is either:

- `'beta'` — To compute `COVBETA` for `beta` only. This is the default.
- `'full'` — To compute `COVBETA` for both `beta` and `SIGMA`.

See Also

`mvregress`, `regstats`, `manova1`

Purpose

Random numbers from multivariate normal distribution

Syntax

```
R = mvnrnd(MU,SIGMA)
r = mvnrnd(MU,SIGMA,cases)
```

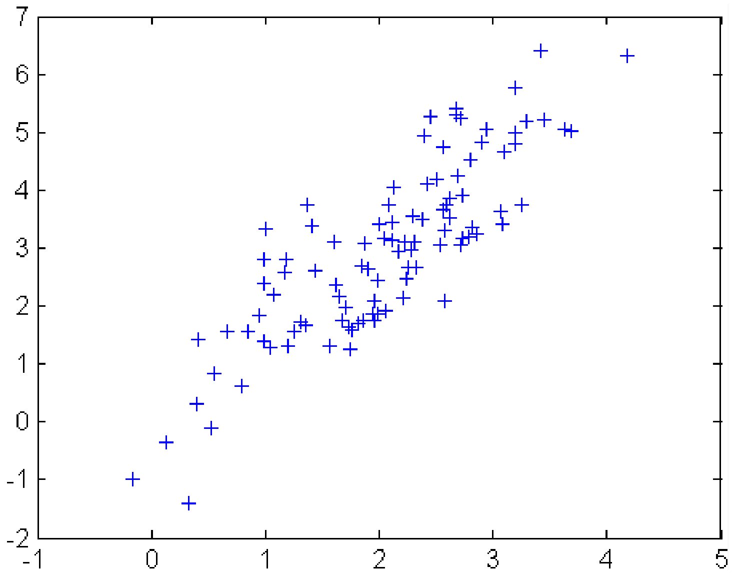
Description

`R = mvnrnd(MU,SIGMA)` returns an n-by-d matrix `R` of random vectors chosen from the multivariate normal distribution with mean `MU`, and covariance `SIGMA`. `MU` is an n-by-d matrix, and `mvnrnd` generates each row of `R` using the corresponding row of `mu`. `SIGMA` is a d-by-d symmetric positive semi-definite matrix, or a d-by-d-by-n array. If `SIGMA` is an array, `mvnrnd` generates each row of `R` using the corresponding page of `SIGMA`, i.e., `mvnrnd` computes `R(i,:)` using `MU(i,:)` and `SIGMA(:,:,i)`. If `MU` is a `1`-by-d vector, `mvnrnd` replicates it to match the trailing dimension of `SIGMA`.

`r = mvnrnd(MU,SIGMA,cases)` returns a `cases`-by-d matrix `R` of random vectors chosen from the multivariate normal distribution with a common `1-by-d` mean vector `MU`, and a common `d-by-d` covariance matrix `SIGMA`.

Example

```
mu = [2 3];
SIGMA = [1 1.5; 1.5 3];
r = mvnrnd(mu,SIGMA,100);
plot(r(:,1),r(:,2),'+')
```



See Also `lhsnorm`, `mvncdf`, `mvnpdf`, `normrnd`

Purpose

Multivariate *t* cumulative distribution function

Syntax

```
y = mvtcdf(X,C,DF)
y = mvtcdf(xl,xu,C,DF)
[y,err] = mvtcdf(...)
[...] = mvntdf(...,options)
```

Description

`y = mvtcdf(X,C,DF)` returns the cumulative probability of the multivariate *t* distribution with correlation parameters `C` and degrees of freedom `DF`, evaluated at each row of `X`. Rows of the *n*-by-*d* matrix `X` correspond to observations or points, and columns correspond to variables or coordinates. `y` is an `n-by-1` vector.

`C` is a symmetric, positive definite, *d*-by-*d* matrix, typically a correlation matrix. If its diagonal elements are not 1, `mvtcdf` scales `C` to correlation form. `DF` is a scalar, or a vector with *n* elements.

The multivariate *t* cumulative probability at `X` is defined as the probability that a random vector `T`, distributed as multivariate *t*, will fall within the semi-infinite rectangle with upper limits defined by `X`, i.e., `Pr{T(1)≤X(1),T(2)≤X(2),...T(`*d*`)≤X(`*d*`)}`.

`y = mvtcdf(xl,xu,C,DF)` returns the multivariate *t* cumulative probability evaluated over the rectangle with lower and upper limits defined by `xl` and `xu`, respectively.

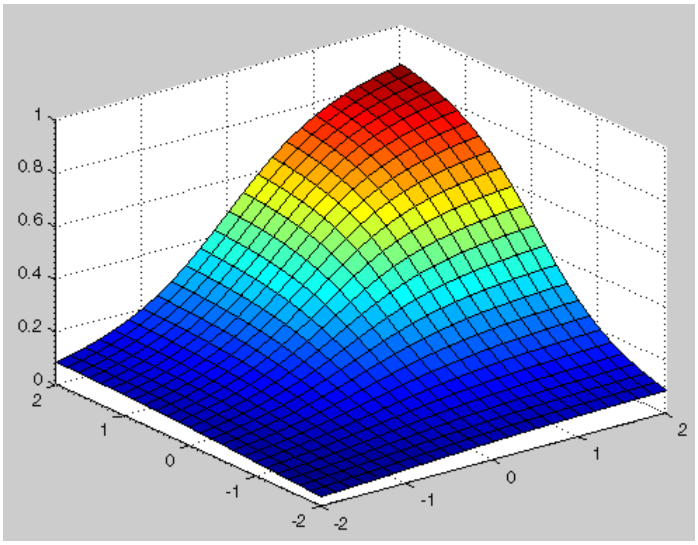
`[y,err] = mvtcdf(...)` returns an estimate of the error in `y`. For bivariate and trivariate distributions, `mvtcdf` uses adaptive quadrature on a transformation of the *t* density, based on methods developed by Genz, as described in the references. The default absolute error tolerance for these cases is `1e-8`. For four or more dimensions, `mvtcdf` uses a quasi-Monte Carlo integration algorithm based on methods developed by Genz and Bretz, as described in the references. The default absolute error tolerance for these cases is `1e-4`.

`[...] = mvntdf(...,options)` specifies control parameters for the numerical integration used to compute `y`. This argument can be created by a call to `statset`. Choices of `statset` parameters are:

- `'TolFun'` — Maximum absolute error tolerance. Default is `1e-8` when $d < 4$, or `1e-4` when $d \geq 4$.
- `'MaxFunEvals'` — Maximum number of integrand evaluations allowed when $d \geq 4$. Default is `1e7`. `'MaxFunEvals'` is ignored when $d < 4$.
- `'Display'` — Level of display output. Choices are `'off'` (the default), `'iter'`, and `'final'`. `'Display'` is ignored when $d < 4$.

Example

```
C = [1 .4; .4 1]; df = 2;
[X1,X2] = meshgrid(linspace(-2,2,25)',linspace(-2,2,25)');
X = [X1(:) X2(:)];
p = mvtcdf(X,C,df);
surf(X1,X2,reshape(p,25,25));
```



See Also

`mvncdf`, `mvnrnd`, `mvtrnd`, `tcdf`

Purpose

Multivariate *t* probability density function

Syntax

```
y = mvtpdf(X,C,df)
```

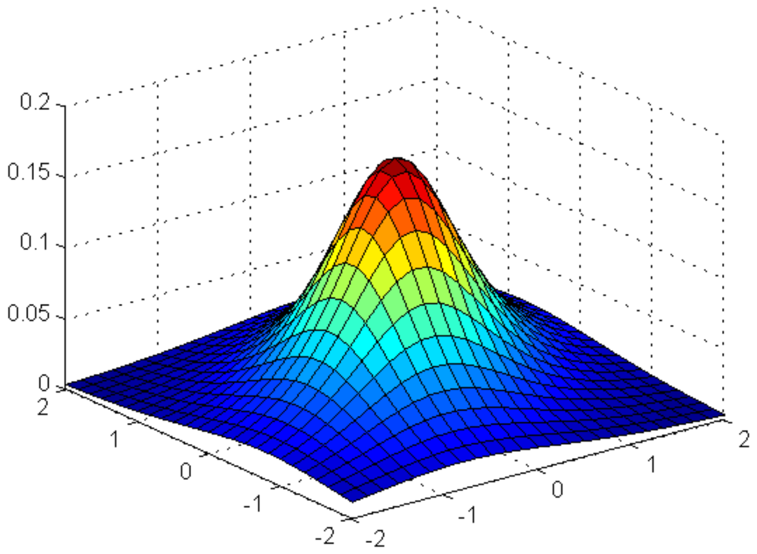
Description

`y = mvtpdf(X,C,df)` returns the probability density of the multivariate *t* distribution with correlation parameters `C` and degrees of freedom `df`, evaluated at each row of `X`. Rows of the *n*-by-*d* matrix `X` correspond to observations or points, and columns correspond to variables or coordinates. `C` is a symmetric, positive definite, *d*-by-*d* matrix, typically a correlation matrix. If its diagonal elements are not 1, `mvtpdf` scales `C` to correlation form. `df` is a scalar, or a vector with *n* elements. `y` is an *n*-by-1 vector.

Example

Visualize a multivariate *t* distribution:

```
[X1,X2] = meshgrid(linspace(-2,2,25)',linspace(-2,2,25)');
X = [X1(:) X2(:)];
C = [1 .4; .4 1];
df = 2;
p = mvtpdf(X,C,df);
surf(X1,X2,reshape(p,25,25));
```



See Also `mvtcdf`, `mvtrnd`, `tpdf`, `mvnpdf`

Purpose

Random numbers from multivariate *t* distribution

Syntax

```
R = mvtrnd(C,df,cases)
R = mvtrnd(C,df)
```

Description

`R = mvtrnd(C,df,cases)` returns a matrix of random numbers chosen from the multivariate *t* distribution, where `C` is a correlation matrix. `df` is the degrees of freedom and is either a scalar or is a vector with `cases` elements. If `p` is the number of columns in `C`, then the output `R` has `cases` rows and `p` columns.

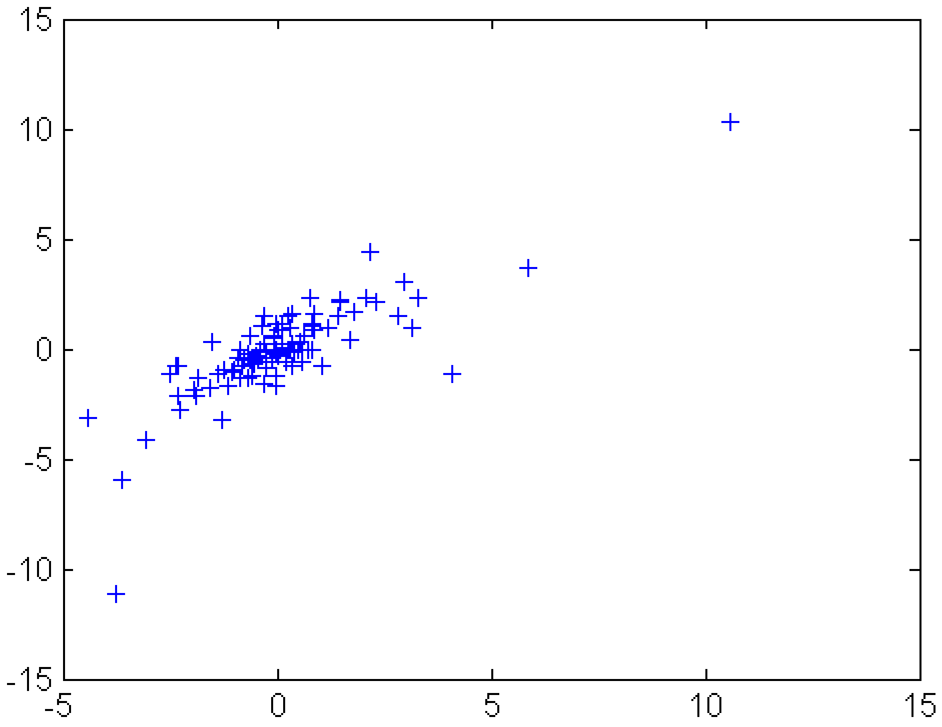
Let `t` represent a row of `R`. Then the distribution of `t` is that of a vector having a multivariate normal distribution with mean 0, variance 1, and covariance matrix `C`, divided by an independent chi-square random value having `df` degrees of freedom. The rows of `R` are independent.

C must be a square, symmetric and positive definite matrix. If its diagonal elements are not all 1 (that is, if `C` is a covariance matrix rather than a correlation matrix), `mvtrnd` computes the equivalent correlation matrix before generating the random numbers.

`R = mvtrnd(C,df)` returns a single random number from the multivariate *t* distribution.

Example

```
SIGMA = [1 0.8;0.8 1];
R = mvtrnd(SIGMA,3,100);
plot(R(:,1),R(:,2),'+')
```



See Also `mvtcdf`, `mvnrnd`, `tcdf`

Purpose

Covariance matrix, ignoring NaNs

Syntax

```
C = nancov(X)
C = nancov(X,Y)
C = nancov(X)
nancov
C = nancov(X,1)
C = nancov(...,'pairwise')
```

Description

`C = nancov(X)`, where `X` is a vector, returns the sample variance of the values in `X`, treating `NaN`s as missing values. If `X` is a matrix, in which each row is an observation and each column a variable, `nancov(X)` is the covariance matrix computed using rows of `X` that do not contain any `NaN` values.

`C = nancov(X,Y)`, where `X` and `Y` are matrices with the same number of elements, is equivalent to `nancov([X(:)  Y(:)])`.

`C = nancov(X)` or `nancov(X,Y)` normalizes the result by $n - 1$ if $n > 0$, where n is the number of observations after removing missing values. This makes `nancov(X)` the best unbiased estimate of the covariance matrix if the observations are from a normal distribution. For $n = 1$ `nancov` normalizes by n.

`C = nancov(X,1)` or `nancov(X,Y,1)` normalizes the result by n. That is, it returns the second moment matrix of the observations about their mean. `nancov(X,Y,0)` is the same as `nancov(X,Y)`, and `nancov(X,0)` is the same as `nancov(X)`.

`C = nancov(...,'pairwise')` computes `C(i,j)` using rows with no `NaN` values in columns `i` or `j`. The result may not be a positive definite matrix. `C = nancov(...,'complete')` is the default, and it omits rows with any `NaN` values, even if they are not in column `i` or `j`.

The mean is removed from each column before calculating the result.

Example

The following example generates random data having nonzero covariance between column 4 and the other columns.

```
X = randn(30,4);     % Uncorrelated data
X(:,4) = sum(X,2);  % Introduce correlation
X(2,3) = NaN;       % Introduce one missing value
C = nancov(X)       % Compute sample covariance
```

See Also

`cov`, `var`, `nanvar`

Purpose

Maximum, ignoring NaNs

Syntax

```
M = nanmax(A)
M = nanmax(A,B)
M = nanmax(A,[],dim)
[M,ndx] = nanmax(...)
```

Description

M = nanmax(A) returns the maximum with NaNs treated as missing. For vectors, nanmax(A) is the largest non-NaN element in A. For matrices, nanmax(A) is a row vector containing the maximum non-NaN element from each column. For N-dimensional arrays, nanmax operates along the first nonsingleton dimension of X.

M = nanmax(A,B) returns an array of the same size as A and B, each of whose entries is the maximum of the corresponding entries of A or B. A scalar input is expanded to an array of the same size as the other input.

M = nanmax(A,[],dim) operates along the dimension dim of X.

[M,ndx] = nanmax(...) also returns the indices of the maximum values in the vector ndx.

Example

```
A = magic(3);
A([1 6 8]) = [NaN NaN NaN]
A =
  NaN   1   6
   3   5  NaN
   4  NaN   2

[nmax,maxidx] = nanmax(A)
nmax =
   4   5   6
maxidx =
   3   2   1
```

See Also

nanmin, nanmean, nanmedian, nanstd, nansum

nanmean

Purpose

Mean, ignoring NaNs

Syntax

```
y = nanmean(X)
y = nanmean(X,dim)
```

Description

`y = nanmean(X)` is the mean computed by treating NaNs as missing values.

For vectors, `nanmean(x)` is the mean of the non-NaN elements of `x`. For matrices, `nanmean(X)` is a row vector containing the mean of the non-NaN elements in each column. For N-dimensional arrays, `nanmean` operates along the first nonsingleton dimension of `X`.

`y = nanmean(X,dim)` takes the mean along dimension `dim` of `X`.

Example

```
m = magic(3);
m([1 6 8]) = [NaN NaN NaN]
m =
  NaN   1   6
   3   5  NaN
   4  NaN   2

nmean = nanmean(m)
nmean =
  3.5000  3.0000  4.0000
```

See Also

`nanmin`, `nanmax`, `nanmedian`, `nanstd`, `nansum`

Purpose

Median, ignoring NaNs

Syntax

```
y = nanmedian(X)
y = nanmedian(X,dim)
```

Description

`y = nanmedian(X)` is the median computed by treating NaNs as missing values.

For vectors, `nanmedian(x)` is the median of the non-NaN elements of `x`. For matrices, `nanmedian(X)` is a row vector containing the median of the non-NaN elements in each column of `X`. For N-dimensional arrays, `nanmedian` operates along the first nonsingleton dimension of `X`.

`y = nanmedian(X,dim)` takes the median along the dimension `dim` of `X`.

Example

```
m = magic(4);
m([1 6 9 11]) = [NaN NaN NaN NaN]
m =
  NaN   2  NaN  13
   5  NaN  10   8
   9   7  NaN  12
   4  14  15   1

nmedian = nanmedian(m)
nmedian =
  5.0000  7.0000  12.5000  10.0000
```

See Also

`nanmin`, `nanmax`, `nanmean`, `nanstd`, `nansum`

nanmin

Purpose

Minimum, ignoring NaNs

Syntax

```
M = nanmin(A)
M = nanmin(A,B)
M = nanmin(A,[],dim)
[M,ndx] = nanmin(...)
```

Description

`M = nanmin(A)` is the minimum computed by treating `NaN`s as missing values. For vectors, `M` is the smallest non-`NaN` element in `A`. For matrices, `M` is a row vector containing the minimum non-`NaN` element from each column. For N-dimensional arrays, `nanmin` operates along the first nonsingleton dimension of `X`.

`M = nanmin(A,B)` returns an array of the same size as `A` and `B`, each of whose entries is the minimum of the corresponding entries of `A` or `B`. A scalar input is expanded to an array of the same size as the other input.

`M = nanmin(A,[],dim)` operates along the dimension `dim` of `X`.

`[M,ndx] = nanmin(...)` also returns the indices of the minimum values in vector `ndx`.

Example

```
A = magic(3);
A([1 6 8]) = [NaN NaN NaN]
A =
  NaN   1   6
    3   5  NaN
    4  NaN   2

[nmin,minidx] = nanmin(A)
nmin =
    3   1   2
minidx =
    2   1   3
```

See Also

`nanmax`, `nanmean`, `nanmedian`, `nanstd`, `nansum`

Purpose

Standard deviation, ignoring NaNs

Syntax

```
Y = nanstd(X)
Y = nanstd(X,1)
nanstd(X,0)
Y = nanstd(X,flag,dim)
```

Description

`Y = nanstd(X)` is the standard deviation computed by treating NaNs as missing values. For vectors, `nanstd(X)` is the standard deviation of the non-NaN elements of `X`. For matrices, `nanstd(X)` is a row vector containing the standard deviations of the non-NaN elements in each column of `X`. For N-dimensional arrays, `nanstd` operates along the first nonsingleton dimension of `X`.

`Y = nanstd` normalizes `Y` by $n - 1$, where n is the sample size. The result `Y` is the square root of an unbiased estimator of the variance of the population from which `X` is drawn, as long as `X` consists of independent, identically distributed samples, and data are missing at random.

`Y = nanstd(X,1)` normalizes `Y` by n. The result `Y` is the square root of the second moment of the sample about its mean. `nanstd(X,0)` is the same as `nanstd(X)`.

`Y = nanstd(X,flag,dim)` takes the standard deviation along the dimension `dim` of `X`. Set `flag` to `0` to normalize the result by $n - 1$; set `flag` to 1 to normalize the result by n.

Example

```
m = magic(3);
m([1 6 8]) = [NaN NaN NaN]
m =
  NaN   1   6
   3   5  NaN
   4  NaN   2

nstd = nanstd(m)
nstd =
  0.7071  2.8284  2.8284
```

See Also nanmax, nanmin, nanmean, nanmedian, nansum

Purpose

Sum, ignoring NaNs

Syntax

```
y = nansum(X)
Y = nansum(X,dim)
```

Description

`y = nansum(X)` is the sum computed by treating `NaN`s as missing values.

For vectors, `nansum(x)` is the sum of the non-`NaN` elements of `x`. For matrices, `nansum(X)` is a row vector containing the sum of the non-`NaN` elements in each column of `X`. For N-dimensional arrays, `nansum` operates along the first nonsingleton dimension of `X`.

`Y = nansum(X,dim)` takes the sum along dimension `dim` of `X`.

Example

```
m = magic(3);
m([1 6 8]) = [NaN NaN NaN]
m =
  NaN   1   6
    3   5  NaN
    4  NaN   2

nsum = nansum(m)
nsum =
    7   6   8
```

See Also

`nanmax`, `nanmin`, `nanmean`, `nanmedian`, `nanstd`

nanvar

Purpose

Variance, ignoring NaNs

Syntax

```
Y = nanvar(X)
Y = nanvar(X,1)
Y = nanvar(X,w)
Y = nanvar(X,w,dim)
```

Description

`Y = nanvar(X)` returns the sample variance of the values in `X`, treating `NaN`s as missing values. For a vector input, `Y` is the variance of the non-`NaN` elements of `X`. For a matrix input, `Y` is a row vector containing the variance of the non-`NaN` elements in each column of `X`. For `N`-dimensional arrays, `nanvar` operates along the first nonsingleton dimension of `X`.

`nanvar` normalizes `Y` by $n - 1$ if $n > 1$, where n is the sample size of the non-`NaN` elements. The result, `Y`, is an unbiased estimator of the variance of the population from which `X` is drawn, as long as `X` consists of independent, identically distributed samples, and data are missing at random. For $n = 1$, `Y` is normalized by N.

`Y = nanvar(X,1)` normalizes `Y` by n. The result Y is the second moment of the sample about its mean. `nanvar(X,0)` is the same as `nanvar(X)`.

`Y = nanvar(X,w)` computes the variance using the weight vector `w`. The length of `w` must equal the length of the dimension over which `nanvar` operates, and its elements must be nonnegative. Elements of `X` corresponding to `NaN` elements of `w` are ignored.

`Y = nanvar(X,w,dim)` takes the variance along the dimension `dim` of `X`. Set `w` to `[]` to use the default normalization by $n - 1$.

See Also

`var`, `nanstd`, `nanmean`, `nanmedian`, `nanmin`, `nanmax`, `nansum`

Purpose

Negative binomial cumulative distribution function

Syntax

```
Y = nbincdf(X,R,P)
```

Description

`Y = nbincdf(X,R,P)` computes the negative binomial cdf at each of the values in `X` using the corresponding parameters in `R` and `P`. `X`, `R`, and `P` can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of `Y`. A scalar input for `X`, `R`, or `P` is expanded to a constant array with the same dimensions as the other inputs.

The negative binomial cdf is

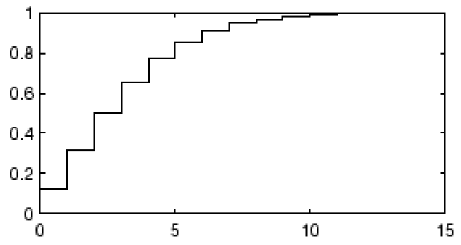
$$y = F(x|r,p) = \sum_{i=0}^{x} \binom{r+i-1}{i} p^r q^i I_{(0,1,\ldots)}(i)$$

The simplest motivation for the negative binomial is the case of successive random trials, each having a constant probability `P` of success. The number of *extra* trials you must perform in order to observe a given number `R` of successes has a negative binomial distribution. However, consistent with a more general interpretation of the negative binomial, `nbincdf` allows `R` to be any positive value, including nonintegers. When `R` is noninteger, the binomial coefficient in the definition of the cdf is replaced by the equivalent expression

$$\frac{\Gamma(r+i)}{\Gamma(r)\Gamma(i+1)}$$

Example

```
x = (0:15);
p = nbincdf(x,3,0.5);
stairs(x,p)
```



See Also `cdf`, `nbinfit`, `nbininv`, `nbinpdf`, `nbinrnd`, `nbinstat`

Purpose

Parameter estimates and confidence intervals for negative binomial distributed data

Syntax

```
parmhat = nbinfit(data)
[parmhat,parmci] = nbinfit(data,alpha)
[...] = nbinfit(data,alpha,options)
```

Description

`parmhat = nbinfit(data)` returns the maximum likelihood estimates (MLEs) of the parameters of the negative binomial distribution given the data in the vector `data`.

`[parmhat,parmci] = nbinfit(data,alpha)` returns MLEs and `100(1-alpha)` percent confidence intervals. By default, `alpha = 0.05`, which corresponds to 95% confidence intervals.

`[...]  = nbinfit(data,alpha,options)` accepts a structure, `options`, that specifies control parameters for the iterative algorithm the function uses to compute maximum likelihood estimates. The negative binomial fit function accepts an `options` structure which you can create using the function `statset`. Enter `statset('nbinfit')` to see the names and default values of the parameters that `nbinfit` accepts in the `options` structure. See the reference page for `statset` for more information about these options.

Note The variance of a negative binomial distribution is greater than its mean. If the sample variance of the data in `data` is less than its sample mean, `nbinfit` cannot compute MLEs. You should use the `poissfit` function instead.

See Also

`nbincdf`, `nbininv`, `nbinpdf`, `nbinrnd`, `nbinstat`, `mle`, `statset`

nbininv

Purpose

Inverse of negative binomial cumulative distribution function

Syntax

```
X = nbininv(Y,R,P)
```

Description

`X = nbininv(Y,R,P)` returns the inverse of the negative binomial cdf with parameters `R` and `P` at the corresponding probabilities in `P`. Since the binomial distribution is discrete, `nbininv` returns the least integer `X` such that the negative binomial cdf evaluated at `X` equals or exceeds `Y`. `Y`, `R`, and `P` can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of `X`. A scalar input for `Y`, `R`, or `P` is expanded to a constant array with the same dimensions as the other inputs.

The simplest motivation for the negative binomial is the case of successive random trials, each having a constant probability `P` of success. The number of *extra* trials you must perform in order to observe a given number `R` of successes has a negative binomial distribution. However, consistent with a more general interpretation of the negative binomial, `nbininv` allows `R` to be any positive value, including nonintegers.

Example

How many times would you need to flip a fair coin to have a 99% probability of having observed 10 heads?

```
flips = nbininv(0.99,10,0.5) + 10
flips =
  33
```

Note that you have to flip at least 10 times to get 10 heads. That is why the second term on the right side of the equals sign is a 10.

See Also

`icdf`, `nbincdf`, `nbinfit`, `nbinpdf`, `nbinrnd`, `nbinstat`

Purpose

Negative binomial probability density function

Syntax

```
Y = nbinpdf(X,R,P)
```

Description

`Y = nbinpdf(X,R,P)` returns the negative binomial pdf at each of the values in `X` using the corresponding parameters in `R` and `P`. `X`, `R`, and `P` can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of `Y`. A scalar input for `X`, `R`, or `P` is expanded to a constant array with the same dimensions as the other inputs. Note that the density function is zero unless the values in `X` are integers.

The negative binomial pdf is

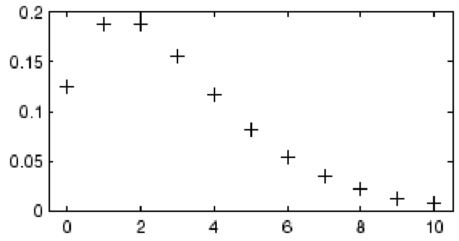
$$y = f(x|r,p) = \binom{r+x-1}{x} p^r q^x I_{(0,1,\ldots)}(x)$$

The simplest motivation for the negative binomial is the case of successive random trials, each having a constant probability `P` of success. The number of *extra* trials you must perform in order to observe a given number `R` of successes has a negative binomial distribution. However, consistent with a more general interpretation of the negative binomial, `nbinpdf` allows `R` to be any positive value, including nonintegers. When `R` is noninteger, the binomial coefficient in the definition of the pdf is replaced by the equivalent expression

$$\frac{\Gamma(r+x)}{\Gamma(r)\Gamma(x+1)}$$

Example

```
x = (0:10);
y = nbinpdf(x,3,0.5);
plot(x,y,'+')
set(gca,'Xlim',[-0.5,10.5])
```



See Also `nbincdf`, `nbinfit`, `nbininv`, `nbinrnd`, `nbinstat`, `pdf`

Purpose

Random numbers from negative binomial distribution

Syntax

```
RND = nbinrnd(R,P)
RND = nbinrnd(R,P,m)
RND = nbinrnd(R,P,m,n)
```

Description

`RND = nbinrnd(R,P)` is a matrix of random numbers chosen from a negative binomial distribution with parameters `R` and `P`. `R` and `P` can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of `RND`. A scalar input for `R` or `P` is expanded to a constant array with the same dimensions as the other input.

`RND = nbinrnd(R,P,m)` generates random numbers with parameters `R` and `P`, where `v` is a row vector. If `v` is a 1-by-2 vector, `R` is a matrix with `v(1)` rows and `v(2)` columns. If `v` is 1-by-n, `R` is an n-dimensional array.

`RND = nbinrnd(R,P,m,n)` generates random numbers with parameters `R` and `P`, where scalars m and n are the row and column dimensions of `RND`.

The simplest motivation for the negative binomial is the case of successive random trials, each having a constant probability `P` of success. The number of *extra* trials you must perform in order to observe a given number `R` of successes has a negative binomial distribution. However, consistent with a more general interpretation of the negative binomial, `nbinrnd` allows `R` to be any positive value, including nonintegers.

Example

Suppose you want to simulate a process that has a defect probability of 0.01. How many units might Quality Assurance inspect before finding three defective items?

```
r = nbinrnd(3,0.01,1,6)+3
r =
  496  142  420  396  851  178
```

See Also

`nbincdf`, `nbinfit`, `nbininv`, `nbinpdf`, `nbinstat`

Purpose

Mean and variance of negative binomial distribution

Syntax

```
[M,V] = nbinstat(R,P)
```

Description

`[M,V] = nbinstat(R,P)` returns the mean of and variance for the negative binomial distribution with parameters `R` and `P`. `R` and `P` can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of `M` and `V`. A scalar input for `R` or `P` is expanded to a constant array with the same dimensions as the other input.

The mean of the negative binomial distribution with parameters r and p is rq / p, where $q = 1 - p$. The variance is rq / p^2.

The simplest motivation for the negative binomial is the case of successive random trials, each having a constant probability `P` of success. The number of *extra* trials you must perform in order to observe a given number `R` of successes has a negative binomial distribution. However, consistent with a more general interpretation of the negative binomial, `nbinstat` allows `R` to be any positive value, including nonintegers.

Example

```
p = 0.1:0.2:0.9;
r = 1:5;
[R,P] = meshgrid(r,p);
[M,V] = nbinstat(R,P)
M =
  9.0000  18.0000  27.0000  36.0000  45.0000
  2.3333  4.6667  7.0000  9.3333  11.6667
  1.0000  2.0000  3.0000  4.0000  5.0000
  0.4286  0.8571  1.2857  1.7143  2.1429
  0.1111  0.2222  0.3333  0.4444  0.5556

V =
  90.0000 180.0000 270.0000 360.0000 450.0000
  7.7778  15.5556  23.3333  31.1111  38.8889
  2.0000  4.0000  6.0000  8.0000  10.0000
  0.6122  1.2245  1.8367  2.4490  3.0612
  0.1235  0.2469  0.3704  0.4938  0.6173
```

See Also `nbincdf`, `nbinfit`, `nbininv`, `nbinpdf`, `nbinrnd`

ncfcdf

Purpose

Noncentral *F* cumulative distribution function

Syntax

```
P = ncfcdf(X,NU1,NU2,DELTA)
```

Description

`P = ncfcdf(X,NU1,NU2,DELTA)` computes the noncentral *F* cdf at each of the values in `X` using the corresponding numerator degrees of freedom in `NU1`, denominator degrees of freedom in `NU2`, and positive noncentrality parameters in `DELTA`. `NU1`, `NU2`, and `DELTA` can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of `P`. A scalar input for `X`, `NU1`, `NU2`, or `DELTA` is expanded to a constant array with the same dimensions as the other inputs.

The noncentral F cdf is

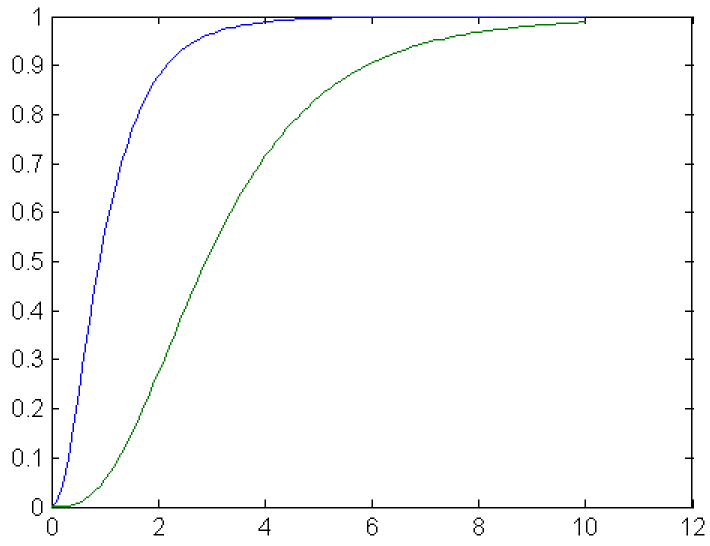
$$F(x|\nu_1,\nu_2,\delta) = \sum_{j=0}^{\infty}\left(\frac{\left(\frac{1}{2}\delta\right)^j}{j!}e^{-\frac{\delta}{2}}\right)I\left(\frac{\nu_1 \cdot x}{\nu_2+\nu_1 \cdot x}\middle|\frac{\nu_1}{2}+j,\frac{\nu_2}{2}\right)$$

where $I(x\,|\,a,b)$ is the incomplete beta function with parameters a and b.

Example

Compare the noncentral F cdf with δ = 10 to the F cdf with the same number of numerator and denominator degrees of freedom (5 and 20 respectively).

```
x = (0.01:0.1:10.01)';
p1 = ncfcdf(x,5,20,10);
p = fcdf(x,5,20);
plot(x,p,'-',x,p1,'-')
```



References

[1] Johnson, N., and S. Kotz, *Distributions in Statistics: Continuous Univariate Distributions-2,* John Wiley and Sons, 1970, pp. 189-200.

See Also

`cdf`, `ncfpdf`, `ncfinv`, `ncfrnd`, `ncfstat`

ncfinv

Purpose

Inverse of noncentral *F* cumulative distribution function

Syntax

```
X = ncfinv(P,NU1,NU2,DELTA)
```

Description

`X = ncfinv(P,NU1,NU2,DELTA)` returns the inverse of the noncentral F cdf with numerator degrees of freedom `NU1`, denominator degrees of freedom `NU2`, and positive noncentrality parameter `DELTA` for the corresponding probabilities in `P`. `P`, `NU1`, `NU2`, and `DELTA` can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of `X`. A scalar input for `P`, `NU1`, `NU2`, or `DELTA` is expanded to a constant array with the same dimensions as the other inputs.

Example

One hypothesis test for comparing two sample variances is to take their ratio and compare it to an F distribution. If the numerator and denominator degrees of freedom are 5 and 20 respectively, then you reject the hypothesis that the first variance is equal to the second variance if their ratio is less than that computed below.

```
critical = finv(0.95,5,20)
critical =
  2.7109
```

Suppose the truth is that the first variance is twice as big as the second variance. How likely is it that you would detect this difference?

```
prob = 1 - ncfcdf(critical,5,20,2)
prob =
  0.1297
```

If the true ratio of variances is 2, what is the typical (median) value you would expect for the F statistic?

```
ncfinv(0.5,5,20,2)
ans =
    1.2786
```

References

[1] Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions, 2nd edition*, John Wiley and Sons, 1993, pp. 102-105.

[2] Johnson, N., and S. Kotz, *Distributions in Statistics: Continuous Univariate Distributions-2,* John Wiley and Sons, 1970, pp. 189-200.

See Also

`icdf`, `ncfcdf`, `ncfpdf`, `ncfrnd`, `ncfstat`

Purpose

Noncentral *F* probability density function

Syntax

```
Y = ncfpdf(X,NU1,NU2,DELTA)
```

Description

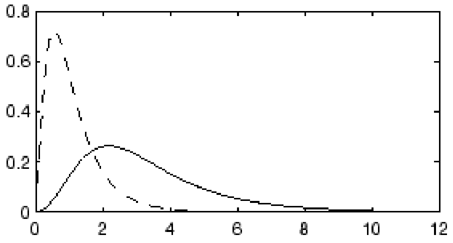
`Y = ncfpdf(X,NU1,NU2,DELTA)` computes the noncentral F pdf at each of the values in `X` using the corresponding numerator degrees of freedom in `NU1`, denominator degrees of freedom in `NU2`, and positive noncentrality parameters in `DELTA`. `X`, `NU1`, `N2`, and `B` can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of `Y`. A scalar input for `P`, `NU1`, `NU2`, or `DELTA` is expanded to a constant array with the same dimensions as the other inputs.

The F distribution is a special case of the noncentral F where $\delta = 0$. As δ increases, the distribution flattens like the plot in the example.

Example

Compare the noncentral F pdf with $\delta = 10$ to the F pdf with the same number of numerator and denominator degrees of freedom (5 and 20 respectively).

```
x = (0.01:0.1:10.01)';
p1 = ncfpdf(x,5,20,10);
p = fpdf(x,5,20);
plot(x,p,'-',x,p1,'-')
```



References

[1] Johnson, N., and S. Kotz, *Distributions in Statistics: Continuous Univariate Distributions-2,* John Wiley and Sons, 1970, pp. 189-200.

See Also

`ncfcdf`, `ncfinv`, `ncfrnd`, `ncfstat`, `pdf`

Purpose

Random numbers from noncentral *F* distribution

Syntax

```
R = ncfrnd(NU1,NU2,DELTA)
R = ncfrnd(NU1,NU2,DELTA,v)
R = ncfrnd(NU1,NU2,DELTA,m,n)
```

Description

`R = ncfrnd(NU1,NU2,DELTA)` returns a matrix of random numbers chosen from the noncentral F distribution with parameters `NU1`, `NU2` and `DELTA`. `NU1`, `NU2`, and `DELTA` can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of `R`. A scalar input for `NU1`, `NU2`, or `DELTA` is expanded to a constant matrix with the same dimensions as the other inputs.

`R = ncfrnd(NU1,NU2,DELTA,v)` returns a matrix of random numbers with parameters `NU1`, `NU2`, and `DELTA`, where `v` is a row vector. If `v` is a 1-by-2 vector, `R` is a matrix with `v(1)` rows and `v(2)` columns. If `v` is 1-by-n, `R` is an n-dimensional array.

`R = ncfrnd(NU1,NU2,DELTA,m,n)` generates random numbers with parameters `NU1`, `NU2`, and `DELTA`, where scalars m and n are the row and column dimensions of `R`.

Example

Compute six random numbers from a noncentral F distribution with 10 numerator degrees of freedom, 100 denominator degrees of freedom and a noncentrality parameter, δ, of 4.0. Compare this to the F distribution with the same degrees of freedom.

```
r = ncfrnd(10,100,4,1,6)
r =
  2.5995  0.8824  0.8220  1.4485  1.4415  1.4864

r1 = frnd(10,100,1,6)
r1 =
  0.9826  0.5911  1.0967  0.9681  2.0096  0.6598
```

References

[1] Johnson, N., and S. Kotz, *Distributions in Statistics: Continuous Univariate Distributions-2,* John Wiley and Sons, 1970, pp. 189-200.

See Also `ncfcdf`, `ncfinv`, `ncfpdf`, `ncfstat`

Purpose

Mean and variance of noncentral *F* distribution

Syntax

```
[M,V] = ncfstat(NU1,NU2,DELTA)
```

Description

`[M,V] = ncfstat(NU1,NU2,DELTA)` returns the mean of and variance for the noncentral F pdf with `NU1` and `NU2` degrees of freedom and noncentrality parameter `DELTA`. `NU1`, `NU2`, and `DELTA` can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of `M` and `V`. A scalar input for `NU1`, `NU2`, or `DELTA` is expanded to a constant array with the same dimensions as the other input.

The mean of the noncentral F distribution with parameters ν_1, ν_2, and δ is

$$\frac{\nu_2(\delta + \nu_1)}{\nu_1(\nu_2 - 2)}$$

where $\nu_2 > 2$.

The variance is

$$2\left(\frac{\nu_2}{\nu_1}\right)^2\left[\frac{(\delta + \nu_1)^2 + (2\delta + \nu_1)(\nu_2 - 2)}{(\nu_2 - 2)^2(\nu_2 - 4)}\right]$$

where $\nu_2 > 4$.

Example

```
[m,v]= ncfstat(10,100,4)
m =
  1.4286
v =
  0.4252
```

References

[1] Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions, 2nd Edition*, John Wiley and Sons, 1993, pp. 73-74.

[2] Johnson, N., and S. Kotz, *Distributions in Statistics: Continuous Univariate Distributions-2,* John Wiley and Sons, 1970, pp. 189-200.

See Also `ncfcdf`, `ncfinv`, `ncfpdf`, `ncfrnd`

Purpose

Noncentral *t* cumulative distribution function

Syntax

```
P = nctcdf(X,NU,DELTA)
```

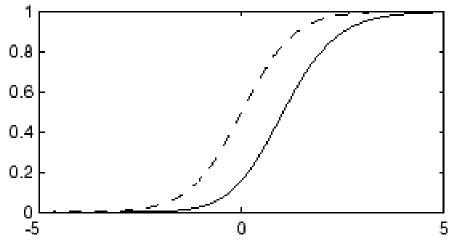
Description

`P = nctcdf(X,NU,DELTA)` computes the noncentral *t* cdf at each of the values in `X` using the corresponding degrees of freedom in `NU` and noncentrality parameters in `DELTA`. `X`, `NU`, and `DELTA` can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of `P`. A scalar input for `X`, `NU`, or `DELTA` is expanded to a constant array with the same dimensions as the other inputs.

Example

Compare the noncentral *t* cdf with `DELTA = 1` to the *t* cdf with the same number of degrees of freedom (10).

```
x = (-5:0.1:5)';
p1 = nctcdf(x,10,1);
p = tcdf(x,10);
plot(x,p,'-',x,p1,'-')
```



References

[1] Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions, 2nd Edition*, John Wiley and Sons, 1993, pp. 147-148.

[2] Johnson, N., and S. Kotz, *Distributions in Statistics: Continuous Univariate Distributions-2,* John Wiley and Sons, 1970, pp. 201-219.

See Also

`cdf`, `nctcdf`, `nctinv`, `nctpdf`, `nctrnd`, `nctstat`

Purpose

Inverse of noncentral *t* cumulative distribution

Syntax

```
X = nctinv(P,NU,DELTA)
```

Description

`X = nctinv(P,NU,DELTA)` returns the inverse of the noncentral *t* cdf with `NU` degrees of freedom and noncentrality parameter `DELTA` for the corresponding probabilities in `P`. `P`, `NU`, and `DELTA` can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of `X`. A scalar input for `P`, `NU`, or `DELTA` is expanded to a constant array with the same dimensions as the other inputs.

Example

```
x = nctinv([0.1 0.2],10,1)
x =
  -0.2914  0.1618
```

References

[1] Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions, 2nd Edition*, John Wiley and Sons, 1993, pp. 147-148.

[2] Johnson, N., and S. Kotz, *Distributions in Statistics: Continuous Univariate Distributions-2,* John Wiley and Sons, 1970, pp. 201-219.

See Also

`icdf`, `nctcdf`, `nctpdf`, `nctrnd`, `nctstat`

Purpose

Noncentral *t* probability density function

Syntax

```
Y = nctpdf(X,V,DELTA)
```

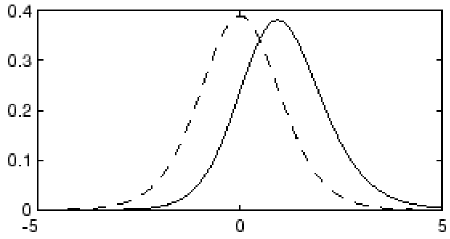
Description

`Y = nctpdf(X,V,DELTA)` computes the noncentral *t* pdf at each of the values in `X` using the corresponding degrees of freedom in `V` and noncentrality parameters in `DELTA`. Vector or matrix inputs for `X`, `V`, and `DELTA` must have the same size, which is also the size of `Y`. A scalar input for `X`, `V`, or `DELTA` is expanded to a constant matrix with the same dimensions as the other inputs.

Example

Compare the noncentral *t* pdf with `DELTA = 1` to the *t* pdf with the same number of degrees of freedom (10).

```
x = (-5:0.1:5)';
p1 = nctpdf(x,10,1);
p = tpdf(x,10);
plot(x,p,'-',x,p1,'-')
```



References

[1] Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions, 2nd Edition*, John Wiley and Sons, 1993, pp. 147-148.

[2] Johnson, N., and S. Kotz, *Distributions in Statistics: Continuous Univariate Distributions-2,* John Wiley and Sons, 1970, pp. 201-219.

See Also

`nctcdf`, `nctinv`, `nctrnd`, `nctstat`, `pdf`

nctrnd

Purpose

Random numbers from noncentral *t* distribution

Syntax

```
R = nctrnd(V,DELTA)
R = nctrnd(V,DELTA,v)
R = nctrnd(V,DELTA,m,n)
```

Description

`R = nctrnd(V,DELTA)` returns a matrix of random numbers chosen from the noncentral T distribution with parameters `V` and `DELTA`. `V` and `DELTA` can be vectors, matrices, or multidimensional arrays. A scalar input for `V` or `DELTA` is expanded to a constant array with the same dimensions as the other input.

`R = nctrnd(V,DELTA,v)` returns a matrix of random numbers with parameters `V` and `DELTA`, where `v` is a row vector. If v is a 1-by-2 vector, `R` is a matrix with `v(1)` rows and `v(2)` columns. If `v` is 1-by-n, `R` is an n-dimensional array.

`R = nctrnd(V,DELTA,m,n)` generates random numbers with parameters `V` and `DELTA`, where scalars m and n are the row and column dimensions of `R`.

Example

```
  nctrnd(10,1,5,1)
  ans =
    1.6576
    1.0617
    1.4491
    0.2930
    3.6297
```

References

[1] Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions, 2nd Edition*, John Wiley and Sons, 1993, pp. 147-148.

[2] Johnson, N., and S. Kotz, *Distributions in Statistics: Continuous Univariate Distributions-2,* John Wiley and Sons, 1970, pp. 201-219.

See Also

`nctcdf`, `nctinv`, `nctpdf`, `nctstat`

Purpose

Mean and variance of noncentral *t* distribution

Syntax

```
[M,V] = nctstat(NU,DELTA)
```

Description

`[M,V] = nctstat(NU,DELTA)` returns the mean of and variance for the noncentral t pdf with `NU` degrees of freedom and noncentrality parameter `DELTA`. `NU` and `DELTA` can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of `M` and `V`. A scalar input for `NU` or `DELTA` is expanded to a constant array with the same dimensions as the other input.

The mean of the noncentral t distribution with parameters ν and δ is

$$\frac{\delta(\nu/2)^{1/2}\Gamma((\nu-1)/2)}{\Gamma(\nu/2)}$$

where $\nu > 1$.

The variance is

$$\frac{\nu}{(\nu-2)}(1+\delta^2)-\frac{\nu}{2}\delta^2\left[\frac{\Gamma((\nu-1)/2)}{\Gamma(\nu/2)}\right]^2$$

where $\nu > 2$.

Example

```
[m,v] = nctstat(10,1)

m =
   1.0837

v =
   1.3255
```

References

[1] Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions, 2nd Edition*, John Wiley and Sons, 1993, pp. 147-148.

[2] Johnson, N., and S. Kotz, *Distributions in Statistics: Continuous Univariate Distributions-2,* John Wiley and Sons, 1970, pp. 201-219.

See Also `nctcdf`, `nctinv`, `nctpdf`, `nctrnd`

Purpose

Noncentral chi-square cumulative distribution function

Syntax

```
P = ncx2cdf(X,V,DELTA)
```

Description

`P = ncx2cdf(X,V,DELTA)` computes the noncentral chi-square cdf at each of the values in `X` using the corresponding degrees of freedom in `V` and positive noncentrality parameters in `DELTA`. `X`, `V`, and `DELTA` can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of `P`. A scalar input for `X`, `V`, or `DELTA` is expanded to a constant array with the same dimensions as the other inputs.

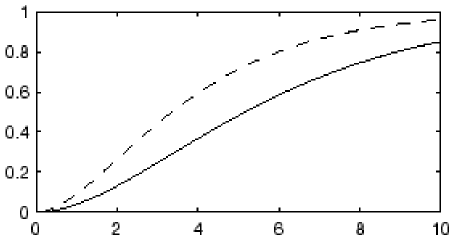
Some texts refer to this distribution as the generalized Rayleigh, Rayleigh-Rice, or Rice distribution.

The noncentral chi-square cdf is

$$F(x|\nu,\delta) = \sum_{j=0}^{\infty}\left(\frac{\left(\frac{1}{2}\delta\right)^{j}}{j!}e^{-\frac{\delta}{2}}\right)Pr[\chi^2_{\nu+2j} \le x]$$

Example

```
x = (0:0.1:10)';
p1 = ncx2cdf(x,4,2);
p = chi2cdf(x,4);
plot(x,p,'-',x,p1,'-')
```



References

[1] Johnson, N., and S. Kotz, *Distributions in Statistics: Continuous Univariate Distributions-2,* John Wiley and Sons, 1970, pp. 130-148.

See Also `cdf`, `ncx2inv`, `ncx2pdf`, `ncx2rnd`, `ncx2stat`

Purpose

Inverse of noncentral chi-square cumulative distribution function

Syntax

```
X = ncx2inv(P,V,DELTA)
```

Description

`X = ncx2inv(P,V,DELTA)` returns the inverse of the noncentral chi-square cdf with parameters `V` and `DELTA` at the corresponding probabilities in `P`. `P`, `V`, and `DELTA` can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of `X`. A scalar input for `P`, `V`, or `DELTA` is expanded to a constant array with the same dimensions as the other inputs.

Algorithm

`ncx2inv` uses Newton's method to converge to the solution.

Example

```
ncx2inv([0.01 0.05 0.1],4,2)
ans =
  0.4858  1.1498  1.7066
```

References

[1] Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions*, 2nd Edition, John Wiley and Sons, 1993, pp. 50–52.

[2] Johnson, N., and S. Kotz, *Distributions in Statistics: Continuous Univariate Distributions 2*, John Wiley and Sons, 1970, pp. 130-148.

See Also

`icdf`, `ncx2cdf`, `ncx2pdf`, `ncx2rnd`, `ncx2stat`

Purpose

Noncentral chi-square probability density function

Syntax

```
Y = ncx2pdf(X,V,DELTA)
```

Description

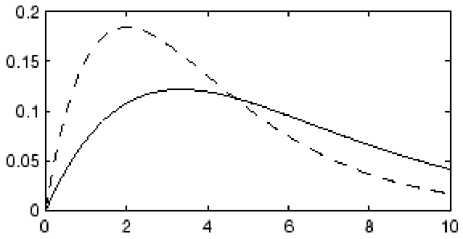
`Y = ncx2pdf(X,V,DELTA)` computes the noncentral chi-square pdf at each of the values in `X` using the corresponding degrees of freedom in `V` and positive noncentrality parameters in `DELTA`. Vector or matrix inputs for `X`, `V`, and `DELTA` must have the same size, which is also the size of `Y`. A scalar input for `X`, `V`, or `DELTA` is expanded to a constant array with the same dimensions as the other inputs.

Some texts refer to this distribution as the generalized Rayleigh, Rayleigh-Rice, or Rice distribution.

Example

As the noncentrality parameter δ increases, the distribution flattens as shown in the plot.

```
x = (0:0.1:10)';
p1 = ncx2pdf(x,4,2);
p = chi2pdf(x,4);
plot(x,p,'-',x,p1,'-')
```



References

[1] Johnson, N., and S. Kotz, *Distributions in Statistics: Continuous Univariate Distributions-2,* John Wiley and Sons, 1970, pp. 130-148.

See Also

`ncx2cdf`, `ncx2inv`, `ncx2rnd`, `ncx2stat`, `pdf`

Purpose

Random numbers from noncentral chi-square distribution

Syntax

```
R = ncx2rnd(V,DELTA)
R = ncx2rnd(V,DELTA,v)
R = ncx2rnd(V,DELTA,m,n)
```

Description

`R = ncx2rnd(V,DELTA)` returns a matrix of random numbers chosen from the noncentral chi-square distribution with parameters `V` and `DELTA`. `V` and `DELTA` can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of `R`. A scalar input for `V` or `DELTA` is expanded to a constant array with the same dimensions as the other input.

`R = ncx2rnd(V,DELTA,v)` returns a matrix of random numbers with parameters `V` and `DELTA`, where `v` is a row vector. If `v` is a 1-by-2 vector, `R` is a matrix with `v(1)` rows and `v(2)` columns. If `v` is 1-by-n, `R` is an n-dimensional array.

`R = ncx2rnd(V,DELTA,m,n)` generates random numbers with parameters `V` and `DELTA`, where scalars m and n are the row and column dimensions of R.

Example

```
ncx2rnd(4,2,6,3)
ans =
  6.8552  5.9650  11.2961
  5.2631  4.2640  5.9495
  9.1939  6.7162  3.8315
 10.3100  4.4828  7.1653
  2.1142  1.9826  4.6400
  3.8852  5.3999  0.9282
```

References

[1] Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions, 2nd Edition*, John Wiley and Sons, 1993, pp. 50-52.

[2] Johnson, N., and S. Kotz, *Distributions in Statistics: Continuous Univariate Distributions-2,* John Wiley and Sons, 1970, pp. 130-148.

See Also `ncx2cdf`, `ncx2inv`, `ncx2pdf`, `ncx2stat`

Purpose

Mean and variance of noncentral chi-square distribution

Syntax

```
[M,V] = ncx2stat(NU,DELTA)
```

Description

`[M,V] = ncx2stat(NU,DELTA)` returns the mean of and variance for the noncentral chi-square pdf with `NU` degrees of freedom and noncentrality parameter `DELTA`. `NU` and `DELTA` can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of `M` and `V`. A scalar input for `NU` or `DELTA` is expanded to a constant array with the same dimensions as the other input.

The mean of the noncentral chi-square distribution with parameters ν and δ is $\nu + \delta$, and the variance is $2(\nu + 2\delta)$.

Example

```
[m,v] = ncx2stat(4,2)
m =
     6
v =
    16
```

References

[1] Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions, 2nd Edition*, John Wiley and Sons, 1993, pp. 50-52.

[2] Johnson, N., and S. Kotz, *Distributions in Statistics: Continuous Univariate Distributions-2,* John Wiley and Sons, 1970, pp. 130-148.

See Also

`ncx2cdf`, `ncx2inv`, `ncx2pdf`, `ncx2rnd`

Purpose

Nonlinear least-squares regression

Syntax

```
beta = nlinfit(X,y,fun,beta0)
[beta,r,J,SIGMA,mse] = nlinfit(X,y,fun,beta0)
[...] = nlinfit(X,y,fun,beta0,options)
```

Description

`beta = nlinfit(X,y,fun,beta0)` estimates the coefficients of a nonlinear regression function using least squares. `y` is a vector of response (dependent variable) values. Typically, `X` is a design matrix of predictor (independent variable) values, with one row for each value in `y`. However, `X` can be any array that `fun` can accept. `fun` is a function handle, specified using the `@` sign, to a function of the form

```
yhat = myfun(beta,X)
```

where `beta` is a coefficient vector. `fun` returns a vector `yhat` of fitted `y` values. `beta0` is a vector containing initial values for the coefficients.

`[beta,r,J,SIGMA,mse] = nlinfit(X,y,fun,beta0)` returns the fitted coefficients `beta`, the residuals `r`, the Jacobian `J` of `fun`, the estimated covariance matrix `SIGMA` for the fitted coefficients, and an estimate `mse` of the variance of the error term. You can use these outputs with `nlpredci` to produce error estimates on predictions, and with `nlparci` to produce error estimates on the estimated coefficients. If you use the robust fitting option (see below), you must use `SIGMA` and may need `mse` as input to `nlpredci` or `nlparci` to insure that the confidence intervals take the robust fit properly into account.

Note `nlintool` provides a GUI for performing nonlinear fits and computing confidence intervals.

`[...]  = nlinfit(X,y,fun,beta0,options)` specifies control parameters for the algorithm used in `nlinfit`. `options` is a structure created by a call to `statset`. Applicable `statset` parameters are:

- `'MaxIter'` — Maximum number of iterations allowed. The default is `100`.
- `'TolFun'` — Termination tolerance on the residual sum of squares. The defaults is `1e-8`.
- `'TolX'` — Termination tolerance on the estimated coefficients `beta`. The default is `1e-8`.
- `'Display'` — Level of display output during estimation. The choices are
  - `'off'` (the default),
  - `'iter'`
  - `'final'`
- `'DerivStep'` — Relative difference used in finite difference gradient calculation. May be a scalar, or the same size as the parameter vector `beta0`. The default is `eps^(1/3)`.
- `'FunValCheck'` — Check for invalid values, such as `NaN` or `Inf`, from the objective function. Values are `'off'` or `'on'` (the default).
- `'Robust'` — Invoke robust fitting option. Values are `'off'` (the default) or `'on'`.
- `'WgtFun'` — Specify the weight function for the robust fitting. It can be `'bisquare'` (the default), `'andrews'`, `'cauchy'`, `'fair'`, `'huber'`, `'logistic'`, `'talwar'`, or `'welsch'`. `'WgtFun'` is only used when `'Robust'` is set to `'on'`. It can also be a function handle that accepts a normalized residual as input and returns the robust weights as output.
- `'Tune'` — The tuning constant used to normalize the residuals before applying the weight function. The value of `'Tune'` must be positive, and the default value is dependent on the weight function. `'Tune'` is required if the weight function is specified as a function handle.

`nlinfit` treats `NaN`s in `y` or `fun(beta,X)` as missing data and ignores the corresponding rows.

Example

Find the coefficients that best fit the data in `reaction.mat`. The data record reaction kinetics as a function of the partial pressures of three chemical reactants: hydrogen, *n*-pentane, and isopentane.

The `hougen` function uses the Hougen-Watson model for reaction kinetics to return the predicted values of the reaction rate.

```
load reaction
betafit = nlinfit(reactants,rate,@hougen,beta)
betafit =
    1.2526
    0.0628
    0.0400
    0.1124
    1.1914
```

Reference

[1] Seber, G. A. F., and C. J. Wild, *Nonlinear Regression*, John Wiley & Sons Inc., 1989.

See Also

`hougen`, `nlintool`, `nlparci`, `nlpredci` , `lsqnonlin`

Note The `lsqnonlin` function in Optimization Toolbox has more outputs related to how well the optimization performed. It can put bounds on the parameters, and it accepts many options to control the optimization algorithm. The `nlinfit` function has more statistics-oriented outputs that are useful, for example, in finding confidence intervals for the coefficients. It also comes with the `nlintool` GUI for visualizing the fitted function.

Purpose

Interactive nonlinear fitting

Syntax

```
nlintool(x,y,fun,beta0)
nlintool(x,y,fun,beta0,alpha)
nlintool(x,y,fun,beta0,alpha,'xname','yname')
```

Description

`nlintool` displays a “vector” of plots, one for each column of the matrix of inputs, `x`. The response variable, `y`, is a column vector that matches the number of rows in `x`.

`nlintool(x,y,fun,beta0)` is a prediction plot that provides a nonlinear curve fit to (x, y) data. It plots a 95% global confidence interval for predictions as two red curves. `beta0` is a vector containing initial guesses for the parameters.

`fun` is a function handle of the form

```
yhat = myfun(beta,x)
```

`nlintool(x,y,fun,beta0,alpha)` plots a `100(1 - alpha)`% confidence interval for predictions.

The default value for `alpha` is 0.05, which produces 95% confidence intervals.

`nlintool(x,y,fun,beta0,alpha,'xname','yname')` labels the plot using the string matrix ’`xname`’ for the `x` variables and the string ’`yname`’ for the `y` variable.

`nlintool` treats `NaN`s in `y` or `fun(beta, X)` as missing data and ignores the corresponding rows.

Example

See “Interactive Nonlinear Regression” on page 8-5 for an example using the graphical interface.

See Also

`nlinfit`, `rstool`

nlparci

Purpose

Confidence intervals for parameters in nonlinear regression

Syntax

```
ci = nlparci(beta,resid,'covar',sigma)
ci = nlparci(beta,resid,'jacobian',J)
ci = nlparci(...,'alpha',alpha)
```

Description

`ci = nlparci(beta,resid,'covar',sigma)` returns the 95% confidence intervals `ci` for the nonlinear least squares parameter estimates `beta`. Before calling `nlparci`, use `nlinfit` to fit a nonlinear regression model and get the coefficient estimates `beta`, residuals `resid`, and estimated coefficient covariance matrix `sigma`.

`ci = nlparci(beta,resid,'jacobian',J)` is an alternative syntax that also computes 95% confidence intervals. `J` is the Jacobian computed by `nlinfit`. If the `'robust'` option is used with `nlinfit`, use the `'covar'` input rather than the `'jacobian'` input so that the required `sigma` parameter takes the robust fitting into account.

`ci = nlparci(...,'alpha',alpha)` returns `100(1-alpha)`% confidence intervals.

`nlparci` treats `NaN`s in `resid` or `J` as missing values, and ignores the corresponding observations.

The confidence interval calculation is valid for systems where the length of `resid` exceeds the length of `beta` and `J` has full column rank. When `J` is ill-conditioned, confidence intervals may be inaccurate.

Example

Continuing the example from `nlinfit`:

```
load reaction
[beta,resid,J,Sigma] = ...
     nlinfit(reactants,rate,'hougen',beta);
ci = nlparci(beta,resid,'jacobian',J)
ci =
   -0.7467    3.2519
   -0.0377    0.1632
   -0.0312    0.1113
   -0.0609    0.2857
   -0.7381    3.1208
```

See Also `nlinfit`, `nlintool`, `nlpredci`

nlpredci

Purpose

Confidence intervals for predictions in nonlinear regression

Syntax

```
[ypred,delta] = nlpredci(modelfun,x,beta,resid,'covar',sigma)
[ypred,delta] = nlpredci(modelfun,x,beta,resid,'jacobian',J)
[...] = nlpredci(...,param1,val1,param2,val2,...)
```

Description

`[ypred,delta] = nlpredci(modelfun,x,beta,resid,'covar',sigma)` returns predictions, `ypred`, and 95% confidence interval half-widths, `delta`, for the nonlinear regression model defined by `modelfun`, at input values `x`. `modelfun` is a function handle, specified using `@`, that accepts two arguments—a coefficient vector and the array `x`—and returns a vector of fitted `y` values. Before calling `nlpredci`, use `nlinfit` to fit `modelfun` by nonlinear least squares and get estimated coefficient values `beta`, residuals `resid`, and estimated coefficient covariance matrix `sigma`.

`[ypred,delta] = nlpredci(modelfun,x,beta,resid,'jacobian',J)` is an alternative syntax that also computes 95% confidence intervals. `J` is the Jacobian computed by `nlinfit`. If the `'robust'` option is used with `nlinfit`, use the `'covar'` input rather than the `'jacobian'` input so that the required `sigma` parameter takes the robust fitting into account.

`[...] = nlpredci(...,`*`param1`*`,`*`val1`*`,`*`param2`*`,`*`val2`*`,...)` accepts optional parameter name/value pairs.

Name	Value
`'alpha'`	A value between 0 and 1 that specifies the confidence level as `100(1-alpha)`%. Default is `0.05`.
`'mse'`	The mean squared error returned by `nlinfit`. This is required to predict new observations (see `'predopt'`) if the robust option is used with `nlinfit`; otherwise, the `'mse'` is computed from the residuals and does not take the robust fitting into account.

Name	Value
'predopt'	Either 'curve' (the default) to compute confidence intervals for the estimated curve (function value) at x, or 'observation' for prediction intervals for a new observation at x. If 'observation'is specified after using a robust option with nlinfit, the 'mse' parameter must be supplied to specify the robust estimate of the mean squared error.
'simopt'	Either 'on' for simultaneous bounds, or 'off' (the default) for nonsimultaneous bounds.

nlpredci treats NaNs in resid or J as missing values, and ignores the corresponding observations.

The confidence interval calculation is valid for systems where the length of resid exceeds the length of beta and J has full column rank at beta. When J is ill-conditioned, predictions and confidence intervals may be inaccurate.

Example

Continuing the example from nlinfit, you can determine the predicted function value at the value newX and the half-width of a confidence interval for it.

```
load reaction;
[beta,resid,J] = nlinfit(reactants,rate,@hougen,beta);
newX = reactants(1:2,:);
[ypred,delta] = nlpredci(@hougen,newX,beta,resid,J);
 ypred =
    8.4179
    3.9542
delta =
    0.2805
    0.2474
```

See Also `nlinfit`, `nlintool`, `nlparci`

Purpose

Node errors of tree

Syntax

```
e = nodeerr(t)
e = nodeerr(t,nodes)
```

Description

`e = nodeerr(t)` returns an *n*-element vector `e` of the errors of the nodes in the tree `t`, where *n* is the number of nodes. For a regression tree, the error `e(i)` for node `i` is the variance of the observations assigned to node `i`. For a classification tree, `e(i)` is the misclassification probability for node `i`.

`e = nodeerr(t,nodes)` takes a vector `nodes` of node numbers and returns the errors for the specified nodes.

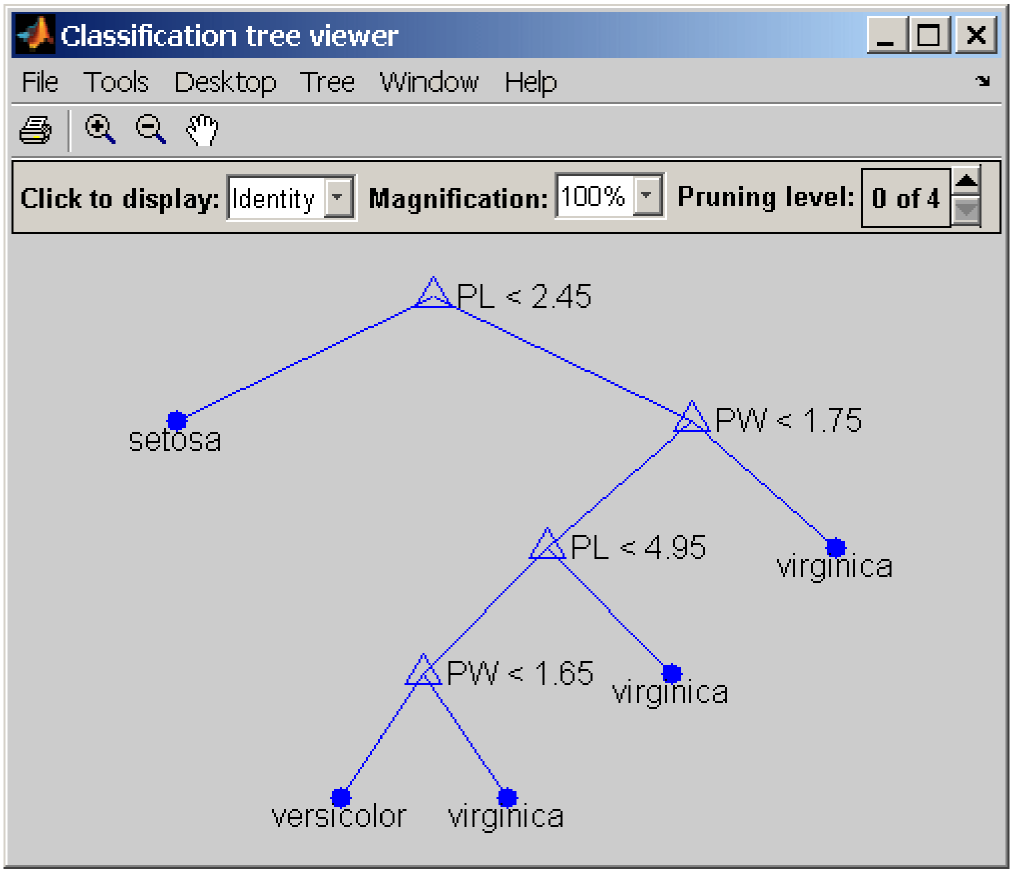
The error `e` is the so-called *resubstitution error* computed by applying the tree to the same data used to create the tree. This error is likely to under estimate the error you would find if you applied the tree to new data. The `test` function provides options to compute the error (or cost) using cross-validation or a test sample.

Example

Create a classification tree for Fisher's iris data:

```
load fisheriris;

t = classregtree(meas,species,'names',{'SL' 'SW' 'PL' 'PW'})
t =
Decision tree for classification
1  if PL<2.45 then node 2 else node 3
2  class = setosa
3  if PW<1.75 then node 4 else node 5
4  if PL<4.95 then node 6 else node 7
5  class = virginica
6  if PW<1.65 then node 8 else node 9
7  class = virginica
8  class = versicolor
9  class = virginica
```

```
view(t)
```



```
e = nodeerr(t)
e =
    0.6667
         0
    0.5000
    0.0926
    0.0217
    0.0208
    0.3333
         0
         0
```

Reference

[1] Breiman, L., et al., *Classification and Regression Trees*, Chapman & Hall, Boca Raton, 1993.

See Also

`classregtree`, `numnodes`, `test`

nodeprob

Purpose

Node probabilities of tree

Syntax

```
p = nodeprob(t)
p = nodeprob(t,nodes)
```

Description

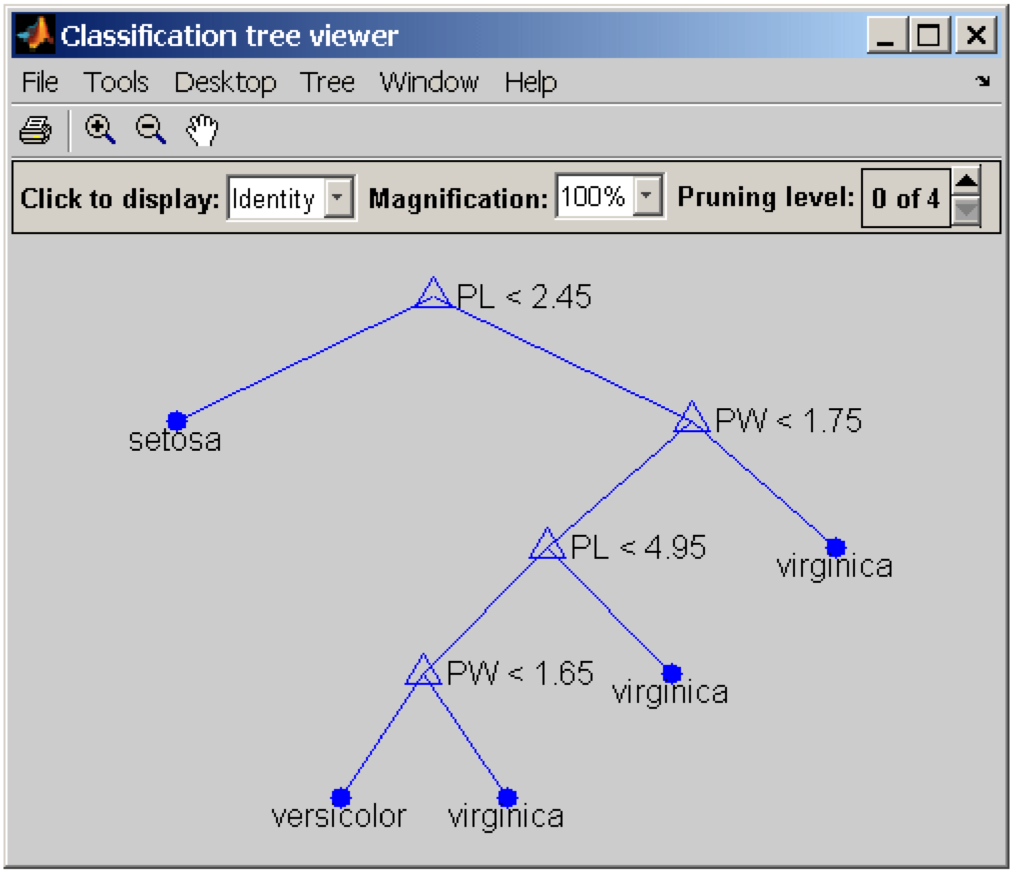
`p = nodeprob(t)` returns an n-element vector `p` of the probabilities of the nodes in the tree `t`, where n is the number of nodes. The probability of a node is computed as the proportion of observations from the original data that satisfy the conditions for the node. For a classification tree, this proportion is adjusted for any prior probabilities assigned to each class.

`p = nodeprob(t,nodes)` takes a vector `nodes` of node numbers and returns the probabilities for the specified nodes.

Example

Create a classification tree for Fisher's iris data:

```
load fisheriris;

t = classregtree(meas,species,'names',{'SL' 'SW' 'PL' 'PW'})
t =
Decision tree for classification
1  if PL<2.45 then node 2 else node 3
2  class = setosa
3  if PW<1.75 then node 4 else node 5
4  if PL<4.95 then node 6 else node 7
5  class = virginica
6  if PW<1.65 then node 8 else node 9
7  class = virginica
8  class = versicolor
9  class = virginica

view(t)
```



```
p = nodeprob(t)
p =
    1.0000
    0.3333
    0.6667
    0.3600
    0.3067
    0.3200
    0.0400
    0.3133
    0.0067
```

Reference

[1] Breiman, L., et al., *Classification and Regression Trees*, Chapman & Hall, Boca Raton, 1993.

See Also

`classregtree`, `numnodes`, `nodesize`

Purpose

Size of tree node

Syntax

```
sizes = nodesize(t)
sizes = nodesize(t,nodes)
```

Description

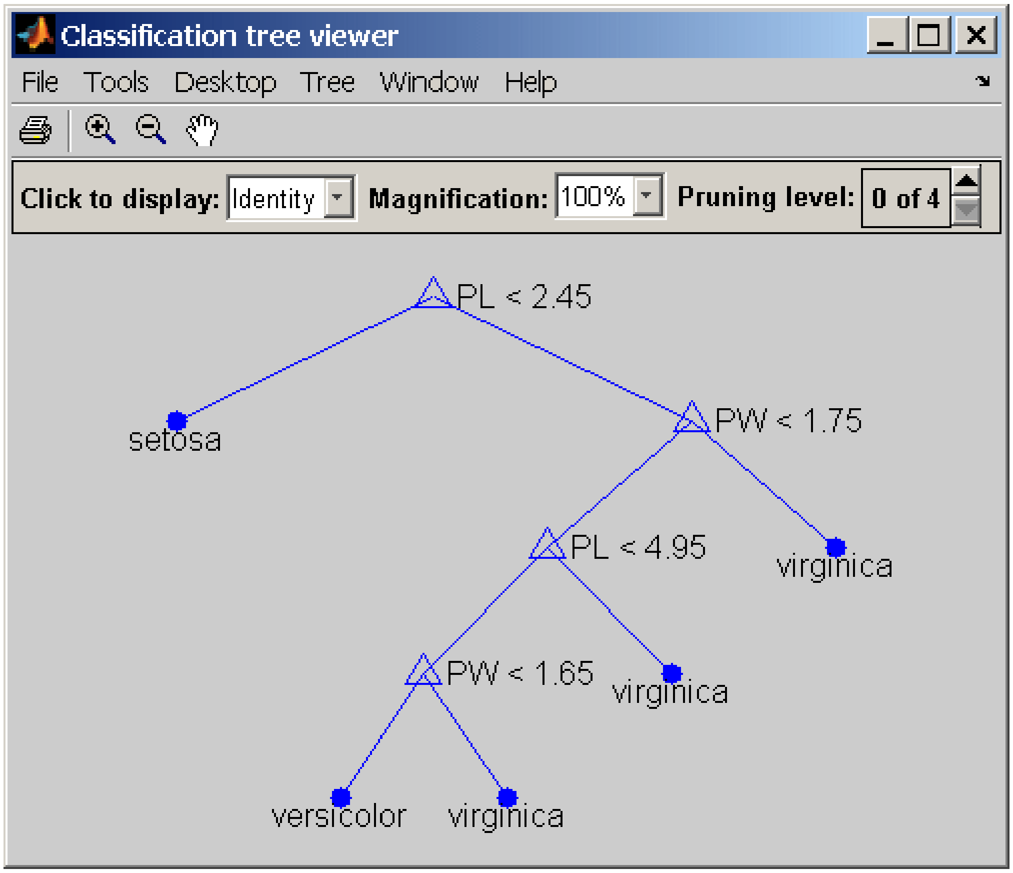
`sizes = nodesize(t)` returns an *n*-element vector `sizes` of the sizes of the nodes in the tree `t`, where *n* is the number of nodes. The size of a node is defined as the number of observations from the data used to create the tree that satisfy the conditions for the node.

`sizes = nodesize(t,nodes)` takes a vector `nodes` of node numbers and returns the sizes for the specified nodes.

Example

Create a classification tree for Fisher's iris data:

```
load fisheriris;

t = classregtree(meas,species,'names',{'SL' 'SW' 'PL' 'PW'})
t =
Decision tree for classification
1  if PL<2.45 then node 2 else node 3
2  class = setosa
3  if PW<1.75 then node 4 else node 5
4  if PL<4.95 then node 6 else node 7
5  class = virginica
6  if PW<1.65 then node 8 else node 9
7  class = virginica
8  class = versicolor
9  class = virginica

view(t)
```



```
sizes = nodesize(t)
sizes =
   150
    50
   100
    54
    46
    48
     6
    47
     1
```

Reference

[1] Breiman, L., et al., *Classification and Regression Trees*, Chapman & Hall, Boca Raton, 1993.

See Also

`classregtree`, `numnodes`

nominal

Purpose

Create nominal array

Syntax

```
B = nominal(A)
B = nominal(A,labels)
B = nominal(A,labels,levels)
B = nominal(A,labels,[],edges)
```

Description

`B = nominal(A)` creates a nominal array `B` from the array `A`. `A` can be numeric, logical, character, categorical, or a cell array of strings. `nominal` creates the levels of `B` from the sorted unique values in `A`, and creates default labels for them.

`B = nominal(A,labels)` labels the levels in `B` using the character array or cell array of strings `labels`. `nominal` assigns labels to levels in `B` in order according to the sorted unique values in `A`.

`B = nominal(A,labels,levels)` creates a nominal array with possible levels defined by `levels`. `levels` is a vector whose values can be compared to those in `A` using the equality operator. `nominal` assigns labels to each level from the corresponding elements of `labels`. If `A` contains any values not present in `levels`, the levels of the corresponding elements of `B` are undefined.

`B = nominal(A,labels,[],edges)` creates a nominal array by binning the numeric array `A` with bin edges given by the numeric vector `edges`. The uppermost bin includes values equal to the right-most edge. `nominal` assigns labels to each level in `B` from the corresponding elements of `labels`. `edges` must have one more element than `labels`.

By default, an element of `B` is undefined if the corresponding element of `A` is `NaN` (when `A` is numeric), an empty string (when `A` is character), or undefined (when `A` is categorical). `nominal` treats such elements as “undefined” or “missing” and does not include entries for them among the possible levels for `B`. To create an explicit level for such elements instead of treating them as undefined, you must use the `levels` input, and include `NaN`, the empty string, or an undefined element.

You may include duplicate labels in `labels` in order to merge multiple values in `A` into a single level in `B`.

Examples

Example 1

Create a nominal array from Fisher's iris data:

```
load fisheriris
species = nominal(species);
data = dataset(species,meas);
summary(data)
species: [150x1 nominal]
   setosa      versicolor      virginica
       50              50             50
meas: [150x4 double]
   min       4.3000         2          1    0.1000
   1st Q     5.1000    2.8000     1.6000    0.3000
   median    5.8000         3     4.3500    1.3000
   3rd Q     6.4000    3.3000     5.1000    1.8000
   max       7.9000    4.4000     6.9000    2.5000
```

Example 2

1. Load patient data from the CSV file `hospital.dat` and store the information in a dataset array with observation names given by the first column in the data (patient identification):

```
patients = dataset('file','hospital.dat',...
                   'delimiter',',',...
                   'ReadObsNames',true);
```

2. Make the {0,1}-valued variable `smoke` nominal, and change the labels to `'No'` and `'Yes'`:

```
patients.smoke = nominal(patients.smoke,{'No','Yes'});
```

3. Add new levels to `smoke` as placeholders for more detailed histories of smokers:

```
patients.smoke = addlevels(patients.smoke,...
                 {'0-5 Years','5-10 Years','LongTerm'});
```

4 Assuming the nonsmokers have never smoked, relabel the 'No' level:

```
patients.smoke = setlabels(patients.smoke,'Never','No');
```

5 Drop the undifferentiated 'Yes' level from smoke:

```
patients.smoke = droplevels(patients.smoke,'Yes');

Warning: OLDLEVELS contains categorical levels that
were present in A, caused some array elements to have
undefined levels.
```

Note that smokers now have an undefined level.

6 Set each smoker to one of the new levels, by observation name:

```
patients.smoke('YPL-320') = '5-10 Years';
```

See Also

ordinal, histc

Purpose

Normal cumulative distribution function

Syntax

```
P = normcdf(X,mu,sigma)
[P,PLO,PUP] = normcdf(X,mu,sigma,pcov,alpha)
```

Description

`P = normcdf(X,mu,sigma)` computes the normal cdf at each of the values in `X` using the corresponding parameters in `mu` and `sigma`. `X`, `mu`, and `sigma` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs. The parameters in `sigma` must be positive.

`[P,PLO,PUP] = normcdf(X,mu,sigma,pcov,alpha)` produces confidence bounds for `P` when the input parameters `mu` and `sigma` are estimates. `pcov` is the covariance matrix of the estimated parameters. `alpha` specifies 100(1 - `alpha`)% confidence bounds. The default value of `alpha` is 0.05. `PLO` and `PUP` are arrays of the same size as `P` containing the lower and upper confidence bounds.

The function `normdf` computes confidence bounds for `P` using a normal approximation to the distribution of the estimate

$$\frac{X-\hat{\mu}}{\hat{\sigma}}$$

and then transforming those bounds to the scale of the output `P`. The computed bounds give approximately the desired confidence level when you estimate `mu`, `sigma`, and `pcov` from large samples, but in smaller samples other methods of computing the confidence bounds might be more accurate.

The normal cdf is

$$p = F(x|\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}}\int_{-\infty}^{x} e^{\frac{-(t-\mu)^2}{2\sigma^2}}\,dt$$

The result, p, is the probability that a single observation from a normal distribution with parameters μ and σ will fall in the interval (-∞ x].

The *standard normal* distribution has μ = 0 and σ = 1.

Examples

What is the probability that an observation from a standard normal distribution will fall on the interval [-1 1]?

```
p = normcdf([-1 1]);
p(2)-p(1)
ans =
  0.6827
```

More generally, about 68% of the observations from a normal distribution fall within one standard deviation, σ, of the mean, μ.

See Also

`cdf`, `normfit`, `normlike`, `normpdf`, `normspec`, `normstat`, `normrnd`, `norminv`, `normplot`

Purpose

Parameter estimates and confidence intervals for normally distributed data

Syntax

```
[muhat,sigmahat] = normfit(data)
[muhat,sigmahat,muci,sigmaci] = normfit(data)
[muhat,sigmahat,muci,sigmaci] = normfit(data,alpha)
[...] = normfit(data,alpha,censoring)
[...] = normfit(data,alpha,censoring,freq)
[...] = normfit(data,alpha,censoring,freq,options)
```

Description

`[muhat,sigmahat] = normfit(data)` returns estimates of the mean, μ, and standard deviation, σ, of the normal distribution given the data in `data`.

`[muhat,sigmahat,muci,sigmaci] = normfit(data)` returns 95% confidence intervals for the parameter estimates on the mean and standard deviation in the arrays `muci` and `sigmaci`, respectively. The first row of `muci` contains the lower bounds of the confidence intervals for μ the second row contains the upper bounds. The first row of `sigmaci` contains the lower bounds of the confidence intervals for σ, and the second row contains the upper bounds.

`[muhat,sigmahat,muci,sigmaci] = normfit(data,alpha)` returns 100(1 - `alpha`) % confidence intervals for the parameter estimates, where `alpha` is a value in the range `[0 1]` specifying the width of the confidence intervals. By default, `alpha` is `0.05`, which corresponds to 95% confidence intervals.

`[...]  = normfit(data,alpha,censoring)` accepts a Boolean vector, `censoring`, of the same size as `data`, which is `1` for observations that are right-censored and `0` for observations that are observed exactly. `data` must be a vector in order to pass in the argument `censoring`.

`[...]  = normfit(data,alpha,censoring,freq)` accepts a frequency vector, `freq`, of the same size as `data`. Typically, `freq` contains integer frequencies for the corresponding elements in `data`, but can contain any nonnegative values. Pass in `[]` for `alpha`, `censoring`, or `freq` to use their default values.

`[...] = normfit(data,alpha,censoring,freq,options)` accepts a structure, `options`, that specifies control parameters for the iterative algorithm the function uses to compute maximum likelihood estimates when there is censoring. The normal fit function accepts an `options` structure which you can create using the function `statset`. Enter `statset('normfit')` to see the names and default values of the parameters that `normfit` accepts in the `options` structure. See the reference page for `statset` for more information about these options.

Example

In this example the data is a two-column random normal matrix. Both columns have $\mu = 10$ and $\sigma = 2$. Note that the confidence intervals below contain the "true values."

```
data = normrnd(10,2,100,2);
[mu,sigma,muci,sigmaci] = normfit(data)
mu =
  10.1455  10.0527
sigma =
  1.9072  2.1256
muci =
  9.7652  9.6288
  10.5258  10.4766
sigmaci =
  1.6745  1.8663
  2.2155  2.4693
```

See Also

`mle`, `statset`, `normlike`, `normpdf`, `normspec`, `normstat`, `normcdf`, `norminv`, `normplot`

Purpose

Inverse of normal cumulative distribution function

Syntax

```
X = norminv(P,mu,sigma)
[X,XLO,XUP] = norminv(P,mu,sigma,pcov,alpha)
```

Description

`X = norminv(P,mu,sigma)` computes the inverse of the normal cdf with parameters `mu` and `sigma` at the corresponding probabilities in `P`. `P`, `mu`, and `sigma` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs. The parameters in `sigma` must be positive, and the values in `P` must lie in the interval [0 1].

`[X,XLO,XUP] = norminv(P,mu,sigma,pcov,alpha)` produces confidence bounds for `X` when the input parameters `mu` and `sigma` are estimates. `pcov` is the covariance matrix of the estimated parameters. `alpha` specifies 100(1 - `alpha`)% confidence bounds. The default value of `alpha` is `0.05`. `XLO` and `XUP` are arrays of the same size as `X` containing the lower and upper confidence bounds.

The function `norminv` computes confidence bounds for `P` using a normal approximation to the distribution of the estimate

$$\hat{\mu} + \hat{\sigma}q$$

where q is the `P`th quantile from a normal distribution with mean 0 and standard deviation 1. The computed bounds give approximately the desired confidence level when you estimate `mu`, `sigma`, and `pcov` from large samples, but in smaller samples other methods of computing the confidence bounds may be more accurate.

The normal inverse function is defined in terms of the normal cdf as

$$x = F^{-1}(p|\mu, \sigma) = \{x : F(x|\mu, \sigma) = p\}$$

where

$$p = F(x|\mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}}\int_{-\infty}^{x} e^{\frac{-(t-\mu)^2}{2\sigma^2}} dt$$

The result, x, is the solution of the integral equation above where you supply the desired probability, p.

Examples

Find an interval that contains 95% of the values from a standard normal distribution.

```
x = norminv([0.025 0.975],0,1)
x =
  -1.9600  1.9600
```

Note that the interval `x` is not the only such interval, but it is the shortest.

```
xl = norminv([0.01 0.96],0,1)
xl =
  -2.3263  1.7507
```

The interval `xl` also contains 95% of the probability, but it is longer than `x`.

See Also

`icdf`, `normfit`, `normlike`, `normpdf`, `normspec`, `normstat`, `normcdf`, `normrnd`, `normplot`

Purpose

Negative log-likelihood for normal distribution

Syntax

```
nlogL = normlike(params,data)
[nlogL,AVAR] = normlike(params,data)
[...] = normlike(param,data,censoring)
[...] = normlike(param,data,censoring,freq)
```

Description

`nlogL = normlike(params,data)` returns the negative of the normal log-likelihood function for the parameters `params(1) = mu` and `params(2) = sigma`, given the vector `data`.

`[nlogL,AVAR] = normlike(params,data)` also returns the inverse of Fisher's information matrix, `AVAR`. If the input parameter values in `params` are the maximum likelihood estimates, the diagonal elements of `AVAR` are their asymptotic variances. `AVAR` is based on the observed Fisher's information, not the expected information.

`[...]  = normlike(param,data,censoring)` accepts a Boolean vector, `censoring`, of the same size as `data`, which is 1 for observations that are right-censored and 0 for observations that are observed exactly.

`[...]  = normlike(param,data,censoring,freq)` accepts a frequency vector, `freq`, of the same size as `data`. The vector `freq` typically contains integer frequencies for the corresponding elements in `data`, but can contain any nonnegative values. Pass in `[]` for `censoring` to use its default value.

`normlike` is a utility function for maximum likelihood estimation.

See Also

`betalike`, `gamlike`, `mle`, `normfit`, `wbllike`, `norminv`, `normpdf`, `normspec`, `normstat`, `normcdf`, `normrnd`, `normplot`

normpdf

Purpose

Normal probability density function

Syntax

```
Y = normpdf(X,mu,sigma)
```

Description

`Y = normpdf(X,mu,sigma)` computes the pdf at each of the values in `X` using the normal distribution with mean `mu` and standard deviation`sigma`. `X`, `mu`, and `sigma` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs. The parameters in `sigma` must be positive.

The normal pdf is

$$y = f(x|\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}}e^{\frac{-(x-\mu)^2}{2\sigma^2}}$$

The *likelihood function* is the pdf viewed as a function of the parameters. Maximum likelihood estimators (MLEs) are the values of the parameters that maximize the likelihood function for a fixed value of `x`.

The *standard normal* distribution has $\mu = 0$ and $\sigma = 1$.

If x is standard normal, then $x\sigma + \mu$ is also normal with mean μ and standard deviation σ. Conversely, if y is normal with mean μ and standard deviation σ, then $x = (y-\mu) / \sigma$ is standard normal.

Examples

```
mu = [0:0.1:2];
[y i] = max(normpdf(1.5,mu,1));
MLE = mu(i)
MLE =
  1.5000
```

See Also

`pdf`, `mvnpdf`, `normfit`, `norminv`, `normplot`, `normspec`, `normstat`, `normcdf`, `normrnd`, `normlike`

Purpose

Normal probability plot

Syntax

```
h = normplot(X)
```

Description

`h = normplot(X)` displays a normal probability plot of the data in `X`. For matrix `X`, normplot displays a line for each column of `X`. `h` is a handle to the plotted lines.

The plot has the sample data displayed with the plot symbol `'+'`. Superimposed on the plot is a line joining the first and third quartiles of each column of `X` (a robust linear fit of the sample order statistics.) This line is extrapolated out to the ends of the sample to help evaluate the linearity of the data.

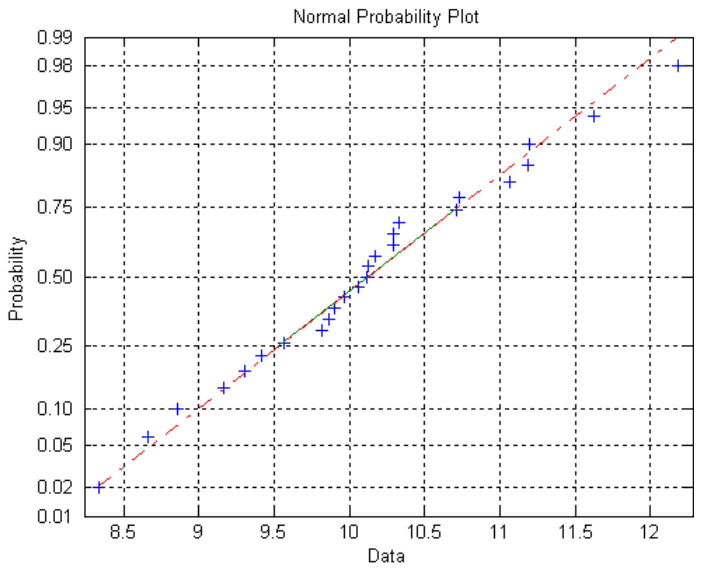
The purpose of a normal probability plot is to graphically assess whether the data in `X` could come from a normal distribution. If the data are normal the plot will be linear. Other distribution types will introduce curvature in the plot. `normplot` uses midpoint probability plotting positions. Use `probplot` when the data included censored observations.

If the data does come from a normal distribution, the plot will appear linear. Other probability density functions will introduce curvature in the plot.

Examples

Generate a normal sample and a normal probability plot of the data.

```
x = normrnd(10,1,25,1);
normplot(x)
```



See Also

`cdfplot`, `wblplot`, `probplot`,`hist`, `normfit`, `norminv`, `normpdf`, `normspec`, `normstat`, `normcdf`, `normrnd`, `normlike`

Purpose

Random numbers from normal distribution

Syntax

```
R = normrnd(mu,sigma)
R = normrnd(mu,sigma,v)
R = normrnd(mu,sigma,m,n)
```

Description

`R = normrnd(mu,sigma)` generates random numbers from the normal distribution with mean parameter `mu` and standard deviation parameter `sigma`. `mu` and `sigma` can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of `R`. A scalar input for `mu` or `sigma` is expanded to a constant array with the same dimensions as the other input.

`R = normrnd(mu,sigma,v)` generates random numbers from the normal distribution with mean parameter `mu` and standard deviation parameter `sigma`, where `v` is a row vector. If v is a 1-by-2 vector, `R` is a matrix with `v(1)` rows and `v(2)` columns. If `v` is 1-by-n, `R` is an n-dimensional array.

`R = normrnd(mu,sigma,m,n)` generates random numbers from the normal distribution with mean parameter `mu` and standard deviation parameter `sigma`, where scalars `m` and `n` are the row and column dimensions of `R`.

Example

```
n1 = normrnd(1:6,1./(1:6))
n1 =
  2.1650  2.3134  3.0250  4.0879  4.8607  6.2827

n2 = normrnd(0,1,[1 5])
n2 =
  0.0591  1.7971  0.2641  0.8717  -1.4462

n3 = normrnd([1 2 3;4 5 6],0.1,2,3)
n3 =
  0.9299  1.9361  2.9640
  4.1246  5.0577  5.9864
```

See Also normfit, norminv, normpdf, normspec, normstat, normcdf, normplot, normlike

Purpose

Plot normal density between specification limits

Syntax

```
p = normspec(specs,mu,sigma)
[p,h] = normspec(specs,mu,sigma)
```

Description

`p = normspec(specs,mu,sigma)` plots the normal density between a lower and upper limit defined by the two elements of the vector `specs`, where mu and sigma are the parameters of the plotted normal distribution.

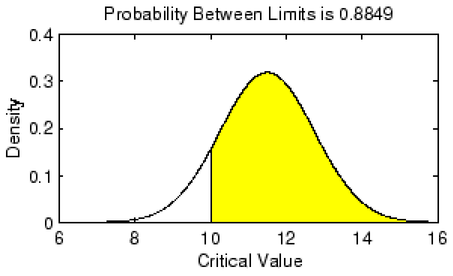
`[p,h] = normspec(specs,mu,sigma)` returns the probability `p` of a sample falling between the lower and upper limits. `h` is a handle to the line objects.

If `specs(1)` is `-Inf`, there is no lower limit, and similarly if `specs(2)` = `Inf`, there is no upper limit.

Example

Suppose a cereal manufacturer produces 10 ounce boxes of corn flakes. Variability in the process of filling each box with flakes causes a 1.25 ounce standard deviation in the true weight of the cereal in each box. The average box of cereal has 11.5 ounces of flakes. What percentage of boxes will have less than 10 ounces?

```
normspec([10 Inf],11.5,1.25)
```



See Also

`capaplot`, `disttool`, `histfit`, `normfit`, `norminv`, `normpdf`, `normrnd`, `normstat`, `normcdf`, `normplot`, , `normlike`

normstat

Purpose

Mean and variance of normal distribution

Syntax

```
[M,V] = normstat(mu,sigma)
```

Description

`[M,V] = normstat(mu,sigma)` returns the mean of and variance for the normal distribution with parameters `mu` and `sigma`. `mu` and `sigma` can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of `M` and `V`. A scalar input for `mu` or `sigma` is expanded to a constant array with the same dimensions as the other input.

The mean of the normal distribution with parameters μ and σ is μ, and the variance is σ^2.

Examples

```
n = 1:5;
[m,v] = normstat(n'*n,n'*n)
m =
     1     2     3     4     5
     2     4     6     8    10
     3     6     9    12    15
     4     8    12    16    20
     5    10    15    20    25

v =
     1     4     9    16    25
     4    16    36    64   100
     9    36    81   144   225
    16    64   144   256   400
    25   100   225   400   625
```

See Also

`normfit`, `norminv`, `normpdf`, `normrnd`, `normspec`, `normcdf`, `normplot`, `normlike`

Purpose

Number of segments of piecewise distribution

Syntax

```
n = nsegments(obj)
```

Description

`n = nsegments(obj)` returns the number of segments `n` in the piecewise distribution object `obj`.

Example

Fit Pareto tails to a *t* distribution at cumulative probabilities 0.1 and 0.9:

```
t = trnd(3,100,1);
obj = paretotails(t,0.1,0.9);

n = nsegments(obj)
n =
     3
```

See Also

`paretotails`, `boundary`, `segment`

numnodes

Purpose

Number of tree nodes

Syntax

```
n = numnodes(t)
```

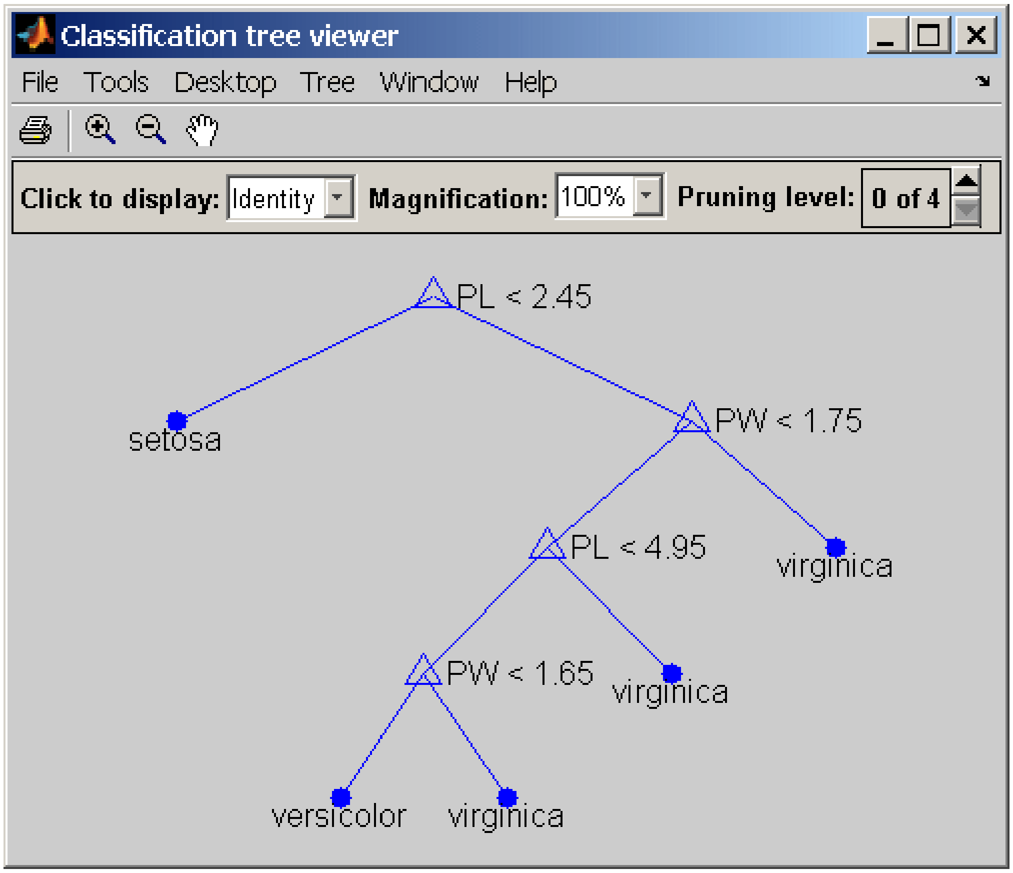
Description

`n = numnodes(t)` returns the number of nodes `n` in the tree `t`.

Example

Create a classification tree for Fisher's iris data:

```
load fisheriris;

t = classregtree(meas,species,'names',{'SL' 'SW' 'PL' 'PW'})
t =
Decision tree for classification
1  if PL<2.45 then node 2 else node 3
2  class = setosa
3  if PW<1.75 then node 4 else node 5
4  if PL<4.95 then node 6 else node 7
5  class = virginica
6  if PW<1.65 then node 8 else node 9
7  class = virginica
8  class = versicolor
9  class = virginica

view(t)
```



```
n = numnodes(t)
n =
      9
```

Reference [1] Breiman, L., et al., *Classification and Regression Trees*, Chapman & Hall, Boca Raton, 1993.

See Also `classregtree`

ordinal

Purpose

Create ordinal array

Syntax

```
B = ordinal(A)
B = ordinal(A,labels)
B = ordinal(A,labels,levels)
B = ordinal(A,labels,[],edges)
```

Description

`B = ordinal(A)` creates an ordinal array `B` from the array `A`. `A` can be numeric, logical, character, categorical, or a cell array of strings. `ordinal` creates the levels of `B` from the sorted unique values in `A`, and creates default labels for them.

`B = ordinal(A,labels)` labels the levels in `B` using the character array or cell array of strings `labels`. `ordinal` assigns labels to levels in `B` in order according to the sorted unique values in `A`.

`B = ordinal(A,labels,levels)` creates an ordinal array with possible levels defined by `levels`. `levels` is a vector whose values can be compared to those in `A` using the equality operator. `ordinal` assigns labels to each level from the corresponding elements of `labels`. If `A` contains any values not present in `levels`, the levels of the corresponding elements of `B` are undefined. Use `[]` for `labels` to allow `ordinal` to create default labels.

`B = ordinal(A,labels,[],edges)` creates an ordinal array by binning the numeric array `A`, with bin edges given by the numeric vector `edges`. The uppermost bin includes values equal to the right-most edge. `ordinal` assigns labels to each level in `B` from the corresponding elements of `labels`. `edges` must have one more element than `labels`.

By default, an element of `B` is undefined if the corresponding element of `A` is `NaN` (when `A` is numeric), an empty string (when `A` is character), or undefined (when `A` is categorical). `ordinal` treats such elements as “undefined” or “missing” and does not include entries for them among the possible levels for `B`. To create an explicit level for such elements instead of treating them as undefined, you must use the `levels` input, and include `NaN`, the empty string, or an undefined element.

You may include duplicate labels in `labels` in order to merge multiple values in `A` into a single level in `B`.

Examples

Example 1

Create an ordinal array with labels from random integer data:

```
x = floor(3*rand(1,1e3));
x(1:5)
ans =
     1     2     1     2     0

o = ordinal(x,{'I','II','III'});
o(1:5)
ans =
     II    III    II    III    I
```

Example 2

Create an ordinal array from the measurements in Fisher's iris data, ignoring decimal lengths:

```
load fisheriris
m = floor(min(meas(:)));
M = floor(max(meas(:)));
labels = num2str((m:M)');
edges = m:M+1;
cms = ordinal(meas,labels,[],edges)

meas(1:5,:)
ans =
    5.1000    3.5000    1.4000    0.2000
    4.9000    3.0000    1.4000    0.2000
    4.7000    3.2000    1.3000    0.2000
    4.6000    3.1000    1.5000    0.2000
    5.0000    3.6000    1.4000    0.2000
cms(1:5,:)
ans =
     5      3      1      0
     4     3     1     0
     4     3     1     0
     4     3     1     0
     5     3     1     0
```

Example 3

Create an age group ordinal array from the data in `hospital.mat`:

```
load hospital
edges = 0:10:100;
labels = strcat(num2str((0:10:90)','%d'),{'s'});
AgeGroup = ordinal(hospital.Age,labels,[],edges);

hospital.Age(1:5)
ans =
    38
    43
    38
    40
    49

AgeGroup(1:5)
ans =
     30s
     40s
     30s
     40s
     40s
```

See Also

`nominal`, `histc`

Purpose Parallel coordinates plot for multivariate data

Syntax

```
parallelcoords(X)
parallelcoords(X,...,'Standardize','on')
parallelcoords(X,...,'Standardize','PCA')
parallelcoords(X,...,'Standardize','PCAStd')
parallelcoords(X,...,'Quantile',alpha)
parallelcoords(X,...,'Group',group)
parallelcoords(X,...,'Labels',labels)
parallelcoords(X,...,PropertyName,PropertyValue,...)
h = parallelcoords(X,...)
```

Description

`parallelcoords(X)` creates a parallel coordinates plot of the multivariate data in the n-by-p matrix `X`. Rows of `X` correspond to observations, columns to variables. A parallel coordinates plot is a tool for visualizing high dimensional data, where each observation is represented by the sequence of its coordinate values plotted against their coordinate indices. `parallelcoords` treats `NaN`s in `X` as missing values and does not plot those coordinate values.

`parallelcoords(X,...,'Standardize','on')` scales each column of `X` to have mean 0 and standard deviation 1 before making the plot.

`parallelcoords(X,...,'Standardize','PCA')` creates a parallel coordinates plot from the principal component scores of `X`, in order of decreasing eigenvalues. `parallelcoords` removes rows of `X` containing missing values (`NaN`s) for principal components analysis (PCA) standardization.

`parallelcoords(X,...,'Standardize','PCAStd')` creates a parallel coordinates plot using the standardized principal component scores.

`parallelcoords(X,...,'Quantile',alpha)` plots only the median and the `alpha` and `1-alpha` quantiles of $f(t)$ at each value of t. This is useful if `X` contains many observations.

`parallelcoords(X,...,'Group',group)` plots the data in different groups with different colors. Groups are defined by `group`, a numeric array containing a group index for each observation. (See "Grouped

Data" on page 2-41.) group can also be a categorical variable, character matrix, or cell array of strings, containing a group name for each observation.

parallelcoords(X,...,'Labels',labels) labels the coordinate tick marks along the horizontal axis using labels, a character array or cell array of strings.

parallelcoords(X,...,*PropertyName*,*PropertyValue*,...) sets properties to the specified property values for all line graphics objects created by parallelcoords.

h = parallelcoords(X,...) returns a column vector of handles to the line objects created by parallelcoords, one handle per row of X. If you use the 'Quantile' input argument, h contains one handle for each of the three lines objects created. If you use both the 'Quantile' and the 'Group' input arguments, h contains three handles for each group.

Examples

```
% Make a grouped plot of the raw data
load fisheriris
labels = {'Sepal Length','Sepal Width',...
          'Petal Length','Petal Width'};
parallelcoords(meas,'group',species,'labels',labels);

% Plot only the median and quartiles of each group
parallelcoords(meas,'group',species,'labels',labels,...
               'quantile',.25);
```

See Also

andrewsplot, glyphplot

Purpose

Parent node of tree node

Syntax

```
p = parent(t)
p = parent(t,nodes)
```

Description

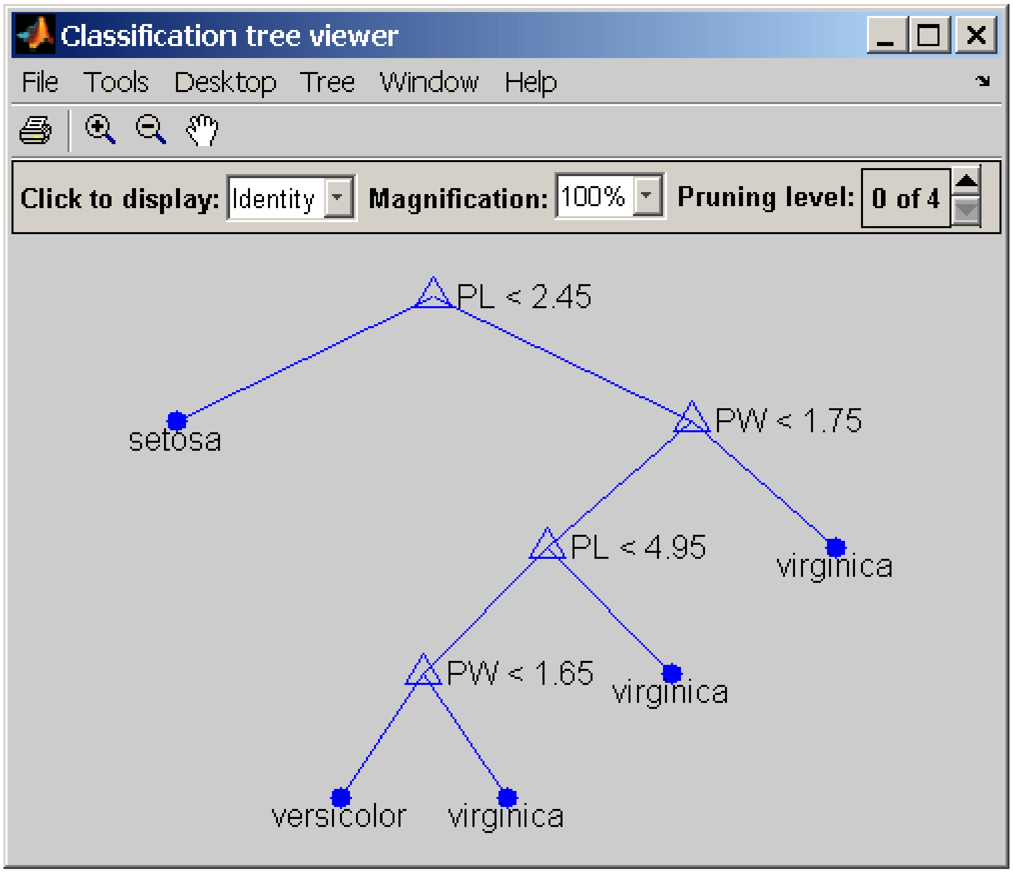
`p = parent(t)` returns an n-element vector `p` containing the number of the parent node for each node in the tree `t`, where n is the number of nodes. The parent of the root node is `0`.

`p = parent(t,nodes)` takes a vector `nodes` of node numbers and returns the parent nodes for the specified nodes.

Example

Create a classification tree for Fisher's iris data:

```
load fisheriris;

t = classregtree(meas,species,'names',{'SL' 'SW' 'PL' 'PW'})
t =
Decision tree for classification
1  if PL<2.45 then node 2 else node 3
2  class = setosa
3  if PW<1.75 then node 4 else node 5
4  if PL<4.95 then node 6 else node 7
5  class = virginica
6  if PW<1.65 then node 8 else node 9
7  class = virginica
8  class = versicolor
9  class = virginica

view(t)
```



```
p = parent(t)
p =
     0
     1
     1
     3
     3
     4
     4
     6
     6
```

Reference

[1] Breiman, L., et al., *Classification and Regression Trees*, Chapman & Hall, Boca Raton, 1993.

See Also

`classregtree`, `numnodes`, `children`

Purpose

Pareto chart

Syntax

```
pareto(y,names)
[h,ax] = pareto(...)
```

Description

`pareto(y,names)` displays a Pareto chart where the values in the vector `y` are drawn as bars in descending order. Each bar is labeled with the associated value in the string matrix or cell array, `names`. `pareto(y)` labels each bar with the index of the corresponding element in `y`.

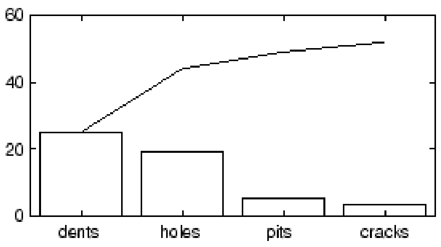
The line above the bars shows the cumulative percentage.

`[h,ax] = pareto(...)` returns a combination of patch and line object handles to the two axes created in `ax`.

Example

Create a Pareto chart from data measuring the number of manufactured parts rejected for various types of defects.

```
defects = {'pits';'cracks';'holes';'dents'};
quantity = [5 3 19 25];
pareto(quantity,defects)
```



See Also

`bar`, `hist`

Purpose

Construct Pareto tails object

Syntax

```
obj = paretotails(x,pl,pu)
obj = paretotails(x,pl,pu,cdffun)
```

Description

`obj = paretotails(x,pl,pu)` creates an object `obj` defining a distribution consisting of the empirical distribution of `x` in the center, and Pareto distributions in the tails. `x` is a real-valued vector of data values whose extreme observations are fit to generalized Pareto distributions (GPDs). `pl` and `pu` identify the lower and upper tail cumulative probabilities such that `100*pl` and `100*(1-pu)` percent of the observations in `x` are, respectively, fit to a GPD by maximum likelihood. If `pl` is `0` or if there are not at least two distinct observations in the lower tail, then no lower Pareto tail is fit. If `pu` is `1` or if there are not at least two distinct observations in the upper tail, then no upper Pareto tail is fit.

`obj = paretotails(x,pl,pu,`*`cdffun`*`)` uses *`cdffun`* to estimate the cdf of `x` between the lower and upper tail probabilities. *`cdffun`* may be any of the following:

- `'ecdf'` — Uses an interpolated empirical cdf, with data values as the midpoints in the vertical steps in the empirical cdf, and computed by linear interpolation between data values. This is the default.
- `'kernel'` — Uses a kernel smoothing estimate of the cdf .
- `@fun` — Uses a handle to a function of the form `[p,xi] = fun(x)` that accepts the input data vector `x` and returns a vector `p` of cdf values and a vector `xi` of evaluation points. Values in `xi` must be sorted and distinct but need not equal the values in `x`.

`cdffun` is used to compute the quantiles corresponding to `pl` and `pu` by inverse interpolation, and to define the fitted distribution between these quantiles.

The output object `obj` is a Pareto tails object with methods to evaluate the cdf, inverse cdf, and other functions of the fitted distribution. These methods are well-suited to copula and other Monte Carlo simulations.

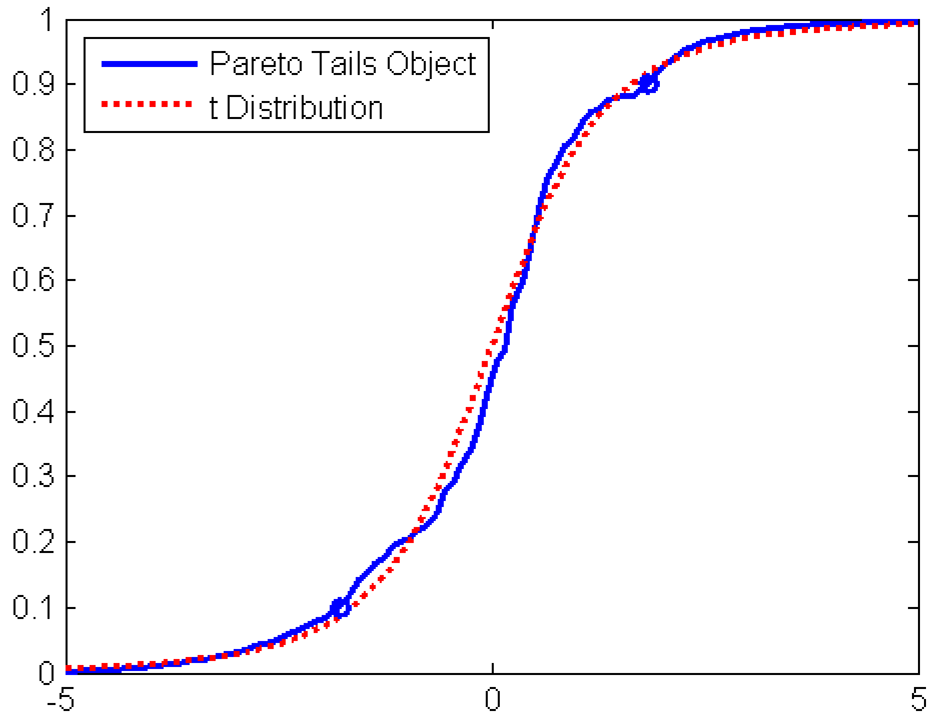
The pdf method in the tails is the GPD density, but in the center it is computed as the slope of the interpolated cdf.

The `paretotails` class is a subclass of the `piecewisedistribution` class, and many of its methods are derived from that class.

Example

Fit Pareto tails to a *t* distribution at cumulative probabilities 0.1 and 0.9:

```
t = trnd(3,100,1);
obj = paretotails(t,0.1,0.9);
[p,q] = boundary(obj);

x = linspace(-5,5);
plot(x,obj.cdf(x),'b-','LineWidth',2)
hold on
plot(x,tcdf(x,3),'r:','LineWidth',2)
plot(q,p,'bo','LineWidth',2,'MarkerSize',5)
legend('Pareto Tails Object','t Distribution','Location','NW')
```



See Also `ecdf`, `ksdensity`, `gpfit`, `cdf (piecewisedistribution)`, `icdf (piecewisedistribution)`

partialcorr

Purpose

Linear or rank partial correlation coefficients

Syntax

```
RHO = partialcorr(X,Z)
RHO = partialcorr(X,Y,Z)
[RHO,PVAL] = partialcorr(...)
[...] = partialcorr(...,param1,val1,param2,val2,...)
```

Description

`RHO = partialcorr(X,Z)` returns the sample linear partial correlation coefficients between pairs of variables in `X` controlling for the variables in `Z`. `X` is an n-by-p matrix, and `Z` is an n-by-q matrix with rows corresponding to observations, and columns corresponding to variables. The output, `RHO`, is a symmetric p-by-p matrix.

`RHO = partialcorr(X,Y,Z)` returns the sample linear partial correlation coefficients between pairs of variables between `X` and `Y`, controlling for the variables in `Z`. `X` is an n-by-$p1$ matrix, `Y` an n-by-$p2$ matrix, and `Z` is an n-by-q matrix, with rows corresponding to observations, and columns corresponding to variables. `RHO` is a $p1$-by-$p2$ matrix, where the (i, j)th entry is the sample linear partial correlation between the ith column in `X` and the jth column in `Y`.

If the covariance matrix of `[X,Z]` is

$$S = \begin{pmatrix} S_{11} & S_{12} \\ {S_{12}}^T & S_{22} \end{pmatrix}$$

then the partial correlation matrix of `X`, controlling for `Z`, can be defined formally as a normalized version of the covariance matrix

$$S\_xy = S_{11} - (S_{12} {S_{22}}^{-1} {S_{12}}^T)$$

`[RHO,PVAL] = partialcorr(...)` also returns `PVAL`, a matrix of p-values for testing the hypothesis of no partial correlation against the alternative that there is a nonzero partial correlation. Each element of `PVAL` is the p-value for the corresponding element of `RHO`. If `PVAL(i,j)` is small, say less than 0.05, then the partial correlation, `RHO(i,j)`, is significantly different from zero.

`[...] = partialcorr(...,param1,val1,param2,val2,...)` specifies additional parameters and their values. Valid parameters include the following:

Parameter	Values
`'type'`	`'Pearson'` (the default) to compute Pearson (linear) partial correlations or ’`Spearman'` to compute Spearman (rank) partial correlations.
`'rows'`	`'all'` (default) to use all rows regardless of missing values (`NaN`s), `'complete'` to use only rows with no missing values, or `'pairwise'` to compute `RHO(i,j)` using rows with no missing values in column i or j.
`'tail'`The alternative hypothesis against which to compute p-values for testing the hypothesis of no partial correlation.	• `'both'` (the default) — the correlation is not zero. • `'right'` — the correlation is greater than zero. • `'left'` — the correlation is less than zero.

The `'pairwise'` option for the `rows` parameter can produce `RHO` that is not positive definite. The `'complete'` option always produces a positive definite `RHO`, but when data is missing, the estimates will in general be based on fewer observations.

`partialcorr` computes p-values for linear and rank partial correlations using a Student’s t distribution for a transformation of the correlation. This is exact for linear partial correlation when `X` and `Z` are normal, but is a large-sample approximation otherwise.

See Also

`corr`, `tiedrank`, `corrcoef`

Purpose

Principal component analysis using covariance matrix

Syntax

```
COEFF = pcacov(V)
[COEFF,latent] = pcacov(V)
[COEFF,latent,explained] = pcacov(V)
```

Description

`COEFF = pcacov(V)` performs principal components analysis on the p-by-p covariance matrix `V` and returns the principal component coefficients, also known as loadings. `COEFF` is a p-by-p matrix, with each column containing coefficients for one principal component. The columns are in order of decreasing component variance.

`pcacov` does not standardize `V` to have unit variances. To perform principal components analysis on standardized variables, use the correlation matrix `R = V./(SD*SD'))`, where `SD = sqrt(diag(V))`, in place of `V`. To perform principal components analysis directly on the data matrix, use `princomp`.

`[COEFF,latent] = pcacov(V)` returns `latent`, a vector containing the principal component variances, that is, the eigenvalues of `V`.

`[COEFF,latent,explained] = pcacov(V)` returns `explained`, a vector containing the percentage of the total variance explained by each principal component.

Example

```
load hald
covx = cov(ingredients);
[COEFF,latent,explained] = pcacov(covx)
COEFF =
  0.0678 -0.6460  0.5673 -0.5062
  0.6785 -0.0200 -0.5440 -0.4933
 -0.0290  0.7553  0.4036 -0.5156
 -0.7309 -0.1085 -0.4684 -0.4844

variances =
  517.7969
  67.4964
  12.4054
```

```
    0.2372

  explained =
    86.5974
    11.2882
    2.0747
    0.0397
```

References

[1] Jackson, J. E., *A User's Guide to Principal Components*, John Wiley and Sons, 1991.

[2] Jolliffe, I. T., *Principal Component Analysis*, 2nd Edition, Springer, 2002.

[3] Krzanowski, W. J., *Principles of Multivariate Analysis*, Oxford University Press, 1988.

[4] Seber, G. A. F., *Multivariate Observations*, Wiley, 1984.

See Also

`barttest`, `biplot`, `factoran`, `pcares`, `princomp` , `rotatefactors`

Purpose

Residuals from principal component analysis

Syntax

```
residuals = pcares(X,ndim)
[residuals,reconstructed] = pcares(X,ndim)
```

Description

`residuals = pcares(X,ndim)` returns the `residuals` obtained by retaining `ndim` principal components of the n-by-p matrix `X`. Rows of `X` correspond to observations, columns to variables. `ndim` is a scalar and must be less than or equal to p. `residuals` is a matrix of the same size as `X`. Use the data matrix, *not* the covariance matrix, with this function.

`pcares` does not normalize the columns of X. To perform the principal components analysis based on standardized variables, that is, based on correlations, use `pcares(zscore(X), ndim)`. You can perform principal components analysis directly on a covariance or correlation matrix, but without constructing residuals, by using `pcacov`.

`[residuals,reconstructed] = pcares(X,ndim)` returns the reconstructed observations; that is, the approximation to `X` obtained by retaining its first `ndim` principal components.

Example

This example shows the drop in the residuals from the first row of the Hald data as the number of component dimensions increases from one to three.

```
load hald
r1 = pcares(ingredients,1);
r2 = pcares(ingredients,2);
r3 = pcares(ingredients,3);

r11 = r1(1,:)
r11 =
  2.0350  2.8304  -6.8378  3.0879

r21 = r2(1,:)
r21 =
  -2.4037  2.6930  -1.6482  2.3425
```

```
r31 = r3(1,:)
r31 =
  0.2008  0.1957  0.2045  0.1921
```

References

[1] Jackson, J. E., *A User's Guide to Principal Components*, John Wiley and Sons, 1991.

[2] Jolliffe, I. T., *Principal Component Analysis*, 2nd Edition, Springer, 2002.

[3] Krzanowski, W. J., *Principles of Multivariate Analysis*, Oxford University Press, 1988.

[4] Seber, G. A. F., *Multivariate Observations*, Wiley, 1984.

See Also

`factoran`, `pcacov`, `princomp`

Purpose

Probability density function for specified distribution

Syntax

```
Y = pdf(name,X,A)
Y = pdf(name,X,A,B)
Y = pdf(name,X,A,B,C)
```

Description

`Y = pdf(name,X,A)` computes the probability density function for the one-parameter family of distributions specified by `name`. Parameter values for the distribution are given in `A`. Densities are evaluated at the values in `X` and returned in `Y`.

If `X` and `A` are arrays, they must be the same size. If `X` is a scalar, it is expanded to a constant matrix the same size as `A`. If `A` is a scalar, it is expanded to a constant matrix the same size as `X`.

`Y` is the common size of `X` and `A` after any necessary scalar expansion.

`Y = pdf(name,X,A,B)` computes the probability density function for two-parameter families of distributions, where parameter values are given in `A` and `B`.

If `X`, `A`, and `B` are arrays, they must be the same size. If `X` is a scalar, it is expanded to a constant matrix the same size as `A` and `B`. If either `A` or `B` are scalars, they are expanded to constant matrices the same size as `X`.

`Y` is the common size of `X`, `A`, and `B` after any necessary scalar expansion.

`Y = pdf(name,X,A,B,C)` computes the probability density function for three-parameter families of distributions, where parameter values are given in `A`, `B`, and `C`.

If `X`, `A`, `B`, and `C` are arrays, they must be the same size. If `X` is a scalar, it is expanded to a constant matrix the same size as `A`, `B`, and `C`. If any of `A`, `B` or `C` are scalars, they are expanded to constant matrices the same size as `X`.

`Y` is the common size of `X`, `A`, `B` and `C` after any necessary scalar expansion.

Acceptable strings for `name` are:

- `'beta'` (Beta distribution)
- `'bino'` (Binomial distribution)
- `'chi2'` (Chi-square distribution)
- `'exp'` (Exponential distribution)
- `'ev'` (Extreme value distribution)
- `'f'` (*F* distribution)
- `'gam'` (Gamma distribution)
- `'gev'` (Generalized extreme value distribution)
- `'gp'` (Generalized Pareto distribution)
- `'geo'` (Geometric distribution)
- `'hyge'` (Hypergeometric distribution)
- `'logn'` (Lognormal distribution)
- `'nbin'` (Negative binomial distribution)
- `'ncf'` (Noncentral *F* distribution)
- `'nct'` (Noncentral *t*distribution)
- `'ncx2'` (Noncentral chi-square distribution)
- `'norm'` (Normal distribution)
- `'poiss'` (Poisson distribution)
- `'rayl'` (Rayleigh distribution)
- `'t'` (*t* distribution)
- `'unif'` (Uniform distribution)
- `'unid'` (Discrete uniform distribution)
- `'wbl'` (Weibull distribution)

Examples

```
p = pdf('Normal',-2:2,0,1)
p =
  0.0540  0.2420  0.3989  0.2420  0.0540

p = pdf('Poisson',0:4,1:5)
p =
  0.3679  0.2707  0.2240  0.1954  0.1755
```

See Also `cdf`, `icdf`, `mle`, `random`

Purpose

Probability density function for piecewise distribution

Syntax

```
P = pdf(obj,X)
```

Description

`P = pdf(obj,X)` returns an array `P` of values of the probability density function for the piecewise distribution object `obj`, evaluated at the values in the array `X`.

Note For a Pareto tails object, the pdf is computed using the generalized Pareto distribution in the tails. In the center, the pdf is computed using the slopes of the cdf, which are interpolated between a set of discrete values. Therefore the pdf in the center is piecewise constant. It is noisy for a *`cdffun`* specified in `paretotails` via the `'ecdf'` option, and somewhat smoother for the `'kernel'` option, but generally not a good estimate of the underlying density of the original data.

Example

Fit Pareto tails to a *t* distribution at cumulative probabilities 0.1 and 0.9:

```
t = trnd(3,100,1);
obj = paretotails(t,0.1,0.9);
[p,q] = boundary(obj)
p =
    0.1000
    0.9000
q =
   -1.7766
    1.8432

pdf(obj,q)
ans =
    0.2367
    0.1960
```

See Also

`paretotails`, `cdf (piecewisedistribution)`

Purpose

Pairwise distance between observations

Syntax

```
Y = pdist(X)
Y = pdist(X,metric)
Y = pdist(X,distfun)
Y = pdist(X,'minkowski',p)
```

Description

`Y = pdist(X)` computes the Euclidean distance between pairs of objects in n-by-p data matrix `X`. Rows of `X` correspond to observations; columns correspond to variables. `Y` is a row vector of length $n(n-1)/2$, corresponding to pairs of observations in `X`. The distances are arranged in the order (2,1), (3,1), ..., (n,1), (3,2), ..., (n,2), ..., (n,n–1)). `Y` is commonly used as a dissimilarity matrix in clustering or multidimensional scaling.

To save space and computation time, `Y` is formatted as a vector. However, you can convert this vector into a square matrix using the `squareform` function so that element i, j in the matrix, where $i < j$, corresponds to the distance between objects i and j in the original data set.

`Y = pdist(X,`*`metric`*`)` computes the distance between objects in the data matrix, `X`, using the method specified by *`metric`*, which can be any of the following character strings.

`'euclidean'`	Euclidean distance (default)
`'seuclidean'`	Standardized Euclidean distance. Each coordinate in the sum of squares is inverse weighted by the sample variance of that coordinate.
`'mahalanobis'`	Mahalanobis distance
`'cityblock'`	City Block metric
`'minkowski'`	Minkowski metric
`'cosine'`	One minus the cosine of the included angle between points (treated as vectors)

`'correlation'`	One minus the sample correlation between points (treated as sequences of values).
`'spearman'`	One minus the sample Spearman's rank correlation between observations, treated as sequences of values
`'hamming'`	Hamming distance, the percentage of coordinates that differ
`'jaccard'`	One minus the Jaccard coefficient, the percentage of nonzero coordinates that differ
`'chebychev'`	Chebychev distance (maximum coordinate difference)

`Y = pdist(X,distfun)` accepts a function handle `distfun` to a metric of the form

```
d = distfun(u,V)
```

which takes as arguments a 1-by-p vector `u`, corresponding to a single row of `X`, and an m-by-p matrix `V`, corresponding to multiple rows of `X`. `distfun` must accept a matrix `V` with an arbitrary number of rows. `distfun` must return an m-by-1 vector of distances `d`, whose kth element is the distance between `u` and `V(k,:)`.

"Parameterizing Functions Called by Function Functions", in the MATLAB Mathematics documentation, explains how to provide the additional parameters to the distance function, if necessary.

`Y = pdist(X,'minkowski',p)` computes the distance between objects in the data matrix, `X`, using the Minkowski metric. `p` is the exponent used in the Minkowski computation which, by default, is 2.

Metric Definitions

Given an m-by-n data matrix `X`, which is treated as m (1-*by*-n) row vectors x_1, x_2, ..., x_m, the various distances between the vector x_r and x_s are defined as follows:

- Euclidean distance

$$d_{rs}^2 = (x_r - x_s)(x_r - x_s)'$$

- Standardized Euclidean distance

$$d_{rs}^2 = (x_r - x_s)D^{-1}(x_r - x_s)'$$

  where D is the diagonal matrix with diagonal elements given by v_j^2, which denotes the variance of the variable X_j over the m objects.

- Mahalanobis distance

$$d_{rs}^2 = (x_r - x_s)V^{-1}(x_r - x_s)'$$

  where V is the sample covariance matrix.

- City Block metric

$$d_{rs} = \sum_{j=1}^{n} \left|x_{rj} - x_{sj}\right|$$

- Minkowski metric

$$d_{rs} = \left\{ \sum_{j=1}^{n} \left|x_{rj} - x_{sj}\right|^p \right\}^{\frac{1}{p}}$$

  Notice that for the special case of $p = 1$, the Minkowski metric gives the City Block metric, and for the special case of $p = 2$, the Minkowski metric gives the Euclidean distance.

- Cosine distance

$$d_{rs} = \left(1 - x_r x'_s / (x'_r x_r)^{\frac{1}{2}} (x'_s x_s)^{\frac{1}{2}}\right)$$

- Correlation distance

$$d_{rs} = 1 - \frac{(x_r - \bar{x}_r)(x_s - \bar{x}_s)'}{[(x_r - \bar{x}_r)(x_r - \bar{x}_r)']^{\frac{1}{2}}[(x_s - \bar{x}_s)(x_s - \bar{x}_s)']^{\frac{1}{2}}}$$

where

$\bar{x}_r = \frac{1}{n}\sum_j x_{rj}$ and $\bar{x}_s = \frac{1}{n}\sum_j x_{sj}$

- Hamming distance

$$d_{rs} = (\#(x_{rj} \neq x_{sj})/n)$$

- Jaccard distance

$$d_{rs} = \frac{\#[(x_{rj} \neq x_{sj}) \wedge ((x_{rj} \neq 0) \vee (x_{sj} \neq 0))]}{\#[(x_{rj} \neq 0) \vee (x_{sj} \neq 0)]}$$

Examples

```
X = [1 2; 1 3; 2 2; 3 1]
X =
    1    2
    1    3
    2    2
    3    1

Y = pdist(X,'mahal')
Y =
  2.3452  2.0000  2.3452  1.2247  2.4495  1.2247

Y = pdist(X)
Y =
  1.0000  1.0000  2.2361  1.4142  2.8284  1.4142

squareform(Y)
ans =
         0    1.0000    1.0000    2.2361
    1.0000         0    1.4142    2.8284
    1.0000    1.4142         0    1.4142
    2.2361    2.8284    1.4142         0
```

See Also `cluster`, `clusterdata`, `cmdscale`, `cophenet`, `dendrogram`, `inconsistent`, `linkage`, `silhouette`, `squareform`

Purpose

Random numbers from Pearson system of distributions

Syntax

```
r = pearsrnd(mu,sigma,skew,kurt,m,n)
[r,type] = pearsrnd(...)
[r,type,c] = pearsrnd(...)
```

Description

`r = pearsrnd(mu,sigma,skew,kurt,m,n)` returns an `m`-by-`n` matrix of random numbers drawn from the distribution in the Pearson system with mean `mu`, standard deviation `sigma`, skewness `skew`, and kurtosis `kurt`. `mu`, `sigma`, `skew`, and `kurt` must be scalars.

Note Because `r` is a random sample, its sample moments, especially the skewness and kurtosis, typically differ somewhat from the specified distribution moments.

Some combinations of moments are not valid for any random variable, and in particular, the kurtosis must be greater than the square of the skewness plus 1. The kurtosis of the normal distribution is defined to be 3.

`r = pearsrnd(mu,sigma,skew,kurt)` returns a scalar value.

`r = pearsrnd(mu,sigma,skew,kurt,m,n,...)` or `r = pearsrnd(mu,sigma,skew,kurt,[m,n,...])` returns an `m`-by-`n`-by-... array.

`[r,type] = pearsrnd(...)` returns the type of the specified distribution within the Pearson system. `type` is a scalar integer from `0` to `7`. Set `m` and `n` to zero to identify the distribution type without generating any random values.

The seven distribution types in the Pearson system correspond to the following distributions:

`0`	Normal distribution
`1`	Four-parameter beta

2	Symmetric four-parameter beta
3	Three-parameter gamma
4	Not related to any standard distribution. Density proportional to (1+((x-a)/b)^2)^(-c) * exp(-d*arctan((x-a)/b)).
5	Inverse gamma location-scale
6	F location-scale
7	Student's t location-scale

`[r,type,c] = pearsrnd(...)` returns the coefficients of the quadratic polynomial that defines the distribution via the differential equation

$$\frac{d(\mathit{log}(p(x)))}{dx} = \frac{(a + x)}{(c(0) + c(1) \cdot x + c(2) \cdot x^2)}.$$

Example

Generate random values from the standard normal distribution.

```
r = pearsrnd(0,1,0,3,100,1);  % equivalent to randn(100,1)
```

Determine the distribution type.

```
[r,type] = pearsrnd(0,1,1,4,0,0);
r =
     []
type =
     1
```

See Also

`random`, `johnsrnd`

Purpose

All permutations

Syntax

```
P = perms(v)
```

Description

`P = perms(v)` where `v` is a row vector of length `n`, creates a matrix whose rows consist of all possible permutations of the `n` elements of `v`. The matrix `P` contains `n!` rows and `n` columns.

`perms` is only practical when `n` is less than 8 or 9.

Example

```
perms([2 4 6])

ans =

    6    4    2
    6    2    4
    4    6    2
    4    2    6
    2    4    6
    2    6    4
```

poisscdf

Purpose

Poisson cumulative distribution function

Syntax

```
P = poisscdf(X,lambda)
```

Description

`P = poisscdf(X,lambda)` computes the Poisson cdf at each of the values in `X` using the corresponding parameters in `lambda`. `X` and `lambda` can be vectors, matrices, or multidimensional arrays that have the same size. A scalar input is expanded to a constant array with the same dimensions as the other input. The parameters in `lambda` must be positive.

The Poisson cdf is

$$p = F(x|\lambda) = e^{-\lambda} \sum_{i=0}^{floor(x)} \frac{\lambda^i}{i!}$$

Examples

For example, consider a Quality Assurance department that performs random tests of individual hard disks. Their policy is to shut down the manufacturing process if an inspector finds more than four bad sectors on a disk. What is the probability of shutting down the process if the mean number of bad sectors (λ) is two?

```
probability = 1-poisscdf(4,2)
probability =
  0.0527
```

About 5% of the time, a normally functioning manufacturing process will produce more than four flaws on a hard disk.

Suppose the average number of flaws (λ) increases to four. What is the probability of finding fewer than five flaws on a hard drive?

```
probability = poisscdf(4,4)
probability =
  0.6288
```

This means that this faulty manufacturing process continues to operate after this first inspection almost 63% of the time.

See Also

`cdf`, `poissfit`, `poissinv`, `poisspdf`, `poissrnd`, `poisstat`

Purpose

Parameter estimates and confidence intervals for Poisson distributed data

Syntax

```
lambdshat = poissfit(data)
[lambdahat,lambdaci] = poissfit(data)
[lambdahat,lambdaci] = poissfit(data,alpha)
```

Description

`lambdshat = poissfit(data)` returns the maximum likelihood estimate (MLE) of the parameter of the Poisson distribution, λ, given the data `data`.

`[lambdahat,lambdaci] = poissfit(data)` also gives 95% confidence intervals in `lamdaci`.

`[lambdahat,lambdaci] = poissfit(data,alpha)` gives `100(1 - alpha)`% confidence intervals. For example `alpha = 0.001` yields 99.9% confidence intervals.

The sample mean is the MLE of λ.

$$\hat{\lambda} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

Example

```
r = poissrnd(5,10,2);
[l,lci] = poissfit(r)
l =
    7.4000    6.3000
lci =
    5.8000    4.8000
    9.1000    7.9000
```

See Also

`betafit`, `binofit`, `expfit`, `gamfit`, `poisscdf`, `poissinv`, `poisspdf`, `poissrnd`, `poisstat`, `unifit`, `wblfit`

Purpose

Inverse of Poisson cumulative distribution function

Syntax

```
X = poissinv(P,lambda)
```

Description

`X = poissinv(P,lambda)` returns the smallest value `X` such that the Poisson cdf evaluated at `X` equals or exceeds `P`. `P` and `lambda` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other input.

Examples

If the average number of defects (λ) is two, what is the 95th percentile of the number of defects?

```
poissinv(0.95,2)
ans =
   5
```

What is the median number of defects?

```
median_defects = poissinv(0.50,2)
median_defects =
   2
```

See Also

`icdf`, `poisscdf`, `poissfit`, `poisspdf`, `poissrnd`, `poisstat`

Purpose

Poisson probability density function

Syntax

```
Y = poisspdf(X,lambda)
```

Description

`Y = poisspdf(X,lambda)` computes the Poisson pdf at each of the values in `X` using the corresponding parameters in `lambda`. `X` and `lambda` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other input. The parameters in `lambda` must all be positive.

The Poisson pdf is

$$y = f(x|\lambda) = \frac{\lambda^x}{x!}e^{-\lambda}I_{(0,1,\ldots)}(x)$$

where x can be any nonnegative integer. The density function is zero unless x is an integer.

Examples

A computer hard disk manufacturer has observed that flaws occur randomly in the manufacturing process at the average rate of two flaws in a 4 GB hard disk and has found this rate to be acceptable. What is the probability that a disk will be manufactured with no defects?

In this problem, $\lambda = 2$ and $x = 0$.

```
p = poisspdf(0,2)
p =
  0.1353
```

See Also

`pdf`, `poisscdf`, `poissfit`, `poissinv`, `poissrnd`, `poisstat`

Purpose

Random numbers from Poisson distribution

Syntax

```
R = poissrnd(lambda)
R = poissrnd(lambda,m)
R = poissrnd(lambda,m,n)
```

Description

`R = poissrnd(lambda)` generates random numbers from the Poisson distribution with mean parameter `lambda`. `lambda` can be a vector, a matrix, or a multidimensional array. The size of `R` is the size of `lambda`.

`R = poissrnd(lambda,m)` generates random numbers from the Poisson distribution with mean parameter `lambda`, where `m` is a row vector. If `m` is a 1-by-2 vector, `R` is a matrix with `m(1)` rows and `m(2)` columns. If `m` is 1-by-n, `R` is an n-dimensional array.

`R = poissrnd(lambda,m,n)` generates random numbers from the Poisson distribution with mean parameter `lambda`, where scalars `m` and `n` are the row and column dimensions of `R`.

Example

Generate a random sample of 10 pseudo-observations from a Poisson distribution with $\lambda = 2$.

```
lambda = 2;

random_sample1 = poissrnd(lambda,1,10)
random_sample1 =
   1   0   1   2   1   3   4   2   0   0

random_sample2 = poissrnd(lambda,[1 10])
random_sample2 =
   1   1   1   5   0   3   2   2   3   4

random_sample3 = poissrnd(lambda(ones(1,10)))
random_sample3 =
   3   2   1   1   0   0   4   0   2   0
```

See Also

`poisscdf`, `poissfit`, `poissinv`, `poisspdf`, `poisstat`

Purpose

Mean and variance of Poisson distribution

Syntax

```
M = poisstat(lambda)
[M,V] = poisstat(lambda)
```

Description

`M = poisstat(lambda)` returns the mean of the Poisson distribution with parameter `lambda`. The size of `M` is the size of `lambda`.

`[M,V] = poisstat(lambda)` also returns the variance `V` of the Poisson distribution.

For the Poisson distribution with parameter λ, both the mean and variance are equal to λ.

Examples

Find the mean and variance for the Poisson distribution with $\lambda = 2$.

```
[m,v] = poisstat([1 2; 3 4])
m =
   1   2
   3   4
v =
   1   2
   3   4
```

See Also

`poisscdf`, `poissfit`, `poissinv`, `poisspdf`, `poissrnd`

Purpose

Polynomial confidence intervals

Syntax

```
Y = polyconf(P,X)
[Y,DELTA] = polyconf(P,X,S)
[Y,DELTA] = polyconf(P,X,S,name1,val1,name2,val2,...)
```

Description

`Y = polyconf(P,X)` returns the value of a polynomial, `P`, evaluated at `X`. The polynomial `P` is a vector of length N+1 whose elements are the coefficients of the polynomial in descending powers.

`[Y,DELTA] = polyconf(P,X,S)` uses the optional output, `S`, created by `polyfit` to generate 95% prediction intervals. If the coefficients in `P` are least squares estimates computed by `polyfit`, and the errors in the data input to `polyfit` were independent, normal, with constant variance, then there is a 95% probability that `Y ± DELTA` will contain a future observation at `X`.

`[Y,DELTA] = polyconf(P,X,S,name1,val1,name2,val2,...)` specifies optional argument name/value pairs chosen from the following list. Argument names are case insensitive and partial matches are allowed.

Name	Value
`'alpha'`	A value between 0 and 1 specifying a confidence level of `100*(1-alpha)`%. Default is `alpha=0.05` for 95% confidence.
`'mu'`	A two-element vector containing centering and scaling parameters as computed by `polyfit`. With this option, `polyconf` uses `(X-mu(1))/mu(2)` in place of `X`.
`'predopt'`	Either `'observation'` (the default) to compute intervals for predicting a new observation at `X`, or `'curve'` to compute confidence intervals for the polynomial evaluated at `X`.
`'simopt'`	Either `'off'` (the default) for nonsimultaneous bounds, or `'on'` for simultaneous bounds.

See Also `polyval`, `polytool`, `polyfit`, `invpred`, `polyvalm`

Purpose

Polynomial fitting

Syntax

```
p = polyfit(x,y,n)
[p,S] = polyfit(x,y,n)
[p,S,mu] = polyfit(x,y,n)
```

Description

`p = polyfit(x,y,n)` finds the coefficients of a polynomial `p(x)` of degree `n` that fits the data, `p(x(i))` to `y(i)`, in a least squares sense. The result `p` is a row vector of length `n+1` containing the polynomial coefficients in descending powers:

$$p(x) = p_1x^n + p_2x^{n-1} + \ldots + p_nx + p_{n+1}$$

`[p,S] = polyfit(x,y,n)` returns the polynomial coefficients `p` and a structure `S` for use with `polyval` to obtain error estimates or predictions. `S` contains fields for the triangular factor (`R`) from a *QR* decomposition of the Vandermonde matrix of `x`, the degrees of freedom (`df`), and the norm of the residuals (`normr`). If the data is random, an estimate of the covariance matrix of `p` is `(Rinv*Rinv')*normr^2/df`, where `Rinv` is the inverse of `R`.

`[p,S,mu] = polyfit(x,y,n)` finds the coefficients of a polynomial in

$$\hat{x} = \frac{x - \mu_1}{\mu_2}$$

where μ_1 = mean(x) and μ_2 = std(x). `mu` is the two-element vector [μ_1, μ_2]. This centering and scaling transformation improves the numerical properties of both the polynomial and the fitting algorithm.

The `polyfit` function is part of the standard MATLAB language.

Example

Fitting a random data set to a first-order polynomial:

```
[p,S] = polyfit(1:10,[1:10] + normrnd(0,1,1,10),1)
p =
  1.1433  -0.7868
S =
  R: [2x2 double]
  df: 8
  normr: 2.3773
```

See Also polyval, polytool, polyconf

Purpose

Interactive plot of fitted polynomials and prediction intervals

Syntax

```
polytool(x,y)
polytool(x,y,n)
polytool(x,y,n,alpha)
polytool(x,y,n,alpha,xname,yname)
h = polytool(...)
```

Description

`polytool(x,y)` fits a line to the vectors `x` and `y` and displays an interactive plot of the result in a graphical interface. You can use the interface to explore the effects of changing the parameters of the fit and to export fit results to the workspace.

`polytool(x,y,n)` initially fits a polynomial of degree `n`. The default is `1`, which produces a linear fit.

`polytool(x,y,n,alpha)` initially plots `100(1 - alpha)`% confidence intervals on the predicted values. The default is `0.05` which results in 95% confidence intervals.

`polytool(x,y,n,alpha,xname,yname)` labels the `x` and `y` values on the graphical interface using the strings `xname` and `yname`. Specify `n` and `alpha` as `[]` to use their default values.

`h = polytool(...)` outputs a vector of handles, `h`, to the line objects in the plot. The handles are returned in the degree: data, fit, lower bounds, upper bounds.

Algorithm

polytool fits by least squares using the regression model

$$y_i = \beta_0 + \beta_1 x_i + \beta_2 x_i^2 + \ldots + \beta_n x_i^n + \varepsilon_i$$

$$\varepsilon_i \sim N(0, \sigma^2) \quad \forall i$$

$$Cov(\varepsilon_i, \varepsilon_j) = 0 \quad \forall i, j$$

Purpose

Polynomial values and prediction intervals

Syntax

```
Y = polyval(p,X)
[Y,DELTA] = polyval(p,X,S)
```

Description

`Y = polyval(p,X)` returns the value of a polynomial given its coefficients, `p`, at the values in `X`.

`[Y,DELTA] = polyval(p,X,S)` uses the optional output `S` generated by `polyfit` to generate error estimates, `Y` ± DELTA. If the errors in the data input to `polyfit` are independent normal with constant variance, `Y` ± DELTA contains at least 50% of future observations at X.

If `p` is a vector whose elements are the coefficients of a polynomial in descending powers, then `polyval(p,X)` is the value of the polynomial evaluated at `X`. If `X` is a matrix or vector, the polynomial is evaluated at each of the elements.

The `polyval` function is part of the standard MATLAB `language`.

Examples

Simulate the function $y = x$, adding normal random errors with a standard deviation of 0.1. Then use `polyfit` to estimate the polynomial coefficients. Note that predicted `Y` values are within DELTA of the integer `X` in every case.

```
[p,S] = polyfit(1:10,(1:10)+normrnd(0,0.1,1,10),1);
X = magic(3);
[Y,D] = polyval(p,X,S)
Y =
  8.0696  1.0486  6.0636
  3.0546  5.0606  7.0666
  4.0576  9.0726  2.0516

D =
  0.0889  0.0951  0.0861
  0.0889  0.0861  0.0870
  0.0870  0.0916  0.0916
```

See Also `polyfit`, `polytool`, `polyconf`

prctile

Purpose

Percentiles of sample

Syntax

```
Y = prctile(X,p)
Y = prctile(X,p,dim)
```

Description

`Y = prctile(X,p)` returns percentiles of the values in `X`. `p` is a scalar or a vector of percent values. When `X` is a vector, `Y` is the same size as `p` and `Y(i)` contains the `p(i)`th percentile. When `X` is a matrix, the `i`th row of `Y` contains the `p(i)`th percentiles of each column of `X`. For N-dimensional arrays, `prctile` operates along the first nonsingleton dimension of `X`.

`Y = prctile(X,p,dim)` calculates percentiles along dimension `dim`. The `dim`'th dimension of `Y` has length `length(p)`.

Percentiles are specified using percentages, from 0 to 100. For an n-element vector `X`, `prctile` computes percentiles as follows:

1. The sorted values in `X` are taken to be the $100(0.5/n)$, $100(1.5/n)$, ..., $100([n\text{-}0.5]/n)$ percentiles.
2. Linear interpolation is used to compute percentiles for percent values between $100(0.5/n)$ and $100([n\text{-}0.5]/n)$.
3. The minimum or maximum values in `X` are assigned to percentiles for percent values outside that range.

`prctile` treats `NaN`s as missing values and removes them.

Examples

```
x = (1:5)'*(1:5)
x =
    1    2    3    4    5
    2    4    6    8   10
    3    6    9   12   15
    4    8   12   16   20
    5   10   15   20   25

y = prctile(x,[25 50 75])
y =
    1.7500    3.5000    5.2500    7.0000    8.7500
    3.0000    6.0000    9.0000   12.0000   15.0000
    4.2500    8.5000   12.7500   17.0000   21.2500
```

Purpose

Principal component analysis

Syntax

```
[COEFF,SCORE] = princomp(X)
[COEFF,SCORE,latent] = princomp(X)
[COEFF,SCORE,latent,tsquare] = princomp(X)
[...] = princomp(X,'econ')
```

Description

`COEFF = princomp(X)` performs principal components analysis on the *n*-by-*p* data matrix `X`, and returns the principal component coefficients, also known as loadings. Rows of `X` correspond to observations, columns to variables. `COEFF` is a *p*-by-*p* matrix, each column containing coefficients for one principal component. The columns are in order of decreasing component variance.

`princomp` centers `X` by subtracting off column means, but does not rescale the columns of `X`. To perform principal components analysis with standardized variables, that is, based on correlations, use `princomp(zscore(X))`. To perform principal components analysis directly on a covariance or correlation matrix, use `pcacov`.

`[COEFF,SCORE] = princomp(X)` returns `SCORE`, the principal component scores; that is, the representation of `X` in the principal component space. Rows of `SCORE` correspond to observations, columns to components.

`[COEFF,SCORE,latent] = princomp(X)` returns `latent`, a vector containing the eigenvalues of the covariance matrix of `X`.

`[COEFF,SCORE,latent,tsquare] = princomp(X)` returns `tsquare`, which contains Hotelling's T^2 statistic for each data point.

The scores are the data formed by transforming the original data into the space of the principal components. The values of the vector `latent` are the variance of the columns of `SCORE`. Hotelling's T^2 is a measure of the multivariate distance of each observation from the center of the data set.

When `n <= p`, `SCORE(:,n:p)` and `latent(n:p)` are necessarily zero, and the columns of `COEFF(:,n:p)` define directions that are orthogonal to `X`.

`[...] = princomp(X,'econ')` returns only the elements of `latent` that are not necessarily zero, and the corresponding columns of `COEFF` and `SCORE`, that is, when `n <= p`, only the first `n-1`. This can be significantly faster when `p` is much larger than `n`.

Example

Compute principal components for the `ingredients` data in the Hald data set, and the variance accounted for by each component.

```
load hald;
[pc,score,latent,tsquare] = princomp(ingredients);
pc,latent
pc =
  0.0678 -0.6460  0.5673 -0.5062
  0.6785 -0.0200 -0.5440 -0.4933
 -0.0290  0.7553  0.4036 -0.5156
 -0.7309 -0.1085 -0.4684 -0.4844
latent =
 517.7969
  67.4964
  12.4054
   0.2372
```

References

[1] Jackson, J. E., *A User's Guide to Principal Components*, John Wiley and Sons, 1991, p. 592.

[2] Jolliffe, I. T., *Principal Component Analysis*, 2nd edition, Springer, 2002.

[3] Krzanowski, W. J., *Principles of Multivariate Analysis*, Oxford University Press, 1988.

[4] Seber, G. A. F., *Multivariate Observations*, Wiley, 1984.

See Also

`barttest`, `biplot`, `canoncorr`, `factoran`, `pcacov`, `pcares` , `rotatefactors`

Purpose

Probability plots

Syntax

```
probplot(Y)
probplot(distribution,Y)
probplot(Y,cens,freq)
probplot(ax,Y)
probplot(...,'noref')
probplot(ax,fun,params)
h = probplot(...)
```

Description

`probplot(Y)` produces a normal probability plot comparing the distribution of the data `Y` to the normal distribution. `Y` can be a single vector, or a matrix with a separate sample in each column. The plot includes a reference line useful for judging whether the data follow a normal distribution. `probplot` uses midpoint probability plotting positions.

`probplot(`*`distribution`*`,Y)` creates a probability plot for the distribution specified by *`distribution`*. Acceptable strings for *`distribution`* are:

- `'exponential'` — Exponential probability plot (nonnegative values)
- `'extreme value'` — Extreme value probability plot (all values)
- `'lognormal'` — Lognormal probability plot (positive values)
- `'normal'` — Normal probability plot (all values)
- `'rayleigh'` — Rayleigh probability plot (positive values)
- `'weibull'` — Weibull probability plot (positive values)

Not all distributions are appropriate for all data sets, and `probplot` will error when asked to create a plot with a data set that is inappropriate for a specified distribution. Appropriate data ranges for each distribution are given parenthetically in the list above.

`probplot(Y,cens,freq)` or `probplot(distname,Y,cens,freq)` requires a vector `Y`. `cens` is a vector of the same size as `Y` and contains

1 for observations that are right-censored and 0 for observations that are observed exactly. freq is a vector of the same size as Y, containing integer frequencies for the corresponding elements in Y.

probplot(ax,Y) takes a handle ax to an existing probability plot, and adds additional lines for the samples in Y. ax is a handle for a set of axes.

probplot(...,'noref') omits the reference line.

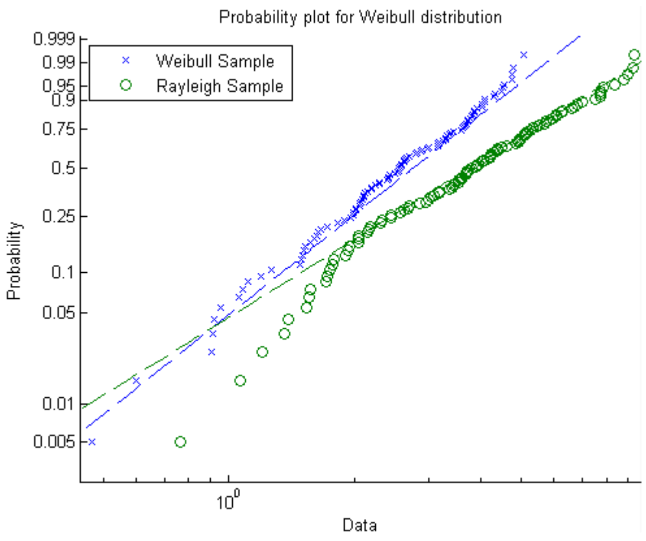
probplot(ax,fun,params) takes a function fun and a set of parameters, params, and adds fitted lines to the axes of an existing probability plot specified by ax. fun is a function handle to a cdf function, specified with @ (for example, @weibcdf). params is the set of parameters required to evaluate fun, and is specified as a cell array or vector. The function must accept a vector of X values as its first argument, then the optional parameters, and must return a vector of cdf values evaluated at X.

h = probplot(...) returns handles to the plotted lines.

Example

The following plot assesses two samples, one from a Weibull distribution and one from a Rayleigh distribution, to see if they may have come from a Weibull population.

```
x1 = wblrnd(3,3,100,1);
x2 = raylrnd(3,100,1);
probplot('weibull',[x1 x2])
legend('Weibull Sample','Rayleigh Sample','Location','NW')
```



See Also

`normplot`, `ecdf`, `wblplot`

Purpose

Procrustes analysis

Syntax

```
d = procrustes(X,Y)
[d,Z] = procrustes(X,Y)
[d,Z,transform] = procrustes(X,Y)
```

Description

`d = procrustes(X,Y)` determines a linear transformation (translation, reflection, orthogonal rotation, and scaling) of the points in matrix `Y` to best conform them to the points in matrix `X`. The goodness-of-fit criterion is the sum of squared errors. `procrustes` returns the minimized value of this dissimilarity measure in `d`. `d` is standardized by a measure of the scale of `X`, given by

```
sum(sum((X-repmat(mean(X,1),size(X,1),1)).^2,1))
```

i.e., the sum of squared elements of a centered version of `X`. However, if `X` comprises repetitions of the same point, the sum of squared errors is not standardized.

`X` and `Y` must have the same number of points (rows), and `procrustes` matches the `i`th point in `Y` to the `i`th point in `X`. Points in `Y` can have smaller dimension (number of columns) than those in `X`. In this case, `procrustes` adds columns of zeros to `Y` as necessary.

`[d,Z] = procrustes(X,Y)` also returns the transformed `Y` values.

`[d,Z,transform] = procrustes(X,Y)` also returns the transformation that maps `Y` to `Z`. `transform` is a structure with fields:

c	Translation component
T	Orthogonal rotation and reflection component
b	Scale component

That is, `Z = transform.b * Y * transform.T + transform.c`.

This example creates some random points in two dimensions, then rotates, scales, translates, and adds some noise to those points. It then uses procrustes to conform Y to X, and plots the original X and Y, and the transformed Y.

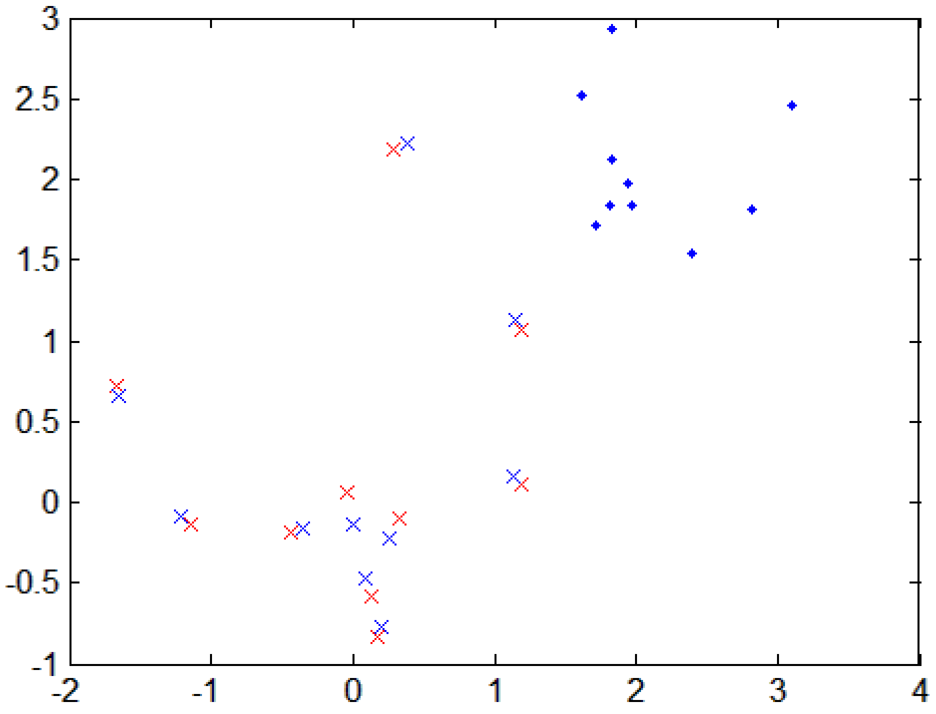
```
X = normrnd(0,1,[10 2]);
S = [0.5 -sqrt(3)/2; sqrt(3)/2 0.5];
Y = normrnd(0.5*X*S+2,0.05,size(X));
[d,Z,tr] = procrustes(X,Y);
plot(X(:,1),X(:,2),'rx',...
     Y(:,1),Y(:,2),'b.',...
     Z(:,1),Z(:,2),'bx');
```

Examples

Example 1

This example creates some random points in two dimensions, then rotates, scales, translates, and adds some noise to those points. It then uses procrustes to conform Y to X, and plots the original X and Y, and the transformed Y.

```
n = 10;
X = normrnd(0,1,[n 2]);
S = [0.5 -sqrt(3)/2; sqrt(3)/2 0.5];
Y = normrnd(0.5*X*S+2,0.05,n,2);
[d,Z,tr] = procrustes(X,Y);
plot(X(:,1),X(:,2),'rx',...
     Y(:,1),Y(:,2),'b.',...
     Z(:,1),Z(:,2),'bx');
```



Example 2

This example modifies the previous example to compute a procrustes solution that does not include scaling:

```
trUnscaled.T = tr.T;
trUnscaled.b = 1;
trUnscaled.c = mean(X) - mean(Y) * trUnscaled.T;
ZUnscaled = Y * trUnscaled.T + repmat(trUnscaled.c,n,1);
dUnscaled = sum((ZUnscaled(:)-X(:)).^2) ...
                   / sum(sum((X - repmat(mean(X,1),n,1)).^2, 1));
```

References

[1] Seber, G. A. F., *Multivariate Observations*, Wiley, 1984

[2] Bulfinch, T., *The Age of Fable; or, Stories of Gods and Heroes*, Sanborn, Carter, and Bazin, Boston, 1855.

See Also

`cmdscale`, `factoran`

Purpose

Produce subtrees by pruning

Syntax

```
t2 = prune(t1,'level',level)
t2 = prune(t1,'nodes',nodes)
t2 = prune(t1)
```

Description

`t2 = prune(t1,'level',level)` takes a decision tree `t1` and a pruning level `level`, and returns the decision tree `t2` pruned to that level. If `level` is 0, there is no pruning. Trees are pruned based on an optimal pruning scheme that first prunes branches giving less improvement in error cost.

`t2 = prune(t1,'nodes',nodes)` prunes the nodes listed in the `nodes` vector from the tree. Any `t1` branch nodes listed in `nodes` become leaf nodes in `t2`, unless their parent nodes are also pruned. Use `view` to display the node numbers for any node you select.

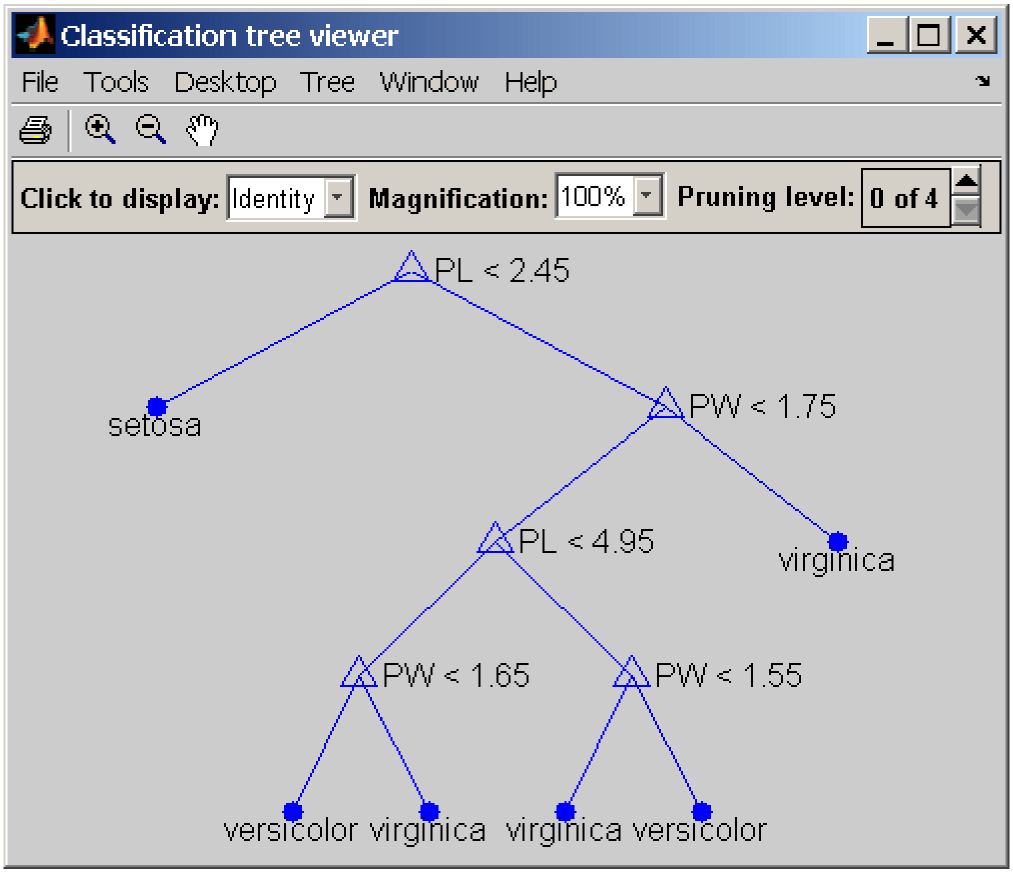
`t2 = prune(t1)` returns the decision tree `t2` that is the full, unpruned `t1`, but with optimal pruning information added. This is useful only if `t1` is created by pruning another tree, or by using the `classregtree` function with the `'prune'` parameter set to `'off'`. If you plan to prune a tree multiple times along the optimal pruning sequence, it is more efficient to create the optimal pruning sequence first.

Pruning is the process of reducing a tree by turning some branch nodes into leaf nodes and removing the leaf nodes under the original branch.

Example

Display the full tree for Fisher's iris data:

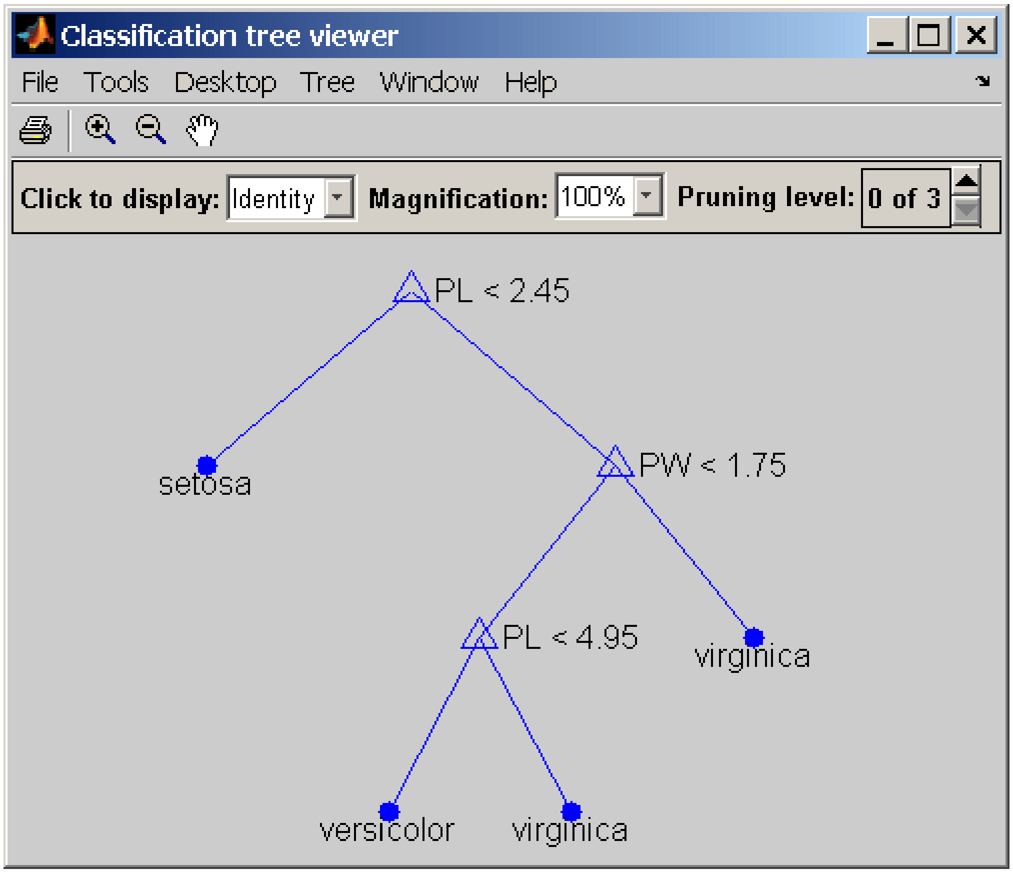
```
load fisheriris;

t1 = classregtree(meas,species,...
                  'names',{'SL' 'SW' 'PL' 'PW'},...
                  'splitmin',5)
t1 =
Decision tree for classification
 1  if PL<2.45 then node 2 else node 3
 2  class = setosa
 3  if PW<1.75 then node 4 else node 5
 4  if PL<4.95 then node 6 else node 7
 5  class = virginica
 6  if PW<1.65 then node 8 else node 9
 7  if PW<1.55 then node 10 else node 11
 8  class = versicolor
 9  class = virginica
10  class = virginica
11  class = versicolor

view(t1)
```


Classification tree viewer
File Tools Desktop Tree Window Help
Click to display: Identity
Magnification: 100%
Pruning level: 0 of 4
PL < 2.45
setosa
PW < 1.75
PL < 4.95
virginica
PW < 1.65
PW < 1.55
versicolor virginica virginica versicolor

Display the next largest tree from the optimal pruning sequence:

```
t2 = prune(t1,'level',1)
t2 =
Decision tree for classification
1  if PL<2.45 then node 2 else node 3
2  class = setosa
3  if PW<1.75 then node 4 else node 5
4  if PL<4.95 then node 6 else node 7
```

```
5  class = virginica
6  class = versicolor
7  class = virginica

view(t2)
```



Reference

[1] Breiman, L., et al., *Classification and Regression Trees*, Chapman & Hall, Boca Raton, 1993.

See Also

classregtree, test, view

Purpose

Quantile-quantile plot of two samples

Syntax

```
qqplot(X)
qqplot(X,Y)
qqplot(X,Y,pvec)
h = qqplot(X,Y,pvec)
```

Description

`qqplot(X)` displays a quantile-quantile plot of the sample quantiles of `X` versus theoretical quantiles from a normal distribution. If the distribution of `X` is normal, the plot will be close to linear.

`qqplot(X,Y)` displays a quantile-quantile plot of two samples. If the samples do come from the same distribution, the plot will be linear.

For matrix `X` and `Y`, qqplot displays a separate line for each pair of columns. The plotted quantiles are the quantiles of the smaller data set.

The plot has the sample data displayed with the plot symbol `'+'`. Superimposed on the plot is a line joining the first and third quartiles of each distribution (this is a robust linear fit of the order statistics of the two samples). This line is extrapolated out to the ends of the sample to help evaluate the linearity of the data.

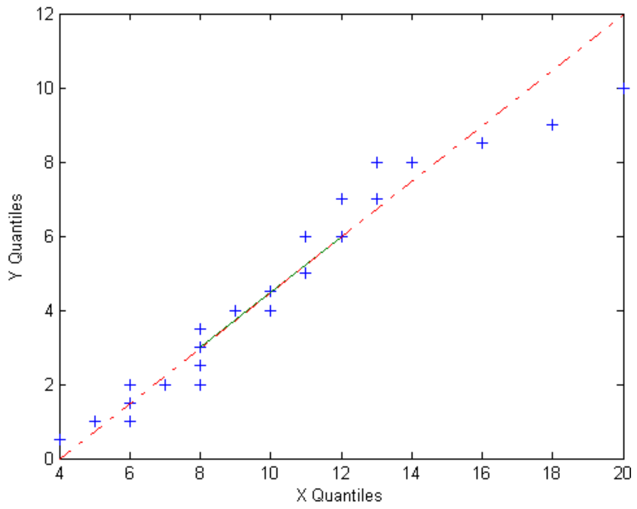
Use `qqplot(X,Y,pvec)` to specify the quantiles in the vector `pvec`.

`h = qqplot(X,Y,pvec)` returns handles to the lines in `h`.

Example

The following example shows a quantile-quantile plot of two samples from Poisson distributions.

```
x = poissrnd(10,50,1);
y = poissrnd(5,100,1);
qqplot(x,y);
```



See Also

`normplot`

Purpose

Quantiles of sample

Syntax

```
Y = quantile(X,p)
Y = quantile(X,p,dim)
```

Description

`Y = quantile(X,p)` returns quantiles of the values in `X`. `p` is a scalar or a vector of cumulative probability values. When `X` is a vector, `Y` is the same size as `p`, and `Y(i)` contains the `p(i)`th quantile. When `X` is a matrix, the *i*th row of `Y` contains the `p(i)`th quantiles of each column of `X`. For N-dimensional arrays, `quantile` operates along the first nonsingleton dimension of `X`.

`Y = quantile(X,p,dim)` calculates quantiles along dimension `dim`. The `dim`th dimension of `Y` has length `length(P)`.

Quantiles are specified using cumulative probabilities from 0 to 1. For an n-element vector `X`, quantile computes quantiles as follows:

1. The sorted values in `X` are taken as the $(0.5/n)$, $(1.5/n)$, ..., $([n\text{-}0.5]/n)$ quantiles.
2. Linear interpolation is used to compute quantiles for probabilities between $(0.5/n)$ and $([n\text{-}0.5]/n)$.
3. The minimum or maximum values in `X` are assigned to quantiles for probabilities outside that range.

`quantile` treats `NaN`s as missing values and removes them.

Examples

```
  y = quantile(x,.50); % the median of x
  y = quantile(x,[.025 .25 .50 .75 .975]); % Summary of x
```

See Also

`prctile`, `iqr`, `median`

Purpose

Gamma distributed random numbers and arrays (unit scale)

Syntax

```
Y = randg
Y = randg(A)
Y = randg(A,m)
Y = randg(A,m,n,...)
Y = randg(A,[m,n,...])
```

Description

`Y = randg` returns a scalar random value chosen from a gamma distribution with unit scale and shape.

`Y = randg(A)` returns a matrix of random values chosen from gamma distributions with unit scale. `Y` is the same size as `A`, and `randg` generates each element of `Y` using a shape parameter equal to the corresponding element of `A`.

`Y = randg(A,m)` returns an `m`-by-`m` matrix of random values chosen from gamma distributions with shape parameters `A`. `A` is either an `m`-by-`m` matrix or a scalar. If `A` is a scalar, `randg` uses that single shape parameter value to generate all elements of `Y`.

`Y = randg(A,m,n,...)` or `Y = randg(A,[m,n,...])` returns an `m`-by-`n`-by-... array of random values chosen from gamma distributions with shape parameters `A`. `A` is either an `m`-by-`n`-by-... array or a scalar.

`randg` produces pseudorandom numbers using the MATLAB functions `rand` and `randn`. The sequence of numbers generated is determined by the states of both generators. To create reproducible output from `randg`, set the states of both `rand` and `randn` to a fixed pair of values before calling `randg`. For example,

```
rand('state',j);
randn('state',s);
r = randg(1,[10,1]);
```

always generates the same 10 values. You can also use the MATLAB generators by calling `rand` and `randn` with the argument `'seed'`. Calling `randg` changes the current states of `rand` and `randn` and therefore alters the outputs of subsequent calls to those functions.

To generate gamma random numbers and specify both the scale and shape parameters, you should call `gamrnd` rather than calling `randg` directly.

References

[1] Marsaglia, G., and Tsang, W. W., "A Simple Method for Generating Gamma Variables," *ACM Transactions on Mathematical Software,* Vol. 26, 2000, pp. 363-372.

See Also

`gamrnd`

Purpose

Random numbers from specified distribution

Syntax

```
Y = random(name,A)
Y = random(name,A,B)
Y = random(name,A,B,C)
Y = random(...,m,n,...)
Y = random(...,[m,n,...])
```

Description

`Y = random(name,A)` returns random numbers `Y` from the one-parameter family of distributions specified by `name`. Parameter values for the distribution are given in `A`.

`Y` is the same size as `A`.

`Y = random(name,A,B)` returns random numbers `Y` from a two-parameter family of distributions. Parameter values for the distribution are given in `A` and `B`.

If `A` and `B` are arrays, they must be the same size. If either `A` or `B` are scalars, they are expanded to constant matrices of the same size.

`Y = random(name,A,B,C)` returns random numbers `Y` from a three-parameter family of distributions. Parameter values for the distribution are given in `A`, `B`, and `C`.

If `A`, `B`, and `C` are arrays, they must be the same size. If any of `A`, `B`, or `C` are scalars, they are expanded to constant matrices of the same size.

`Y = random(...,m,n,...)` or `Y = random(...,[m,n,...])` returns an `m`-by-`n`-by... matrix of random numbers.

If any of `A`, `B`, or `C` are arrays, then the specified dimensions must match the common dimensions of `A`, `B`, and `C` after any necessary scalar expansion.

Acceptable strings for `name` are:

- `'beta'` (Beta distribution)
- `'bino'` (Binomial distribution)
- `'chi2'` (Chi-square distribution)

- `'exp'` (Exponential distribution)
- `'ev'` (Extreme value distribution)
- `'f'` (*F* distribution)
- `'gam'` (Gamma distribution)
- `'gev'` (Generalized extreme value distribution)
- `'gp'` (Generalized Pareto distribution)
- `'geo'` (Geometric distribution)
- `'hyge'` (Hypergeometric distribution)
- `'logn'` (Lognormal distribution)
- `'nbin'` (Negative binomial distribution)
- `'ncf'` (Noncentral *F* distribution)
- `'nct'` (Noncentral *t*distribution)
- `'ncx2'` (Noncentral chi-square distribution)
- `'norm'` (Normal distribution)
- `'poiss'` (Poisson distribution)
- `'rayl'` (Rayleigh distribution)
- `'t'` (*t* distribution)
- `'unif'` (Uniform distribution)
- `'unid'` (Discrete uniform distribution)
- `'wbl'` (Weibull distribution)

Examples

```
rn = random('Normal',0,1,2,4)
rn =
  1.1650  0.0751  -0.6965  0.0591
  0.6268  0.3516  1.6961  1.7971

rp = random('Poisson',1:6,1,6)
rp =
   0   0   1   2   5   7
```

See Also

`cdf`, `pdf`, `icdf`, `mle`

random (piecewisedistribution)

Purpose Random numbers from piecewise distribution

Syntax

```
r = random(obj)
R = random(obj,n)
R = random(obj,m,n)
R = random(obj,[m,n])
R = random(obj,m,n,p,...)
R = random(obj,[m,n,p,...])
```

Description

`r = random(obj)` generates a pseudo-random number `r` drawn from the piecewise distribution object `obj`.

`R = random(obj,n)` generates an n-by-n matrix of pseudo-random numbers `R`.

`R = random(obj,m,n)` or `R = random(obj,[m,n])` generates an m-by-n matrix of pseudo-random numbers `R`.

`R = random(obj,m,n,p,...)` or `R = random(obj,[m,n,p,...])` generates an m-by-n-by-p-by-... array of pseudo-random numbers `R`.

Example

Fit Pareto tails to a *t* distribution at cumulative probabilities 0.1 and 0.9:

```
t = trnd(3,100,1);
obj = paretotails(t,0.1,0.9);

r = random(obj)
r =
    0.8285
```

See Also

`paretotails`, `cdf (piecewisedistribution)`, `icdf (piecewisedistribution)`

Purpose

Random sample, with or without replacement

Syntax

```
y = randsample(n,k)
y = randsample(population,k)
y = randsample(...,replace)
y = randsample(...,true,w)
```

Description

`y = randsample(n,k)` returns a `1`-by-`k` vector `y` of values sampled uniformly at random, without replacement, from the integers 1 to `n`.

`y = randsample(population,k)` returns `k` values sampled uniformly at random, without replacement, from the values in the vector `population`.

`y = randsample(...,replace)` returns a sample taken with replacement if `replace` is `true`, or without replacement if `replace` is `false`. The default is `false`.

`y = randsample(...,true,w)` returns a weighted sample taken with replacement, using a vector of positive weights `w`, whose length is `n`. The probability that the integer i is selected for an entry of `y` is `w(i)/sum(w)`. Usually, `w` is a vector of probabilities. `randsample` does not support weighted sampling without replacement.

Example

The following command generates a random sequence of the characters `A`, `C`, `G`, and `T`, with replacement, according to the specified probabilities.

```
R = randsample('ACGT',48,true,[0.15 0.35 0.35 0.15])
```

See Also

`rand`, `randperm`

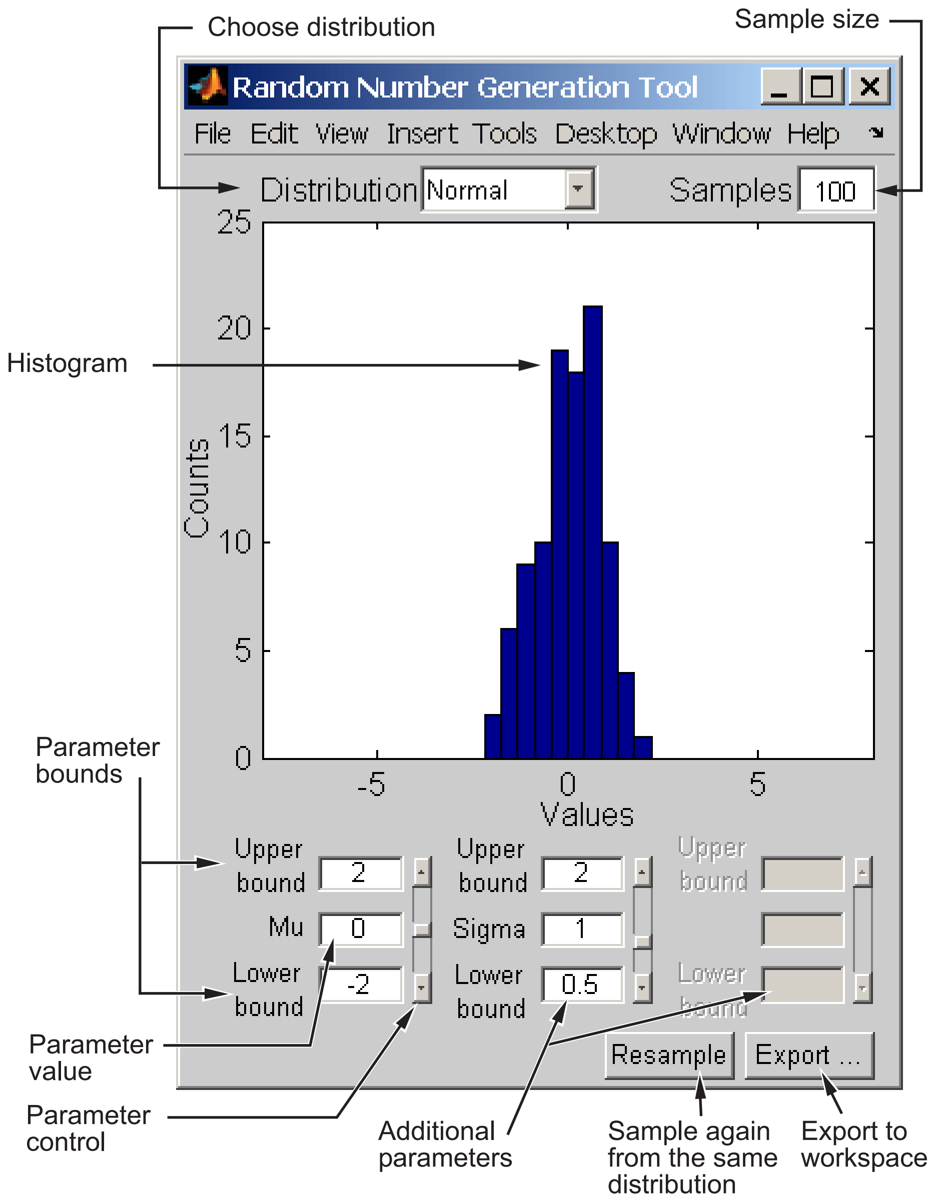
Purpose Interactive random number generation

Syntax

```
randtool
```

Description `randtool` opens the Random Number Generation Tool.

The Random Number Generation Tool is a graphical user interface that generates random samples from specified probability distributions and displays the samples as histograms. Use the tool to explore the effects of changing parameters and sample size on the distributions.


Choose distribution
Sample size
Random Number Generation Tool
File Edit View Insert Tools Desktop Window Help
Distribution Normal
Samples 100
Histogram
Counts
Values
Parameter bounds
Upper bound 2
Mu 0
Lower bound -2
Upper bound 2
Sigma 1
Lower bound 0.5
Upper bound
Lower bound
Resample
Export ...
Parameter value
Parameter control
Additional parameters
Sample again from the same distribution
Export to workspace

Start by selecting a distribution, then enter the desired sample size.

You can also

- Use the controls at the bottom of the window to set parameter values for the distribution and to change their upper and lower bounds.
- Draw another sample from the same distribution, with the same size and parameters.
- Export the current sample to your workspace. A dialog box enables you to provide a name for the sample.

See Also

`disttool`, `dfittool`

Purpose

Sample range

Syntax

```
range(X)
y = range(X,dim)
```

Description

`range(X)` returns the difference between the maximum and the minimum of a sample. For vectors, `range(x)` is the range of the elements. For matrices, `range(X)` is a row vector containing the range of each column of X. For N-dimensional arrays, `range` operates along the first nonsingleton dimension of `X`.

`y = range(X,dim)` operates along the dimension `dim` of `X`.

`range` treats `NaN`s as missing values and ignores them.

The range is an easily-calculated estimate of the spread of a sample. Outliers have an undue influence on this statistic, which makes it an unreliable estimator.

Example

The range of a large sample of standard normal random numbers is approximately six. This is the motivation for the process capability indices C_p and C_{pk} in statistical quality control applications.

```
rv = normrnd(0,1,1000,5);
near6 = range(rv)
near6 =
  6.1451  6.4986  6.2909  5.8894  7.0002
```

See Also

`std`, `iqr`, `mad`

Purpose

Wilcoxon rank sum test

Syntax

```
p = ranksum(x,y)
[p,h] = ranksum(x,y)
[p,h] = ranksum(x,y,'alpha',alpha)
[p,h] = ranksum(...,'method',method)
[p,h,stats] = ranksum(...)
```

Description

`p = ranksum(x,y)` performs a two-sided rank sum test of the null hypothesis that data in the vectors `x` and `y` are independent samples from identical continuous distributions with equal medians, against the alternative that they do not have equal medians. `x` and `y` can have different lengths. The *p*-value of the test is returned in `p`. The test is equivalent to a Mann-Whitney *U*-test.

`[p,h] = ranksum(x,y)` returns the result of the test in `h`. `h = 1` indicates a rejection of the null hypothesis at the 5% significance level. `h = 0` indicates a failure to reject the null hypothesis at the 5% significance level.

`[p,h] = ranksum(x,y,'alpha',alpha)` performs the test at the (100*`alpha`)% significance level. The default, when unspecified, is `alpha = 0.05`.

`[p,h] = ranksum(...,'method',`*`method`*`)` computes the *p*-value using either an exact algorithm, when *`method`* is `'exact'`, or a normal approximation, when *`method`* is `'approximate'`. The default, when unspecified, is the exact method for small samples and the approximate method for large samples.

`[p,h,stats] = ranksum(...)` returns the structure `stats` with the following fields:

- `ranksum` — Value of the rank sum test statistic
- `zval` — Value of the *z*-statistic (computed only for large samples)

Example

Test the hypothesis of equal medians for two independent unequal-sized samples. The sampling distributions are identical except for a shift of 0.25.

```
x = unifrnd(0,1,10,1);
y = unifrnd(0.25,1.25,15,1);
[p,h] = ranksum(x,y)
p =
  0.0375
h =
   1
```

The test rejects the null hypothesis of equal medians at the default 5% significance level.

References

[1] Gibbons, J. D., *Nonparametric Statistical Inference*, 2nd edition, M. Dekker, 1985.

[2] Hollander, M., and D. A. Wolfe, *Nonparametric Statistical Methods*, Wiley, 1973.

See Also

`kruskalwallis`, `signrank`, `signtest`, `ttest2`

Purpose Rayleigh cumulative distribution function

Syntax

```
P = raylcdf(X,B)
```

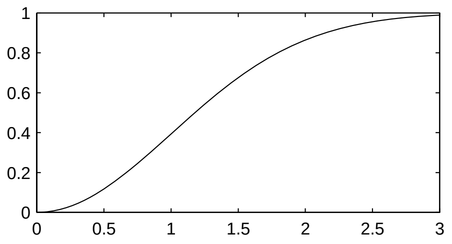
Description `P = raylcdf(X,B)` computes the Rayleigh cdf at each of the values in `X` using the corresponding parameters in B. `X` and `B` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input for `X` or `B` is expanded to a constant array with the same dimensions as the other input.

The Rayleigh cdf is

$$y = F(x|b) = \int_0^x \frac{t}{b^2} e^{\left(\frac{-t^2}{2b^2}\right)} dt$$

Example

```
x = 0:0.1:3;
p = raylcdf(x,1);
plot(x,p)
```



Reference [1] Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions, 2nd edition*, Wiley, 1993, pp. 134-136.

See Also `cdf`, `raylinv`, `raylpdf`, `raylrnd`, `raylstat`

Purpose

Parameter estimates and confidence intervals for Rayleigh distributed data

Syntax

```
raylfit(data,alpha)
[phat,pci] = raylfit(data,alpha)
```

Description

`raylfit(data,alpha)` returns the maximum likelihood estimates of the parameter of the Rayleigh distribution given the data in the vector `data`.

`[phat,pci] = raylfit(data,alpha)` returns the maximum likelihood estimate and 100(1 - `alpha`)% confidence interval given the data. The default value of the optional parameter `alpha` is 0.05, corresponding to 95% confidence intervals.

See Also

`raylcdf`, `raylinv`, `raylpdf`, `raylrnd`, `raylstat`, `mle`

Purpose

Inverse of Rayleigh cumulative distribution function

Syntax

```
X = raylinv(P,B)
```

Description

`X = raylinv(P,B)` returns the inverse of the Rayleigh cumulative distribution function with parameter `B` at the corresponding probabilities in `P`. `P` and `B` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input for `P` or `B` is expanded to a constant array with the same dimensions as the other input.

Example

```
x = raylinv(0.9,1)
x =
  2.1460
```

See Also

`icdf`, `raylcdf`, `raylpdf`, `raylrnd`, `raylstat`

Purpose

Rayleigh probability density function

Syntax

```
Y = raylpdf(X,B)
```

Description

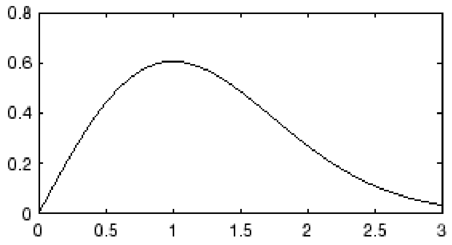
`Y = raylpdf(X,B)` computes the Rayleigh pdf at each of the values in `X` using the corresponding parameters in `B`. `X` and `B` can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of `Y`. A scalar input for `X` or `B` is expanded to a constant array with the same dimensions as the other input.

The Rayleigh pdf is

$$y = f(x|b) = \frac{x}{b^2}e^{\left(\frac{-x^2}{2b^2}\right)}$$

Example

```
x = 0:0.1:3;
p = raylpdf(x,1);
plot(x,p)
```



See Also

`pdf`, `raylcdf`, `raylinv`, `raylrnd`, `raylstat`

Purpose

Random numbers from Rayleigh distribution

Syntax

```
R = raylrnd(B)
R = raylrnd(B,v)
R = raylrnd(B,m,n)
```

Description

`R = raylrnd(B)` returns a matrix of random numbers chosen from the Rayleigh distribution with parameter `B`. `B` can be a vector, a matrix, or a multidimensional array. The size of `R` is the size of `B`.

`R = raylrnd(B,v)` returns a matrix of random numbers chosen from the Rayleigh distribution with parameter `B`, where `v` is a row vector. If `v` is a 1-by-2 vector, `R` is a matrix with `v(1)` rows and `v(2)` columns. If `v` is 1-by-n, `R` is an n-dimensional array.

`R = raylrnd(B,m,n)` returns a matrix of random numbers chosen from the Rayleigh distribution with parameter `B`, where scalars `m` and `n` are the row and column dimensions of `R`.

Example

```
  r = raylrnd(1:5)
  r =
    1.7986  0.8795  3.3473  8.9159  3.5182
```

See Also

`random`, `raylcdf`, `raylinv`, `raylpdf`, `raylstat`

Purpose

Mean and variance of Rayleigh distribution

Syntax

```
[M,V] = raylstat(B)
```

Description

`[M,V] = raylstat(B)` returns the mean of and variance for the Rayleigh distribution with parameter `B`.

The mean of the Rayleigh distribution with parameter b is $b\sqrt{\pi/2}$ and the variance is

$$\frac{4-\pi}{2}b^2$$

Example

```
[mn,v] = raylstat(1)
mn =
  1.2533
v =
  0.4292
```

See Also

`raylcdf`, `raylinv`, `raylpdf`, `raylrnd`

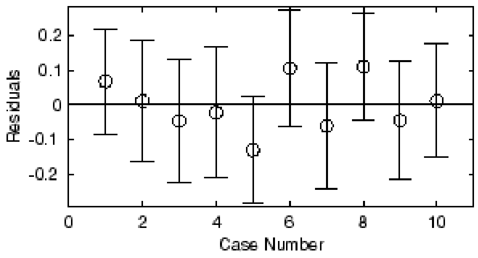
Purpose Residual case order plot

Syntax `rcoplot(r,rint)`

Description `rcoplot(r,rint)` displays an errorbar plot of the confidence intervals on the residuals from a regression. The residuals appear in the plot in case order. Inputs `r` and `rint` are outputs from the `regress` function.

Example

```
X = [ones(10,1) (1:10)'];
y = X*[10;1]+normrnd(0,0.1,10,1);
[b,bint,r,rint] = regress(y,X,0.05);
rcoplot(r,rint);
```



The figure shows a plot of the residuals with error bars showing 95% confidence intervals on the residuals. All the error bars pass through the zero line, indicating that there are no outliers in the data.

See Also `regress`

Purpose

Add polynomial to current plot

Syntax

```
h = refcurve(p)
```

Description

`refcurve` adds a graph of the polynomial `p` to the current axes. The function for a polynomial of degree n is:

$$y = p_1x^{\mathrm{n}} + p_2x^{(\mathrm{n}-1)} + \ldots + p_{\mathrm{n}}x + p_{\mathrm{n}+1}$$

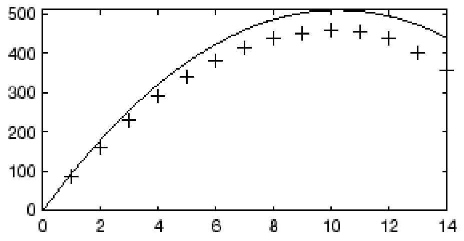
Note that p_1 goes with the highest order term.

`h = refcurve(p)` returns the handle to the curve.

Example

Plot data for the height of a rocket against time, and add a reference curve showing the theoretical height (assuming no air friction). The initial velocity of the rocket is 100 m/sec.

```
h = [85 162 230 289 339 381 413 ...
     437 452 458 456 440 400 356];
plot(h,'+')
refcurve([-4.9 100 0])
```



See Also

`polyfit`, `polyval`, `refline`

refline

Purpose

Add reference line to current axes

Syntax

```
refline(slope,intercept)
refline(slope)
h = refline(slope,intercept)
refline
```

Description

`refline(slope,intercept)` adds a reference line with the given slope and intercept to the current axes.

`refline(slope)`, where slope is a two-element vector, adds the line

```
y = slope(2)+slope(1)*x
```

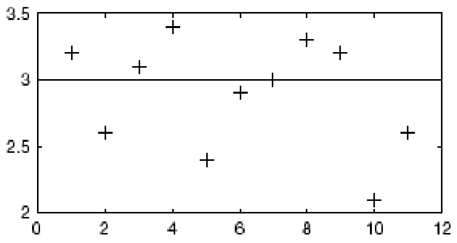
to the figure.

`h = refline(slope,intercept)` returns the handle to the line.

`refline` with no input arguments superimposes the least squares line on each line object in the current figure (except `LineStyles` `'-'`, `'--'`, `'.-'`). This behavior is equivalent to `lsline`.

Example

```
y = [3.2 2.6 3.1 3.4 2.4 2.9 3.0 3.3 3.2 2.1 2.6]';
plot(y,'+')
refline(0,3)
```



See Also

`lsline`, `polyfit`, `polyval`, `refcurve`

Purpose

Multiple linear regression

Syntax

```
b = regress(y,X)
[b,bint] = regress(y,X)
[b,bint,r] = regress(y,X)
[b,bint,r,rint] = regress(y,X)
[b,bint,r,rint,stats] = regress(y,X)
[...] = regress(y,X,alpha)
```

Description

`b = regress(y,X)` returns a p-by-1 vector `b` of coefficient estimates for a multilinear regression of the responses in `y` on the predictors in `X`. `X` is an n-by-p matrix of p predictors at each of n observations. `y` is an n-by-1 vector of observed responses.

`regress` treats `NaN`s in `X` or `y` as missing values, and ignores them.

If the columns of `X` are linearly dependent, `regress` obtains a basic solution by setting the maximum number of elements of `b` to zero.

`[b,bint] = regress(y,X)` returns a p-by-2 matrix `bint` of 95% confidence intervals for the coefficient estimates. The first column of `bint` contains lower confidence bounds for each of the p coefficient estimates; the second column contains upper confidence bounds.

If the columns of `X` are linearly dependent, `regress` returns zeros in elements of `bint` corresponding to the zero elements of `b`.

`[b,bint,r] = regress(y,X)` returns an n-by-1 vector `r` of residuals.

`[b,bint,r,rint] = regress(y,X)` returns an n-by-2 matrix `rint` of intervals that can be used to diagnose outliers. If the interval `rint(i,:)` for observation `i` does not contain zero, then the corresponding residual is larger than expected at the 5% significance level, suggesting an outlier.

`[b,bint,r,rint,stats] = regress(y,X)` returns a 1-by-4 vector `stats` that contains, in order, the R^2 statistic, the F statistic and its p-value, and an estimate of the error variance.

Note When computing statistics, `X` should include a column of ones so that the model contains a constant term. The F statistic and its p-value are computed under this assumption, and they are not correct for models without a constant. The R^2 statistic can be negative for models without a constant, indicating that the model is not appropriate for the data.

`[...] = regress(y,X,alpha)` uses a `100*(1-alpha)`% confidence level to compute `bint`, and a `(100*alpha)`% significance level to compute `rint`.

Example

Load data on cars; identify weight and horsepower as predictors, mileage as the response:

```
load carsmall
x1 = Weight;
x2 = Horsepower; % Contains NaN data
y = MPG;
```

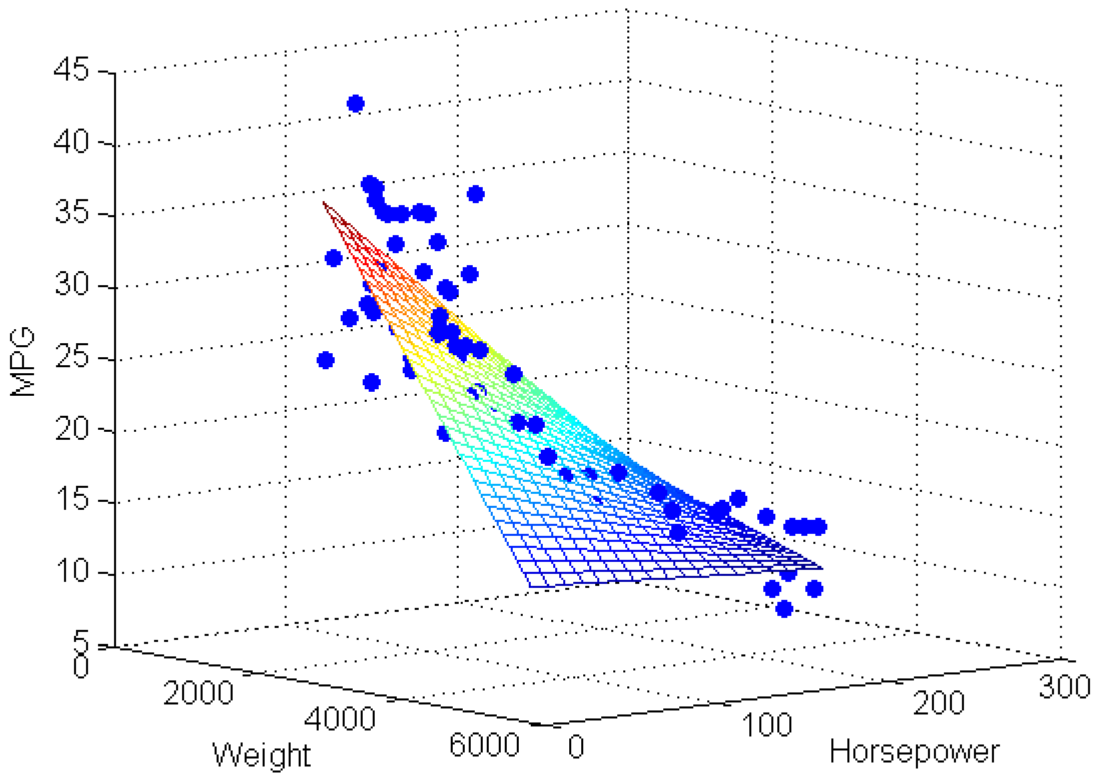
Compute regression coefficients for a linear model with an interaction term:

```
X = [ones(size(x1)) x1 x2 x1.*x2];
b = regress(y,X) % Removes NaN data
b =
  60.7104
  -0.0102
  -0.1882
   0.0000
```

Plot the data and the model:

```
scatter3(x1,x2,y,'filled')
hold on
x1fit = min(x1):100:max(x1);
x2fit = min(x2):10:max(x2);
[X1FIT,X2FIT] = meshgrid(x1fit,x2fit);
```

```
YFIT = b(1) + b(2)*X1FIT + b(3)*X2FIT + b(4)*X1FIT.*X2FIT;
mesh(X1FIT,X2FIT,YFIT)
xlabel('Weight')
ylabel('Horsepower')
zlabel('MPG')
view(50,10)
```



Reference

[1] Chatterjee, S., A. S. Hadi, “Influential Observations, High Leverage Points, and Outliers in Linear Regression,” *Statistical Science*, 1986, pp. 379- 416.

See Also

`regstats`, `mvregress`, `robustfit`, `stepwisefit`

regstats

Purpose

Regression diagnostics for linear models

Syntax

```
regstats(y,X,model)
stats = regstats(...)
stats = regstats(y,X,model,whichstats)
```

Description

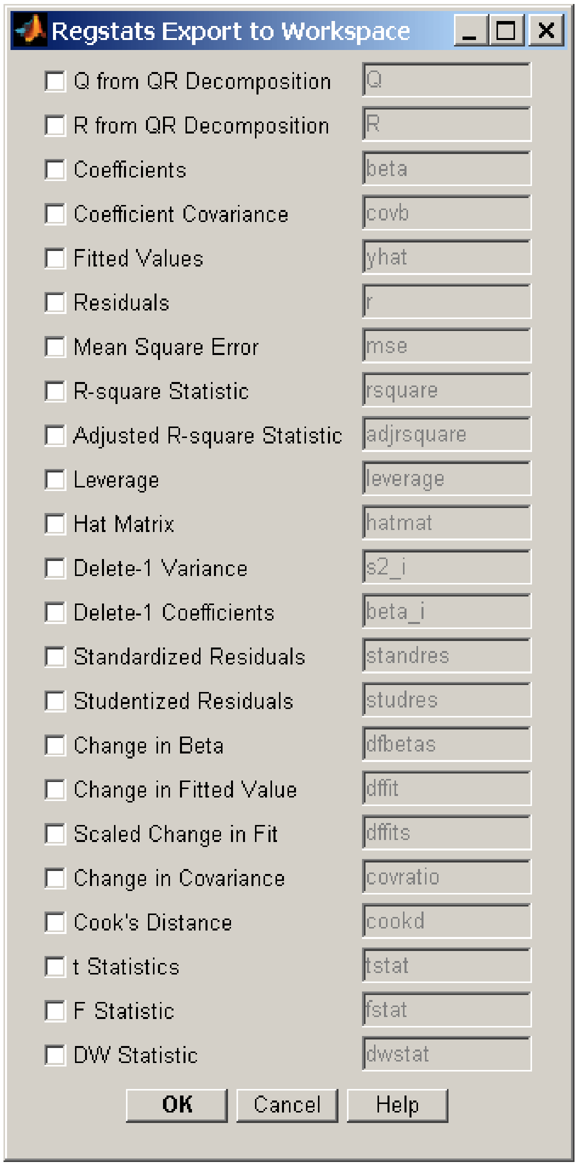
`regstats(y,X,model)` performs a multilinear regression of the responses in `y` on the predictors in `X`. `X` is an n-by-p matrix of p predictors at each of n observations. `y` is an n-by-1 vector of observed responses.

The optional input *`model`* controls the regression model. By default, `regstats` uses a linear additive model with a constant term. *`model`* can be any one of the following strings:

- `'linear'` — Constant and linear terms (the default)
- `'interaction'` — Constant, linear, and interaction terms
- `'quadratic'` — Constant, linear, interaction, and squared terms
- `'purequadratic'` — Constant, linear, and squared terms

To specify a polynomial model of arbitrary order, or a model without a constant term, use a matrix for `model` as described in `x2fx`.

With this syntax, the function displays a graphical user interface (GUI) with a list of diagnostic statistics, as shown in the following figure.


Regstats Export to Workspace

☐ Q from QR Decomposition	Q
☐ R from QR Decomposition	R
☐ Coefficients	beta
☐ Coefficient Covariance	covb
☐ Fitted Values	yhat
☐ Residuals	r
☐ Mean Square Error	mse
☐ R-square Statistic	rsquare
☐ Adjusted R-square Statistic	adjrsquare
☐ Leverage	leverage
☐ Hat Matrix	hatmat
☐ Delete-1 Variance	s2_i
☐ Delete-1 Coefficients	beta_i
☐ Standardized Residuals	standres
☐ Studentized Residuals	studres
☐ Change in Beta	dfbetas
☐ Change in Fitted Value	dffit
☐ Scaled Change in Fit	dffits
☐ Change in Covariance	covratio
☐ Cook's Distance	cookd
☐ t Statistics	tstat
☐ F Statistic	fstat
☐ DW Statistic	dwstat

When you select check boxes corresponding to the statistics you want to compute and click **OK**, `regstats` returns the selected statistics to the MATLAB workspace. The names of the workspace variables are displayed on the right-hand side of the interface. You can change the name of the workspace variable to any valid MATLAB variable name.

`stats = regstats(...)` creates the structure `stats`, whose fields contain all of the diagnostic statistics for the regression. This syntax does not open the GUI. The fields of `stats` are:

`Q`	Q from the QR decomposition of the design matrix
`R`	R from the QR decomposition of the design matrix
`beta`	Regression coefficients
`covb`	Covariance of regression coefficients
`yhat`	Fitted values of the response data
`r`	Residuals
`mse`	Mean squared error
`rsquare`	R^2 statistic
`adjrsquare`	Adjusted R^2 statistic
`leverage`	Leverage
`hatmat`	Hat matrix
`s2_i`	Delete-1 variance
`beta_i`	Delete-1 coefficients
`standres`	Standardized residuals
`studres`	Studentized residuals
`dfbetas`	Scaled change in regression coefficients
`dffit`	Change in fitted values
`dffits`	Scaled change in fitted values
`covratio`	Change in covariance

`cookd`	Cook's distance
`tstat`	t statistics for coefficients
`fstat`	F statistic

Note that the fields names of `stats` correspond to the names of the variables returned to the MATLAB workspace when you use the GUI. For example, `stats.beta` corresponds to the variable `beta` that is returned when you select **Coefficients** in the GUI and click **OK**.

`stats = regstats(y,X,`*`model`*`,`*`whichstats`*`)` returns only the statistics that you specify in *`whichstats`*. *`whichstats`* can be a single string such as `'leverage'` or a cell array of strings such as `{'leverage' 'standres' 'studres'}`. Set *`whichstats`* to `'all'` to return all of the statistics.

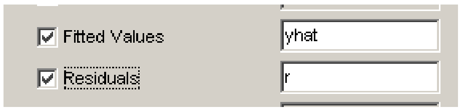
Note The F statistic is computed under the assumption that the model contains a constant term. It is not correct for models without a constant. The R^2 statistic can be negative for models without a constant, which indicates that the model is not appropriate for the data.

Example

Open the `regstats` GUI using data from `hald.mat`:

```
load hald
regstats(heat,ingredients,'linear');
```

Select **Fitted Values** and **Residuals** in the GUI:



☑ Fitted Values: yhat

☑ Residuals: r

Click **OK** to export the fitted values and residuals to the MATLAB workspace in variables named `yhat` and `r`, respectively.

You can create the same variables using the `stats` output, without opening the GUI:

```
whichstats = {'yhat','r'};
stats = regstats(heat,ingredients,'linear',whichstats);
yhat = stats.yhat;
r = stats.r;
```

Reference

[1] Belsley, D. A., E. Kuh, R. E. Welsch, *Regression Diagnostics*, Wiley, 1980.

[2] Chatterjee, S., A. S. Hadi, “Influential Observations, High Leverage Points, and Outliers in Linear Regression,” *Statistical Science*, 1986, pp. 379-416.

[3] Cook, R. D., S. Weisberg, *Residuals and Influence in Regression*, Wiley, 1982.

[4] Goodall, C. R., “Computation using the QR decomposition,” *Handbook in Statistics, Volume 9*, Elsevier/North-Holland, 1993.

See Also

`x2fx`, `regress`, `stepwise`, `leverage`

Purpose

Reorder levels of categorical array

Syntax

```
B = reorderlevels(A,newlevels)
```

Description

`B = reorderlevels(A,newlevels)` reorders the levels of the categorical array `A`. `newlevels` is a cell array of strings or a two-dimensional character matrix that specifies the new order. `newlevels` must be a reordering of `getlabels(A)`.

The order of the levels of an ordinal array has significance for relational operators, minimum and maximum, and for sorting.

Example

Reorder hockey standings:

```
standings = ordinal(1:3,{'Leafs','Canadiens','Bruins'});
getlabels(standings)
ans =
    'Leafs'  'Canadiens'  'Bruins'

standings = reorderlevels(standings,...
            {'Canadiens','Leafs','Bruins'});
getlabels(standings)
ans =
    'Canadiens'  'Leafs'  'Bruins'
```

See Also

`addlevels`, `droplevels`, `mergelevels`, `islevel`, `getlabels`

Purpose

Convert array to dataset variables

Syntax

```
B = replacedata(A,X)
B = replacedata(A,X,vars)
```

Description

`B = replacedata(A,X)` creates a dataset array `B` with the same variables as the dataset array `A`, but with the data for those variables replaced by the data in the array `X`. `replacedata` creates each variable in `B` using one or more columns from `X`, in order. `X` must have as many columns as the total number of columns in all of the variables in `A`, and as many rows as `A` has observations.

`B = replacedata(A,X,vars)` creates a dataset array `B` with the same variables as the dataset array `A`, but with the data for the variables specified in `vars` replaced by the data in the array `X`. The remaining variables in `B` are copies of the corresponding variables in `A`. `vars` is a positive integer, a vector of positive integers, a variable name, a cell array containing one or more variable names, or a logical vector. Each variable in `B` has as many columns as the corresponding variable in `A`. `X` must have as many columns as the total number of columns in all the variables specified in `vars`.

Example

Use `double` or `single` as complementary operations with `replacedata` when processing variables outside of a dataset array:

```
data = dataset({rand(3,3),'Var1','Var2','Var3'})
data =
    Var1       Var2       Var3
    0.81472    0.91338     0.2785
    0.90579    0.63236    0.54688
    0.12699    0.09754    0.95751
X = double(data,'Var2');
X = zscore(X);
data = replacedata(data,X,'Var2')
data =
    Var1       Var2       Var3
    0.81472    0.88219     0.2785
    0.90579     0.20413     0.54688
    0.12699     -1.0863     0.95751
```

See Also

`dataset`

Purpose

Ridge regression

Syntax

```
b = ridge(y,X,k)
b = ridge(y,X,k,scaled)
```

Description

`b = ridge(y,X,k)` returns a vector `b` of coefficient estimates for a multilinear ridge regression of the responses in `y` on the predictors in `X`. `X` is an n-by-p matrix of p predictors at each of n observations. `y` is an n-by-1 vector of observed responses. `k` is a vector of ridge parameters. If `k` has m elements, `b` is p-by-m. By default, `b` is computed after centering and scaling the predictors to have mean 0 and standard deviation 1. The model does not include a constant term, and `X` should not contain a column of ones.

`b = ridge(y,X,k,scaled)` uses the {0,1}-valued flag `scaled` to determine if the coefficient estimates in `b` are restored to the scale of the original data. `ridge(y,X,k,0)` performs this additional transformation. In this case, `b` contains p+1 coefficients for each value of `k`, with the first row corresponding to a constant term in the model. `ridge(y,X,k,1)` is the same as `ridge(y,X,k)`. In this case, `b` contains p coefficients, without a coefficient for a constant term.

The relationship between `b0 = ridge(y,X,k,0)` and `b1 = ridge(y,X,k,1)` is given by

```
m = mean(X);
s = std(X,0,1)';
b1_scaled = b1./s;
b0 = [mean(y)-m*temp; b1_scaled]
```

This can be seen by replacing the x_i (i = 1, ..., n) in the multilinear model $y = b_0^0 + b_1^0 x_1 + \ldots + b_n^0 x_n$ with the z-scores $z_i = (x_i - \mu_i)/\sigma_i$, and replacing y with $y - \mu_y$.

In general, `b1` is more useful for producing plots in which the coefficients are to be displayed on the same scale, such as a *ridge trace* (a plot of the regression coefficients as a function of the ridge parameter). `b0` is more useful for making predictions.

Ridge regression is used when the predictors in a multiple linear regression are correlated. This can arise, for example, when data are collected without an experimental design. When the columns of X are correlated, the correlation matrix $(X^TX)^{-1}$ may be close to singular. As a result, the least squares estimate

$$b = \hat{\beta} = (X^T X)^{-1} X^T y$$

becomes highly sensitive to random errors in y, producing a large variance in b.

Ridge regression addresses the issue by estimating regression coefficients using

$$b = \hat{\beta} = (X^T X + kI)^{-1} X^T y$$

where k is the ridge parameter and I is the identity matrix. Small positive values of k can be used to improve the conditioning of the problem, and reduce the variance of the estimates. While biased, the reduced variance of ridge estimates often result in a smaller mean square error when compared to least-squares estimates.

Example

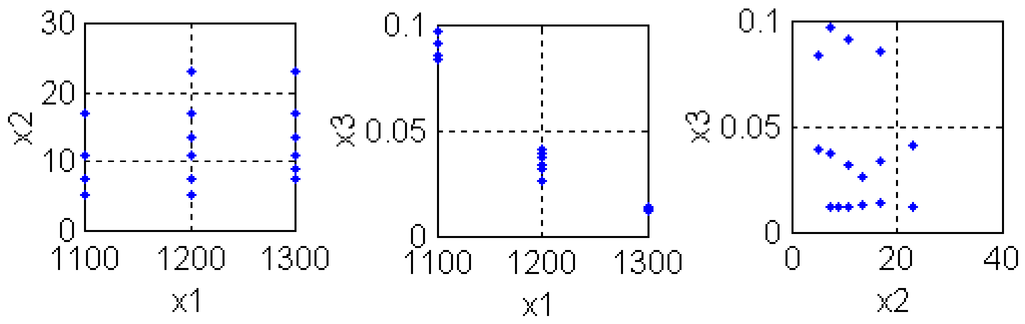
Load the data in `acetylene.mat`, with predictors `x1`, `x2`, `x3` and response `y`:

```
load acetylene
```

Plot the predictors against each other:

```
subplot(1,3,1)
plot(x1,x2,'.')
xlabel('x1'); ylabel('x2'); grid on; axis square

subplot(1,3,2)
plot(x1,x3,'.')
xlabel('x1'); ylabel('x3'); grid on; axis square
```

```
subplot(1,3,3)
plot(x2,x3,'.')
xlabel('x2'); ylabel('x3'); grid on; axis square
```



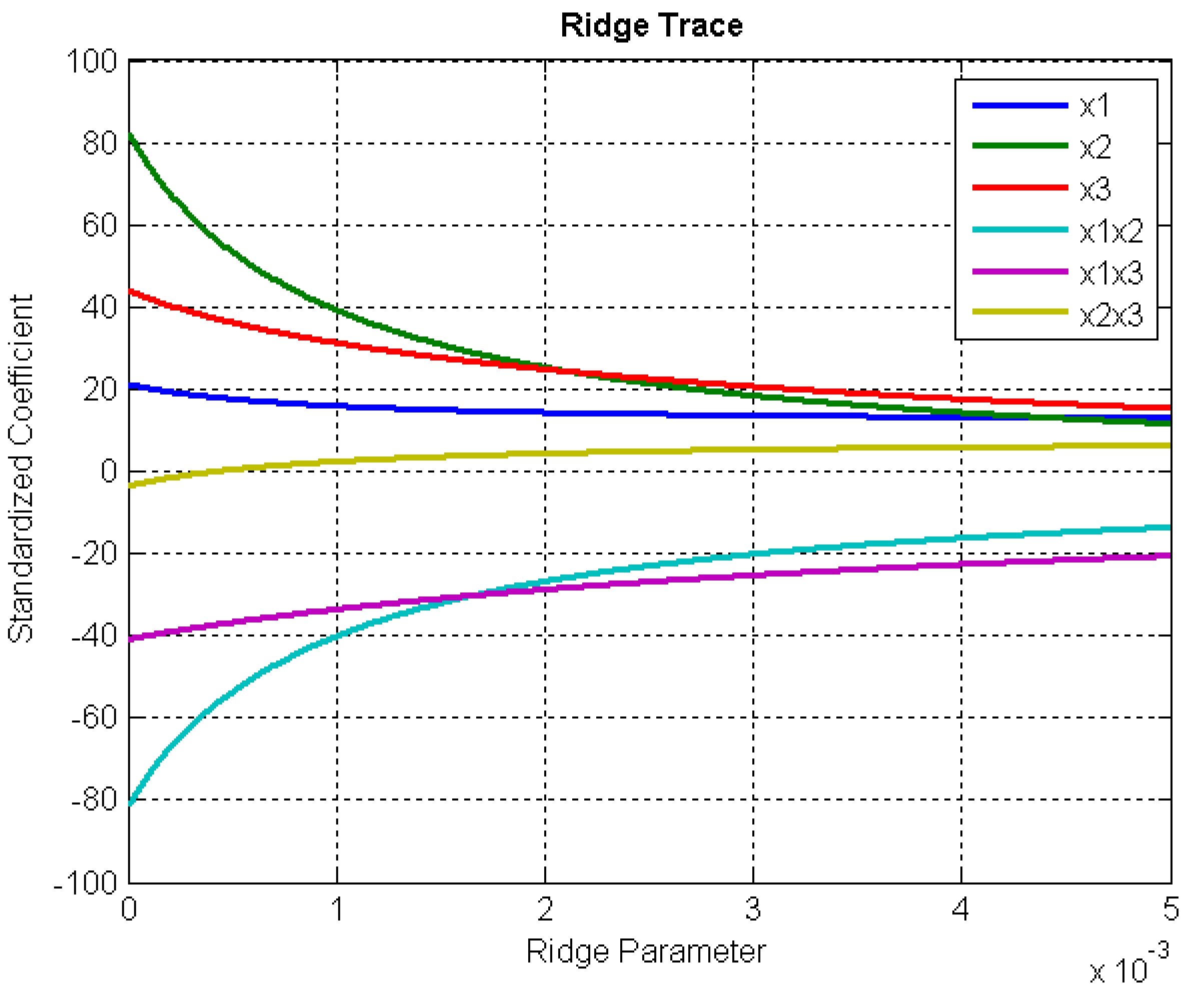
Note the correlation between `x1` and the other two predictors.

Compute coefficient estimates for a multilinear model with interaction terms, for a range of ridge parameters:

```
X = [x1 x2 x3];
D = x2fx(X,'interaction');
D(:,1) = []; % No constant term
k = 0:1e-5:5e-3;
b = ridge(y,D,k);
```

Plot the ridge trace:

```
figure
plot(k,b,'LineWidth',2)
ylim([-100 100])
grid on
xlabel('Ridge Parameter')
ylabel('Standardized Coefficient')
title('{\bf Ridge Trace}')
legend('constant','x1','x2','x3','x1x2','x1x3','x2x3')
```



The estimates stabilize to the right of the plot. Note that the `x2x3` interaction term changes sign at a value of the ridge parameter ~ 5 x 10^{-4}.

Reference

[1] Hoerl, A.E., R.W. Kennard, “Ridge Regression: Biased Estimation for Nonorthogonal Problems,” *Technometrics*, Vol. 12, Number 1, pp. 55-67, 1970.

[2] Hoerl, A.E., R.W. Kennard, “Ridge Regression: Applications to Nonorthogonal Problems,” *Technometrics*, Vol. 12, Number 1, pp. 69-82, 1970.

[3] Marquardt, D.W., “Generalized Inverses, Ridge Regression, Biased Linear Estimation, and Nonlinear Estimation,” *Technometrics*, Vol. 12, Number 3, pp. 591-612, 1970.

[4] Marquardt, D.W., R.D. Snee, “Ridge Regression in Practice,” *The American Statistician*, Vol. 29, Number 1, pp. 3-20, 1975.

See Also

`regress`, `stepwise`

Purpose

Node risks of tree

Syntax

```
r = risk(t)
r = risk(t,nodes)
```

Description

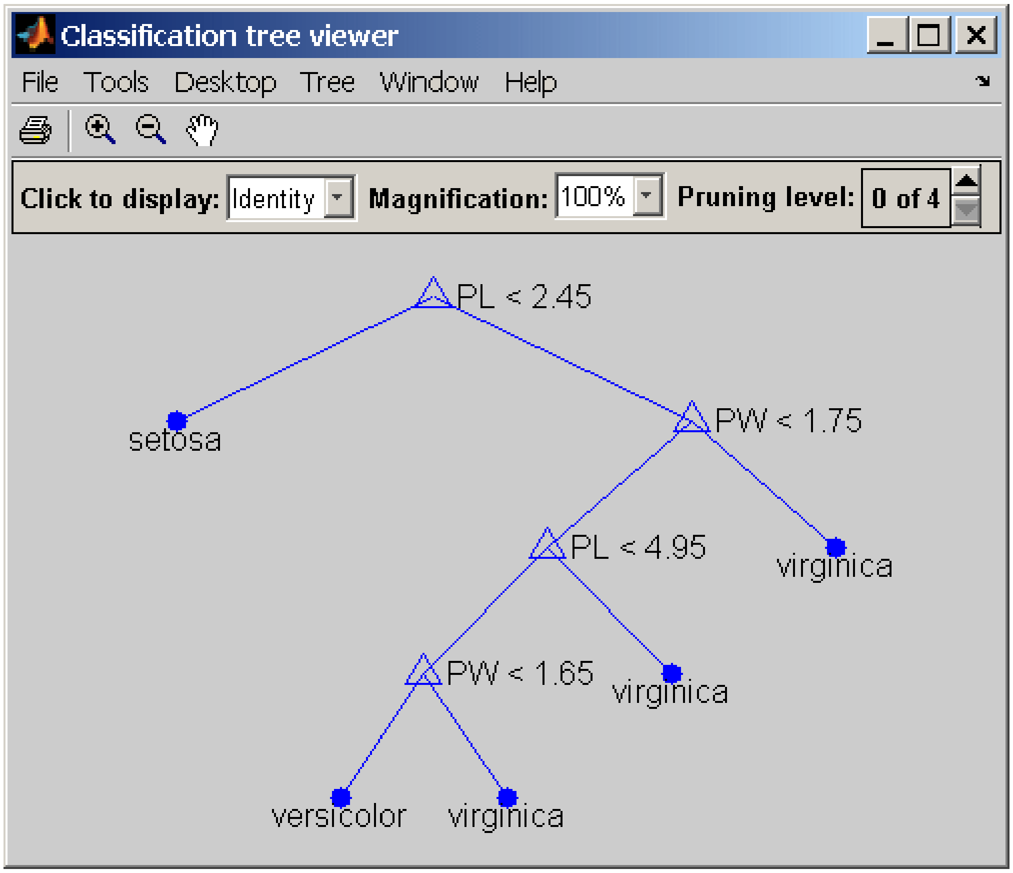
`r = risk(t)` returns an *n*-element vector `r` of the risk of the nodes in the tree `t`, where *n* is the number of nodes. The risk `r(i)` for node `i` is the node error `e(i)` (computed by `nodeerr`) weighted by the node probability `p(i)` (computed by `nodeprob`).

`r = risk(t,nodes)` takes a vector `nodes` of node numbers and returns the risk values for the specified nodes.

Example

Create a classification tree for Fisher's iris data:

```
load fisheriris;

t = classregtree(meas,species,'names',{'SL' 'SW' 'PL' 'PW'})
t =
Decision tree for classification
1  if PL<2.45 then node 2 else node 3
2  class = setosa
3  if PW<1.75 then node 4 else node 5
4  if PL<4.95 then node 6 else node 7
5  class = virginica
6  if PW<1.65 then node 8 else node 9
7  class = virginica
8  class = versicolor
9  class = virginica

view(t)
```



```
e = nodeerr(t);
p = nodeprob(t);
r = risk(t);

r
r =
    0.6667
         0
    0.3333
    0.0333
    0.0067
    0.0067
    0.0133
         0
         0

e.*p
ans =
    0.6667
         0
    0.3333
    0.0333
    0.0067
    0.0067
    0.0133
         0
         0
```

Reference

[1] Breiman, L., et al., *Classification and Regression Trees*, Chapman & Hall, Boca Raton, 1993.

See Also

`classregtree`, `nodeerr`, `nodeprob`

Purpose

Interactive robust regression

Syntax

```
robustdemo
robustdemo(x,y)
```

Description

`robustdemo` demonstrates the difference between ordinary least squares regression and robust regression. It displays a scatter plot of x and y values, where y is roughly a linear function of x, but one point is an outlier. The bottom of the figure shows the fitted equations using both least squares and robust fitting, plus estimates of the root mean squared errors.

Use the left mouse button to select a point and move it, and the fitted lines will update. Use the right mouse button to click on a point and view two of its properties:

- leverage — a measure of how much influence the point has on the least squares fit
- weight — the weight the point was given in the robust fit

`robustdemo(x,y)` uses the `x` and `y` data vectors you supply, in place of the sample data supplied with the function.

Example

See "Robust Regression" on page 7-26.

See Also

`robustfit`, `leverage`

Purpose

Robust linear regression

Syntax

```
b = robustfit(X,y)
b = robustfit(X,y,wfun,tune)
b = robustfit(X,y,wfun,tune,const)
[b,stats] = robustfit(...)
```

Description

`b = robustfit(X,y)` returns a p-by-1 vector `b` of coefficient estimates for a robust multilinear regression of the responses in `y` on the predictors in `X`. `X` is an n-by-p matrix of p predictors at each of n observations. `y` is an n-by-1 vector of observed responses. By default, the algorithm uses iteratively reweighted least squares with a bisquare weighting function.

Note By default, `robustfit` adds a first column of ones to `X`, corresponding to a constant term in the model. Do not enter a column of ones directly into `X`. You can change the default behavior of `robustfit` using the input *`const`*, below.

`robustfit` treats `NaN`s in `X` or `y` as missing values, and removes them.

`b = robustfit(X,y,`*`wfun`*`,tune)` specifies a weighting function *`wfun`*. `tune` is a tuning constant that is divided into the residual vector before computing weights.

The weighting function *`wfun`* can be any one of the following strings:

Weight Function	Equation	Default Tuning Constant
`'andrews'`	`w = (abs(r)<pi) .* sin(r) ./ r`	1.339
`'bisquare'` (default)	`w = (abs(r)<1) .* (1 - r.^2).^2`	4.685
`'cauchy'`	`w = 1 ./ (1 + r.^2)`	2.385

Weight Function	Equation	Default Tuning Constant
'fair'	`w = 1 ./ (1 + abs(r))`	1.400
'huber'	`w = 1 ./ max(1, abs(r))`	1.345
'logistic'	`w = tanh(r) ./ r`	1.205
'ols'	Ordinary least squares (no weighting function)	None
'talwar'	`w = 1 * (abs(r)<1)`	2.795
'welsch'	`w = exp(-(r.^2))`	2.985

If `tune` is unspecified, the default value in the table is used. Default tuning constants give coefficient estimates that are approximately 95% as statistically efficient as the ordinary least squares estimates, provided the response has a normal distribution with no outliers. Decreasing the tuning constant increases the downweight assigned to large residuals; increasing the tuning constant decreases the downweight assigned to large residuals.

The value `r` in the weight functions is equal to

```
resid/(tune*s*sqrt(1-h))
```

where `resid` is the vector of residuals from the previous iteration, `h` is the vector of leverage values from a least squares fit, and `s` is an estimate of the standard deviation of the error term given by

```
s = MAD/0.6745
```

Here `MAD` is the median absolute deviation of the residuals from their median. The constant 0.6745 makes the estimate unbiased for the normal distribution. If there are *p* columns in `X`, the smallest *p* absolute deviations are excluded when computing the median.

You can write your own M-file weight function. The function must take a vector of scaled residuals as input and produce a vector of weights as output. In this case, *wfun* is specified using a function handle @ (as in @myfun), and the input tune is required.

b = robustfit(X,y,*wfun*,tune,*const*) controls whether or not the model will include a constant term. *const* is 'on' to include the constant term (the default), or 'off' to omit it. When *const* is 'on', robustfit adds a first column of ones to X. When *const* is 'off', robustfit does not alter X.

[b,stats] = robustfit(...) returns the structure stats, whose fields contain diagnostic statistics from the regression. The fields of stats are:

- ols_s — Sigma estimate (RMSE) from ordinary least squares
- robust_s — Robust estimate of sigma
- mad_s — Estimate of sigma computed using the median absolute deviation of the residuals from their median; used for scaling residuals during iterative fitting
- s — Final estimate of sigma, the larger of robust_s and a weighted average of ols_s and robust_s
- se — Standard error of coefficient estimates
- t — Ratio of b to se
- p — p-values for t
- covb — Estimated covariance matrix for coefficient estimates
- coeffcorr — Estimated correlation of coefficient estimates
- w — Vector of weights for robust fit
- h — Vector of leverage values for least squares fit
- dfe — Degrees of freedom for error
- R — R factor in QR decomposition of X

The `robustfit` function estimates the variance-covariance matrix of the coefficient estimates using `inv(X'*X)*stats.s^2`. Standard errors and correlations are derived from this estimate.

Example

Generate data with the trend `y = 10-2*x`, then change one value to simulate an outlier:

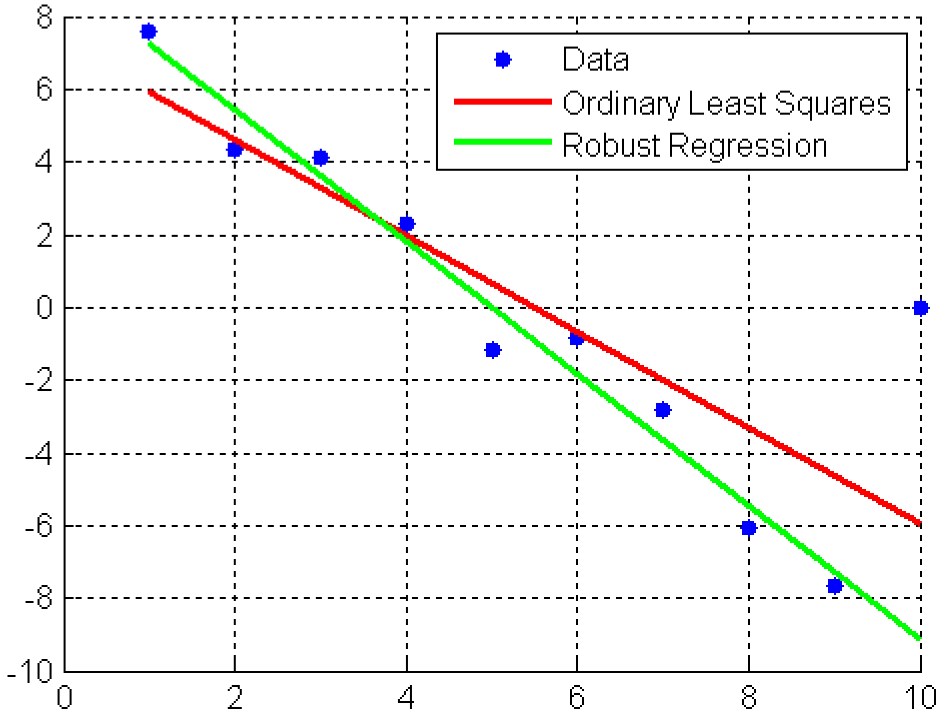
```
x = (1:10)';
y = 10 - 2*x + randn(10,1);
y(10) = 0;
```

Use both ordinary least squares and robust regression to estimate a straight line fit:

```
bls = regress(y,[ones(10,1) x])
bls =
  7.2481
 -1.3208

brob = robustfit(x,y)
brob =
  9.1063
 -1.8231
```

A scatter plot of the data together with the fits shows that the robust fit is less influenced by the outlier than the least squares fit:

```
scatter(x,y,'filled'); grid on; hold on
plot(x,bls(1)+bls(2)*x,'r','LineWidth',2);
plot(x,brob(1)+brob(2)*x,'g','LineWidth',2)
legend('Data','Ordinary Least Squares','Robust Regression')
```



References

[1] DuMouchel, W. H., F. L. O'Brien, "Integrating a Robust Option into a Multiple Regression Computing Environment," *Computer Science and Statistics*: *Proceedings of the 21st Symposium on the Interface*, Alexandria, VA, American Statistical Association, 1989.

[2] Holland, P. W., R. E. Welsch, "Robust Regression Using Iteratively Reweighted Least-Squares," *Communications in Statistics: Theory and Methods*, *A6*, 1977, pp. 813-827.

[3] Huber, P. J., *Robust Statistics*, Wiley, 1981.

[4] Street, J. O., R. J. Carroll, D. Ruppert, "A Note on Computing Robust Regression Estimates via Iteratively Reweighted Least Squares," *The American Statistician*, *42*, 1988, pp. 152-154.

See Also `regress`, `robustdemo`

Purpose

Rotation of factor analysis or principal components analysis loadings

Syntax

```
B = rotatefactors(A)
B = rotatefactors(A,'Method','orthomax','Coeff',gamma)
B = rotatefactors(A,'Method','procrustes','Target',target)
B = rotatefactors(A,'Method','pattern','Target',target)
B = rotatefactors(A,'Method','promax')
[B,T] = rotatefactors(A,...)
```

Description

`B = rotatefactors(A)` rotates the d-by-m loadings matrix `A` to maximize the varimax criterion, and returns the result in `B`. Rows of `A` and `B` correspond to variables and columns correspond to factors, for example, the (i, j)th element of `A` is the coefficient for the ith variable on the jth factor. The matrix `A` usually contains principal component coefficients created with `princomp` or `pcacov`, or factor loadings estimated with `factoran`.

`B = rotatefactors(A,'Method','orthomax','Coeff',gamma)` rotates `A` to maximize the orthomax criterion with the coefficient `gamma`, i.e., `B` is the orthogonal rotation of `A` that maximizes

```
sum(D*sum(B.^4,1) - GAMMA*sum(B.^2,1).^2)
```

The default value of 1 for `gamma` corresponds to varimax rotation. Other possibilities include `gamma` = 0, $m/2$, and $d(m - 1)/(d + m - 2)$, corresponding to quartimax, equamax, and parsimax. You can also supply the strings `'varimax'`, `'quartimax'`, `'equamax'`, or `'parsimax'` for the `'method'` parameter and omit the `'Coeff'` parameter.

If `'Method'` is `'orthomax'`, `'varimax'`, `'quartimax'`, `'equamax'`, or `'parsimax'`, then additional parameters are

- `'Normalize'` — Flag indicating whether the loadings matrix should be row-normalized for rotation. If `'on'` (the default), rows of `A` are normalized prior to rotation to have unit Euclidean norm, and unnormalized after rotation. If `'off'`, the raw loadings are rotated and returned.

- `'Reltol'` — Relative convergence tolerance in the iterative algorithm used to find `T`. The default is `sqrt(eps)`.
- `'Maxit'` — Iteration limit in the iterative algorithm used to find `T`. The default is `250`.

`B = rotatefactors(A,'Method','procrustes','Target',target)` performs an oblique procrustes rotation of `A` to the *d*-by-*m* target loadings matrix `target`.

`B = rotatefactors(A,'Method','pattern','Target',target)` performs an oblique rotation of the loadings matrix `A` to the *d*-by-*m* target pattern matrix `target`, and returns the result in `B`. `target` defines the "restricted" elements of `B`, i.e., elements of `B` corresponding to zero elements of `target` are constrained to have small magnitude, while elements of `B` corresponding to nonzero elements of `target` are allowed to take on any magnitude.

If `'Method'` is `'procrustes'` or `'pattern'`, an additional parameter is `'Type'`, the type of rotation. If `'Type'` is `'orthogonal'`, the rotation is orthogonal, and the factors remain uncorrelated. If `'Type'` is `'oblique'` (the default), the rotation is oblique, and the rotated factors might be correlated.

When `'Method'` is `'pattern'`, there are restrictions on `target`. If `A` has *m* columns, then for orthogonal rotation, the *j*th column of `target` must contain at least *m* - *j* zeros. For oblique rotation, each column of `target` must contain at least *m* - 1 zeros.

`B = rotatefactors(A,'Method','promax')` rotates A to maximize the promax criterion, equivalent to an oblique Procrustes rotation with a target created by an orthomax rotation. Use the four orthomax parameters to control the orthomax rotation used internally by promax.

An additional parameter for 'promax' is `'Power'`, the exponent for creating promax target matrix. `'Power'` must be `1` or greater. The default is `4`.

`[B,T] = rotatefactors(A,...)` returns the rotation matrix `T` used to create `B`, that is, `B = A*T`. `inv(T'*T)` is the correlation matrix of the rotated factors. For orthogonal rotation, this is the identity matrix, while for oblique rotation, it has unit diagonal elements but nonzero off-diagonal elements.

Examples

```
 X = randn(100,10);
 L = princomp(X);

% Default (normalized varimax) rotation of
% the first three components from a PCA.
[L1,T] = rotatefactors(L(:,1:3));

% Equamax rotation of the first three
% components from a PCA.
[L2, T] = rotatefactors(L(:,1:3),'method','equamax');

% Promax rotation of the first three factors from an FA.
L = factoran(X,3,'Rotate','none');
L3, T] = rotatefactors(L,'method','promax','power',2);

% Pattern rotation of the first three factors from an FA.
Tgt = [1 1 1 1 1 0 1 0 1; ...
       0 0 0 1 1 1 0 0 0; ...
       1 0 0 1 0 1 1 1 1]';
[L4,T] = rotatefactors(L,'method','pattern','target',Tgt);
inv(T'*T) % the correlation matrix of the rotated factors
```

References

[1] Harman, H. H., *Modern Factor Analysis*, 3rd edition, University of Chicago Press, 1976.

[2] Lawley, D. N. and A. E. Maxwell, A. E., *Factor Analysis as a Statistical Method*, 2nd edition, American Elsevier Publishing, 1971.

See Also

`biplot`, `factoran`, `princomp`, `pcacov`, `procrustes`

Purpose

D-optimal design of experiments row exchange algorithm

Syntax

```
settings = rowexch(nfactors,nruns)
[settings,X] = rowexch(nfactors,nruns)
[settings,X] = rowexch(nfactors,nruns,model)
[settings,X] = rowexch(...,param1,val1,param2,val2,...)
```

Description

`settings = rowexch(nfactors,nruns)` generates the factor-settings matrix, `settings`, for a D-optimal design using a linear additive model with a constant term. `settings` has `nruns` rows and `nfactors` columns.

`[settings,X] = rowexch(nfactors,nruns)` also generates the associated matrix `X` of term settings, often called the design matrix.

`[settings,X] = rowexch(nfactors,nruns,`*`model`*`)` produces a design for fitting a specified regression model. The input, *`model`*, can be one of these strings:

`'linear'`	Includes constant and linear terms (the default).
`'interaction'`	Includes constant, linear, and cross product terms.
`'quadratic'`	Includes interactions plus squared terms.
`'purequadratic'`	Includes constant, linear and squared terms.

`[settings,X] = rowexch(...,`*`param1`*`,`*`val1`*`,`*`param2`*`,`*`val2`*`,...)` provides more control over the design generation through a set of parameter/value pairs. Valid parameters are:

`'bounds'`	Lower and upper bounds for each factor, specified as a `2`-by-`nfactors` matrix. Alternatively, this value can be a cell array containing `nfactors` elements, each element specifying the vector of allowable values for the corresponding factor.
`'categorical'`	Indices of categorical predictors.

`'display'`	Either `'on'` or `'off'` to control display of iteration counter. The default is `'on'`.
`'excludefun'`	Function to exclude undesirable runs.
`'init'`	Initial design as an `nruns`-by-`nfactors` matrix. The default is a randomly selected set of points.
`'levels'`	Vector of number of levels for each factor.
`'tries'`	Number of times to try to generate a design from a new starting point, using random points for each try except possibly the first. The default is 1.
`'maxiter'`	Maximum number of iterations. The default is 10.

If the `'excludefcn'` function is `F`, it must support the syntax `B=F(S)`, where `S` is a matrix of `K`-by-`nfactors` columns containing settings, and `B` is a vector of `K` boolean values. `B(j)` is true if the jth row of `S` should be excluded.

Examples

This example illustrates that the D-optimal design for three factors in eight runs, using an interactions model, is a two-level full-factorial design.

```
s = rowexch(3,8,'interaction')
s =
     1    -1    -1
     1     1     1
    -1    -1     1
     1    -1     1
     1     1    -1
    -1     1     1
    -1     1    -1
    -1    -1    -1
```

Example of the design for three categorical factors taking three levels each—multiple tries may be required to find the best design.

```
s = sortrows(rowexch(3,9,'linear','cat',1:3,'levels',3,'tries',10))
```

```
s =
     1     1     2
     1     2     3
     1     3     1
     2     1     3
     2     2     1
     2     3     2
     3     1     1
     3     2     2
     3     3     3
```

This example may display warnings that the starting design is rank deficient.

Algorithm

The `rowexch` function searches for a D-optimal design using a row-exchange algorithm. It first generates a candidate set of points that are eligible to be included in the design, and then iteratively exchanges design points for candidate points in an attempt to reduce the variance of the coefficients that would be estimated using this design. If you need to use a candidate set that differs from the default one, call the `candgen` and `candexch` functions in place of `rowexch`.

See Also

`bbdesign`, `candexch`, `candgen`, `ccdesign`, `cordexch`, `x2fx`

Purpose

Demo of design of experiments and surface fitting

Syntax

```
rsmdemo
```

Description

`rsmdemo` creates a GUI that simulates a chemical reaction. To start, you have a budget of 13 test reactions. Try to find out how changes in each reactant affect the reaction rate. Determine the reactant settings that maximize the reaction rate. Estimate the run-to-run variability of the reaction. Now run a designed experiment using the model pop-up. Compare your previous results with the output from response surface modeling or nonlinear modeling of the reaction. The GUI has the following elements:

- A **Run** button to perform one reactor run at the current settings
- An **Export** button to export the *x* and *y* data to the base workspace
- Three sliders with associated data entry boxes to control the partial pressures of the chemical reactants: Hydrogen, *n*-pentane, and isopentane
- A text box to report the reaction rate
- A text box to keep track of the number of test reactions you have left

Example

See “Design of Experiments Demo” on page 11-11.

See Also

`rstool`, `nlintool`, `cordexch`

Purpose

Interactive multidimensional response surface modeling

Syntax

```
rstool(X,Y,model)
rstool(x,y,model,alpha)
rstool(x,y,model,alpha,xname,yname)
```

Description

`rstool(X,Y,`*`model`*`)` displays a graphical user interface for fitting and visualizing a polynomial response surface for `Y` as a function of the predictors in `X`. Distinct predictor variables should appear in different columns of `X`. `Y` can be a vector, corresponding to a single response, or a matrix, with columns corresponding to multiple responses. `Y` must have as many elements (or rows, if it is a matrix) as `X` has rows.

`rstool` displays a family of subplots, one for each combination of columns in `X` and `Y`. The subplots show 95% global confidence intervals for predictions as two red curves.

The optional input *`model`* controls the regression model. By default, `rstool` uses a linear additive model with a constant term. *`model`* can be any one of the following strings:

- `'linear'` — Constant and linear terms (the default)
- `'interaction'` — Constant, linear, and interaction terms
- `'quadratic'` — Constant, linear, interaction, and squared terms
- `'purequadratic'` — Constant, linear, and squared terms

To specify a polynomial model of arbitrary order, or a model without a constant term, use a matrix for `model` as described in `x2fx`.

To use the interface:

- Drag the dashed blue reference line to examine predicted values.
- Specify a predictor by typing its value into the editable text field.
- Use the pop-up menu to change the model.

- Use the **Export** push button to export fitted coefficients and regression statistics to the base workspace. Exported coefficients appear in the order defined by the `x2fx` function.

`rstool(x,y,`*`model`*`,alpha)` plots `100(1-alpha)`% global confidence intervals for predictions.

`rstool(x,y,`*`model`*`,alpha,xname,yname)` labels the axes using the names in the strings `xname` and `yname`. To label each subplot differently, `xname` and `yname` can be cell arrays of strings.

Drag the dashed blue reference line and watch the predicted values update simultaneously. Alternatively, you can get a specific prediction by typing the value of *x* into an editable text field. Use the pop-up menu to interactively change the model. Click the **Export** button to move specified variables to the base workspace.

Example

See "Quadratic Response Surface Models" on page 7-12.

See Also

`x2fx`, `nlintool`

runstest

Purpose

Runs test for randomness

Syntax

```
h = runstest(x)
h = runstest(x,v)
h = runstest(x,'ud')
h = runstest(...,param1,val1,param2,val2,...)
[h,p] = runstest(...)
[h,p,stats] = runstest(...)
```

Description

`h = runstest(x)` performs a runs test on the sequence of observations in the vector `x`. This is a test of the null hypothesis that the values in `x` come in random order, against the alternative that they do not. The test is based on the number of runs of consecutive values above or below the mean of `x`. Too few runs indicate a tendency for high and low values to cluster. Too many runs indicate a tendency for high and low values to alternate. The test returns the logical value `h = 1` if it rejects the null hypothesis at the 5% significance level, and `h = 0` if it cannot. The test treats `NaN` values in `x` as missing values, and ignores them.

`runstest` uses a test statistic which is approximately normally distributed when the null hypothesis is true. It is the difference between the number of runs and its mean, divided by its standard deviation.

`h = runstest(x,v)` performs the test using runs above or below the value `v`. Values exactly equal to `v` are discarded.

`h = runstest(x,'ud')` performs a test for the number of runs up or down. This also tests the hypothesis that the values in `x` come in random order. Too few runs indicate a trend. Too many runs indicate an oscillation. Values exactly equal to the preceding value are discarded.

`h = runstest(...,`*`param1`*`,`*`val1`*`,`*`param2`*`,`*`val2`*`,...)` specifies additional parameters and their values. Valid parameter/value pairs are the following:

- `'alpha'` — A scalar giving the significance level of the test
- `'method'` — Either `'exact'` to compute the *p*-value using an exact algorithm, or `'approximate'` to use a normal approximation. The

default is `'exact'` for runs above/below, and for runs up/down when the length of `x` is 50 or less. The `'exact'` method is not available for runs up/down when the length of `x` is 51 or greater.

- `'tail'` — Performs the test against one of the following alternative hypotheses:
  - `'both'` — two-tailed test (sequence is not random)
  - `'right'` — right-tailed test (like values separate for runs above/below, direction alternates for runs up/down)
  - `'left'` — left-tailed test (like values cluster for runs above/below, values trend for runs up/down)

`[h,p] = runstest(...)` returns the *p*-value of the test. The output `p` is computed from either the test statistic or the exact distribution of the number of runs, depending on the value of the `'method'` parameter.

`[h,p,stats] = runstest(...)` returns a structure `stats` with the following fields:

- `nruns` — The number of runs
- `n1` — The number of values above `v`
- `n0` — The number of values below `v`
- `z` — The test statistic

Example

```
x = randn(40,1);
[h,p] = runstest(x,median(x))
h =
     0
p =
    0.6286
```

See Also

`signrank`, `signtest`

sampsizepwr

Purpose

Sample size and power for hypothesis test

Syntax

```
n = sampsizepwr(testtype,p0,p1)
n = sampsizepwr(testtype,p0,p1,power)
power = sampsizepwr(testtype,p0,p1,[],n)
p1 = sampsizepwr(testtype,p0,[],power,n)
[...] = sampsizepwr(...,n,param1,val1,param2,val2,...)
```

Description

`n = sampsizepwr(testtype,p0,p1)` returns the sample size `n` required for a two-sided test of the specified type to have a power (probability of rejecting the null hypothesis when the alternative hypothesis is true) of 0.90 when the significance level (probability of rejecting the null hypothesis when the null hypothesis is true) is 0.05. `p0` specifies parameter values under the null hypothesis. `p1` specifies the single parameter value being tested under the alternative hypothesis.

The following values are available for *`testtype`*:

- `'z'` — *z*-test for normally distributed data with known standard deviation. `p0` is a two-element vector `[mu0 sigma0]` of the mean and standard deviation, respectively, under the null hypothesis. `p1` is the value of the mean under the alternative hypothesis.
- `'t'` — *t*-test for normally distributed data with unknown standard deviation. `p0` is a two-element vector `[mu0 sigma0]` of the mean and standard deviation, respectively, under the null hypothesis. `p1` is the value of the mean under the alternative hypothesis.
- `'var'` — Chi-square test of variance for normally distributed data. `p0` is the variance under the null hypothesis. `p1` is the variance under the alternative hypothesis.
- `'p'` — Test of the *p* parameter (success probability) for a binomial distribution. `p0` is the value of *p* under the null hypothesis. `p1` is the value of *p* under the alternative hypothesis.

  The `'p'` test is a discrete test for which increasing the sample size does not always increase the power. For `n` values larger than 200, there may be values smaller than the returned `n` value that also produce the desired size and power.

`n = sampsizepwr(`*`testtype`*`,p0,p1,power)` returns the sample size `n` such that the power is `power` for the parameter value `p1`.

`power = sampsizepwr(`*`testtype`*`,p0,p1,[],n)` returns the power achieved for a sample size of `n` when the true parameter value is `p1`.

`p1 = sampsizepwr(`*`testtype`*`,p0,[],power,n)` returns the parameter value detectable with the specified sample size `n` and power `power`.

When computing `p1` for the `'p'` test, if no alternative can be rejected for a given null hypothesis and significance level, the function displays a warning message and returns `NaN`.

`[...]  = sampsizepwr(...,n,`*`param1`*`,`*`val1`*`,`*`param2`*`,`*`val2`*`,...)` specifies one or more of the following name/value pairs:

- `'alpha'` — Significance level of the test (default 0.05)
- `'tail'` — The type of test is one of the following:
  - `'both'` — Two-sided test for an alternative not equal to `p0`
  - `'right'` — One-sided test for an alternative larger than `p0`
  - `'left'` — One-sided test for an alternative smaller than `p0`

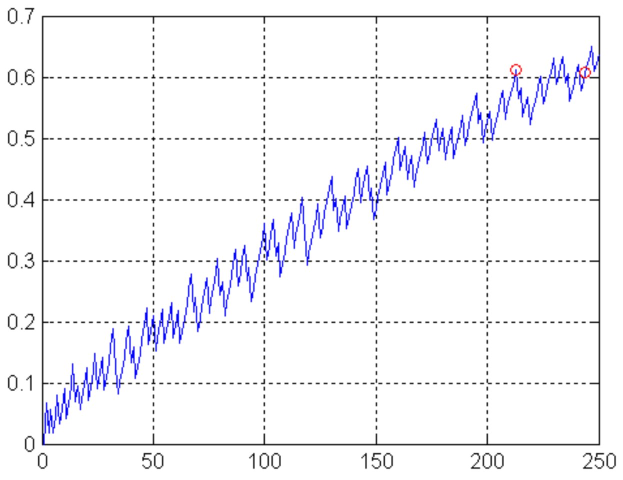
Example

Compute the mean closest to 100 that can be determined to be significantly different from 100 using a *t*-test with a sample size of 60 and a power of 0.8.

```
mu1 = sampsizepwr('t',[100 10],[],.8,60)
mu1 =
  103.6770
```

Compute the sample size n required to distinguish $p = 0.23$ from $p = 0.2$ with a binomial test. The result is approximate, so make a plot to see if any smaller n values also have the required power of 0.5.

```
napprox = sampsizepwr('p',.2,.26,.6)
Warning: Values N>200 are approximate.  Plotting the power as a function
of N may reveal lower N values that have the required power.
napprox =
   244

nn = 1:250;
pwr = sampsizepwr('p',.2,.26,[],nn);
nexact = min(nn(pwr>=.6))
nexact =
   213

plot(nn,pwr,'b-',[napprox nexact],pwr([napprox nexact]),'ro');
grid on
```



See Also

vartest, ttest, ztest, binocdf

Purpose

2-D scatter plot with marginal histograms

Syntax

```
scatterhist(x,y)
scatterhist(x,y,nbins)
h = scatterhist(...)
```

Description

`scatterhist(x,y)` creates a 2-D scatterplot of the data in the vectors `x` and `y`, and puts a univariate histogram on the horizontal and vertical axes of the plot. `x` and `y` must be the same length.

The function is useful for viewing properties of random samples produced by functions such as `copularnd`, `mvnrnd`, `lhsdesign`.

`scatterhist(x,y,nbins)` also accepts a two-element vector `nbins` specifying the number of bins for the `x` and `y` histograms. The default is to compute the number of bins using a Scott rule based on the sample standard deviation. Any `NaN` values in either `x` or `y` are treated as missing, and are removed from both `x` and `y`. Therefore the plots reflect points for which neither `x` nor `y` has a missing value.

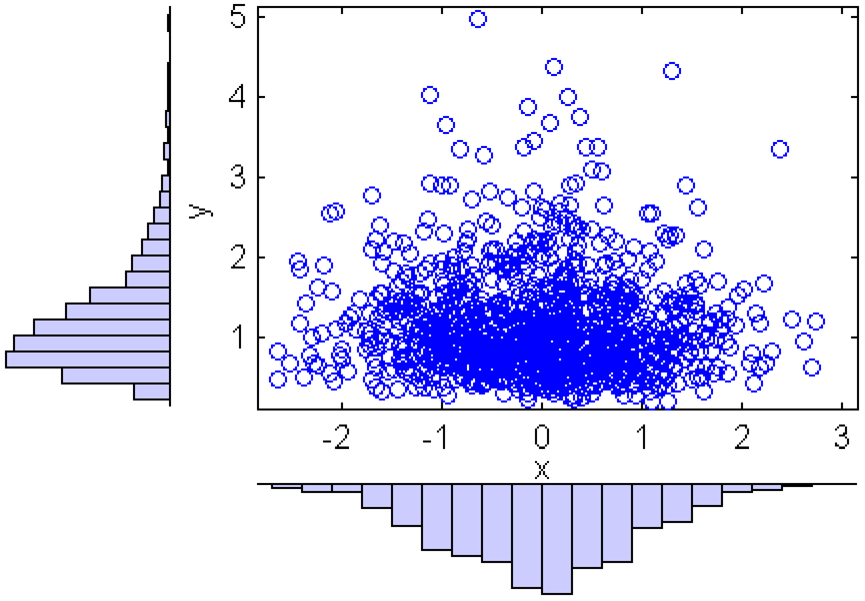
`h = scatterhist(...)` returns a vector of three axes handles for the scatterplot, the histogram along the horizontal axis, and the histogram along the vertical axis, respectively.

Examples

Example 1

Independent normal and lognormal random samples:

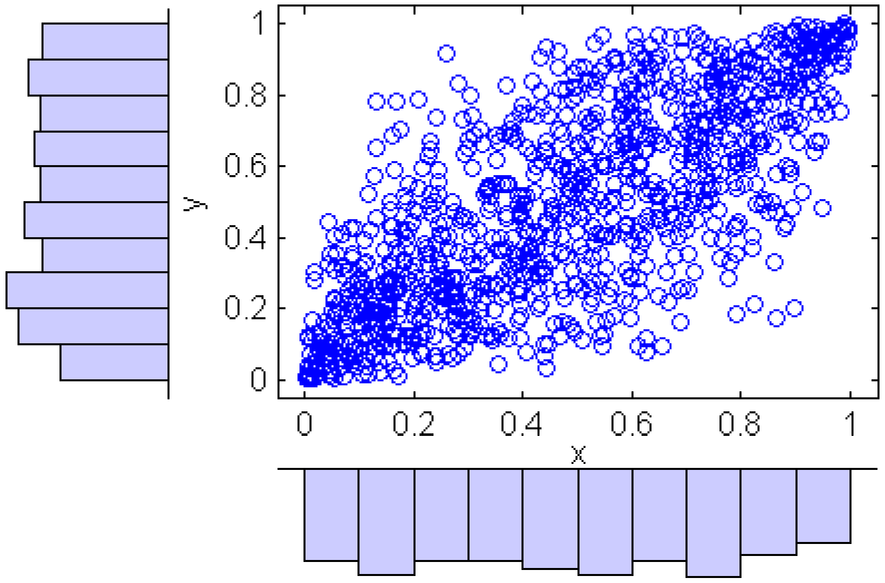
```
x = randn(1000,1);
y = exp(.5*randn(1000,1));
scatterhist(x,y)
```



Example 2

Marginal uniform samples that are not independent:

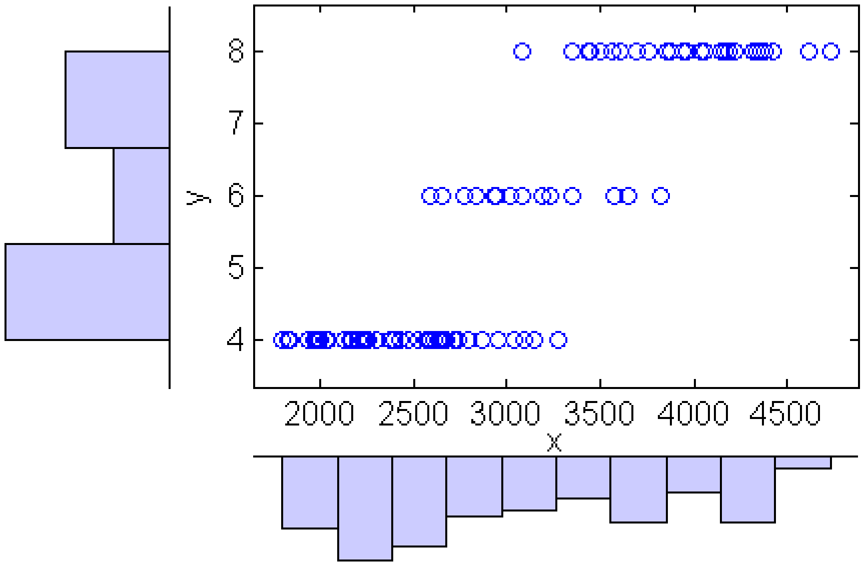
```
u = copularnd('Gaussian',.8,1000);
scatterhist(u(:,1),u(:,2))
```



Example 3

Mixed discrete and continuous data:

```
cars = load('carsmall');
scatterhist(cars.Weight,cars.Cylinders,[10 3])
```



See Also `scatter`, `hist`

Purpose

Segment of piecewise distribution containing input values

Syntax

`S = segment(obj,X,P)`

Description

`S = segment(obj,X,P)` returns an array `S` of integers indicating which segment of the piecewise distribution object `obj` contains each value of `X` or, alternatively, `P`. One of `X` and `P` must be empty (`[]`). If `X` is non-empty, `S` is determined by comparing `X` with the quantile boundary values defined for `obj`. If `P` is non-empty, `S` is determined by comparing `P` with the probability boundary values.

Example

Fit Pareto tails to a *t* distribution at cumulative probabilities 0.1 and 0.9:

```
t = trnd(3,100,1);
obj = paretotails(t,0.1,0.9);

pvals = 0:0.2:1;
s = segment(obj,[],pvals)
s =
     1     2     2     2     2     3
```

See Also

`paretotails`, `boundary`, `nsegments`

Purpose

Display and define dataset array properties

Syntax

```
set(A)
set(A,PropertyName)
A = set(A,PropertyName,PropertyValue,...)
```

Description

`set(A)` displays all properties of the dataset array `A` and their possible values.

`set(A,`*`PropertyName`*`)` displays possible values for the property specified by the string *`PropertyName`*.

`A = set(A,`*`PropertyName`*`,`*`PropertyValue`*`,...)` sets property name/value pairs.

Example

Create a dataset array from Fisher's iris data and add a description:

```
load fisheriris
NumObs = size(meas,1);
ObsNames = strcat({'Obs'},num2str((1:NumObs)','%d'));
iris = dataset({nominal(species),'species'},...
               {meas,'SL','SW','PL','PW'},...
               'obsnames',ObsNames);
iris = set(iris,'Description','Fisher''s Iris Data');
get(iris)
   Description: 'Fisher's Iris Data'
   Units: {}
   DimNames: {'Observations' 'Variables'}
   UserData: []
   ObsNames: {150x1 cell}
   VarNames: {'species' 'SL' 'SW' 'PL' 'PW'}
```

See Also

`get`, `summary (dataset)`

Purpose

Define labels of levels in categorical array

Syntax

```
A = setlabels(A,labels)
A = setlabels(A,labels,levels)
```

Description

`A = setlabels(A,labels)` labels the levels in the categorical array `A` using the cell array of strings or two-dimensional character matrix `labels`. Labels are assigned in the order given in `labels`.

`A = setlabels(A,labels,levels)` labels only the levels specified in the cell array of strings or two-dimensional character matrix `levels`.

Examples

Example 1

Relabel the species in Fisher's iris data using new categories:

```
load fisheriris
species = nominal(species);
species = mergelevels(...
          species,{'setosa','virginica'},'parent');
species = setlabels(species,'hybrid','versicolor');
getlabels(species)
ans =
    'hybrid'    'parent'
```

Example 2

1 Load patient data from the CSV file `hospital.dat` and store the information in a dataset array with observation names given by the first column in the data (patient identification):

```
patients = dataset('file','hospital.dat',...
                   'delimiter',',',...
                   'ReadObsNames',true);
```

2 Make the {0,1}-valued variable `smoke` nominal, and change the labels to `'No'` and `'Yes'`:

```
patients.smoke = nominal(patients.smoke,{'No','Yes'});
```

3 Add new levels to smoke as placeholders for more detailed histories of smokers:

```
patients.smoke = addlevels(patients.smoke,...
                 {'0-5 Years','5-10 Years','LongTerm'});
```

4 Assuming the nonsmokers have never smoked, relabel the 'No' level:

```
patients.smoke = setlabels(patients.smoke,'Never','No');
```

5 Drop the undifferentiated 'Yes' level from smoke:

```
patients.smoke = droplevels(patients.smoke,'Yes');

Warning: OLDLEVELS contains categorical levels that
were present in A, caused some array elements to have
undefined levels.
```

Note that smokers now have an undefined level.

6 Set each smoker to one of the new levels, by observation name:

```
patients.smoke('YPL-320') = '5-10 Years';
```

See Also

getlabels

Purpose

One-sample or paired-sample Wilcoxon signed rank test

Syntax

```
p = signrank(x)
p = signrank(x,m)
p = signrank(x,y)
[p,h] = signrank(...)
[p,h] = signrank(...,'alpha',alpha)
[p,h] = signrank(...,'method',method)
[p,h,stats] = signrank(...)
```

Description

`p = signrank(x)` performs a two-sided signed rank test of the null hypothesis that data in the vector `x` comes from a continuous, symmetric distribution with zero median, against the alternative that the distribution does not have zero median. The *p*-value of the test is returned in `p`.

`p = signrank(x,m)` performs a two-sided signed rank test of the null hypothesis that data in the vectors `x` and `y` are independent samples from a continuous, symmetric distribution with median `m`, against the alternative that the distribution does not have median `m`. `m` must be a scalar.

`p = signrank(x,y)` performs a paired, two-sided signed rank test of the null hypothesis that data in the vector `x-y` come from a continuous, symmetric distribution with zero median, against the alternative that the distribution does not have zero median. `x` and `y` must have equal lengths. Note that a hypothesis of zero median for `x-y` is not equivalent to a hypothesis of equal median for `x` and `y`.

`[p,h] = signrank(...)` returns the result of the test in `h`. `h = 1` indicates a rejection of the null hypothesis at the 5% significance level. `h = 0` indicates a failure to reject the null hypothesis at the 5% significance level.

`[p,h] = signrank(...,'alpha',alpha)` performs the test at the (100*`alpha`)% significance level. The default, when unspecified, is `alpha = 0.05`.

`[p,h] = signrank(...,'method',method)` computes the p-value using either an exact algorithm, when *`method`* is `'exact'`, or a normal approximation, when *`method`* is `'approximate'`. The default, when unspecified, is the exact method for small samples and the approximate method for large samples.

`[p,h,stats] = signrank(...)` returns the structure `stats` with the following fields:

- `signedrank` — Value of the signed rank test statistic
- `zval` — Value of the z-statistic (computed only for large samples)

Example

Test the hypothesis of zero median for the difference between two paired samples.

```
before = lognrnd(2,.25,10,1);
after = before+trnd(2,10,1);
[p,h] = signrank(before,after)
p =
   0.5566
h =
   0
```

The sampling distribution of the difference between `before` and `after` is symmetric with zero median. At the default 5% significance level, the test fails to reject to the null hypothesis of zero median in the difference.

References

[1] Gibbons, J. D., *Nonparametric Statistical Inference*, 2nd edition, M. Dekker, 1985.

[2] Hollander, M. and D. A. Wolfe, *Nonparametric Statistical Methods*, Wiley, 1973.

See Also

`ranksum`, `ttest`, `ztest`

Purpose

One-sample or paired-sample sign test

Syntax

```
p = signtest(x)
p = signtest(x,m)
p = signtest(x,y)
[p,h] = signtest(...)
[p,h] = signtest(...,'alpha',alpha)
[p,h] = signtest(...,'method',method)
[p,h,stats] = signtest(...)
```

Description

`p = signtest(x)` performs a two-sided sign test of the null hypothesis that data in the vector `x` come from a continuous distribution with zero median, against the alternative that the distribution does not have zero median. The *p*-value of the test is returned in `p`

`p = signtest(x,m)` performs a two-sided sign test of the null hypothesis that data in the vector `x` come from a continuous distribution with median `m`, against the alternative that the distribution does not have median `m`. `m` must be a scalar.

`p = signtest(x,y)` performs a paired, two-sided sign test of the null hypothesis that data in the vector `x-y` come from a continuous distribution with zero median, against the alternative that the distribution does not have zero median. `x` and `y` must be the same length. Note that a hypothesis of zero median for `x-y` is not equivalent to a hypothesis of equal median for `x` and `y`.

`[p,h] = signtest(...)` returns the result of the test in `h`. `h = 1` indicates a rejection of the null hypothesis at the 5% significance level. `h = 0` indicates a failure to reject the null hypothesis at the 5% significance level.

`[p,h] = signtest(...,'alpha',alpha)` performs the test at the (100*`alpha`)% significance level. The default, when unspecified, is `alpha = 0.05`.

`[p,h] = signtest(...,'method',`*`method`*`)` computes the *p*-value using either an exact algorithm, when *`method`* is `'exact'`, or a normal approximation, when `method` is `'approximate'`. The default, when unspecified, is the exact method for small samples and the approximate method for large samples.

`[p,h,stats] = signtest(...)` returns the structure `stats` with the following fields:

- `sign` — Value of the sign test statistic
- `zval` — Value of the *z*-statistic (computed only for large samples)

Example

Test the hypothesis of zero median for the difference between two paired samples.

```
before = lognrnd(2,.25,10,1);
after = before + (lognrnd(0,.5,10,1) - 1);
[p,h] = signtest(before,after)
p =
   0.3438
h =
   0
```

The sampling distribution of the difference between `before` and `after` is symmetric with zero median. At the default 5% significance level, the test fails to reject to the null hypothesis of zero median in the difference.

References

[1] Gibbons, J. D., *Nonparametric Statistical Inference*, 2nd edition, M. Dekker, 1985.

[2] Hollander, M. and D. A. Wolfe, *Nonparametric Statistical Methods*, Wiley, 1973.

See Also

`ranksum`, `signrank`, `ttest`, `ztest`

Purpose

Silhouette plot for clustered data

Syntax

```
silhouette(X,clust)
s = silhouette(X,clust)
[s,h] = silhouette(X,clust)
[...] = silhouette(X,clust,metric)
[...] = silhouette(X,clust,distfun,p1,p2,...)
```

Description

`silhouette(X,clust)` plots cluster silhouettes for the *n*-by-*p* data matrix `X`, with clusters defined by `clust`. Rows of `X` correspond to points, columns correspond to coordinates. `clust` can be a categorical variable, numeric vector, character matrix, or cell array of strings containing a cluster name for each point. (See "Grouped Data" on page 2-41.) `silhouette` treats `NaN`s or empty strings in `clust` as missing values, and ignores the corresponding rows of `X`. By default, `silhouette` uses the squared Euclidean distance between points in `X`.

`s = silhouette(X,clust)` returns the silhouette values in the *n*-by-1 vector `s`, but does not plot the cluster silhouettes.

`[s,h] = silhouette(X,clust)` plots the silhouettes, and returns the silhouette values in the *n*-by-1 vector `s`, and the figure handle in `h`.

`[...] = silhouette(X,clust,`*`metric`*`)` plots the silhouettes using the inter-point distance function specified in *`metric`*. Choices for *`metric`* are:

`'Euclidean'`	Euclidean distance
`'sqEuclidean'`	Squared Euclidean distance (default)
`'cityblock'`	Sum of absolute differences
`'cosine'`	One minus the cosine of the included angle between points (treated as vectors)
`'correlation'`	One minus the sample correlation between points (treated as sequences of values)
`'Hamming'`	Percentage of coordinates that differ

`'Jaccard'`	Percentage of nonzero coordinates that differ
Vector	A numeric distance matrix in upper triangular vector form, such as is created by `pdist`. `X` is not used in this case, and can safely be set to `[]`.

`[...] = silhouette(X,clust,distfun,p1,p2,...)` accepts a function handle `distfun` to a metric of the form

```
d = distfun(X0,X,p1,p2,...)
```

where `X0` is a `1`-by-`p` point, `X` is an `n`-by-`p` matrix of points, and `p1,p2,...` are optional additional arguments. The function `distfun` returns an `n`-by-`1` vector `d` of distances between `X0` and each point (row) in `X`. The arguments `p1, p2,...` are passed directly to the function `distfun`.

Remarks

The silhouette value for each point is a measure of how similar that point is to points in its own cluster compared to points in other clusters, and ranges from -1 to +1. It is defined as

```
S(i) = (min(b(i,:),2) - a(i)) ./ max(a(i),min(b(i,:),2))
```

where `a(i)` is the average distance from the `i`th point to the other points in its cluster, and `b(i,k)` is the average distance from the `i`th point to points in another cluster `k`.

Examples

```
X = [randn(10,2)+ones(10,2);
randn(10,2)-ones(10,2)];
cidx = kmeans(X,2,'distance','sqeuclid');
s = silhouette(X,cidx,'sqeuclid');
```

References

[1] Kaufman L., and P. J. Rousseeuw, *Finding Groups in Data: An Introduction to Cluster Analysis*, Wiley, 1990.

See Also

`dendrogram`, `kmeans`, `linkage`, `pdist`

Purpose

Markov chain slice sampler

Syntax

```
rnd = slicesample(initial,nsamples,'pdf',pdf)
rnd = slicesample(...,'width',w)
rnd = slicesample(...,'burnin',k)
rnd = slicesample(...,'thin',m)
[rnd,neval] = slicesample(...)
```

Description

`rnd = slicesample(initial,nsamples,'pdf',pdf)` generates `nsamples` random samples from a target distribution whose density function is defined by `pdf` using the slice sampling method. `initial` is a row vector or scalar containing the initial value of the random sample sequences. `initial` must be within the domain of the target distribution. `nsamples` is the number of samples to be generated. `pdf` is a function handle created using `@`. `pdf` accepts only one argument that must be the same type and size as `initial`. It defines a function that is proportional to the target density function. If the log density function is preferred, `'pdf'` can be replaced with `'logpdf'`. The log density function is not necessarily normalized.

`rnd = slicesample(...,'width',w)` performs slice sampling for the target distribution with a typical width `w`. `w` is a scalar or vector. If it is a scalar, all dimensions are assumed to have the same typical widths. If it is a vector, each element of the vector is the typical width of the marginal target distribution in that dimension. The default value of `w` is `10`.

`rnd = slicesample(...,'burnin',k)` generates random samples with values between the starting point and the `k`th point omitted in the generated sequence. Values beyond the `k`th point are kept. `k` is a nonnegative integer with default value of `0`.

`rnd = slicesample(...,'thin',m)` generates random samples with `m-1` out of `m` values omitted in the generated sequence. `m` is a positive integer with default value of `1`.

`[rnd,neval] = slicesample(...)` also returns `neval`, the averaged number of function evaluations that occurred in the slice sampling. `neval` is a scalar.

Example

Generate random samples from a distribution with a user-defined pdf.

First, define the function that is proportional to the pdf for a multi-modal distribution.

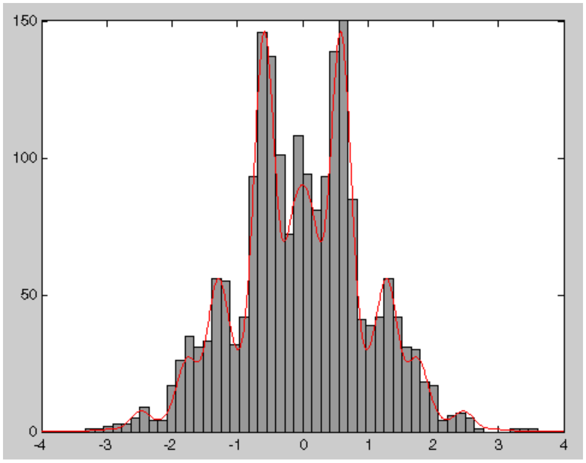
```
f = @(x) exp( -x.^2/2).*(1+(sin(3*x)).^2).* ...
    (1+(cos(5*x).^2));
```

Next, use the `slicesample` function to generate the random samples for the function defined above.

```
x = slicesample(1,2000,'pdf',f,'thin',5,'burnin',1000);
```

Now, plot a histogram of the random samples generated.

```
hist(x,50)
set(get(gca,'child'),'facecolor',[0.6 .6 .6]);
hold on
xd = get(gca,'XLim'); % Gets the xdata of the bins
binwidth = (xd(2)-xd(1)); % Finds the width of each bin
% Use linspace to normalize the histogram
y = 5.6398*binwidth*f(linspace(xd(1),xd(2),1000));
plot(linspace(xd(1),xd(2),1000),y,'r')
```



See Also `rand`, `mhsample`, `randsample`

skewness

Purpose

Sample skewness

Syntax

```
y = skewness(X)
y = skewness(X,flag)
```

Description

`y = skewness(X)` returns the sample skewness of `X`. For vectors, `skewness(x)` is the skewness of the elements of `x`. For matrices, `skewness(X)` is a row vector containing the sample skewness of each column. For N-dimensional arrays, `skewness` operates along the first nonsingleton dimension of `X`.

`y = skewness(X,flag)` specifies whether to correct for bias (`flag = 0`) or not (`flag = 1`, the default). When `X` represents a sample from a population, the skewness of `X` is biased; that is, it will tend to differ from the population skewness by a systematic amount that depends on the size of the sample. You can set `flag = 0` to correct for this systematic bias.

`skewness(X,flag,dim)` takes the skewness along dimension `dim` of `X`.

`skewness` treats `NaN`s as missing values and removes them.

Remarks

Skewness is a measure of the asymmetry of the data around the sample mean. If skewness is negative, the data are spread out more to the left of the mean than to the right. If skewness is positive, the data are spread out more to the right. The skewness of the normal distribution (or any perfectly symmetric distribution) is zero.

The skewness of a distribution is defined as

$$y = \frac{E(x-\mu)^3}{\sigma^3}$$

where μ is the mean of x, σ is the standard deviation of x, and $E(t)$ represents the expected value of the quantity t.

Example

```
X = randn([5 4])
X =
  1.1650  1.6961 -1.4462 -0.3600
  0.6268  0.0591 -0.7012 -0.1356
  0.0751  1.7971  1.2460 -1.3493
  0.3516  0.2641 -0.6390 -1.2704
 -0.6965  0.8717  0.5774  0.9846

y = skewness(X)
y =
  -0.2933  0.0482  0.2735  0.4641
```

See Also

kurtosis, mean, moment, std, var

Purpose

Sort ordinal array

Syntax

```
B = sort(A)
B = sort(A,dim)
B = sort(A,dim,mode)
[B,I] = sort(A,...)
```

Description

`B = sort(A)`, when `A` is an ordinal vector, sorts the elements of `A` in ascending order. For ordinal matrices, `sort(A)` sorts each column of `A` in ascending order. For *N*-D ordinal arrays, `sort(A)` sorts the along the first nonsingleton dimension of `A`. `B` is an ordinal array with the same levels as `A`.

`B = sort(A,dim)` sorts `A` along dimension `dim`.

`B = sort(A,dim,mode)` sorts `A` in the order specified by `mode`. `mode` is `'ascend'` for ascending order, or `'descend'` for descending order.

`[B,I] = sort(A,...)` also returns an index matrix `I`. If `A` is a vector, then `B = A(I)`. If `A` is an m-by-n matrix and `dim` is `1`, then `B(:,j) = A(I(:,j),j)` for `j = 1:n`.

Elements with undefined levels are sorted to the end.

Example

Sort the columns of an ordinal array in ascending order:

```
A = ordinal([6 2 5; 2 4 1; 3 2 4],...
            {'lo','med','hi'},[],[0 2 4 6])
A =
     hi       med      hi
     med      hi       lo
     med      med      hi

B = sort(A)
B =
     med      med      lo
     med      med      hi
     hi       hi       hi
```

See Also `sortrows (ordinal)`

sortrows (dataset)

Purpose Sort rows of dataset array

Syntax

```
B = sortrows(A)
B = sortrows(A,vars)
B = sortrows(A,'obsnames')
B = sortrows(A,vars,mode)
[B,idx] = sortrows(A)
```

Description `B = sortrows(A)` returns a copy of the dataset array `A`, with the observations sorted in ascending order by all of the variables in `A`. The observations in `B` are sorted first by the first variable, next by the second variable, and so on. The variables in `A` must be scalar valued (i.e., column vectors) and be from a class for which a `sort` method exists.

`B = sortrows(A,vars)` sorts the observations in `A` by the variables specified by `vars`. `vars` is a positive integer, a vector of positive integers, variable names, a cell array containing one or more variable names, or a logical vector.

`B = sortrows(A,'obsnames')` sorts the observations in `A` by observation name.

`B = sortrows(A,vars,`*`mode`*`)` sorts in the direction specified by *`mode`*. *`mode`* is `'ascend'` (the default) or `'descend'`. Use `[]` for `vars` to sort using all variables.

`[B,idx] = sortrows(A)` also returns an index vector `idx` such that `B = A(idx,:)`.

Example Sort the data in `hospital.mat` by age and then by last name:

```
load hospital
hospital(1:5,1:3)
ans =
            LastName        Sex       Age
 YPL-320    'SMITH'         Male      38
 GLI-532    'JOHNSON'       Male      43
 PNI-258    'WILLIAMS'      Female    38
 MIJ-579    'JONES'         Female    40
 XLK-030    'BROWN'        Female    49

hospital = sortrows(hospital,{'Age','LastName'});
hospital(1:5,1:3)
ans =
            LastName       Sex       Age
 REV-997    'ALEXANDER'    Male      25
 FZR-250    'HALL'         Male      25
 LIM-480    'HILL'         Female    25
 XUE-826    'JACKSON'      Male      25
 SCQ-914    'JAMES'        Male      25
```

See Also `sortrows (ordinal)`

sortrows (ordinal)

Purpose

Sort rows of ordinal array

Syntax

```
B = sortrows(A)
B = sortrows(A,col)
[B,I] = sortrows(A)
[B,I] = sortrows(A,col)
```

Description

`B = sortrows(A)` sorts the rows of the two-dimensional ordinal matrix `A` in ascending order, as a group. `B` is an ordinal array with the same levels as `A`.

`B = sortrows(A,col)` sorts `A` based on the columns specified in the vector `col`. If an element of `col` is positive, the corresponding column in `A` is sorted in ascending order; if an element of `col` is negative, the corresponding column in `A` is sorted in descending order.

`[B,I] = sortrows(A)` and `[B,I] = sortrows(A,col)` also returns an index matrix `I` such that `B = A(I,:)`.

Elements with undefined levels are sorted to the end.

Example

Sort the rows of an ordinal array in ascending order for the first column, and then in descending order for the second column:

```
A = ordinal([6 2 5; 2 4 1; 3 2 4],...
            {'lo','med','hi'},[],[0 2 4 6])
A =
     hi       med      hi
     med      hi       lo
     med      med      hi

B = sortrows(A,[1 -2])
B =
     med      hi       lo
     med      med      hi
     hi       med      hi
```

See Also

`sort`, `sortrows (dataset)`

Purpose

Reformat distance matrix

Syntax

```
Z = squareform(y)
y = squareform(Z)
Z = squareform(y,'tovector')
Y = squareform(Z,'tomatrix')
```

Description

`Z = squareform(y)`, where `y` is a vector as created by the `pdist` function, converts `y` into a square, symmetric format `Z`, in which `Z(i,j)` denotes the distance between the `i`th and `j`th objects in the original data.

`y = squareform(Z)`, where `Z` is a square, symmetric matrix with zeros along the diagonal, creates a vector `y` containing the `Z` elements below the diagonal. `y` has the same format as the output from the `pdist` function.

`Z = squareform(y,'tovector')` forces `squareform` to treat `y` as a vector.

`Y = squareform(Z,'tomatrix')` forces `squareform` to treat `Z` as a matrix.

The last two formats are useful if the input has a single element, so that it is ambiguous whether the input is a vector or square matrix.

Example

```
y = 1:6
y =
   1   2   3   4   5   6

X = [0 1 2 3; 1 0 4 5; 2 4 0 6; 3 5 6 0]
X =
   0   1   2   3
   1   0   4   5
   2   4   0   6
   3   5   6   0
```

Then `squareform(y) = X` and `squareform(X) = y`.

See Also `pdist`

Purpose

Parameter values from statistics options structure

Syntax

```
val = statget(options,param)
val = statget(options,param,default)
```

Description

`val = statget(options,param)` returns the value of the parameter specified by the string `param` in the statistics options structure `options`. If the parameter is not defined in `options`, `statget` returns `[]`. You need to type only enough leading characters to define the parameter name uniquely. Case is ignored for parameter names.

`val = statget(options,param,default)` returns `default` if the specified parameter is not defined in the optimization options structure `options`.

Examples

This statement returns the value of the `Display` statistics options parameter from the structure called `my_options`.

```
val = statget(my_options,'Display')
```

This statement returns the value of the `Display` statistics options parameter from the structure called `my_options` (as in the previous example) except that if the `Display` parameter is not defined, it returns the value `'final'`.

```
optnew = statget(my_options,'Display','final');
```

See Also

`statset`

Purpose

Create or edit statistics options structure

Syntax

```
options = statset(param1,val1,param2,val2,...)
options = statset(oldopts,param1,val1,param2,val2,...)
options = statset(oldopts,newopts)
statset
options = statset
options = statset(statfun)
```

Description

`options = statset(param1,val1,param2,val2,...)` creates a statistics options structure `options` in which the named parameters have the specified values. Any unspecified parameters are set to `[]`. When you pass `options` to a statistics function, a parameter set to `[]` indicates that the function uses its default value for that parameter. Case is ignored for parameter names, and unique partial matches are allowed.

NOTE: For parameters that are string-valued, the complete string is required for the value; if an invalid string is provided, the default is used.

`options = statset(oldopts,param1,val1,param2,val2,...)` creates a copy of `oldopts` with the named parameters altered with the specified values.

`options = statset(oldopts,newopts)` combines an existing options structure, `oldopts`, with a new options structure, `newopts`. Any parameters in `newopts` with nonempty values overwrite the corresponding old parameters in `oldopts`.

`statset` with no input arguments and no output arguments displays all parameter names and their possible values, with defaults shown in `{}` when the default is the same for all functions that use that option. Use `statset(statfun)` (see below) to see function-specific defaults for a specific function.

`options = statset` with no input arguments creates an options structure where all fields are set to `[]`.

options = statset(statfun) creates an options structure with all the parameter names and default values relevant to the optimization function named in statfun. statset sets parameters in options to [] for parameters that are not valid for statfun. For example, statset('factoran') or statset(@factoran) returns an options structure containing all the parameter names and default values relevant to the function factoran.

Parameters

The following table lists the valid parameter strings for statset, their meanings, and their allowed values. You can also view these parameters and allowed values by typing statset at the command line.

Parameter	Meaning	Allowed Value
'DerivStep'	Relative difference used in finite difference derivative calculations. May be a scalar or the same size as the parameter vector.	Positive scalar or vector
'Display'	Amount of information displayed by the algorithm.	• 'off' — displays no information • 'final' — displays the final output • 'notify' — displays output only if the algorithm fails to converge
'FunValCheck'	Check for invalid values, such as NaN or Inf, from the objective function.	• 'off' • 'on'
'GradObject'	Objective function can return a gradient vector as a second output.	• 'off' • 'on'

Parameter	Meaning	Allowed Value
`'MaxFunEvals'`	Maximum number of objective function evaluations allowed.	Positive integer
`'MaxIter'`	Maximum number of iterations allowed.	Positive integer
`'Robust'`	Invoke robust fitting option.	• `'off'` (default) • `'on'`
`'TolBnd'`	Parameter bound tolerance.	Positive scalar
`'TolFun'`	Termination tolerance for the objective function value.	Positive scalar
`'TolX'`	Termination tolerance for the parameters.	Positive scalar

Parameter	Meaning	Allowed Value
`'Tune'`	The tuning constant used to normalize the residuals before applying the weight function. The default value depends upon the weight function. This parameter is required if the weight function is specified as a function handle.	Positive scalar
`'WgtFun'`	Specify the weight function for robust fitting. This weight function is only valid when `'Robust'` is set `'on'`. This can also be a function handle that accepts a normalized residual as input and returns the robust weights as output.	• `'bisquare'` (default) • `'andrews'` • `'cauchy'` • `'fair'` • `'huber'` • `'logistic'` • `'talwar'` • `'welsch'`

Example

Suppose you want to change the default parameters for the function `evfit`, which fits data to an extreme value distribution. To see the defaults for `evfit`, enter

```
statset('evfit')
ans =
  Display: 'off'
  MaxFunEvals: []
  MaxIter: []
  TolBnd: []
  TolFun: []
  TolX: 1.0000e-006
```

```
      GradObj: []
      DerivStep: []
      FunValCheck: []
```

Note that the only parameters `evfit` uses are `Display` and `TolX`. To change the value of `TolX` to `1e-8`, enter

```
my_opts = statset('TolX',1e-8)
my_opts =
  Display: []
  MaxFunEvals: []
  MaxIter: []
  TolBnd: []
  TolFun: []
  TolX: 1.0000e-008
  GradObj: []
  DerivStep: []
  FunValCheck: []
```

When you pass `my_opts` into `evfit` with the command

```
evfit(data,[],[],[],my_opts)
```

`evfit` uses its default value `'notify'` for `Display` and overrides the default value of `TolX` with `1e-8`.

See Also

`evfit`, `factoran`, `gamfit`, `lognfit`, `nbinfit`, `normfit`, `statget`

Purpose

Standard deviation of sample

Syntax

```
y = std(X)
Y = std(X,1)
Y = std(X,flag,dim)
```

Description

`y = std(X)` computes the sample standard deviation of the data in X. For vectors, `std(x)` is the standard deviation of the elements in `x`. For matrices, std(X) is a row vector containing the standard deviation of each column of `X`. For N-dimensional arrays, `std` operates along the first nonsingleton dimension of `X`.

`std` normalizes by n-1 where n is the sample size. The result `Y` is the square root of an unbiased estimator of the variance of the population from which `X` is drawn, as long as `X` consists of independent, identically distributed samples.

The standard deviation is

$$y = \left(\frac{1}{n-1}\sum_{i=1}^{n}(x_i-\bar{x})^2\right)^{\frac{1}{2}}$$

where the sample mean is $\bar{x} = \frac{1}{n}\sum x_i$.

The `std` function is part of the standard MATLAB language.

`Y = std(X,1)` normalizes `Y` by n. The result `Y` is the square root of the second moment of the sample about its mean. `std(X,0)` is the same as `std(X)`.

`Y = std(X,flag,dim)` takes the standard deviation along the dimension `dim` of `X`. Set `flag` to `0` to normalize `Y` by n-1; set `flag` to `1` to normalize by n.

Examples

In each column, the expected value of `y` is one.

```
x = normrnd(0,1,100,6);
y = std(x)
```

```
y =
  0.9536  1.0628  1.0860  0.9927  0.9605  1.0254

y = std(-1:2:1)
y =
  1.4142
```

See Also cov, var

Purpose

Interactive stepwise regression

Syntax

```
stepwise(X,y)
stepwise(X,y,inmodel,penter,premove)
```

Description

`stepwise(X,y)` displays a graphical user interface for performing a multilinear regression of the responses in `y` on a subset of the predictors in `X`. Distinct predictor variables should appear in different columns of `X`. Initially, no predictors are included in the model. Click on predictor names to switch them into and out of the model.

Note `stepwise` automatically includes a constant term in all models. Do not enter a column of ones directly into `X`.

For each predictor in the model, the interactive tool plots the predictor's least squares coefficient as a blue filled circle. For each predictor not in the model, the interactive tool plots a filled red circle to indicate the coefficient the predictor would have if you add it to the model. Horizontal bars in the plot indicate 90% confidence intervals (colored) and 95% confidence intervals (black).

`stepwise` treats `NaN`s in either `X` or `y` as missing values, and ignores them.

`stepwise(X,y,inmodel,penter,premove)` specifies the initial state of the model and the confidence levels to use. `inmodel` is either a logical vector, whose length is the number of columns in `X`, or a vector of indices, whose values range from 1 to the number of columns in `X`, specifying the predictors that are included in the initial model. The default is to include no columns of `X`. `penter` specifies the maximum *p*-value that a predictor can have for the interactive tool to recommend adding it to the model. The default value of `penter` is `0.05`. `premove` specifies the minimum p-value that a predictor can have for the interactive tool to recommend removing it from the model. The default value of `premove` is `0.10`.

Examples

See “Quadratic Response Surface Models” on page 7-12 and “Stepwise Regression Demo” on page 7-16.

Reference

[1] Draper, N., and H. Smith, *Applied Regression Analysis,* 2nd edition, John Wiley and Sons, 1981, pp. 307-312.

See Also

`regress`, `rstool`, `stepwisefit`

Purpose

Stepwise regression

Syntax

```
b = stepwisefit(X,y)
[b,se,pval,inmodel,stats,nextstep,history] = stepwisefit(...)
[...] = stepwisefit(X,y,param1,val1,param2,val2,...)
```

Description

`b = stepwisefit(X,y)` uses a stepwise method to perform a multilinear regression of the responses in `y` on the predictors in `X`. Distinct predictor variables should appear in different columns of `X`. `b` is a vector of estimated coefficients for all of the predictors in `X`. The value of `b` for a column not included in the final model is the coefficient estimate that you result from adding that column to the model.

Note `stepwise` automatically includes a constant term in all models. Do not enter a column of ones directly into `X`.

`stepwisefit` treats `NaN`s in either `X` or `y` as missing values, and ignores them.

`[b,se,pval,inmodel,stats,nextstep,history] = stepwisefit(...)` returns the following additional results:

- `se` is a vector of standard errors for `b`.
- `pval` is a vector of p-values for testing whether `b` is 0.
- `inmodel` is a logical vector, whose length equals the number of columns in `X`, specifying which predictors are in the final model. A `1` in position `j` indicates that the *j*th predictor is in the final model; a `0` indicates that the corresponding predictor in not in the final model.
- `stats` is a structure containing additional statistics.
- `nextstep` is the recommended next step—either the index of the next predictor to move in or out, or 0 if no further steps are recommended.
- `history` is a structure containing information about the history of steps taken.

`[...] = stepwisefit(X,y,`*`param1`*`,`*`val1`*`,`*`param2`*`,`*`val2`*`,...)` specifies one or more of the name/value pairs described in the following table.

Parameter Name	Parameter Value
`'inmodel'`	Logical vector specifying the predictors to include in the initial fit. The default is a vector specifying no predictors.
`'penter'`	Maximum p-value for a predictor to be added. The default is `0.05`.
`'premove'`	Minimum p-value for a predictor to be removed. The default is `0.10`.
`'display'`	`'on'` displays information about each step. `'off'` omits the information.
`'maxiter'`	Maximum number of steps to take (default is no maximum)
`'keep'`	Logical vector specifying the predictors to keep in their initial state. The default is a vector specifying no predictors.
`'scale'`	`'on'` scales each column of `X` by its standard deviation before fitting. `'off'` does not scale (the default).

Example

```
load hald
stepwisefit(ingredients, heat, 'penter', .08)
Initial columns included: none
Step 1, added column 4, p=0.000576232
Step 2, added column 1, p=1.10528e-006
Step 3, added column 2, p=0.0516873
Step 4, removed column 4, p=0.205395
Final columns included: 1 2
```

```
ans = 
    'Coeff'    'Std.Err.'   'Status'   'P'            
    [ 1.4683]  [ 0.1213]   'In'      [2.6922e-007]
    [ 0.6623]  [ 0.0459]   'In'      [5.0290e-008]
    [ 0.2500]  [ 0.1847]   'Out'      [   0.2089]
    [-0.2365]  [ 0.1733]   'Out'      [   0.2054]

ans =
    1.4683
    0.6623
    0.2500
   -0.2365
```

See Also

addedvarplot, regress, rstool, stepwise

summary (categorical)

Purpose Summary statistics for categorical array

Syntax

```
summary(A)
C = summary(A)
[C,labels] = summary(A)
```

Description `summary(A)` displays the number of elements in the categorical array `A` equal to each of the possible levels in `A`. If `A` contains any undefined elements, the output also includes the number of undefined elements.

`C = summary(A)` returns counts of the number of elements in the categorical array `A` equal to each of the possible levels in `A`. If `A` is a matrix or *N*-dimensional array, `C` is a matrix or array with rows corresponding to the levels of `A`. If `A` contains any undefined elements, `C` contains one more row than the number of levels of `A`, with the number of undefined elements in `c(end)` or `c(end,:)`.

`[C,labels] = summary(A)` also returns the list of categorical level labels corresponding to the counts in `C`.

Example Count the number of patients in each age group in the data in `hospital.mat`:

```
load hospital
edges = 0:10:100;
labels = strcat(num2str((0:10:90)','%d'),{'s'});
AgeGroup = ordinal(hospital.Age,labels,[],edges);
[c,labels] = summary(AgeGroup);

Table = dataset({labels,'AgeGroup'},{c,'Count'});
Table(3:6,:)
ans =
    AgeGroup     Count
    '20s'        15
    '30s'        41
    '40s'        42
    '50s'         2
```

See Also `islevel`, `ismember`, `levelcounts`

Purpose

Summary statistics for dataset array

Syntax

`summary(A)`

Description

`summary(A)` displays summaries of the variables in the dataset array `A`.

Summary information depends on the type of the variables in the data set:

- For numerical variables, `summary` computes a five-number summary of the data, giving the minimum, the first quartile, the median, the third quartile, and the maximum.
- For logical variables, `summary` counts the number of `trues` and `falses` in the data.
- For categorical variables, `summary` counts the number of data at each level.

Examples

Example 1

Summarize Fisher's iris data:

```
load fisheriris
species = nominal(species);
data = dataset(species,meas);
summary(data)
species: [150x1 nominal]
  setosa   versicolor   virginica    <undefined>
      50           50          50              0
meas: [150x4 double]
  min       4.3000         2         1    0.1000
  1st Q     5.1000    2.8000    1.6000    0.3000
  median    5.8000         3    4.3500    1.3000
  3rd Q     6.4000    3.3000    5.1000    1.8000
  max       7.9000    4.4000    6.9000    2.5000
```

Example 2

Summarize the data in `hospital.mat`:

```
load hospital
summary(hospital)

A dataset array created from the data file hospital.dat.
It has the first column of that file as observations
names, and has had several other columns converted to a
more convenient form.

LastName: [100x1 cell string]
Sex: [100x1 nominal]
   Female    Male
       53      47
Age: [100x1 double, Units = Yrs]
   min    1st Q    median    3rd Q    max
    25       32        39       44     50
Weight: [100x1 double, Units = Lbs]
   min    1st Q       median      3rd Q       max
   111    130.5000    142.5000    180.5000    202
Smoker: [100x1 logical]
   true    false
     34       66
BloodPressure: [100x2 double, Units = mm Hg]
   min             109         68
   1st Q     117.5000    77.5000
   median          122    81.5000
   3rd Q     127.5000         89
   max             138         99
Trials: [100x1 cell, Units = Counts]
```

See Also

`get`, `set`, `grpstats (dataset)`

surfht

Purpose Interactive contour plot

Syntax

```
surfht(Z)
surfht(x,y,Z)
```

Description `surfht(Z)` is an interactive contour plot of the matrix `Z` treating the values in `Z` as height above the plane. The *x*-values are the column indices of `Z` while the *y*-values are the row indices of `Z`.

`surfht(x,y,Z)` where `x` and `y` are vectors specify the *x* and *y*-axes on the contour plot. The length of `x` must match the number of columns in `Z`, and the length of `y` must match the number of rows in `Z`.

There are vertical and horizontal reference lines on the plot whose intersection defines the current *x*-value and *y*-value. You can drag these dotted white reference lines and watch the interpolated *z*-value (at the top of the plot) update simultaneously. Alternatively, you can get a specific interpolated *z*-value by typing the *x*-value and *y*-value into editable text fields on the *x*-axis and *y*-axis respectively.

Purpose

Frequency table

Syntax

```
TABLE = tabulate(x)
tabulate(x)
```

Description

`TABLE = tabulate(x)` creates a frequency table of data in vector `x`. Information in `TABLE` is arranged as follows:

- 1st column — The unique values of `x`
- 2nd column — The number of instances of each value
- 3rd column — The percentage of each value

If `x` is a numeric array, `TABLE` is a numeric matrix. If the elements of `x` are non-negative integers, `TABLE` includes 0 counts for integers between `1` and `max(x)` that do not appear in `x`.

If `x` is a categorical variable, character array, or cell array of strings, `TABLE` is a cell array.

`tabulate(x)` with no output arguments displays the table in the command window.

Example

```
tabulate([1 2 4 4 3 4])
  Value   Count   Percent
  1       1       16.67%
  2       1       16.67%
  3       1       16.67%
  4       3       50.00%
```

See Also

`pareto`

tblread

Purpose

Read tabular data from file

Syntax

```
[data,varnames,casenames] = tblread
[data,varnames,casenames] = tblread(filename)
[data,varnames,casenames] = tblread(filename,delimiter)
```

Description

`[data,varnames,casenames] = tblread` displays the File Open dialog box for interactive selection of a tabular data file. The file format has variable names in the first row, case names in the first column and data starting in the (2, 2) position. Outputs are:

- `data` — Numeric matrix with a value for each variable-case pair
- `varnames` — String matrix containing the variable names in the first row of the file
- `casenames` — String matrix containing the names of each case in the first column of the file

`[data,varnames,casenames] = tblread(`*`filename`*`)` allows command line specification of the name of a file in the current directory, or the complete path name of any file, using the string *`filename`*.

`[data,varnames,casenames] = tblread(`*`filename`*`,`*`delimiter`*`)` reads from the file using *`delimiter`* as the delimiting character. Accepted values for *`delimiter`* are:

- `' '` or `'space'`
- `'\t'` or `'tab'`
- `','` or `'comma'`
- `';'` or `'semi'`
- `'|'` or `'bar'`

The default value of *`delimiter`* is `'space'`.

Example

```
[data,varnames,casenames] = tblread('sat.dat')
data =
  470  530
  520  480

varnames =
Male
Female

casenames =
Verbal
Quantitative
```

See Also

`tblwrite`, `tdfread`, `caseread`

tblwrite

Purpose

Write tabular data to file

Syntax

```
tblwrite(data,varnames,casenames)
tblwrite(data,varnames,casenames,filename)
tblwrite(data,varnames,casenames,filename,delimiter)
```

Description

`tblwrite(data,varnames,casenames)` displays the **File Open** dialog box for interactive specification of the tabular data output file. The file format has variable names in the first row, case names in the first column and `data` starting in the (2,2) position.

`varnames` is a string matrix containing the variable names. `casenames` is a string matrix containing the names of each case in the first column. `data` is a numeric matrix with a value for each variable-case pair.

`tblwrite(data,varnames,casenames,`*`filename`*`)` specifies a file in the current directory, or the complete path name of any file in the string *`filename`*.

`tblwrite(data,varnames,casenames,`*`filename`*`,`*`delimiter`*`)` writes to the file using *`delimiter`* as the delimiting character. The following table lists the accepted character values for *`delimiter`* and their equivalent string values.

Character	String
`' '`	`'space'`
`'\t'`	`'tab'`
`','`	`'comma'`
`';'`	`'semi'`
`'|'`	`'bar'`

The default value of `'`*`delimiter`*`'` is `'space'`.

Example

Continuing the example from `tblread`:

```
tblwrite(data,varnames,casenames,'sattest.dat')
```

```
type sattest.dat
              Male  Female
Verbal      470  530
Quantitative  520  480
```

See Also `casewrite`, `tblread`

Purpose

Student's *t* cumulative distribution function

Syntax

```
P = tcdf(X,V)
```

Description

`P = tcdf(X,V)` computes Student's *t* cdf at each of the values in `X` using the corresponding degrees of freedom in `V`. `X` and `V` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs.

The *t* cdf is

$$p = F(x|\nu) = \int_{-\infty}^{x} \frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\Gamma\left(\frac{\nu}{2}\right)} \frac{1}{\sqrt{\nu\pi}} \frac{1}{\left(1+\frac{t^2}{\nu}\right)^{\frac{\nu+1}{2}}} dt$$

The result, p, is the probability that a single observation from the t distribution with ν degrees of freedom will fall in the interval $[-\infty, x)$.

Examples

```
mu = 1; % Population mean
sigma = 2; % Population standard deviation
n = 100; % Sample size
x = normrnd(mu,sigma,n,1); % Random sample from population
xbar = mean(x); % Sample mean
s = std(x); % Sample standard deviation
t = (xbar-mu)/(s/sqrt(n)) % t-statistic
t =
   0.2489
p = 1-tcdf(t,n-1) % Probability of larger t-statistic
p =
   0.4020
```

This probability is the same as the *p*-value returned by a *t*-test of the null hypothesis that the sample comes from a normal population with mean μ:

```
[h,ptest] = ttest(x,mu,0.05,'right')
h =
     0
ptest =
    0.4020
```

See Also

`cdf`, `tinv`, `tpdf`, `trnd`, `tstat`

tdfread

Purpose

Read file containing tab-delimited numeric and text values

Syntax

```
tdfread
tdfread(filename)
tdfread(filename,delimiter)
s = tdfread(filename,...)
```

Description

`tdfread` displays the **File Open** dialog box for interactive selection of a data file, then reads data from the file. The file should have variable names separated by tabs in the first row, and data values separated by tabs in the remaining rows. `tdfread` creates variables in the workspace, one for each column of the file. The variable names are taken from the first row of the file. If a column of the file contains only numeric data in the second and following rows, `tdfread` creates a `double` variable. Otherwise, `tdfread` creates a `char` variable. After all values are imported, `tdfread` displays information about the imported values using the format of the `tdfread` command.

`tdfread(`*`filename`*`)` allows command line specification of the name of a file in the current directory, or the complete path name of any file, using the string *`filename`*.

`tdfread(`*`filename`*`,`*`delimiter`*`)` indicates that the character specified by *`delimiter`* separates columns in the file. Accepted values for *`delimiter`* are:

- `' '` or `'space'`
- `'\t'` or `'tab'`
- `','` or `'comma'`
- `';'` or `'semi'`
- `'|'` or `'bar'`

The default delimiter is `'tab'`.

`s = tdfread(`*`filename`*`,...)` returns a scalar structure `s` whose fields each contain a variable.

Example

```
type sat2.dat

Test,Gender,Score
Verbal,Male,470
Verbal,Female,530
Quantitative,Male,520
Quantitative,Female,480

tdfread('sat2.dat',',')

 Name      Size      Bytes Class
 Gender    4x6        48 char array
 Score     4x1        32 double array
 Test      4x12       96 char array

Grand total is 76 elements using 176 bytes
```

See Also

`tblread`, `caseread`

Purpose

Error rate of tree

Syntax

```
cost = test(t,'resubstitution')
cost = test(t,'test',X,y)
cost = test(t,'crossvalidate',X,y)
[cost,secost,ntnodes,bestlevel] = test(...)
[...] = test(...,param1,val1,param2,val2,...)
```

Description

`cost = test(t,'resubstitution')` computes the cost of the tree `t` using a resubstitution method. `t` is a decision tree as created by `classregtree`. The cost of the tree is the sum over all terminal nodes of the estimated probability of a node times the cost of a node. If `t` is a classification tree, the cost of a node is the sum of the misclassification costs of the observations in that node. If `t` is a regression tree, the cost of a node is the average squared error over the observations in that node. `cost` is a vector of cost values for each subtree in the optimal pruning sequence for `t`. The resubstitution cost is based on the same sample that was used to create the original tree, so it under estimates the likely cost of applying the tree to new data.

`cost = test(t,'test',X,y)` uses the matrix of predictors `X` and the response vector `y` as a test sample, applies the decision tree `t` to that sample, and returns a vector `cost` of cost values computed for the test sample. `X` and `y` should not be the same as the learning sample, that is, the sample that was used to fit the tree `t`.

`cost = test(t,'crossvalidate',X,y)` uses 10-fold cross-validation to compute the cost vector. `X` and `y` should be the learning sample, that is, the sample that was used to fit the tree `t`. The function partitions the sample into 10 subsamples, chosen randomly but with roughly equal size. For classification trees, the subsamples also have roughly the same class proportions. For each subsample, `test` fits a tree to the remaining data and uses it to predict the subsample. It pools the information from all subsamples to compute the cost for the whole sample.

`[cost,secost,ntnodes,bestlevel] = test(...)` also returns the vector `secost` containing the standard error of each `cost` value, the vector `ntnodes` containing the number of terminal nodes for each subtree, and the scalar `bestlevel` containing the estimated best level of pruning. A `bestlevel` of `0` means no pruning. The best level is the one that produces the smallest tree that is within one standard error of the minimum-cost subtree.

`[...] = test(...,param1,val1,param2,val2,...)` specifies optional parameter name/value pairs chosen from the following:

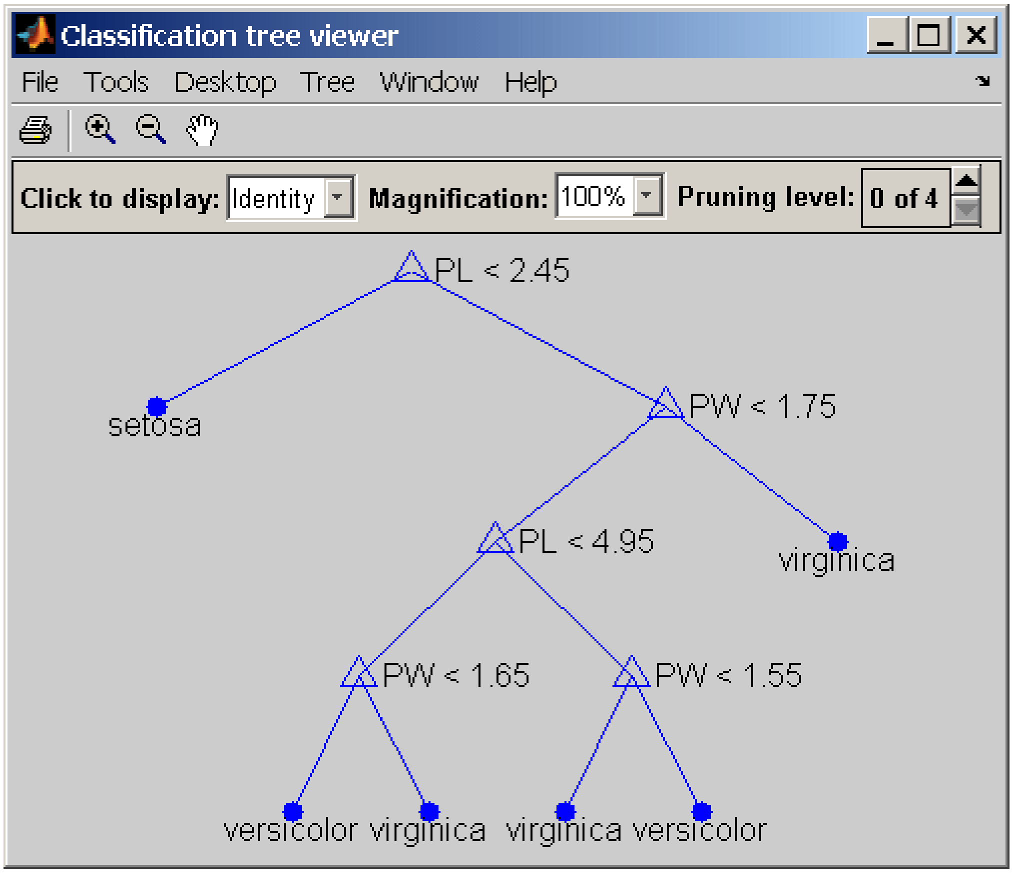
- `'nsamples'` — The number of cross-validation samples (default is `10`).
- `'treesize'` — Either `'se'` (default) to choose the smallest tree whose cost is within one standard error of the minimum cost, or `'min'` to choose the minimal cost tree (not meaningful for resubstitution error calculations).

Example

Find the best tree for Fisher's iris data using cross-validation. Start with a large tree:

```
load fisheriris;

t = classregtree(meas,species,...
                 'names',{'SL' 'SW' 'PL' 'PW'},...
                 'splitmin',5)
t =
Decision tree for classification
 1  if PL<2.45 then node 2 else node 3
 2  class = setosa
 3  if PW<1.75 then node 4 else node 5
 4  if PL<4.95 then node 6 else node 7
 5  class = virginica
 6  if PW<1.65 then node 8 else node 9
 7  if PW<1.55 then node 10 else node 11
 8  class = versicolor
 9  class = virginica
10  class = virginica
11  class = versicolor
```

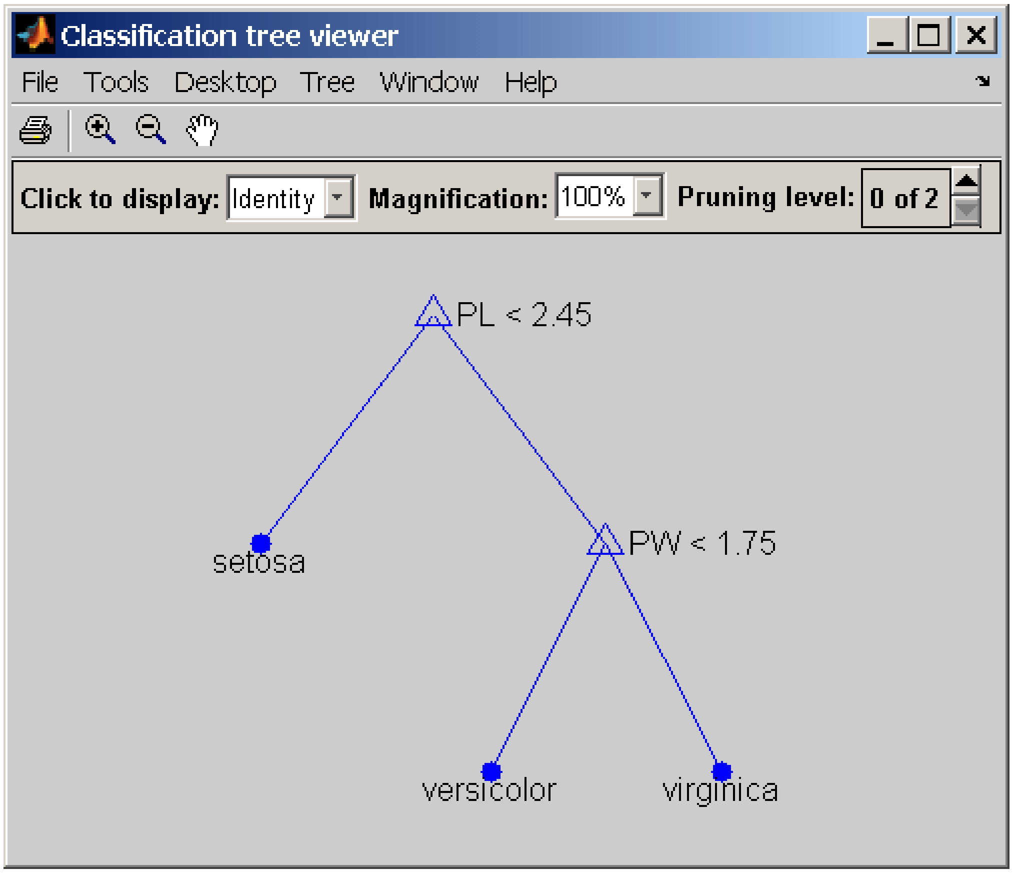
```
view(t)
```



Find the minimum-cost tree:

```
[c,s,n,best] = test(t,'cross',meas,species);
tmin = prune(t,'level',best)
tmin =
Decision tree for classification
1  if PL<2.45 then node 2 else node 3
2  class = setosa
```

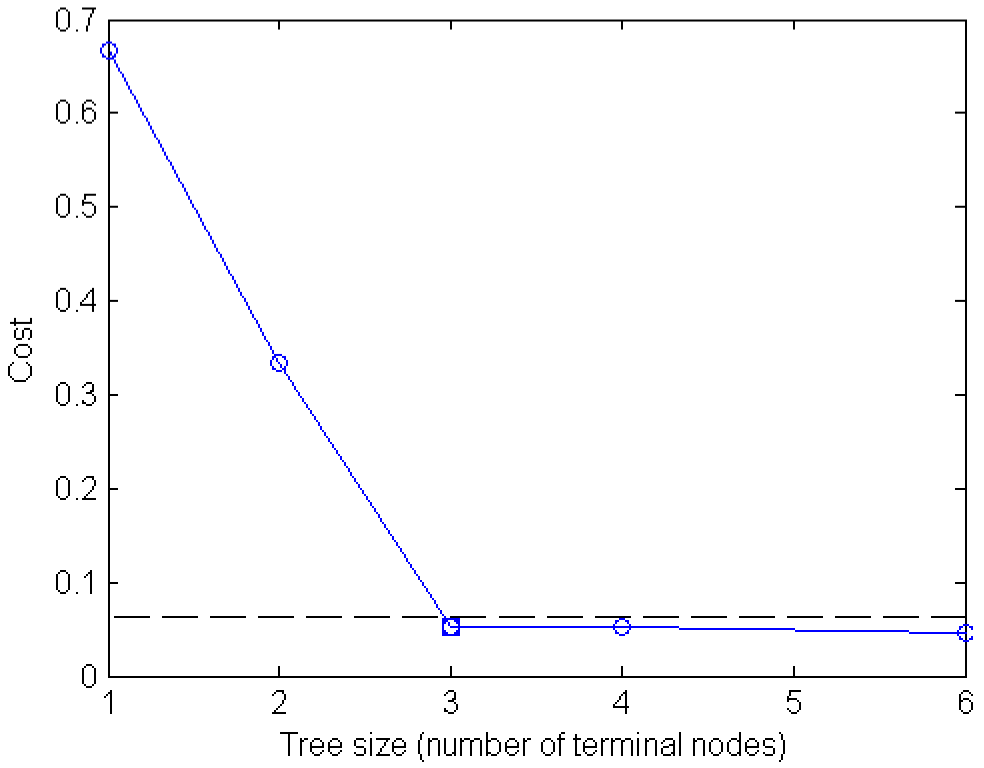
```
3  if PW<1.75 then node 4 else node 5
4  class = versicolor
5  class = virginica

view(tmin)
```


Classification tree viewer
File Tools Desktop Tree Window Help
Click to display: Identity
Magnification: 100%
Pruning level: 0 of 2
PL < 2.45
setosa
PW < 1.75
versicolor
virginica

Plot the smallest tree within one standard error of the minimum cost tree:

```
[mincost,minloc] = min(c);
```

```
plot(n,c,'b-o',...
     n(best+1),c(best+1),'bs',...
     n,(mincost+s(minloc))*ones(size(n)),'k--')
xlabel('Tree size (number of terminal nodes)')
ylabel('Cost')
```



The solid line shows the estimated cost for each tree size, the dashed line marks one standard error above the minimum, and the square marks the smallest tree under the dashed line.

Reference

[1] Breiman, L., et al., *Classification and Regression Trees*, Chapman & Hall, Boca Raton, 1993.

See Also

`classregtree`, `eval`, `view`, `prune`

Purpose

Compute ranks of sample, adjusting for ties

Syntax

```
[R,TIEADJ] = tiedrank(X)
[R,TIEADJ] = tiedrank(X,1)
[R,TIEADJ] = tiedrank(X,0,1)
```

Description

`[R,TIEADJ] = tiedrank(X)` computes the ranks of the values in the vector `X`. If any `X` values are tied, `tiedrank` computes their average rank. The return value `TIEADJ` is an adjustment for ties required by the nonparametric tests `signrank` and `ranksum`, and for the computation of Spearman's rank correlation.

`[R,TIEADJ] = tiedrank(X,1)` computes the ranks of the values in the vector `X`. `TIEADJ` is a vector of three adjustments for ties required in the computation of Kendall's tau.`tiedrank(X,0)` is the same as `tiedrank(X)`.

`[R,TIEADJ] = tiedrank(X,0,1)` computes the ranks from each end, so that the smallest and largest values get rank 1, the next smallest and largest get rank 2, etc. These ranks are used in the Ansari-Bradley test.

See Also

`ansaribradley`, `corr`, `partialcorr`, `ranksum`, `signrank`

Purpose

Inverse of Student's *t* cumulative distribution function

Syntax

```
X = tinv(P,V)
```

Description

`X = tinv(P,V)` computes the inverse of Student's *t* cdf with parameter `V` for the corresponding probabilities in `P`. `P` and `V` can be vectors, matrices, or multidimensional arrays that have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs. The values in `P` must lie on the interval [0 1].

The t inverse function in terms of the *t* cdf is

$$x = F^{-1}(p|\nu) = \{x: F(x|\nu) = p\}$$

where

$$p = F(x|\nu) = \int_{-\infty}^{x} \frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\Gamma\left(\frac{\nu}{2}\right)} \frac{1}{\sqrt{\nu\pi}} \frac{1}{\left(1+\frac{t^2}{\nu}\right)^{\frac{\nu+1}{2}}} dt$$

The result, x, is the solution of the cdf integral with parameter ν, where you supply the desired probability p.

Examples

What is the 99th percentile of the *t* distribution for one to six degrees of freedom?

```
percentile = tinv(0.99,1:6)
percentile =
  31.8205  6.9646  4.5407  3.7469  3.3649  3.1427
```

See Also

`icdf`, `tcdf`, `tpdf`, `trnd`, `tstat`

Purpose

Student's *t* probability density function

Syntax

```
Y = tpdf(X,V)
```

Description

`Y = tpdf(X,V)` computes Student's *t* pdf at each of the values in `X` using the corresponding degrees of freedom in `V`. `X` and `V` can be vectors, matrices, or multidimensional arrays that have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs.

Student's *t* pdf is

$$y = f(x|\nu) = \frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\Gamma\left(\frac{\nu}{2}\right)} \frac{1}{\sqrt{\nu\pi}} \frac{1}{\left(1+\frac{x^2}{\nu}\right)^{\frac{\nu+1}{2}}}$$

Examples

The mode of the t distribution is at $x = 0$. This example shows that the value of the function at the mode is an increasing function of the degrees of freedom.

```
tpdf(0,1:6)
ans =
  0.3183  0.3536  0.3676  0.3750  0.3796  0.3827
```

The *t* distribution converges to the standard normal distribution as the degrees of freedom approaches infinity. How good is the approximation for $v = 30$?

```
difference = tpdf(-2.5:2.5,30)-normpdf(-2.5:2.5)
difference =
  0.0035  -0.0006  -0.0042  -0.0042  -0.0006  0.0035
```

See Also

`pdf`, `tcdf`, `tinv`, `trnd`, `tstat`

Purpose

Plot classification and regression trees

Syntax

```
treedisp(t)
treedisp(t,param1,val1,param2,val2,...)
```

Description

Note This function is superseded by the `view` method for the `classregtree` class and is maintained only for backwards compatibility. It accepts objects `t` created with the `classregtree` constructor.

`treedisp(t)` takes as input a decision tree `t` as computed by the `treefit` function, and displays it in a figure window. Each branch in the tree is labeled with its decision rule, and each terminal node is labeled with the predicted value for that node.

For each branch node, the left child node corresponds to the points that satisfy the condition, and the right child node corresponds to the points that do not satisfy the condition.

The **Click to display** pop-up menu at the top of the figure enables you to display more information about each node:

`Identity`	The node number, whether the node is a branch or a leaf, and the rule that governs the node
`Variable ranges`	The range of each of the predictor variables for that node
`Node statistics`	Descriptive statistics for the observations falling into this node

After you select the type of information you want, click any node to display the information for that node.

The **Pruning level** button displays the number of levels that have been cut from the tree and the number of levels in the unpruned tree. For example, `1 of 6` indicates that the unpruned tree has six levels, and that one level has been cut from the tree. Use the spin button to change the pruning level.

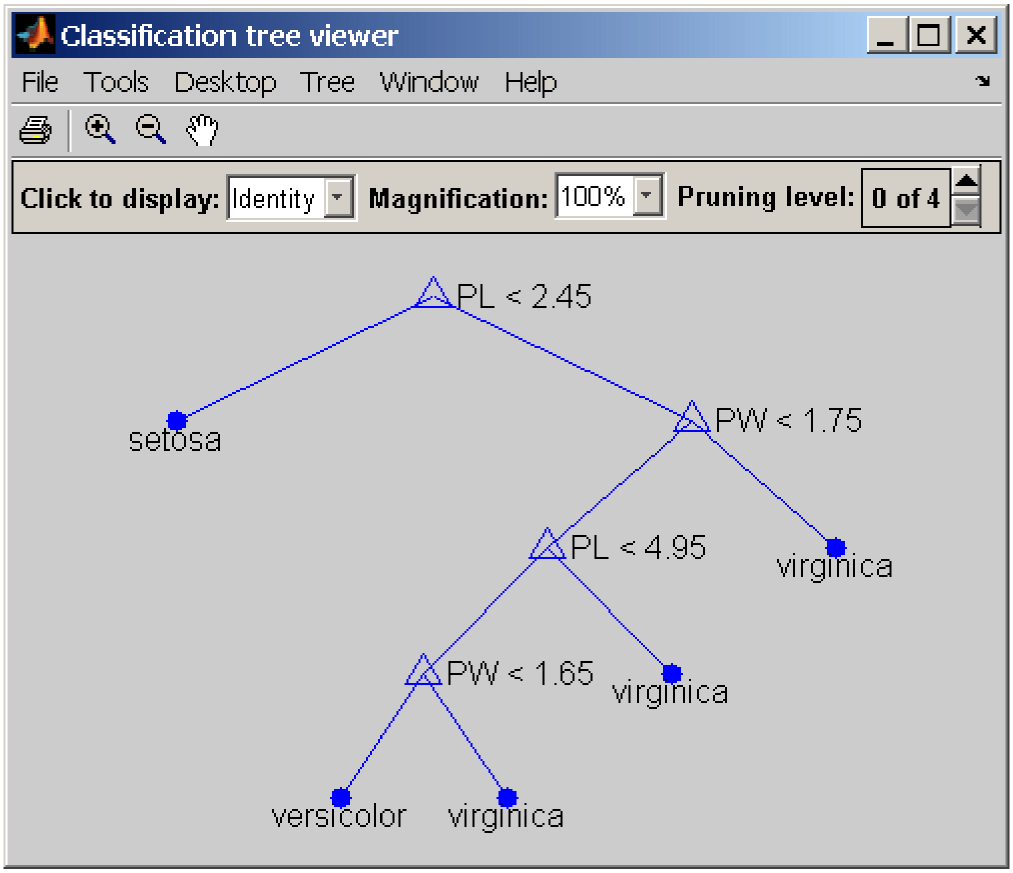
`treedisp(t,`*`param1`*`,`*`val1`*`,`*`param2`*`,`*`val2`*`,...)` specifies optional parameter name-value pairs. Valid parameter strings are:

`'names'`	A cell array of names for the predictor variables, in the order in which they appear in the `X` matrix from which the tree was created (see `treefit`)
`'prunelevel'`	Initial pruning level to display

Examples

Create and graph classification tree for Fisher's iris data. The names in this example are abbreviations for the column contents (sepal length, sepal width, petal length, and petal width).

```
load fisheriris;
t = treefit(meas,species);
treedisp(t,'names',{'SL' 'SW' 'PL' 'PW'});
```



Click to display: Identity
Magnification: 100%
Pruning level: 0 of 4

Reference

[1] Breiman, L., et al., *Classification and Regression Trees*, Chapman & Hall, Boca Raton, 1993.

See Also

`treefit`, `treeprune`, `treetest`

Purpose

Fit tree-based model for classification or regression

Syntax

```
t = treefit(X,y)
t = treefit(X,y,param1,val1,param2,val2,...)
```

Description

Note This function is superseded by the `classregtree` constructor and is maintained only for backwards compatibility. It returns objects `t` in the `classregtree` class.

`t = treefit(X,y)` creates a decision tree `t` for predicting response `y` as a function of predictors `X`. `X` is an `n`-by-`m` matrix of predictor values. `y` is either a vector of `n` response values (for regression), or a character array or cell array of strings containing `n` class names (for classification). Either way, `t` is a binary tree where each non-terminal node is split based on the values of a column of `X`.

`t = treefit(X,y,`*`param1`*`,`*`val1`*`,`*`param2`*`,`*`val2`*`,...)` specifies optional parameter name-value pairs. Valid parameter strings are:

For all trees:

`'catidx'`	Vector of indices of the columns of `X`. `treefit` treats these columns as unordered categorical values.
`'method'`	Either `'classification'` (default if `y` is text) or `'regression'` (default if `y` is numeric).
`'splitmin'`	A number `n` such that impure nodes must have `n` or more observations to be split (default 10).
`'prune'`	`'on'` (default) to compute the full tree and a sequence of pruned subtrees, or `'off'` for the full tree without pruning.

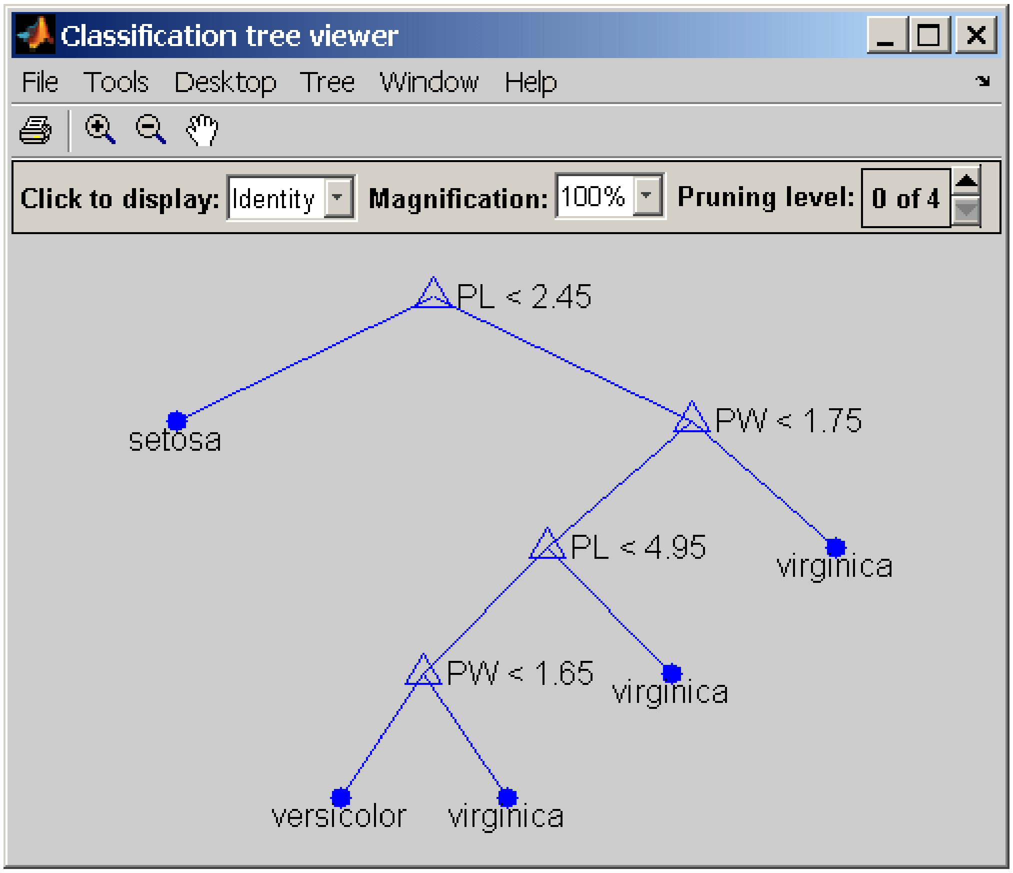
For classification trees only:

`'cost'`	`p`-by-`p` matrix `C`, where `p` is the number of distinct response values or class names in the input `y`. `C(i,j)` is the cost of classifying a point into class `j` if its true class is `i`. (The default has `C(i,j)=1` if `i~=j`, and `C(i,j)=0` if `i=j`.) `C` can also be a structure `S` with two fields: `S.group` containing the group names (see "Grouped Data" on page 2-41), and `S.cost` containing a matrix of cost values.
`'splitcriterion'`	Criterion for choosing a split: either `'gdi'` (default) for Gini's diversity index, `'twoing'` for the twoing rule, or `'deviance'` for maximum deviance reduction.
`'priorprob'`	Prior probabilities for each class, specified as a vector (one value for each distinct group name) or as a structure `S` with two fields: `S.group` containing the group names, and `S.prob` containing a vector of corresponding probabilities.

Examples

Create a classification tree for Fisher's iris data:

```
load fisheriris;
t = treefit(meas,species);
treedisp(t,'names',{'SL' 'SW' 'PL' 'PW'});
```



Reference

[1] Breiman, L., et al., *Classification and Regression Trees*, Chapman & Hall, Boca Raton, 1993.

See Also

`treedisp`, `treetest`

Purpose

Produce sequence of subtrees by pruning

Syntax

```
t2 = treeprune(t1,'level',level)
t2 = treeprune(t1,'nodes',nodes)
t2 = treeprune(t1)
```

Description

Note This function is superseded by the `prune` method for the `classregtree` class and is maintained only for backwards compatibility. It accepts objects `t1` created with the `classregtree` constructor and returns objects `t2` in the `classregtree` class.

`t2 = treeprune(t1,'level',level)` takes a decision tree `t1` as created by the `treefit` function, and a pruning level, and returns the decision tree `t2` pruned to that level. Setting `level` to `0` means no pruning. Trees are pruned based on an optimal pruning scheme that first prunes branches giving less improvement in error cost.

`t2 = treeprune(t1,'nodes',nodes)` prunes the nodes listed in the `nodes` vector from the tree. Any `t1` branch nodes listed in `nodes` become leaf nodes in `t2`, unless their parent nodes are also pruned. The `treedisp` function can display the node numbers for any node you select.

`t2 = treeprune(t1)` returns the decision tree `t2` that is the same as `t1`, but with the optimal pruning information added. This is useful only if you created `t1` by pruning another tree, or by using the `treefit` function with pruning set `'off'`. If you plan to prune a tree multiple times, it is more efficient to create the optimal pruning sequence first.

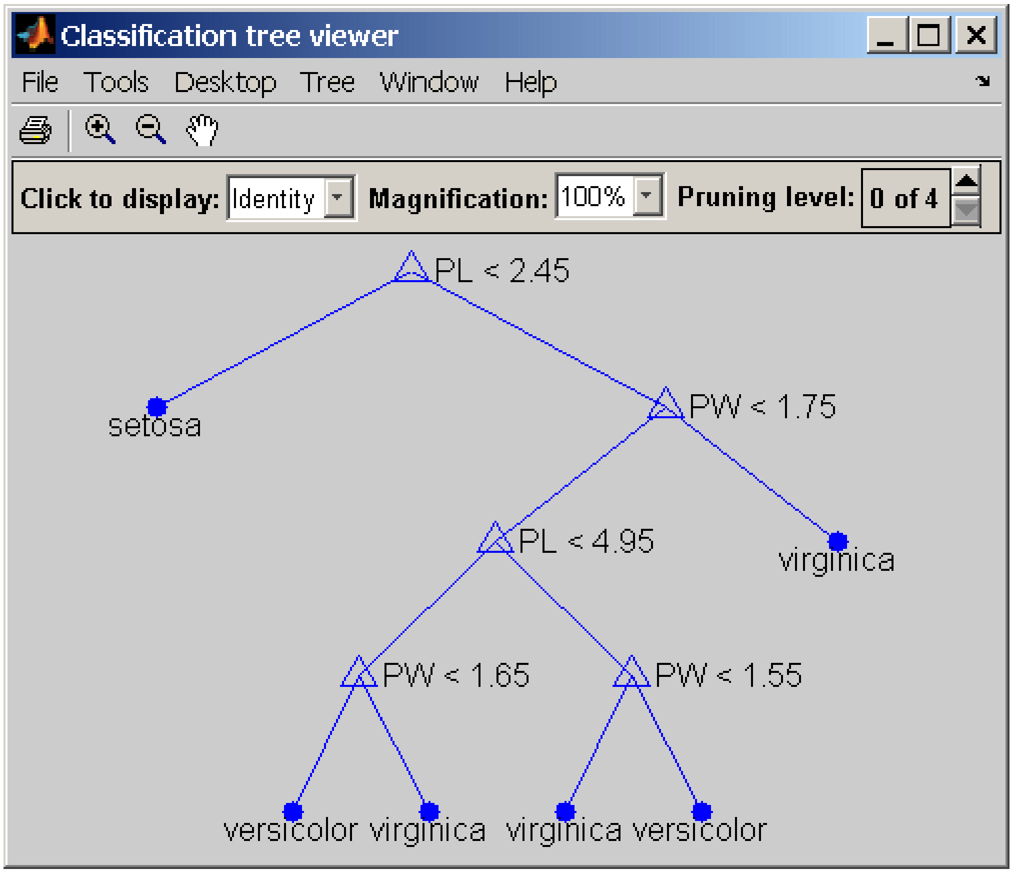
Pruning is the process of reducing a tree by turning some branch nodes into leaf nodes, and removing the leaf nodes under the original branch.

Examples

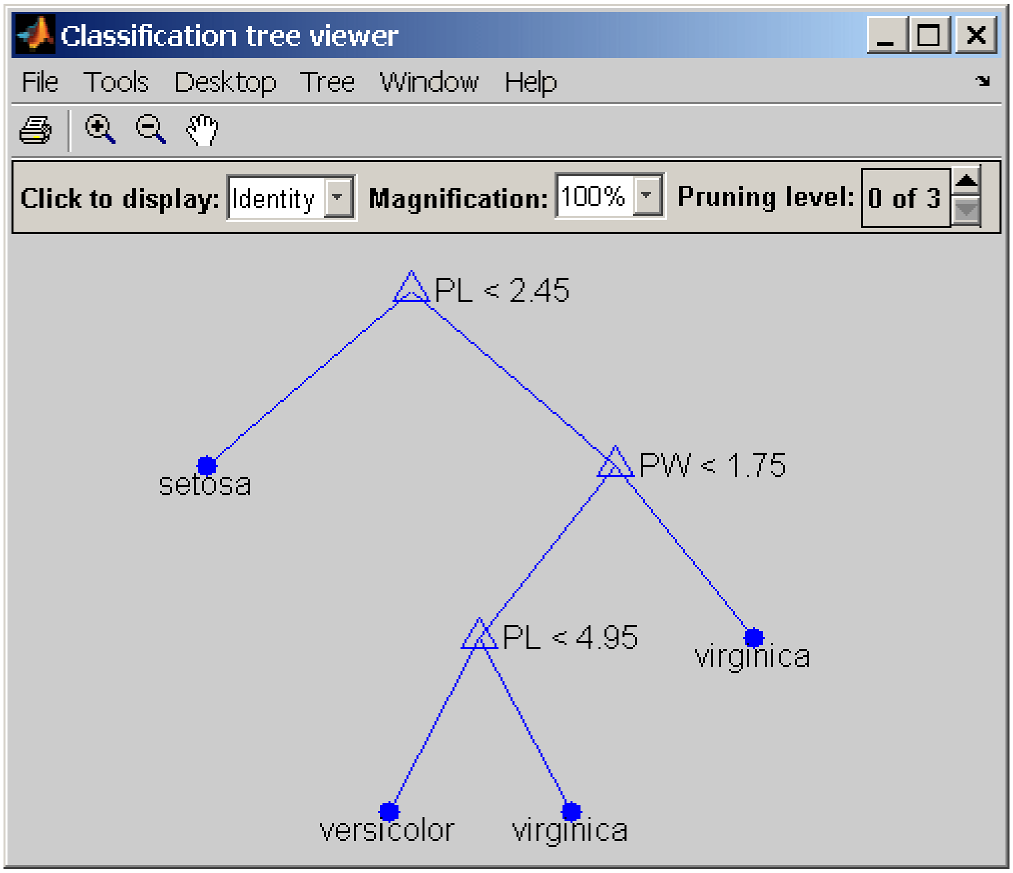
Display the full tree for Fisher's iris data, as well as the next largest tree from the optimal pruning sequence:

```
load fisheriris;
t1 = treefit(meas,species,'splitmin',5);
```

```
treedisp(t1,'names',{'SL' 'SW' 'PL' 'PW'});
```



```
t2 = treeprune(t1,'level',1);
treedisp(t2,'names',{'SL' 'SW' 'PL' 'PW'});
```



Reference [1] Breiman, L., et al., *Classification and Regression Trees*, Chapman & Hall, Boca Raton, 1993.

See Also `treefit`, `treetest`, `treedisp`

Purpose

Compute error rate for tree

Syntax

```
cost = treetest(t,'resubstitution')
cost = treetest(t,'test',X,y)
cost = treetest(t,'crossvalidate',X,y)
[cost,secost,ntnodes,bestlevel] = treetest(...)
[...] = treetest(...,param1,val1,param2,val2,...)
```

Description

Note This function is superseded by the `test` method for the `classregtree` class and is maintained only for backwards compatibility. It accepts objects `t` created with the `classregtree` constructor.

`cost = treetest(t,'resubstitution')` computes the cost of the tree `t` using a resubstitution method. `t` is a decision tree as created by the `treefit` function. The cost of the tree is the sum over all terminal nodes of the estimated probability of that node times the node's cost. If `t` is a classification tree, the cost of a node is the sum of the misclassification costs of the observations in that node. If `t` is a regression tree, the cost of a node is the average squared error over the observations in that node. `cost` is a vector of cost values for each subtree in the optimal pruning sequence for `t`. The resubstitution cost is based on the same sample that was used to create the original tree, so it underestimates the likely cost of applying the tree to new data.

`cost = treetest(t,'test',X,y)` uses the predictor matrix `X` and response `y` as a test sample, applies the decision tree `t` to that sample, and returns a vector `cost` of cost values computed for the test sample. `X` and `y` should not be the same as the learning sample, which is the sample that was used to fit the tree `t`.

`cost = treetest(t,'crossvalidate',X,y)` uses 10-fold cross-validation to compute the cost vector. `X` and `y` should be the learning sample, which is the sample that was used to fit the tree `t`. The function partitions the sample into 10 subsamples, chosen randomly but with roughly equal size. For classification trees, the subsamples also have roughly the same class proportions. For each subsample, `treetest` fits a tree to the remaining data and uses it to predict the subsample. It pools the information from all subsamples to compute the cost for the whole sample.

`[cost,secost,ntnodes,bestlevel] = treetest(...)` also returns the vector `secost` containing the standard error of each `cost` value, the vector `ntnodes` containing number of terminal nodes for each subtree, and the scalar `bestlevel` containing the estimated best level of pruning. `bestlevel = 0` means no pruning, i.e., the full unpruned tree. The best level is the one that produces the smallest tree that is within one standard error of the minimum-cost subtree.

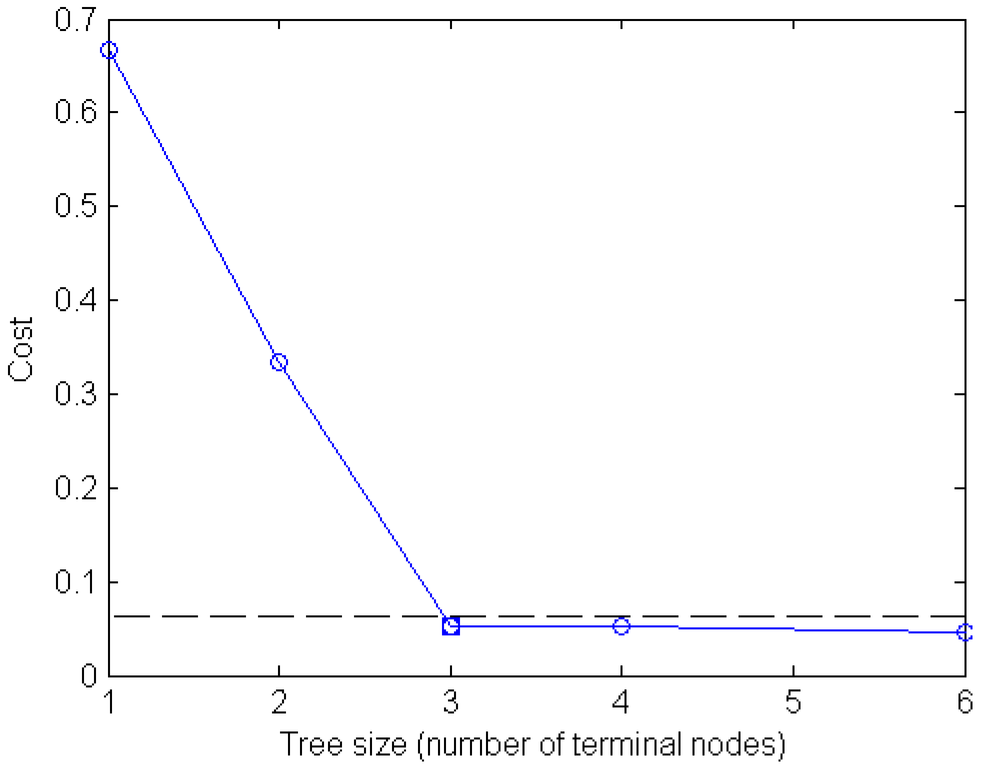
`[...] = treetest(...,`*`param1`*`,`*`val1`*`,`*`param2`*`,`*`val2`*`,...)` specifies optional parameter name-value pairs chosen from the following:

`'nsamples'`	The number of cross-validations samples (default is 10).
`'treesize'`	Either `'se'` (default) to choose the smallest tree whose cost is within one standard error of the minimum cost, or `'min'` to choose the minimal cost tree.

Examples

Find the best tree for Fisher's iris data using cross-validation. The solid line shows the estimated cost for each tree size, the dashed line marks one standard error above the minimum, and the square marks the smallest tree under the dashed line.

```
% Start with a large tree.
load fisheriris;
t = treefit(meas,species','splitmin',5);

% Find the minimum-cost tree.
[c,s,n,best] = treetest(t,'cross',meas,species);
tmin = treeprune(t,'level',best);

% Plot smallest tree within 1 std of minimum cost tree.
[mincost,minloc] = min(c);
plot(n,c,'b-o',n,c+s,'r:',...
          n(best+1),c(best+1),'bs',...
          n,(mincost+s(minloc))*ones(size(n)),'k ');
xlabel('Tree size (number of terminal nodes)')
ylabel('Cost')
```



Reference

[1] Breiman, L., et al., *Classification and Regression Trees*, Chapman & Hall, Boca Raton, 1993.

See Also

`treefit`, `treedisp`

Purpose

Compute fitted value for decision tree applied to data

Syntax

```
yfit = treeval(t,X)
yfit = treeval(t,X,subtrees)
[yfit,node] = treeval(...)
[yfit,node,cname] = treeval(...)
```

Description

Note This function is superseded by the `eval` method for the `classregtree` class and is maintained only for backwards compatibility. It accepts objects `t` created with the `classregtree` constructor.

`yfit = treeval(t,X)` takes a classification or regression tree `t` as produced by the `treefit` function and a matrix `X` of predictor values, and produces a vector `yfit` of predicted response values. For a regression tree, `yfit(i)` is the fitted response value for a point having the predictor values `X(i,:)`. For a classification tree, `yfit(i)` is the class number into which the tree would assign the point with data `X(i,:)`. To convert the number into a class name, use the third output argument, `cname` (described below).

`yfit = treeval(t,X,subtrees)` takes an additional vector `subtrees` of pruning levels, with `0` representing the full, unpruned tree. `T` must include a pruning sequence as created by the `treefit` or `prunetree` function. If `subtree` has k elements and `X` has n rows, the output `yfit` is an n-by-k matrix, with the `j`th column containing the fitted values produced by the `subtrees(j)` subtree. `subtrees` must be sorted in ascending order.

`[yfit,node] = treeval(...)` also returns an array `node` of the same size as `yfit` containing the node number assigned to each row of `X`. The `treedisp` function can display the node numbers for any node you select.

`[yfit,node,cname] = treeval(...)` is valid only for classification trees. It returns a cell array `cname` containing the predicted class names.

Examples

Find the predicted classifications for Fisher's iris data:

```
load fisheriris;
t = treefit(meas,species);  % Create decision tree
sfit = treeval(t,meas);     % Find assigned class numbers
sfit = t.classname(sfit);   % Get class names
mean(strcmp(sfit,species))  % Proportion correctly classified
ans =
   0.9800
```

Reference

[1] Breiman, L., et al., *Classification and Regression Trees*, Chapman & Hall, Boca Raton, 1993.

See Also

`treefit`, `treeprune`, `treetest`

Purpose

Mean of sample, excluding extreme values

Syntax

```
m = trimmean(X,percent)
trimmean(X,percent,dim)
```

Description

`m = trimmean(X,percent)` calculates the mean of a sample X excluding the highest and lowest (percent`/2`)% of the observations. For a vector input, m is the trimmed mean of `X`. For a matrix input, `m` is a row vector containing the trimmed mean of each column of `X`. For N-dimensional arrays, `trimmean` operates along the first nonsingleton dimension of `X`. `percent` is a scalar between `0` and `100`.

`trimmean(X,percent,dim)` takes the trimmed mean along dimension `dim` of `X`.

Remarks

The trimmed mean is a robust estimate of the location of a sample. If there are outliers in the data, the trimmed mean is a more representative estimate of the center of the body of the data than the mean. However, if the data is all from the same probability distribution, then the trimmed mean is less efficient than the sample mean as an estimator of the location of the data.

Examples

This example shows a Monte Carlo simulation of the efficiency of the 10% trimmed mean relative to the sample mean for normal data.

```
x = normrnd(0,1,100,100);
m = mean(x);
trim = trimmean(x,10);
sm = std(m);
strim = std(trim);
efficiency = (sm/strim).^2
efficiency =
  0.9702
```

See Also

`mean`, `median`, `geomean`, `harmmean`

Purpose

Random numbers from Student's *t* distribution

Syntax

```
R = trnd(V)
R = trnd(v,m)
R = trnd(V,m,n)
```

Description

`R = trnd(V)` generates random numbers from Student's *t* distribution with `V` degrees of freedom. `V` can be a vector, a matrix, or a multidimensional array. The size of `R` is the size of `V`.

`R = trnd(v,m)` generates random numbers from Student's *t* distribution with `v` degrees of freedom, where `v` is a row vector. If `v` is a 1-by-2 vector, `R` is a matrix with `v(1)` rows and `v(2)` columns. If `v` is 1-by-*n*, `R` is an *n*-dimensional array.

`R = trnd(V,m,n)` generates random numbers from Student's *t* distribution with `V` degrees of freedom, where scalars `m` and `n` are the row and column dimensions of `R`.

Example

```
noisy = trnd(ones(1,6))
noisy =
  19.7250  0.3488  0.2843  0.4034  0.4816  -2.4190

numbers = trnd(1:6,[1 6])
numbers =
  -1.9500  -0.9611  -0.9038  0.0754  0.9820  1.0115

numbers = trnd(3,2,6)
numbers =
 -0.3177 -0.0812 -0.6627  0.1905 -1.5585 -0.0433
  0.2536  0.5502  0.8646  0.8060 -0.5216  0.0891
```

See Also

`tcdf`, `tinv`, `tpdf`, `tstat`

Purpose

Mean and variance of Student's *t* distribution

Syntax

```
[M,V] = tstat(NU)
```

Description

`[M,V] = tstat(NU)` returns the mean of and variance for Student's *t* distribution with parameters specified by `NU`. `M` and `V` are the same size as `NU`.

The mean of the Student's *t* distribution with parameter ν is zero for values of ν greater than 1. If ν is one, the mean does not exist. The variance for values of ν greater than 2 is $\nu/(\nu-2)$.

Examples

Find the mean of and variance for 1 to 30 degrees of freedom.

```
[m,v] = tstat(reshape(1:30,6,5))
m =
  NaN   0    0    0    0
   0    0    0    0    0
   0    0    0    0    0
   0    0    0    0    0
   0    0    0    0    0
   0    0    0    0    0

v =
     NaN    1.4000  1.1818  1.1176  1.0870
     NaN    1.3333  1.1667  1.1111  1.0833
  3.0000    1.2857  1.1538  1.1053  1.0800
  2.0000    1.2500  1.1429  1.1000  1.0769
  1.6667    1.2222  1.1333  1.0952  1.0741
  1.5000    1.2000  1.1250  1.0909  1.0714
```

Note that the variance does not exist for one and two degrees of freedom.

See Also

`tcdf`, `tinv`, `tpdf`, `trnd`

Purpose

One-sample or paired-sample *t*-test

Syntax

```
h = ttest(x)
h = ttest(x,m)
h = ttest(x,y)
h = ttest(...,alpha)
h = ttest(...,alpha,tail)
h = ttest(...,alpha,tail,dim)
[h,p] = ttest(...)
[h,p,ci] = ttest(...)
[h,p,ci,stats] = ttest(...)
```

Description

`h = ttest(x)` performs a *t*-test of the null hypothesis that data in the vector `x` are a random sample from a normal distribution with mean 0 and unknown variance, against the alternative that the mean is not 0. The result of the test is returned in `h`. `h = 1` indicates a rejection of the null hypothesis at the 5% significance level. `h = 0` indicates a failure to reject the null hypothesis at the 5% significance level.

`x` can also be a matrix or an *N*-dimensional array. For matrices, `ttest` performs separate *t*-tests along each column of `x` and returns a vector of results. For *N*-dimensional arrays, `ttest` works along the first non-singleton dimension of `x`.

The test treats `NaN` values as missing data, and ignores them.

`h = ttest(x,m)` performs a *t*-test of the null hypothesis that data in the vector `x` are a random sample from a normal distribution with mean `m` and unknown variance, against the alternative that the mean is not `m`.

`h = ttest(x,y)` performs a paired *t*-test of the null hypothesis that data in the difference `x-y` are a random sample from a normal distribution with mean 0 and unknown variance, against the alternative that the mean is not 0. `x` and `y` must be vectors of the same length, or arrays of the same size.

`h = ttest(...,alpha)` performs the test at the (100*`alpha`)% significance level. The default, when unspecified, is `alpha = 0.05`.

`h = ttest(...,alpha,tail)` performs the test against the alternative specified by the string `tail`. There are three options for `tail`:

- `'both'` — Mean is not 0 (or `m`) (two-tailed test). This is the default, when `tail` is unspecified.
- `'right'` — Mean is greater than 0 (or `m`) (right-tail test)
- `'left'` — Mean is less than 0 (or `m`) (left-tail test)

`tail` must be a single string, even when `x` is a matrix or an *N*-dimensional array.

`h = ttest(...,alpha,tail,dim)` works along dimension `dim` of `x`, or of `x-y` for a paired test. Use `[]` to pass in default values for `m`, `alpha`, or `tail`.

`[h,p] = ttest(...)` returns the *p*-value of the test. The *p*-value is the probability, under the null hypothesis, of observing a value as extreme or more extreme of the test statistic

$$t = \frac{\bar{x} - \mu}{s / \sqrt{n}}$$

where $\bar{x}$ is the sample mean, μ = 0 (or `m`) is the hypothesized population mean, *s* is the sample standard deviation, and *n* is the sample size. Under the null hypothesis, the test statistic will have Student's *t* distribution with $n - 1$ degrees of freedom.

`[h,p,ci] = ttest(...)` returns a 100*(1 – `alpha`)% confidence interval on the population mean, or on the difference of population means for a paired test.

`[h,p,ci,stats] = ttest(...)` returns the structure `stats` with the following fields:

- `tstat` — Value of the test statistic
- `df` — Degrees of freedom of the test
- `sd` — Sample standard deviation

Example

Simulate a random sample of size 100 from a normal distribution with mean 0.1:

```
x = normrnd(0.1,1,1,100);
```

Test the null hypothesis that the sample comes from a normal distribution with mean 0:

```
[h,p,ci] = ttest(x,0)
h =
     0
p =
    0.8323
ci =
   -0.1650    0.2045
```

The test fails to reject the null hypothesis at the default $\alpha = 0.05$ significance level. Under the null hypothesis, the probability of observing a value as extreme or more extreme of the test statistic, as indicated by the *p*-value, is much greater than α. The 95% confidence interval on the mean contains 0.

Simulate a larger random sample of size 1000 from the same distribution:

```
y = normrnd(0.1,1,1,1000);
```

Test again if the sample comes from a normal distribution with mean 0:

```
[h,p,ci] = ttest(y,0)
h =
     1
p =
    0.0160
ci =
    0.0142    0.1379
```

This time the test rejects the null hypothesis at the default $\alpha = 0.05$ significance level. The *p*-value has fallen below $\alpha = 0.05$ and the 95% confidence interval on the mean does not contain 0.

Because the *p*-value of the sample `y` is greater than 0.01, the test will fail to reject the null hypothesis when the significance level is lowered to $\alpha = 0.01$:

```
[h,p,ci] = ttest(y,0,0.01)
h =
     0
p =
   0.0160
ci =
  -0.0053    0.1574
```

Notice that at the lowered significance level the 99% confidence interval on the mean widens to contain 0.

This example will produce slightly different results each time it is run, because of the random sampling.

See Also

`ttest2`, `ztest`

Purpose

Two-sample *t*-test

Syntax

```
h = ttest2(x,y)
h = ttest2(x,y,alpha)
h = ttest2(x,y,alpha,tail)
h = ttest2(x,y,alpha,tail,vartype)
h = ttest(x,y,alpha,tail,vartype,dim)
[h,p] = ttest2(...)
[h,p,ci] = ttest2(...)
[h,p,ci,stats] = ttest2(...)
```

Description

`h = ttest2(x,y)` performs a *t*-test of the null hypothesis that data in the vectors `x` and `y` are independent random samples from normal distributions with equal means and equal but unknown variances, against the alternative that the means are not equal. The result of the test is returned in `h`. `h = 1` indicates a rejection of the null hypothesis at the 5% significance level. `h = 0` indicates a failure to reject the null hypothesis at the 5% significance level. `x` and `y` need not be vectors of the same length.

`x` and `y` can also be matrices or *N*-dimensional arrays. Matrices `x` and `y` must have the same number of columns, in which case `ttest2` performs separate *t*-tests along each column and returns a vector of results. *N*-dimensional arrays `x` and `y` must have the same size along all but the first non-singleton dimension, in which case `ttest2` works along the first non-singleton dimension.

The test treats `NaN` values as missing data, and ignores them.

`h = ttest2(x,y,alpha)` performs the test at the (100*`alpha`)% significance level. The default, when unspecified, is `alpha = 0.05`.

`h = ttest2(x,y,alpha,tail)` performs the test against the alternative specified by the string `tail`. There are three options for `tail`:

- `'both'` — Means are not equal (two-tailed test). This is the default, when `tail` is unspecified.
- `'right'` — Mean of `x` is greater than mean of `y` (right-tail test)

- `'left'` — Mean of `x` is less than mean of `y` (left-tail test)

`tail` must be a single string, even when `x` is a matrix or an *N*-dimensional array.

`h = ttest2(x,y,alpha,tail,`*`vartype`*`)` performs the test under the assumption of equal or unequal population variances, as specified by the string *`vartype`*. There are two options for *`vartype`*:

- `'equal'` — Assumes equal variances. This is the default, when `vartype` is unspecified.
- `'unequal'` — Does not assume equal variances. This is the Behrens-Fisher problem.

`vartype` must be a single string, even when `x` is a matrix or an *N*-dimensional array.

If `vartype` is `'equal'`, the test computes a pooled sample standard deviation using

$$s = \sqrt{\frac{(n-1)s_x^2 + (m-1)s_y^2}{n+m-2}}$$

where s_x and s_y are the sample standard deviations of `x` and `y`, respectively, and n and m are the sample sizes of `x` and `y`, respectively.

`h = ttest(x,y,alpha,tail,vartype,dim)` works along dimension `dim` of `x` and `y`. Use `[]` to pass in default values for `alpha`, `tail`, or `vartype`.

`[h,p] = ttest2(...)` returns the *p*-value of the test. The *p*-value is the probability, under the null hypothesis, of observing a value as extreme or more extreme of the test statistic

$$t = \frac{\bar{x} - \bar{y}}{\sqrt{\frac{s_x^2}{n} + \frac{s_y^2}{m}}}$$

where $\bar{x}$ and $\bar{y}$ are the sample means, s_x and s_y are the sample standard deviations (replaced by the pooled standard deviation s in the default case where `vartype` is `'equal'`), and n and m are the sample sizes.

In the default case where `vartype` is `'equal'`, the test statistic, under the null hypothesis, has Student's t distribution with $n + m - 2$ degrees of freedom.

In the case where `vartype` is `'unequal'`, the test statistic, under the null hypothesis, has an approximate Student's t distribution with a number of degrees of freedom given by Satterthwaite's approximation.

`[h,p,ci] = ttest2(...)` returns a 100*(1 – `alpha`)% confidence interval on the difference of population means.

`[h,p,ci,stats] = ttest2(...)` returns structure `stats` with the following fields:

- `tstat` — Value of the test statistic
- `df` — Degrees of freedom of the test
- `sd` — Pooled sample standard deviation (in the default case where `vartype` is `'equal'`) or a vector with the sample standard deviations (in the case where `vartype` is `'unequal'`).

Example

Simulate random samples of size 1000 from normal distributions with means 0 and 0.1, respectively, and standard deviations 1 and 2, respectively:

```
x = normrnd(0,1,1,1000);
y = normrnd(0.1,2,1,1000);
```

Test the null hypothesis that the samples come from populations with equal means, against the alternative that the means are unequal. Perform the test assuming unequal variances:

```
[h,p,ci] = ttest2(x,y,[],[],'unequal')
h =
     1
```

```
p =
    0.0102
ci =
   -0.3227   -0.0435
```

The test rejects the null hypothesis at the default $\alpha = 0.05$ significance level. Under the null hypothesis, the probability of observing a value as extreme or more extreme of the test statistic, as indicated by the p-value, is less than α. The 95% confidence interval on the mean of the difference does not contain 0.

This example will produce slightly different results each time it is run, because of the random sampling.

See Also

`ttest`, `ztest`

type

Purpose

Type of tree

Syntax

```
ttype = type(t)
```

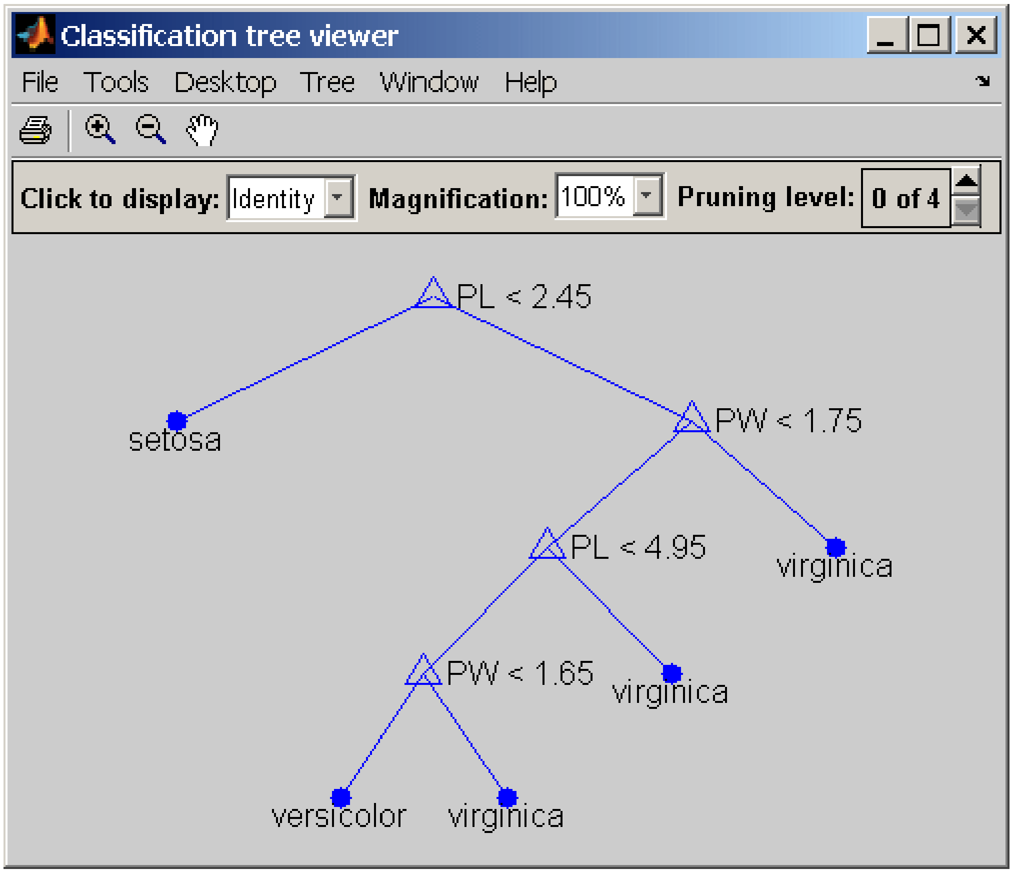
Description

`ttype = type(t)` returns the type of the tree `t`. `ttype` is `'regression'` for regression trees and `'classification'` for classification trees.

Example

Create a classification tree for Fisher's iris data:

```
load fisheriris;

t = classregtree(meas,species,'names',{'SL' 'SW' 'PL' 'PW'})
t =
Decision tree for classification
1  if PL<2.45 then node 2 else node 3
2  class = setosa
3  if PW<1.75 then node 4 else node 5
4  if PL<4.95 then node 6 else node 7
5  class = virginica
6  if PW<1.65 then node 8 else node 9
7  class = virginica
8  class = versicolor
9  class = virginica

view(t)
```



```
ttype = type(t)
ttype =
classification
```

Reference

[1] Breiman, L., et al., *Classification and Regression Trees*, Chapman & Hall, Boca Raton, 1993.

See Also

`classregtree`

unidcdf

Purpose

Discrete uniform cumulative distribution function

Syntax

```
P = unidcdf(X,N)
```

Description

`P = unidcdf(X,N)` computes the discrete uniform cdf at each of the values in `X` using the corresponding parameters in `N`. `X` and `N` can be vectors, matrices, or multidimensional arrays that have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs. The maximum observable values in `N` must be positive integers.

The discrete uniform cdf is

$$p = F(x|N) = \frac{floor(x)}{N} I_{(1, \ldots, N)}(x)$$

The result, p, is the probability that a single observation from the discrete uniform distribution with maximum N will be a positive integer less than or equal to x. The values x do not need to be integers.

Examples

What is the probability of drawing a number 20 or less from a hat with the numbers from 1 to 50 inside?

```
probability = unidcdf(20,50)
probability =
  0.4000
```

See Also

`cdf`, `unidinv`, `unidpdf`, `unidrnd`, `unidstat`

Purpose

Inverse of discrete uniform cumulative distribution function

Syntax

```
X = unidinv(P,N)
```

Description

`X = unidinv(P,N)` returns the smallest positive integer `X` such that the discrete uniform cdf evaluated at `X` is equal to or exceeds `P`. You can think of `P` as the probability of drawing a number as large as `X` out of a hat with the numbers `1` through `N` inside.

`P` and `N` can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of `X`. A scalar input for `N` or `P` is expanded to a constant array with the same dimensions as the other input. The values in `P` must lie on the interval [0 1] and the values in `N` must be positive integers.

Examples

```
x = unidinv(0.7,20)
x =
    14

y = unidinv(0.7 + eps,20)
y =
    15
```

A small change in the first parameter produces a large jump in output. The cdf and its inverse are both step functions. The example shows what happens at a step.

See Also

`icdf`, `unidcdf`, `unidpdf`, `unidrnd`, `unidstat`

Purpose

Discrete uniform probability density function

Syntax

```
Y = unidpdf(X,N)
```

Description

`Y = unidpdf(X,N)` computes the discrete uniform pdf at each of the values in `X` using the corresponding parameters in `N`. `X` and `N` can be vectors, matrices, or multidimensional arrays that have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs. The parameters in `N` must be positive integers.

The discrete uniform pdf is

$$y = f(x|N) = \frac{1}{N} I_{(1,\ldots,N)}(x)$$

You can think of y as the probability of observing any one number between 1 and n.

Examples

For fixed `n`, the uniform discrete pdf is a constant.

```
y = unidpdf(1:6,10)
y =
  0.1000  0.1000  0.1000  0.1000  0.1000  0.1000
```

Now fix `x`, and vary `n`.

```
likelihood = unidpdf(5,4:9)
likelihood =
  0  0.2000  0.1667  0.1429  0.1250  0.1111
```

See Also

`pdf`, `unidcdf`, `unidinv`, `unidrnd`, `unidstat`

Purpose

Random numbers from discrete uniform distribution

Syntax

```
R = unidrnd(N)
R = unidrnd(N,v)
R = unidrnd(N,m,n)
```

Description

The discrete uniform distribution arises from experiments equivalent to drawing a number from one to `N` out of a hat.

`R = unidrnd(N)` generates random numbers for the discrete uniform distribution with maximum `N`. The parameters in `N` must be positive integers. `N` can be a vector, a matrix, or a multidimensional array. The size of `R` is the size of `N`.

`R = unidrnd(N,v)` generates random numbers for the discrete uniform distribution with maximum `N`, where `v` is a row vector. If v is a 1-by-2 vector, `R` is a matrix with `v(1)` rows and `v(2)` columns. If `v` is 1-by-n, `R` is an n-dimensional array.

`R = unidrnd(N,m,n)` generates random numbers for the discrete uniform distribution with maximum `N`, where scalars m and n are the row and column dimensions of `R`.

Example

In the Massachusetts lottery, a player chooses a four-digit number. Generate random numbers for Monday through Saturday.

```
numbers = unidrnd(10000,1,6)-1
numbers =
    4564  185  8214  4447  6154  7919
```

See Also

`unidcdf`, `unidinv`, `unidpdf`, `unidstat`

Purpose

Mean of and variance for discrete uniform distribution

Syntax

```
[M,V] = unidstat(N)
```

Description

`[M,V] = unidstat(N)` returns the mean of and variance for the discrete uniform distribution with parameter `N`.

The mean of the discrete uniform distribution with parameter N is $(N+1)/2$. The variance is $(N^2-1)/12$.

Examples

```
[m,v] = unidstat(1:6)
m =
  1.0000  1.5000  2.0000  2.5000  3.0000  3.5000
v =
  0  0.2500  0.6667  1.2500  2.0000  2.9167
```

See Also

`unidcdf`, `unidinv`, `unidpdf`, `unidrnd`

Purpose

Continuous uniform cumulative distribution function

Syntax

```
P = unifcdf(X,A,B)
```

Description

`P = unifcdf(X,A,B)` computes the uniform cdf at each of the values in `X` using the corresponding parameters in `A` and `B` (the minimum and maximum values, respectively). `X`, `A`, and `B` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant matrix with the same dimensions as the other inputs.

The uniform cdf is

$$p = F(x|a,b) = \frac{x-a}{b-a} I_{[a,\,b]}(x)$$

The standard uniform distribution has A = `0` and B = `1`.

Examples

What is the probability that an observation from a standard uniform distribution will be less than 0.75?

```
probability = unifcdf(0.75)
probability =
  0.7500
```

What is the probability that an observation from a uniform distribution with `a = -1` and `b = 1` will be less than 0.75?

```
probability = unifcdf(0.75,-1,1)
probability =
  0.8750
```

See Also

`cdf`, `unifinv`, `unifit`, `unifpdf`, `unifrnd`, `unifstat`

unifinv

Purpose

Inverse of continuous uniform cumulative distribution function

Syntax

```
X = unifinv(P,A,B)
```

Description

`X = unifinv(P,A,B)` computes the inverse of the uniform cdf with parameters `A` and `B` (the minimum and maximum values, respectively) at the corresponding probabilities in `P`. `P`, `A`, and `B` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs.

The inverse of the uniform cdf is

$$x = F^{-1}(p|a,b) = a + p(a-b)I_{[0,1]}(p)$$

The standard uniform distribution has A = 0 and B = 1.

Examples

What is the median of the standard uniform distribution?

```
median_value = unifinv(0.5)
median_value =
  0.5000
```

What is the 99th percentile of the uniform distribution between -1 and 1?

```
percentile = unifinv(0.99,-1,1)
percentile =
  0.9800
```

See Also

`icdf`, `unifcdf`, `unifit`, `unifpdf`, `unifrnd`, `unifstat`

Purpose

Parameter estimates for uniformly distributed data

Syntax

```
[ahat,bhat] = unifit(data)
[ahat,bhat,ACI,BCI] = unifit(data)
[ahat,bhat,ACI,BCI] = unifit(data,alpha)
```

Description

`[ahat,bhat] = unifit(data)` returns the maximum likelihood estimates (MLEs) of the parameters of the uniform distribution given the data in `data`.

`[ahat,bhat,ACI,BCI] = unifit(data)` also returns 95% confidence intervals, `ACI` and `BCI`, which are matrices with two rows. The first row contains the lower bound of the interval for each column of the matrix `data`. The second row contains the upper bound of the interval.

`[ahat,bhat,ACI,BCI] = unifit(data,alpha)` enables you to control of the confidence level `alpha`. For example, if `alpha = 0.01` then `ACI` and `BCI` are 99% confidence intervals.

Example

```
r = unifrnd(10,12,100,2);
[ahat,bhat,aci,bci] = unifit(r)
ahat =
  10.0154  10.0060
bhat =
  11.9989  11.9743
aci =
  9.9551  9.9461
  10.0154  10.0060
bci =
  11.9989  11.9743
  12.0592  12.0341
```

See Also

`betafit`, `binofit`, `expfit`, `gamfit`, `normfit`, `poissfit`, `unifcdf`, `unifinv`, `unifpdf`, `unifrnd`, `unifstat`, `wblfit`

unifpdf

Purpose

Continuous uniform probability density function

Syntax

```
Y = unifpdf(X,A,B)
```

Description

`Y = unifpdf(X,A,B)` computes the continuous uniform pdf at each of the values in `X` using the corresponding parameters in `A` and `B`. `X`, `A`, and `B` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs. The parameters in `B` must be greater than those in `A`.

The continuous uniform distribution pdf is

$$y = f(x|a,b) = \frac{1}{b-a} I_{[a,b]}(x)$$

The standard uniform distribution has A = `0` and B = `1`.

Examples

For fixed `a` and `b`, the uniform pdf is constant.

```
x = 0.1:0.1:0.6;
y = unifpdf(x)
y =
   1    1    1    1    1    1
```

What if `x` is not between `a` and `b`?

```
y = unifpdf(-1,0,1)
y =
   0
```

See Also

`pdf`, `unifcdf`, `unifinv`, `unifrnd`, `unifstat`

Purpose Random numbers from continuous uniform distribution

Syntax

```
R = unifrnd(A,B)
R = unifrnd(A,B,m)
R = unifrnd(A,B,m,n)
```

Description

`R = unifrnd(A,B)` generates uniform random numbers with parameters `A` and `B`. Vector or matrix inputs for `A` and `B` must have the same size, which is also the size of `R`. A scalar input for `A` or `B` is expanded to a constant matrix with the same dimensions as the other input.

`R = unifrnd(A,B,m)` generates uniform random numbers with parameters `A` and `B`, where `m` is a 1-by-2 vector that contains the row and column dimensions of `R`.

`R = unifrnd(A,B,m,n)` generates uniform random numbers with parameters `A` and `B`, where scalars `m` and `n` are the row and column dimensions of `R`.

Examples

```
random = unifrnd(0,1:6)
random =
  0.2190  0.0941  2.0366  2.7172  4.6735  2.3010

random = unifrnd(0,1:6,[1 6])
random =
  0.5194  1.6619  0.1037  0.2138  2.6485  4.0269

random = unifrnd(0,1,2,3)
random =
  0.0077  0.0668  0.6868
  0.3834  0.4175  0.5890
```

See Also `unifcdf`, `unifinv`, `unifpdf`, `unifstat`

unifstat

Purpose

Mean and variance of continuous uniform distribution

Syntax

```
[M,V] = unifstat(A,B)
```

Description

`[M,V] = unifstat(A,B)` returns the mean of and variance for the continuous uniform distribution with parameters specified by `A` and `B`. Vector or matrix inputs for `A` and `B` must have the same size, which is also the size of `M` and `V`. A scalar input for `A` or `B` is expanded to a constant matrix with the same dimensions as the other input.

The mean of the continuous uniform distribution with parameters a and b is $(a+b)/2$, and the variance is $(b-a)^2/12$.

Examples

```
a = 1:6;
b = 2.*a;
[m,v] = unifstat(a,b)
m =
  1.5000  3.0000  4.5000  6.0000  7.5000  9.0000
v =
  0.0833  0.3333  0.7500  1.3333  2.0833  3.0000
```

See Also

`unifcdf`, `unifinv`, `unifpdf`, `unifrnd`

Purpose

Parameters of generalized Pareto distribution upper tail

Syntax

```
params = upperparams(obj)
```

Description

`params = upperparams(obj)` returns the 2-element vector `params` of shape and scale parameters, respectively, of the upper tail of the Pareto tails object `obj`. `upperparams` does not return a location parameter.

Example

Fit Pareto tails to a *t* distribution at cumulative probabilities 0.1 and 0.9:

```
t = trnd(3,100,1);
obj = paretotails(t,0.1,0.9);

lowerparams(obj)
ans =
   -0.1901    1.1898
upperparams(obj)
ans =
    0.3646    0.5103
```

See Also

`paretotails`, `lowerparams`

Purpose

Variance of sample

Syntax

```
y = var(X)
y = var(x)
y = var(x,1)
y = var(X,w)
var(X,w,dim)
```

Description

`y = var(X)` computes the variance of the data in `X`. For vectors, `var(x)` is the variance of the elements in `x`. For matrices, `var(X)` is a row vector containing the variance of each column of `X`.

`y = var(x)` normalizes by $n - 1$ where n is the sequence length. For normally distributed data, this makes `var(x)` the minimum variance unbiased estimator MVUE of σ^2(the second parameter).

`y = var(x,1)` normalizes by n and yields the second moment of the sample data about its mean (moment of inertia).

`y = var(X,w)` computes the variance using the vector of positive weights `w`. The number of elements in `w` must equal the number of rows in the matrix `X`. For vector `x`, `w` and `x` must match in length.

`var(X,w,dim)` takes the variance along the dimension `dim` of `X`. Pass in `0` for `w` to use the default normalization by $n - 1$, or 1 to use n.

`var` supports both common definitions of variance. Let SS be the sum of the squared deviations of the elements of a vector `x` from their mean. Then, `var(x)` = $SS/(n - 1)$ is the MVUE, and `var(x,1)` = SS/n is the maximum likelihood estimator (MLE) of σ^2.

Examples

```
x = [-1 1];
w = [1 3];

v1 = var(x)
v1 =
   2

v2 = var(x,1)
```

```
v2 =
   1

v3 = var(x,w)
v3 =
  0.7500
```

See Also cov, std

vartest

Purpose

One-sample chi-square variance test

Syntax

```
H = vartest(X,V)
H = vartest(X,V,alpha)
H = vartest(X,V,alpha,tail)
[H,P] = vartest(...)
[H,P,CI] = vartest(...)
[H,P,CI,STATS] = vartest(...)
[...] = vartest(X,V,alpha,tail,dim)
```

Description

`H = vartest(X,V)` performs a chi-square test of the hypothesis that the data in the vector `X` comes from a normal distribution with variance `V`, against the alternative that `X` comes from a normal distribution with a different variance. The result is `H = 0` if the null hypothesis (variance is `V`) cannot be rejected at the 5% significance level, or `H = 1` if the null hypothesis can be rejected at the 5% level.

`X` may also be a matrix or an n-dimensional array. For matrices, `vartest` performs separate tests along each column of `X`, and returns a row vector of results. For n-dimensional arrays, `vartest` works along the first nonsingleton dimension of `X`. `V` must be a scalar.

`H = vartest(X,V,alpha)` performs the test at the significance level `(100*alpha)`%. `alpha` has a default value of 0.05 and must be a scalar.

`H = vartest(X,V,alpha,`*`tail`*`)` performs the test against the alternative hypothesis specified by *`tail`*, where *`tail`* is a single string from the following choices:

- `'both'` — Variance is not `V` (two-tailed test). This is the default.
- `'right'` — Variance is greater than `V` (right-tailed test).
- `'left'` — Variance is less than `V` (left-tailed test).

`[H,P] = vartest(...)` returns the p-value, i.e., the probability of observing the given result, or one more extreme, by chance if the null hypothesis is true. Small values of `P` cast doubt on the validity of the null hypothesis.

`[H,P,CI] = vartest(...)` returns a `100*(1-alpha)`% confidence interval for the true variance.

`[H,P,CI,STATS] = vartest(...)` returns the structure `STATS` with the following fields:

- `'chisqstat'` — Value of the test statistic
- `'df'` — Degrees of freedom of the test

`[...]  = vartest(X,V,alpha,`*`tail`*`,dim)` works along dimension `dim` of `X`. Pass in `[]` for `alpha` or `tail` to use their default values.

Example

Determine whether the standard deviation is significantly different from 7?

```
load carsmall
```

```
[h,p,ci] = vartest(MPG,7^2)
```

See Also

`ttest`, `ztest`,`vartest2`

vartest2

Purpose

Two-sample *F*-test for equal variances

Syntax

```
H = vartest2(X,Y)
H = vartest2(X,Y,alpha)
H = vartest2(X,Y,alpha,tail)
[H,P] = vartest2(...)
[H,P,CI] = vartest2(...)
[H,P,CI,STATS] = vartest2(...)
[...] = vartest2(X,Y,alpha,tail,dim)
```

Description

`H = vartest2(X,Y)` performs an *F* test of the hypothesis that two independent samples, in the vectors `X` and `Y`, come from normal distributions with the same variance, against the alternative that they come from normal distributions with different variances. The result is `H = 0` if the null hypothesis (variances are equal) cannot be rejected at the 5% significance level, or `H = 1` if the null hypothesis can be rejected at the 5% level. `X` and `Y` can have different lengths. `X` and `Y` can also be matrices or n-dimensional arrays.

For matrices, `vartest2` performs separate tests along each column, and returns a vector of results. `X` and `Y` must have the same number of columns. For n-dimensional arrays, `vartest2` works along the first nonsingleton dimension. `X` and `Y` must have the same size along all the remaining dimensions.

`H = vartest2(X,Y,alpha)` performs the test at the significance level `(100*alpha)`%. `alpha` must be a scalar.

`H = vartest2(X,Y,alpha,`*`tail`*`)` performs the test against the alternative hypothesis specified by *`tail`*, where *`tail`* is one of the following single strings:

- `'both'` — Variance is not `Y` (two-tailed test). This is the default.
- `'right'` — Variance is greater than `Y` (right-tailed test).
- `'left'` — Variance is less than `Y` (left-tailed test).

`[H,P] = vartest2(...)` returns the p-value, i.e., the probability of observing the given result, or one more extreme, by chance if the null hypothesis is true. Small values of `P` cast doubt on the validity of the null hypothesis.

`[H,P,CI] = vartest2(...)` returns a `100*(1-alpha)`% confidence interval for the true variance ratio `var(X)/var(Y)`.

`[H,P,CI,STATS] = vartest2(...)` returns a structure with the following fields:

- `'fstat'` — Value of the test statistic
- `'df1'` — Numerator degrees of freedom of the test
- `'df2'` — Denominator degrees of freedom of the test

`[...] = vartest2(X,Y,alpha,`*`tail`*`,dim)` works along dimension `dim` of `X`. To pass in the default values for `alpha` or `tail` use `[]`.

Example

Is the variance significantly different for two model years, and what is a confidence interval for the ratio of these variances?

```
load carsmall
```

```
[H,P,CI] =
vartest2(MPG(Model_Year==82),MPG(Model_Year==76))
```

See Also

`ansaribradley`, `vartest`, `vartestn`, `ttest2`

Purpose

Bartlett multiple-sample test for equal variances

Syntax

```
vartestn(X)
vartestn(X,group)
P = vartestn(...)
[P,STATS] = vartestn(...)
[...] = vartestn(...,displayopt)
[...] = vartestn(...,testtype)
```

Description

`vartestn(X)` performs Bartlett's test for equal variances for the columns of the matrix `X`. This is a test of the null hypothesis that the columns of `X` come from normal distributions with the same variance, against the alternative that they come from normal distributions with different variances. The result is a display of a box plot of the groups, and a summary table of statistics.

`vartestn(X,group)` requires a vector `X`, and a `group` argument that is a categorical variable, vector, string array, or cell array of strings with one row for each element of `X`. The `X` values corresponding to the same value of `group` are placed in the same group. (See "Grouped Data" on page 2-41.) The function tests for equal variances across groups.

`vartestn` treats `NaN`s as missing values and ignores them.

`P = vartestn(...)` returns the *p*-value, i.e., the probability of observing the given result, or one more extreme, by chance if the null hypothesis of equal variances is true. Small values of `P` cast doubt on the validity of the null hypothesis.

`[P,STATS] = vartestn(...)` returns a structure with the following fields:

- `'chistat'` — Value of the test statistic
- `'df'` — Degrees of freedom of the test

`[...] = vartestn(...,`*`displayopt`*`)` determines if a box plot and table are displayed. *`displayopt`* may be `'on'` (the default) or `'off'` .

`[...] = vartestn(...,testtype)` sets the test type. When *`testtype`* is `'robust'`, `vartestn` performs Levene's test in place of Bartlett's test, which is a useful alternative when the sample distributions are not normal, and especially when they are prone to outliers. For this test the `STATS` output structure has a field named `'fstat'` containing the test statistic, and `'df1'` and `'df2'` containing its numerator and denominator degrees of freedom. When *`testtype`* is `'classical'` `vartestn` performs Bartlett's test.

Example

Does the variance of mileage measurements differ significantly from one model year to another?

```
load carsmall
```

```
vartestn(MPG,Model_Year)
```

See Also

`vartest`, `vartest2`, `anova1`

Purpose

View tree

Syntax

```
view(t)
view(t,param1,val1,param2,val2,...)
```

Description

`view(t)` displays the decision tree `t` as computed by `classregtree` in a figure window. Each branch in the tree is labeled with its decision rule, and each terminal node is labeled with the predicted value for that node. Click any node to get more information about it. The information displayed is specified by the **Click to display** pop-up menu at the top of the figure.

`view(t,param1,val1,param2,val2,...)` specifies optional parameter name/value pairs:

- `'names'` — A cell array of names for the predictor variables, in the order in which they appear in the matrix `X` from which the tree was created. (See `classregtree`.)
- `'prunelevel'` — Initial pruning level to display.

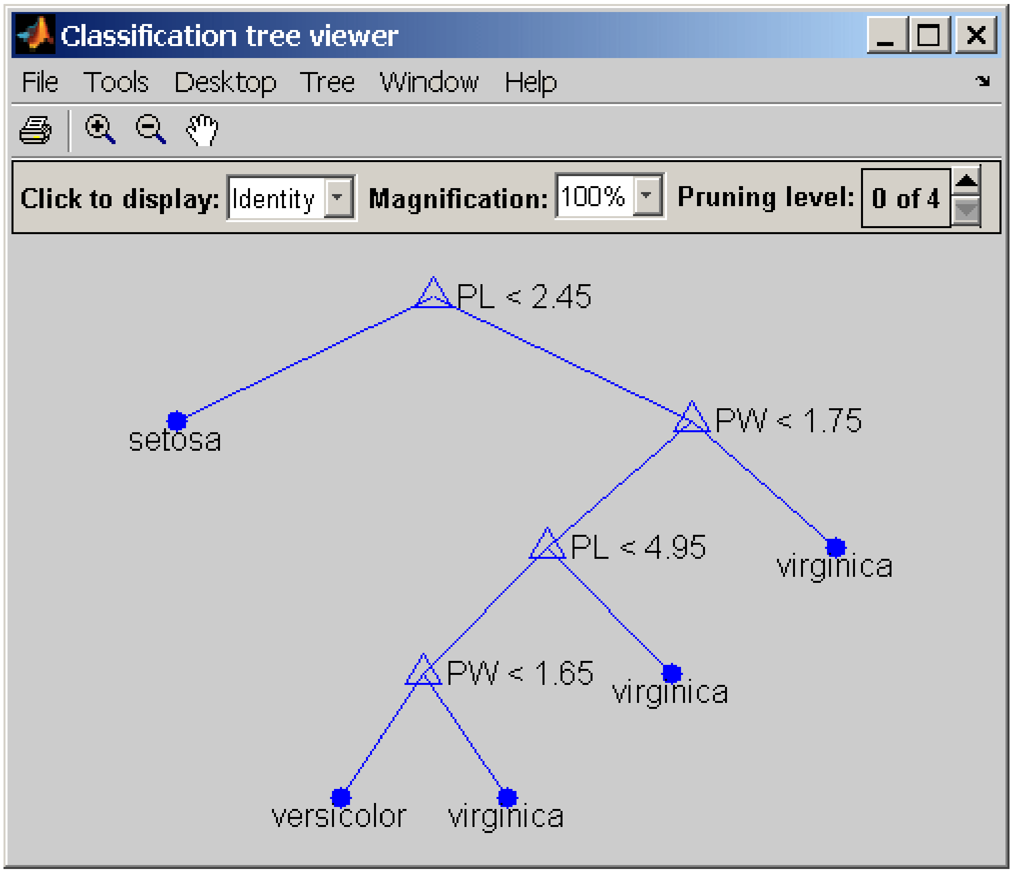
For each branch node, the left child node corresponds to the points that satisfy the condition, and the right child node corresponds to the points that do not satisfy the condition.

Example

Create a classification tree for Fisher's iris data:

```
load fisheriris;

t = classregtree(meas,species,'names',{'SL' 'SW' 'PL' 'PW'})
t =
Decision tree for classification
1  if PL<2.45 then node 2 else node 3
2  class = setosa
3  if PW<1.75 then node 4 else node 5
4  if PL<4.95 then node 6 else node 7
5  class = virginica
6  if PW<1.65 then node 8 else node 9
```

```
7  class = virginica
8  class = versicolor
9  class = virginica

view(t)
```



Reference [1] Breiman, L., et al., *Classification and Regression Trees*, Chapman & Hall, Boca Raton, 1993.

See Also `classregtree`, `eval`, `test`, `prune`

Purpose

Weibull cumulative distribution function

Syntax

```
P = wblcdf(X,A,B)
[P,PLO,PUP] = wblcdf(X,A,B,PCOV,alpha)
```

Description

`P = wblcdf(X,A,B)` computes the cdf of the Weibull distribution with scale parameter `A` and shape parameter `B`, at each of the values in `X`. `X`, `A`, and `B` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array of the same size as the other inputs. The default values for `A` and `B` are both `1`. The parameters `A` and `B` must be positive.

`[P,PLO,PUP] = wblcdf(X,A,B,PCOV,alpha)` returns confidence bounds for `P` when the input parameters `A` and `B` are estimates. `PCOV` is the 2-by-2 covariance matrix of the estimated parameters. `alpha` has a default value of 0.05, and specifies 100(1 - `alpha`)% confidence bounds. `PLO` and `PUP` are arrays of the same size as `P` containing the lower and upper confidence bounds.

The function `wblcdf` computes confidence bounds for `P` using a normal approximation to the distribution of the estimate

$$\hat{b}(\log x - \log \hat{a})$$

and then transforms those bounds to the scale of the output `P`. The computed bounds give approximately the desired confidence level when you estimate `mu`, `sigma`, and `PCOV` from large samples, but in smaller samples other methods of computing the confidence bounds might be more accurate.

The Weibull cdf is

$$p = F(x|a,b) = \int_0^x b a^{-b} t^{b-1} e^{-\left(\frac{t}{a}\right)^b} dt = 1 - e^{-\left(\frac{x}{a}\right)^b} I_{(0,\infty)}(x)$$

Examples

What is the probability that a value from a Weibull distribution with parameters `a` = `0.15` and `b` = `0.8` is less than 0.5?

```
probability = wblcdf(0.5, 0.15, 0.8)
probability =
  0.9272
```

How sensitive is this result to small changes in the parameters?

```
[A, B] = meshgrid(0.1:0.05:0.2,0.2:0.05:0.3);
probability = wblcdf(0.5, A, B)
probability =
  0.7484  0.7198  0.6991
  0.7758  0.7411  0.7156
  0.8022  0.7619  0.7319
```

See Also

`cdf`, `wblfit`, `wblinv`, `wbllike`, `wblpdf`, `wblplot`, `wblrnd`, `wblstat`

Purpose

Parameter estimates and confidence intervals for Weibull distributed data

Syntax

```
parmhat = wblfit(data)
[parmhat,parmci] = wblfit(data)
parmhat,parmci] = wblfit(data,alpha)
[...] = wblfit(data,alpha,censoring)
[...] = wblfit(data,alpha,censoring,freq)
[...] = wblfit(...,options)
```

Description

`parmhat = wblfit(data)` returns the maximum likelihood estimates, `parmhat`, of the parameters of the Weibull distribution given the values in the vector `data`, which must be positive. `parmhat` is a two-element row vector: `parmhat(1)` estimates the Weibull parameter a, and `parmhat(2)` estimates the Weibull parameter b, in the pdf

$$y = f(x|a,b) = b a^{-b} x^{b-1} e^{-\left(\frac{x}{a}\right)^b} I_{(0,\infty)}(x)$$

`[parmhat,parmci] = wblfit(data)` returns 95% confidence intervals for the estimates of a and b in the 2-by-2 matrix `parmci`. The first row contains the lower bounds of the confidence intervals for the parameters, and the second row contains the upper bounds of the confidence intervals.

`[parmhat,parmci] = wblfit(data,alpha)` returns 100(1 - `alpha`)% confidence intervals for the parameter estimates.

`[...]  = wblfit(data,alpha,censoring)` accepts a Boolean vector, `censoring`, of the same size as `data`, which is `1` for observations that are right-censored and `0` for observations that are observed exactly.

`[...]  = wblfit(data,alpha,censoring,freq)` accepts a frequency vector, `freq`, of the same size as `data`. The vector `freq` typically contains integer frequencies for the corresponding elements in `data`, but can contain any non-negative values. Pass in `[]` for `alpha`, `censoring`, or `freq` to use their default values.

`[...] = wblfit(...,options)` accepts a structure, `options`, that specifies control parameters for the iterative algorithm the function uses to compute maximum likelihood estimates. The Weibull fit function accepts an `options` structure that can be created using the function `statset`. Enter `statset ('wblfit')` to see the names and default values of the parameters that `lognfit` accepts in the `options` structure. See the reference page for `statset` for more information about these options.

Example

```
data = wblrnd(0.5,0.8,100,1);
[parmhat, parmci] = wblfit(data)
parmhat =
  0.5861  0.8567
parmci =
  0.4606  0.7360
  0.7459  0.9973
```

See Also

`wblcdf`, `wblinv`, `wbllike`, `wblpdf`, `wblrnd`, `wblstat`, `mle`, `statset`

Purpose

Inverse of Weibull cumulative distribution function

Syntax

```
X = wblinv(P,A,B)
[X,XLO,XUP] = wblinv(P,A,B,PCOV,alpha)
```

Description

`X = wblinv(P,A,B)` returns the inverse cumulative distribution function (cdf) for a Weibull distribution with scale parameter `A` and shape parameter `B`, evaluated at the values in `P`. `P`, `A`, and `B` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array of the same size as the other inputs. The default values for `A` and `B` are both `1`.

`[X,XLO,XUP] = wblinv(P,A,B,PCOV,alpha)` returns confidence bounds for `X` when the input parameters `A` and `B` are estimates. `PCOV` is a 2-by-2 matrix containing the covariance matrix of the estimated parameters. `alpha` has a default value of 0.05, and specifies 100(1 - `alpha`)% confidence bounds. `XLO` and `XUP` are arrays of the same size as `X` containing the lower and upper confidence bounds.

The function `wblinv` computes confidence bounds for `X` using a normal approximation to the distribution of the estimate

$$\log\hat{a} - \frac{\log q}{\hat{b}}$$

where q is the `P`th quantile from a Weibull distribution with scale and shape parameters both equal to 1. The computed bounds give approximately the desired confidence level when you estimate `mu`, `sigma`, and `PCOV` from large samples, but in smaller samples other methods of computing the confidence bounds might be more accurate.

The inverse of the Weibull cdf is

$$x = F^{-1}(p|a,b) = \left[a\ln\left(\frac{1}{1-p}\right)\right]^{\frac{1}{b}} I_{[0,1]}(p)$$

Examples

The lifetimes (in hours) of a batch of light bulbs has a Weibull distribution with parameters a = 200 and b = 6. What is the median lifetime of the bulbs?

```
life = wblinv(0.5, 200, 6)
life =
 188.1486
```

What is the 90th percentile?

```
life = wblinv(0.9, 200, 6)
life =
  229.8261
```

See Also

`wblcdf`, `wblfit`, `wbllike`, `wblpdf`, `wblrnd`, `wblstat`, `icdf`

Purpose

Negative log-likelihood for Weibull distribution

Syntax

```
nlogL = wbllike(params,data)
[logL,AVAR] = wbllike(params,data)
[...] = wbllike(params,data,censoring)
[...] = wbllike(params,data,censoring,freq)
```

Description

`nlogL = wbllike(params,data)` returns the Weibull log-likelihood with parameters `params(1)` = a and `params(2)` = b given the `data` x_i.

`[logL,AVAR] = wbllike(params,data)` also returns `AVAR`, which is the asymptotic variance-covariance matrix of the parameter estimates if the values in `params` are the maximum likelihood estimates. `AVAR` is the inverse of Fisher's information matrix. The diagonal elements of `AVAR` are the asymptotic variances of their respective parameters.

`[...] = wbllike(params,data,censoring)` accepts a Boolean vector, `censoring`, of the same size as `data`, which is 1 for observations that are right-censored and 0 for observations that are observed exactly.

`[...] = wbllike(params,data,censoring,freq)` accepts a frequency vector, `freq`, of the same size as `data`. `freq` typically contains integer frequencies for the corresponding elements in `data`, but can contain any nonnegative values. Pass in `[]` for `censoring` to use its default value.

The Weibull negative log-likelihood for uncensored data is

$$(-\log L) = -\log \prod_{i=1} f(a,b|x_i) = -\sum_{i=1}^{n} \log f(a,b|x_i)$$

where f is the Weibull pdf.

`wbllike` is a utility function for maximum likelihood estimation.

Example

This example continues the example from `wblfit`.

```
r = wblrnd(0.5,0.8,100,1);
[logL, AVAR] = wbllike(wblfit(r),r)
```

```
logL =
  47.3349
AVAR =
  0.0048  0.0014
  0.0014  0.0040
```

Reference

[1] Patel, J. K., C. H. Kapadia, and D. B. Owen, *Handbook of Statistical Distributions,* Marcel-Dekker, 1976.

See Also

`betalike`, `gamlike`, `mle`, `normlike`, `wblcdf`, `wblfit`, `wblinv`, `wblpdf`, `wblrnd`, `wblstat`

Purpose

Weibull probability density function

Syntax

```
Y = wblpdf(X,A,B)
```

Description

`Y = wblpdf(X,A,B)` computes the Weibull pdf at each of the values in `X` using the corresponding parameters in `A` and `B`. `X`, `A`, and `B` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array of the same size as the other inputs. The parameters in `A` and `B` must be positive.

The Weibull pdf is

$$= f(x|a,b) = ba^{-b}x^{b-1}e^{-\left(\frac{x}{a}\right)^b} I_{(0,\infty)}(x)$$

Some references refer to the Weibull distribution with a single parameter. This corresponds to `wblpdf` with `A = 1`.

Examples

The exponential distribution is a special case of the Weibull distribution.

```
lambda = 1:6;
y = wblpdf(0.1:0.1:0.6,lambda,1)
y =
  0.9048  0.4524  0.3016  0.2262  0.1810  0.1508

y1 = exppdf(0.1:0.1:0.6,lambda)
y1 =
  0.9048  0.4524  0.3016  0.2262  0.1810  0.1508
```

Reference

[1] Devroye, L., *Non-Uniform Random Variate Generation*, Springer-Verlag, 1986.

See Also

`pdf`, `wblcdf`, `wblfit`, `wblinv`, `wbllike`, `wblplot`, `wblrnd`, `wblstat`

wblplot

Purpose

Weibull probability plot

Syntax

```
wblplot(X)
h = wblplot(X)
```

Description

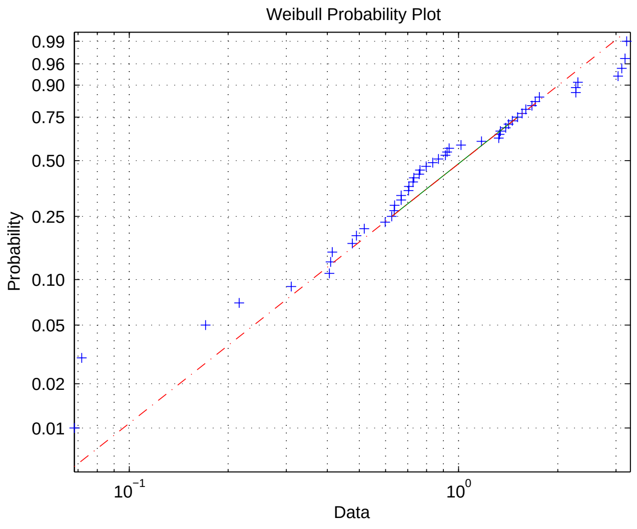
`wblplot(X)` displays a Weibull probability plot of the data in `X`. If `X` is a matrix, `wblplot` displays a plot for each column.

`h = wblplot(X)` returns handles to the plotted lines.

The purpose of a Weibull probability plot is to graphically assess whether the data in `X` could come from a Weibull distribution. If the data are Weibull the plot will be linear. Other distribution types might introduce curvature in the plot. `wblplot` uses midpoint probability plotting positions. Use `probplot` when the data included censored observations.

Example

```
r = wblrnd(1.2,1.5,50,1);
wblplot(r)
```



See Also

`probplot`, `normplot`, `wblcdf`, `wblfit`, `wblinv`, `wbllike`, `wblpdf`, `wblrnd`, `wblstat`

Purpose

Random numbers from Weibull distribution

Syntax

```
R = wblrnd(A,B)
R = wblrnd(A,B,v)
R = wblrnd(A,B,m,n)
```

Description

`R = wblrnd(A,B)` generates random numbers for the Weibull distribution with parameters `A` and `B`. The input arguments `A` and `B` can be either scalars or matrices. `A` and `B`, can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array of the same size as the other input.

`R = wblrnd(A,B,v)` generates random numbers for the Weibull distribution with parameters `A` and `B`, where `v` is a row vector. If `v` is a 1-by-2 vector, `R` is a matrix with `v(1)` rows and `v(2)` columns. If `v` is 1-by-n, `R` is an n-dimensional array.

`R = wblrnd(A,B,m,n)` generates random numbers for the Weibull distribution with parameters `A` and `B`, where scalars m and n are the row and column dimensions of `R`.

Devroye [1] refers to the Weibull distribution with a single parameter; this is `wblrnd` with A = `1`.

Example

```
n1 = wblrnd(0.5:0.5:2,0.5:0.5:2)
n1 =
  0.0178  0.0860  2.5216  0.9124

n2 = wblrnd(1/2,1/2,[1 6])
n2 =
  0.0046  1.7214  2.2108  0.0367  0.0531  0.0917
```

Reference

[1] Devroye, L., *Non-Uniform Random Variate Generation*, Springer-Verlag, 1986.

See Also

`wblcdf`, `wblfit`, `wblinv`, `wbllike`, `wblpdf`, `wblplot`, `wblstat`

Purpose

Mean and variance of Weibull distribution

Syntax

```
[M,V] = wblstat(A,B)
```

Description

`[M,V] = wblstat(A,B)` returns the mean of and variance for the Weibull distribution with parameters specified by `A` and `B`. Vector or matrix inputs for `A` and `B` must have the same size, which is also the size of `M` and `V`. A scalar input for `A` or `B` is expanded to a constant matrix with the same dimensions as the other input.

The mean of the Weibull distribution with parameters *a* and *b* is

$$a[\Gamma(1+b^{-1})]$$

and the variance is

$$a^2\left[\Gamma(1+2b^{-1})-\Gamma(1+b^{-1})^2\right]$$

Examples

```
[m,v] = wblstat(1:4,1:4)
m =
  1.0000  1.7725  2.6789  3.6256
v =
  1.0000  0.8584  0.9480  1.0346

wblstat(0.5,0.7)
ans =
  0.6329
```

See Also

`wblcdf`, `wblfit`, `wblinv`, `wbllike`, `wblpdf`, `wblplot`, `wblrnd`

Purpose

Random numbers from Wishart distribution

Syntax

```
W = wishrnd(sigma,df)
W = wishrnd(sigma,df,D)
[W,D] = wishrnd(sigma,df)
```

Description

`W = wishrnd(sigma,df)` generates a random matrix `W` having the Wishart distribution with covariance matrix `sigma` and with `df` degrees of freedom.

`W = wishrnd(sigma,df,D)` expects `D` to be the Cholesky factor of `sigma`. If you call `wishrnd` multiple times using the same value of `sigma`, it's more efficient to supply `D` instead of computing it each time.

`[W,D] = wishrnd(sigma,df)` returns `D` so you can provide it as input in future calls to `wishrnd`.

See Also

`iwishrnd`

Purpose

Convert predictor matrix to design matrix

Syntax

```
D = x2fx(X,model)
D = x2fx(X,model,categ)
D = x2fx(X,model,categ,catlevels)
```

Description

`D = x2fx(X,`*`model`*`)` converts a matrix of predictors `X` to a design matrix `D` for regression analysis. Distinct predictor variables should appear in different columns of `X`.

The optional input *`model`* controls the regression model. By default, `x2fx` returns the design matrix for a linear additive model with a constant term. *`model`* can be any one of the following strings:

- `'linear'` — Constant and linear terms (the default)
- `'interaction'` — Constant, linear, and interaction terms
- `'quadratic'` — Constant, linear, interaction, and squared terms
- `'purequadratic'` — Constant, linear, and squared terms

If `X` has n columns, the order of the columns of `D` for a full quadratic model is:

1. The constant term
2. The linear terms (the columns of `X`, in order 1, 2, ..., n)
3. The interaction terms (pairwise products of the columns of `X`, in order (1, 2), (1, 3), ..., (1, n), (2, 3), ..., (n–1, n))
4. The squared terms (in order 1, 2, ..., n)

Other models use a subset of these terms, in the same order.

Alternatively, *`model`* can be a matrix specifying polynomial terms of arbitrary order. In this case, *`model`* should have one column for each column in `X` and one row for each term in the model. The entries in any row of *`model`* are powers for the corresponding columns of `X`. For example, if `X` has columns `X1`, `X2`, and `X3`, then a row `[0 1 2]` in *`model`* would specify the term `(X1.^0).*(X2.^1).*(X3.^2)`. A row of all zeros in *`model`* specifies a constant term, which can be omitted.

`D = x2fx(X,`*`model`*`,categ)` treats columns with numbers listed in the vector `categ` as categorical variables. Terms involving categorical variables produce dummy variable columns in `D`. Dummy variables are computed under the assumption that possible categorical levels are completely enumerated by the unique values that appear in the corresponding column of `X`.

`D = x2fx(X,`*`model`*`,categ,catlevels)` accepts a vector `catlevels` the same length as `categ`, specifying the number of levels in each categorical variable. In this case, values in the corresponding column of `X` must be integers in the range from 1 to the specified number of levels. Not all of the levels need to appear in `X`.

Examples

Example 1

The following converts 2 predictors `X1` and `X2` (the columns of `X`) into a design matrix for a full quadratic model with terms `constant`, `X1`, `X2`, `X1.*X2`, `X1.^2`, and `X2.^2`.

```
X = [1 10
     2 20
     3 10
     4 20
     5 15
     6 15];

D = x2fx(X,'quadratic')
D =
     1     1    10    10     1   100
     1     2    20    40     4   400
     1     3    10    30     9   100
     1     4    20    80    16   400
     1     5    15    75    25   225
     1     6    15    90    36   225
```

Example 2

The following converts 2 predictors `X1` and `X2` (the columns of `X`) into a design matrix for a quadratic model with terms `constant`, `X1`, `X2`, `X1.*X2`, and `X1.^2`.

```
X = [1 10
     2 20
     3 10
     4 20
     5 15
     6 15];
model = [0 0
         1 0
         0 1
         1 1
         2 0];

D = x2fx(X,model)
D =
     1     1    10    10     1
     1     2    20    40     4
     1     3    10    30     9
     1     4    20    80    16
     1     5    15    75    25
     1     6    15    90    36
```

See Also

`x2fx` is a utility used by a variety of other functions, such as `rstool`, `regstats`, `candexch`, `candgen`, `cordexch`, and `rowexch`.

zscore

Purpose

Standardized *z*-scores

Syntax

```
Z = zscore(X)
[Z,mu,sigma] = zscore(X)
```

Description

`Z = zscore(X)` returns a centered, scaled version of `X`, the same size as `X`. For vector input `x`, output is the vector of *z*-scores `z = (x mean(x))./std(x)`. For matrix input `X`, *z*-scores are computed using the mean and standard deviation along each column of `X`. For higher-dimensional arrays, *z*-scores are computed using the mean and standard deviation along the first non-singleton dimension.

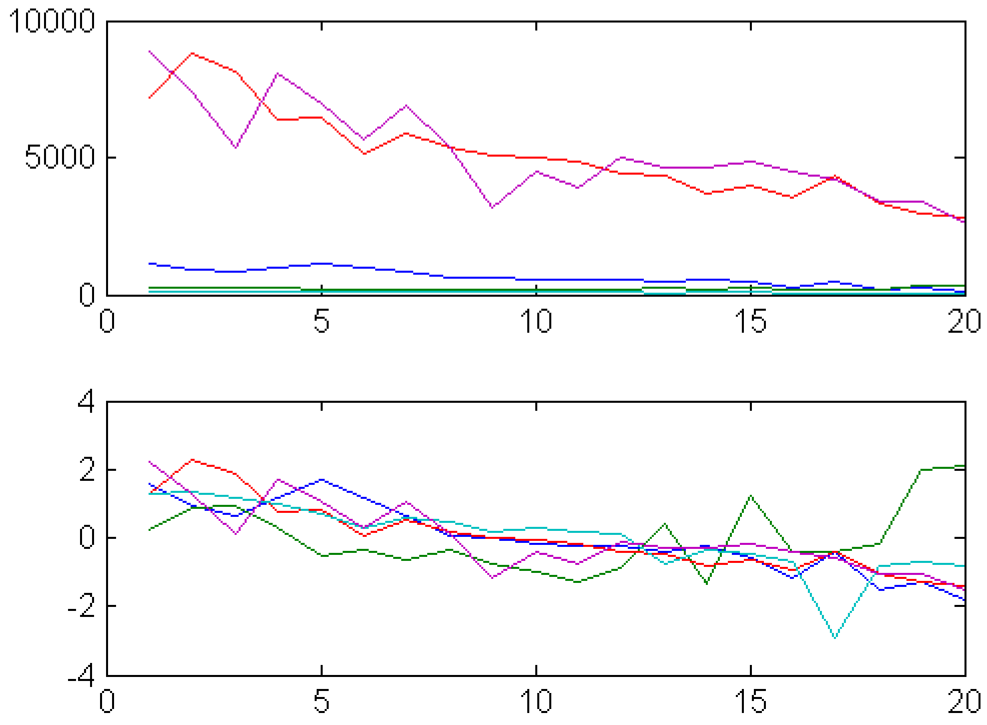
The columns of `Z` have mean zero and standard deviation one (unless a column of `X` is constant, in which case that column of `Z` is constant at `0`). *z*-scores are used to put data on the same scale before further analysis.

`[Z,mu,sigma] = zscore(X)` also returns `mean(X)` in `mu` and `std(X)` in `sigma`.

Example

Compare the predictors in the Moore data on original and standardized scales:

```
load moore
predictors = moore(:,1:5);
subplot(2,1,1),plot(predictors)
subplot(2,1,2),plot(zscore(predictors))
```



See Also `mean`, `std`

Purpose

One-sample z-test

Syntax

```
h = ztest(x,m,sigma)
h = ztest(...,alpha)
h = ztest(...,alpha,tail)
h = ztest(...,alpha,tail,dim)
[h,p] = ztest(...)
[h,p,ci] = ztest(...)
[h,p,ci,zval] = ztest(...)
```

Description

`h = ztest(x,m,sigma)` performs a z-test of the null hypothesis that data in the vector `x` are a random sample from a normal distribution with mean `m` and standard deviation `sigma`, against the alternative that the mean is not `m`. The result of the test is returned in `h`. `h = 1` indicates a rejection of the null hypothesis at the 5% significance level. `h = 0` indicates a failure to reject the null hypothesis at the 5% significance level.

`x` can also be a matrix or an N-dimensional array. For matrices, `ztest` performs separate z-tests along each column of `x` and returns a vector of results. For N-dimensional arrays, `ztest` works along the first non-singleton dimension of `x`.

The test treats `NaN` values as missing data, and ignores them.

`h = ztest(...,alpha)` performs the test at the (100*`alpha`)% significance level. The default, when unspecified, is `alpha = 0.05`.

`h = ztest(...,alpha,`*`tail`*`)` performs the test against the alternative specified by the string *`tail`*. There are three options for *`tail`*:

- `'both'` — Mean is not `m` (two-tailed test). This is the default, when `tail` is unspecified.
- `'right'` — Mean is greater than `m` (right-tail test)
- `'left'` — Mean is less than `m` (left-tail test)

`tail` must be a single string, even when `x` is a matrix or an N-dimensional array.

`h = ztest(...,alpha,tail,dim)` works along dimension `dim` of `x`. Use `[]` to pass in default values for `alpha` or `tail`.

`[h,p] = ztest(...)` returns the p-value of the test. The p-value is the probability, under the null hypothesis, of observing a value as extreme or more extreme of the test statistic

$$z = \frac{\overline{x} - \mu}{\sigma / \sqrt{n}}$$

where $\overline{x}$ is the sample mean, μ = `m` is the hypothesized population mean, σ is the population standard deviation, and n is the sample size. Under the null hypothesis, the test statistic will have a standard normal distribution, $N(0,1)$.

`[h,p,ci] = ztest(...)` returns a 100*(1 – `alpha`)% confidence interval on the population mean.

`[h,p,ci,zval] = ztest(...)` returns the value of the test statistic.

Example

Simulate a random sample of size 100 from a normal distribution with mean 0.1 and standard deviation 1:

```
x = normrnd(0.1,1,1,100);
```

Test the null hypothesis that the sample comes from a standard normal distribution:

```
[h,p,ci] = ztest(x,0,1)
h =
     0
p =
   0.1391
ci =
  -0.0481    0.3439
```

The test fails to reject the null hypothesis at the default $\alpha = 0.05$ significance level. Under the null hypothesis, the probability of observing a value as extreme or more extreme of the test statistic, as indicated by the p-value, is greater than α. The 95% confidence interval on the mean contains 0.

Simulate a larger random sample of size 1000 from the same distribution:

```
y = normrnd(0.1,1,1,1000);
```

Test again if the sample comes from a normal distribution with mean 0:

```
[h,p,ci] = ztest(y,0,1)
h =
     1
p =
  5.5160e-005
ci =
    0.0655    0.1895
```

This time the test rejects the null hypothesis at the default $\alpha = 0.05$ significance level. The p-value has fallen below $\alpha = 0.05$ and the 95% confidence interval on the mean does not contain 0.

Because the p-value of the sample `y` is less than 0.01, the test will still reject the null hypothesis when the significance level is lowered to $\alpha = 0.01$:

```
[h,p,ci] = ztest(y,0,1,0.01)
h =
     1
p =
  5.5160e-005
ci =
    0.0461    0.2090
```

This example will produce slightly different results each time it is run, because of the random sampling.

See Also `ttest`, `ttest2`

Bibliography

[1] Atkinson, A. C., and A. N. Donev, *Optimum Experimental Designs*, Oxford University Press, 1992.

[2] Bates, D. M., and D. G. Watts, *Nonlinear Regression Analysis and Its Applications*, Wiley, 1988.

[3] Bernoulli, J., *Ars Conjectandi*, Thurnisius, Basel, 1713.

[4] Box, G. E. P., W. G. Hunter, and J. S. Hunter, *Statistics for Experimenters*, Wiley-Interscience, 1978.

[5] Box, G. E. P., and N. R. Draper, *Empirical Model-Building and Response Surfaces*, Wiley, 1987.

[6] Breiman, L., J. Friedman, R. Olshen, and C. Stone, *Classification and Regression Trees*, Wadsworth, 1984.

[7] Bulmer, M. G., *Principles of Statistics*, Dover, 1979.

[8] Bury, K., *Statistical Distributions in Engineering*, Cambridge University Press, 1999.

[9] Chatterjee, S., and A. S. Hadi, "Influential Observations, High Leverage Points, and Outliers in Linear Regression," *Statistical Science*, 1, pp. 379-416, 1986.

[10] Collett, D., *Modeling Binary Data*, Chapman & Hall, 2002.

[11] Conover, W.J., *Practical Nonparametric Statistics*, Wiley, 1980

[12] Deb, P., and M. Sefton, “The distribution of a Lagrange multiplier test of normality,” *Economics Letters*, Vol. 51, pp. 123-130, 1996.

[13] Dempster, A. P., N.M. Laird, and D. B. Rubin, “Maximum Likelihood from Incomplete Data via the EM Algorithm,” *Journal of the Royal Statistical Society*, Series B, Vol. 39, No. 1, 1977, pp. 1-37.

[14] Devroye, L., *Non-Uniform Random Variate Generation*, Springer-Verlag, 1986.

[15] Dobson, A. J., *An Introduction to Generalized Linear Models*, Chapman & Hall, 1990.

[16] Draper, N. R., and H. Smith, *Applied Regression Analysis*, Wiley-Interscience, 1998.

[17] Durbin, R., S. Eddy, A. Krogh, and G. Mitchison, *Biological Sequence Analysis*, Cambridge University Press, 1998.

[18] Efron, B., and R. J. Tibshirani, *An Introduction to the Bootstrap*, Chapman & Hall, 1993.

[19] Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions*, Wiley-Interscience, 2000.

[20] Gibbons, J. D., *Nonparametric Statistical Inference*, Marcel Dekker, 1985.

[21] Hald, A., *Statistical Theory with Engineering Applications*, Wiley, 1960.

[22] Hastie, T., R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning*, Springer, 2001.

[23] Hogg, R. V., and J. Ledolter, *Engineering Statistics*, MacMillan, 1987.

[24] Hollander, M., and D. A. Wolfe, *Nonparametric Statistical Methods*, Wiley, 1999.

[25] Jarque, C.M., and A.K. Bera, “A test for normality of observations and regression residuals,” *International Statistical Review*, Vol. 55, No. 2, 1987, pp. 1-10.

[26] Johnson, N. L., S. Kotz, and N. Balakrishnan, *Continuous Univariate Distributions,* Volume 1, Wiley-Interscience, 1993.

[27] Johnson, N. L., S. Kotz, and N. Balakrishnan, *Continuous Univariate Distributions,* Volume 2, Wiley-Interscience, 1994.

[28] Johnson, N. L., S. Kotz, and N. Balakrishnan, *Discrete Multivariate Distributions*, Wiley-Interscience, 1997.

[29] Johnson, N. L., N. Balakrishnan, and S. Kotz, *Continuous Multivariate Distributions*, Volume 1, Wiley-Interscience, 2000.

[30] Johnson, N. L., S. Kotz, and A. W. Kemp, *Univariate Discrete Distributions*, Wiley-Interscience, 1993.

[31] Krzanowski, W. J., *Principles of Multivariate Analysis: A User's Perspective*, Oxford University Press, 1988.

[32] Lawless, J. F., *Statistical Models and Methods for Lifetime Data*, Wiley-Interscience, 2002.

[33] Lilliefors, H.W., "On the Komogorov-Smirnov test for normality with mean and variance unknown," *Journal of the American Statistical Association*, vol. 62, 1967, pp. 399-402.

[34] Lilliefors, H.W., "On the Kolmogorov-Smirnov test for the exponential distribution with mean unknown," *Journal of the American Statistical Association*, vol. 64, 1969, pp. 387-389.

[35] Little, Roderick J. A. and Donald B. Rubin, *Statistical Analysis with Missing Data*, 2nd ed., John Wiley & Sons, Inc., 2002.

[36] Mardia, K. V., J. T. Kent, and J. M. Bibby, *Multivariate Analysis*, Academic Press, 1980.

[37] Martinez, W. L., and A. R. Martinez, *Computational Statistics with MATLAB*, Chapman & Hall/CRC, 2002.

[38] McCullagh, P., and J. A. Nelder, *Generalized Linear Models*, Chapman & Hall, 1990.

[39] Meeker, W. Q., and Escobar, L. A., *Statistical Methods for Reliability Data*, Wiley, 1998.

[40] Meng, Xiao-Li and Donald B. Rubin, “Maximum Likelihood Estimation via the ECM Algorithm,” *Biometrika*, Vol. 80, No. 2, 1993, pp. 267-278

[41] Montgomery, D. C., *Design and Analysis of Experiments*, Wiley, 2001.

[42] Moore, J., *Total Biochemical Oxygen Demand of Dairy Manures,* Ph.D. thesis, University of Minnesota, Department of Agricultural Engineering, 1975.

[43] Poisson, S. D., *Recherches sur la Probabilité des Jugements en Matière Criminelle et en Matière Civile, Précédées des Regles Générales du Calcul des Probabilités*, Bachelier, Imprimeur-Libraire pour les Mathematiques, Paris, 1837.

[44] Rice, J. A., *Mathematical Statistics and Data Analysis*, Duxbury Press, 1994.

[45] Seber, G. A. F., *Linear Regression Analysis*, Wiley-Interscience, 2003.

[46] Seber, G. A. F., *Multivariate Observations*, Wiley, 1984.

[47] Seber, G. A. F., and C. J. Wild, *Nonlinear Regression*, Wiley-Interscience, 2003.

[48] Sexton, Joe and Anders Rygh Swensen, “ECM Algorithms that Converge at the Rate of EM,” *Biometrika*, Vol. 87, No. 3, 2000, pp. 651-662.

[49] Snedecor, G. W., and W. G. Cochran, *Statistical Methods*, Iowa State Press, 1989.

[50] Student, “On the Probable Error of the Mean,” *Biometrika*, 6:1-25, 1908.

[51] Vellemen, P. F., and D. C. Hoaglin, *Application, Basics, and Computing of Exploratory Data Analysis*, Duxbury Press, 1981.

[52] Weibull, W., "A Statistical Theory of the Strength of Materials," *Ingeniors Vetenskaps Akademiens Handlingar*, Stockholm: Royal Swedish Institute for Engineering Research, No. 151, 1939.

[53] Wild, C. J., and G. A. F. Seber, *Chance Encounters: A First Course in Data Analysis and Inference*, Wiley, 1999.

Index