Statistics Toolbox™ 7
User's Guide

MATLAB®

The MathWorks™

Accelerating the pace of engineering and science
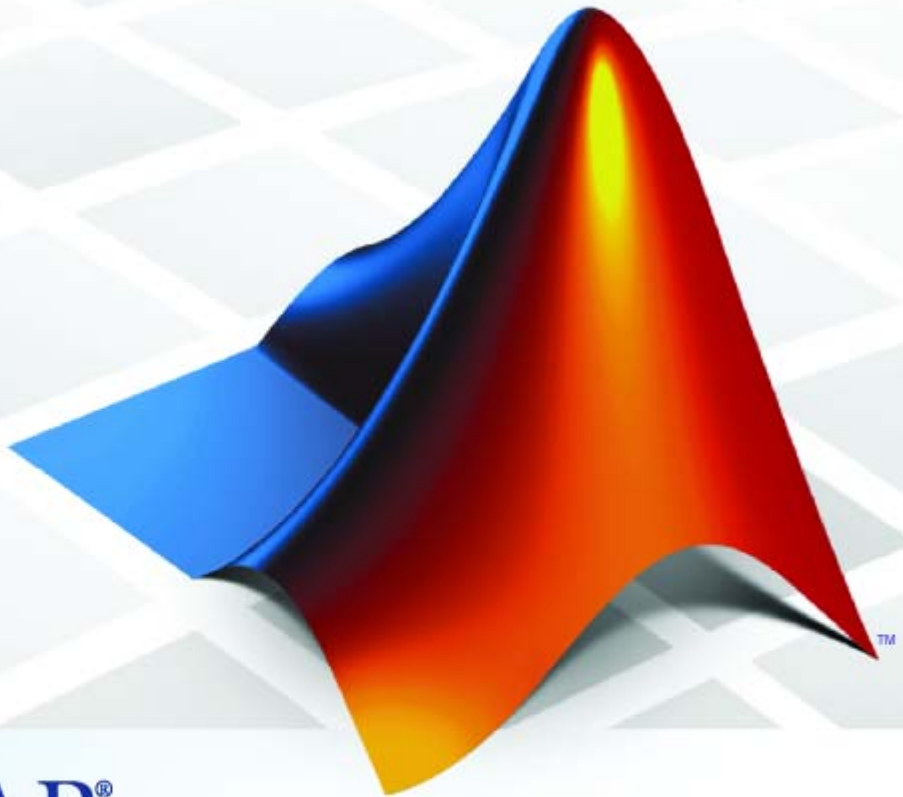
How to Contact The MathWorks

Statistics Toolbox™ User's Guide

Trademarks

Patents

Contents

Statistical Visualization

4

Probability Distributions

5

6

7

Regression Analysis

8

Multivariate Methods

9

Cluster Analysis

10

Classification

11

12   Markov Models

Design of Experiments

13

Statistical Process Control

14

15

16   Class Reference

Functions — Alphabetical List

17

Data Sets

A

Distribution Reference

B

Bibliography

C

Index

Getting Started

Product Overview

Statistics Toolbox™ software 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 data analysis

Code for the building-block functions is open and extensible. Use the MATLAB Editor to review, copy, and edit M-file code for any function. 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 using sliders, input fields, push buttons, etc. and the GUIs automatically call building-block functions.

2

Organizing Data

Introduction

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 "Other 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 "Classes (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 software 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 2-23 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

In this section...
"Numerical Data" on page 2-4
"Heterogeneous Data" on page 2-7
"Statistical Functions" on page 2-9

Numerical Data

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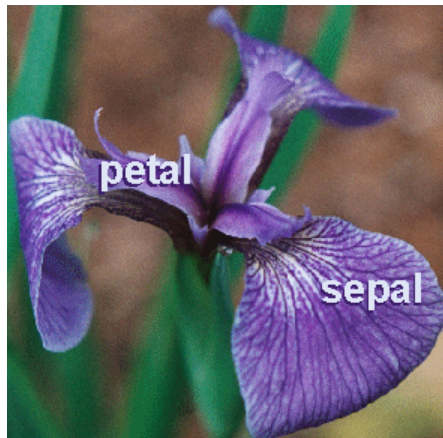
The observations in meas are of three different species of iris (setosa, versicolor, and virginica), which can be separated from one another using the 150-by-1 cell array of strings species:

```
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 NaNs to indicate "missing values." These limitations are addressed by dataset arrays (see "Dataset Arrays" on page 2-23), which are specifically designed for statistical data.

Heterogeneous Data

MATLAB data types include two container variables—cell arrays and structure arrays—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. MATLAB and Statistics Toolbox statistical functions 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 limitation of cell arrays is addressed by dataset arrays (see "Dataset Arrays" on page 2-23), 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 limitations as cell arrays. Once again, dataset arrays (see "Dataset Arrays" on page 2-23), designed specifically for general statistical data, address these limitations.

Statistical Functions

One of the advantages of working in the MATLAB language 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 MATLAB and Statistics Toolbox statistical functions 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.

MATLAB and Statistics Toolbox statistical functions, 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.

MATLAB statistical functions 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. Statistics Toolbox functions are more flexible. Many toolbox functions accept data input arguments in the form of both numerical arrays and dataset arrays (see "Dataset Arrays" on page 2-23), which are specifically designed for storing general statistical data.

Statistical Arrays

In this section...
"Introduction" on page 2-11
"Categorical Arrays" on page 2-13
"Dataset Arrays" on page 2-23

Introduction

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

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

- "Categorical Arrays" on page 2-13
- "Dataset Arrays" on page 2-23

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 methods for assembling, accessing, manipulating, and processing the collected data. Basic array operations parallel those for numerical, cell, and structure arrays.

Categorical Arrays

- "Categorical Data" on page 2-13
- "Categorical Arrays" on page 2-14
- "Using Categorical Arrays" on page 2-16

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 using MATLAB 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 software, 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 2-33.

Categorical arrays come in two types, depending on whether the collected data is understood to be nominal or ordinal. Nominal arrays are constructed with `nominal`; ordinal arrays are constructed with `ordinal`. 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 2-16, and the reference pages for `nominal` and `ordinal`, for further examples.

Categorical arrays are implemented as objects of the `categorical` class. The class is abstract, defining properties and methods common to both the

`nominal` class and `ordinal` class. Use the corresponding constructors, `nominal` or `ordinal`, to create categorical arrays. Methods of the classes are used to display, summarize, convert, concatenate, and access the collected data. Many of these methods are invoked using operations analogous to those for numerical arrays, and do not need to be called directly (for example, `[ ]` invokes `horzcat`). Other methods, such as `reorderlevels`, must be called directly.

Using Categorical Arrays

This section provides an extended tutorial example demonstrating the use of categorical arrays with methods of the `nominal` class and `ordinal` class.

- "Constructing Categorical Arrays" on page 2-16
- "Accessing Categorical Arrays" on page 2-18
- "Combining Categorical Arrays" on page 2-19
- "Computing with Categorical Arrays" on page 2-20

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 Variable Editor (see "Viewing and Editing Workspace Variables with the Variable Editor" in the MATLAB documentation). 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 Variable 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

Use `ordinal` 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 Variable 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. Use `sort` to sort ordinal data in ascending order:

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

When displayed in the Variable 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 `getlabels` 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 ordinal 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 2-33.

Use ismember 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)
```
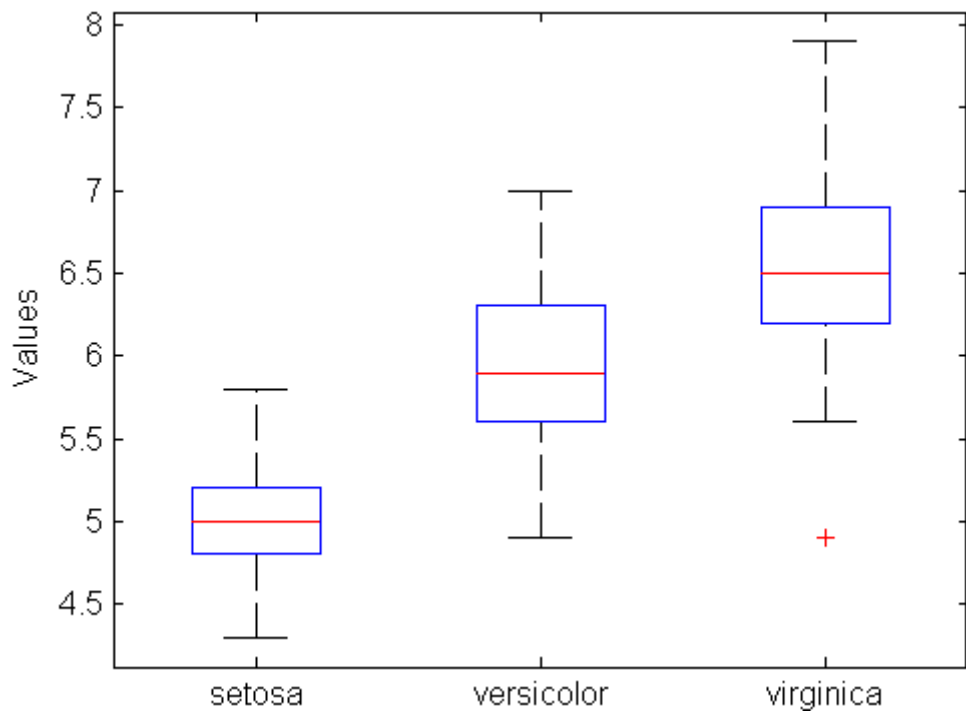
Dataset Arrays

Statistical Data

MATLAB data containers (variables) are 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 variables created with `dataset`. 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);
NameObs = strcat({'Obs'},num2str((1:NumObs)','%d'));
iris = dataset({nominal(species),'species'},...
               {meas,'SL','SW','PL','PW'},...
               'ObsNames',NameObs);
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 `set` and accessed with `get`:

```
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'}
```

Dataset arrays are implemented as objects of the `dataset` class. Methods of the class are used to display, summarize, convert, concatenate, and access the collected data. Many of these methods are invoked using operations analogous to those for numerical arrays, and do not need to be called directly (for example, `[]` invokes `horzcat`). Other methods, such as `sortrows`, must be called directly.

Using Dataset Arrays

This section provides an extended tutorial example demonstrating the use of dataset arrays with methods of the `dataset` class.

- "Constructing Dataset Arrays" on page 2-25
- "Accessing Dataset Arrays" on page 2-27
- "Combining Dataset Arrays" on page 2-29
- "Computing with Dataset Arrays" on page 2-31

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).

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

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 `set` 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 `get` 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 the reference on the `dataset` class.

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, apply `zscore` 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 `join`, 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. Use summary to provide 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 datasetfun, 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. Logical vectors can be used to indicate membership (or not) in a single group.

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. Grouping variables with multiple columns are used to specify different groups within multiple variables.

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 2-34. Examples are given in "Using Grouping Variables" on page 2-35.

Functions for Grouped Data

The following table lists Statistics Toolbox functions 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 2-33.

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 2-35 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)
cvpartition	c = cvpartition(group,'Kfold',k) or c = cvpartition(group,'Holdout',p)
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)

Function	Basic Syntax for Grouped Data
`kruskalwallis`	`p = kruskalwallis(X,group)`
`maineffectsplot`	`maineffectsplot(X,group)`
`manova1`	`d = manova1(X,group)`
`multivarichart`	`multivarichart(x,group)`
`parallelcoords`	`parallelcoords(X, ...  ,'Group',group)`
`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 2-33. A list of functions accepting grouping variables as input arguments is given in "Functions for Grouped Data" on page 2-34.

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 methods of the `categorical` class.

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 2-23) from `meas`:

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

When you call `grpstats` with a dataset array as an argument, you invoke the `grpstats` method of the `dataset` class, rather than the `grpstats` function. The method has a slightly different syntax than the 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. Use `ismember` 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')
```

3
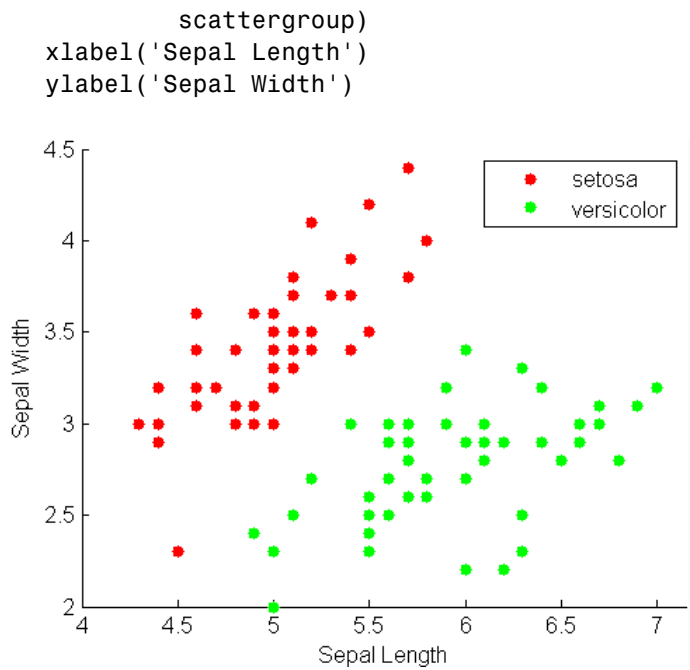
Descriptive Statistics

Introduction

Large, complex data sets need to be summarized—both numerically and visually—to convey their essence to the data analyst and to allow for further processing. This chapter focuses on numerical summaries; Chapter 4, "Statistical Visualization" focuses on visual summaries.

Measures of Central Tendency

Measures of central tendency locate a distribution of data along an appropriate scale.

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
median	50th percentile
mode	Most frequent value
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
moment	Central moment of all orders
range	Range
std	Standard deviation
var	Variance

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
```

Measures of Shape

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

The *quantile* of order p ($0 \leq p \leq 1$) 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 $0 \leq p \leq 100$.

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
     1.8293     4.6728     5.6459     6.0766     7.1546
```

A box plot helps to visualize the statistics:

```
boxplot(x)
```
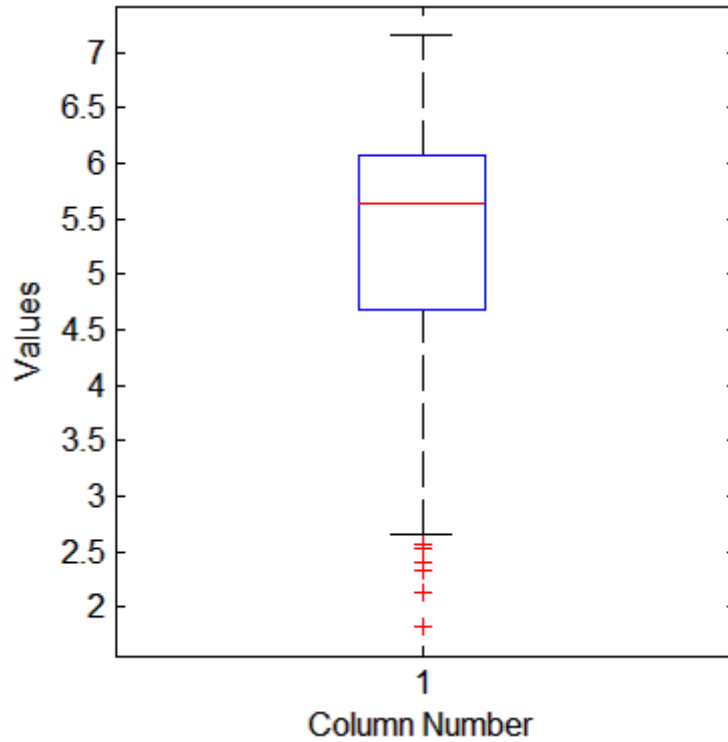
The long lower tail and plus signs show the lack of symmetry in the sample values. For more information on box plots, see "Box Plots" on page 4-6.

The shape of a data distribution is also measured by the Statistics Toolbox functions `skewness`, `kurtosis`, and, more generally, `moment`.

Resampling Statistics

In this section...
"The Bootstrap" on page 3-9
"The Jackknife" on page 3-12

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 [31] 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
```
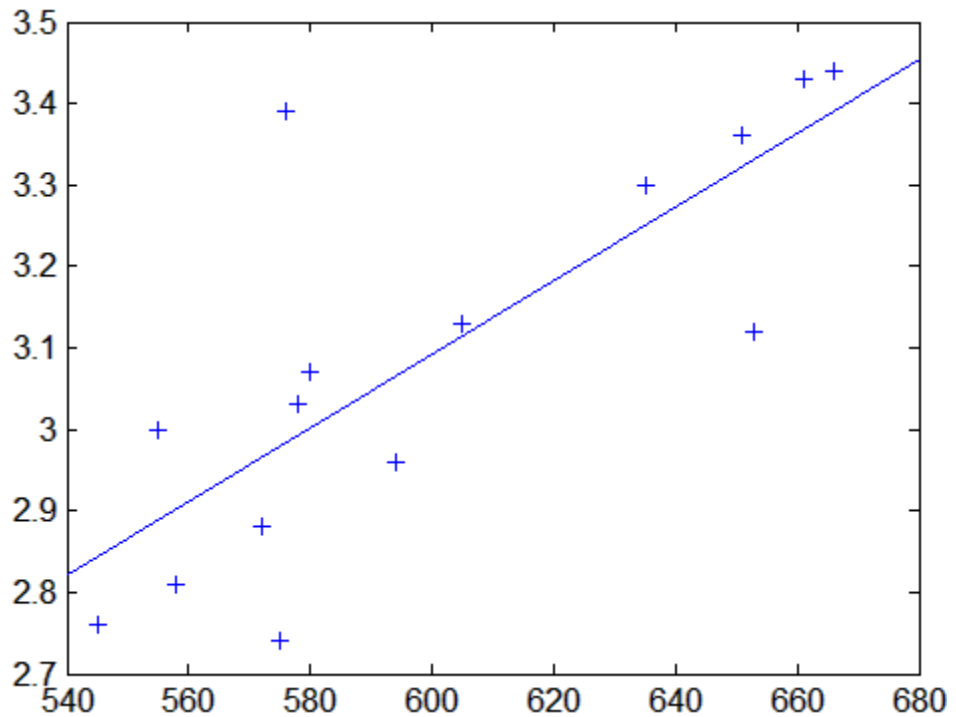
The least-squares fit line indicates that higher LSAT scores go with higher law school GPAs. But how certain is this conclusion? The plot provides some intuition, but nothing quantitative.

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 describing the positive connection between LSAT and GPA; though it 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(rhos1000,30)
set(get(gca,'Children'),'FaceColor',[.8 .8 1])
```
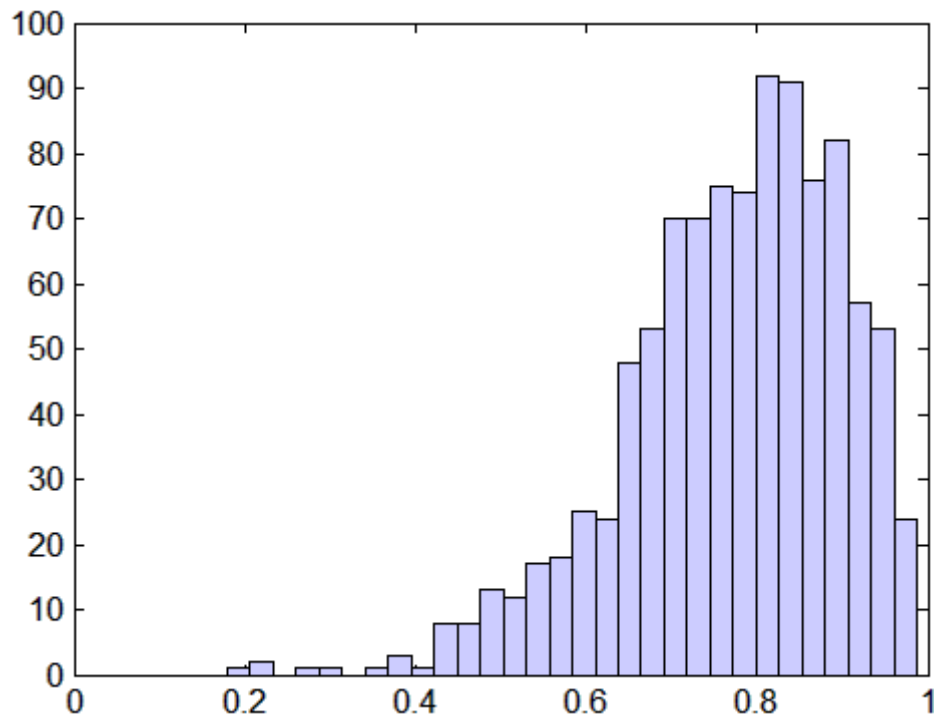

Nearly all the estimates lie on the interval [0.4 1.0].

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.

The Jackknife

Similar to the bootstrap is the *jackknife*, which uses resampling to estimate the bias and variance of sample statistics. The jackknife is implemented by the Statistics Toolbox function `jackknife`.

Data with Missing Values

Many data sets have one or more missing values. It is convenient to code missing values as NaN (Not a Number) to preserve the structure of data sets across multiple variables and observations.

For example:

```
X = magic(3);
X([1 5]) = [NaN NaN]
X =
   NaN    1    6
     3  NaN    7
     4    9    2
```

Normal MATLAB arithmetic operations yield NaN values when operands are NaN:

```
s1 = sum(X)
s1 =
   NaN   NaN   15
```

Removing the NaN values would destroy the matrix structure. Removing the rows containing the NaN values would discard data. Statistics Toolbox functions in the following table remove NaN values only for the purposes of computation.

Function	Description
nancov	Covariance matrix, ignoring NaN values
nanmax	Maximum, ignoring NaN values
nanmean	Mean, ignoring NaN values
nanmedian	Median, ignoring NaN values
nanmin	Minimum, ignoring NaN values
nanstd	Standard deviation, ignoring NaN values
nansum	Sum, ignoring NaN values
nanvar	Variance, ignoring NaN values

For example:

```
s2 = nansum(X)
s2 =
    7    10    15
```

Other Statistics Toolbox functions also ignore NaN values. These include iqr, kurtosis, mad, prctile, range, skewness, and trimmean.

4

Statistical Visualization

Introduction

Statistics Toolbox data visualization functions add to the extensive graphics capabilities already in MATLAB.

- 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 is used to 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.

Advanced Statistics Toolbox visualization functions are available for specialized statistical analyses.

Note For information on creating visualizations of data by group, see "Grouped Data" on page 2-33.

Scatter Plots

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

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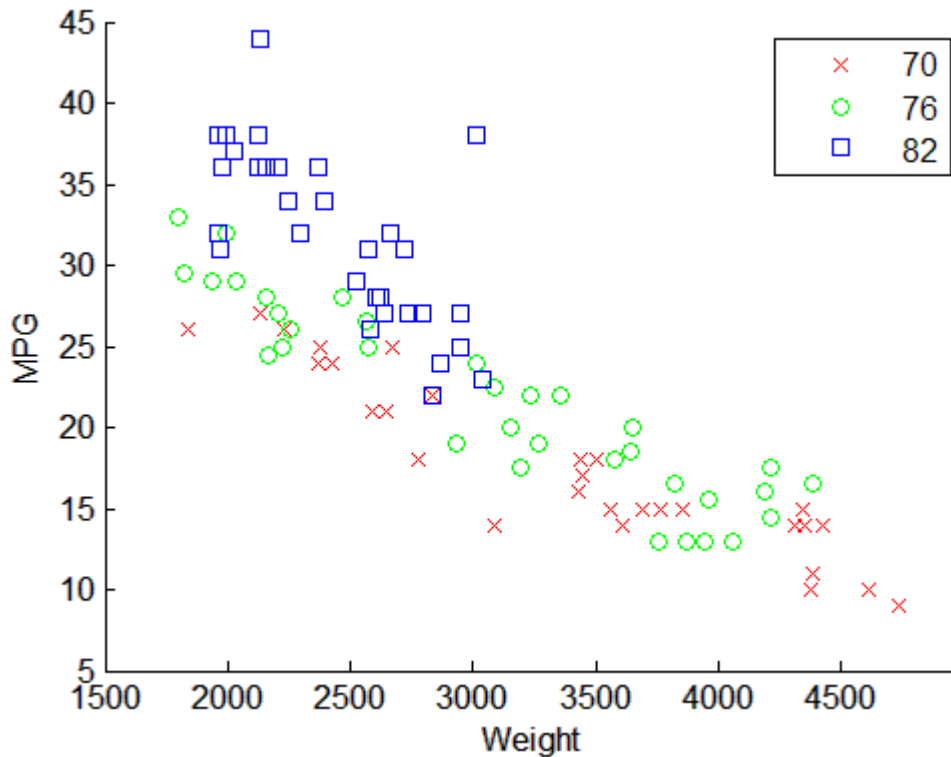
This shows that not only is there a strong relationship between the weight of a car and its mileage, but also that newer cars tend to be lighter and have better gas mileage than older cars.

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')
```
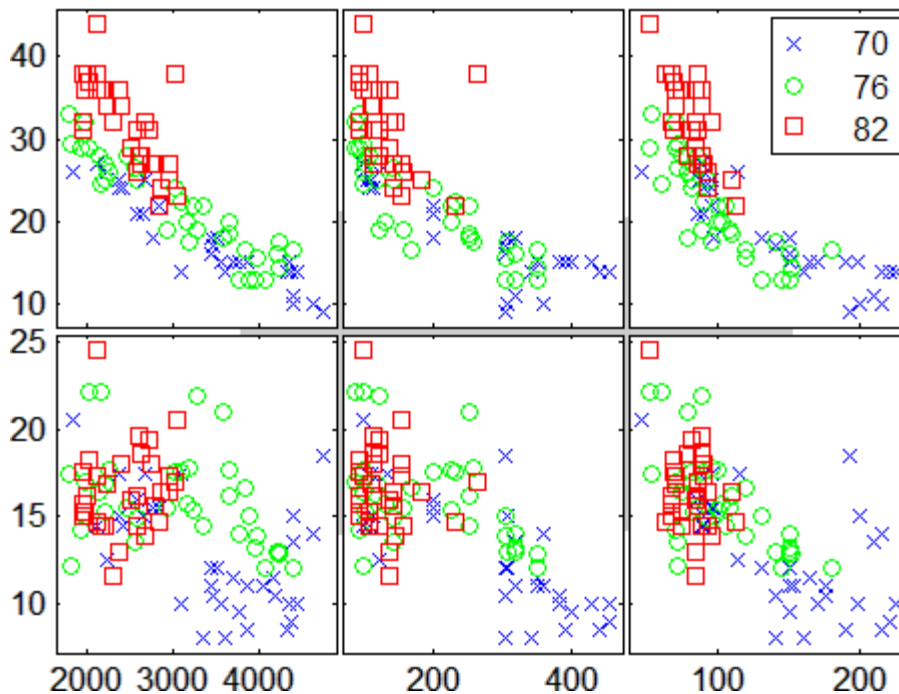

The upper right subplot displays MPG against Horsepower, and shows that over the years the horsepower of the cars has decreased but the gas mileage has improved.

The gplotmatrix function can also graph all pairs from a single list of variables, along with histograms for each variable. See "MANOVA" on page 7-39.

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'})
```
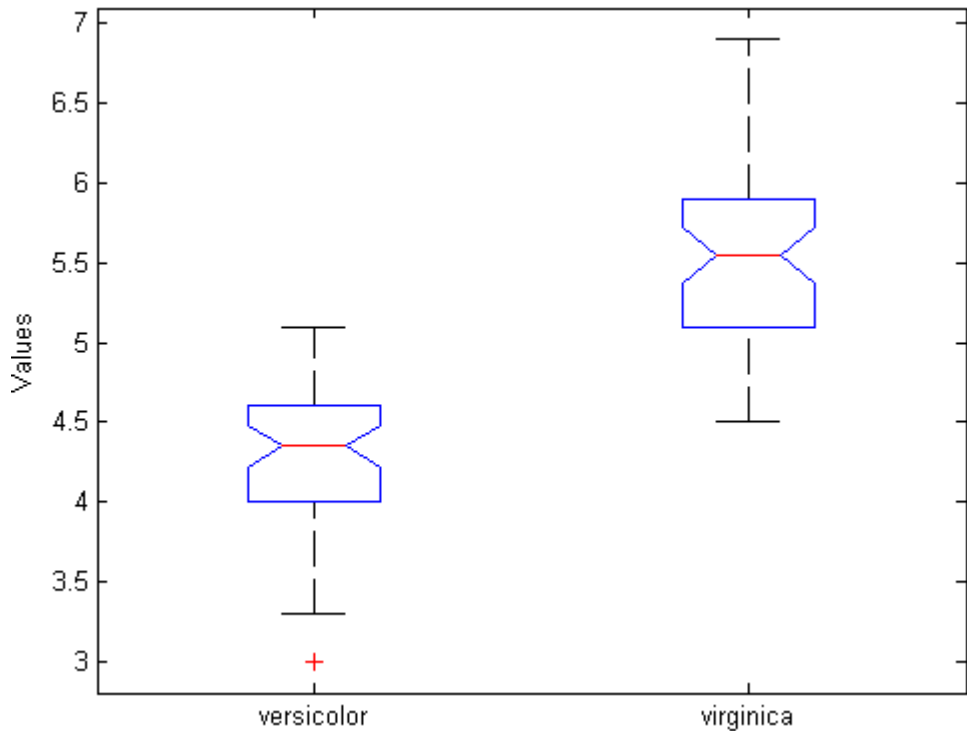

This plot has the following features:

- 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

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)
```

Normal Probability Plot
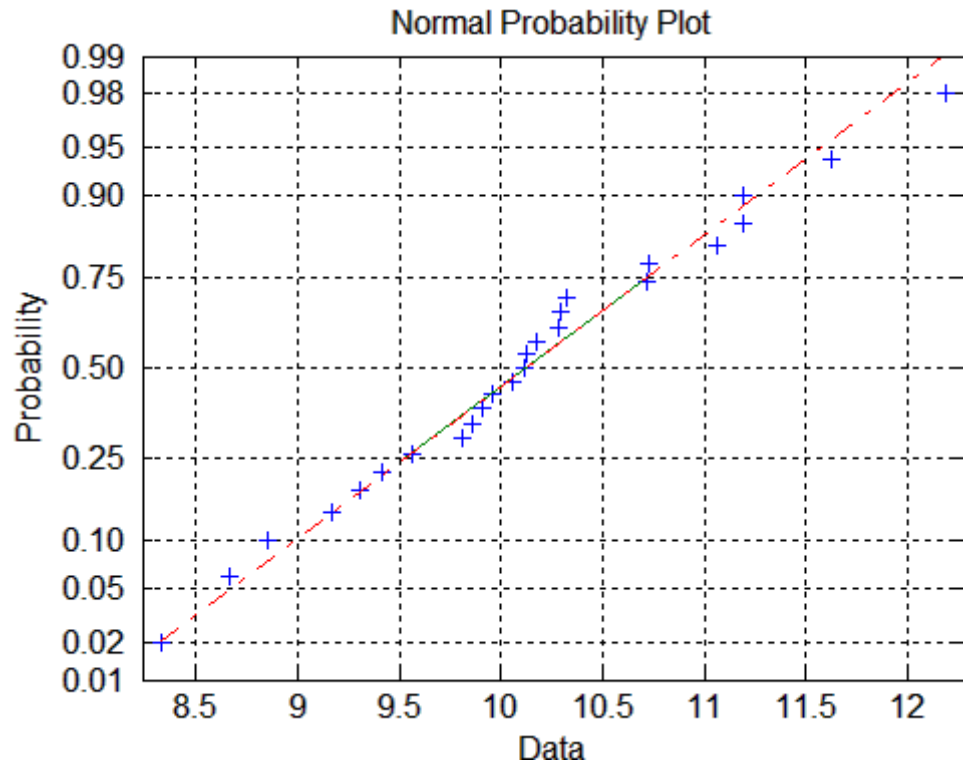
The plus signs plot the empirical probability versus the data value for each point in the data. A solid line connects the 25th and 75th percentiles in the data, and a dashed line extends it to the ends of the data. The *y*-axis values are probabilities from zero to one, but the scale is not linear. The 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)
```


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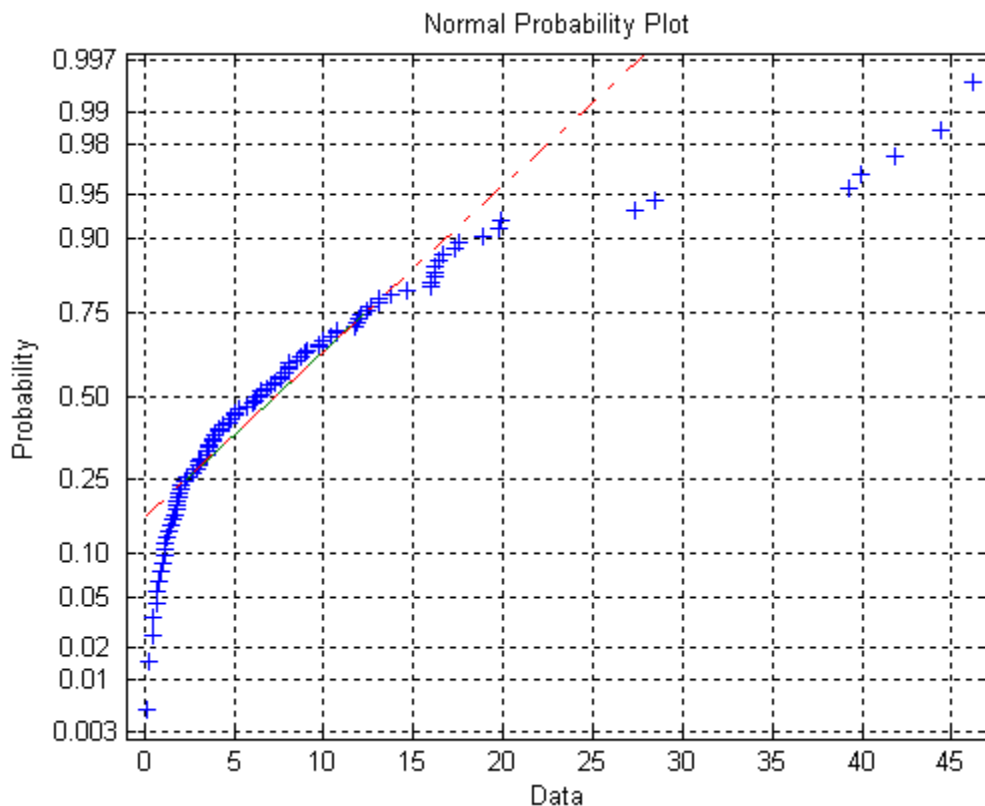
The plot is strong evidence that the underlying distribution is not normal.

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);
```
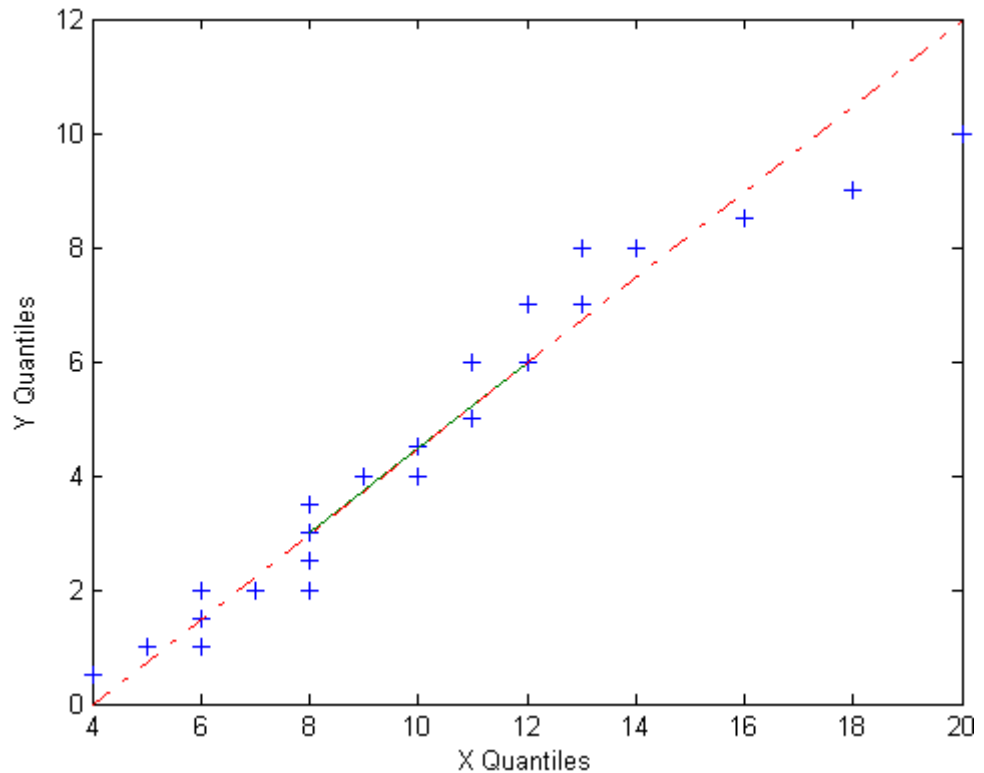

Even though the parameters and sample sizes are different, the approximate linear relationship suggests that the two samples may come from the same distribution family. As with normal probability plots, hypothesis tests, as described in Chapter 6, "Hypothesis Tests", can provide additional justification 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);
```
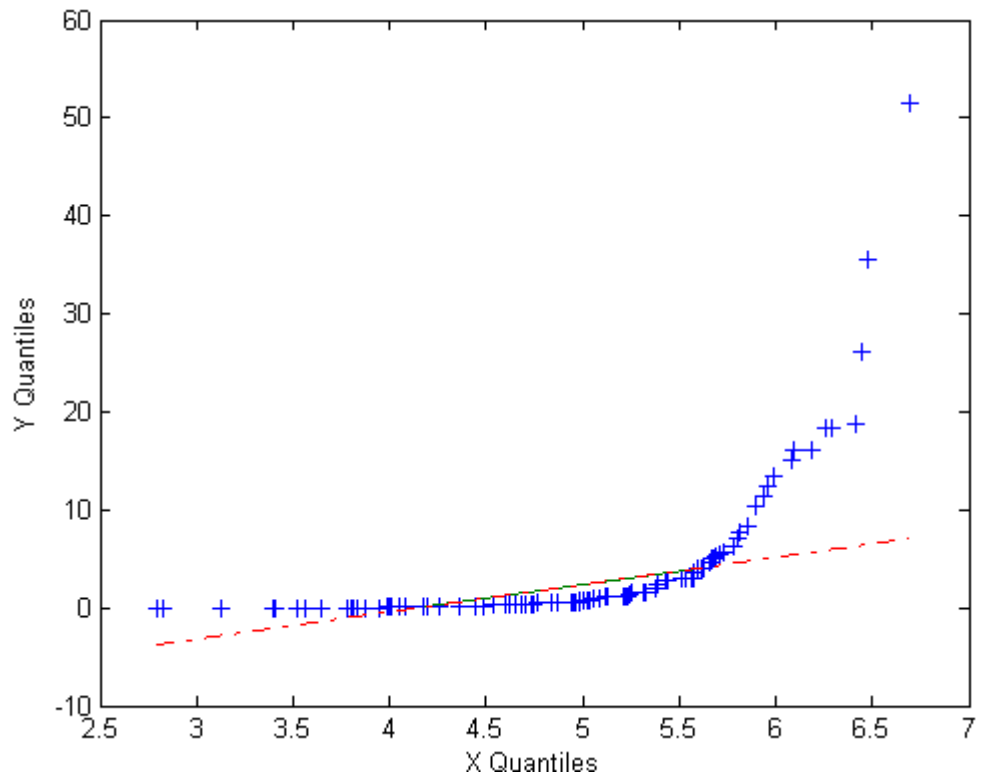

These samples clearly are not from the same distribution family.

Cumulative Distribution 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')
```
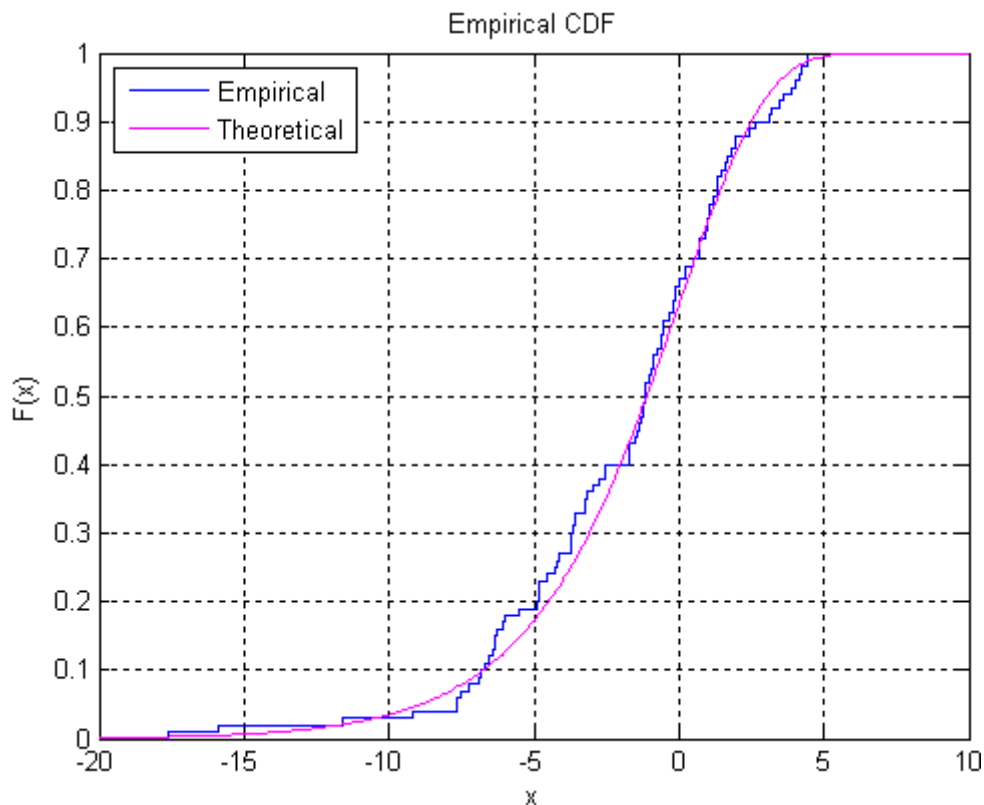
Other Probability Plots

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')
```
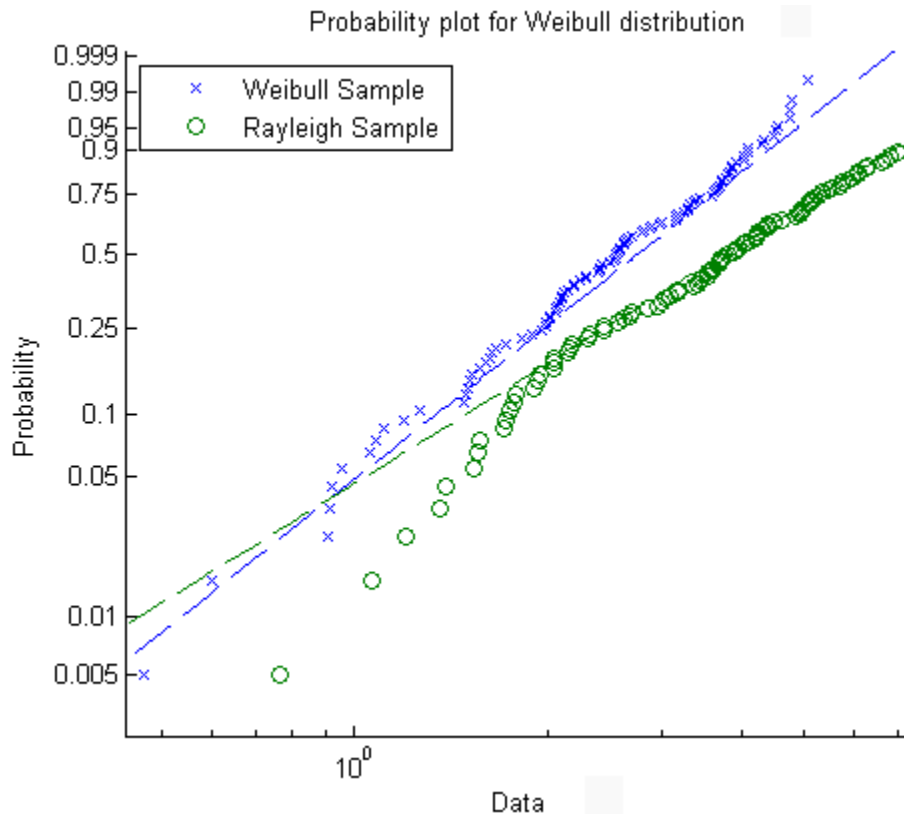

The plot gives justification for modeling the first sample with a Weibull distribution; much less so for the second sample.

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

5

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 data or statistics do not follow any standard probability distribution (as, for example, with multimodal 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.

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

Tables of Supported Distributions

Probability distributions supported by Statistics Toolbox software are cross-referenced with their supporting functions and GUIs in the following tables. 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 fully described in Appendix B, "Distribution Reference".

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, mle,dfittool	gevlike	gevrnd, random, randtool
Generalized Pareto	gppdf, pdf	gpcdf, cdf	gpinv, icdf	gpstat	gpfit, mle,dfittool	gplike	gprnd, random, randtool
Inverse Gaussian					dfittool		
Johnson system					johnsrnd		johnsrnd
Logistic					dfittool		
Loglogistic					dfittool		
Lognormal	lognpdf, pdf	logncdf, cdf	logninv, icdf	lognstat	lognfit, mle, dfittool	lognlike	lognrnd, random, randtool
Nakagami					dfittool		

Name	pdf	cdf	inv	stat	fit	like	rnd
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`		`pearsrnd`
Piecewise	`pdf`	`cdf`	`icdf`		`paretotails`		`random`
Rayleigh	`raylpdf, pdf`	`raylcdf, cdf`	`raylinv, icdf`	`raylstat`	`raylfit, mle, dfittool`		`raylrnd, random, randtool`
Rician					`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
t location-scale					dfittool		

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	copulafit		copularnd
Gaussian mixture	pdf	cdf			fit		random
t copula	copulapdf	copulacdf		copulastat	copulafit		copularnd
Clayton copula	copulapdf	copulacdf		copulastat	copulafit		copularnd
Frank copula	copulapdf	copulacdf		copulastat	copulafit		copularnd
Gumbel copula	copulapdf	copulacdf		copulastat	copulafit		copularnd
Inverse Wishart							iwishrnd
Multivariate normal	mvnpdf	mvncdf					mvnrnd
Multivariate *t*	mvtpdf	mvtcdf					mvtrnd
Wishart							wishrnd

Distribution Functions

Introduction

For each distribution supported by Statistics Toolbox software, a selection of the distribution functions described in this section 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

- "Parametric Estimation" on page 5-9
- "Nonparametric Estimation" on page 5-13

Parametric Estimation

Probability density functions (pdfs) for supported Statistics Toolbox distributions 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
set(get(gca,'Children'),'FaceColor',[.8 .8 1])
grid on
```
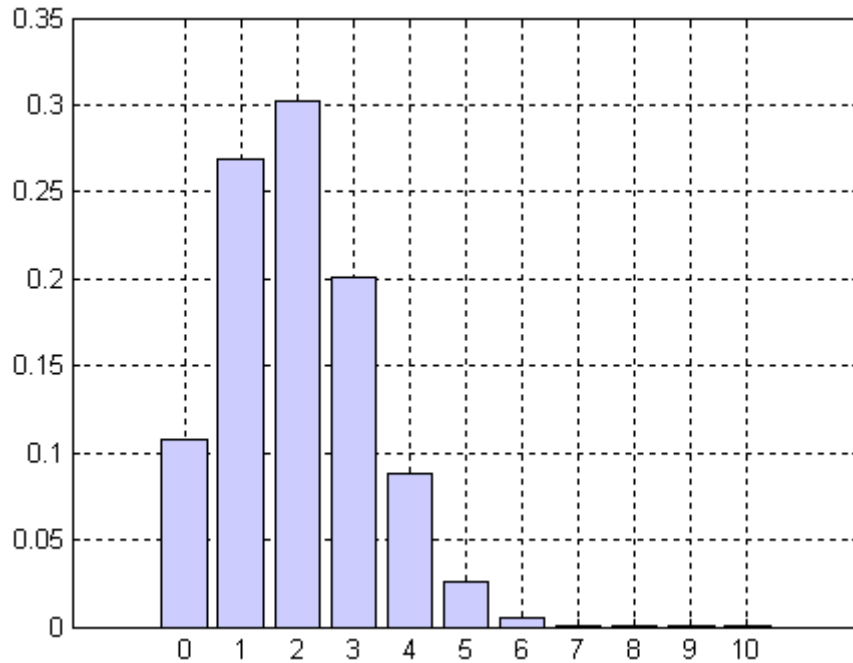
For continuous distributions, the pdf assigns a probability *density* to each outcome. The probability of any single outcome is zero. The pdf must be integrated over a set of outcomes to compute the probability that an outcome falls within that set. The integral over the entire set of outcomes is 1.

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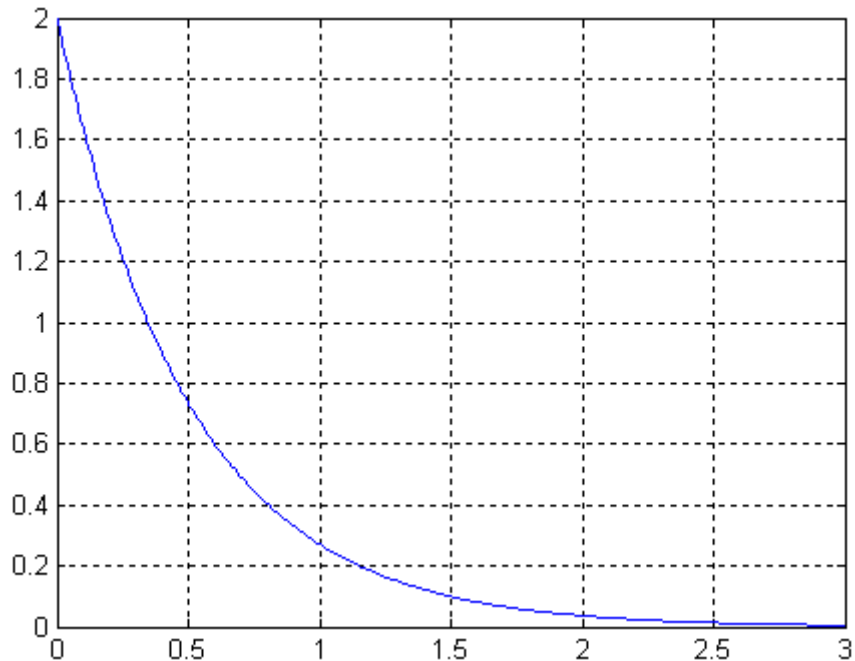
Probabilities for continuous pdfs can be computed with the `quad` function. In the example above, the probability of failure in the time interval [0, 1] is computed as follows:

```
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
```

Nonparametric Estimation

- "Introduction" on page 5-13
- "Kernel Bandwidth" on page 5-14
- "Kernel Smoothing" on page 5-16
- "Using Density Estimates" on page 5-17

Introduction. A distribution of data can be described graphically with a histogram:

```
cars = load('carsmall','MPG','Origin');
MPG = cars.MPG;
hist(MPG)
set(get(gca,'Children'),'FaceColor',[.8 .8 1])
```
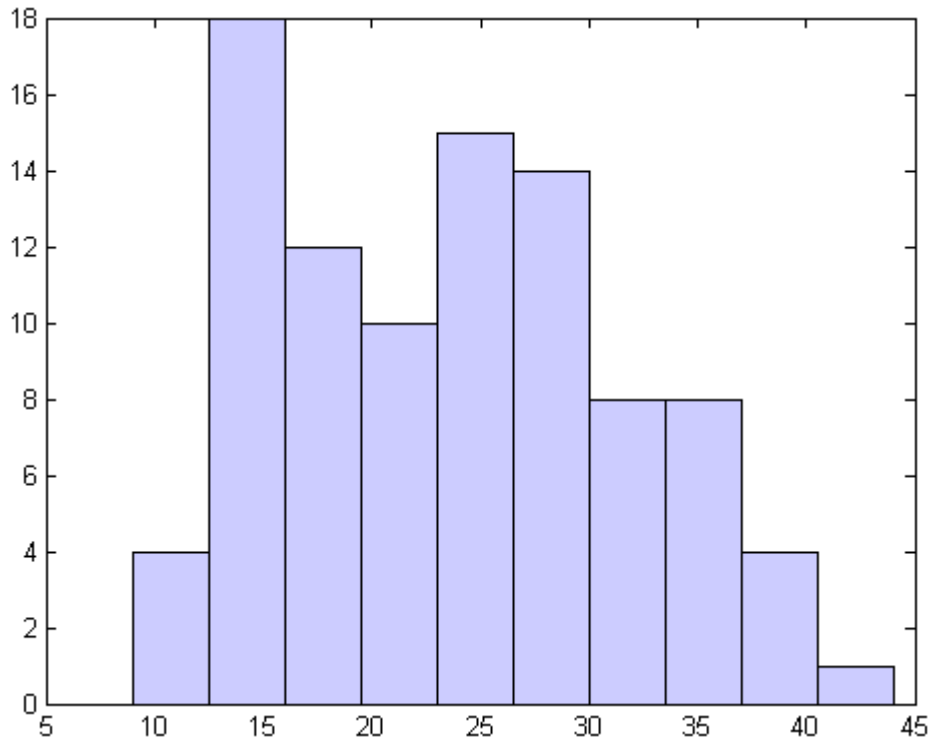
You can also describe a data distribution by estimating its density. The ksdensity function does this using a kernel smoothing method. A nonparametric density estimate of the data above, using the default kernel and bandwidth, is given by:

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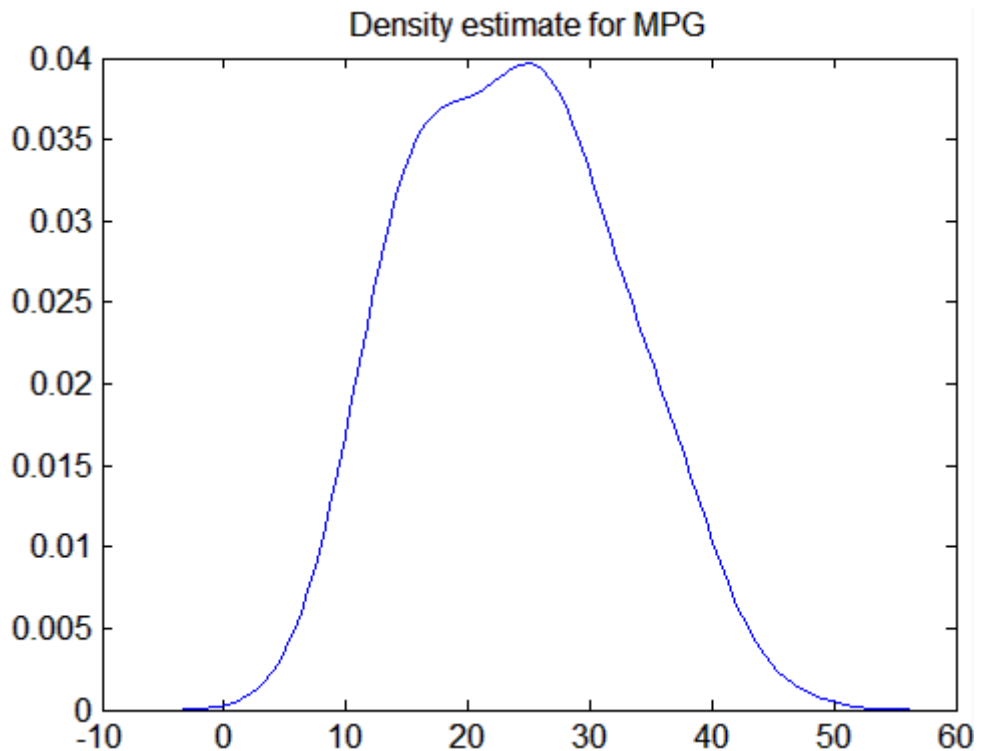

Kernel Bandwidth. The choice of kernel bandwidth controls the smoothness of the probability density curve. The following graph shows the density estimate for the same mileage data using different bandwidths. The default bandwidth is in blue and looks like the preceding graph. Estimates for smaller and 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
```
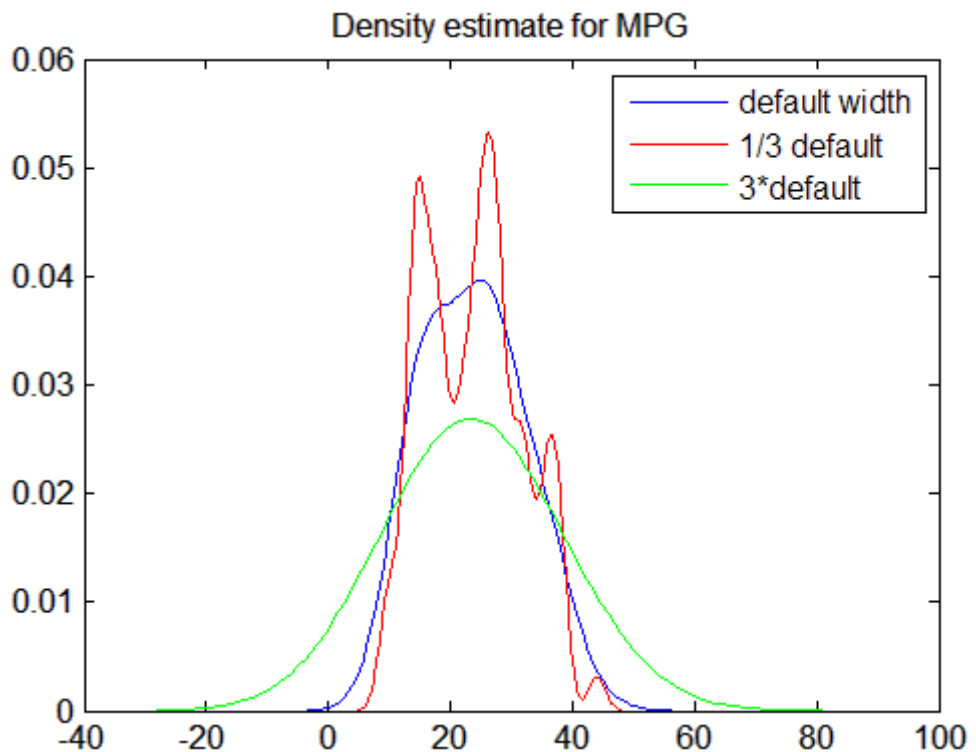
The default bandwidth seems to be doing a good job—reasonably smooth, but not so smooth as to obscure features of the data. This bandwidth is the one that is theoretically optimal for estimating densities for the normal distribution.

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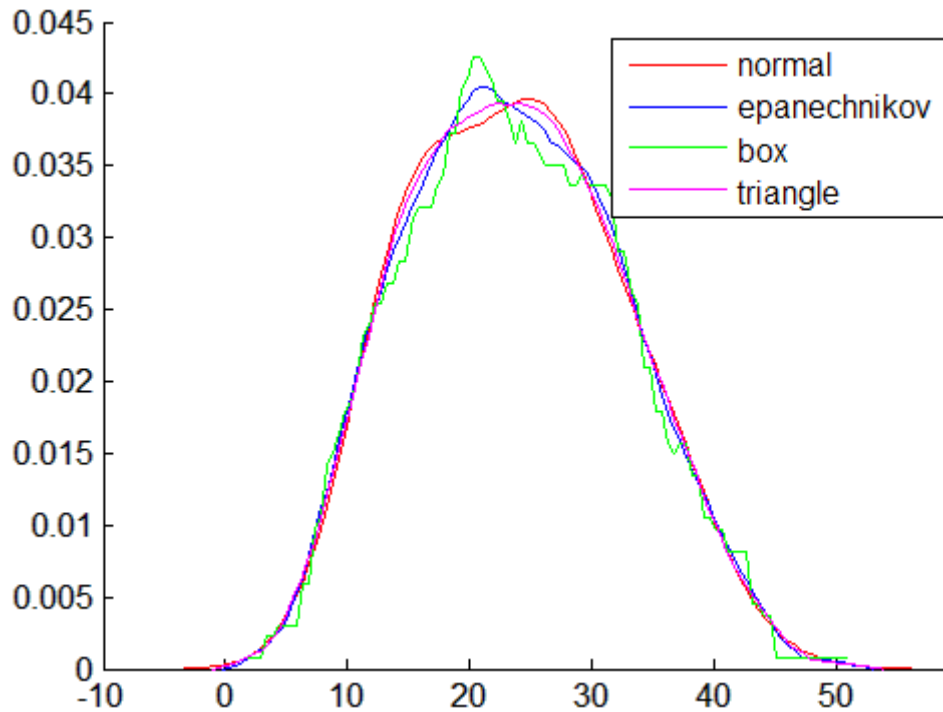
The density estimates are roughly comparable, but the box kernel produces a density that is rougher than the others.

Using Density Estimates. 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:

```
Origin = cellstr(cars.Origin);

I = strcmp('USA',Origin);
J = strcmp('Japan',Origin);
K = ~(I|J);
MPG_USA = MPG(I);
MPG_Japan = MPG(J);
MPG_Europe = MPG(K);
```

```
[fI,xI] = ksdensity(MPG_USA);
plot(xI,fI,'b')
hold on

[fJ,xJ] = ksdensity(MPG_Japan);
plot(xJ,fJ,'r')

[fK,xK] = ksdensity(MPG_Europe);
plot(xK,fK,'g')

legend('USA','Japan','Europe')
hold off
```
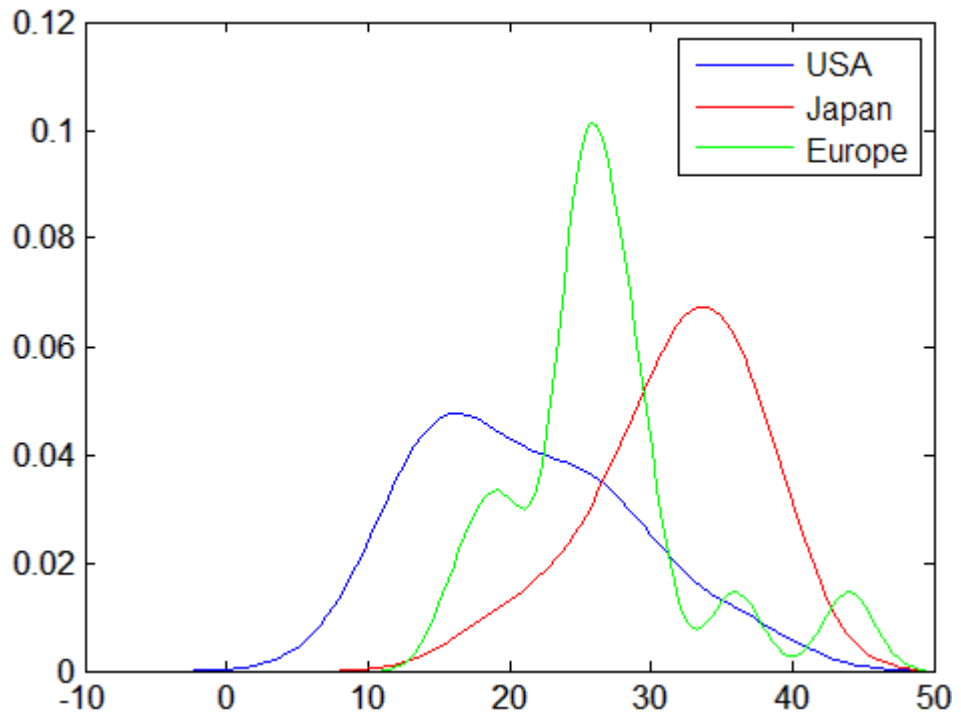

For piecewise probability density estimation, using kernel smoothing in the center of the distribution and Pareto distributions in the tails, see "Fitting Piecewise Distributions" on page 5-29.

Cumulative Distribution Functions

- "Parametric Estimation" on page 5-19
- "Nonparametric Estimation" on page 5-20

Parametric Estimation

Cumulative distribution functions (cdfs) for supported Statistics Toolbox distributions 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
```

Nonparametric Estimation

The `ksdensity` function 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 adapts 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)
hold on
xx=linspace(5,15,100);
yy = normcdf(xx,10,2);
plot(xx,yy,'r:')
hold off
legend('Empirical cdf','Normal cdf',2)
```
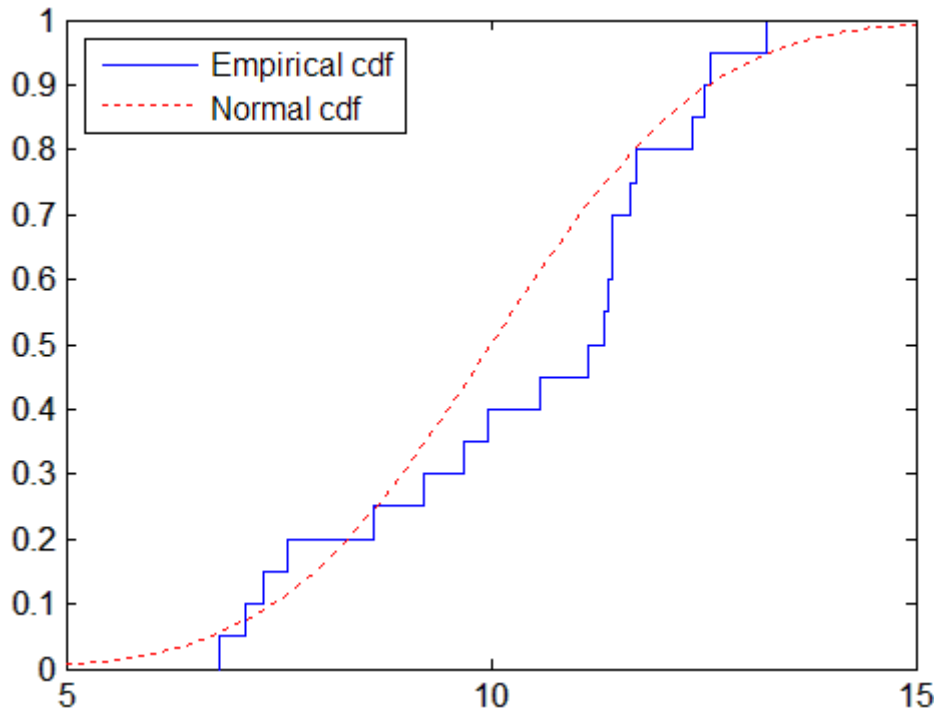
The empirical cdf is especially useful in survival analysis applications. In such applications the data may be censored, that is, not observed exactly. Some individuals may fail during a study, and you can observe their failure time exactly. Other individuals may drop out of the study, or may not fail 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 5-29.

Inverse Cumulative Distribution Functions

Inverse cumulative distribution functions for supported Statistics Toolbox distributions 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
```

5-23

```
>> 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 Statistics Toolbox distributions 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)
```
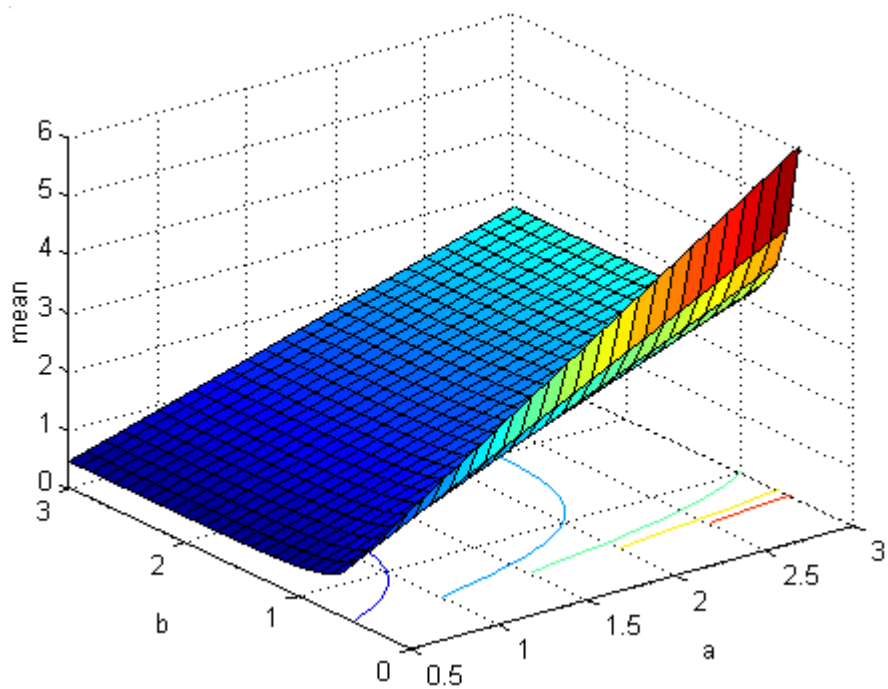
Distribution Fitting Functions

- "Fitting Supported Distributions" on page 5-27
- "Fitting Piecewise Distributions" on page 5-29

Fitting Supported Distributions

Distribution fitting functions for supported Statistics Toolbox distributions 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 software 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)
set(get(gca,'Children'),'FaceColor',[.8 .8 1])
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)
```
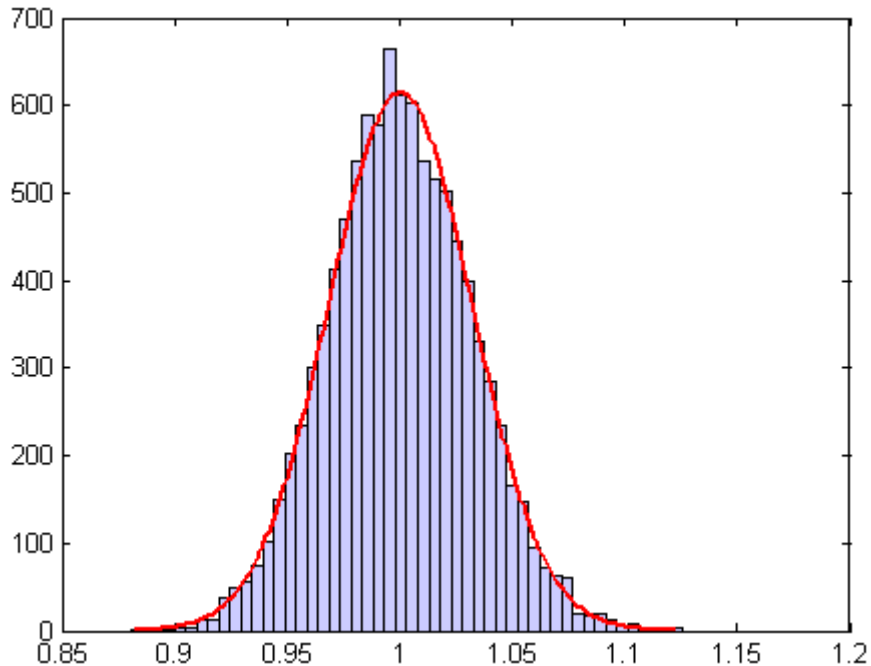
Fitting Piecewise Distributions

The parametric methods discussed in "Fitting Supported Distributions" on page 5-27 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 Statistics Toolbox distributions.

The empirical distributions computed by `ecdf` and discussed in "Nonparametric Estimation" on page 5-20 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 generalized Pareto distributions (GPDs) in the tails. The output is an object of the `paretotails`

class, with associated methods to evaluate the cdf, inverse cdf, and other functions of the fitted distribution.

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');
t = @(data,mu,sig,df)cdf('tlocationscale',data,mu,sig,df);
h = probplot(gca,t,p);
set(h,'color','r','linestyle','-')
title('{\bf Probability Plot}')
legend('Data','Normal','t','Location','NW')
```
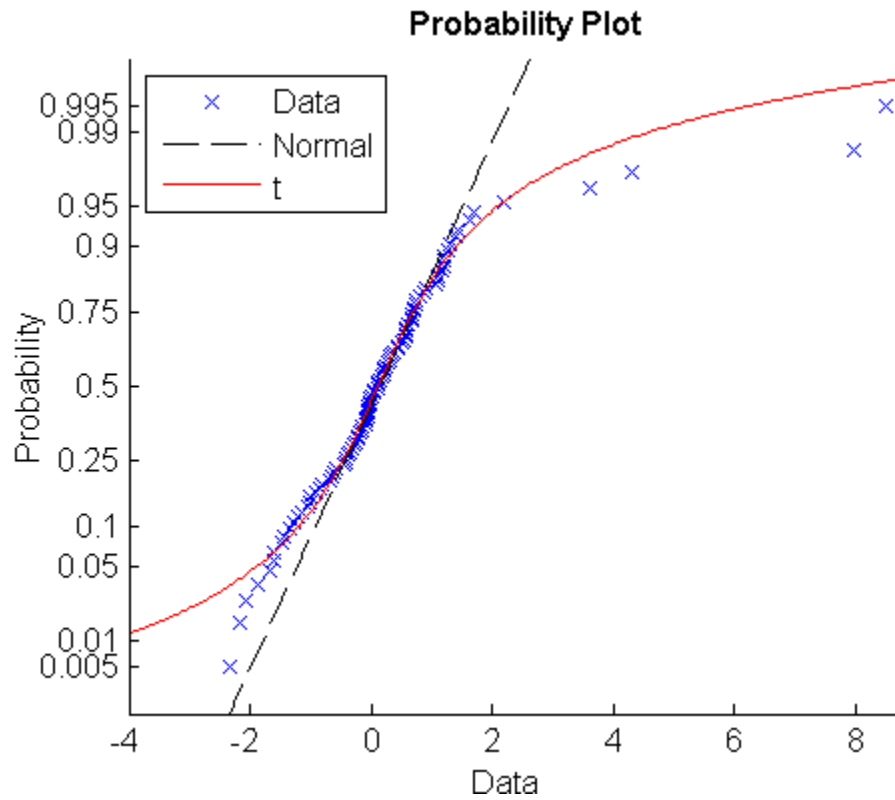
On the other hand, the empirical distribution provides a perfect fit, but the outliers make the tails very discrete:

```
ecdf(data)
```
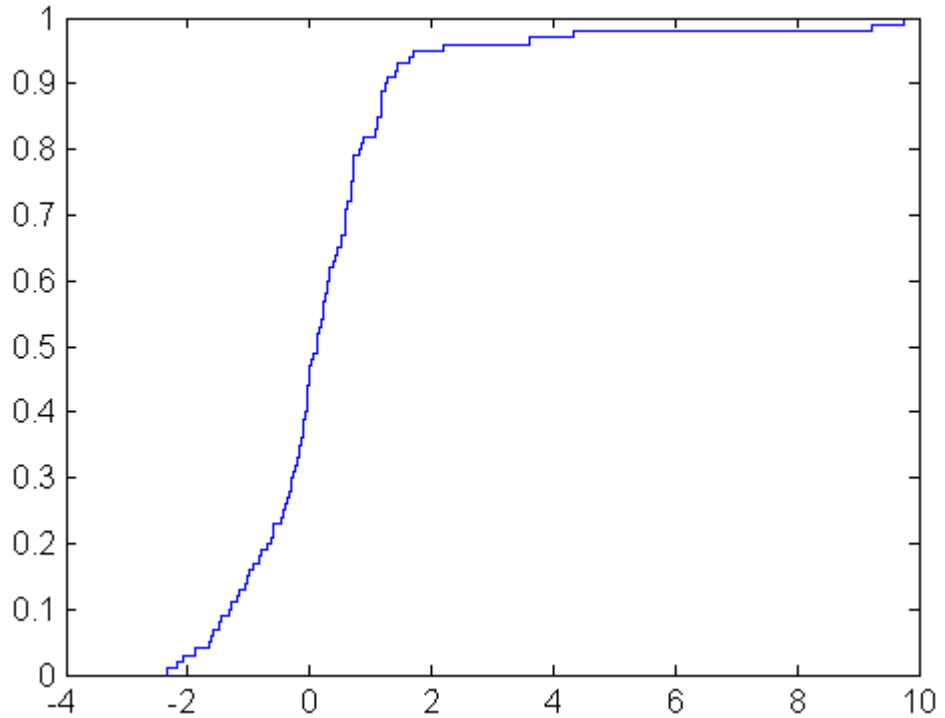
Random samples generated from this distribution by inversion might include, for example, values around 4.33 and 9.25, but nothing in-between.

The `paretotails` function 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,cdf(pfit,x))
```
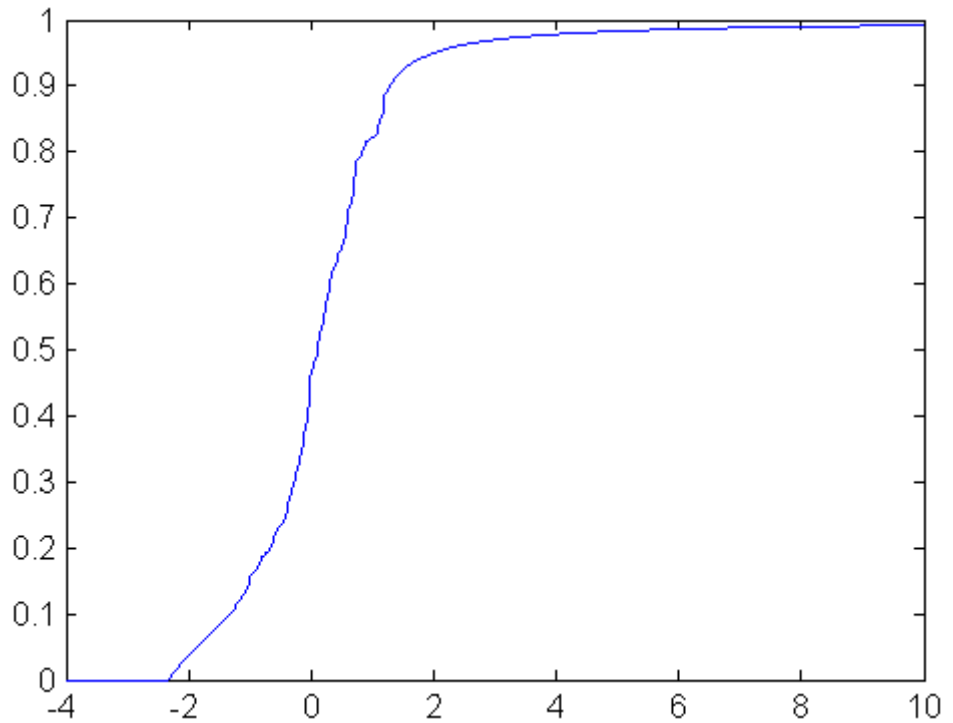

Access information about the fit using the methods of the `paretotails` class. Options allow for nonparametric estimation of the center of the cdf.

Negative Log-Likelihood Functions

Negative log-likelihood functions for supported Statistics Toolbox distributions 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 MATLAB function `fminsearch`. Additional search algorithms are implemented by Optimization Toolbox™ functions and Genetic Algorithm and Direct Search Toolbox™ functions.

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 5-27 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, Statistics Toolbox negative log-likelihood

functions 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)
```
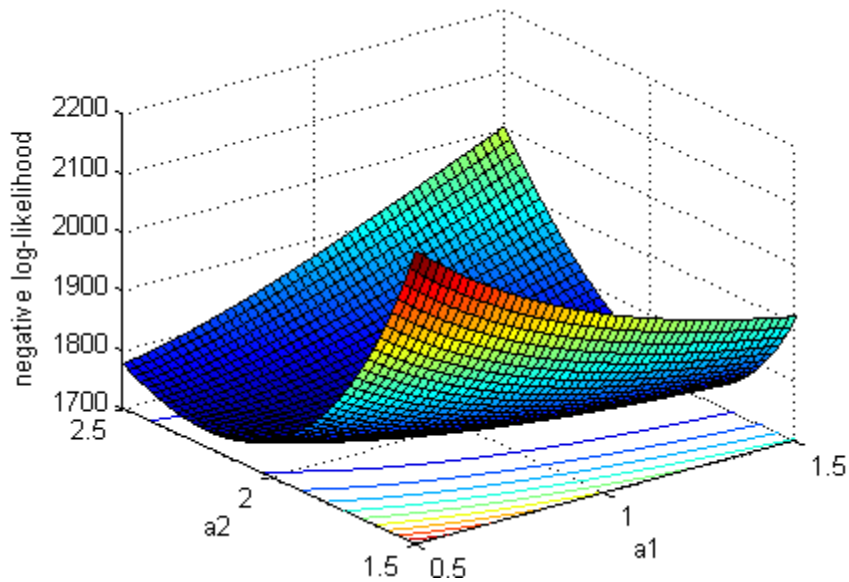
The MATLAB function fminsearch is used to search for the minimum of the likelihood surface:

```
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')
```
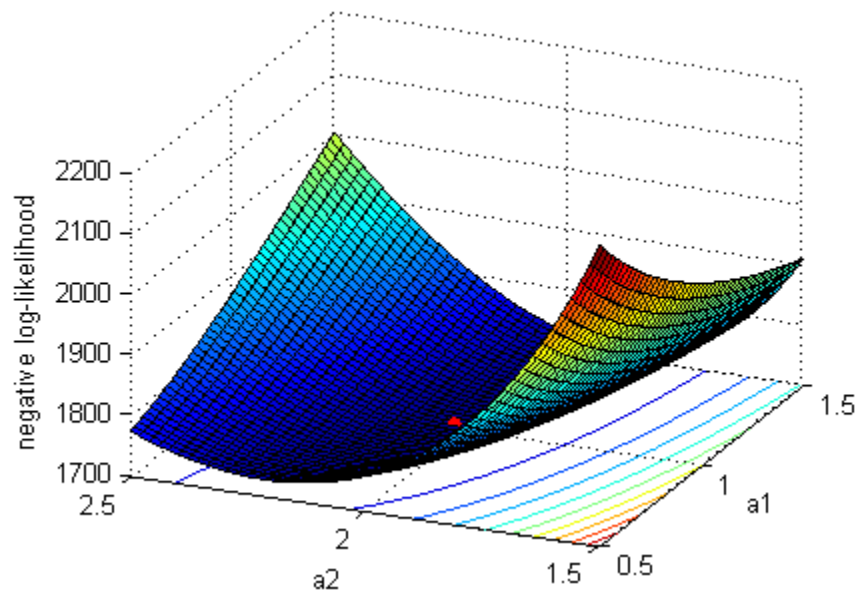
Random Number Generators

Random number generators (RNGs) for supported Statistics Toolbox distributions all end with rnd, as in binornd or exprnd. Specific RNG names for specific distributions are 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 software depend on the MATLAB base generators rand and/or randn, using the techniques discussed in "Common Generation Methods" on page 5-132 to generate random numbers from particular distributions. Dependencies of specific RNGs are listed in the next table.

MATLAB resets the state of the base RNGs each time it is started. Thus, by default, dependent RNGs in Statistics Toolbox software 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 ensure 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'])
h = get(gca,'Children');
set(h(1),'FaceColor',[.8 .8 1])
legend('UNIFRND','BINORND')
```
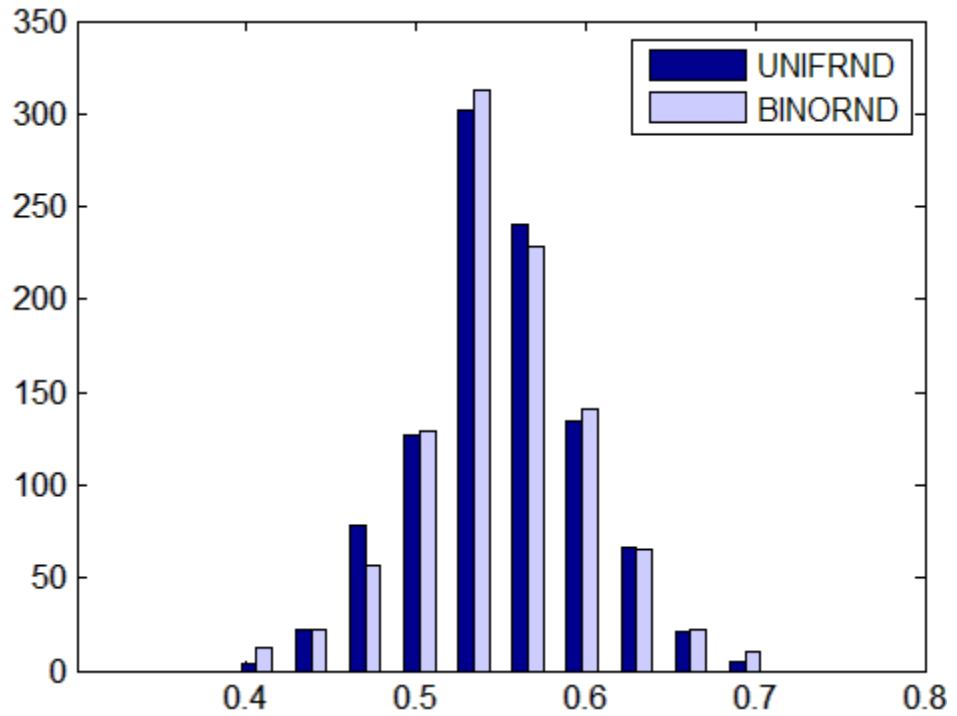
Dependencies of the Random Number Generators

The following table lists the dependencies of Statistics Toolbox RNGs on the MATLAB base RNGs rand and/or randn. Set the states and methods of the RNGs in the right column to assure reproducibility/variability of the outputs of the RNGs in the left 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

Introduction

This section describes Statistics Toolbox GUIs that provide convenient, interactive access to the distribution functions described in "Distribution Functions" on page 5-9.

Distribution Function Tool

To interactively see the influence of parameter changes on the shapes of the pdfs and cdfs of supported Statistics Toolbox distributions, 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.

Choose distribution

Function type (cdf or pdf)

Function plot

Function value

Draggable reference lines

Parameter bounds

Parameter value

Parameter control

Additional parameters
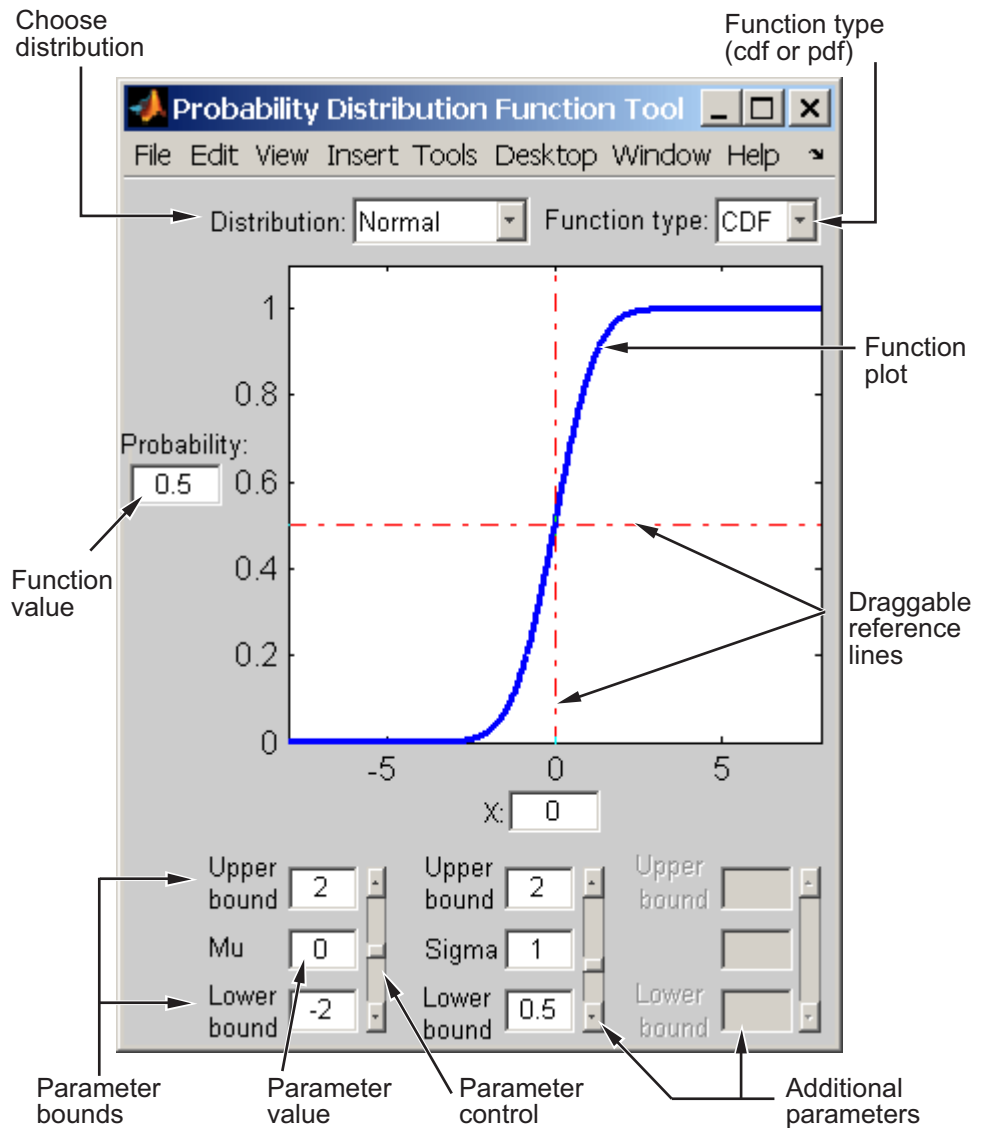
Start by selecting a distribution. Then choose the function type: probability density function (pdf) or cumulative distribution function (cdf).

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)

Task buttons

Import data
from workspace

Create a new fit

Manage multiple fits

Evaluate distribution
at selected points

Exclude data
from fit
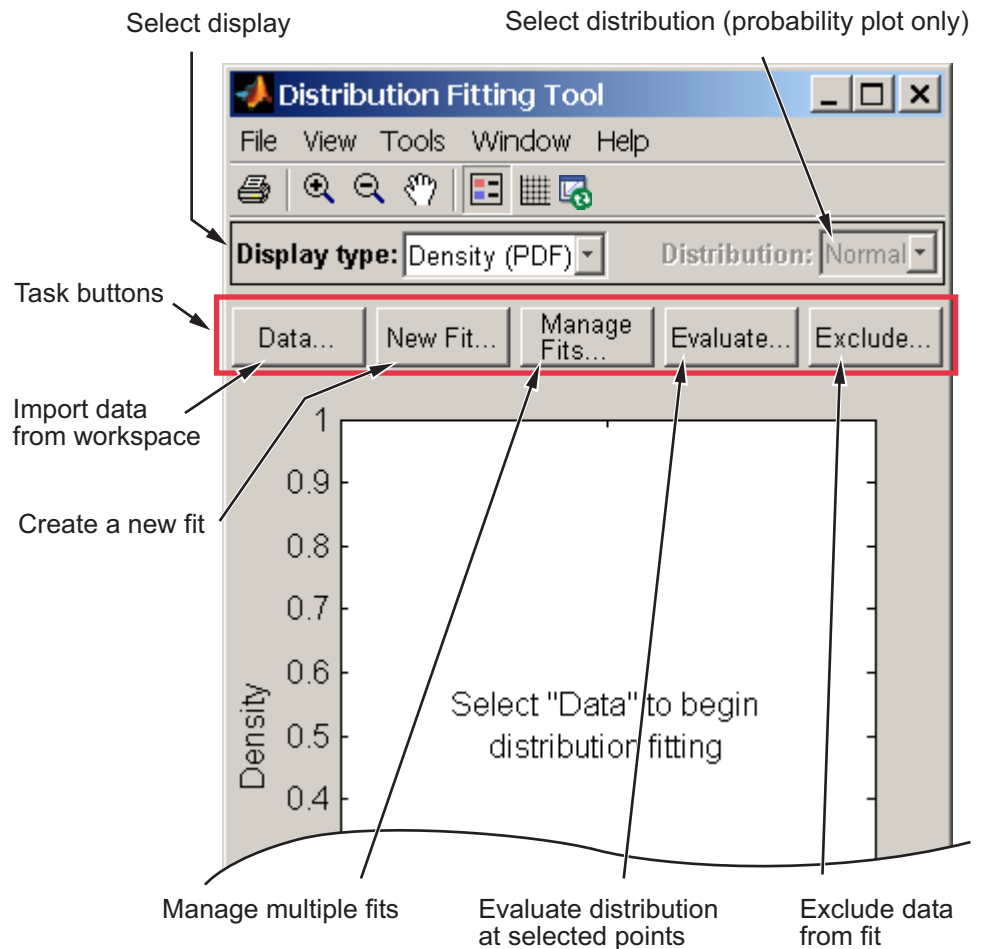
Plot Buttons. Buttons at the top of the tool allow you to adjust the plot displayed in the main window:

- — 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 5-54.

- **New Fit** — Create new fits. See "Creating a New Fit" on page 5-59.

- **Manage Fits** — Manage existing fits. See "Managing Fits" on page 5-66.

- **Evaluate** — Evaluate fits at any points you choose. See "Evaluating Fits" on page 5-68.

- **Exclude** — Create rules specifying which values to exclude when fitting a distribution. See "Excluding Data" on page 5-71.

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 5-77.

- 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 5-78.

- Define and import custom distributions. See "Using Custom Distributions" on page 5-80.

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 5-47

- "Import Data into the Distribution Fitting Tool" on page 5-47

- "Create a New Fit" on page 5-50

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.

5-47

Select data   Enter name for data set
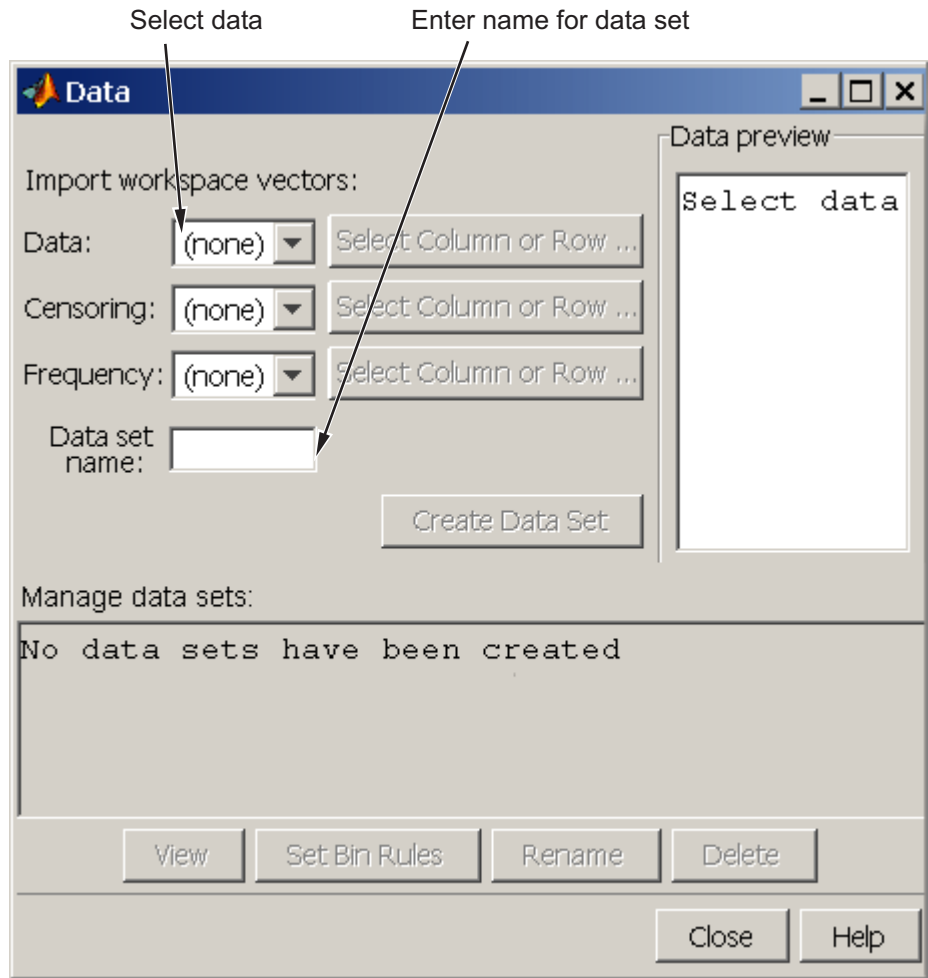
The **Data** field displays all numeric arrays in the MATLAB workspace. Select
data from the drop-down list, as shown in the following figure.
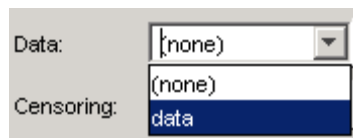
This displays a histogram of the data in the **Data preview** pane.

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.
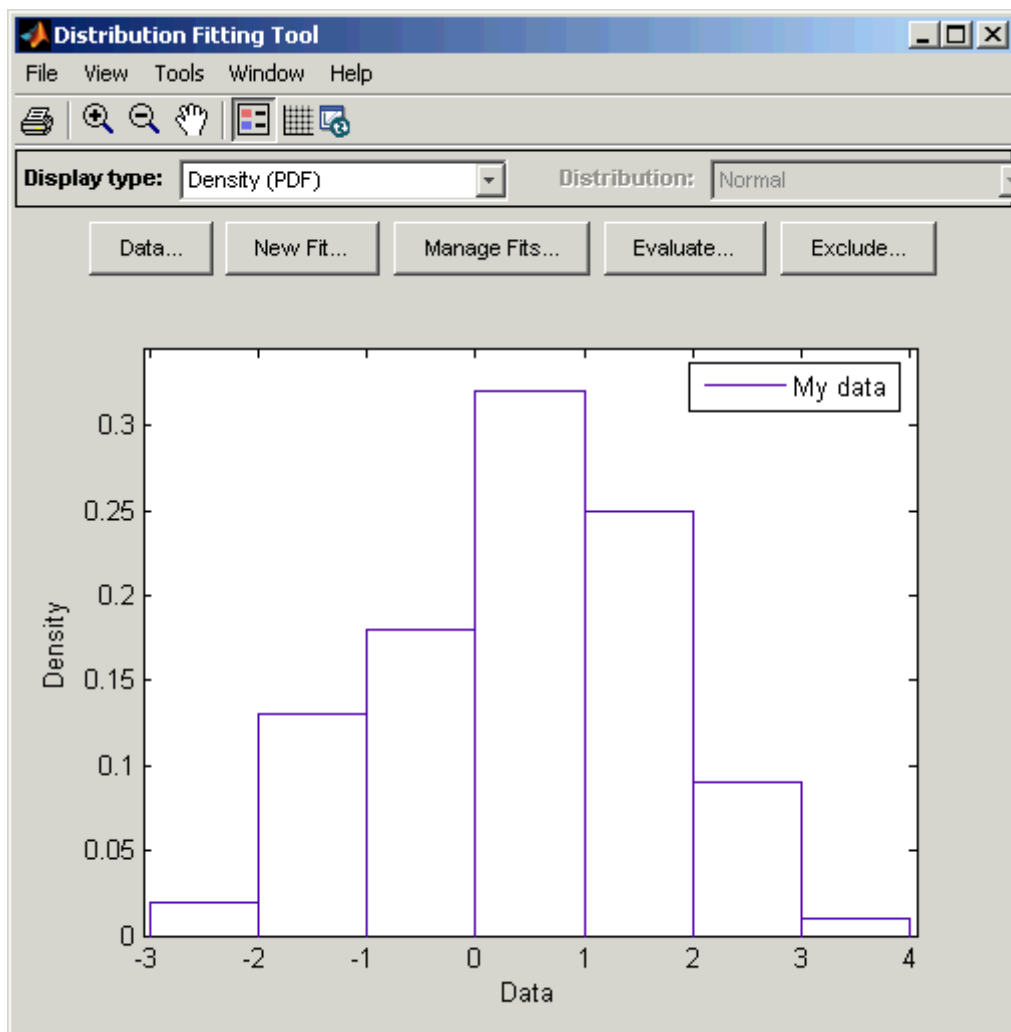
> **Note** Because the example uses random data, you might see a slightly different histogram if you try this example for yourself.

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.

Select data set name ⎯⎯⎯⎯⎯⎯⎯   ⎯⎯⎯⎯⎯ Specify distribution type
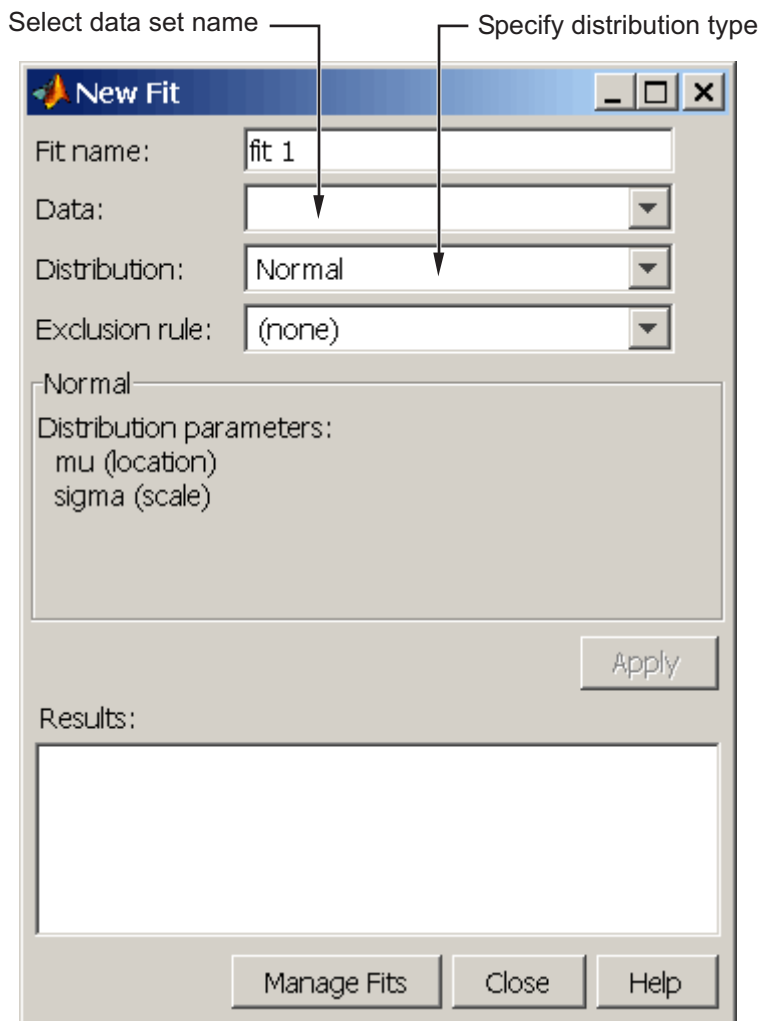
To fit a normal distribution, the default entry of the **Distribution** field, to My data:

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.
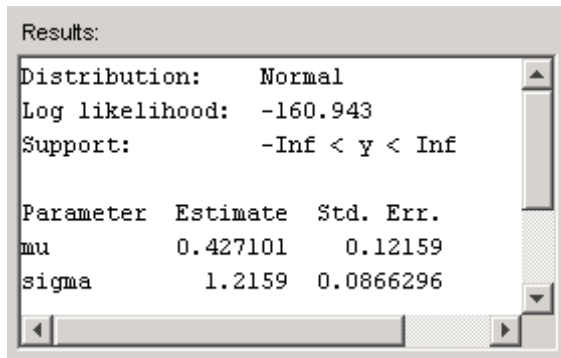
The main window of the Distribution Fitting Tool displays a plot of the normal distribution with this mean and standard deviation, as shown in the following figure.
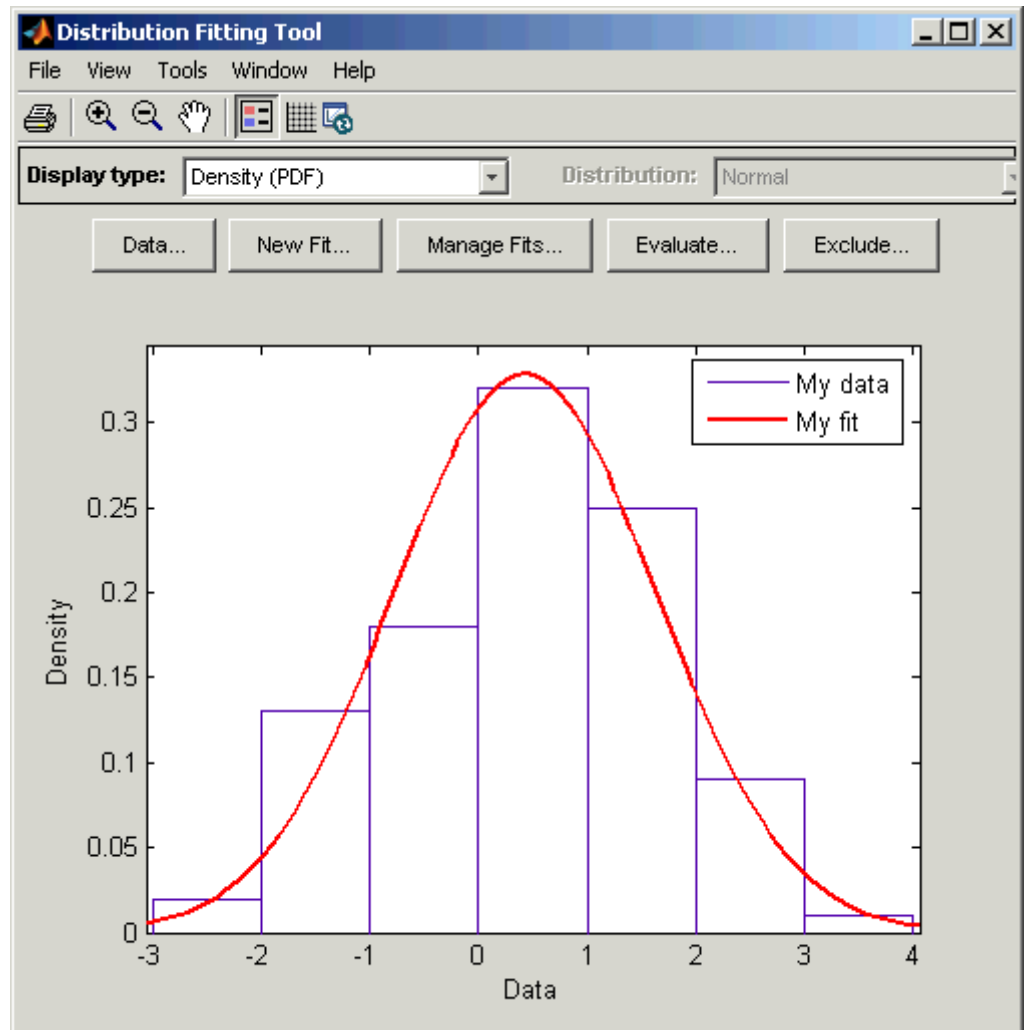
Creating and Managing Data Sets

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.
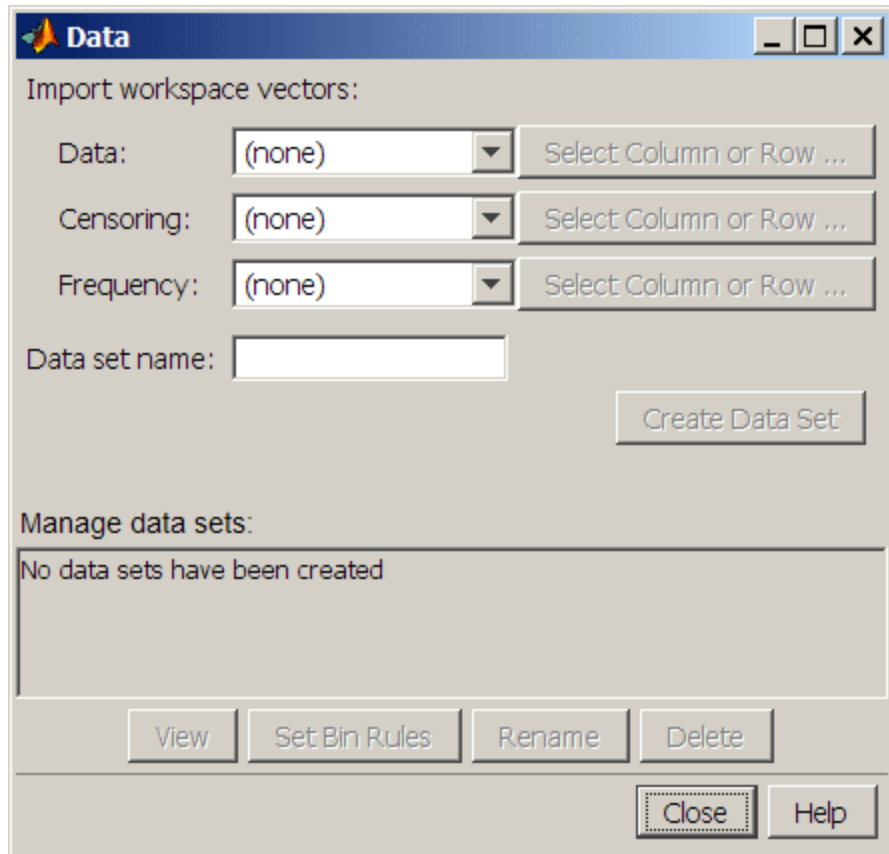
Importing Data. The **Import workspace vectors** pane enables you to create a data set by importing a vector from the MATLAB workspace. The following sections describe the fields of the **Import workspace vectors** pane and give appropriate values for vectors imported from the MATLAB workspace.

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 Variable 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 5-47, 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.
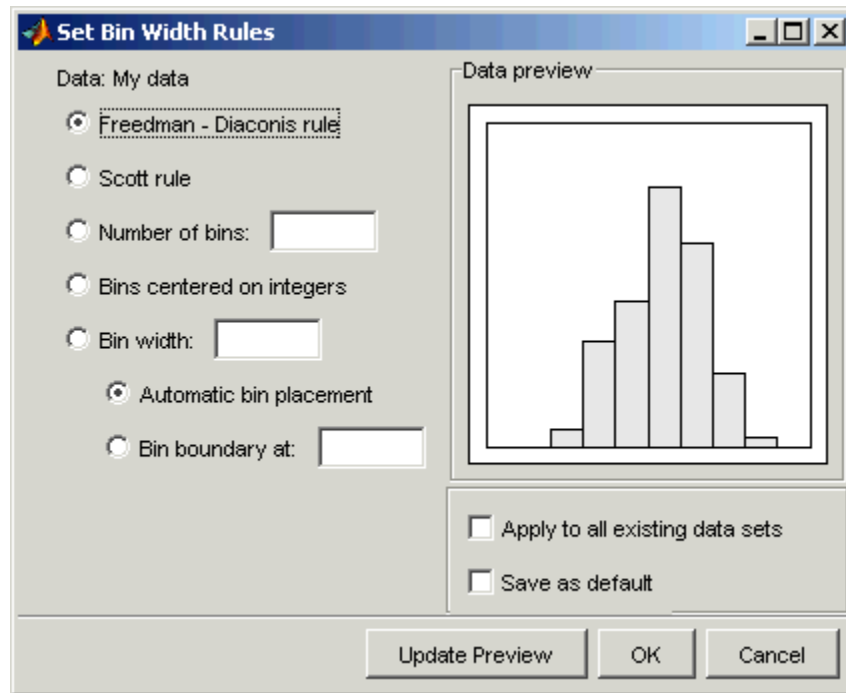
You can select from the following rules:

- **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 5-47, `My data` appears in the **Data** field, as shown in the following figure.

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 5-62 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 5-63.

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 5-71.

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 software (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 Statistics Toolbox fitting functions are described in "Additional Distributions Available in the Distribution Fitting Tool" on page 5-81.

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 5-63 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.

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.
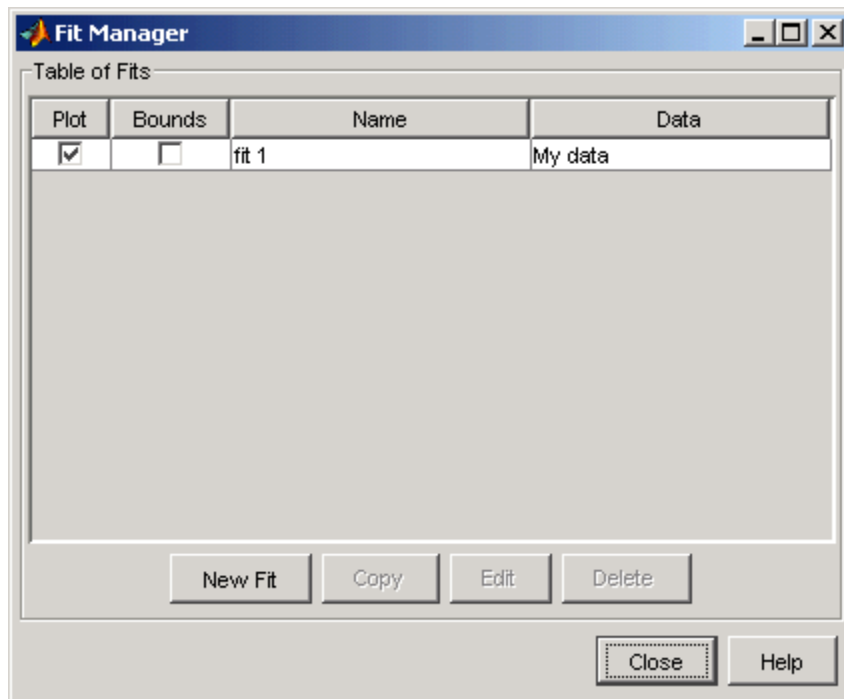
The **Table of fits** displays a list of the fits you create.

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.
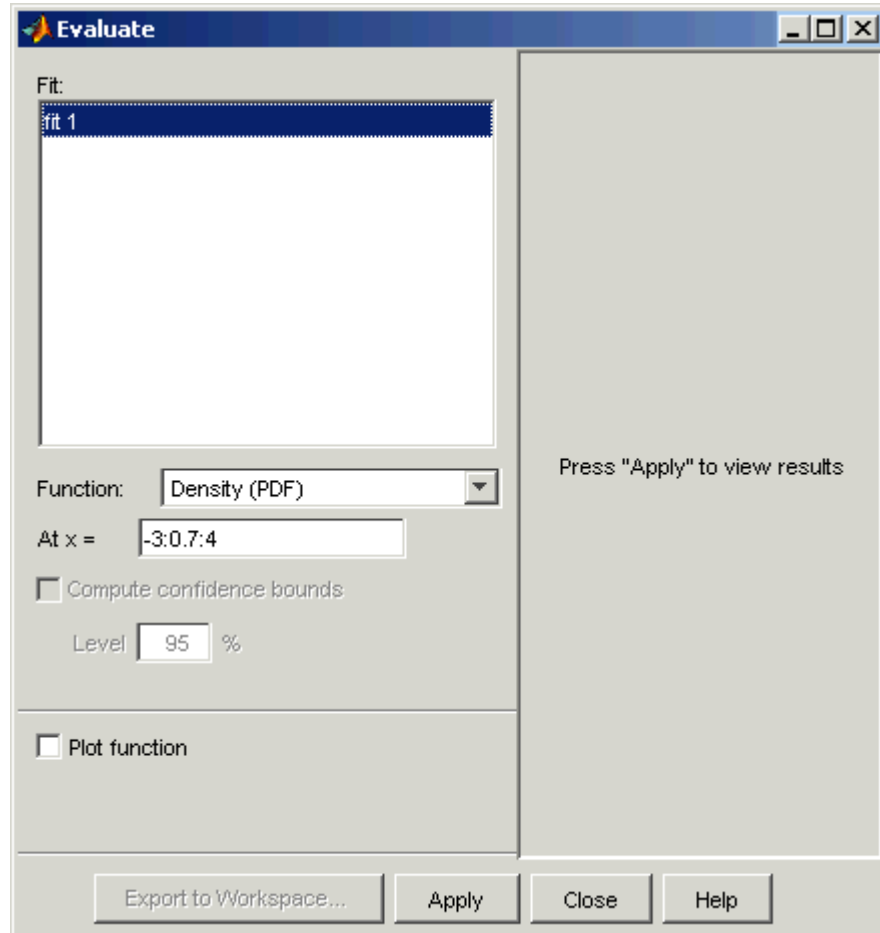
The Evaluate window contains the following items:

- **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 5-47, at the points in the vector `-3:0.5:3`.
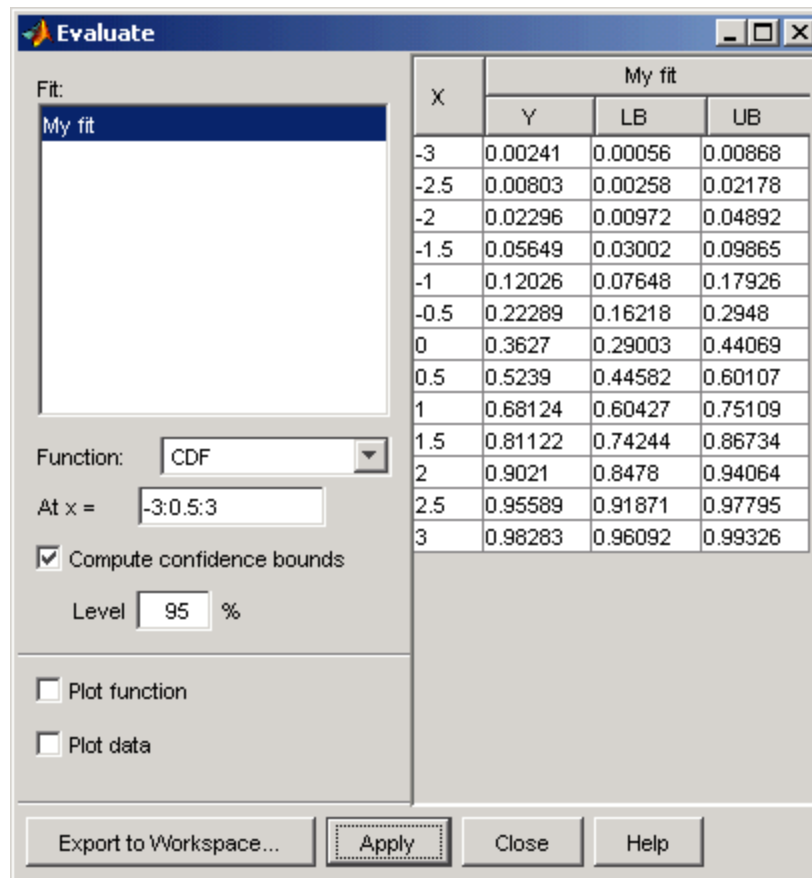
The window displays the following values in the columns of the table to the right of the **Fit** pane:

- 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.
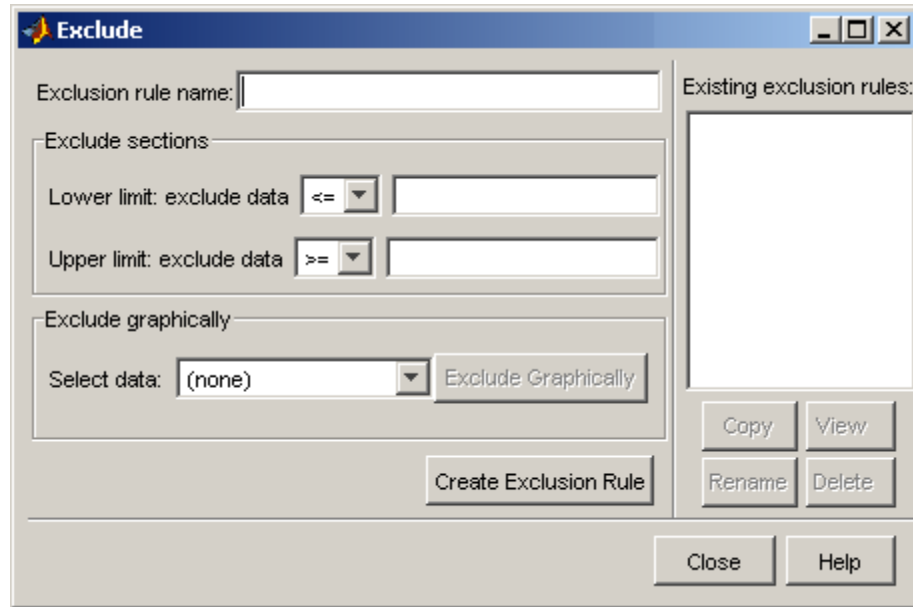
The following sections describe how to create an exclusion rule.

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.
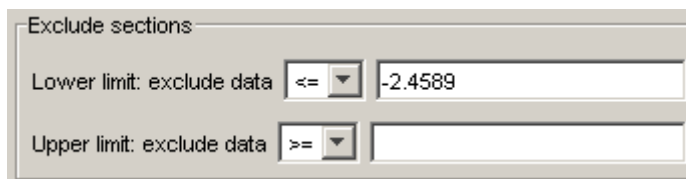
Exclude Graphically. The **Exclude Graphically** button enables you to define the exclusion rule by displaying a plot of the values in a data set and selecting the bounds for the excluded data with the mouse. For example, if you created the data set My data, described in "Creating and Managing Data Sets" on page 5-54, 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.
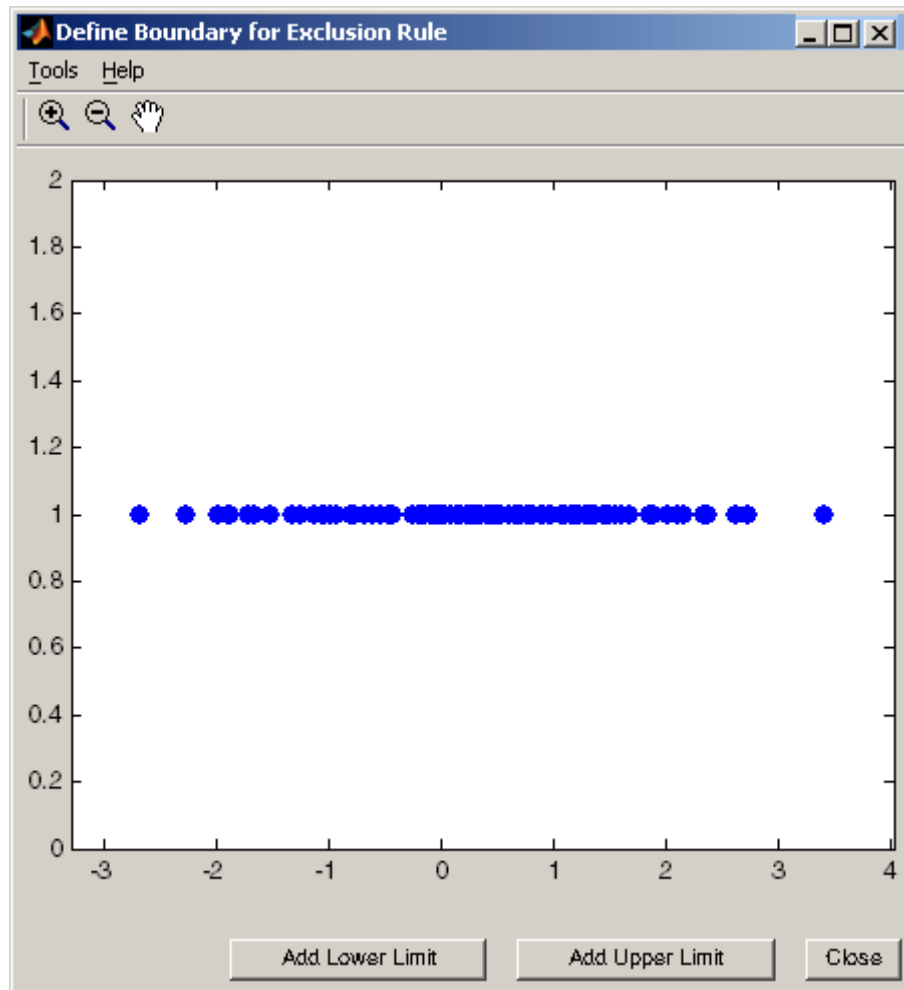
To set a lower limit for the boundary of the excluded region, click **Add Lower Limit**. This displays a vertical line on the left side of the plot window. Move the line with the mouse to the point you where you want the lower limit, as shown in the following figure.
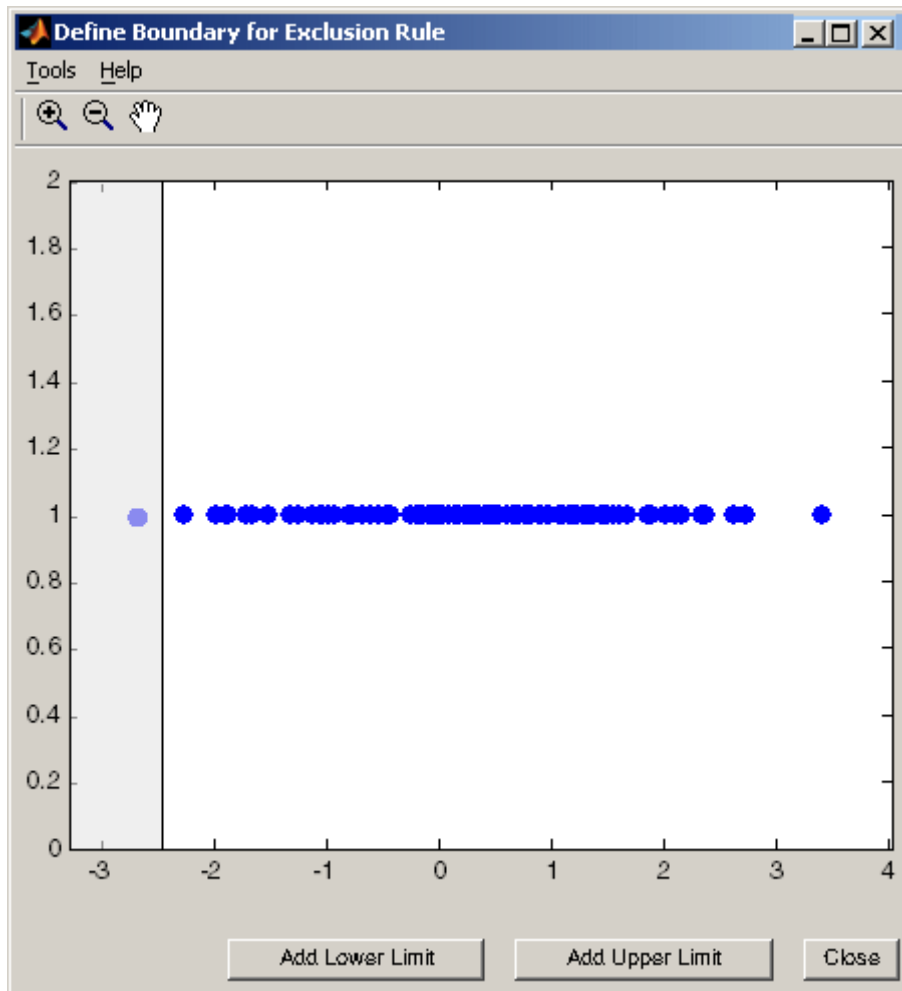
Moving the vertical line changes the value displayed in the **Lower limit: exclude data** field in the Exclude window, as shown in the following figure.

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.
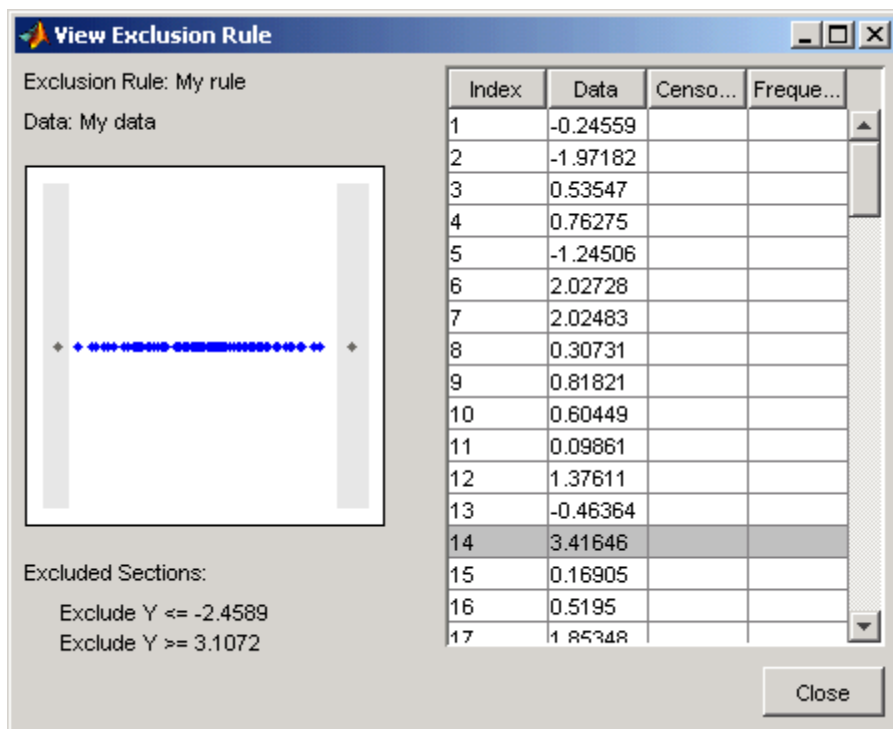
The shaded areas in the plot graphically display which data points are excluded. The table to the right lists all data points. The shaded rows indicate excluded points:

- **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 5-59, 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.
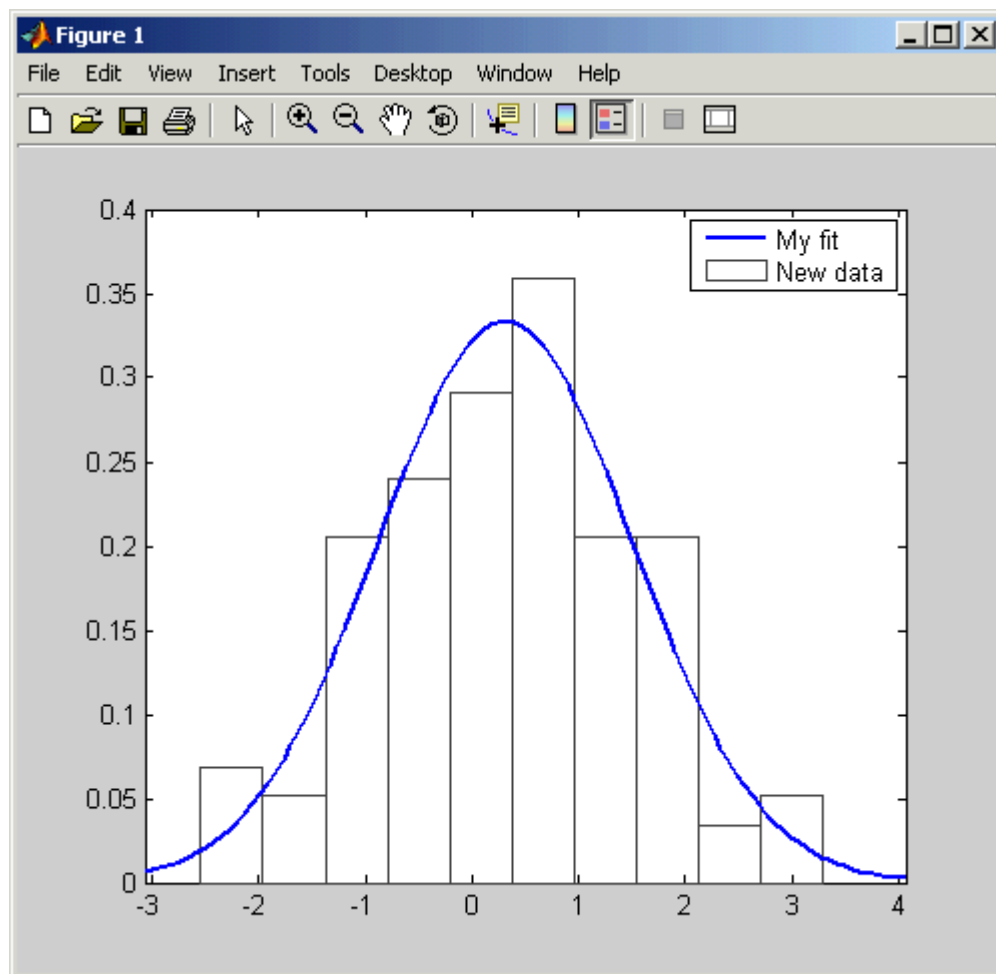
Note By default, the M-file labels the data in the legend using the same name as the data set in the Distribution Fitting Tool. You can change the label using the legend command, as illustrated by the preceding example.

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 5-9. 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 B-10
- "Inverse Gaussian Distribution" on page B-42
- "Loglogistic Distribution" on page B-46
- "Logistic Distribution" on page B-45
- "Nakagami Distribution" on page B-63
- "Rician Distribution" on page B-85
- "t Location-Scale Distribution" on page B-88

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.
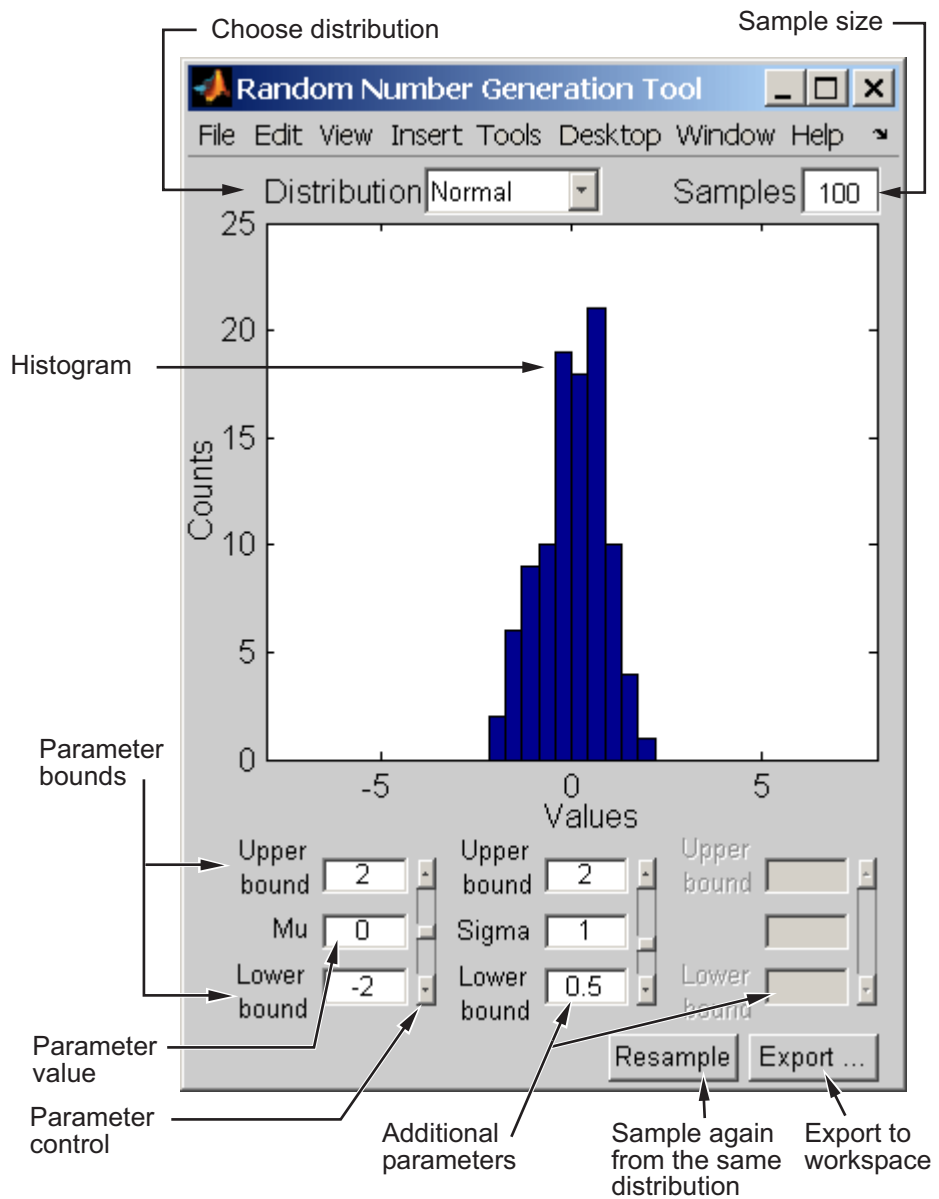
Start by selecting a distribution, then enter the desired sample size.

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.

Pearson and Johnson Systems

Introduction

As described in "Introduction" on page 5-2, choosing an appropriate parametric family of distributions to model your data can be based on *a priori* or *a posteriori* knowledge of the data-producing process, but the choice is often difficult. The *Pearson and Johnson systems* can make such a choice unnecessary. Each system 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.

The following parameters define each member of the Pearson and Johnson systems

- Mean — Estimated by `mean`

- Standard deviation — Estimated by `std`

- Skewness — Estimated by `skewness`

- Kurtosis — Estimated by `kurtosis`

These statistics can also be computed with the `moment` function. The Johnson system, while based on these four parameters, is more naturally described using quantiles, estimated by the `quantile` function.

The Statistics Toolbox functions `pearsrnd` and `johnsrnd` take input arguments defining a distribution (parameters or quantiles, respectively) and return the type and the coefficients of the distribution in the corresponding system. Both functions also generate random numbers from the specified distribution.

As an example, load the data in `carbig.mat`, which includes a variable `MPG` containing measurements of the gas mileage for each car.

```
load carbig
MPG = MPG(~isnan(MPG));
[n,x] = hist(MPG,15);
bar(x,n)
set(get(gca,'Children'),'FaceColor',[.8 .8 1])
```
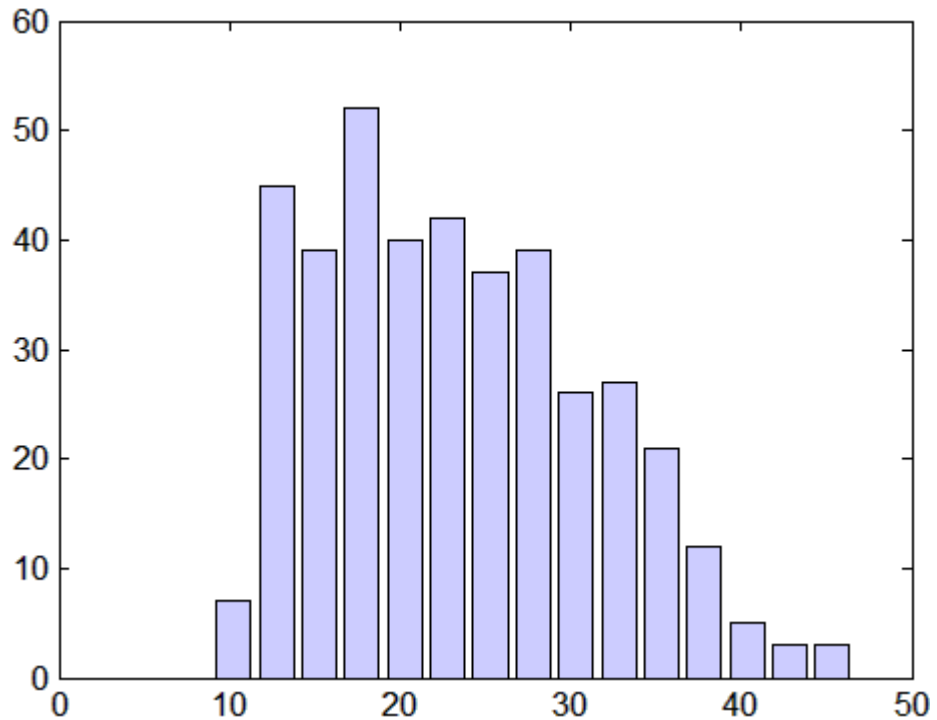

The following two sections model the distribution with members of the Pearson and Johnson systems, respectively.

Pearson Systems

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
```
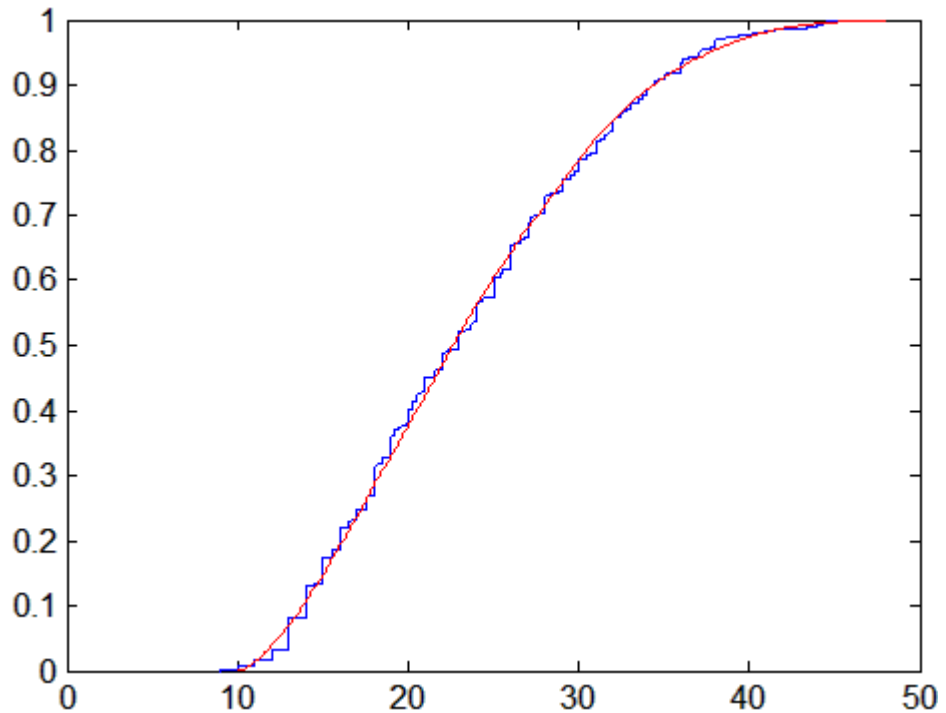
Johnson Systems

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
```
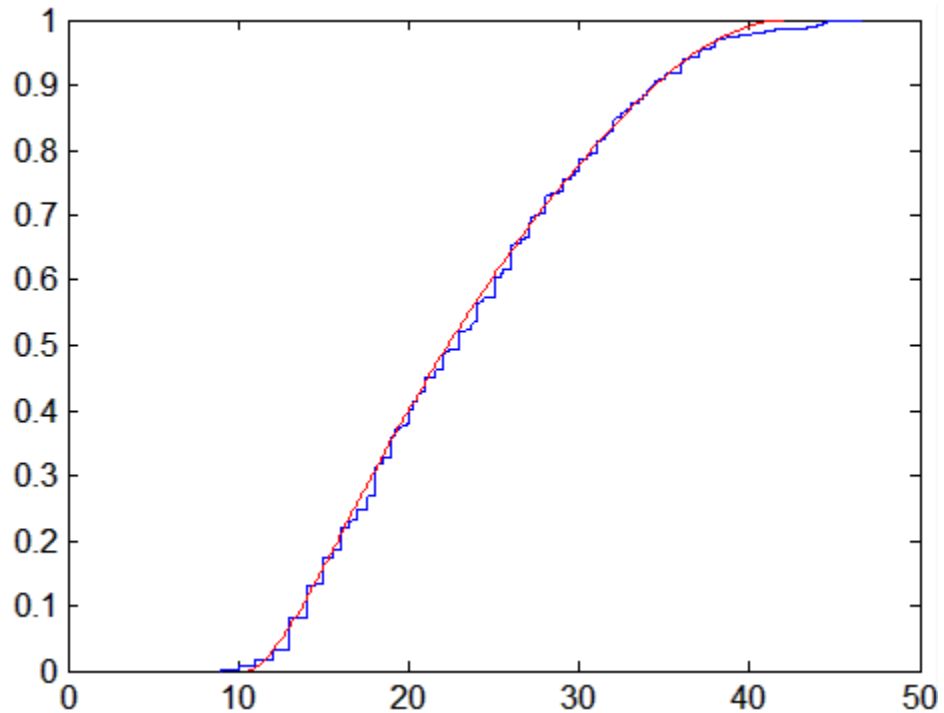
In some applications, it may be important to match the quantiles better in some regions of the data than in others. To do that, specify four evenly spaced standard normal quantiles at which you want to match the data, instead of the default -1.5, -0.5, 0.5, and 1.5. For example, you might care more about matching the data in the right tail than in the left, and so you 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);
```

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
```
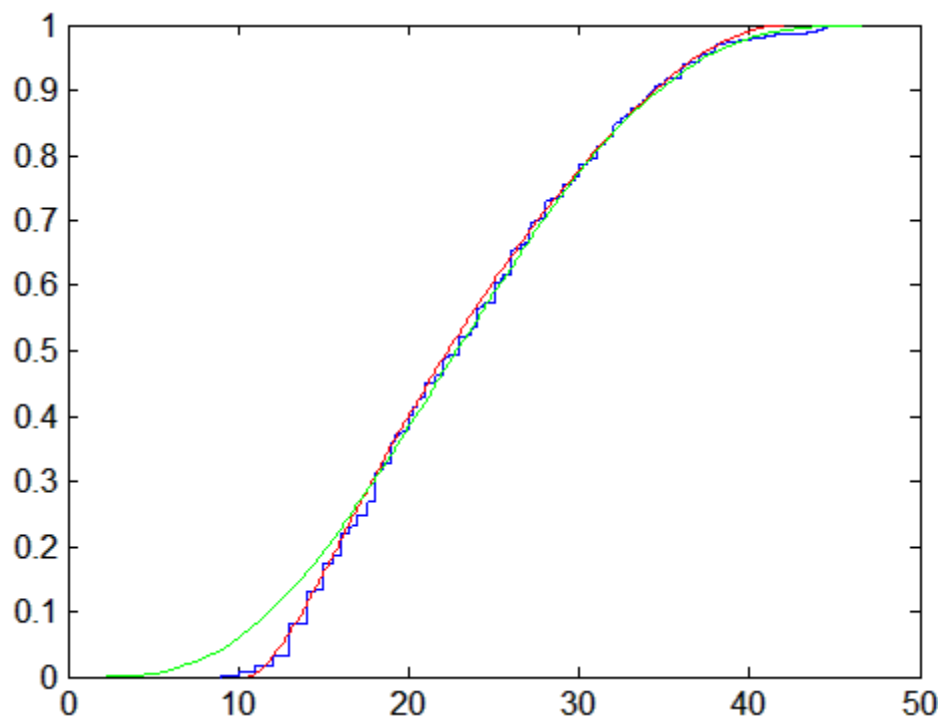
Multivariate Modeling

In this section...
"Gaussian Mixture Models" on page 5-91
"Copulas" on page 5-99

Gaussian Mixture Models

- "Introduction" on page 5-91
- "Creating Gaussian Mixture Models" on page 5-91
- "Simulating Gaussian Mixtures" on page 5-97

Introduction

Gaussian mixture models are formed by combining multivariate normal density components. For information on individual multivariate normal densities, see "Multivariate Normal Distribution" on page B-53 and related distribution functions listed under "Multivariate Distributions" on page 5-8.

In Statistics Toolbox software, mixture models of the `gmdistribution` class are fit to data using an expectation maximization (EM) algorithm, which assigns posterior probabilities to each component density with respect to each observation. The fitting method uses an iterative algorithm that converges to a local optimum.

Clustering with Gaussian mixture models is described in "Gaussian Mixture Models" on page 10-28. This section describes their creation.

Creating Gaussian Mixture Models

- "Specifying a Model" on page 5-92
- "Fitting a Model to Data" on page 5-94

Specifying a Model. Use the `gmdistribution` constructor to create Gaussian mixture models with specified means, covariances, and mixture proportions. The following creates an object of the `gmdistribution` class defining a two-component mixture of bivariate Gaussian distributions:

```
MU = [1 2;-3 -5]; % Means
SIGMA = cat(3,[2 0;0 .5],[1 0;0 1]); % Covariances
p = ones(1,2)/2; % Mixing proportions

obj = gmdistribution(MU,SIGMA,p);
```

Display properties of the object with the MATLAB function `fieldnames`:

```
properties = fieldnames(obj)
properties =
    'NDimensions'
    'DistName'
    'NComponents'
    'PComponents'
    'mu'
    'Sigma'
    'NlogL'
    'AIC'
    'BIC'
    'Converged'
    'Iters'
    'SharedCov'
    'CovType'
    'RegV'
```

The `gmdistribution` reference page describes these properties. To access the value of a property, use dot indexing:

```
dimension = obj.NDimensions
dimension =
    2

name = obj.DistName
name =
gaussian mixture distribution
```

Use the methods pdf and cdf to compute values and visualize the object:

```
ezsurf(@(x,y)pdf(obj,[x y]),[-10 10],[-10 10])
```


pdf(obj,[x,y])

```
ezsurf(@(x,y)cdf(obj,[x y]),[-10 10],[-10 10])
```

5-93
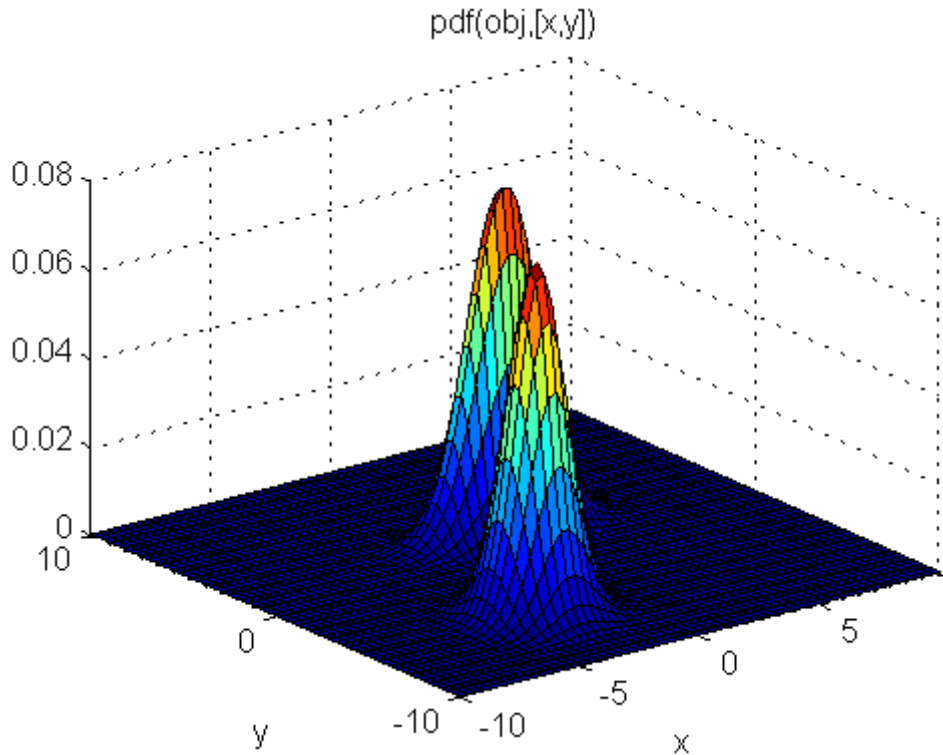
cdf(obj,[x,y])
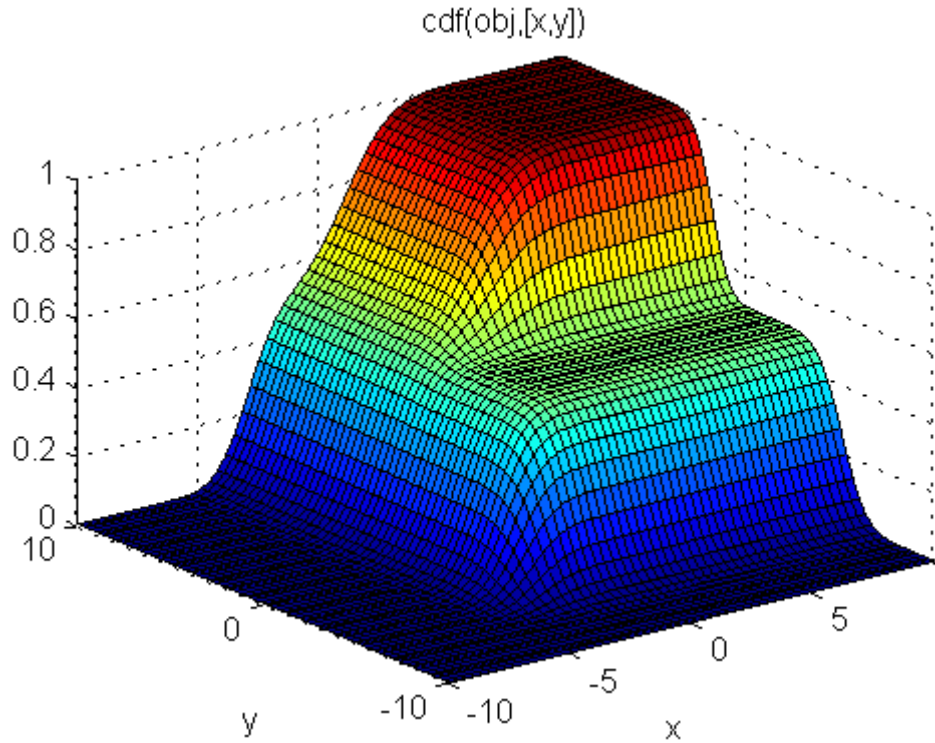
Fitting a Model to Data. You can also create Gaussian mixture models by fitting a parametric model with a specified number of components to data. The fit method of the gmdistribution class uses the syntax obj = gmdistribution.fit(X,k), where X is a data matrix and k is the specified number of components. Choosing a suitable number of components k is essential for creating a useful model of the data—too few components will fail to model the data accurately; too many components will lead to an over-fit model with singular covariance matrices.

The following example illustrates this approach.

First, create some data from a mixture of two bivariate Gaussian distributions using the mvnrnd function:

```
MU1 = [1 2];
SIGMA1 = [2 0; 0 .5];
```

```
MU2 = [-3 -5];
SIGMA2 = [1 0; 0 1];
X = [mvnrnd(MU1,SIGMA1,1000);mvnrnd(MU2,SIGMA2,1000)];

scatter(X(:,1),X(:,2),10,'.')
hold on
```


Next, fit a two-component Gaussian mixture model:

```
options = statset('Display','final');
obj = gmdistribution.fit(X,2,'Options',options);
10 iterations, log-likelihood = -7046.78

h = ezcontour(@(x,y)pdf(obj,[x y]),[-8 6],[-8 6]);
```

Among the properties of the fit are the parameter estimates:

```
ComponentMeans = obj.mu
ComponentMeans =
    0.9391    2.0322
   -2.9823   -4.9737

ComponentCovariances = obj.Sigma
ComponentCovariances(:,:,1) =
    1.7786   -0.0528
   -0.0528    0.5312
ComponentCovariances(:,:,2) =
    1.0491   -0.0150
   -0.0150    0.9816

MixtureProportions = obj.PComponents
MixtureProportions =
```

```
      0.5000     0.5000
```

The Akaike information is minimized by the two-component model:

```
AIC = zeros(1,4);
obj = cell(1,4);
for k = 1:4
    obj{k} = gmdistribution.fit(X,k);
    AIC(k)= obj{k}.AIC;
end

[minAIC,numComponents] = min(AIC);
numComponents
numComponents =
     2

model = obj{2}
model =
Gaussian mixture distribution
with 2 components in 2 dimensions
Component 1:
Mixing proportion: 0.500000
Mean:     0.9391    2.0322
Component 2:
Mixing proportion: 0.500000
Mean:    -2.9823   -4.9737
```

Both the Akaike and Bayes information are negative log-likelihoods for the data with penalty terms for the number of estimated parameters. They are often used to determine an appropriate number of components for a model when the number of components is unspecified.

Simulating Gaussian Mixtures

Use the method `random` of the `gmdistribution` class to generate random data from a Gaussian mixture model created with `gmdistribution` or `fit`.

For example, the following specifies a `gmdistribution` object consisting of a two-component mixture of bivariate Gaussian distributions:

```
MU = [1 2;-3 -5];
```

```
SIGMA = cat(3,[2 0;0 .5],[1 0;0 1]);
p = ones(1,2)/2;
obj = gmdistribution(MU,SIGMA,p);

ezcontour(@(x,y)pdf(obj,[x y]),[-10 10],[-10 10])
hold on
```
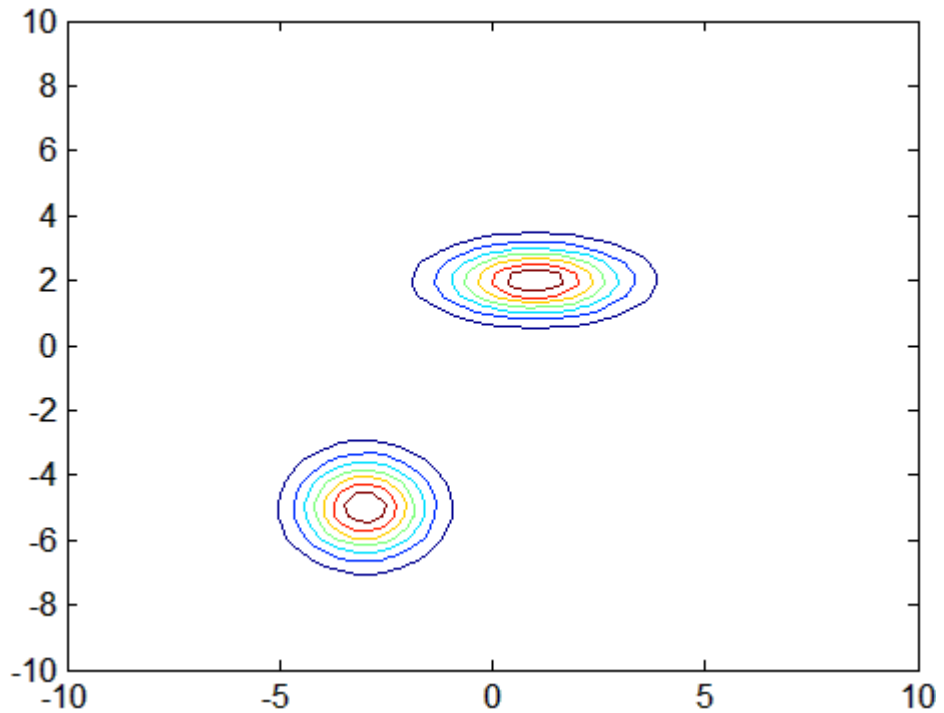

Use random (gmdistribution) to generate 1000 random values:

```
Y = random(obj,1000);
```

```
scatter(Y(:,1),Y(:,2),10,'.')
```
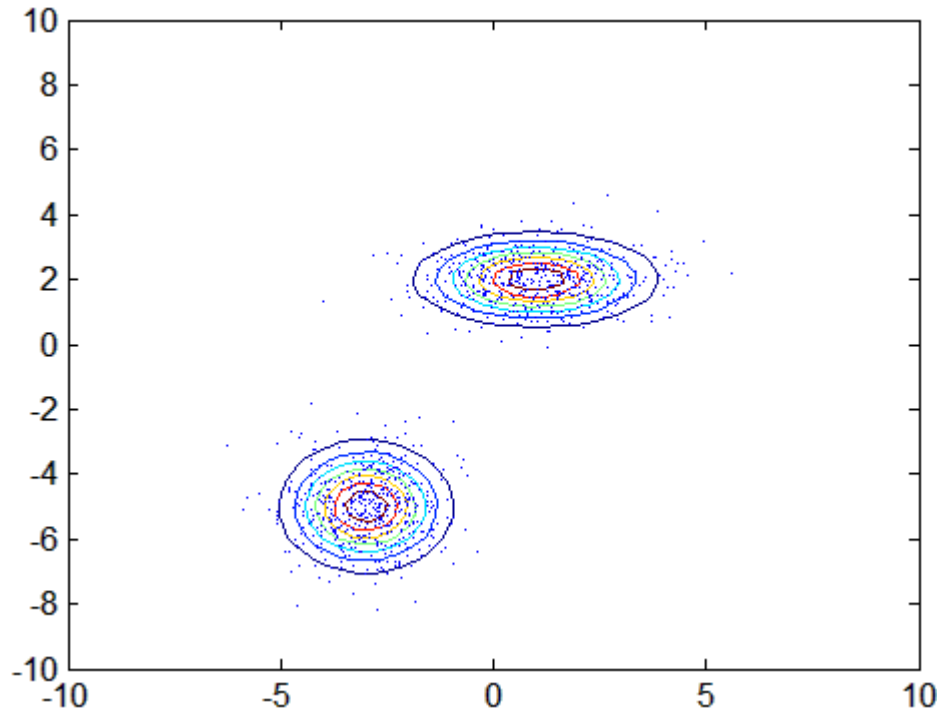
Copulas

- "Introduction" on page 5-100
- "Dependence Between Simulation Inputs" on page 5-100
- "Constructing Dependent Bivariate Distributions" on page 5-104
- "Rank Correlation Coefficients" on page 5-108
- "Bivariate Copulas" on page 5-111
- "Copulas in Higher Dimensions" on page 5-118
- "Archimedean Copulas" on page 5-120
- "Copulas and Nonparametric Marginal Distributions" on page 5-122
- "Fitting Copulas to Data" on page 5-127

Introduction

Copulas are functions that describe dependencies among variables, and provide a way to create distributions that model correlated multivariate data. Using a copula, you can construct a multivariate distribution by specifying marginal univariate distributions, and then choose a copula to provide a correlation structure between variables. Bivariate distributions, as well as distributions in higher dimensions, are possible.

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 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, you might know from real data that the same random conditions affect both sources; ignoring that in the simulation could lead to the wrong conclusions.

The `lognrnd` function simulates independent lognormal random variables. In the following example, the `mvnrnd` function generates n pairs of independent normal random variables, and then exponentiates them. Notice that the covariance matrix used here is diagonal:

```
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([0 5 0 5])
axis equal
xlabel('X1')
ylabel('X2')
```
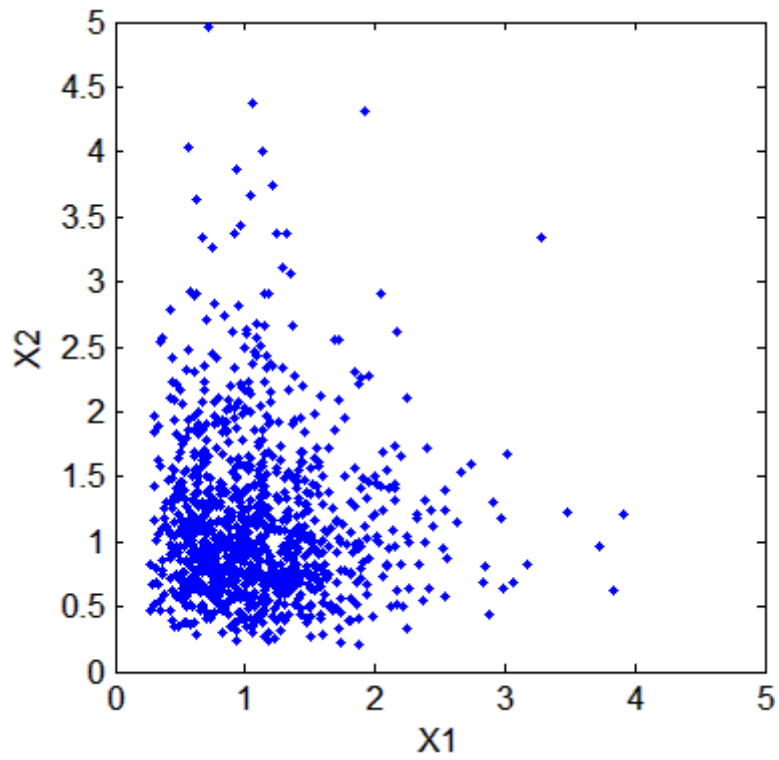
Dependent bivariate lognormal random variables are also easy to generate using a covariance matrix with nonzero off-diagonal terms:

```
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([0 5 0 5])
axis equal
xlabel('X1')
ylabel('X2')
```
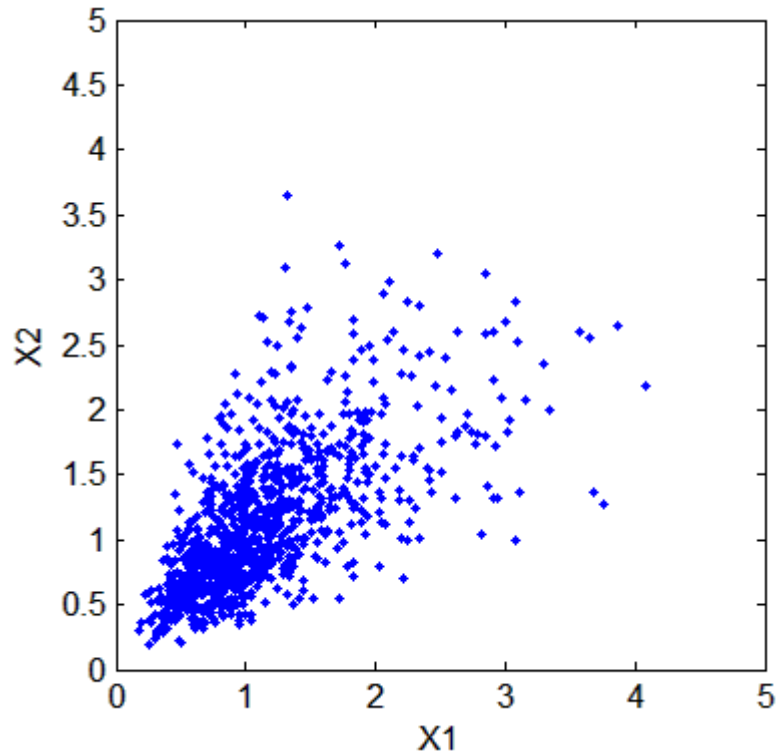
It is clear that there is a tendency in the second data set for large values of X1 to be associated with large values of X2, and similarly for small values. This dependence is determined by the correlation parameter, ρ, of the underlying bivariate normal. The conclusions drawn from the simulation could well depend on whether or not X1 and X2 were generated with dependence. The bivariate lognormal distribution is a simple solution in this case; it 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 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.

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 that is more generally applicable.

1 Generate pairs of values from a bivariate normal distribution. There is statistical dependence between these two variables, and each has a normal marginal distribution.

2 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 can be found, this method can 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

$$\Pr\{U \leq u\} = \Pr\{\Phi(Z) \leq u\} = \Pr(Z \leq \Phi^{-1}(u)\} = u$$

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')
set(get(gca,'Children'),'FaceColor',[.8 .8 1])
```

1000 Simulated N(0,1) Random Values
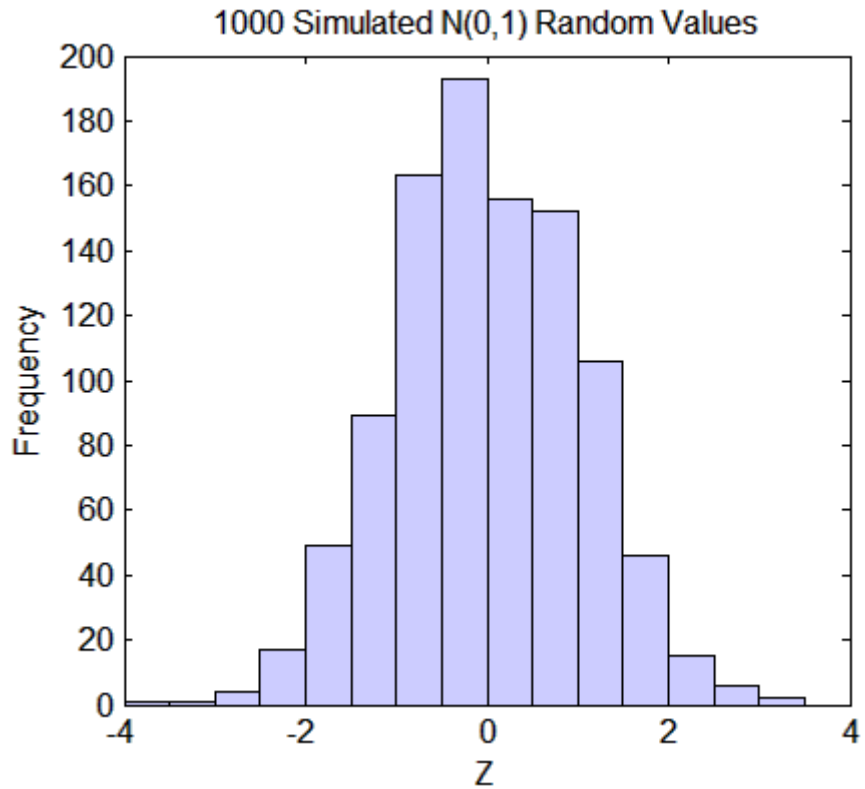

```
u = normcdf(z);

hist(u,.05:.1:.95)
title('1000 Simulated N(0,1) Values Transformed to Unif(0,1)')
xlabel('U')
ylabel('Frequency')
set(get(gca,'Children'),'FaceColor',[.8 .8 1])
```
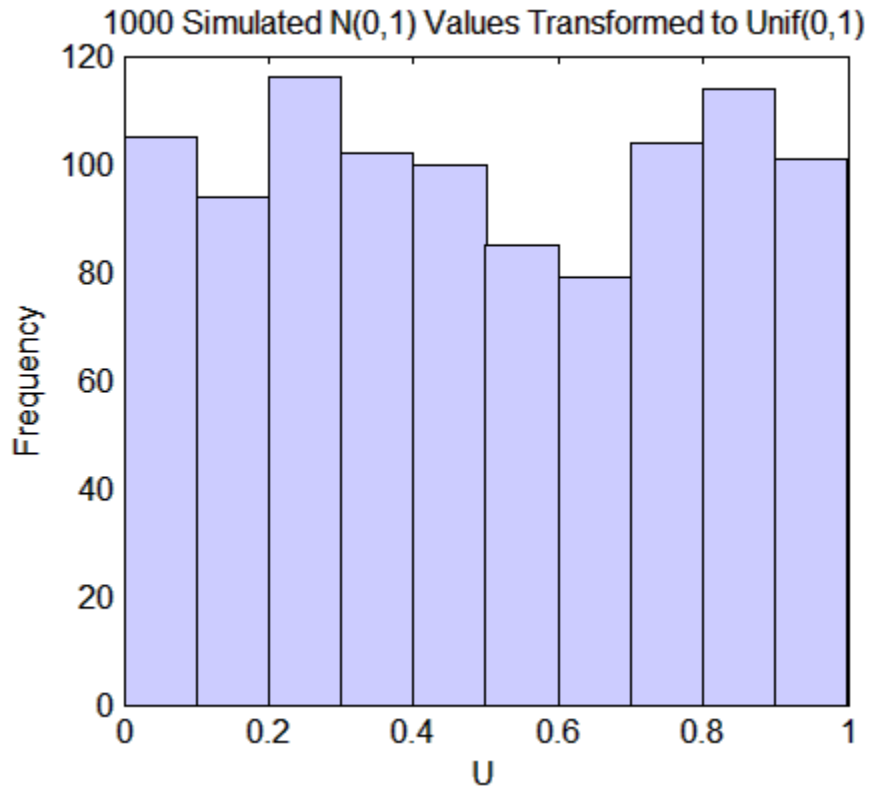
Borrowing from the theory of univariate random number generation, applying the inverse cdf of any distribution, F, to a Unif(0,1) random variable results in a random variable whose distribution is exactly F (see "Inversion Methods" on page 5-135). The proof is essentially the opposite of the preceding 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')
set(get(gca,'Children'),'FaceColor',[.8 .8 1])
```

1000 Simulated N(0,1) Values Transformed to Gamma(2,1)
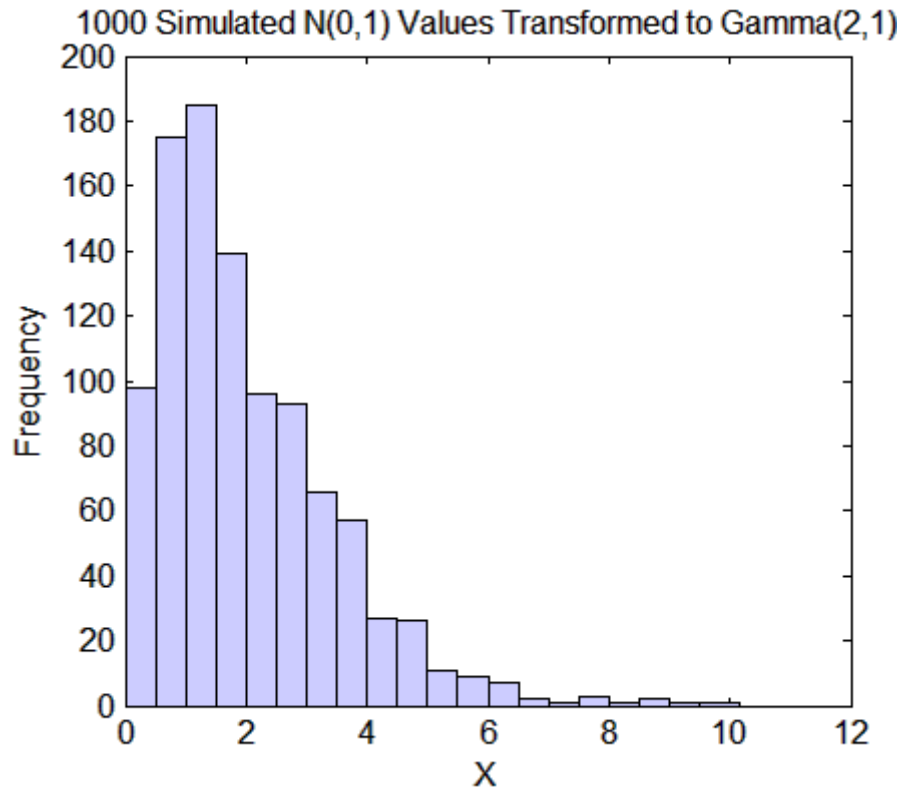
This two-step transformation can be applied to each variable of a standard bivariate normal, creating dependent random variables with arbitrary marginal distributions. Because the transformation works on each component separately, the two resulting random variables need not even have the same marginal distributions. The transformation is defined as:

$$Z = [Z_1, Z_2] \sim N\left([0,0], \begin{bmatrix} 1 & \rho \\ \rho & 1 \end{bmatrix}\right)$$

$$U = [\Phi(Z_1), \Phi(Z_2)]$$

$$X = [G_1(U_1), G_2(U_2)]$$

where G_1 and G_2 are inverse cdfs of two possibly different distributions. For example, the following generates 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)];

scatterhist(X(:,1),X(:,2))
```
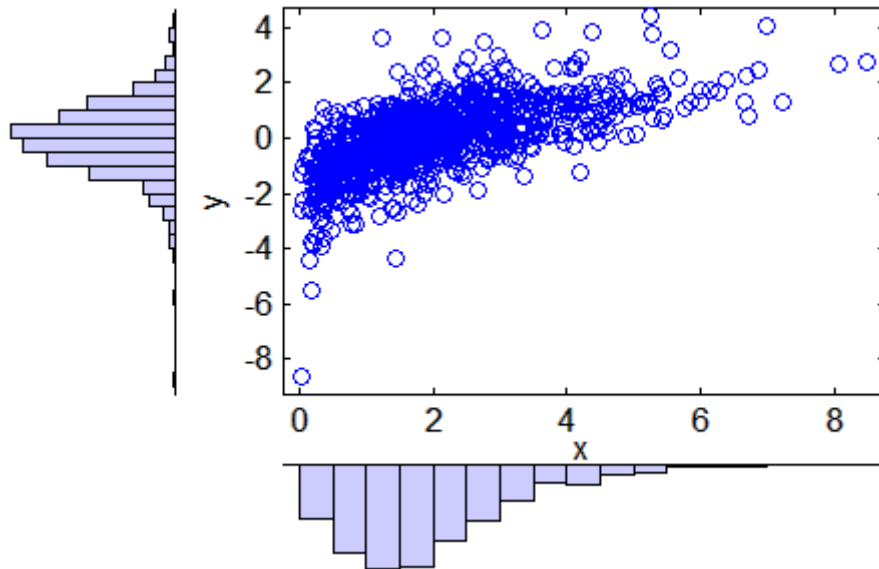

This plot has histograms alongside a scatter plot to show both the marginal distributions, and the dependence.

Rank Correlation Coefficients

Dependence between X1 and X2 in this construction is determined by the correlation parameter, ρ, of the underlying bivariate normal. However, the linear correlation of X1 and X2 is not ρ. For example, in the original lognormal case, a closed form for that correlation is:

$$cor(X1, X2) = \frac{e^{\rho\sigma^2} - 1}{e^{\sigma^2} - 1}$$

which is strictly less than ρ, unless ρ is exactly 1. In more general cases such as the Gamma/t construction, the linear correlation between X1 and X2 is difficult or impossible to express in terms of ρ, but simulations 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.

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:

$$\tau = \frac{2}{\pi}\arcsin(\rho) \quad \text{or} \quad \rho = \sin(\tau\frac{\pi}{2})$$

$$\rho_s = \frac{6}{\pi}\arcsin(\frac{\rho}{2}) \quad \text{or} \quad \rho = 2\sin(\rho_s\frac{\pi}{6})$$

The following plot shows the relationship:

```
rho = -1:.01:1;
tau = 2.*asin(rho)./pi;
```

```
rho_s = 6.*asin(rho./2)./pi;

plot(rho,tau,'b-','LineWidth',2)
hold on
plot(rho,rho_s,'g-','LineWidth',2)
plot([-1 1],[-1 1],'k:','LineWidth',2)
axis([-1 1 -1 1])
xlabel('rho')
ylabel('Rank correlation coefficient')
legend('Kendall''s {\it\tau}', ...
       'Spearman''s {\it\rho_s}', ...
       'location','NW')
```
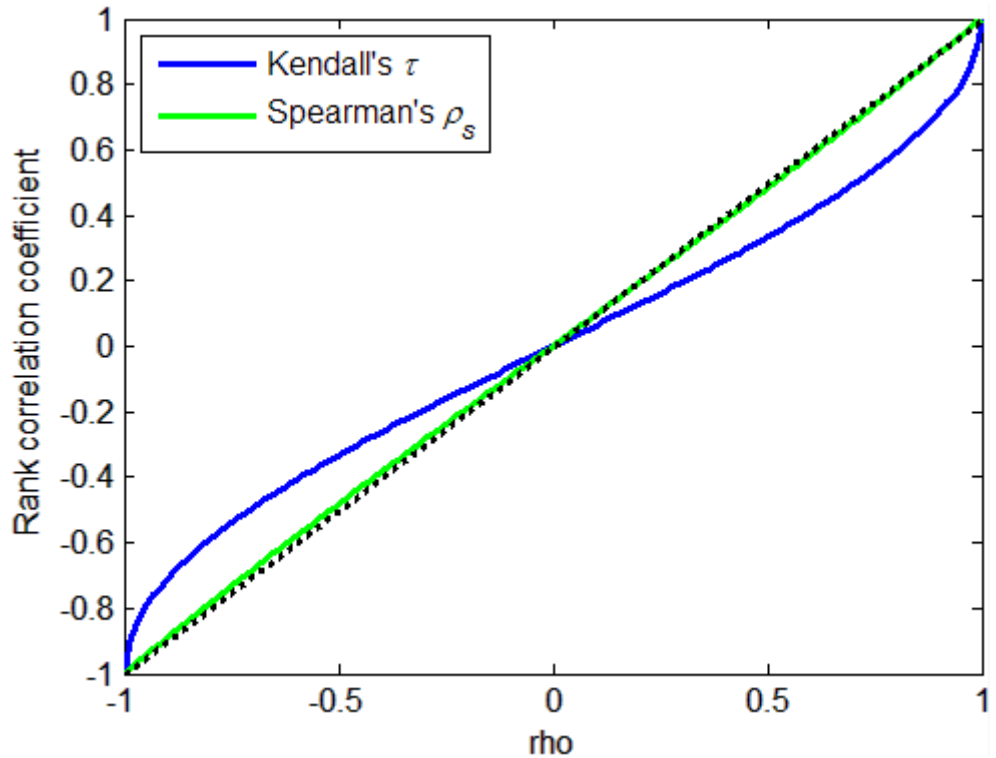
Thus, it is easy to create the desired rank correlation between X1 and X2, regardless of their marginal distributions, by choosing the correct ρ parameter value for the linear correlation between Z1 and Z2.

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.

Bivariate Copulas

The first step of the construction described in the previous section defines what is known as 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 functions compute:

- Probability density functions (`copulapdf`) and the cumulative distribution functions (`copulacdf`) for Gaussian copulas
- Rank correlations from linear correlations (`copulastat`) and vice versa (`copulaparam`)
- Random vectors (`copularnd`)
- Parameters for copulas fit to data (`copulafit`)

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('{\it\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('{\it\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('{\it\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('{\it\rho} = -0.8')
xlabel('U1')
ylabel('U2')
```
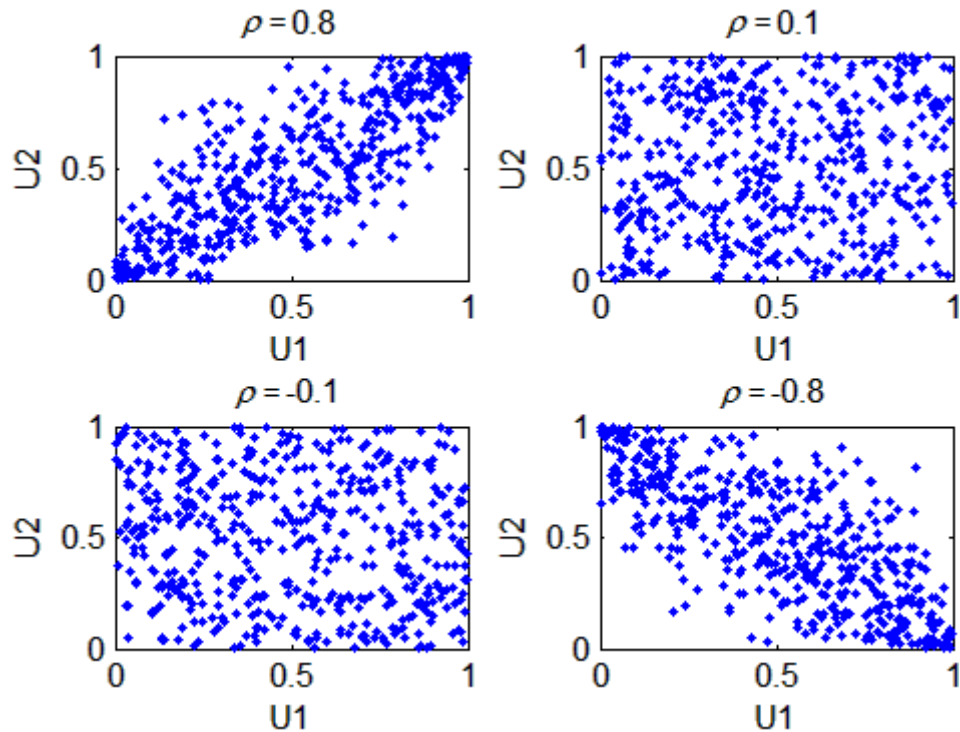
The dependence between U1 and U2 is completely separate from the marginal distributions of X1 = G(U1) and X2 = G(U2). X1 and X2 can be given any marginal distributions, and still have the same rank correlation. This is one of the main appeals of copulas—they allow this separate specification of dependence and marginal distribution. You can also compute the pdf (copulapdf) and the cdf (copulacdf) for a copula. For example, these plots show the pdf and cdf for ρ = .8:

```
u1 = linspace(1e-3,1-1e-3,50);
u2 = linspace(1e-3,1-1e-3,50);
[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')
```
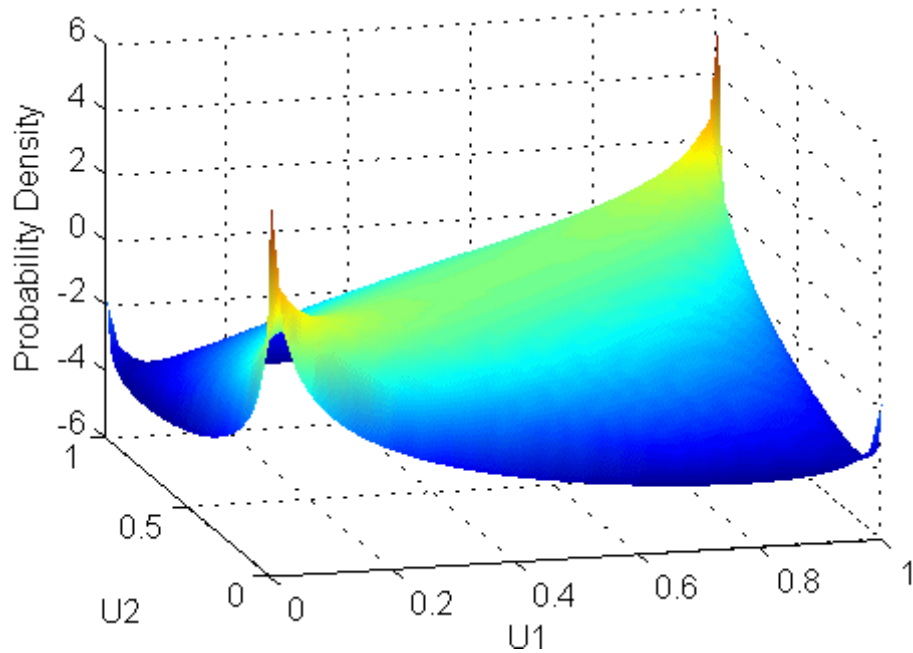


```
u1 = linspace(1e-3,1-1e-3,50);
u2 = linspace(1e-3,1-1e-3,50);
[U1,U2] = meshgrid(u1,u2);
F = copulacdf('t',[U1(:) U2(:)],Rho,5);
F = reshape(F,size(U1));

surf(u1,u2,F,'FaceColor','interp','EdgeColor','none')
view([-15,20])
xlabel('U1')
ylabel('U2')
zlabel('Cumulative Probability')
```
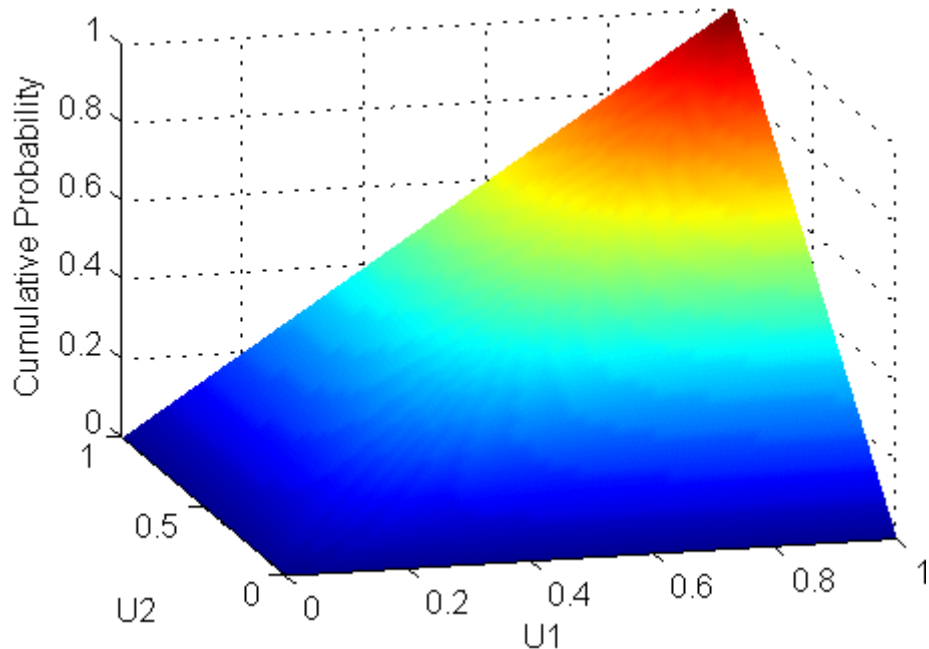
A different family of copulas can be constructed by starting from a bivariate t distribution and transforming using the corresponding t cdf. The bivariate t distribution is parameterized with P, the linear correlation matrix, and v, the degrees of freedom. Thus, for example, you can speak of a t_1 or a t_5 copula, based on the multivariate t with one and five degrees of freedom, respectively.

Just as for Gaussian copulas, Statistics Toolbox functions for t copulas compute:

- Probability density functions (`copulapdf`) and the cumulative distribution functions (`copulacdf`) for Gaussian copulas

- Rank correlations from linear correlations (`copulastat`) and vice versa (`copulaparam`)

- Random vectors (`copularnd`)

- Parameters for copulas fit to data (`copulafit`)

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('{\it\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('{\it\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('{\it\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('{\it\rho} = -0.8')
xlabel('U1')
ylabel('U2')
```
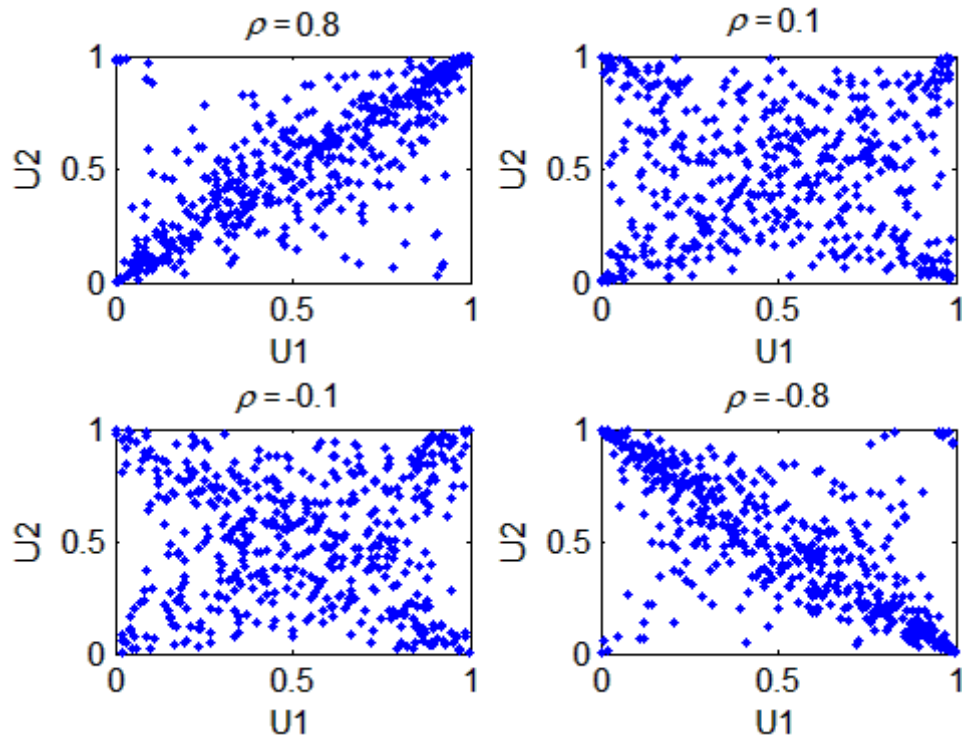
A t copula has uniform marginal distributions for U1 and U2, just as a Gaussian copula does. The rank correlation τ or ρ_s between components in a t copula is also the same function of ρ as for a Gaussian. However, as these plots demonstrate, a t_1 copula differs quite a bit from a Gaussian copula, even when their components have the same rank correlation. The difference is in their dependence structure. Not surprisingly, as the degrees of freedom parameter v is made larger, a t_v 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 using copularnd:

```
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)];

scatterhist(X(:,1),X(:,2))
```
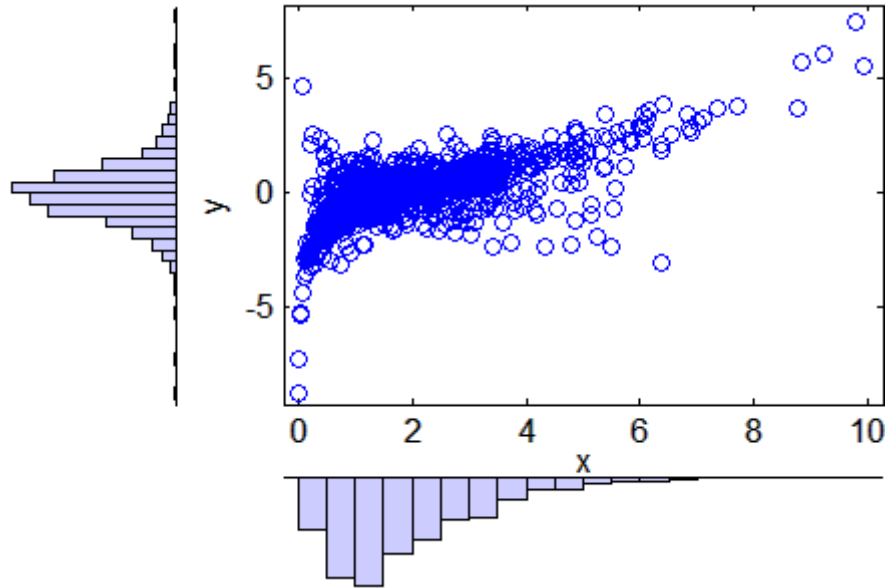

Compared to the bivariate Gamma/t distribution constructed earlier, which was based on a Gaussian copula, the distribution constructed here, based on a t_1 copula, has the same marginal distributions and the same rank correlation between variables but a very different dependence structure. This illustrates 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 and copularnd, 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')
```
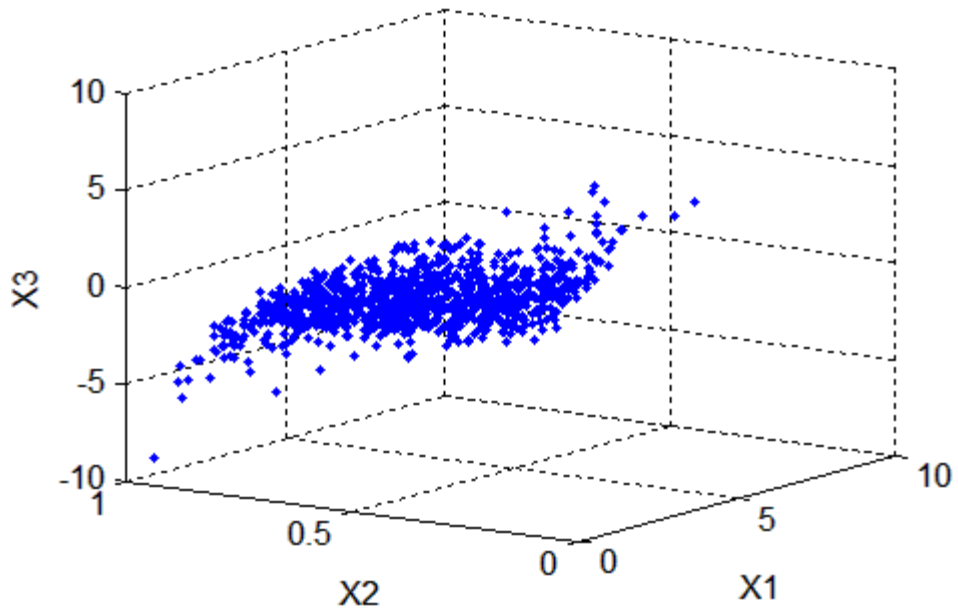

Notice that the relationship between the linear correlation parameter ρ and, for example, Kendall's τ, holds for each entry in the correlation matrix P used here. You can verify that the sample rank correlations of the data are approximately equal to the theoretical values:

```
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 functions are available for three bivariate Archimedean copula families:

- Clayton copulas

- Frank copulas

- Gumbel copulas

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, and 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 with copularnd. Repeat the same procedure for the Frank and Gumbel copulas:

```
n = 500;

U = copularnd('Clayton',alpha,n);
subplot(3,1,1)
plot(U(:,1),U(:,2),'.');
title(['Clayton Copula, {\it\alpha} = ',sprintf('%0.2f',alpha)])
xlabel('U1')
ylabel('U2')

alpha = copulaparam('Frank',tau,'type','kendall');
U = copularnd('Frank',alpha,n);
subplot(3,1,2)
plot(U(:,1),U(:,2),'.')
title(['Frank Copula, {\it\alpha} = ',sprintf('%0.2f',alpha)])
xlabel('U1')
ylabel('U2')

alpha = copulaparam('Gumbel',tau,'type','kendall');
U = copularnd('Gumbel',alpha,n);
subplot(3,1,3)
plot(U(:,1),U(:,2),'.')
title(['Gumbel Copula, {\it\alpha} = ',sprintf('%0.2f',alpha)])
xlabel('U1')
ylabel('U2')
```

Clayton Copula, $\alpha = 2.88$

Frank Copula, $\alpha = 7.68$

Gumbel Copula, $\alpha = 2.44$
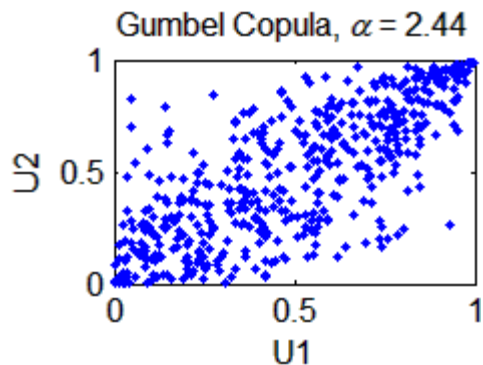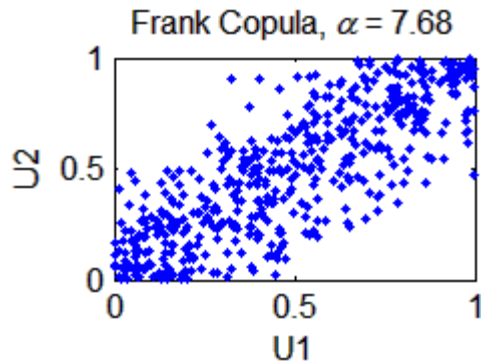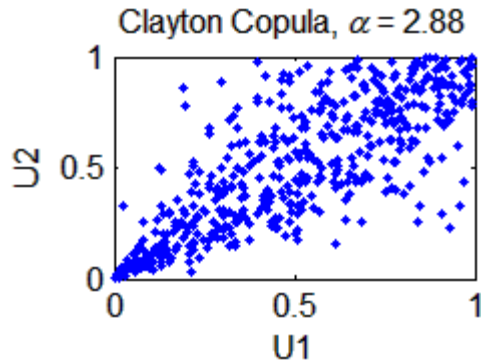
Copulas and Nonparametric Marginal Distributions

To simulate dependent multivariate data using a copula, you must specify each of the following:

- The copula family (and any shape parameters)

- The rank correlations among variables

- Marginal distributions for each variable

Suppose you have return data for two stocks and want 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')
set(get(gca,'Children'),'FaceColor',[.8 .8 1])

subplot(2,1,2)
hist(stocks(:,2),10)
xlim([-3.5 3.5])
xlabel('X2')
ylabel('Frequency')
set(get(gca,'Children'),'FaceColor',[.8 .8 1])
```
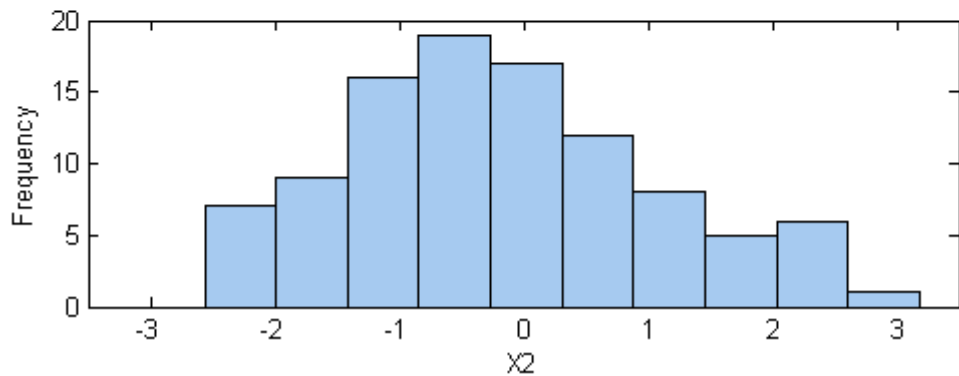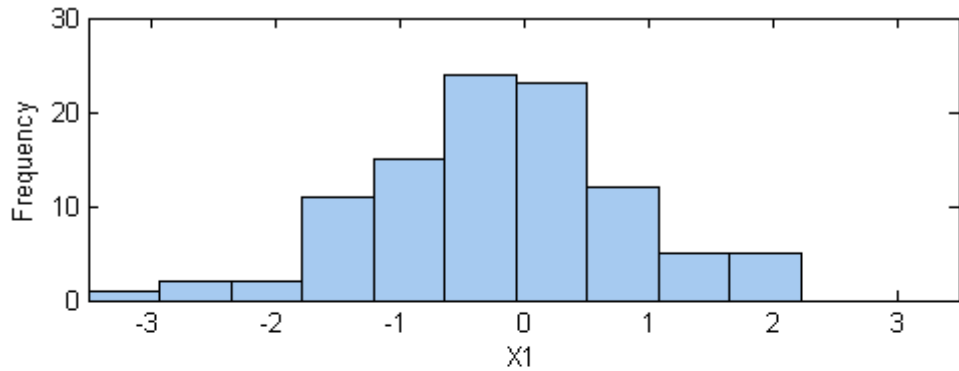
You could fit a parametric model separately to each dataset, and use those estimates as the marginal distributions. However, a parametric model may not be sufficiently flexible. Instead, you can use a nonparametric model to transform to the marginal distributions. All that is needed is a way to compute 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, 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 with `ksdensity`. For example, compare the empirical cdf to a kernel smoothed cdf estimate for the first variable:

```
[Fi,xi] = ecdf(stocks(:,1));

stairs(xi,Fi,'b','LineWidth',2)
hold on

Fi_sm = ksdensity(stocks(:,1),xi,'function','cdf','width',.15);

plot(xi,Fi_sm,'r-','LineWidth',1.5)
xlabel('X1')
ylabel('Cumulative Probability')
legend('Empirical','Smoothed','Location','NW')
grid on
```
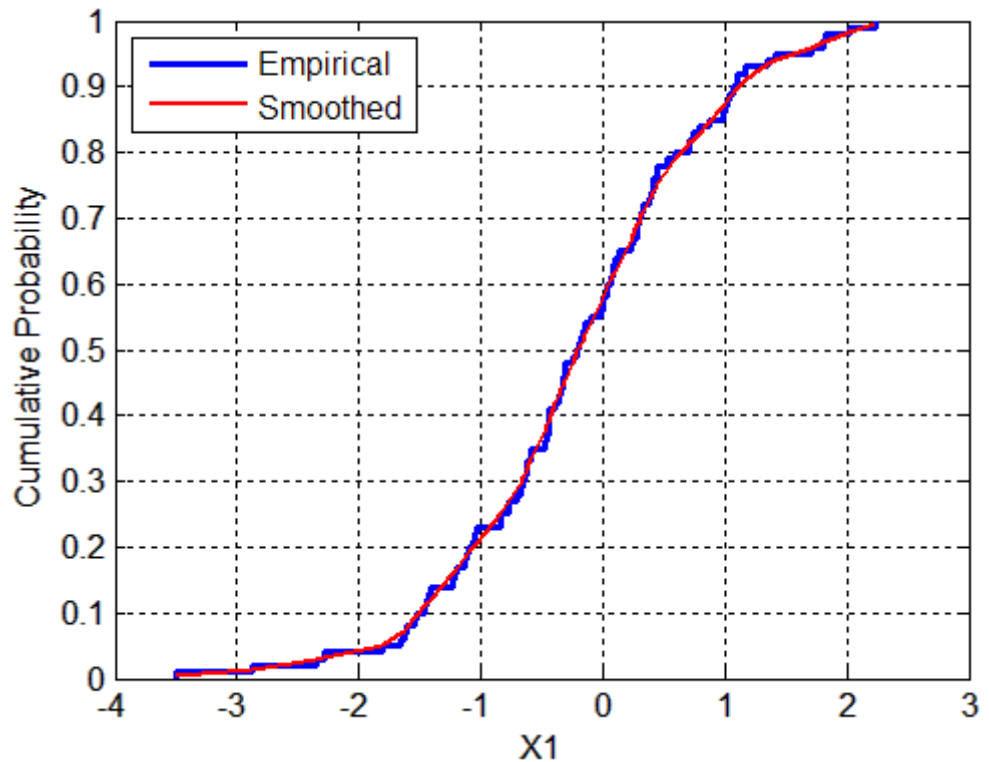

For the simulation, experiment with different copulas and correlations.
Here, you will use a bivariate *t* copula with a fairly small degrees of freedom

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 using `copulaparam`:

```
nu = 5;

tau = corr(stocks(:,1),stocks(:,2),'type','kendall')
tau =
      0.51798

rho = copulaparam('t', tau, nu, 'type','kendall')
rho =
      0.72679
```

Next, use `copularnd` to 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(0.00001,0.99999,1000);
G1 = ksdensity(stocks(:,1),p,'function','icdf','width',0.15);
X1 = interp1(p,G1,U(:,1),'spline');
G2 = ksdensity(stocks(:,2),p,'function','icdf','width',0.15);
X2 = interp1(p,G2,U(:,2),'spline');

scatterhist(X1,X2)
```
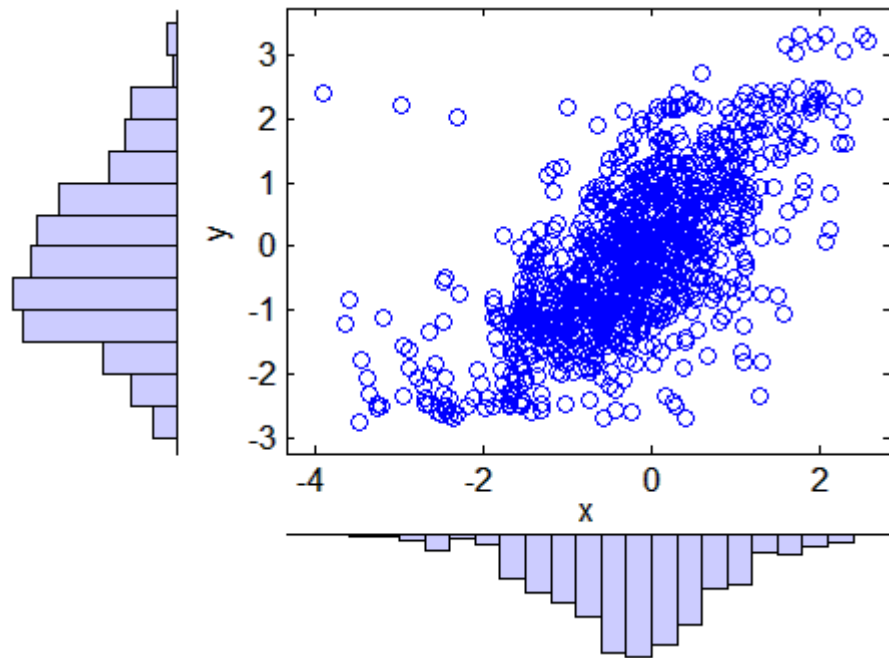
The marginal histograms of the simulated data are a smoothed version of the histograms for the original data. The amount of smoothing is controlled by the bandwidth input to ksdensity.

Fitting Copulas to Data

The copulafit function is used to calibrate copulas with data. To generate data Xsim with a distribution "just like" (in terms of marginal distributions and correlations) the distribution of data in the matrix X:

1 Fit marginal distributions to the columns of X.

2 Use appropriate cdf functions to transform X to U, so that U has values between 0 and 1.

3 Use copulafit to fit a copula to U.

4 Generate new data Usim from the copula.

5 Use appropriate inverse cdf functions to transform `Usim` to `Xsim`.

The following example illustrates the procedure.

Load and plot simulated stock return data:

```
load stockreturns
x = stocks(:,1);
y = stocks(:,2);

scatterhist(x,y)
```


Transform the data to the copula scale (unit square) using a kernel estimator of the cumulative distribution function:

```
u = ksdensity(x,x,'function','cdf');
v = ksdensity(y,y,'function','cdf');

scatterhist(u,v)
xlabel('u')
ylabel('v')
```


Fit a *t* copula:

```
[Rho,nu] = copulafit('t',[u v],'Method','ApproximateML')
Rho =
    1.0000    0.7220
    0.7220    1.0000
nu =
  2.8934e+006
```

Generate a random sample from the t copula:

```
r = copularnd('t',Rho,nu,1000);
u1 = r(:,1);
v1 = r(:,2);

scatterhist(u1,v1)
xlabel('u')
ylabel('v')
set(get(gca,'children'),'marker','.')
```
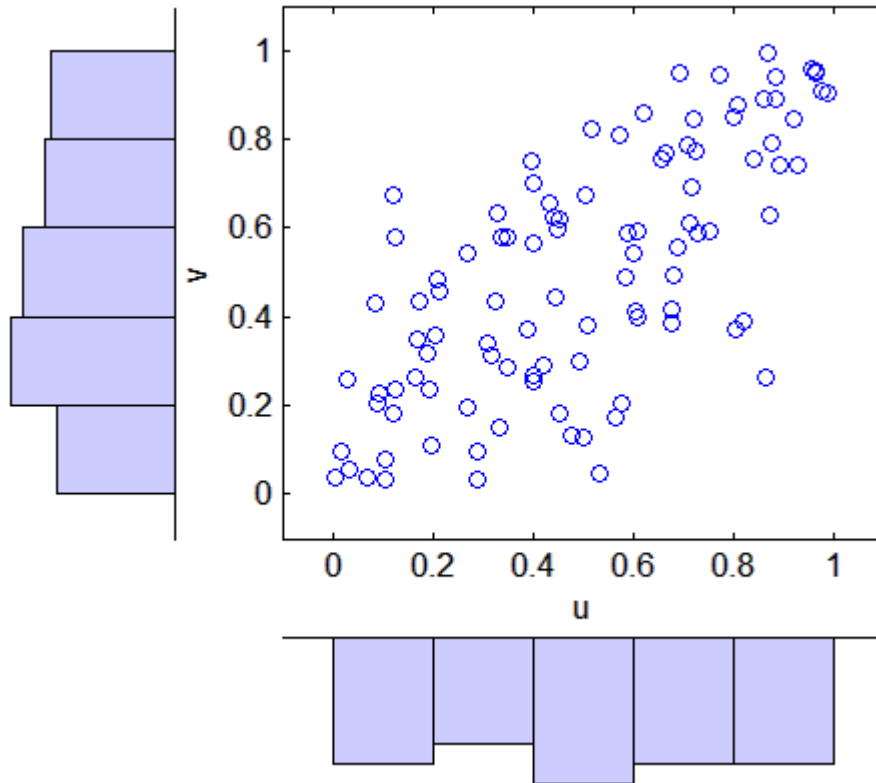

Transform the random sample back to the original scale of the data:

```
x1 = ksdensity(u,u1,'function','icdf');
```

```
y1 = ksdensity(v,v1,'function','icdf');

scatterhist(x1,y1)
set(get(gca,'children'),'marker','.')
```
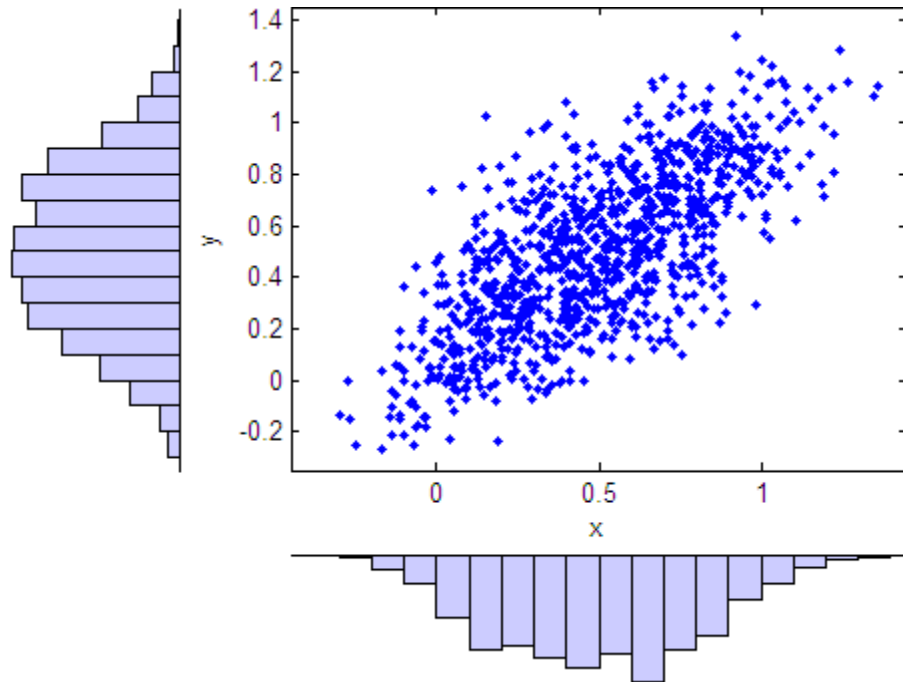

As the example illustrates, copulas integrate naturally with other distribution fitting functions.

Random Number Generation

In this section...
"Introduction" on page 5-132
"Common Generation Methods" on page 5-132
"Markov Chain Samplers" on page 5-141
"Quasi-Random Numbers" on page 5-143

Introduction

Random number generators for supported distributions are discussed in "Random Number Generators" on page 5-38.

A GUI for generating random numbers from supported distributions is discussed in "Random Number Generation Tool" on page 5-81.

This section discusses additional topics in random number generation.

Common Generation Methods

- "Introduction" on page 5-132
- "Direct Methods" on page 5-133
- "Inversion Methods" on page 5-135
- "Acceptance-Rejection Methods" on page 5-137

Introduction

In statistics, *randomness* is a notion commonly associated with groups of, rather than individual, samples. More accurately, it is a notion associated with a process that produces samples. D. H. Lehmer, an early pioneer in computing, said:

"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... "

The notion of randomness is formalized by the Central Limit Theorem, which says that sample averages from a random process always approximate a normal distribution. This formalization is central to the theory of statistical estimation.

Pseudo-random numbers are generated by deterministic algorithms. They are "random" in the sense that, on average, they pass statistical tests regarding their distribution and correlation. *Random number generators* (RNGs), like those in MATLAB and Statistics Toolbox software, are algorithms for generating pseudo-random numbers with a specified distribution.

Methods for generating pseudo-random numbers usually start with uniform random numbers, like those produced by the MATLAB `rand` function. Random numbers from other distributions are produced using the methods described in this section.

Direct Methods

Direct methods make direct use of the definition of the distribution.

For example, consider binomial random numbers. A binomial random number is the number of heads in N tosses of a coin with probability p of a heads on any single 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 direct 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)
```

5-133

```
set(get(gca,'Children'),'FaceColor',[.8 .8 1])
```
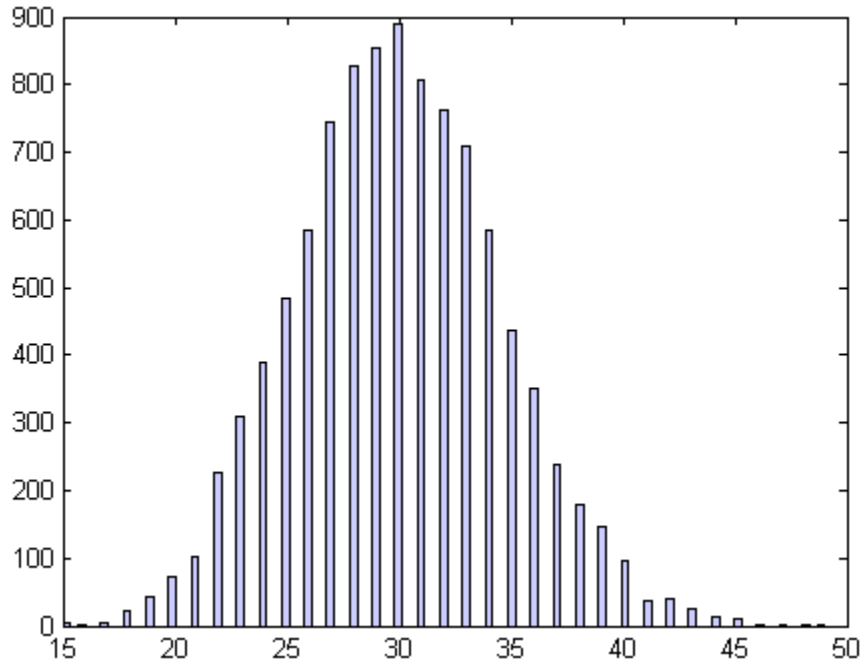

The Statistics Toolbox function `binornd` uses a modified direct method, based on the definition of a binomial random variable as the sum of Bernoulli random variables.

The method above is easily converted to a random number generator for the Poisson distribution with parameter λ. 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, create a version of the above generator that inputs λ rather than N and p, and 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 is 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);
```

To compare the distribution of the generated random numbers 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)
set(get(gca,'Children'),'FaceColor',[.8 .8 1])
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)
```
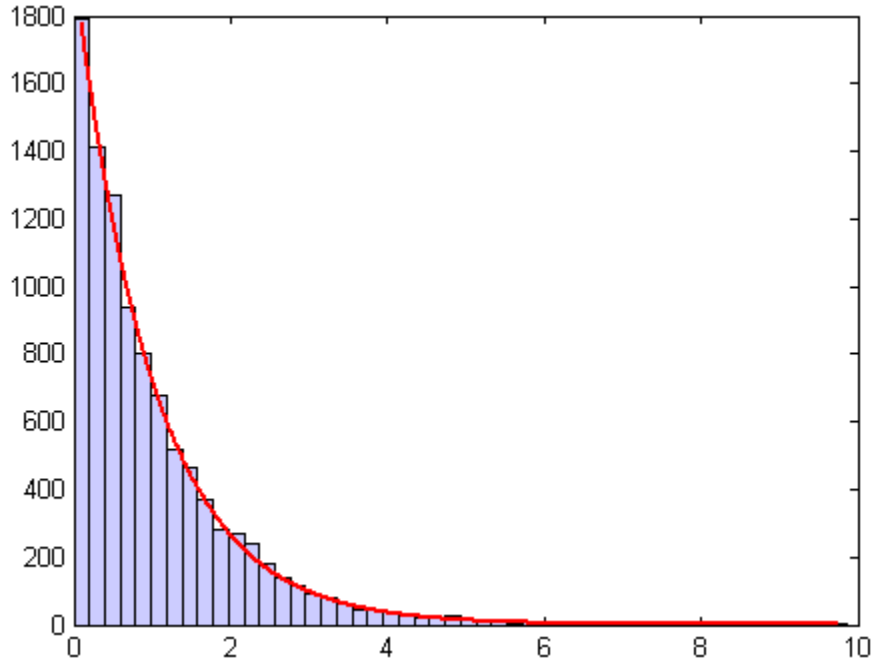
Inversion methods also work for discrete distributions. To generate a random number X from a discrete distribution with probability mass vector $P(X = x_i) = p_i$ where $x_0 < x_1 < x_2 < \dots$, generate a uniform random number u on $(0,1)$ and then set $X = x_i$ if $F(x_{i-1}) < u < F(x_i)$.

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
```

Use the function 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)
set(get(gca,'Children'),'FaceColor',[.8 .8 1])
```
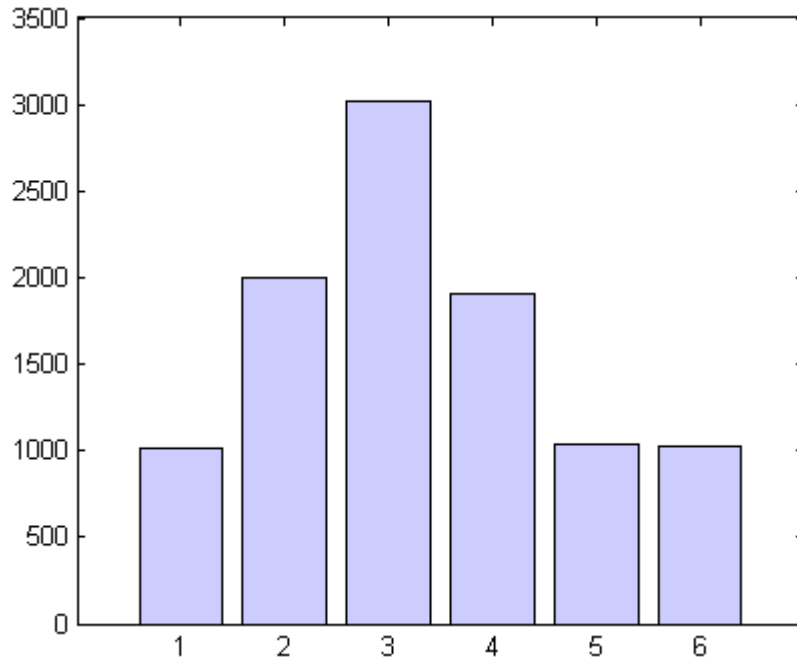

Acceptance-Rejection Methods

The functional form of some distributions makes it difficult or time-consuming to generate random numbers using direct or inversion methods. Acceptance-rejection methods provide an alternative in these cases.

Acceptance-rejection methods begin with uniform random numbers, but require 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 $f(x) \leq cg(x)$ for some c and all x.
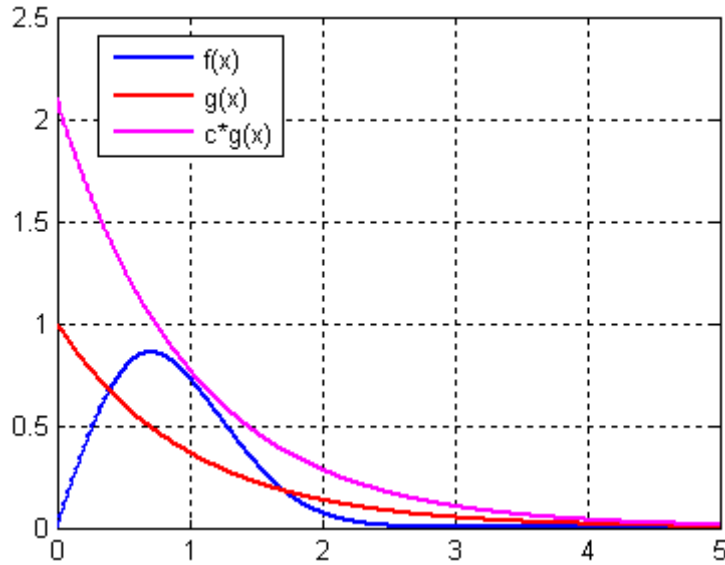
A continuous acceptance-rejection RNG proceeds as follows:

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, a "cheap" method is required for generating random numbers from g, and the scalar c should be small. The expected number of iterations to produce a single 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
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 following compares the distribution of random numbers generated by the acceptance-rejection method with those generated by raylrnd:

```
Y = raylrnd(1,1e4,1);
hist([X Y])
h = get(gca,'Children');
set(h(1),'FaceColor',[.8 .8 1])
legend('A-R RNG','Rayleigh RNG')
```
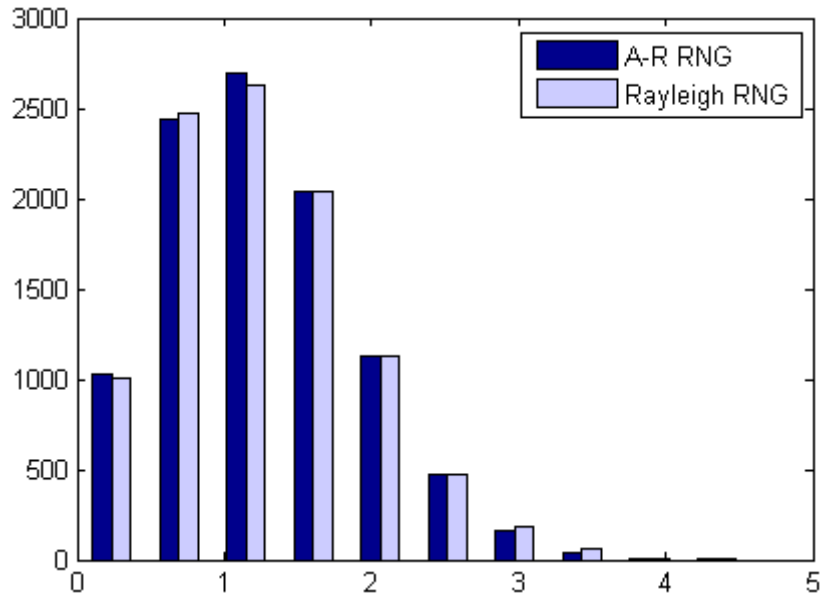
The Statistics Toolbox function `raylrnd` uses a transformation method, expressing a Rayleigh random variable in terms of a chi-square random variable, which can be computed using `randn`.

Acceptance-rejection methods also work for 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.

Markov Chain Samplers

Introduction

The methods discussed in "Common Generation Methods" on page 5-132 may be inadequate when sampling distributions are difficult to represent in computations. Such distributions arise, for example, in Bayesian data analysis and in the large combinatorial problems of Markov chain Monte Carlo (MCMC) simulations. An alternative is to construct a Markov chain with a stationary distribution equal to the target sampling distribution, using the states of the chain to generate random numbers after an initial burn-in period in which the state distribution converges to the target.

Metropolis-Hastings Sampler

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.

To generate random numbers:

1 Assume an 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 $t \rightarrow t+1$, 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 Sampler

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, then do the following to generate random numbers:

1 Assume an 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.

Quasi-Random Numbers

Quasi-Random Sequences

Quasi-random number generators (QRNGs) produce highly uniform samples of the unit hypercube. They are designed to minimize the *discrepancy* between the distribution of generated points and a distribution with equal proportions of points in each sub-cube of a uniform partition of the hypercube. As a result, QRNGs systematically fill the "holes" in any initial segment of the generated quasi-random sequence.

Unlike the pseudo-random sequences described in the Introduction to common generation methods, quasi-random sequences fail many statistical tests for randomness. Approximating true randomness, however, is not their goal. Quasi-random sequences seek to fill space uniformly, and to do so in such a way that initial segments approximate this behavior up to a specified density.

Applications of QRNGs include:

- **Quasi-Monte Carlo (QMC) integration.** Monte Carlo techniques are often used to evaluate difficult, multi-dimensional integrals without a closed-form solution. QMC uses quasi-random sequences to improve the convergence properties of these techniques.

- **Space-filling experimental designs.** In many experimental settings, taking measurements at every factor setting is expensive or infeasible. Quasi-random sequences provide efficient, uniform sampling of the design space.

- **Global optimization.** Optimization algorithms typically find a local optimum in the neighborhood of an initial value. By using a quasi-random

5-143

sequence of initial values, searches for global optima uniformly sample the basins of attraction of all local minima.

Quasi-Random Point Sets

- "Introduction" on page 5-144
- "Example: Quasi-Random Point Sets" on page 5-145

Introduction. Statistics Toolbox functions support quasi-random sequences of the following types:

- **Halton sequences.** Produced by the `haltonset` function. These sequences use different prime bases to form successively finer uniform partitions of the unit interval in each dimension.

- **Sobol sequences.** Produced by the `sobolset` function. These sequences use a base of 2 to form successively finer uniform partitions of the unit interval, and then reorder the coordinates in each dimension.

- **Latin hypercube sequences.** Produced by the `lhsdesign` function. Though not quasi-random in the sense of minimizing discrepancy, these sequences nevertheless produce sparse uniform samples useful in experimental designs.

Quasi-random sequences are functions from the positive integers to the unit hypercube. To be useful in application, an initial *point set* of a sequence must be generated. Point sets are matrices of size n-by-d, where n is the number of points and d is the dimension of the hypercube being sampled. The functions `haltonset` and `sobolset` construct point sets with properties of a specified quasi-random sequence. Initial segments of the point sets are generated by the `net` method of the `qrandset` class (parent class of the `haltonset` class and `sobolset` class), but points can be generated and accessed more generally using parenthesis indexing.

Because of the way in which quasi-random sequences are generated, they may contain undesirable correlations, especially in their initial segments, and especially in higher dimensions. To address this issue, quasi-random point sets often *skip*, *leap* over, or *scramble* values in a sequence. The `haltonset` and `sobolset` functions allow you to specify both a `Skip` and a `Leap` property of a quasi-random sequence, and the `scramble` method of the `qrandset` class

allows you apply a variety of scrambling techniques. Scrambling reduces correlations while also improving uniformity.

Example: Quasi-Random Point Sets. For example, the following uses `haltonset` to construct a 2-dimensional Halton point set—an object p of the `haltonset` class—that skips the first 1000 values of the sequence and then retains every 101st point:

```
p = haltonset(2,'Skip',1e3,'Leap',1e2)
p =
    Halton point set in 2 dimensions (8.918019e+013 points)
    Properties:
              Skip : 1000
              Leap : 100
    ScrambleMethod : none
```

The object p encapsulates properties of the specified quasi-random sequence. The point set is finite, with a length determined by the Skip and Leap properties and by limits on the size of point set indices (maximum value of 2^{53}).

Use `scramble` to apply reverse-radix scrambling:

```
p = scramble(p,'RR2')
p =
    Halton point set in 2 dimensions (8.918019e+013 points)
    Properties:
              Skip : 1000
              Leap : 100
    ScrambleMethod : RR2
```

Use `net` to generate the first 500 points:

```
X0 = net(p,500);
```

This is equivalent to:

```
X0 = p(1:500,:);
```

Values of the point set X0 are not generated and stored in memory until you access p using `net` or parenthesis indexing.

To appreciate the nature of quasi-random numbers, create a scatter of the two dimensions in X0:

```
scatter(X0(:,1),X0(:,2),5,'r')
axis square
title('{\bf Quasi-Random Scatter}')
```
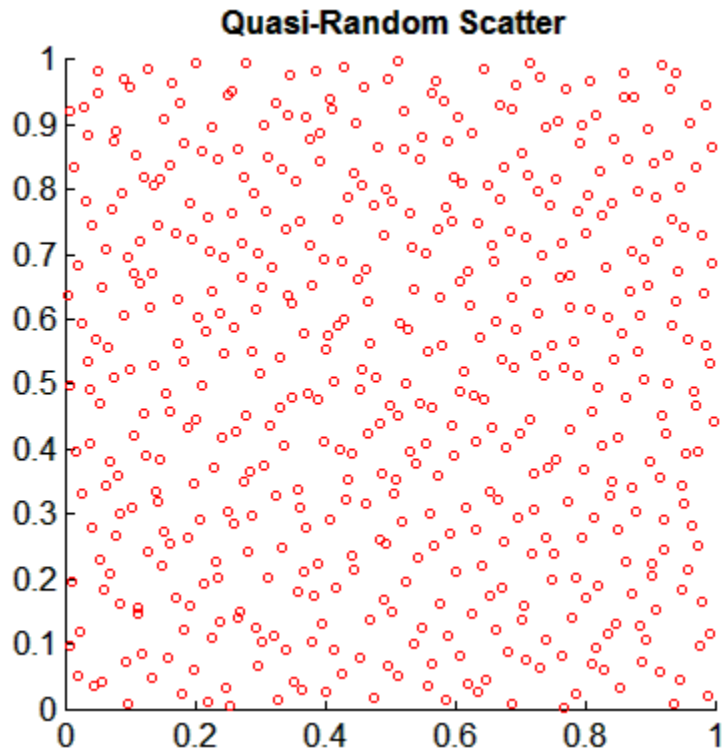

Compare this to a scatter of uniform pseudo-random numbers generated by the MATLAB rand function:

```
X = rand(500,2);

scatter(X(:,1),X(:,2),5,'b')
axis square
title('{\bf Uniform Random Scatter}')
```

Uniform Random Scatter
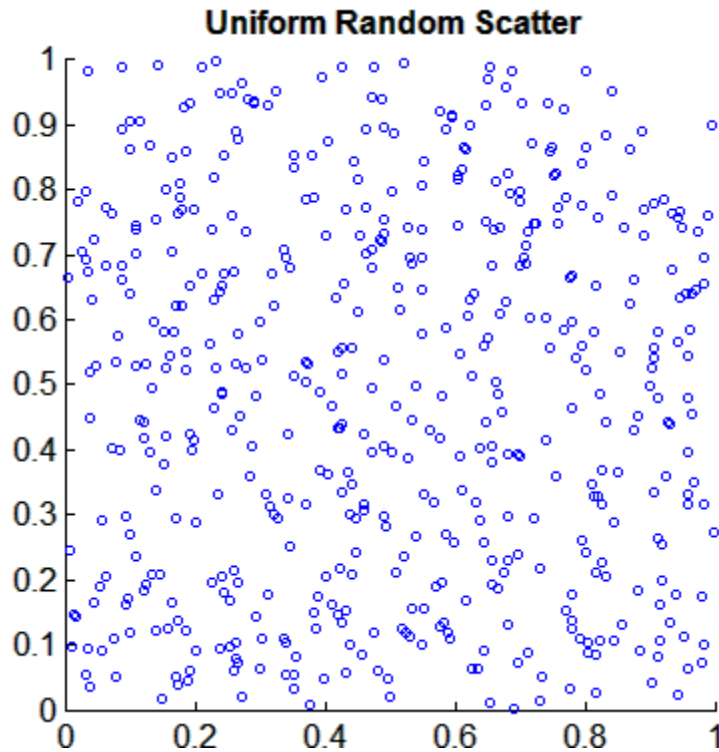
The quasi-random scatter appears more uniform, avoiding the clumping in the pseudo-random scatter.

In a statistical sense, quasi-random numbers are too uniform to pass traditional tests of randomness. For example, a Kolmogorov-Smirnov test, performed by kstest, is used to assess whether or not a point set has a uniform random distribution. When performed repeatedly on uniform pseudo-random samples, such as those generated by rand, the test produces a uniform distribution of p-values:

```
nTests = 1e5;
sampSize = 50;
PVALS = zeros(nTests,1);
for test = 1:nTests
    x = rand(sampSize,1);
    [h,pval] = kstest(x,[x,x]);
```

```
      PVALS(test) = pval;
end

hist(PVALS,100)
set(get(gca,'Children'),'FaceColor',[.8 .8 1])
xlabel('{\it p}-values')
ylabel('Number of Tests')
```
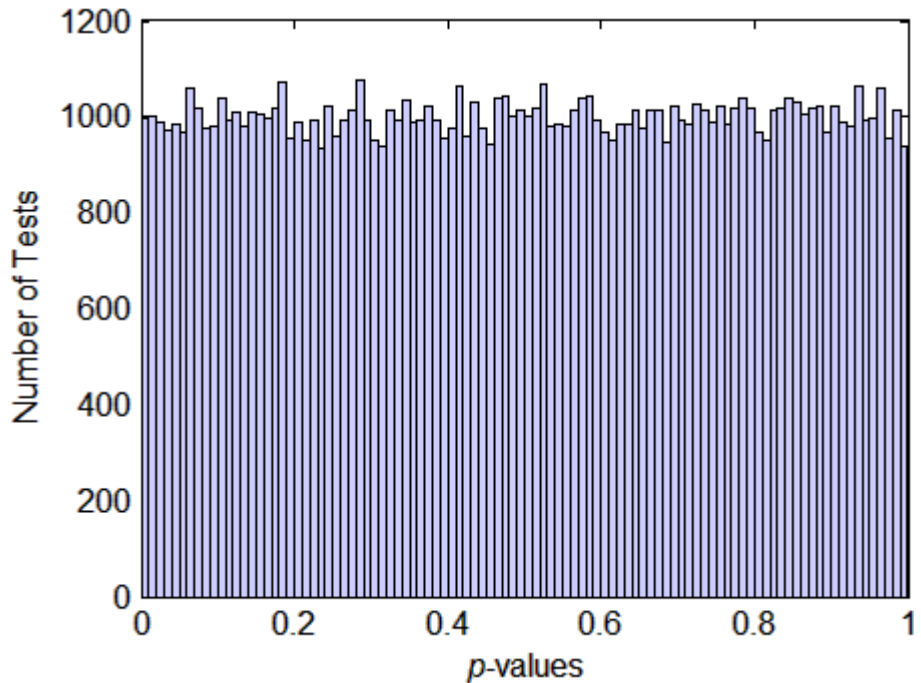

The results are quite different when the test is performed repeatedly on uniform quasi-random samples:

```
p = haltonset(1,'Skip',1e3,'Leap',1e2);
p = scramble(p,'RR2');

nTests = 1e5;
sampSize = 50;
PVALS = zeros(nTests,1);
for test = 1:nTests
```

```
        x = p(test:test+(sampSize-1),:);
        [h,pval] = kstest(x,[x,x]);
        PVALS(test) = pval;
    end

    hist(PVALS,100)
    set(get(gca,'Children'),'FaceColor',[.8 .8 1])
    xlabel('{\it p}-values')
    ylabel('Number of Tests')
```
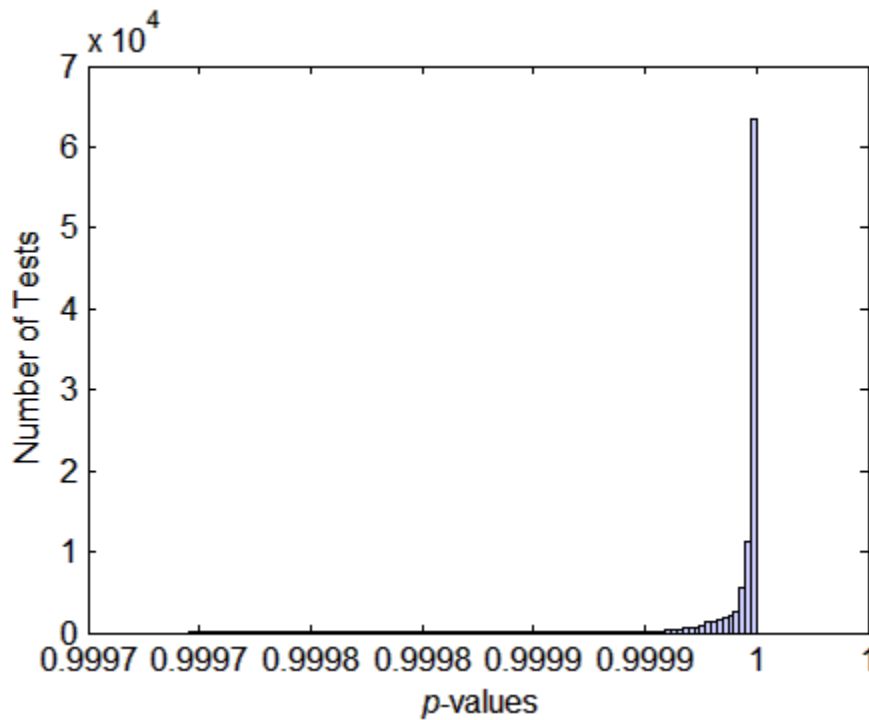

Small p-values call into question the null hypothesis that the data are uniformly distributed. If the hypothesis is true, about 5% of the p-values are expected to fall below 0.05. The results are remarkably consistent in their failure to challenge the hypothesis.

Quasi-Random Streams

- "Introduction" on page 5-150
- "Example: Quasi-Random Streams" on page 5-150

Introduction. Quasi-random *streams*, produced by the `qrandstream` function, are used to generate sequential quasi-random outputs, rather than point sets of a specific size. Streams are used like pseudo-RNGS, such as `rand`, when client applications require a source of quasi-random numbers of indefinite size that can be accessed intermittently. Properties of a quasi-random stream, such as its type (Halton or Sobol), dimension, skip, leap, and scramble, are set when the stream is constructed.

In implementation, quasi-random streams are essentially very large quasi-random point sets, though they are accessed differently. The *state* of a quasi-random stream is the scalar index of the next point to be taken from the stream. Use the `qrand` method of the `qrandstream` class to generate points from the stream, starting from the current state. Use the `reset` method to reset the state to 1. Unlike point sets, streams do not support parenthesis indexing.

Example: Quasi-Random Streams. For example, the following code, taken from the example at the end of "Quasi-Random Point Sets" on page 5-144, uses `haltonset` to create a quasi-random point set p, and then repeatedly increments the index into the point set, `test`, to generate different samples:

```
p = haltonset(1,'Skip',1e3,'Leap',1e2);
p = scramble(p,'RR2');

nTests = 1e5;
sampSize = 50;
PVALS = zeros(nTests,1);
for test = 1:nTests
    x = p(test:test+(sampSize-1),:);
    [h,pval] = kstest(x,[x,x]);
    PVALS(test) = pval;
end
```

The same results are obtained by using qrandstream to construct a quasi-random stream q based on the point set p and letting the stream take care of increments to the index:

```
p = haltonset(1,'Skip',1e3,'Leap',1e2);
p = scramble(p,'RR2');
q = qrandstream(p)

nTests = 1e5;
sampSize = 50;
PVALS = zeros(nTests,1);
for test = 1:nTests
    X = qrand(q,sampSize);
    [h,pval] = kstest(X,[X,X]);
    PVALS(test) = pval;
end
```

6

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 functions are available 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 Statistics Toolbox hypothesis tests are given in the reference pages for implementing functions.

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)
```
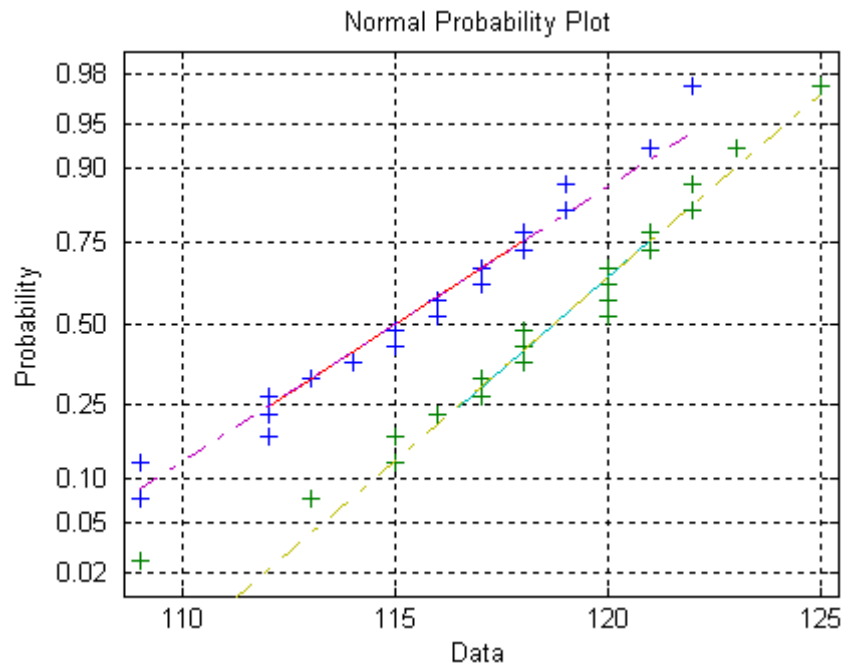
Both scatters approximately follow straight lines through the first and third quartiles of the samples, indicating approximate normal distributions. The February sample (the right-hand line) shows a slight departure from normality in the lower tail. A shift in the mean from January to February is evident.

A hypothesis test is 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)')
```
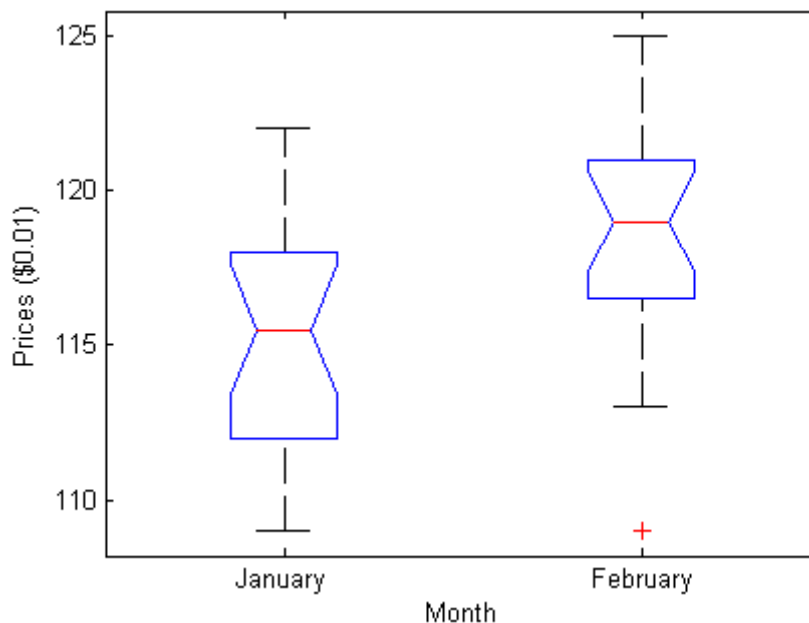

The plot displays the distribution of the samples around their medians. The heights of the notches in each box are computed so that the side-by-side boxes have nonoverlapping notches when their medians are different at a default 5% significance level. The computation is based on an assumption of normality 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 =
    0.0095
h =
     1
```

The test rejects the null hypothesis of equal medians at the default 5% significance level.

Available Hypothesis Tests

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.
linhyptest	Linear hypothesis test. Tests if $H*b = c$ for parameter estimates b with estimated covariance H and specified c, against the alternative that $H*b \neq c$.
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.

Note In addition to the previous functions, Statistics Toolbox functions are available for analysis of variance (ANOVA), which perform hypothesis tests in the context of linear modeling. These functions are discussed in Chapter 7, "Analysis of Variance".

7

Analysis of Variance

Introduction

Analysis of variance (ANOVA) is a procedure for assigning sample variance to different sources and deciding whether the variation arises within or among different population groups. Samples are described in terms of variation around group means and variation of group means around an overall mean. If variations within groups are small relative to variations between groups, a difference in group means may be inferred. Chapter 6, "Hypothesis Tests" are used to quantify decisions.

This chapter treats ANOVA among groups, that is, among categorical predictors. ANOVA for regression, with continuous predictors, is discussed in "Tabulating Diagnostic Statistics" on page 8-13.

Multivariate analysis of variance (MANOVA), for data with multiple measured responses, is also discussed in this chapter.

ANOVA

One-Way ANOVA

Introduction

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.

- a_j is a matrix whose columns are the group means. (The "dot *j*" notation means that a applies to all rows of column *j*. That is, the value a_{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.

Example: One-Way ANOVA
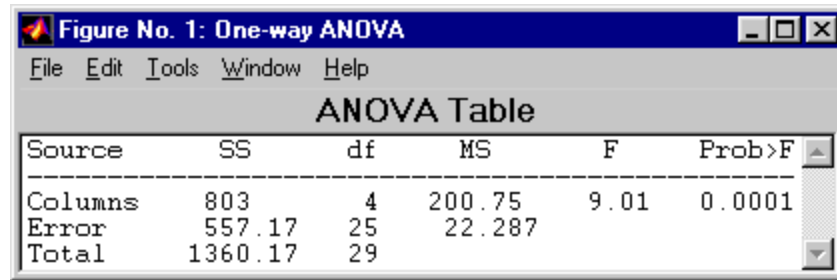
The data below comes from a study by Hogg and Ledolter [46] of bacteria counts in shipments of milk. The columns of the matrix hogg represent different shipments. The rows are bacteria counts from cartons of milk chosen randomly from each shipment. Do some shipments have higher counts than others?

```
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.

Figure No. 1: One-way ANOVA

File   Edit   Tools   Window   Help

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 shipments 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.

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.
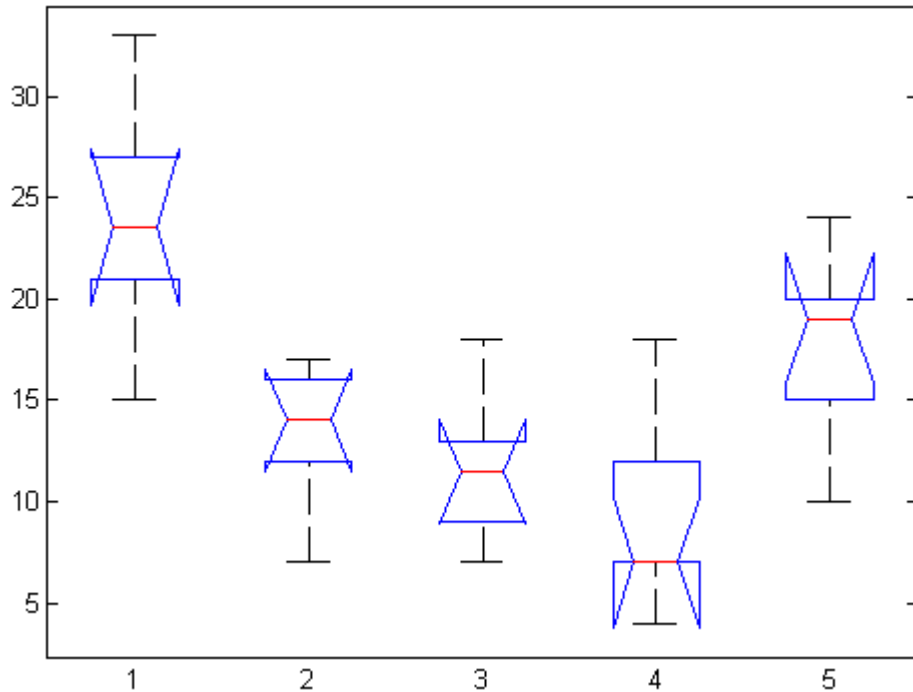
Multiple Comparisons

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.

```
load hogg
[p,tbl,stats] = anova1(hogg);
[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.
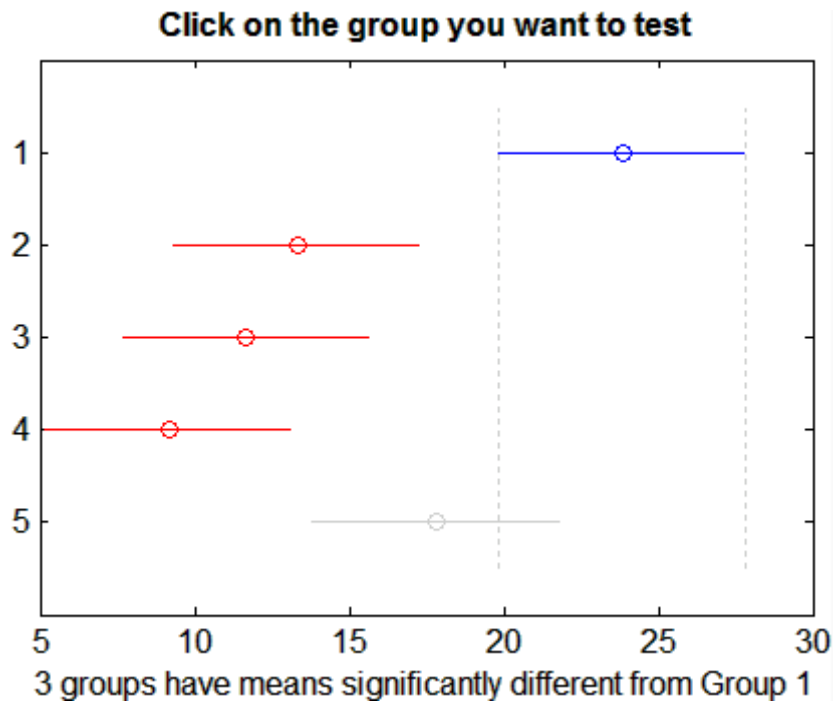
The graph shows that group 1 is significantly different from groups 2, 3, and 4. By using the mouse to select group 4, you can determine that it is also significantly different from group 5. Other pairs are not significantly different.

Two-Way ANOVA

- "Introduction" on page 7-9
- "Example: Two-Way ANOVA" on page 7-10

Introduction

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.

- β_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
```
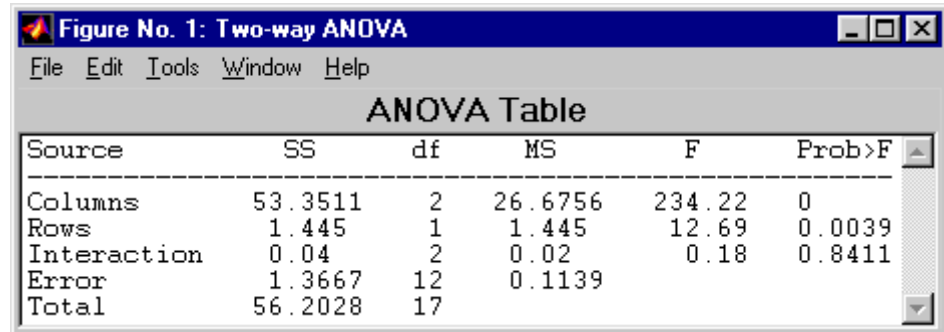
There are three models of cars (columns) and two factories (rows). The reason there are six rows in `mileage` instead of two is that each factory provides

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.

```
Figure No. 1: Two-way ANOVA                              _ □ ×
File  Edit  Tools  Window  Help

                      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.

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 ANOVA

- "Introduction" on page 7-12
- "N-Way ANOVA with a Small Data Set" on page 7-13
- "N-Way ANOVA with a Large Data Set" on page 7-15
- "ANOVA with Random Effects" on page 7-19

Introduction

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.

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
```

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
```
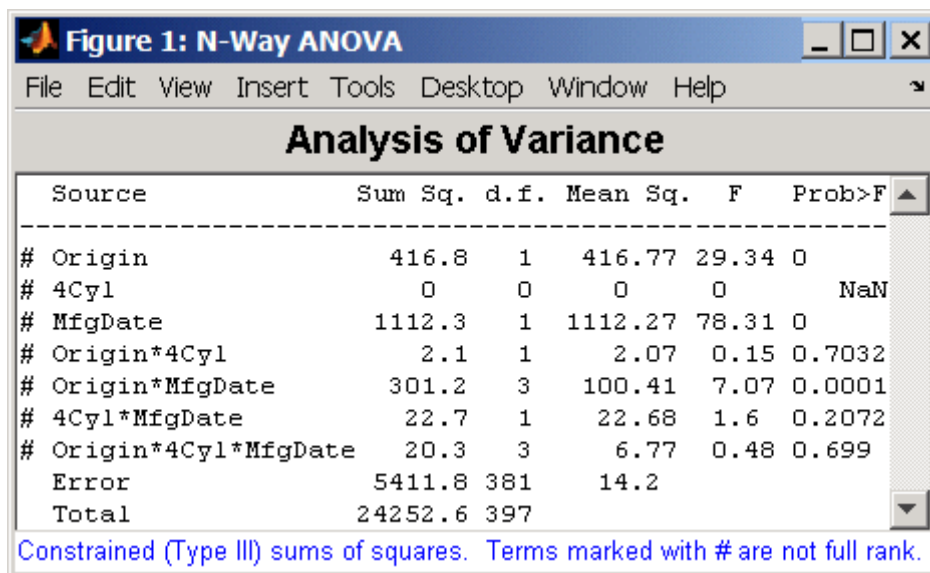

Note that many terms are marked by a # symbol as not having full rank, and one of them has zero degrees of freedom and is missing a *p*-value. This can happen when there are missing factor combinations and the model has higher-order terms. In this case, the cross-tabulation below shows that there are 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.
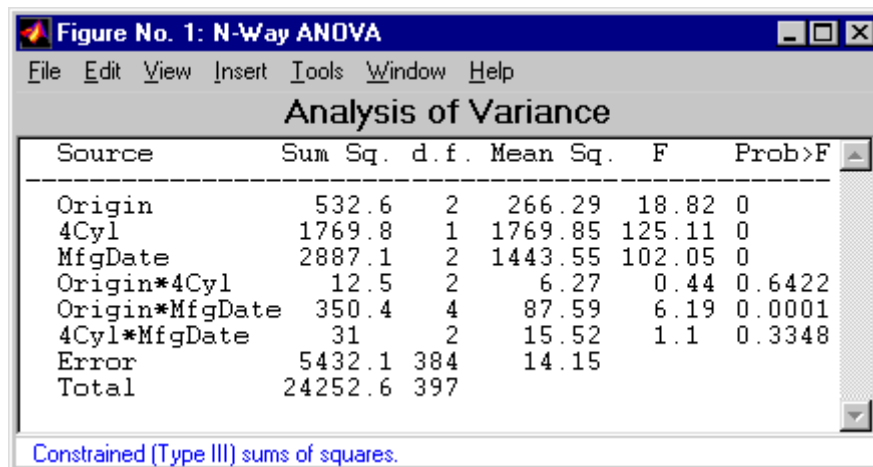
```
[p,tbl,stats,terms] = anovan(MPG,{org cyl4 when},2,3,varnames);
terms

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.
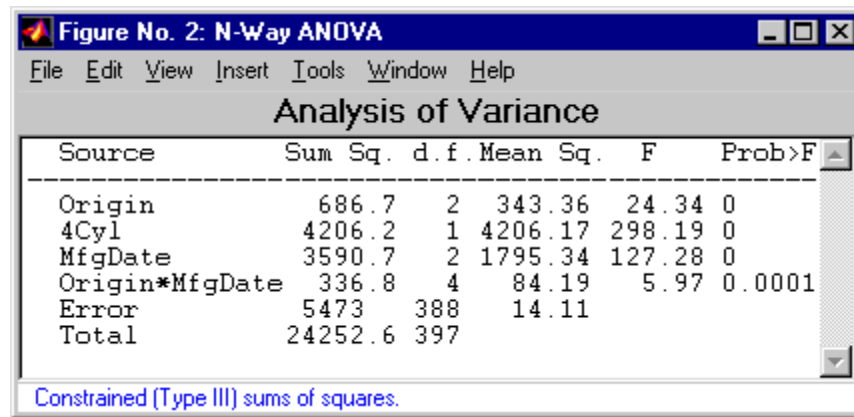
```
terms([4 6],:) = []

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.

ANOVA with Random Effects

Introduction. 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.

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 7-20, 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 7-20, 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)
```
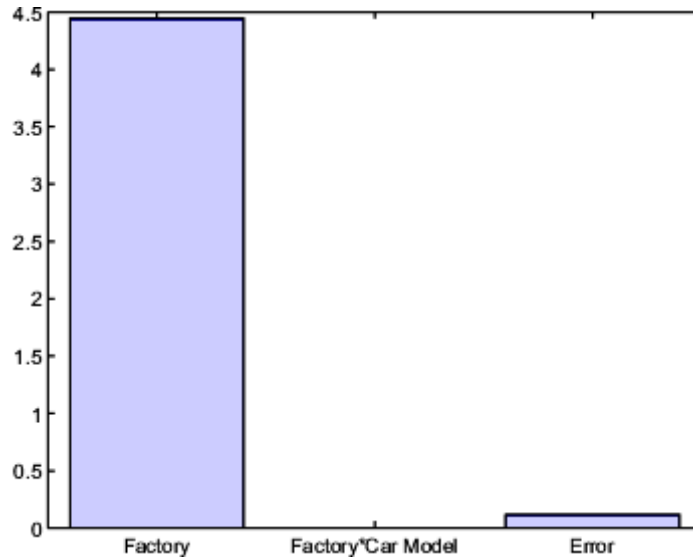

You can also compute confidence bounds for the variance estimate. The anovan function does this by computing confidence bounds for the variance expected mean squares, and finding lower and upper limits on each variance component containing all of these bounds. This procedure leads to a set of bounds 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

- "Introduction" on page 7-27
- "Analysis of Covariance Tool" on page 7-27
- "Confidence Bounds" on page 7-32
- "Multiple Comparisons" on page 7-34

Introduction

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.

Analysis of Covariance Tool

The `aoctool` function opens an interactive graphical environment for fitting and prediction with analysis of covariance (ANOCOVA) models. It fits the following models for the *i*th 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	$y = (\alpha + \alpha_i) + (\beta + \beta_i)x + \varepsilon$

For example, in the parallel lines model 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 tool imposes the constraints
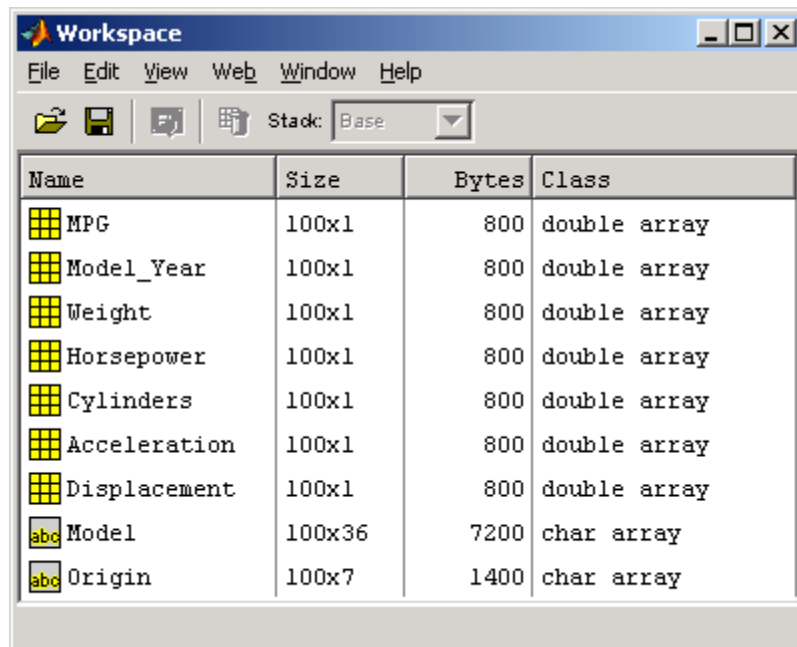
$$\sum \alpha_j = \sum \beta_j = 0$$

The following steps describe the use of `aoctool`.

1 **Load the data.** The Statistics Toolbox data set `carsmall.mat` contains information on 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.

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);
```

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.
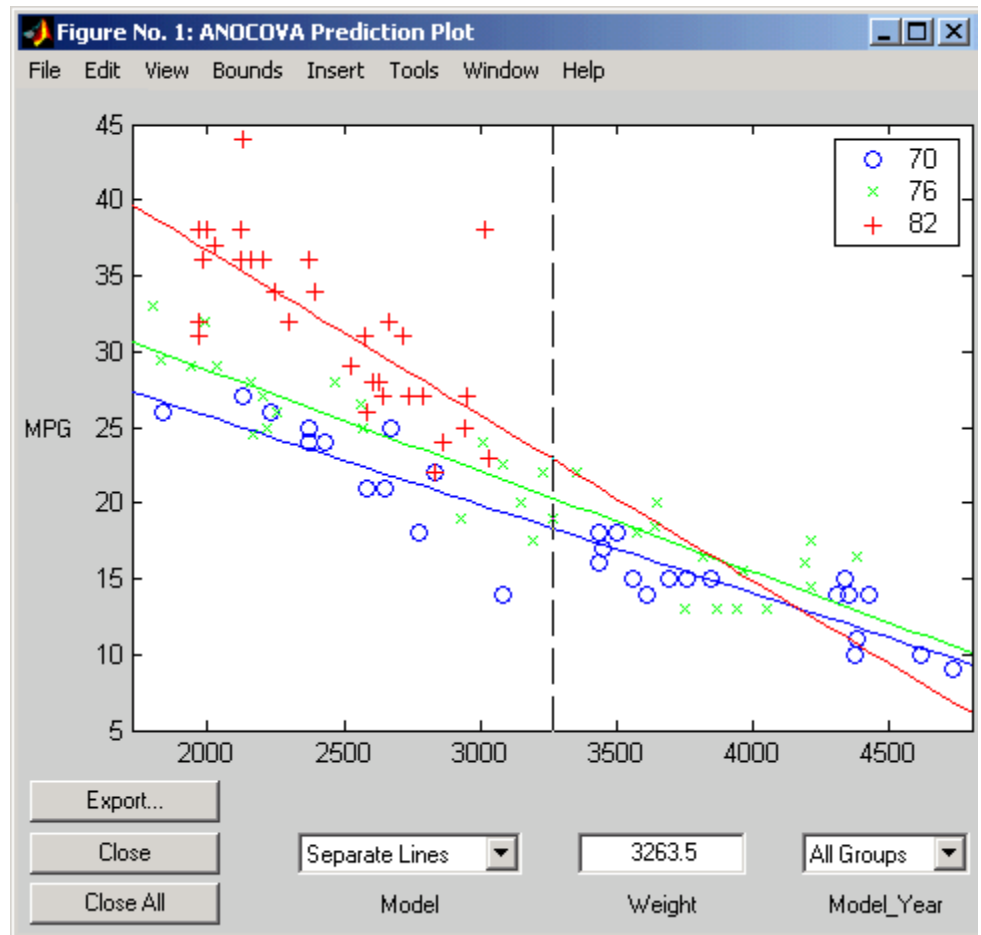
The coefficients of the three lines appear in the figure titled ANOCOVA Coefficients. You can see that the slopes are roughly -0.0078, with a small deviation for each group:

- 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

File Edit View Insert Tools Window Help

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

File Edit View Insert Tools Window Help

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*Weigh	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.
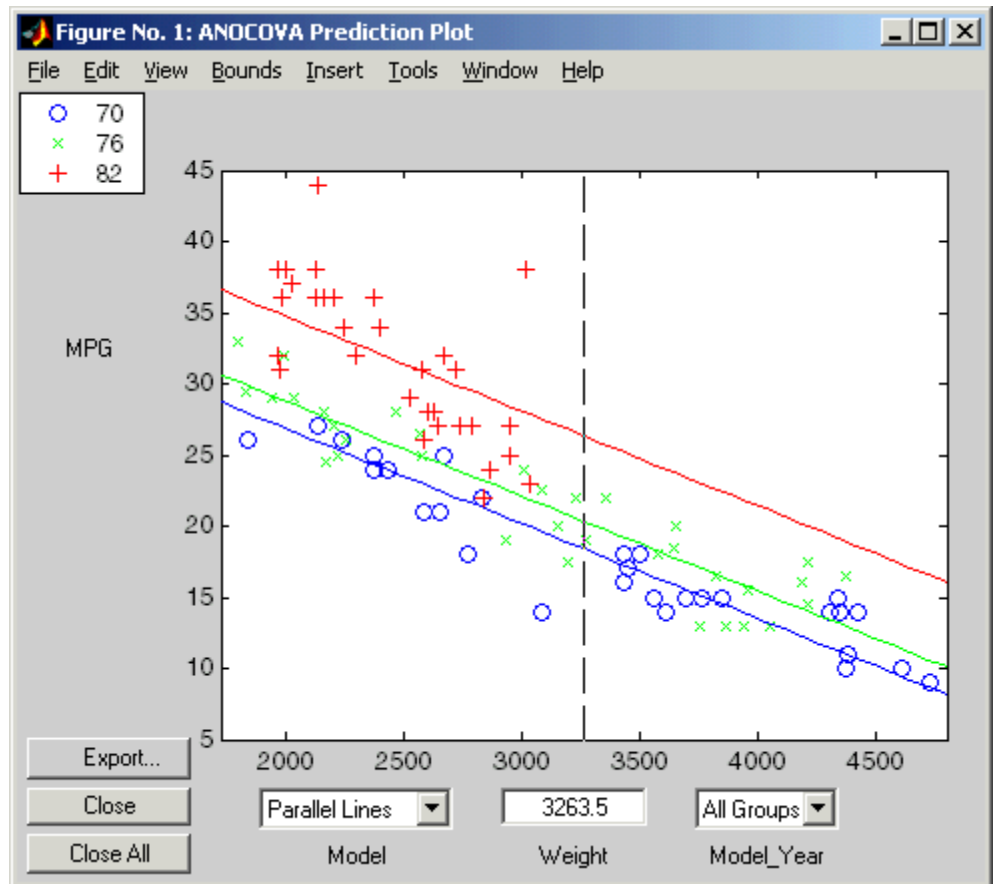
Though this fit looks reasonable, it is significantly worse than the `Separate Lines` model. Use the **Model** pop-up menu again to return to the original model.

Confidence Bounds

The example in "Analysis of Covariance Tool" on page 7-27 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.
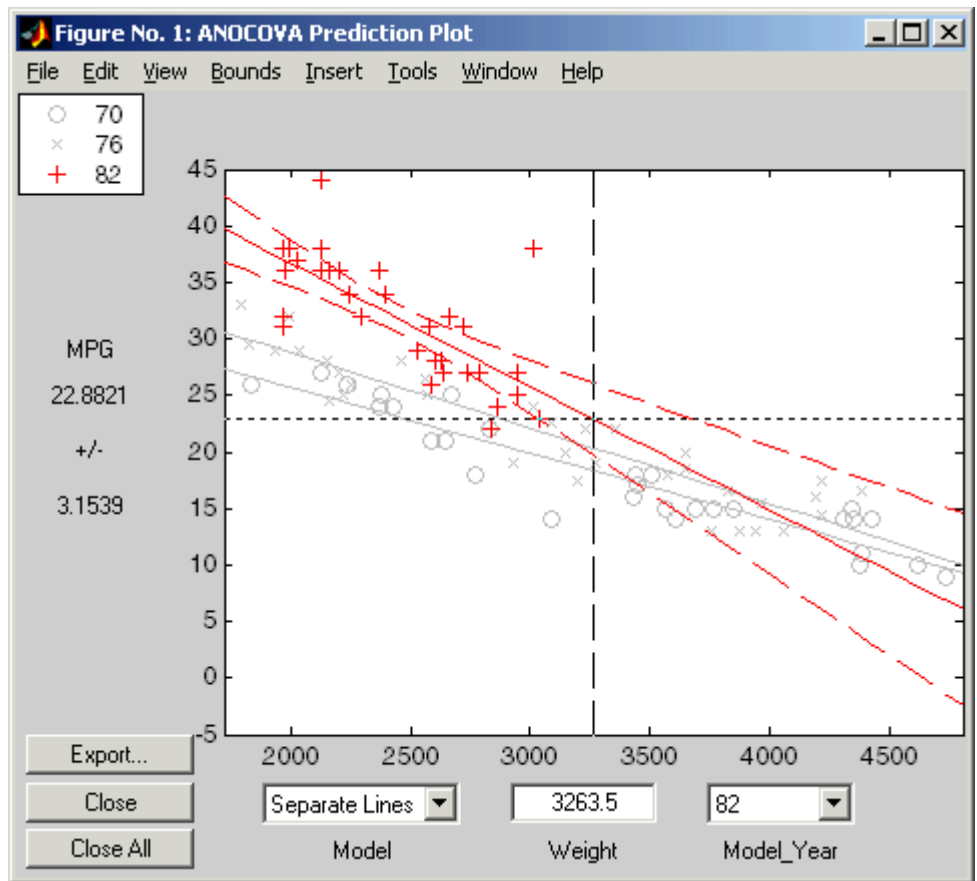
The dashed lines form an envelope around the fitted line for model year 82. Under the assumption that the true relationship is linear, these bounds provide a 95% confidence region for the true line. Note that the fits for the other model years are well outside these confidence bounds for `Weight` values 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.
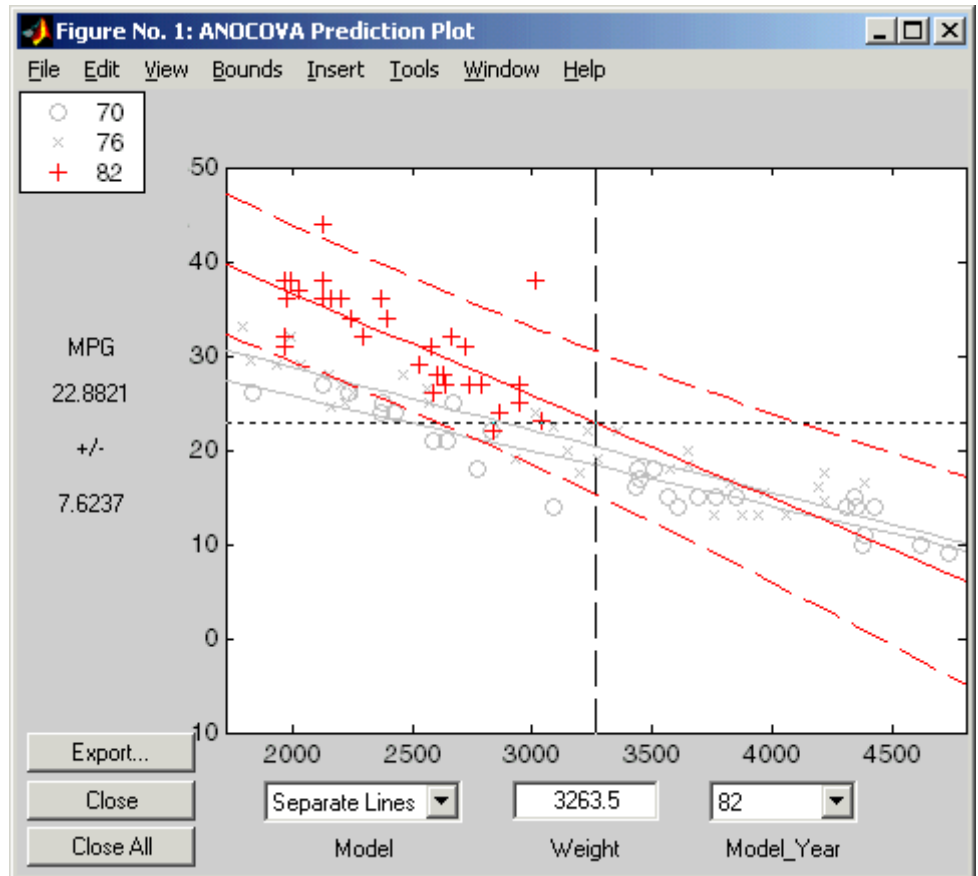
Like the `polytool` function, the `aoctool` function has cross hairs that you can use to manipulate the `Weight` and watch the estimate and confidence bounds along the *y*-axis update. These values appear only when a single group is selected, not when `All Groups` is selected.

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 "Analysis of Covariance Tool" on page 7-27 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.
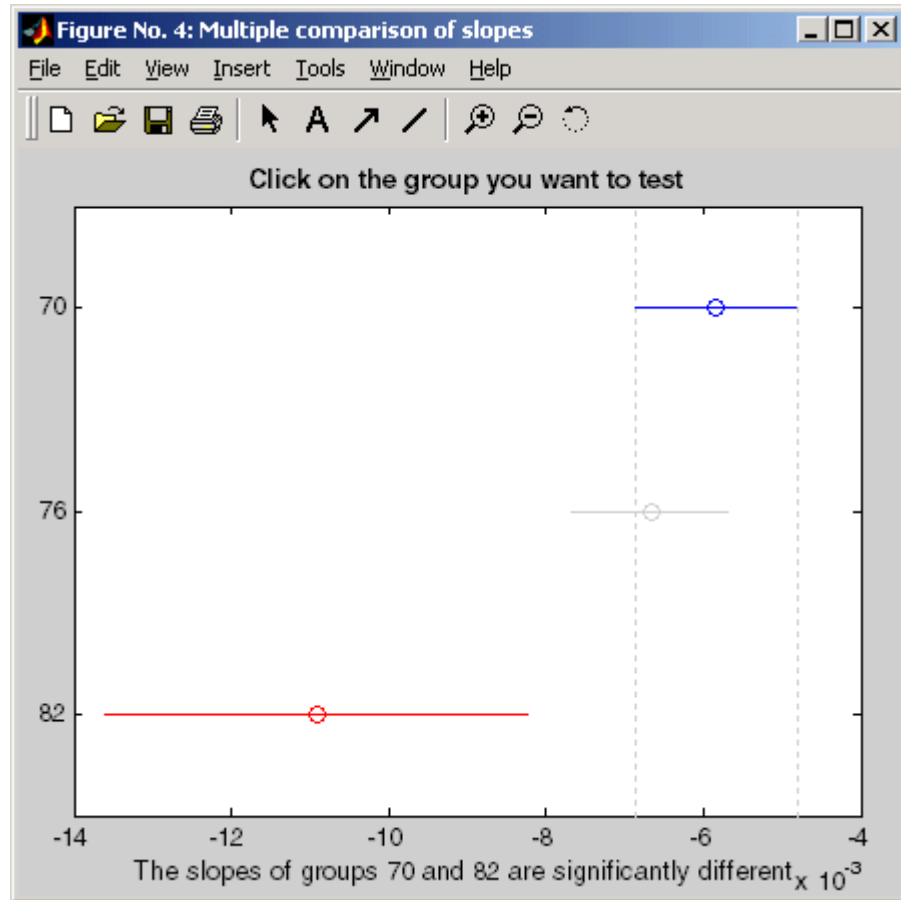
Note that the stats structure was created in the initial call to the aoctool function, so it is based on the initial model fit (typically a separate-lines model). If you change the model interactively and want to base your multiple comparisons on the new model, you need to run aoctool again to get another stats structure, this time specifying your new model as the initial model.

Nonparametric Methods

- "Introduction" on page 7-36

Introduction

Statistics Toolbox functions include nonparametric versions of one-way and two-way analysis of variance. Unlike classical tests, nonparametric tests make only mild assumptions about the data, and are appropriate when the distribution of the data is non-normal. On the other hand, they are less powerful than classical methods for normally distributed data.

Both of the nonparametric functions described here will return a `stats` structure that can be used as an input to the `multcompare` function for multiple comparisons.

Kruskal-Wallis Test

The example "Example: One-Way ANOVA" on page 7-4 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:

```
load hogg

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 7-10 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.

```
load mileage
p = friedman(mileage,3)
p =
  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.

MANOVA

In this section...
"Introduction" on page 7-39
"ANOVA with Multiple Responses" on page 7-39

Introduction

The analysis of variance technique in "Example: One-Way ANOVA" on page 7-4 takes a set of grouped data and determine whether the mean of a variable differs significantly among groups. Often there are multiple response 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 the problem.

ANOVA with Multiple Responses

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')
```
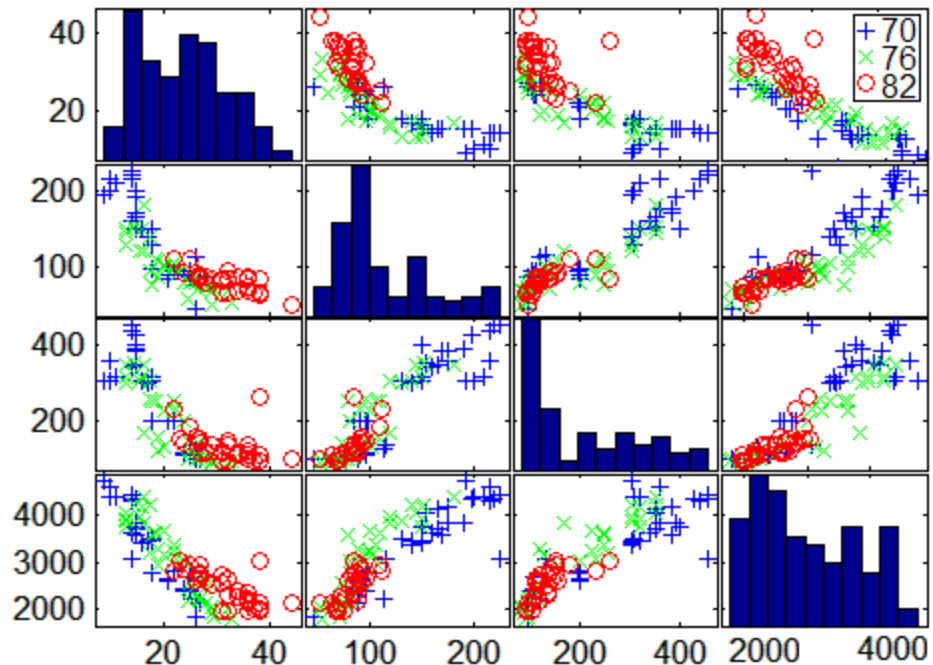

(When the second argument of `gplotmatrix` is empty, the function graphs the columns of the `x` argument against each other, and places histograms along the diagonals. The empty fourth argument produces a graph with the default colors. The fifth argument controls the symbols used to distinguish between 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 Component Analysis" on page 9-23) 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
```
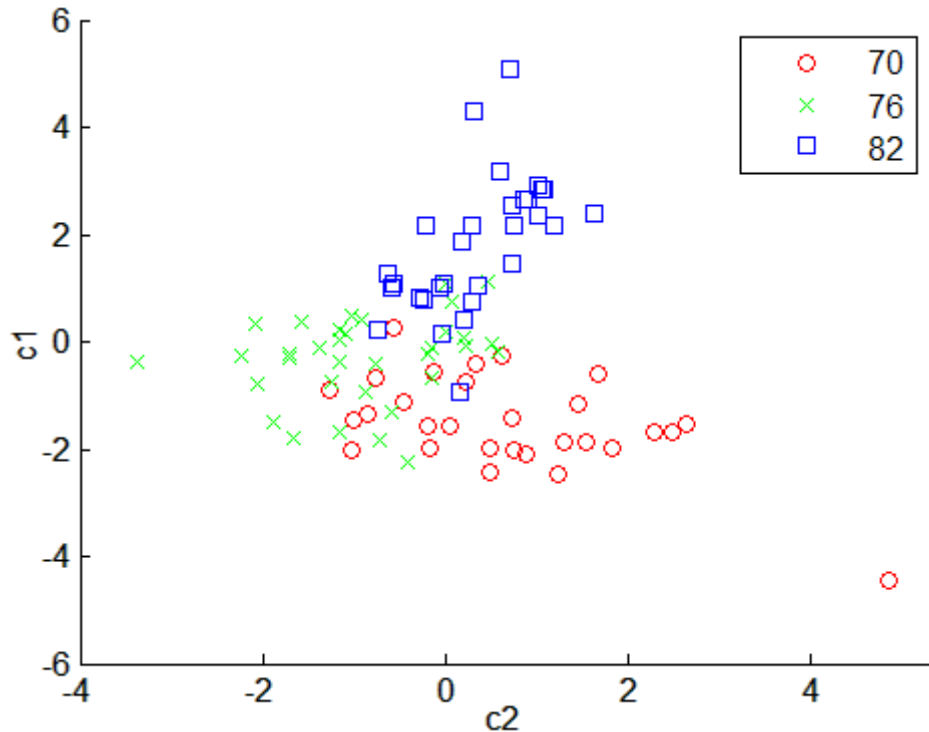
Roughly speaking, the first canonical variable, c1, separates the 1982 cars (which have high values of c1) from the older cars. The second canonical variable, c2, reveals some separation between the 1970 and 1976 cars.

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.

8

Regression Analysis

Introduction

Regression is the process of fitting models to data. The process depends on the model. If a model is parametric, regression estimates the parameters from the data. If a model is linear in the parameters, estimation is based on methods from linear algebra that minimize the norm of a residual vector. If a model is nonlinear in the parameters, estimation is based on search methods from optimization that minimize the norm of a residual vector. Nonparametric models, like "Regression Trees" on page 8-84, use methods all their own.

This chapter considers data and models with continuous predictors and responses. Categorical predictors are the subject of Chapter 7, "Analysis of Variance". Categorical responses are the subject of Chapter 11, "Classification".

Linear Regression

Linear Regression Models

In statistics, linear regression models often take the form of something like this:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_1 x_2 + \beta_4 x_1^2 + \beta_5 x_2^2 + \varepsilon$$

Here a response variable y is modeled as a combination of constant, linear, interaction, and quadratic terms formed from two predictor variables x_1 and x_2. Uncontrolled factors and experimental errors are modeled by ε. Given data on x_1, x_2, and y, *regression* estimates the model parameters β_j ($j = 1, ..., 5$).

More general linear regression models represent the relationship between a continuous response y and a continuous or categorical predictor $\boldsymbol{x}$ in the form:

$$y = \beta_1 f_1(\boldsymbol{x}) + ... + \beta_p f_p(\boldsymbol{x}) + \varepsilon$$

The response is modeled as a linear combination of (not necessarily linear) functions of the predictor, plus a random error ε. The expressions $f_j(\boldsymbol{x})$ ($j = 1, ..., p$) are the *terms* of the model. The β_j ($j = 1, ..., p$) are the *coefficients*. Errors

ε are assumed to be uncorrelated and distributed with mean 0 and constant (but unknown) variance.

Examples of linear regression models with a scalar predictor variable x include:

- Linear additive (straight-line) models — Terms are $f_1(x) = 1$ and $f_2(x) = x$.
- Polynomial models — Terms are $f_1(x) = 1$, $f_2(x) = x$, ..., $f_p(x) = x^{p-1}$.
- Chebyshev orthogonal polynomial models — Terms are $f_1(x) = 1$, $f_2(x) = x$, ..., $f_p(x) = 2xf_{p-1}(x) - f_{p-2}(x)$.
- Fourier trigonometric polynomial models — Terms are $f_1(x) = 1/2$ and sines and cosines of different frequencies.

Examples of linear regression models with a vector of predictor variables $\boldsymbol{x} = (x_1, ..., x_N)$ include:

- Linear additive (hyperplane) models — Terms are $f_1(\boldsymbol{x}) = 1$ and $f_k(\boldsymbol{x}) = x_k$ ($k = 1, ..., N$).
- Pairwise interaction models — Terms are linear additive terms plus $g_{k_1 k_2}(\boldsymbol{x}) = x_{k_1} x_{k_2}$ ($k_1, k_2 = 1, ..., N, k_1 \neq k_2$).
- Quadratic models — Terms are pairwise interaction terms plus $h_k(\boldsymbol{x}) = x_k^2$ ($k = 1, ..., N$).
- Pure quadratic models — Terms are quadratic terms minus the $g_{k_1 k_2}(\boldsymbol{x})$ terms.

Whether or not the predictor $\boldsymbol{x}$ is a vector of predictor variables, *multivariate regression* refers to the case where the response $\boldsymbol{y} = (y_1, ..., y_M)$ is a vector of M response variables. See "Multivariate Regression" on page 8-57 for more on multivariate regression models.

Given n independent observations $(\boldsymbol{x}_1, y_1)$, ..., $(\boldsymbol{x}_n, y_n)$ of the predictor $\boldsymbol{x}$ and the response y, the linear regression model becomes an n-by-p system of equations:

$$
\underbrace{\begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix}}_{y} = \underbrace{\begin{pmatrix} f_1(\boldsymbol{x}_1) & \cdots & f_p(\boldsymbol{x}_1) \\ \vdots & \ddots & \vdots \\ f_1(\boldsymbol{x}_n) & \cdots & f_p(\boldsymbol{x}_n) \end{pmatrix}}_{X} \underbrace{\begin{pmatrix} \beta_1 \\ \vdots \\ \beta_p \end{pmatrix}}_{\beta} + \underbrace{\begin{pmatrix} \varepsilon_1 \\ \vdots \\ \varepsilon_n \end{pmatrix}}_{\varepsilon}
$$

X is the *design matrix* of the system. The columns of X are the terms of the model evaluated at the predictors. To fit the model to the data, the system must be solved for the p coefficient values in $\beta = (\beta_1, ..., \beta_p)^T$.

The MATLAB backslash operator \ (`mldivide`) solves systems of linear equations. Ignoring the unknown error ε, MATLAB estimates model coefficients in $y = X\beta$ using

```
betahat = X\y
```

where `X` is the design matrix and `y` is the vector of observed responses. MATLAB returns the least-squares solution to the system; `betahat` minimizes the norm of the *residual* vector `y-X*beta` over all `beta`. If the system is consistent, the norm is `0` and the solution is exact. In this case, the regression model interpolates the data. In more typical regression cases where $n > p$ and the system is overdetermined, the least-squares solution estimates model coefficients obscured by the error ε.

The least-squares estimator `betahat` has several important statistical properties. First, it is unbiased, with expected value β. Second, by the Gauss-Markov theorem, it has minimum variance among all unbiased estimators formed from linear combinations of the response data. Under the additional assumption that ε is normally distributed, `betahat` is a maximum likelihood estimator. The assumption also implies that the estimates themselves are normally distributed, which is useful for computing confidence intervals. Even without the assumption, by the Central Limit theorem, the estimates have an approximate normal distribution if the sample size is large enough.

Visualize the least-squares estimator as follows.
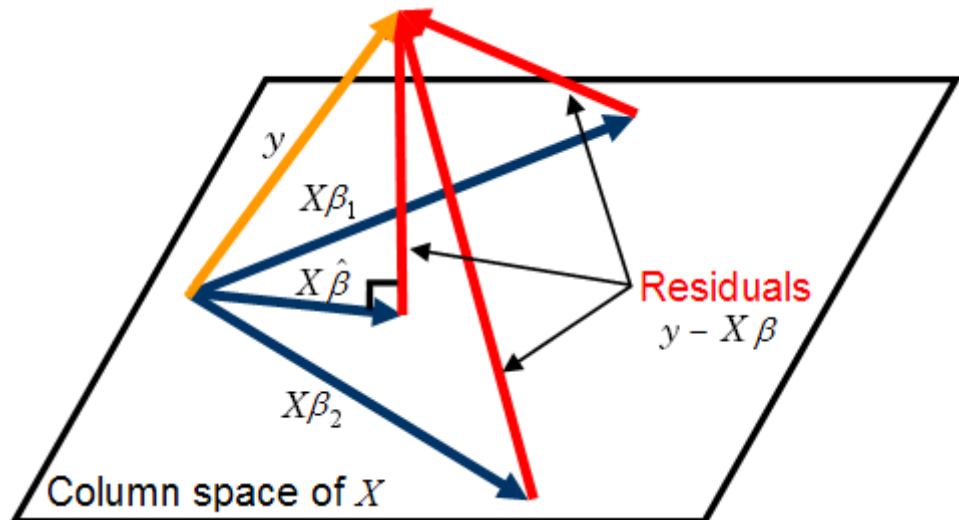
For `betahat` to minimize `norm(y-X*beta)`, `y-X*betahat` must be perpendicular to the column space of X, which contains all linear combinations of the model terms. This requirement is summarized in the *normal equations*, which express vanishing inner products between `y-X*betahat` and the columns of X:

$$X^T \left(y - X\hat{\beta} \right) = 0$$

or

$$X^T X\hat{\beta} = X^T y$$

If X is *n*-by-*p*, the normal equations are a *p*-by-*p* square system with solution `betahat = inv(X'*X)*X'*y`, where `inv` is the MATLAB inverse operator. The matrix `inv(X'*X)*X'` is the *pseudoinverse* of X, computed by the MATLAB function `pinv`.

The normal equations are often badly conditioned relative to the original system $y = X\beta$ (the coefficient estimates are much more sensitive to the model error ε), so the MATLAB backslash operator avoids solving them directly.

Instead, a *QR* (orthogonal, triangular) decomposition of X is used to create a simpler, more stable triangular system:

$$
\begin{aligned}
X^T X \hat{\beta} &= X^T y \\
(QR)^T (QR) \hat{\beta} &= (QR)^T y \\
R^T Q^T Q R \hat{\beta} &= R^T Q^T y \\
R^T R \hat{\beta} &= R^T Q^T y \\
R \hat{\beta} &= Q^T y
\end{aligned}
$$

Statistics Toolbox functions like `regress` and `regstats` call the MATLAB backslash operator to perform linear regression. The *QR* decomposition is also used for efficient computation of confidence intervals.

Once `betahat` is computed, the model can be evaluated at the predictor data:

```
yhat = X*betahat
```

or

```
yhat = X*inv(X'*X)*X'*y
```

`H = X*inv(X'*X)*X'` is the *hat matrix*. It is a square, symmetric *n*-by-*n* matrix determined by the predictor data. The diagonal elements `H(i,i)` ($i = 1, ..., n$) give the *leverage* of the *i*th observation. Since `yhat = H*y`, leverage values determine the influence of the observed response `y(i)` on the predicted response `yhat(i)`. For leverage values near 1, the predicted response approximates the observed response. The Statistics Toolbox function `leverage` computes leverage values from a *QR* decomposition of X.

Component residual values in `y-yhat` are useful for detecting failures in model assumptions. Like the errors in ε, residuals have an expected value of 0. Unlike the errors, however, residuals are correlated, with nonconstant variance. Residuals may be "Studentized" (scaled by an estimate of their standard deviation) for comparison. Studentized residuals are used by Statistics Toolbox functions like `regress` and `robustfit` to identify outliers in the data.

Multiple Linear Regression

Introduction

The system of linear equations

$$
\underbrace{\begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix}}_{y} = \underbrace{\begin{pmatrix} f_1(\boldsymbol{x}_1) & \cdots & f_p(\boldsymbol{x}_1) \\ \vdots & \ddots & \vdots \\ f_1(\boldsymbol{x}_n) & \cdots & f_p(\boldsymbol{x}_n) \end{pmatrix}}_{X} \underbrace{\begin{pmatrix} \beta_1 \\ \vdots \\ \beta_p \end{pmatrix}}_{\beta} + \underbrace{\begin{pmatrix} \varepsilon_1 \\ \vdots \\ \varepsilon_n \end{pmatrix}}_{\varepsilon}
$$

in "Linear Regression Models" on page 8-3 expresses a response y as a linear combination of model terms $f_j(\boldsymbol{x})$ (j = 1, ..., p) at each of the observations $(\boldsymbol{x}_1, y_1)$, ..., $(\boldsymbol{x}_n, y_n)$.

If the predictor $\boldsymbol{x}$ is multidimensional, so are the functions f_j that form the terms of the model. For example, if the predictor is $\boldsymbol{x} = (x_1, x_2)$, terms for the model might include $f_1(\boldsymbol{x}) = x_1$ (a linear term), $f_2(\boldsymbol{x}) = x_1^2$ (a quadratic term), and $f_3(\boldsymbol{x}) = x_1 x_2$ (a pairwise interaction term). Typically, the function $f(\boldsymbol{x}) = 1$ is included among the f_j, so that the design matrix X contains a column of 1s and the model contains a constant (y-intercept) term.

Multiple linear regression models are useful for:

- Understanding which terms $f_j(\boldsymbol{x})$ have greatest effect on the response (coefficients β_j with greatest magnitude)
- Finding the direction of the effects (signs of the β_j)
- Predicting unobserved values of the response ($y(\boldsymbol{x})$ for new $\boldsymbol{x}$)

The Statistics Toolbox functions `regress` and `regstats` are used for multiple linear regression analysis.

Programmatic Multiple Linear Regression

For example, the file `moore.mat` contains the 20-by-6 data matrix `moore`. The first five columns are measurements of five predictor variables. The final column contains the observed responses. Coefficient estimates `betahat` for a linear additive model are found with `regress` as follows:

```
load moore
X1 = [ones(size(moore,1),1) moore(:,1:5)];
y = moore(:,6);
betahat = regress(y,X1)
betahat =
    -2.1561
    -0.0000
     0.0013
     0.0001
     0.0079
     0.0001
```

Before using `regress` give the design matrix `X1` a first column of 1s to include a constant term in the model, `betahat(1)`.

The MATLAB backslash (`mldivide`) operator, which `regress` calls, obtains the same result:

```
betahat = X1\y
betahat =
    -2.1561
    -0.0000
     0.0013
     0.0001
     0.0079
     0.0001
```

The advantage of working with `regress` is that it allows for additional inputs and outputs relevant to statistical analysis of the regression. For example:

```
alpha = 0.05;
[betahat,Ibeta,res,Ires,stats] = regress(y,X1,alpha);
```

returns not only the coefficient estimates in `betahat`, but also

- `Ibeta` — A p-by-2 matrix of 95% confidence intervals for the coefficient estimates, using a `100*(1-alpha)`% confidence level. The first column contains lower confidence bounds for each of the p coefficient estimates; the second column contains upper confidence bounds.

- `res` — An n-by-1 vector of residuals.

- `Ires` — An n-by-2 matrix of intervals that can be used to diagnose outliers. If the interval `Ires(i,:)` for observation `i` does not contain zero, the corresponding residual is larger than expected in `100*(1-alpha)`% of new observations, suggesting an outlier.

- `stats` — A 1-by-4 vector that contains, in order, the R^2 statistic, the F statistic and its p-value, and an estimate of the error variance. The statistics are computed assuming the model contains a constant term, and are incorrect otherwise.

Visualize the residuals, in case (row number) order, with the `rcoplot` function:

```
rcoplot(res,Ires)
```
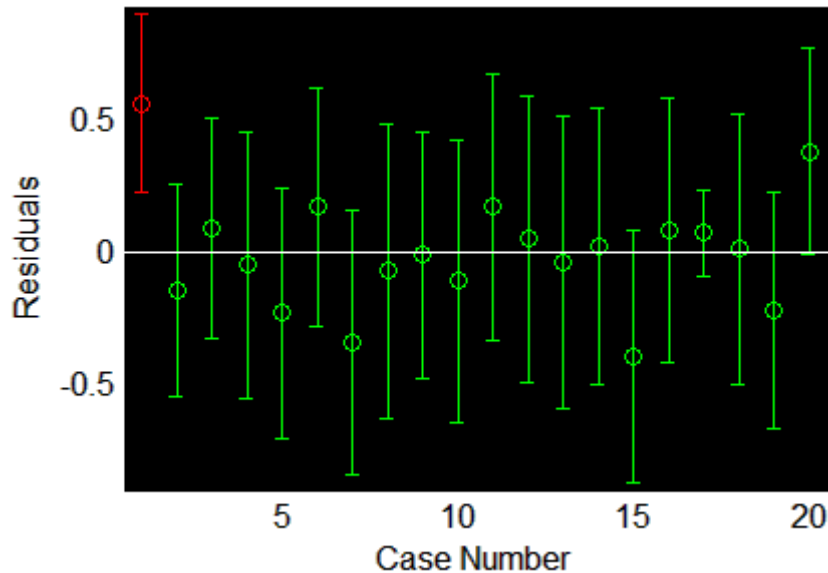
The interval around the first residual, shown in red when plotted, does not contain zero. This indicates that the residual is larger than expected in 95% of new observations, and suggests the data point is an outlier.

Outliers in regression appear for a variety of reasons:

1 If there is sufficient data, 5% of the residuals, by the definition of `rint`, are too big.

2 If there is a systematic error in the model (that is, if the model is not appropriate for generating the data under model assumptions), the mean of the residuals is not zero.

3 If the errors in the model are not normally distributed, the distributions of the residuals may be skewed or leptokurtic (with heavy tails and more outliers).

When errors are normally distributed, `Ires(i,:)` is a confidence interval for the mean of `res(i)` and checking if the interval contains zero is a test of the null hypothesis that the residual has zero mean.

Interactive Multiple Linear Regression

The function `regstats` also performs multiple linear regression, but computes more statistics than `regress`. By default, `regstats` automatically adds a first column of 1s to the design matrix (necessary for computing the F statistic and its p-value), so a constant term should not be included explicitly as for `regress`. For example:

```
X2 = moore(:,1:5);
stats = regstats(y,X2);
```

creates a structure `stats` with fields containing regression statistics. An optional input argument allows you to specify which statistics are computed.

To interactively specify the computed statistics, call `regstats` without an output argument. For example:

```
regstats(y,X2)
```

opens the following interface.

Select the check boxes corresponding to the statistics you want to compute and click **OK**. Selected statistics are returned to the MATLAB workspace. Names

of container variables for the statistics appear on the right-hand side of the interface, where they can be changed to any valid MATLAB variable name.

Tabulating Diagnostic Statistics

The regstats function computes statistics that are typically used in regression diagnostics. Statistics can be formatted into standard tabular displays in a variety of ways. For example, the tstat field of the stats output structure of regstats is itself a structure containing statistics related to the estimated coefficients of the regression. Dataset arrays (see "Dataset Arrays" on page 2-23) provide a natural tabular format for the information:

```
t = stats.tstat;
CoeffTable = dataset({t.beta,'Coef'},{t.se,'StdErr'}, ...
                     {t.t,'tStat'},{t.pval,'pVal'})
CoeffTable =
    Coef          StdErr         tStat        pVal
        -2.1561       0.91349       -2.3603       0.0333
     -9.0116e-006    0.00051835    -0.017385     0.98637
        0.0013159     0.0012635      1.0415       0.31531
        0.0001278    7.6902e-005     1.6618       0.11876
        0.0078989        0.014       0.56421      0.58154
        0.00014165   7.3749e-005     1.9208       0.075365
```

The MATLAB function fprintf gives you control over tabular formatting. For example, the fstat field of the stats output structure of regstats is a structure with statistics related to the analysis of variance (ANOVA) of the regression. The following commands produce a standard regression ANOVA table:

```
f = stats.fstat;

fprintf('\n')
fprintf('Regression ANOVA');
fprintf('\n\n')

fprintf('%6s','Source');
fprintf('%10s','df','SS','MS','F','P');
fprintf('\n')

fprintf('%6s','Regr');
```

```
fprintf('%10.4f',f.dfr,f.ssr,f.ssr/f.dfr,f.f,f.pval);
fprintf('\n')

fprintf('%6s','Resid');
fprintf('%10.4f',f.dfe,f.sse,f.sse/f.dfe);
fprintf('\n')

fprintf('%6s','Total');
fprintf('%10.4f',f.dfe+f.dfr,f.sse+f.ssr);
fprintf('\n')
```

The result looks like this:

```
Regression ANOVA

Source       df        SS        MS         F         P
  Regr    5.0000    4.1084    0.8217   11.9886    0.0001
 Resid   14.0000    0.9595    0.0685
 Total   19.0000    5.0679
```

Robust Regression

- "Introduction" on page 8-14
- "Programmatic Robust Regression" on page 8-15
- "Interactive Robust Regression" on page 8-16

Introduction

The models described in "Linear Regression Models" on page 8-3 are based on certain assumptions, such as a normal distribution of errors in the observed responses. If the distribution of errors is asymmetric or prone to outliers, model assumptions are invalidated, and parameter estimates, confidence intervals, and other computed statistics become unreliable. The Statistics Toolbox function robustfit is useful in these cases. The function implements a robust fitting method that is less sensitive than ordinary least squares to large changes in small parts of the data.

Robust regression works by assigning a weight to each data point. Weighting is done automatically and iteratively using a process called *iteratively*

reweighted least squares. In the first iteration, each point is assigned equal weight and model coefficients are estimated using ordinary least squares. At subsequent iterations, weights are recomputed so that points farther from model predictions in the previous iteration are given lower weight. Model coefficients are then recomputed using weighted least squares. The process continues until the values of the coefficient estimates converge within a specified tolerance.

Programmatic Robust Regression

The example in "Multiple Linear Regression" on page 8-8 shows an outlier when ordinary least squares is used to model the response variable as a linear combination of the five predictor variables. To determine the influence of the outlier, compare the coefficient estimates computed by `regress`:

```
load moore
X1 = [ones(size(moore,1),1) moore(:,1:5)];
y = moore(:,6);
betahat = regress(y,X1)
betahat =
   -2.1561
   -0.0000
    0.0013
    0.0001
    0.0079
    0.0001
```

to those computed by `robustfit`:

```
X2 = moore(:,1:5);
robustbeta = robustfit(X2,y)
robustbeta =
   -1.7516
    0.0000
    0.0009
    0.0002
    0.0060
    0.0001
```

By default, `robustfit` automatically adds a first column of 1s to the design matrix, so a constant term does not have to be included explicitly as for

regress. In addition, the order of inputs is reversed for `robustfit` and `regress`.

To understand the difference in the coefficient estimates, look at the final weights given to the data points in the robust fit:

```
[robustbeta,stats] = robustfit(X2,y);
stats.w'
ans =
  Columns 1 through 5
    0.0246    0.9986    0.9763    0.9323    0.9704
  Columns 6 through 10
    0.8597    0.9180    0.9992    0.9590    0.9649
  Columns 11 through 15
    0.9769    0.9868    0.9999    0.9976    0.8122
  Columns 16 through 20
    0.9733    0.9892    0.9988    0.8974    0.6774
```

The first data point has a very low weight compared to the other data points, and so is effectively ignored in the robust regression.

Interactive Robust Regression

The `robustdemo` function shows the difference between ordinary least squares and robust fitting for data with a single predictor. You can use data provided with the demo, or you can supply your own data. The following steps show you how to use `robustdemo`.

1 Start the demo. To begin using `robustdemo` with the built-in data, simply type the function name:

```
robustdemo
```

Least squares: Y = -0.188327 + 1.10351*X   RMS error = 2.21375
Robust: Y = -1.77278 + 1.50415*X   RMS error = 1.43663

The resulting figure shows a scatter plot with two fitted lines. The red line is the fit using ordinary least-squares regression. The green line is the fit using robust regression. At the bottom of the figure are the equations for the fitted lines, together with the estimated root mean squared errors for each fit.

2 View leverages and robust weights. Right-click on any data point to see its least-squares leverage and robust weight.

Least squares: Y = -0.188327 + 1.10351*X   RMS error = 2.21375
Robust: Y = -1.77278 + 1.50415*X   RMS error = 1.43663

In the built-in data, the right-most point has a relatively high leverage of 0.35. The point exerts a large influence on the least-squares fit, but its small robust weight shows that it is effectively excluded from the robust fit.

3 **See how changes in the data affect the fits.** With the left mouse button, click and hold on any data point and drag it to a new location. When you release the mouse button, the displays update.

Least squares: Y = -0.928677 + 1.30648*X   RMS error = 1.28809

Robust: Y = -1.07823 + 1.34094*X   RMS error = 1.34891

Bringing the right-most data point closer to the least-squares line makes the two fitted lines nearly identical. The adjusted right-most data point has significant weight in the robust fit.

Stepwise Regression

Introduction

Multiple linear regression models, as described in "Multiple Linear Regression" on page 8-8, are built from a potentially large number of predictive terms. The number of interaction terms, for example, increases exponentially with the number of predictor variables. If there is no theoretical

basis for choosing the form of a model, and no assessment of correlations among terms, it is possible to include redundant terms in a model that confuse the identification of significant effects.

Stepwise regression is a systematic method for adding and removing terms from a multilinear model based on their statistical significance in a regression. The method begins with an initial model and then compares the explanatory power of incrementally larger and smaller models. At each step, the p-value of an F-statistic is computed to test models with and without a potential term. If a term is not currently in the model, the null hypothesis is that the term would have a zero coefficient if added to the model. If there is sufficient evidence to reject the null hypothesis, the term is added to the model. Conversely, if a term is currently in the model, the null hypothesis is that the term has a zero coefficient. If there is insufficient evidence to reject the null hypothesis, the term is removed from the model. The method proceeds as follows:

1 Fit the initial model.

2 If any terms not in the model have p-values less than an entrance tolerance (that is, if it is unlikely that they would have zero coefficient if added to the model), add the one with the smallest p-value and repeat this step; otherwise, go to step 3.

3 If any terms in the model have p-values greater than an exit tolerance (that is, if it is unlikely that the hypothesis of a zero coefficient can be rejected), remove the one with the largest p-value and go to step 2; otherwise, end.

Depending on the terms included in the initial model and the order in which terms are moved in and out, the method may build different models from the same set of potential terms. The method terminates when no single step improves the model. There is no guarantee, however, that a different initial model or a different sequence of steps will not lead to a better fit. In this sense, stepwise models are locally optimal, but may not be globally optimal.

Statistics Toolbox functions for stepwise regression are:

- `stepwisefit` — A function that proceeds automatically from a specified initial model and entrance/exit tolerances

- `stepwise` — An interactive tool that allows you to explore individual steps in the regression

Programmatic Stepwise Regression

For example, load the data in `hald.mat`, which contains observations of the heat of reaction of various cement mixtures:

```
load hald
whos
  Name          Size      Bytes    Class    Attributes

  Description   22x58     2552     char
  hald          13x5       520     double
  heat          13x1       104     double
  ingredients   13x4       416     double
```

The response (`heat`) depends on the quantities of the four predictors (the columns of `ingredients`).

Use `stepwisefit` to carry out the stepwise regression algorithm, beginning with no terms in the model and using entrance/exit tolerances of 0.05/0.10 on the *p*-values:

```
stepwisefit(ingredients,heat,...
            'penter',0.05,'premove',0.10);
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]
```

`stepwisefit` automatically includes an intercept term in the model, so you do not add it explicitly to `ingredients` as you would for `regress`. For terms not in the model, coefficient estimates and their standard errors are those that result if the term is added.

The `inmodel` parameter is used to specify terms in an initial model:

```
initialModel = ...
            [false true false false]; % Force in 2nd term
```

```
stepwisefit(ingredients,heat,...
            'inmodel',initialModel,...
            'penter',.05,'premove',0.10);
Initial columns included:  2
Step 1, added column 1, p=2.69221e-007
Final columns included:  1 2
    '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]
```

The preceding two models, built from different initial models, use different subsets of the predictive terms. Terms 2 and 4, swapped in the two models, are highly correlated:

```
term2 = ingredients(:,2);
term4 = ingredients(:,4);
R = corrcoef(term2,term4)
R =
    1.0000   -0.9730
   -0.9730    1.0000
```

To compare the models, use the stats output of stepwisefit:

```
[betahat1,se1,pval1,inmodel1,stats1] = ...
         stepwisefit(ingredients,heat,...
         'penter',.05,'premove',0.10,...
         'display','off');
[betahat2,se2,pval2,inmodel2,stats2] = ...
         stepwisefit(ingredients,heat,...
                     'inmodel',initialModel,...
                     'penter',.05,'premove',0.10,...
                     'display','off');
RMSE1 = stats1.rmse
RMSE1 =
    2.7343
RMSE2 = stats2.rmse
RMSE2 =
    2.4063
```

The second model has a lower Root Mean Square Error (RMSE).

An *added variable plot* is used to determine the unique effect of adding a new term to a model. The plot shows the relationship between the part of the response unexplained by terms already in the model and the part of the new term unexplained by terms already in the model. The "unexplained" parts are measured by the residuals of the respective regressions. A scatter of the residuals from the two regressions forms the added variable plot.

For example, suppose you want to add `term2` to a model that already contains the single term `term1`. First, consider the ability of `term2` alone to explain the response:

```
load hald
term2 = ingredients(:,2);

[b2,Ib2,res2] = regress(heat,[ones(size(term2)) term2]);

scatter(term2,heat)
xlabel('Term 2')
ylabel('Heat')
hold on
x2 = 20:80;
y2 = b2(1) + b2(2)*x2;
plot(x2,y2,'r')
title('{\bf Response Explained by Term 2: Ignoring Term 1}')
```
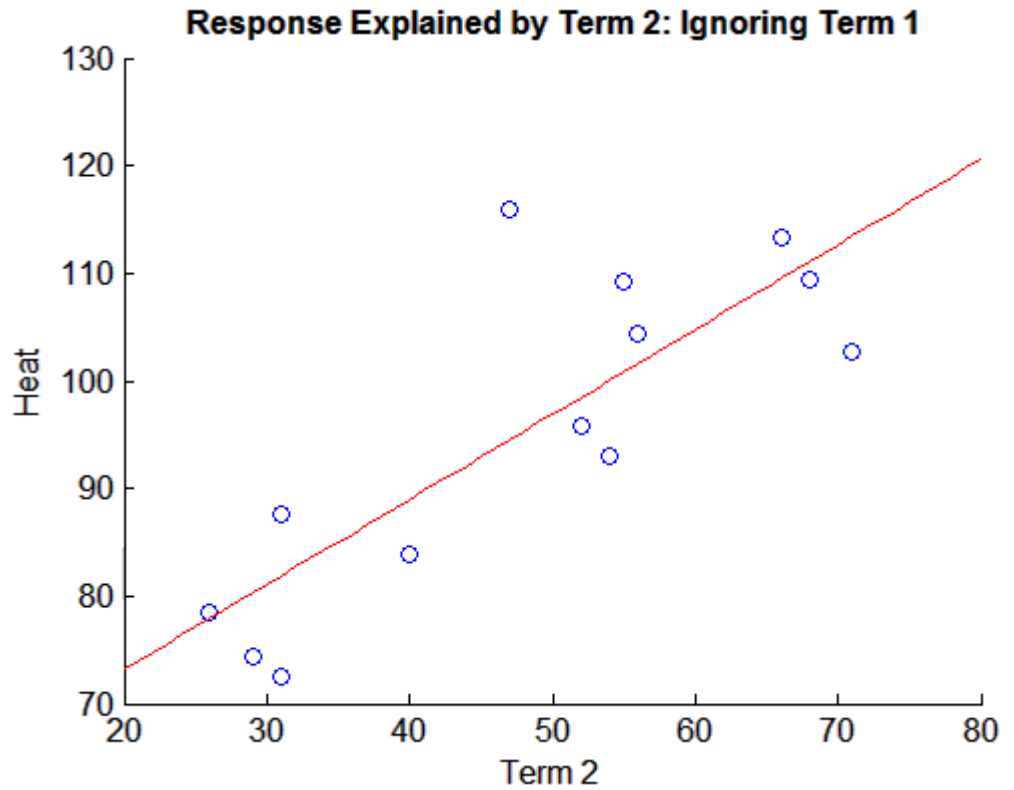
Next, consider the following regressions involving the model term `term1`:

```
term1 = ingredients(:,1);
[b1,Ib1,res1] = regress(heat,[ones(size(term1)) term1]);
[b21,Ib21,res21] = regress(term2,[ones(size(term1)) term1]);
bres = regress(res1,[ones(size(res21)) res21]);
```

A scatter of the residuals `res1` vs. the residuals `res12` forms the added variable plot:

```
figure
scatter(res21,res1)
xlabel('Residuals: Term 2 on Term 1')
ylabel('Residuals: Heat on Term 1')
hold on
```

```
xres = -30:30;
yres = bres(1) + bres(2)*xres;
plot(xres,yres,'r')
title('{\bf Response Explained by Term 2: Adjusted for Term 1}')
```


Response Explained by Term 2: Adjusted for Term 1
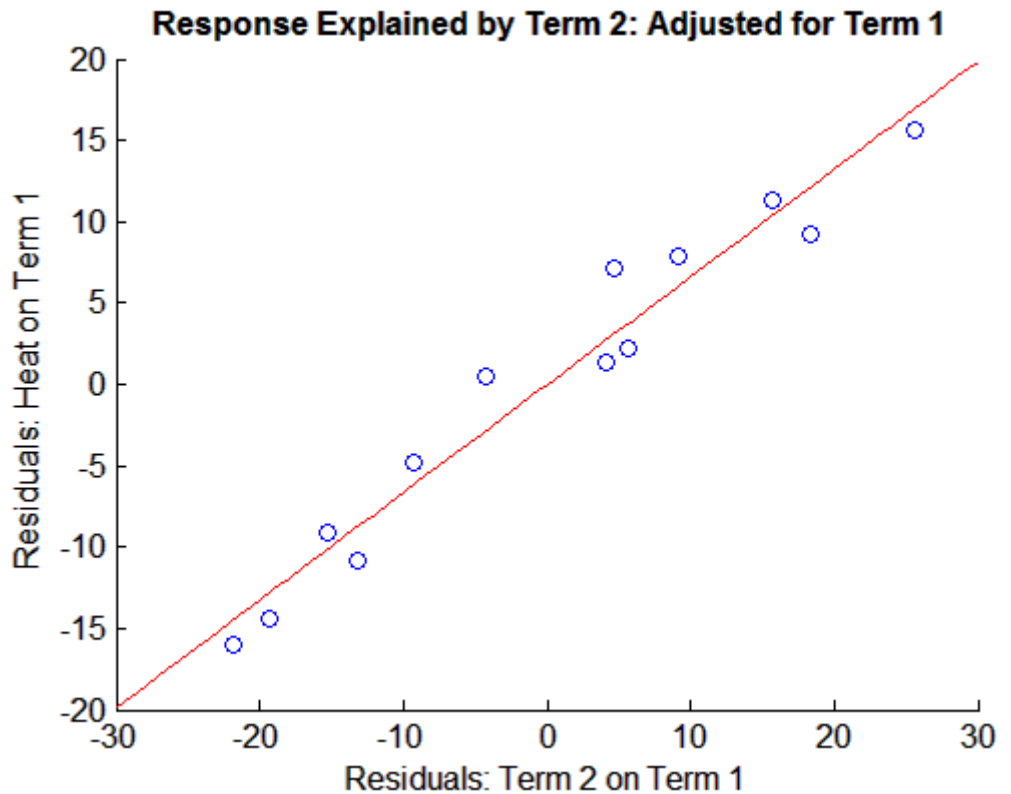
Since the plot adjusted for `term1` shows a stronger relationship (less variation along the fitted line) than the plot ignoring `term1`, the two terms act jointly to explain extra variation. In this case, adding `term2` to a model consisting of `term1` would reduce the RMSE.

The Statistics Toolbox function `addedvarplot` produces added variable plots. The previous plot is essentially the one produced by the following:

```
figure
```

```
inmodel = [true false false false];
addedvarplot(ingredients,heat,2,inmodel)
```
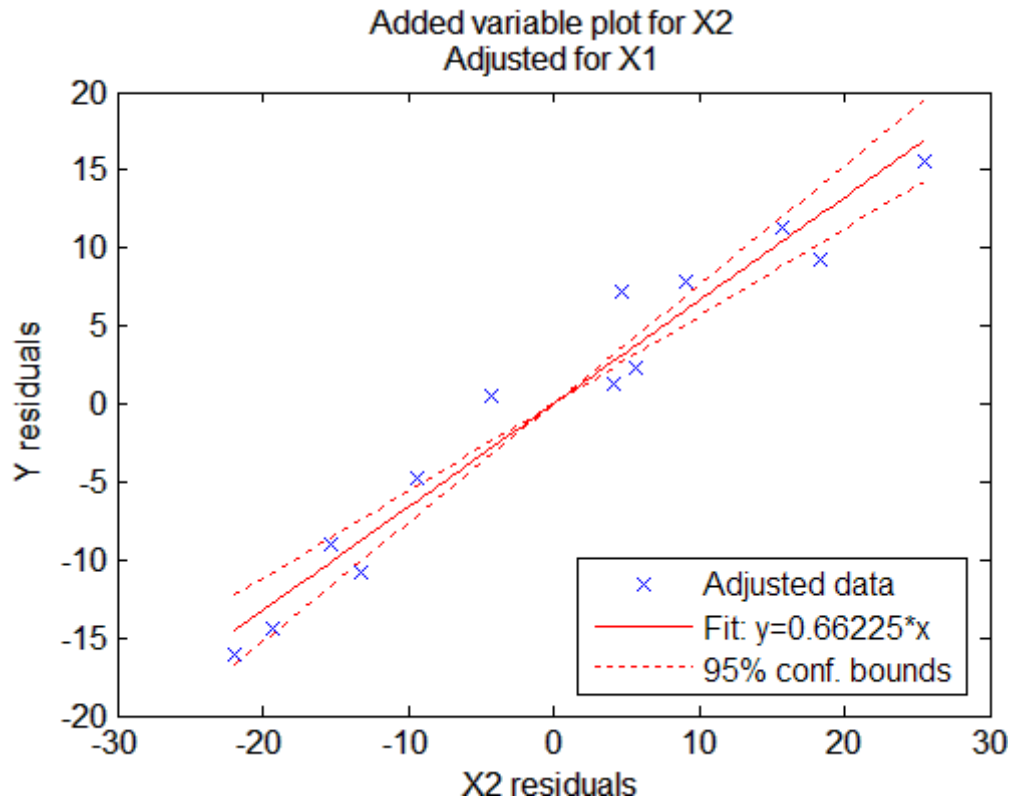

In addition to the scatter of residuals, the plot shows 95% confidence intervals on predictions from the fitted line. The fitted line has intercept zero because, under the assumptions outlined in "Linear Regression Models" on page 8-3, both of the plotted variables have mean zero. The slope of the fitted line is the coefficient that `term2` would have if it was added to the model with `term1`.

The `addevarplot` function is useful for considering the unique effect of adding a new term to an existing model with any number of terms.

Interactive Stepwise Regression

The `stepwise` interface provides interactive features that allow you to investigate individual steps in a stepwise regression, and to build models from arbitrary subsets of the predictive terms. To open the interface with data from `hald.mat`:

```
load hald
stepwise(ingredients,heat)
```

The upper left of the interface displays estimates of the coefficients for all potential terms, with horizontal bars indicating 90% (colored) and 95% (grey) confidence intervals. The red color indicates that, initially, the terms are not in the model. Values displayed in the table are those that would result if the terms were added to the model.

The middle portion of the interface displays summary statistics for the entire model. These statistics are updated with each step.

The lower portion of the interface, **Model History**, displays the RMSE for the model. The plot tracks the RMSE from step to step, so you can compare the optimality of different models. Hover over the blue dots in the history to see which terms were in the model at a particular step. Click on a blue dot in the history to open a copy of the interface initialized with the terms in the model at that step.

Initial models, as well as entrance/exit tolerances for the p-values of F-statistics, are specified using additional input arguments to `stepwise`. Defaults are an initial model with no terms, an entrance tolerance of 0.05, and an exit tolerance of 0.10.

To center and scale the input data (compute z-scores) to improve conditioning of the underlying least-squares problem, select `Scale Inputs` from the **Stepwise** menu.

You proceed through a stepwise regression in one of two ways:

1 Click **Next Step** to select the recommended next step. The recommended next step either adds the most significant term or removes the least significant term. When the regression reaches a local minimum of RMSE, the recommended next step is "Move no terms." You can perform all of the recommended steps at once by clicking **All Steps**.

2 Click a line in the plot or in the table to toggle the state of the corresponding term. Clicking a red line, corresponding to a term not currently in the model, adds the term to the model and changes the line to blue. Clicking a blue line, corresponding to a term currently in the model, removes the term from the model and changes the line to red.

To call `addedvarplot` and produce an added variable plot from the `stepwise` interface, select **Added Variable Plot** from the **Stepwise** menu. A list of terms is displayed. Select the term you want to add, and then click **OK**.

Click **Export** to display a dialog box that allows you to select information from the interface to save to the MATLAB workspace. Check the information you want to export and, optionally, change the names of the workspace variables to be created. Click **OK** to export the information.

Ridge Regression

- "Introduction" on page 8-29
- "Example: Ridge Regression" on page 8-30

Introduction

Coefficient estimates for the models described in "Multiple Linear Regression" on page 8-8 rely on the independence of the model terms. When terms are correlated and the columns of the design matrix X have an approximate linear dependence, the matrix $(X^TX)^{-1}$ becomes close to singular. As a result, the least-squares estimate

$$\hat{\beta} = (X^T X)^{-1} X^T y$$

becomes highly sensitive to random errors in the observed response y, producing a large variance. This situation of *multicollinearity* can arise, for example, when data are collected without an experimental design.

Ridge regression addresses the problem by estimating regression coefficients using

$$\hat{\beta} = (X^T X + kI)^{-1} X^T y$$

where k is the *ridge parameter* and I is the identity matrix. Small positive values of k 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.

The Statistics Toolbox function `ridge` carries out ridge regression.

Example: Ridge Regression

For example, load the data in `acetylene.mat`, with observations of the predictor variables `x1`, `x2`, `x3`, and the response variable `y`:

```
load acetylene
```

Plot the predictor variables 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 predictor variables.

Use `ridge` and `x2fx` to compute coefficient estimates for a multilinear model with interaction terms, for a range of ridge parameters:

```
X = [x1 x2 x3];
D = x2fx(X,'interaction');
```
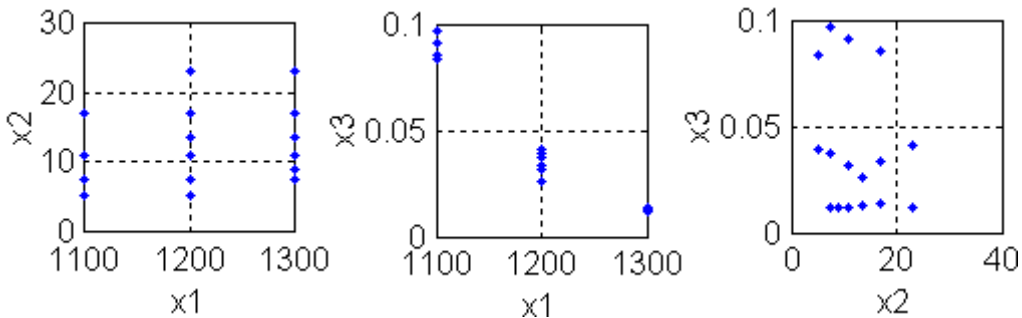
```
D(:,1) = []; % No constant term
k = 0:1e-5:5e-3;
betahat = ridge(y,D,k);
```

Plot the ridge trace:

```
figure
plot(k,betahat,'LineWidth',2)
ylim([-100 100])
grid on
xlabel('Ridge Parameter')
ylabel('Standardized Coefficient')
title('{\bf Ridge Trace}')
legend('x1','x2','x3','x1x2','x1x3','x2x3')
```
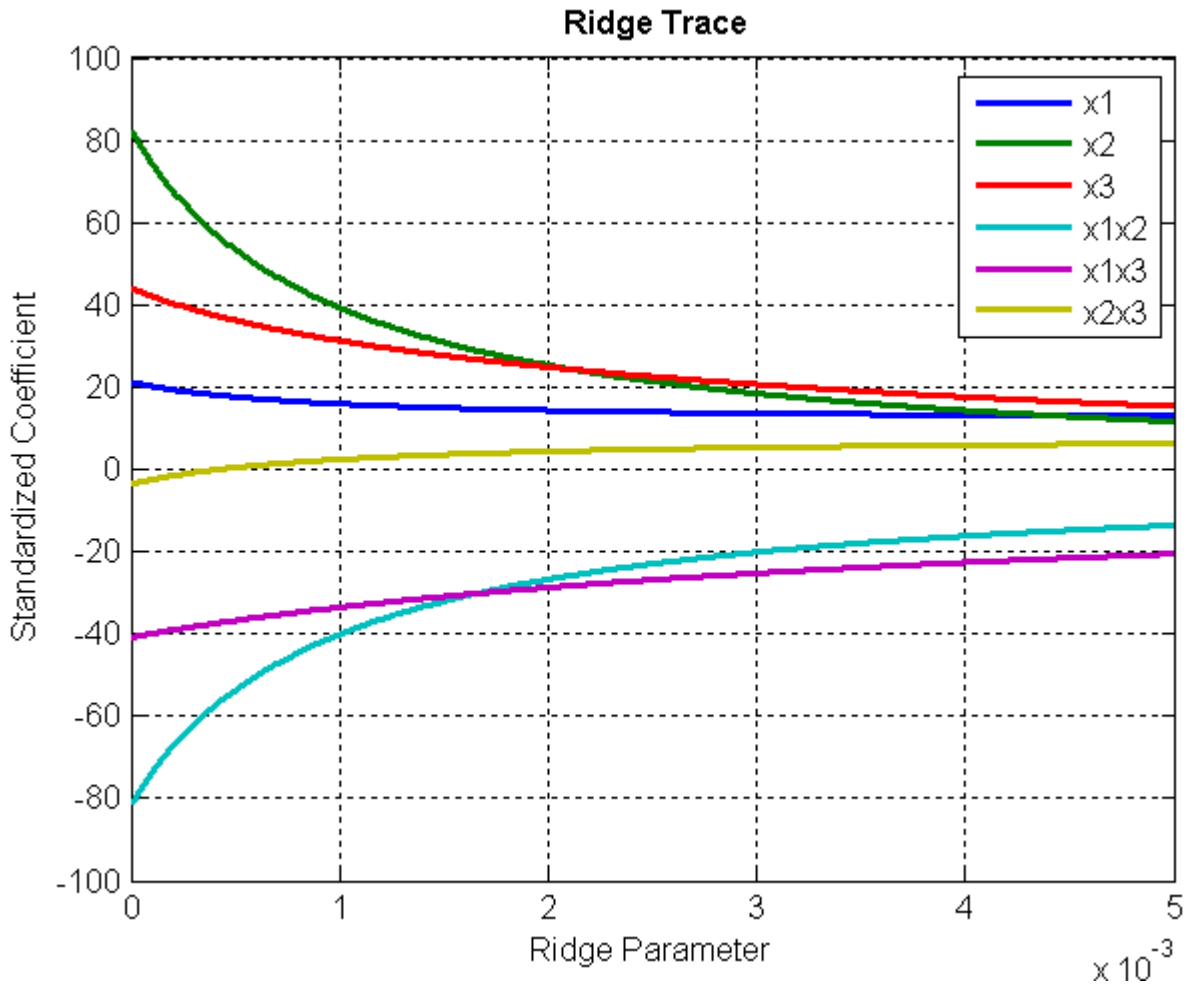
The estimates stabilize to the right of the plot. Note that the coefficient of the x2x3 interaction term changes sign at a value of the ridge parameter $\approx 5 \times 10^{-4}$.

Partial Least Squares

- "Introduction" on page 8-33

• "Example: Partial Least Squares" on page 8-33

Introduction

Partial least-squares (*PLS*) regression is a technique used with data that contain correlated predictor variables. This technique constructs new predictor variables, known as *components*, as linear combinations of the original predictor variables. PLS constructs these components while considering the observed response values, leading to a parsimonious model with reliable predictive power.

The technique is something of a cross between multiple linear regression and principal component analysis:

• Multiple linear regression finds a combination of the predictors that best fit a response.

• Principal component analysis finds combinations of the predictors with large variance, reducing correlations. The technique makes no use of response values.

• PLS finds combinations of the predictors that have a large covariance with the response values.

PLS therefore combines information about the variances of both the predictors and the responses, while also considering the correlations among them.

PLS shares characteristics with other regression and feature transformation techniques. It is similar to ridge regression in that it is used in situations with correlated predictors. It is similar to stepwise regression (or more general feature selection techniques) in that it can be used to select a smaller set of model terms. PLS differs from these methods, however, by transforming the original predictor space into the new component space.

The Statistics Toolbox function `plsregress` carries out PLS regression.

Example: Partial Least Squares

For example, consider the data on biochemical oxygen demand in `moore.mat`, padded with noisy versions of the predictors to introduce correlations:

```
load moore
```

```
y = moore(:,6);              % Response
X0 = moore(:,1:5);           % Original predictors
X1 = X0+10*randn(size(X0));  % Correlated predictors
X = [X0,X1];
```

Use `plsregress` to perform PLS regression with the same number of components as predictors, then plot the percentage variance explained in the response as a function of the number of components:

```
[XL,yl,XS,YS,beta,PCTVAR] = plsregress(X,y,10);

plot(1:10,cumsum(100*PCTVAR(2,:)),'-bo');
xlabel('Number of PLS components');
ylabel('Percent Variance Explained in y');
```
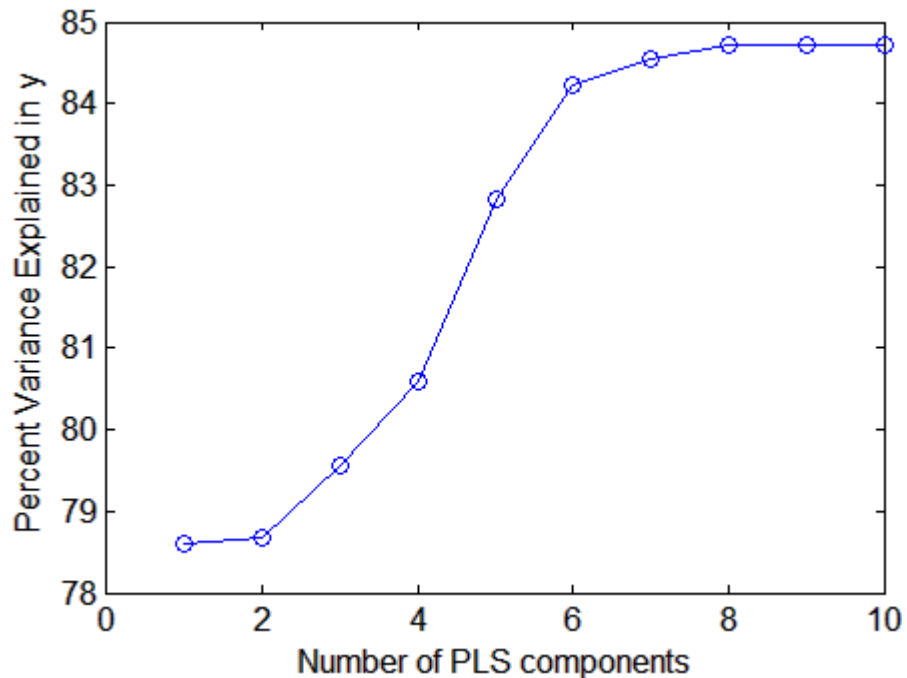

Choosing the number of components in a PLS model is a critical step. The plot gives a rough indication, showing nearly 80% of the variance in y explained

by the first component, with as many as five additional components making significant contributions.

The following computes the six-component model:

```
[XL,yl,XS,YS,beta,PCTVAR,MSE,stats] = plsregress(X,y,6);
yfit = [ones(size(X,1),1) X]*beta;

plot(y,yfit,'o')
```
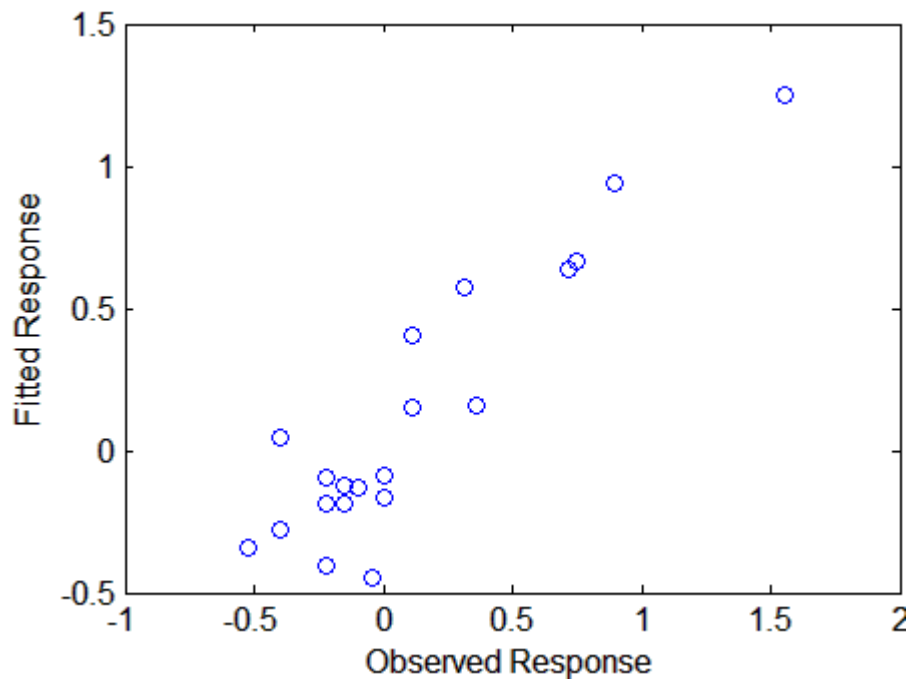

The scatter shows a reasonable correlation between fitted and observed responses, and this is confirmed by the R^2 statistic:

```
TSS = sum((y-mean(y)).^2);
RSS = sum((y-yfit).^2);
Rsquared = 1 - RSS/TSS
Rsquared =
    0.8421
```

A plot of the weights of the ten predictors in each of the six components shows that two of the components (the last two computed) explain the majority of the variance in X:

```
plot(1:10,stats.W,'o-');
legend({'c1','c2','c3','c4','c5','c6'},'Location','NW')
xlabel('Predictor');
ylabel('Weight');
```
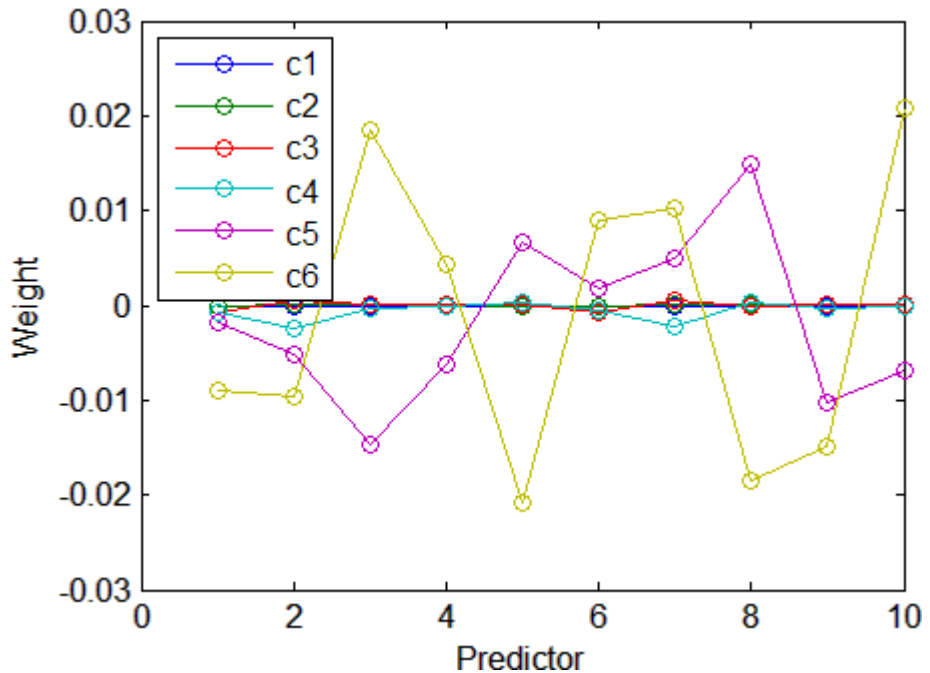

A plot of the mean-squared errors suggests that as few as two components may provide an adequate model:

```
[axes,h1,h2] = plotyy(0:6,MSE(1,:),0:6,MSE(2,:));
set(h1,'Marker','o')
set(h2,'Marker','o')
legend('MSE Predictors','MSE Response')
xlabel('Number of Components')
```
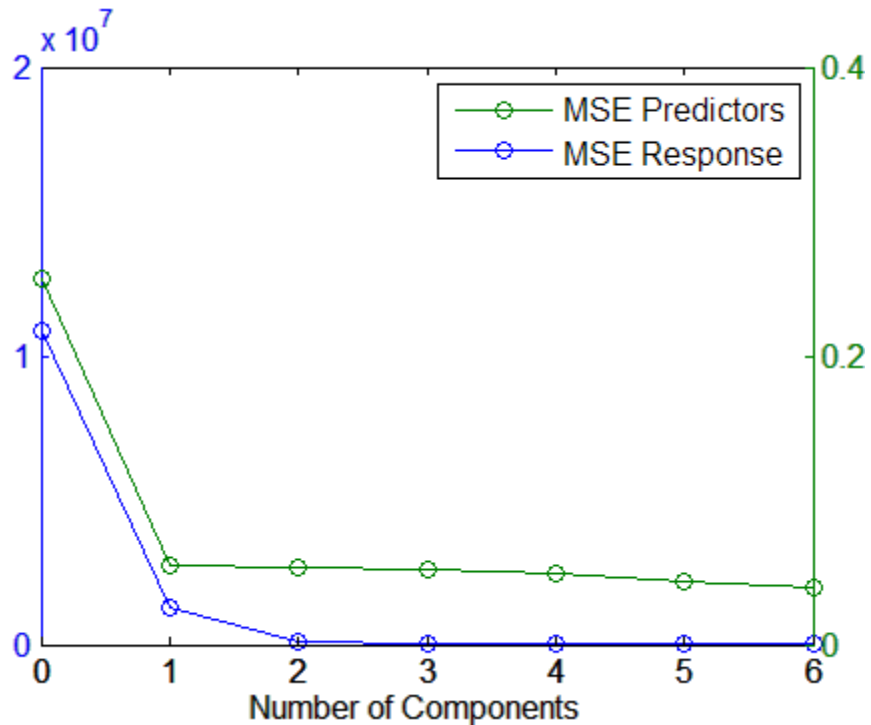
The calculation of mean-squared errors by `plsregress` is controlled by optional parameter name/value pairs specifying cross-validation type and the number of Monte Carlo repetitions.

Polynomial Models

- "Introduction" on page 8-37
- "Programmatic Polynomial Regression" on page 8-38
- "Interactive Polynomial Regression" on page 8-43

Introduction

Polynomial models are a special case of the linear models discussed in "Linear Regression Models" on page 8-3. Polynomial models have the advantages of being simple, familiar in their properties, and reasonably flexible for following

data trends. They are also robust with respect to changes in the location and scale of the data (see "Conditioning Polynomial Fits" on page 8-41). However, polynomial models may be poor predictors of new values. They oscillate between data points, especially as the degree is increased to improve the fit. Asymptotically, they follow power functions, leading to inaccuracies when extrapolating other long-term trends. Choosing a polynomial model is often a trade-off between a simple description of overall data trends and the accuracy of predictions made from the model.

Programmatic Polynomial Regression

- "Functions for Polynomial Fitting" on page 8-38

- "Displaying Polynomial Fits" on page 8-40

- "Conditioning Polynomial Fits" on page 8-41

Functions for Polynomial Fitting. To fit polynomials to data, MATLAB and Statistics Toolbox software offer a number of dedicated functions. The MATLAB function `polyfit` computes least-squares coefficient estimates for polynomials of arbitrary degree. For example:

```
x = 0:5; % x data
y = [2 1 4 4 3 2]; % y data
p = polyfit(x,y,3) % Degree 3 fit
p =
   -0.1296    0.6865   -0.1759    1.6746
```

Polynomial coefficients in p are listed from highest to lowest degree, so $p(x) \approx -0.13\,x^3 + 0.69\,x^2 - 0.18\,x + 1.67$. For convenience, `polyfit` sets up the Vandermonde design matrix (`vander`) and calls backslash (`mldivide`) to perform the fit.

Once the coefficients of a polynomial are collected in a vector p, use the MATLAB function `polyval` to evaluate the polynomial at arbitrary inputs. For example, the following plots the data and the fit over a range of inputs:

```
plot(x,y,'ro','LineWidth',2) % Plot data
hold on
xfit = -1:0.01:6;
yfit = polyval(p,xfit);
```

```
plot(xfit,yfit,'LineWidth',2) % Plot fit
ylim([0,5])
grid on
```
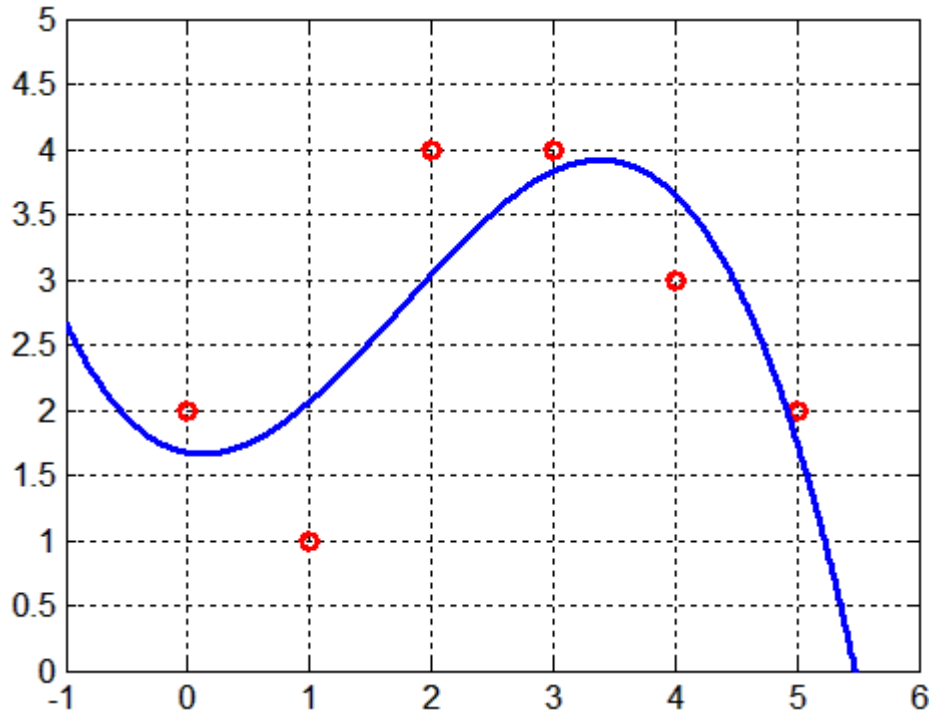

Use the MATLAB function `roots` to find the roots of p:

```
r = roots(p)
r =
   5.4786
  -0.0913 + 1.5328i
  -0.0913 - 1.5328i
```

The MATLAB function `poly` solves the inverse problem, finding a polynomial with specified roots. `poly` is the inverse of `roots` up to ordering, scaling, and round-off error.

An optional output from `polyfit` is passed to `polyval` or to the Statistics Toolbox function `polyconf` to compute prediction intervals for the fit. For example, the following computes 95% prediction intervals for new observations at each value of the predictor `x`:

```
[p,S] = polyfit(x,y,3);
[yhat,delta] = polyconf(p,x,S);
PI = [yhat-delta;yhat+delta]'
PI =
   -5.3022    8.6514
   -4.2068    8.3179
   -2.9899    9.0534
   -2.1963    9.8471
   -2.6036    9.9211
   -5.2229    8.7308
```

Optional input arguments to `polyconf` allow you to compute prediction intervals for estimated values (`yhat`) as well as new observations, and to compute the bounds simultaneously for all `x` instead of nonsimultaneously (the default). The confidence level for the intervals can also be set.

Displaying Polynomial Fits. The documentation example function `polydemo` combines the functions `polyfit`, `polyval`, `roots`, and `polyconf` to produce a formatted display of data with a polynomial fit.

Note Statistics Toolbox documentation example files are located in the `\help\toolbox\stats\examples` subdirectory of your MATLAB root directory (`matlabroot`). This subdirectory is not on the MATLAB path at installation. To use the M-files in this subdirectory, either add the subdirectory to the MATLAB path (`addpath`) or make the subdirectory your current working directory (`cd`).

For example, the following uses `polydemo` to produce a display of simulated data with a quadratic trend, a fitted polynomial, and 95% prediction intervals for new observations:

```
x = -5:5;
y = x.^2 - 5*x - 3 + 5*randn(size(x));
p = polydemo(x,y,2,0.05)
```

```
p =
    0.8107   -4.5054   -1.1862
```


Fit: $0.81\,x^2 - 4.51\,x - 1.19$

Real Roots: 5.81   -0.25
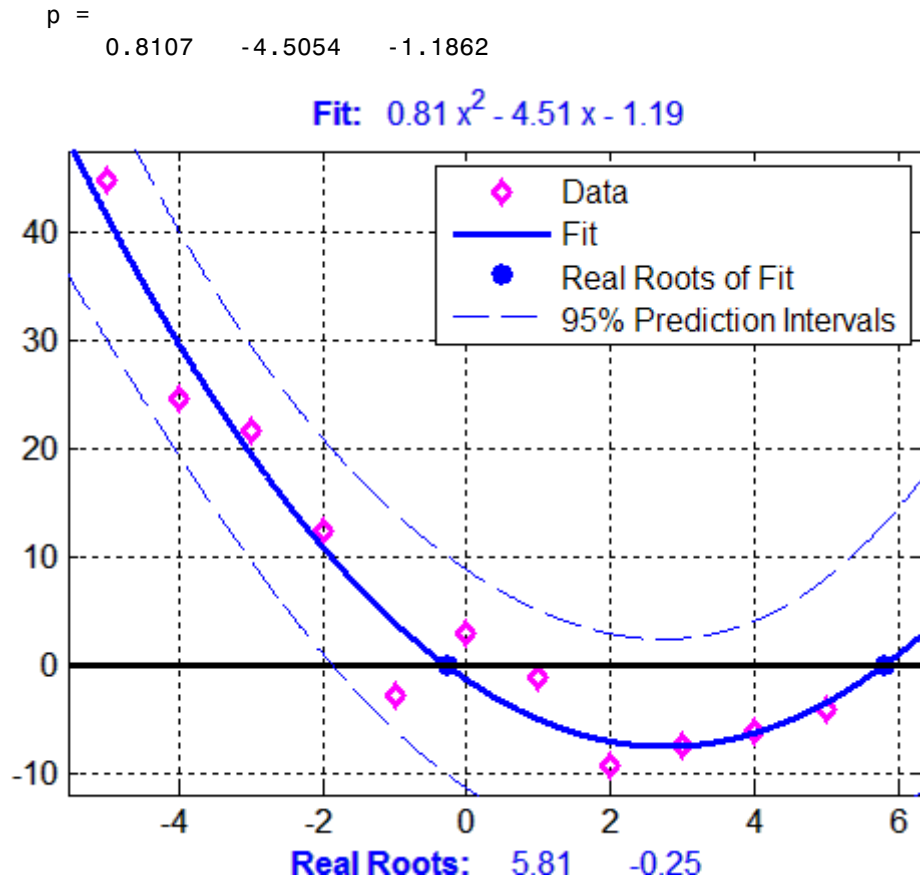
polydemo calls the documentation example function polystr to convert the coefficient vector p into a string for the polynomial expression displayed in the figure title.

Conditioning Polynomial Fits. If x and y data are on very different scales, polynomial fits may be badly conditioned, in the sense that coefficient estimates are very sensitive to random errors in the data. For example, using polyfit to estimate coefficients of a cubic fit to the U.S. census data in census.mat produces the following warning:

```
load census
x = cdate;
```

```
y = pop;
p = polyfit(x,y,3);
Warning: Polynomial is badly conditioned.
         Add points with distinct X values,
         reduce the degree of the polynomial,
         or try centering and scaling as
         described in HELP POLYFIT.
```

The following implements the suggested centering and scaling, and demonstrates the robustness of polynomial fits under these transformations:

```
plot(x,y,'ro') % Plot data
hold on

z = zscore(x); % Compute z-scores of x data
zfit = linspace(z(1),z(end),100);
pz = polyfit(z,y,3); % Compute conditioned fit
yfit = polyval(pz,zfit);

xfit = linspace(x(1),x(end),100);
plot(xfit,yfit,'b-') % Plot conditioned fit vs. x data
grid on
```
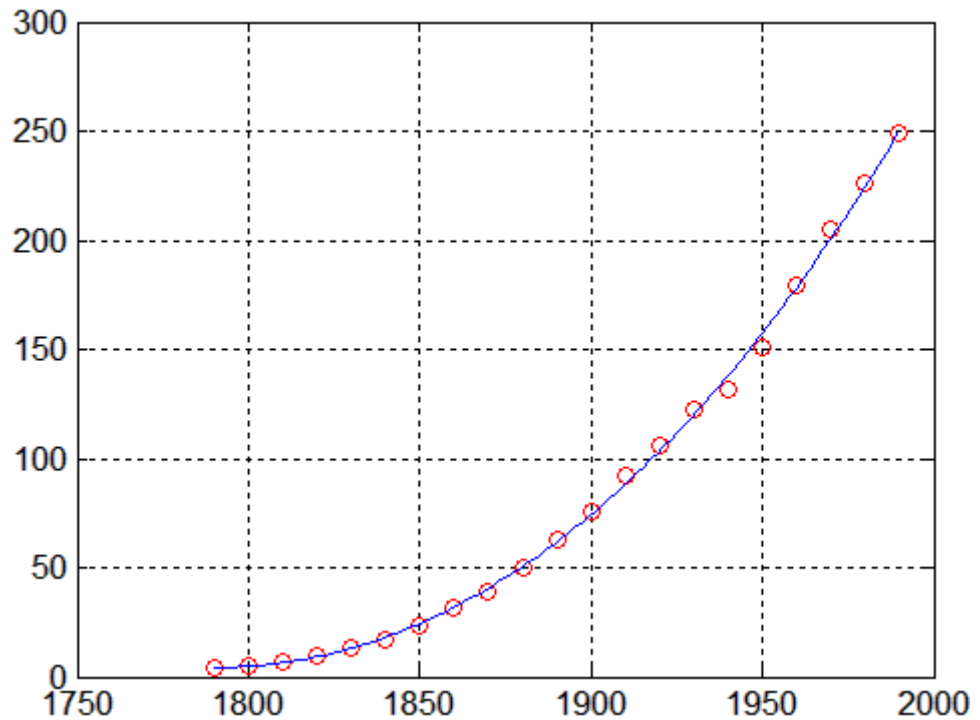
Interactive Polynomial Regression

The functions `polyfit`, `polyval`, and `polyconf` are interactively applied to data using two graphical interfaces for polynomial fitting:

- "The Basic Fitting Tool" on page 8-43
- "The Polynomial Fitting Tool" on page 8-44

The Basic Fitting Tool. The Basic Fitting Tool is a MATLAB interface, discussed in "Interactive Fitting" in the MATLAB documentation. The tool allows you to:

- Fit interpolants and polynomials of degree ≤ 10
- Plot residuals and compute their norm
- Interpolate or extrapolate values from the fit

• Save results to the MATLAB workspace

The Polynomial Fitting Tool. The Statistics Toolbox function `polytool` opens the Polynomial Fitting Tool. For example, the following opens the interface using simulated data with a quadratic trend and displays a fitted polynomial with 95% prediction intervals for new observations:

```
x = -5:5;
y = x.^2 - 5*x - 3 + 5*randn(size(x));
polytool(x,y,2,0.05)
```
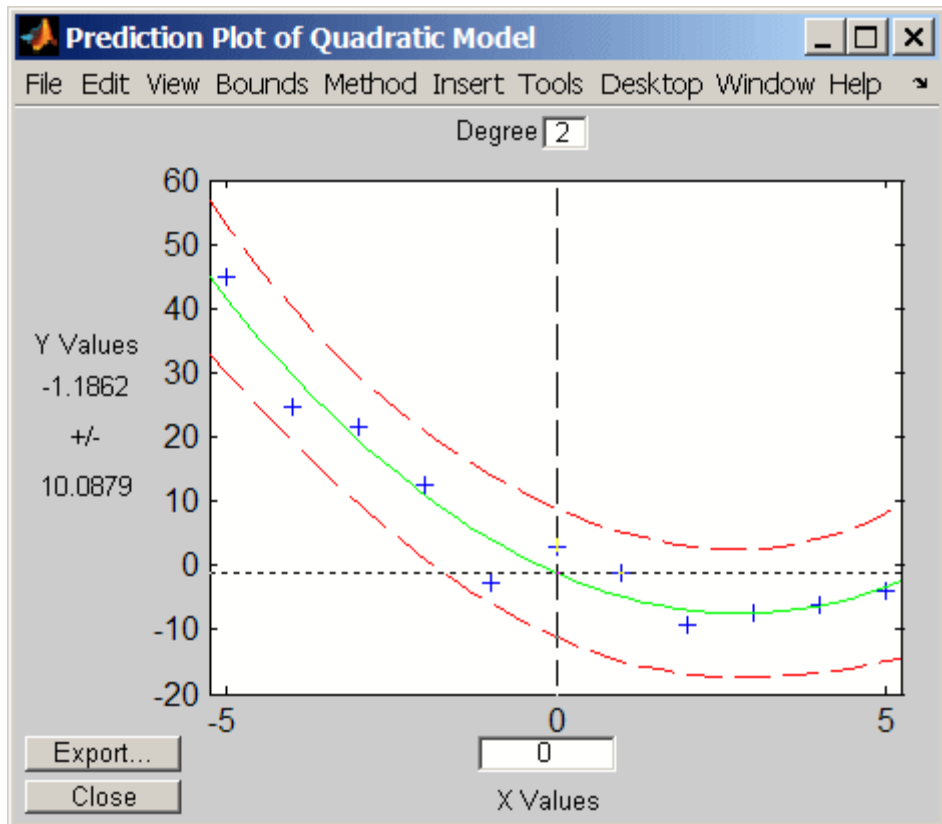

The tool allows you to:

- Interactively change the degree of the fit. Change the value in the **Degree** text box at the top of the figure.

- Evaluate the fit and the bounds using a movable crosshair. Click, hold, and drag the crosshair to change its position.

- Export estimated coefficients, predicted values, prediction intervals, and residuals to the MATLAB workspace. Click **Export** to a open a dialog box with choices for exporting the data.

Options for the displayed bounds and the fitting method are available through menu options at the top of the figure:

- The **Bounds** menu lets you choose between bounds on new observations (the default) and bounds on estimated values. It also lets you choose between nonsimultaneous (the default) and simultaneous bounds. See `polyconf` for a description of these options.

- The **Method** menu lets you choose between ordinary least-squares regression and robust regression, as described in "Robust Regression" on page 8-14.

Response Surface Models

- "Introduction" on page 8-45
- "Programmatic Response Surface Methodology" on page 8-46
- "Interactive Response Surface Methodology" on page 8-51

Introduction

Polynomial models are generalized to any number of predictor variables x_i ($i = 1, ..., N$) as follows:

$$y(x) = a_0 + \sum_{i=0}^{N} a_i x_i + \sum_{i<j}^{N} a_{ij} x_i x_j + \sum_{i=0}^{N} a_{ii} x_i^2 + \dots$$

The model includes, from left to right, an intercept, linear terms, quadratic interaction terms, and squared terms. Higher order terms would follow, as necessary.

Response surface models are multivariate polynomial models. They typically arise in the design of experiments (see Chapter 13, "Design of Experiments"), where they are used to determine a set of design variables that optimize a response. Linear terms alone produce models with response surfaces that are hyperplanes. The addition of interaction terms allows for warping of the hyperplane. Squared terms produce the simplest models in which the response surface has a maximum or minimum, and so an optimal response.

Response surface methodology (RSM) is the process of adjusting predictor variables to move the response in a desired direction and, iteratively, to an optimum. The method generally involves a combination of both computation and visualization. The use of quadratic response surface models makes the method much simpler than standard nonlinear techniques for determining optimal designs.

Programmatic Response Surface Methodology

The file `reaction.mat` contains simulated data on the rate of a chemical reaction:

```
load reaction
```

The variables include:

- `rate` — A 13-by-1 vector of observed reaction rates

- `reactants` — A 13-by-3 matrix of reactant concentrations

- `xn` — The names of the three reactants

- `yn` — The name of the response

In "Nonlinear Regression" on page 8-58, the nonlinear Hougen-Watson model is fit to the data using `nlinfit`. However, there may be no theoretical basis for choosing a particular model to fit the data. A quadratic response surface model provides a simple way to determine combinations of reactants that lead to high reaction rates.

As described in "Multiple Linear Regression" on page 8-8, the `regress` and `regstats` functions fit linear models—including response surface models—to data using a design matrix of model terms evaluated at predictor data. The

`x2fx` function converts predictor data to design matrices for quadratic models. The `regstats` function calls `x2fx` when instructed to do so.

For example, the following fits a quadratic response surface model to the data in `reaction.mat`:

```
stats = regstats(rate,reactants,'quadratic','beta');
b = stats.beta; % Model coefficients
```

The 10-by-1 vector `b` contains, in order, a constant term and then the coefficients for the model terms x_1, x_2, x_3, x_1x_2, x_1x_3, x_2x_3, x_1^2, x_2^2, and x_3^2, where x_1, x_2, and x_3 are the three columns of `reactants`. The order of coefficients for quadratic models is described in the reference page for `x2fx`.

Since the model involves only three predictors, it is possible to visualize the entire response surface using a color dimension for the reaction rate:

```
x1 = reactants(:,1);
x2 = reactants(:,2);
x3 = reactants(:,3);

xx1 = linspace(min(x1),max(x1),25);
xx2 = linspace(min(x2),max(x2),25);
xx3 = linspace(min(x3),max(x3),25);

[X1,X2,X3] = meshgrid(xx1,xx2,xx3);

RATE = b(1) + b(2)*X1 + b(3)*X2 + b(4)*X3 + ...
       b(5)*X1.*X2 + b(6)*X1.*X3 + b(7)*X2.*X3 + ...
       b(8)*X1.^2 + b(9)*X2.^2 + b(10)*X3.^2;

hmodel = scatter3(X1(:),X2(:),X3(:),5,RATE(:),'filled');
hold on
hdata = scatter3(x1,x2,x3,'ko','filled');
axis tight
xlabel(xn(1,:))
ylabel(xn(2,:))
zlabel(xn(3,:))
hbar = colorbar;
ylabel(hbar,yn);
title('{\bf Quadratic Response Surface Model}')
```

```
legend(hdata,'Data','Location','NE')
```


Quadratic Response Surface Model
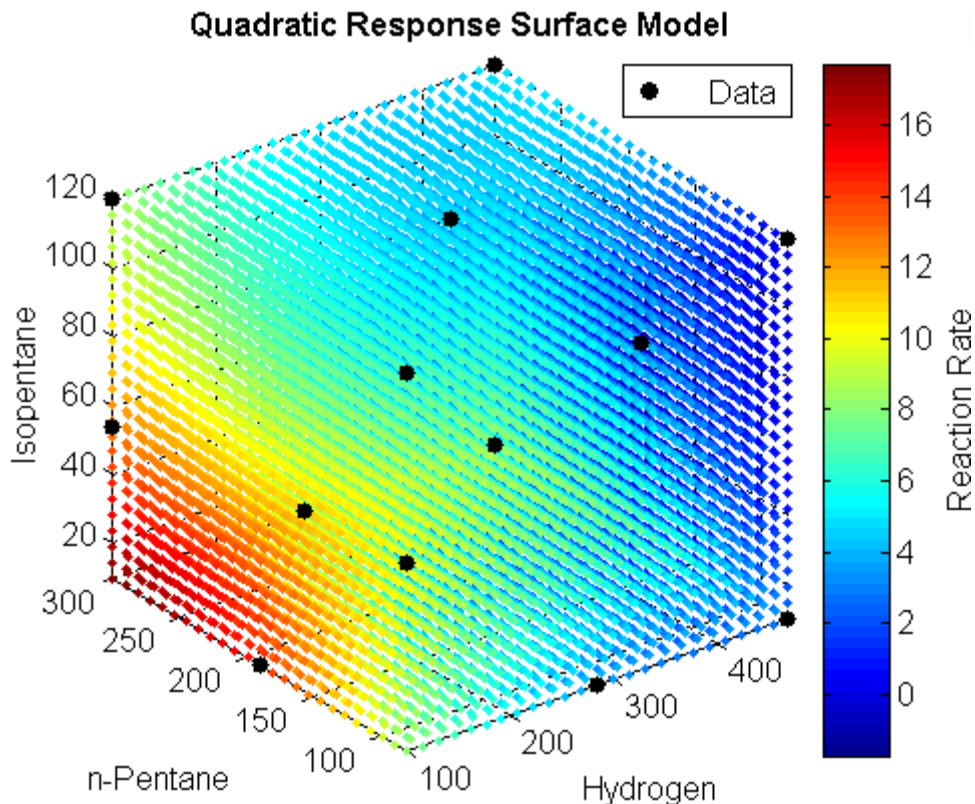
The plot show a general increase in model response, within the space of the observed data, as the concentration of *n*-pentane increases and the concentrations of hydrogen and isopentane decrease.

Before trying to determine optimal values of the predictors, perhaps by collecting more data in the direction of increased reaction rate indicated by the plot, it is helpful to evaluate the geometry of the response surface. If $x = (x_1, x_2, x_3)^T$ is the vector of predictors, and H is the matrix such that $x^T H x$ gives the quadratic terms of the model, the model has a unique optimum if and only if H is positive definite. For the data in this example, the model does not have a unique optimum:

```
H = [b(8),b(5)/2,b(6)/2; ...
     b(5)/2,b(9),b(7)/2; ...
     b(6)/2,b(7)/2,b(10)];
lambda = eig(H)
lambda =
  1.0e-003 *
   -0.1303
    0.0412
    0.4292
```

The negative eigenvalue shows a lack of positive definiteness. The saddle in the model is visible if the range of the predictors in the plot (xx1, xx2, and xx3) is expanded:

Quadratic Response Surface Model

8-49
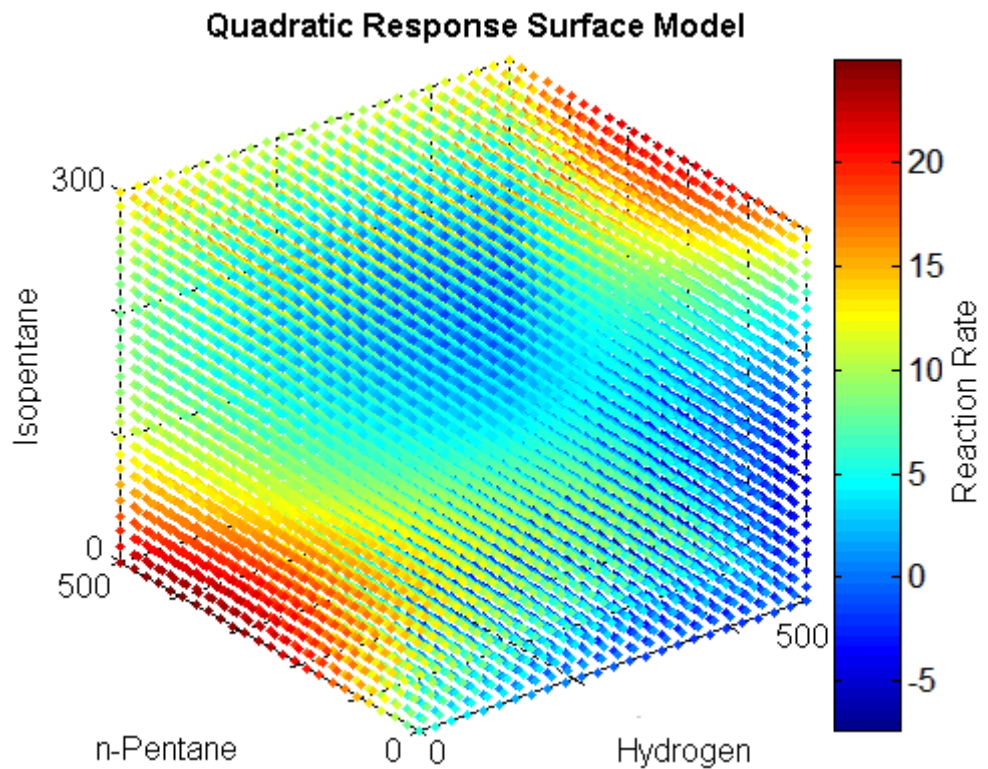
When the number of predictors makes it impossible to visualize the entire response surface, 3-, 2-, and 1-dimensional slices provide local views. The MATLAB function slice displays 2-dimensional contours of the data at fixed values of the predictors:

```
delete(hmodel)
X2slice = 200; % Fix n-Pentane concentration
slice(X1,X2,X3,RATE,[],X2slice,[])
```


Quadratic Response Surface Model
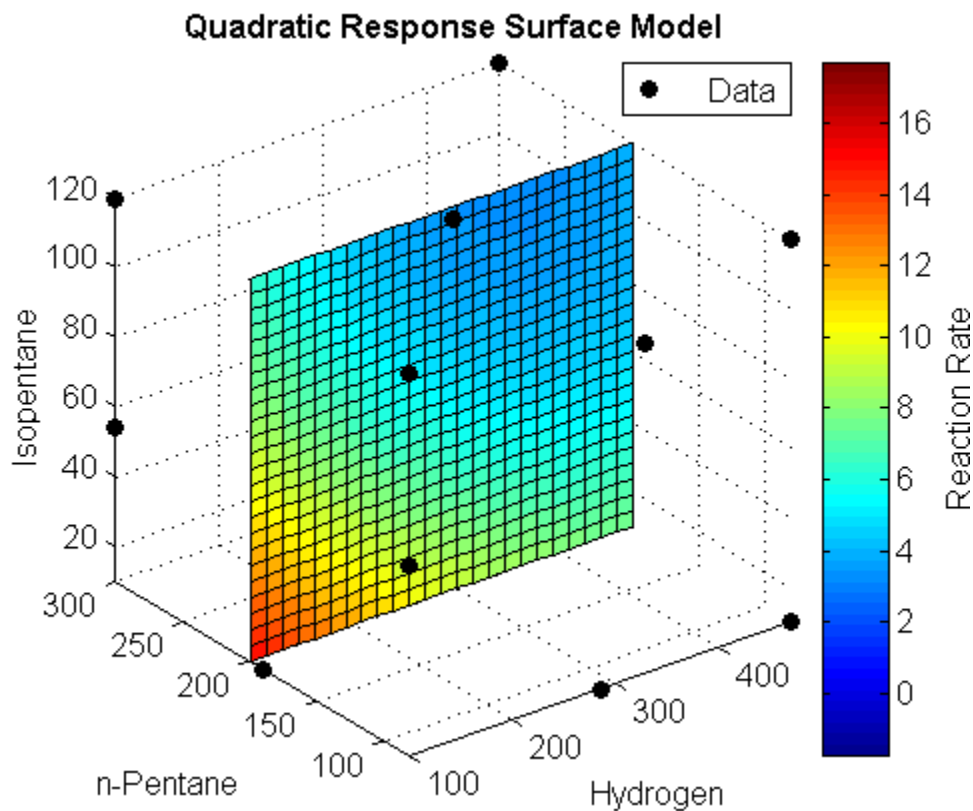
One-dimensional contours are displayed by the Response Surface Tool, rstool, described in the next section.

Interactive Response Surface Methodology

The Statistics Toolbox function `rstool` opens a GUI for interactively investigating simultaneous one-dimensional contours of multidimensional response surface models. For example, the following opens the interface with a quadratic response surface fit to the data in `reaction.mat`:

```
load reaction
alpha = 0.01; % Significance level
rstool(reactants,rate,'quadratic',alpha,xn,yn)
```
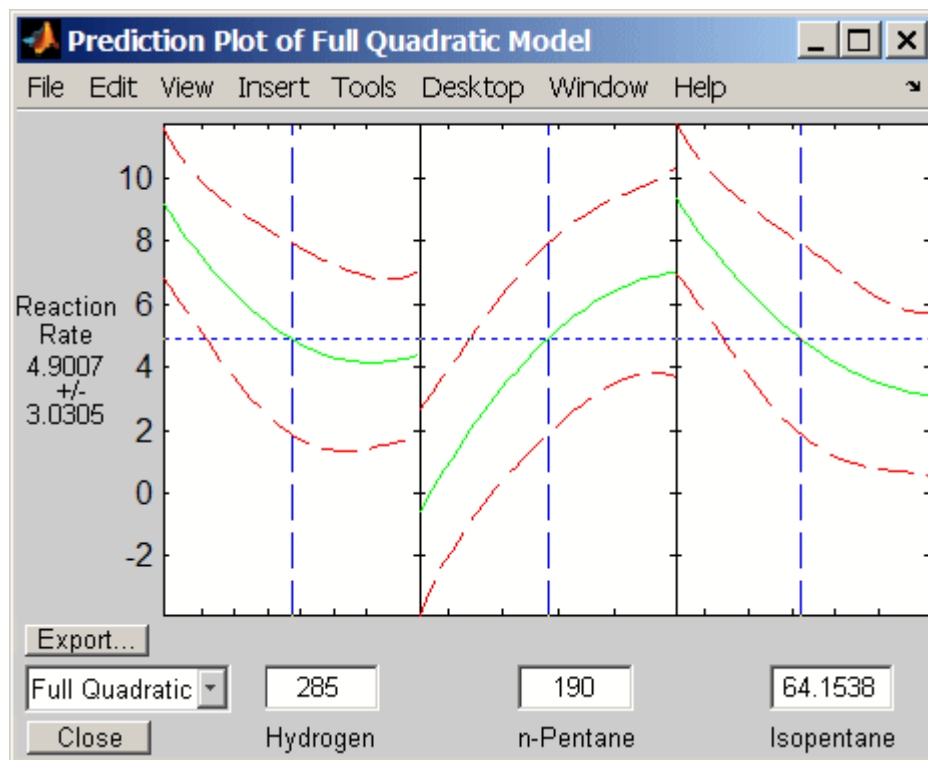

A sequence of plots is displayed, each showing a contour of the response surface against a single predictor, with all other predictors held fixed. Confidence intervals for new observations are shown as dashed red curves above and below the response. Predictor values are displayed in the text boxes on the horizontal axis and are marked by vertical dashed blue lines

in the plots. Predictor values are changed by editing the text boxes or by dragging the dashed blue lines. When you change the value of a predictor, all plots update to show the new point in predictor space.

Note The Statistics Toolbox demonstration function `rsmdemo` generates simulated data for experimental settings specified by either the user or by a *D*-optimal design generated by `cordexch`. It uses the `rstool` interface to visualize response surface models fit to the data, and it uses the `nlintool` interface to visualize a nonlinear model fit to the data.

Generalized Linear Models

Introduction

Linear regression models describe a linear relationship between a response and one or more predictive terms. Many times, however, a nonlinear relationship exists. "Nonlinear Regression" on page 8-58 describes general nonlinear models. A special class of nonlinear models, known as *generalized linear models*, makes use of linear methods.

Recall that linear models have the following characteristics:

- At each set of values for the predictors, the response has a normal distribution with mean μ.

- A coefficient vector b defines a linear combination Xb of the predictors X.

- The model is $\mu = Xb$.

In generalized linear models, these characteristics are generalized as follows:

- At each set of values for the predictors, 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 Xb of the predictors X.

• A *link function f* defines the model as $f(\mu) = Xb$.

Example: Generalized Linear Models

The following data are derived from `carbig.mat`, which contains measurements of large cars of various weights. Each weight in `w` has a corresponding number of cars in `total` and a corresponding number of poor-mileage cars in `poor`:

```
w = [2100 2300 2500 2700 2900 3100 ...
     3300 3500 3700 3900 4100 4300]';
total = [48 42 31 34 31 21 23 23 21 16 17 21]';
poor = [1 2 0 3 8 8 14 17 19 15 17 21]';
```

A plot shows that the proportion of poor-mileage cars follows an S-shaped sigmoid:

```
plot(w,poor./total,'x','LineWidth',2)
grid on
xlabel('Weight')
ylabel('Proportion of Poor-Mileage Cars')
```
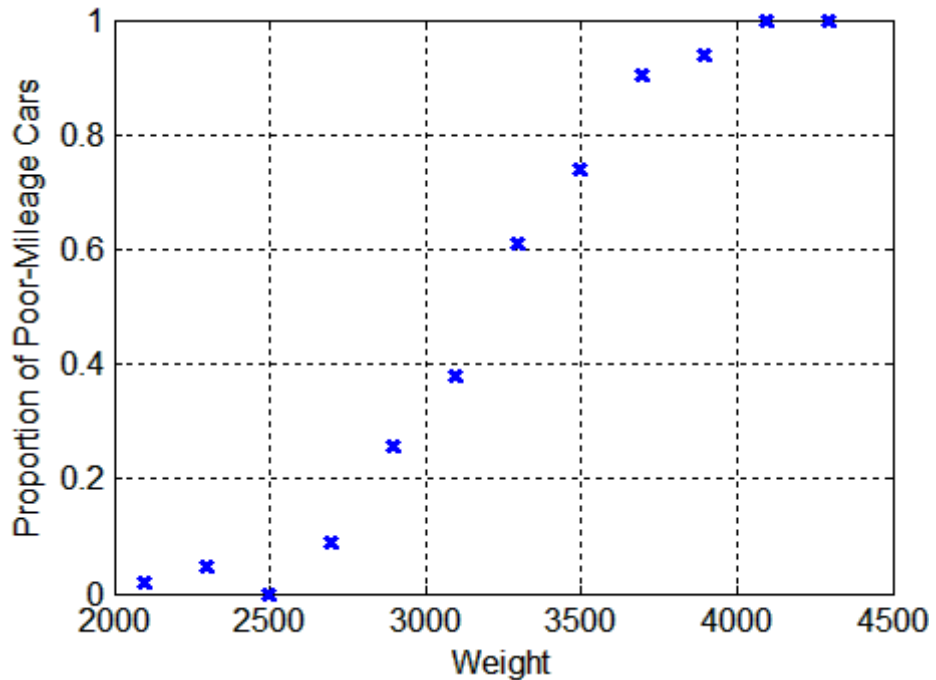
The *logistic model* is useful for proportion data. It defines the relationship between the proportion p and the weight w by:

$$\log[p/(1-p)] = b_1 + b_2 w$$

Some of the proportions in the data are `0` and `1`, making the left-hand side of this equation undefined. To keep the proportions within range, add relatively small perturbations to the `poor` and `total` values. A semi-log plot then shows a nearly linear relationship, as predicted by the model:

```
p_adjusted = (poor+.5)./(total+1);
semilogy(w,p_adjusted./(1-p_adjusted),'x','LineWidth',2)
grid on
xlabel('Weight')
ylabel('Adjusted  p / (1 - p)')
```
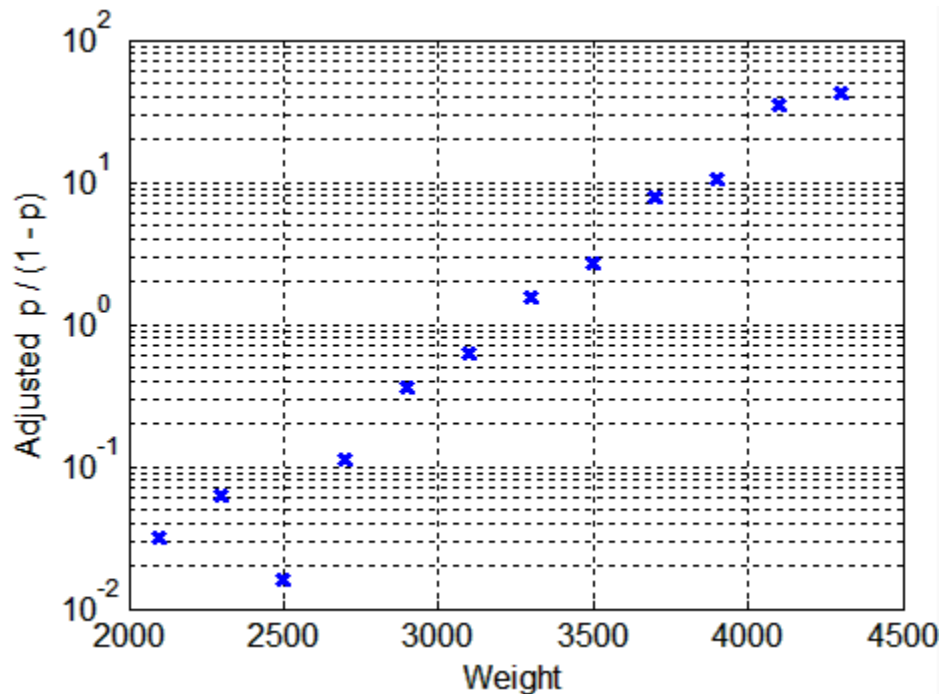
It is reasonable to assume that the values of poor follow binomial distributions, with the number of trials given by total and the percentage of successes depending on w. This distribution can be accounted for in the context of a logistic model by using a generalized linear model with link function $\log(\mu/(1-\mu)) = Xb$.

Use the glmfit function to carry out the associated regression:

```
b = glmfit(w,[poor total],'binomial','link','logit')
b =
  -13.3801
    0.0042
```

To use the coefficients in b to compute fitted proportions, invert the logistic relationship:

$$p = 1/(1 + \exp(-b_1 - b_2 w))$$

Use the `glmval` function to compute the fitted values:

```
x = 2100:100:4500;
y = glmval(b,x,'logit');

plot(w,poor./total,'x','LineWidth',2)
hold on
plot(x,y,'r-','LineWidth',2)
grid on
xlabel('Weight')
ylabel('Proportion of Poor-Mileage Cars')
```
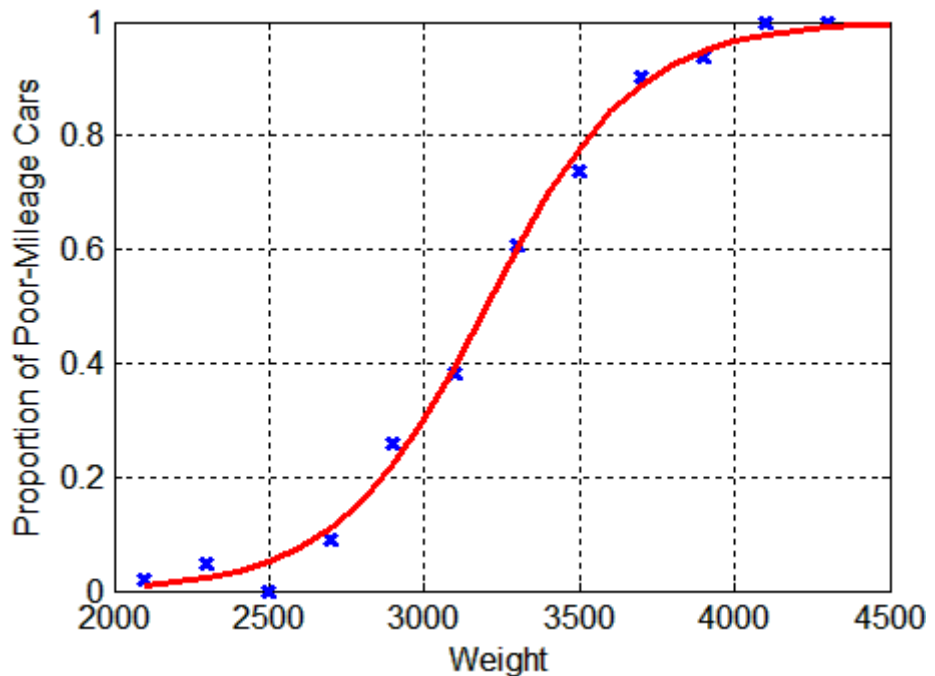

The previous is an example of *logistic regression*. For an example of a kind of *stepwise logistic regression*, analogous to stepwise regression for linear models, see "Sequential Feature Selection" on page 9-15.

Multivariate Regression

Whether or not the predictor x is a vector of predictor variables, *multivariate regression* refers to the case where the response $y = (y_1, ..., y_M)$ is a vector of M response variables.

The Statistics Toolbox functions `mvregress` and `mvregresslike` are used for multivariate regression analysis.

Nonlinear Regression

In this section...
"Nonlinear Regression Models" on page 8-58
"Parametric Models" on page 8-59
"Mixed-Effects Models" on page 8-64
"Regression Trees" on page 8-84

Nonlinear Regression Models

The models described in "Linear Regression Models" on page 8-3 are often called *empirical models*, because they are based solely on observed data. Model parameters typically have no relationship to any mechanism producing the data. To increase the accuracy of a linear model within the range of observations, the number of terms is simply increased.

Nonlinear models, on the other hand, typically involve parameters with specific physical interpretations. While they require *a priori* assumptions about the data-producing process, they are often more parsimonious than linear models, and more accurate outside the range of observed data.

Parametric nonlinear models represent the relationship between a continuous response variable and one or more predictor variables (either continuous or categorical) in the form $y = f(X, \beta) + \varepsilon$, 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.

Nonparametric models do not attempt to characterize the relationship between predictors and response with model parameters. Descriptions are often graphical, as in the case of "Regression Trees" on page 8-84.

Parametric Models

- "A Parametric Nonlinear Model" on page 8-59
- "Confidence Intervals for Parameter Estimates" on page 8-61
- "Confidence Intervals for Predicted Responses" on page 8-61
- "Interactive Nonlinear Parametric Regression" on page 8-62

A Parametric Nonlinear Model

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

$$rate = \frac{\beta_1 x_2 - x_3 / \beta_5}{1 + \beta_2 x_1 + \beta_3 x_2 + \beta_4 x_3}$$

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 reaction data:

```
load reaction
```

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.

The function `nlinfit` has robust options, similar to those for `robustfit`, for fitting nonlinear models to data with outliers.

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     8.4179     0.2805
     3.7900     3.9542     0.2474
     4.8200     4.9109     0.1766
     0.0200    -0.0110     0.1875
     2.7500     2.6358     0.1578
    14.3900    14.3402     0.4236
     2.5400     2.5662     0.2425
```

```
 4.3500     4.0385     0.1638
13.0000    13.0292     0.3426
 8.5000     8.3904     0.3281
 0.0500    -0.0216     0.3699
11.3200    11.4701     0.3237
 3.1300     3.4326     0.1749
```

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 Parametric 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

```
load reaction
nlintool(reactants,rate,@hougen,beta,0.01,xn,yn)
```
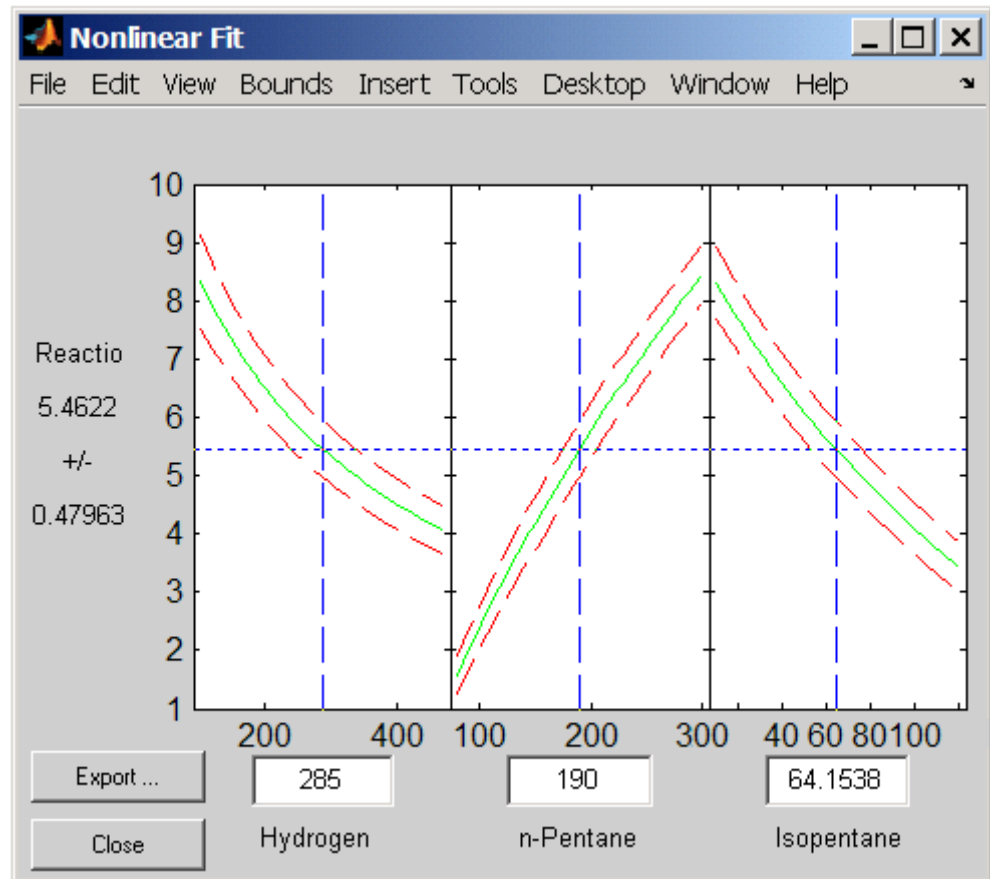
You see three plots. The response variable for all plots is the reaction rate, plotted in green. The red lines show confidence intervals on predicted responses. The first plot shows hydrogen as the predictor, the second shows *n*-pentane, and the third shows isopentane.

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 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.

Note The Statistics Toolbox demonstration function rsmdemo generates simulated data for experimental settings specified by either the user or by a *D*-optimal design generated by cordexch. It uses the rstool interface to visualize response surface models fit to the data, and it uses the nlintool interface to visualize a nonlinear model fit to the data.

Mixed-Effects Models

- "Introduction" on page 8-64
- "Mixed-Effects Model Hierarchy" on page 8-65
- "Specifying Mixed-Effects Models" on page 8-67
- "Example: Mixed-Effects Models" on page 8-69

Introduction

In statistics, an *effect* is anything that influences the value of a response variable at a particular setting of the predictor variables. Effects are translated into model parameters. In linear models, effects become coefficients, representing the proportional contributions of model terms. In nonlinear models, effects often have specific physical interpretations, and appear in more general nonlinear combinations.

Fixed effects represent population parameters, assumed to be the same each time data is collected. Estimating fixed effects is the traditional domain of regression modeling. *Random effects*, by comparison, are sample-dependent random variables. In modeling, random effects act like additional error terms, and their distributions and covariances must be specified.

For example, consider a model of the elimination of a drug from the bloodstream. The model uses time t as a predictor and the concentration of the drug C as the response. The nonlinear model term $C_0 e^{-rt}$ combines parameters C_0 and r, representing, respectively, an initial concentration and an elimination rate. If data is collected across multiple individuals, it

is reasonable to assume that the elimination rate is a random variable r_i depending on individual i, varying around a population mean $\bar{r}$. The term $C_0 e^{-rt}$ becomes

$$C_0 e^{-[\bar{r} + (r_i - \bar{r})]t} = C_0 e^{-(\beta + b_i)t},$$

where $\beta = \bar{r}$ is a fixed effect and $b_i = r_i - \bar{r}$ is a random effect.

Random effects are useful when data falls into natural groups. In the drug elimination model, the groups are simply the individuals under study. More sophisticated models might group data by an individual's age, weight, diet, etc. Although the groups are not the focus of the study, adding random effects to a model extends the reliability of inferences beyond the specific sample of individuals.

Mixed-effects models account for both fixed and random effects. As with all regression models, their purpose is to describe a response variable as a function of the predictor variables. Mixed-effects models, however, recognize correlations within sample subgroups. In this way, they provide a compromise between ignoring data groups entirely and fitting each group with a separate model.

Mixed-Effects Model Hierarchy

Suppose data for a nonlinear regression model falls into one of m distinct groups $i = 1, ..., m$. To account for the groups in a model, write response j in group i as:

$$y_{ij} = f(\varphi, x_{ij}) + \varepsilon_{ij}$$

y_{ij} is the response, x_{ij} is a vector of predictors, φ is a vector of model parameters, and ε_{ij} is the measurement or process error. The index j ranges from 1 to n_i, where n_i is the number of observations in group i. The function f specifies the form of the model. Often, x_{ij} is simply an observation time t_{ij}. The errors are usually assumed to be independent and identically, normally distributed, with constant variance.

Estimates of the parameters in φ describe the population, assuming those estimates are the same for all groups. If, however, the estimates vary by group, the model becomes

$$y_{ij} = f(\varphi_i, x_{ij}) + \varepsilon_{ij}$$

In a mixed-effects model, φ_i may be a combination of a fixed and a random effect:

$$\varphi_i = \beta + b_i$$

The random effects b_i are usually described as multivariate normally distributed, with mean zero and covariance Ψ. Estimating the fixed effects β and the covariance of the random effects Ψ provides a description of the population that does not assume the parameters φ_i are the same across groups. Estimating the random effects b_i also gives a description of specific groups within the data.

Model parameters do not have to be identified with individual effects. In general, *design matrices A and B* are used to identify parameters with linear combinations of fixed and random effects:

$$\varphi_i = A\beta + Bb_i$$

If the design matrices differ among groups, the model becomes

$$\varphi_i = A_i\beta + B_ib_i$$

If the design matrices also differ among observations, the model becomes

$$\varphi_{ij} = A_{ij}\beta + B_{ij}b_i$$
$$y_{ij} = f(\varphi_{ij}, x_{ij}) + \varepsilon_{ij}$$

Some of the group-specific predictors in x_{ij} may not change with observation j. Calling those v_i, the model becomes

$$y_{ij} = f(\varphi_{ij}, x_{ij}, v_i) + \varepsilon_{ij}$$

Specifying Mixed-Effects Models

Suppose data for a nonlinear regression model falls into one of m distinct groups $i = 1, ..., m$. (Specifically, suppose that the groups are not nested.) To specify a general nonlinear mixed-effects model for this data:

1 Define group-specific model parameters φ_i as linear combinations of fixed effects β and random effects b_i.

2 Define response values y_i as a nonlinear function f of the parameters and group-specific predictor variables X_i.

The model is:

$$\varphi_i = A_i\beta + B_i b_i$$
$$y_i = f(\varphi_i, X_i) + \varepsilon_i$$
$$b_i \sim N(0, \Psi)$$
$$\varepsilon_i \sim N(0, \sigma^2)$$

This formulation of the nonlinear mixed-effects model uses the following notation:

φ_i	A vector of group-specific model parameters
β	A vector of fixed effects, modeling population parameters
b_i	A vector of multivariate normally distributed group-specific random effects
A_i	A group-specific design matrix for combining fixed effects
B_i	A group-specific design matrix for combining random effects
X_i	A data matrix of group-specific predictor values
y_i	A data vector of group-specific response values
f	A general, real-valued function of φ_i and X_i
ε_i	A vector of group-specific errors, assumed to be independent, identically, normally distributed, and independent of b_i
Ψ	A covariance matrix for the random effects
σ^2	The error variance, assumed to be constant across observations

For example, consider a model of the elimination of a drug from the bloodstream. The model incorporates two overlapping phases:

- An initial phase p during which drug concentrations reach equilibrium with surrounding tissues

- A second phase q during which the drug is eliminated from the bloodstream

For data on multiple individuals i, the model is

$$y_{ij} = C_{pi}e^{-r_{pi}t_{ij}} + C_{qi}e^{-r_{qi}t_{ij}} + \varepsilon_{ij},$$

where y_{ij} is the observed concentration in individual i at time t_{ij}. The model allows for different sampling times and different numbers of observations for different individuals.

The elimination rates r_{pi} and r_{qi} must be positive to be physically meaningful. Enforce this by introducing the log rates $R_{pi} = \log(r_{pi})$ and $R_{qi} = \log(r_{qi})$ and reparametrizing the model:

$$y_{ij} = C_{pi}e^{-\exp(R_{pi})t_{ij}} + C_{qi}e^{-\exp(R_{qi})t_{ij}} + \varepsilon_{ij}$$

Choosing which parameters to model with random effects is an important consideration when building a mixed-effects model. One technique is to add random effects to all parameters, and use estimates of their variances to determine their significance in the model. An alternative is to fit the model separately to each group, without random effects, and look at the variation of the parameter estimates. If an estimate varies widely across groups, or if confidence intervals for each group have minimal overlap, the parameter is a good candidate for a random effect.

To introduce fixed effects β and random effects b_i for all model parameters, reexpress the model as follows:

$$
\begin{aligned}
y_{ij} &= [\bar{C}_p + (C_{pi} - \bar{C}_p)]e^{-\exp[\bar{R}_p + (R_{pi} - \bar{R}_p)]t_{ij}} + \\
&\quad [\bar{C}_q + (C_{qi} - \bar{C}_q)]e^{-\exp[\bar{R}_q + (R_{qi} - \bar{R}_q)]t_{ij}} + \varepsilon_{ij} \\
&= (\beta_1 + b_{1i})e^{-\exp(\beta_2 + b_{2i})t_{ij}} + \\
&\quad (\beta_3 + b_{3i})e^{-\exp(\beta_4 + b_{4i})t_{ij}} + \varepsilon_{ij}
\end{aligned}
$$

In the notation of the general model:

$$
\beta = \begin{pmatrix} \beta_1 \\ \vdots \\ \beta_4 \end{pmatrix}, b_i = \begin{pmatrix} b_{i1} \\ \vdots \\ b_{i4} \end{pmatrix}, y_i = \begin{pmatrix} y_{i1} \\ \vdots \\ y_{in_i} \end{pmatrix}, X_i = \begin{pmatrix} t_{i1} \\ \vdots \\ t_{in_i} \end{pmatrix},
$$

where n_i is the number of observations of individual i. In this case, the design matrices A_i and B_i are, at least initially, 4-by-4 identity matrices. Design matrices may be altered, as necessary, to introduce weighting of individual effects, or time dependency.

Fitting the model and estimating the covariance matrix Ψ often leads to further refinements. A relatively small estimate for the variance of a random effect suggests that it can be removed from the model. Likewise, relatively small estimates for covariances among certain random effects suggests that a full covariance matrix is unnecessary. Since random effects are unobserved, Ψ must be estimated indirectly. Specifying a diagonal or block-diagonal covariance pattern for Ψ can improve convergence and efficiency of the fitting algorithm.

The Statistics Toolbox function `nlmefit` fits the general nonlinear mixed-effects model to data, estimating the fixed and random effects. The function also estimates the covariance matrix Ψ for the random effects. Additional diagnostic outputs allow you to assess tradeoffs between the number of model parameters and the goodness of fit.
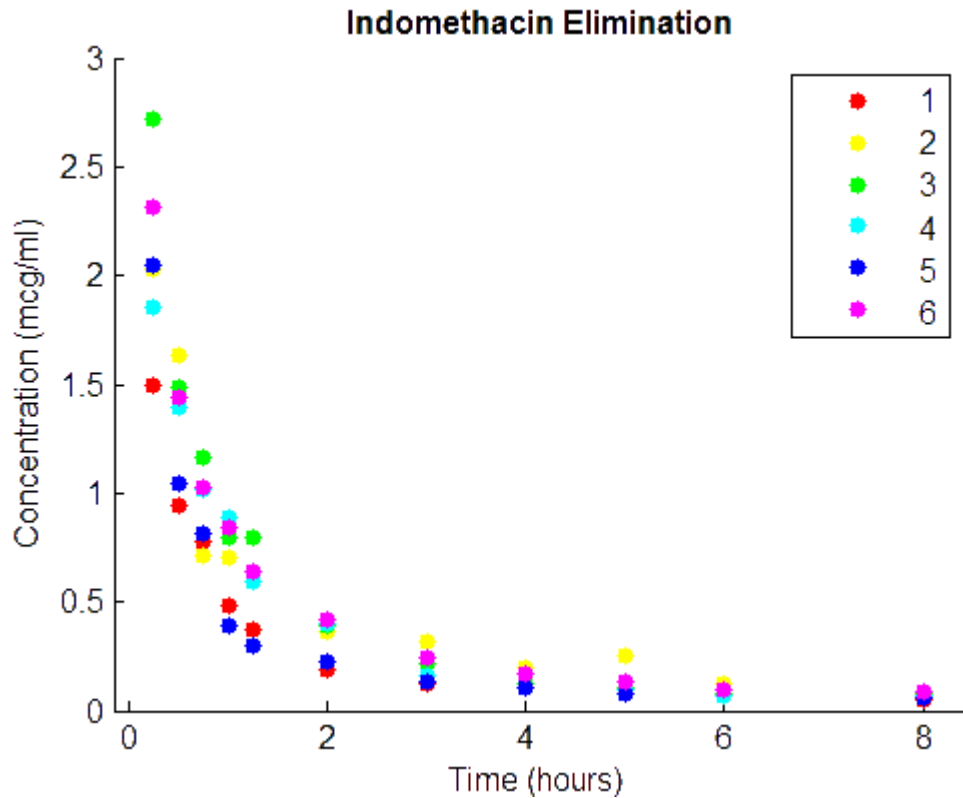
Example: Mixed-Effects Models

The data in `indomethacin.mat` record concentrations of the drug indomethacin in the bloodstream of six subjects over an eight-hour period:

```
load indomethacin

gscatter(time,concentration,subject)
xlabel('Time (hours)')
ylabel('Concentration (mcg/ml)')
title('{\bf Indomethacin Elimination}')
hold on
```


"Specifying Mixed-Effects Models" on page 8-67 discusses a useful model for this type of data. Construct the model via an anonymous function as follows:

```
model = @(phi,t)(phi(1)*exp(-exp(phi(2))*t) + ...
                 phi(3)*exp(-exp(phi(4))*t));
```

Use the `nlinfit` function to fit the model to all of the data, ignoring subject-specific effects:

```
phi0 = [1 1 1 1];
[phi,res] = nlinfit(time,concentration,model,phi0);

numObs = length(time);
numParams = 4;
df = numObs-numParams;
mse = (res'*res)/df
mse =
    0.0304

tplot = 0:0.01:8;
plot(tplot,model(phi,tplot),'k','LineWidth',2)
hold off
```
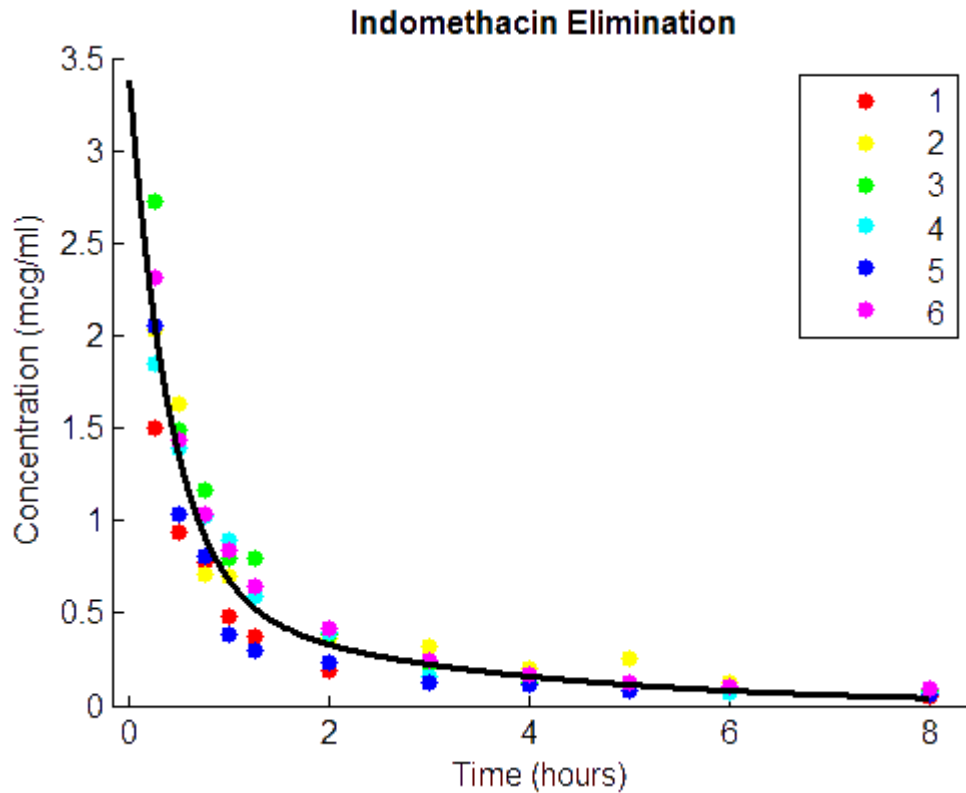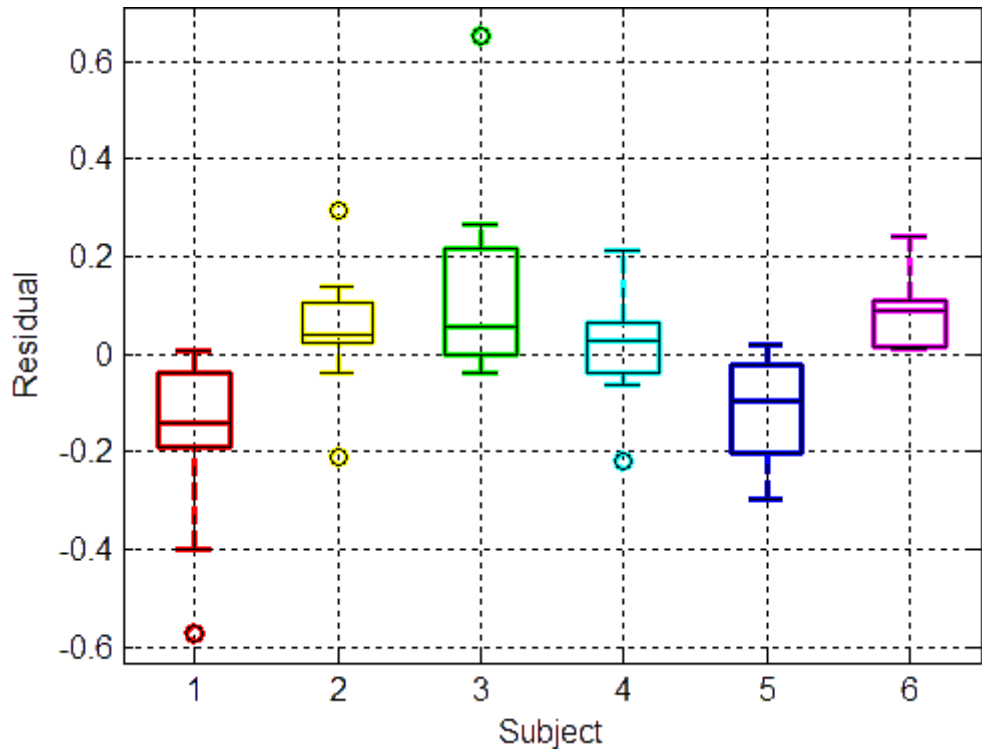
A box plot of residuals by subject shows that the boxes are mostly above or below zero, indicating that the model has failed to account for subject-specific effects:

```
colors = 'rygcbm';
h = boxplot(res,subject,'colors',colors,'symbol','o');
set(h(~isnan(h)),'LineWidth',2)
hold on
boxplot(res,subject,'colors','k','symbol','ko')
grid on
xlabel('Subject')
ylabel('Residual')
hold off
```

To account for subject-specific effects, fit the model separately to the data for each subject:

```
phi0 = [1 1 1 1];
PHI = zeros(4,6);
RES = zeros(11,6);
for I = 1:6
    tI = time(subject == I);
    cI = concentration(subject == I);
    [PHI(:,I),RES(:,I)] = nlinfit(tI,cI,model,phi0);
end

PHI
PHI =
    0.1915    0.4989    1.6757    0.2545    3.5661    0.9685
   -1.7878   -1.6354   -0.4122   -1.6026    1.0408   -0.8731
```

```
          2.0293      2.8277      5.4683      2.1981      0.2915      3.0023
          0.5794      0.8013      1.7498      0.2423     -1.5068      1.0882

numParams = 24;
df = numObs-numParams;
mse = (RES(:)'*RES(:))/df
mse =
    0.0057

gscatter(time,concentration,subject)
xlabel('Time (hours)')
ylabel('Concentration (mcg/ml)')
title('{\bf Indomethacin Elimination}')
hold on
for I = 1:6
    plot(tplot,model(PHI(:,I),tplot),'Color',colors(I))
end
axis([0 8 0 3.5])
hold off
```
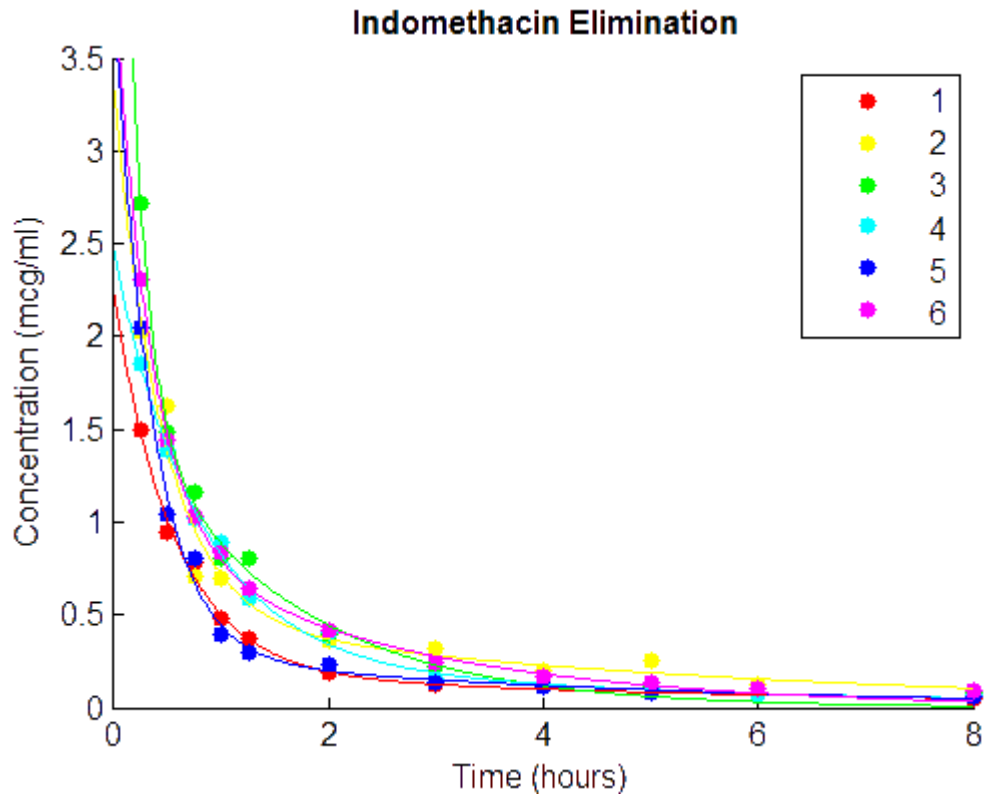
PHI gives estimates of the four model parameters for each of the six subjects. The estimates vary considerably, but taken as a 24-parameter model of the data, the mean-squared error of `0.0057` is a significant reduction from `0.0304` in the original four-parameter model.
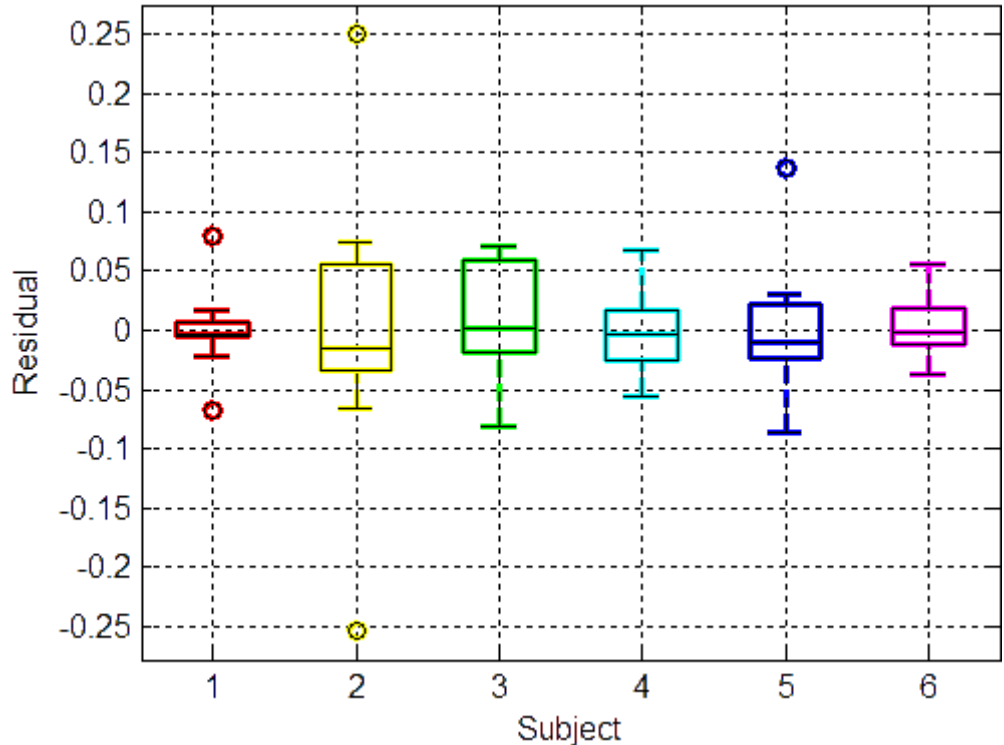
A box plot of residuals by subject shows that the larger model accounts for most of the subject-specific effects:

```
h = boxplot(RES,'colors',colors,'symbol','o');
set(h(~isnan(h)),'LineWidth',2)
hold on
boxplot(RES,'colors','k','symbol','ko')
grid on
xlabel('Subject')
```

```
ylabel('Residual')
hold off
```


The spread of the residuals (the vertical scale of the boxplot) is much smaller than in the previous boxplot, and the boxes are now mostly centered on zero.

While the 24-parameter model successfully accounts for variations due to the specific subjects in the study, it does not consider the subjects as representatives of a larger population. The sampling distribution from which the subjects are drawn is likely of more interest than the sample itself. The purpose of mixed-effects models is to account for subject-specific variations more broadly, as random effects varying around population means.

Use the `nlmefit` function to fit a mixed-effects model to the data.

The following anonymous function, `nlme_model`, adapts the four-parameter `model` used by `nlinfit` to the calling syntax of `nlmefit` by allowing separate parameters for each individual. By default, `nlmefit` assigns random effects to all the model parameters. Also by default, `nlmefit` assumes a diagonal covariance matrix (no covariance among the random effects) to avoid overparametrization and related convergence issues.

```
nlme_model = @(PHI,t)(PHI(:,1).*exp(-exp(PHI(:,2)).*t) + ...
                      PHI(:,3).*exp(-exp(PHI(:,4)).*t));

phi0 = [1 1 1 1];
[phi,PSI,stats] = nlmefit(time,concentration,subject, ...
                          [],nlme_model,phi0)
phi =
     0.4606
    -1.3459
     2.8277
     0.7729
PSI =
     0.0124        0        0        0
          0   0.0000        0        0
          0        0   0.3264        0
          0        0        0   0.0250
stats =
       logl: 54.5884
        mse: 0.0066
        aic: -91.1767
        bic: -71.4698
     sebeta: NaN
        dfe: 57
```

The mean-squared error of `0.0066` is comparable to the `0.0057` of the 24-parameter model without random effects, and significantly better than the `0.0304` of the four-parameter model without random effects.

The estimated covariance matrix `PSI` shows that the variance of the second random effect is essentially zero, suggesting that you can remove it to simplify the model. To do this, use the `REParamsSelect` parameter to specify the indices of the parameters to be modeled with random effects in `nlmefit`:

```
[phi,PSI,stats] = nlmefit(time,concentration,subject, ...
```

```
                                     [],nlme_model,phiO, ...
                                     'REParamsSelect',[1 3 4])
            phi =
                0.4606
               -1.3460
                2.8277
                0.7729
            PSI =
                0.0124          0          0
                     0     0.3270          0
                     0          0     0.0250
            stats =
                  logl: 54.5876
                   mse: 0.0066
                   aic: -93.1752
                   bic: -75.6580
                sebeta: NaN
                   dfe: 58
```

The log-likelihood `logl` is almost identical to what it was with random effects for all of the parameters, the Akaike information criterion `aic` is reduced from `-91.1767` to `-93.1752`, and the Bayesian information criterion `bic` is reduced from `-71.4698` to `-75.6580`. These measures support the decision to drop the second random effect.

Refitting the simplified model with a full covariance matrix allows for identification of correlations among the random effects. To do this, use the `CovPattern` parameter to specify the pattern of nonzero elements in the covariance matrix:

```
   [phi,PSI,stats] = nlmefit(time,concentration,subject, ...
                             [],nlme_model,phiO, ...
                             'REParamsSelect',[1 3 4], ...
                             'CovPattern',ones(3))
   phi =
       0.5613
      -1.1407
       2.8148
       0.8293
   PSI =
```

```
     0.0236    0.0500    0.0032
     0.0500    0.4768    0.1152
     0.0032    0.1152    0.0321
stats =
       logl: 58.4731
        mse: 0.0061
        aic: -94.9462
        bic: -70.8600
      sebeta: NaN
        dfe: 55
```

The estimated covariance matrix PSI shows that the random effects on the last two parameters have a relatively strong correlation, and both have a relatively weak correlation with the first random effect. This structure in the covariance matrix is more apparent if you convert PSI to a correlation matrix using corrcov:

```
RHO = corrcov(PSI)
RHO =
     1.0000    0.4707    0.1179
     0.4707    1.0000    0.9316
     0.1179    0.9316    1.0000

imagesc(RHO)
set(gca,'XTick',[1 2 3],'YTick',[1 2 3])
title('{\bf Random Effect Correlation}')
h = colorbar;
set(get(h,'YLabel'),'String','Correlation');
```

Random Effect Correlation
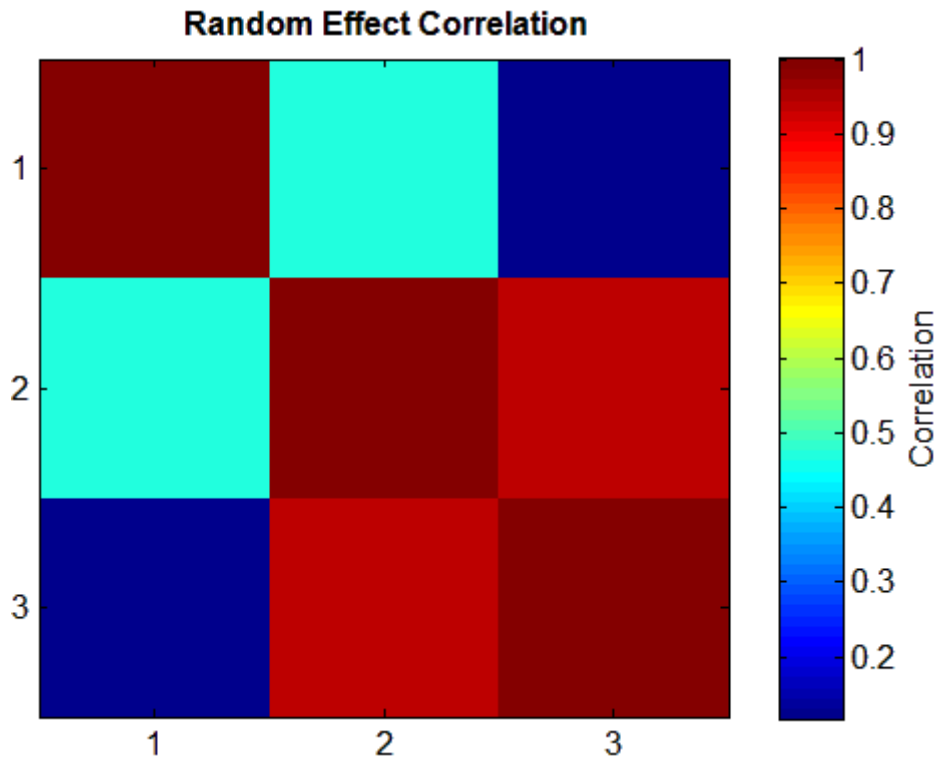
Incorporate this structure into the model by changing the specification of the covariance pattern to block-diagonal:

```
P = [1 0 0;0 1 1;0 1 1] % Covariance pattern
P =
     1     0     0
     0     1     1
     0     1     1

[phi,PSI,stats,b] = nlmefit(time,concentration,subject, ...
                            [],nlme_model,phi0, ...
                            'REParamsSelect',[1 3 4], ...
                            'CovPattern',P)
phi =
    0.5850
```

```
        -1.1087
         2.8056
         0.8476
PSI =
      0.0331            0            0
           0       0.4793       0.1069
           0       0.1069       0.0294
stats =
        logl: 57.4996
         mse: 0.0061
         aic: -96.9992
         bic: -77.2923
      sebeta: NaN
         dfe: 57
b =
     -0.2438       0.0723       0.2014       0.0592      -0.2181       0.1289
     -0.8500      -0.1237       0.9538      -0.7267       0.5895       0.1571
     -0.1591       0.0033       0.1568      -0.2144       0.1834       0.0300
```

The block-diagonal covariance structure reduces aic from -94.9462 to
-96.9992 and bic from -70.8600 to -77.2923 without significantly affecting
the log-likelihood. These measures support the covariance structure used in
the final model.

The output b gives predictions of the three random effects for each of the six
subjects. These are combined with the estimates of the fixed effects in phi
to produce the mixed-effects model.

The following commands plot the mixed-effects model for each of the six
subjects. For comparison, the model without random effects is also shown.

```
PHI = repmat(phi,1,6) + ...                       % Fixed effects
      [b(1,:);zeros(1,6);b(2,:);b(3,:)];          % Random effects

RES = zeros(11,6); % Residuals
colors = 'rygcbm';
for I = 1:6
    fitted_model = @(t)(PHI(1,I)*exp(-exp(PHI(2,I))*t) + ...
                        PHI(3,I)*exp(-exp(PHI(4,I))*t));
    tI = time(subject == I);
```

```
        cI = concentration(subject == I);
        RES(:,I) = cI - fitted_model(tI);

        subplot(2,3,I)
        scatter(tI,cI,20,colors(I),'filled')
        hold on
        plot(tplot,fitted_model(tplot),'Color',colors(I))
        plot(tplot,model(phi,tplot),'k')
        axis([0 8 0 3.5])
        xlabel('Time (hours)')
        ylabel('Concentration (mcg/ml)')
        legend(num2str(I),'Subject','Fixed')
end
```
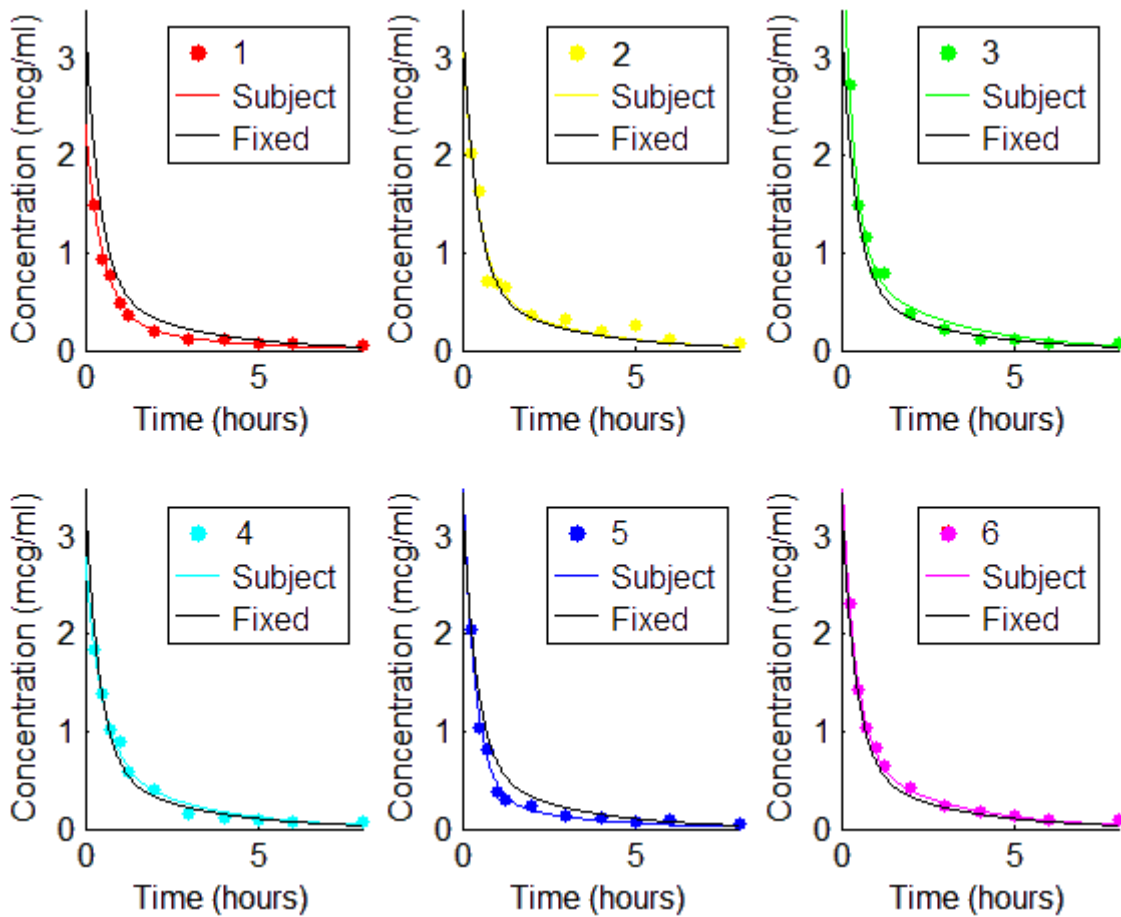
If obvious outliers in the data (visible in previous box plots) are ignored, a normal probability plot of the residuals shows reasonable agreement with model assumptions on the errors:

```
normplot(RES(:))
```
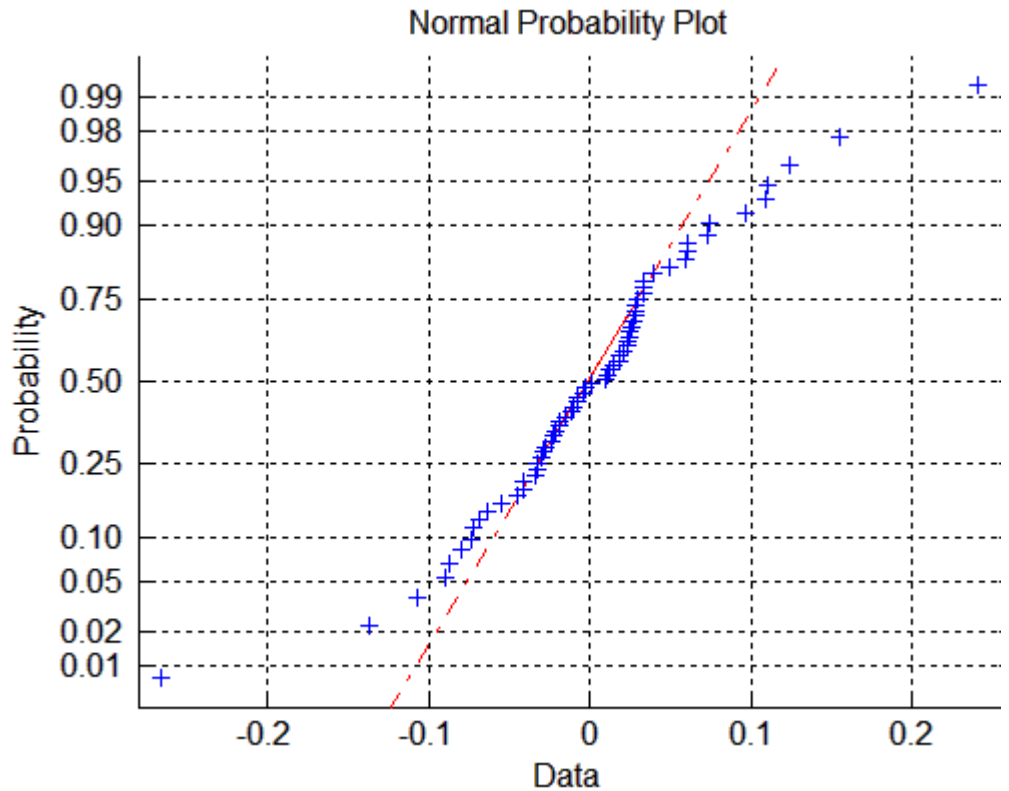
Regression Trees

- "Introduction" on page 8-84

- "Example: Regression Trees" on page 8-85

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 8-59. 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 are categorical, classification trees are a natural modification.

Note This section demonstrates methods for objects of the `@classregtree` class. These methods supersede the functions `treefit`, `treedisp`, `treeval`, `treeprune`, and `treetest`, which are maintained in Statistics Toolbox software only for backwards compatibility.

Algorithm Reference. The algorithms used by Statistics Toolbox classification and regression tree functions are based on those in Breiman, L., et al., *Classification and Regression Trees*, Chapman & Hall, Boca Raton, 1993.

Example: 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 of the `classregtree` class 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.
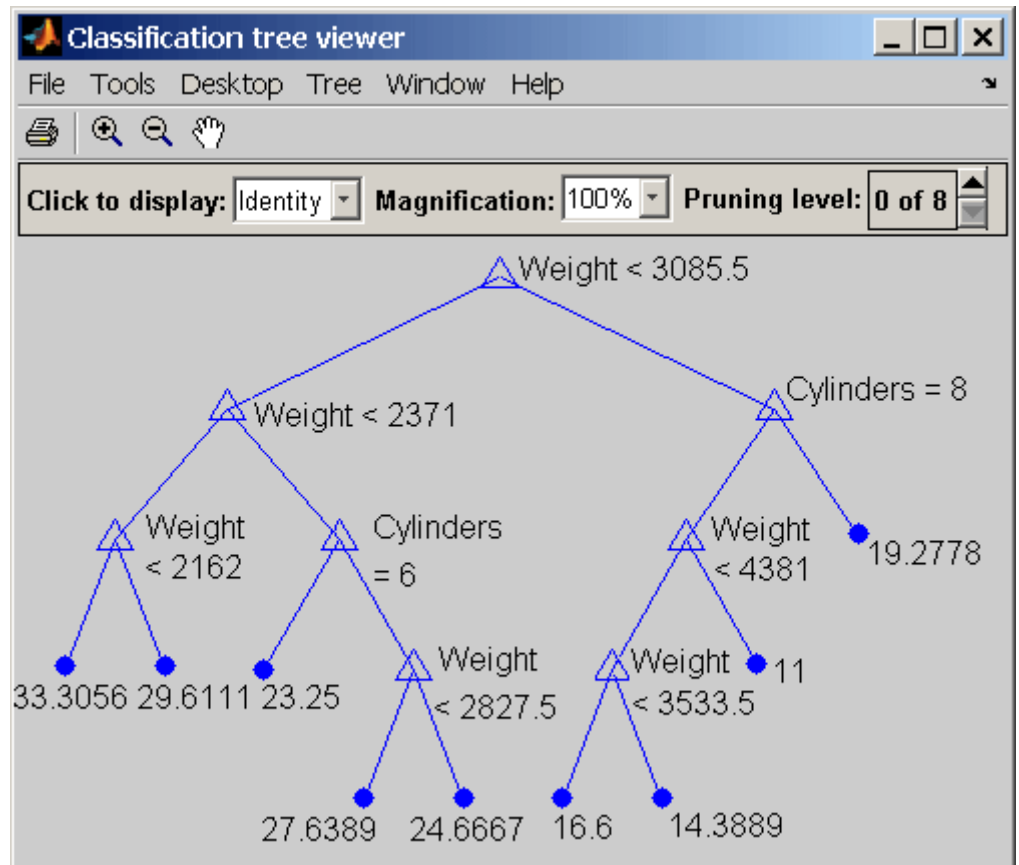
Use the `type` method of the `classregtree` class to show the type of the tree:

```
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 of the `classregtree` class:

```
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 of the `classregtree` class.

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 of the `classregtree` class, 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 of the `classregtree` class.

9

Multivariate Methods

Introduction

Large, high-dimensional data sets are common in the modern era of computer-based instrumentation and electronic data storage. High-dimensional data present many challenges for statistical visualization, analysis, and modeling.

Data visualization, of course, is impossible beyond a few dimensions. As a result, pattern recognition, data preprocessing, and model selection must rely heavily on numerical methods.

A fundamental challenge in high-dimensional data analysis is the so-called *curse of dimensionality*. Observations in a high-dimensional space are necessarily sparser and less representative than those in a low-dimensional space. In higher dimensions, data over-represent the edges of a sampling distribution, because regions of higher-dimensional space contain the majority of their volume near the surface. (A d-dimensional spherical shell has a volume, relative to the total volume of the sphere, that approaches 1 as d approaches infinity.) In high dimensions, typical data points at the interior of a distribution are sampled less frequently.

Often, many of the dimensions in a data set—the measured features—are not useful in producing a model. Features may be irrelevant or redundant. Regression and classification algorithms may require large amounts of storage and computation time to process raw data, and even if the algorithms are successful the resulting models may contain an incomprehensible number of terms.

Because of these challenges, multivariate statistical methods often begin with some type of *dimension reduction*, in which data are approximated by points in a lower-dimensional space. Dimension reduction is the goal of the methods presented in this chapter. Dimension reduction often leads to simpler models and fewer measured variables, with consequent benefits when measurements are expensive and visualization is important.

Multidimensional Scaling

Introduction

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. There are no points to 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 metrics 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

- "Introduction" on page 9-4

• "Example: Multidimensional Scaling" on page 9-6

Introduction

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.

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: Multidimensional Scaling

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
```
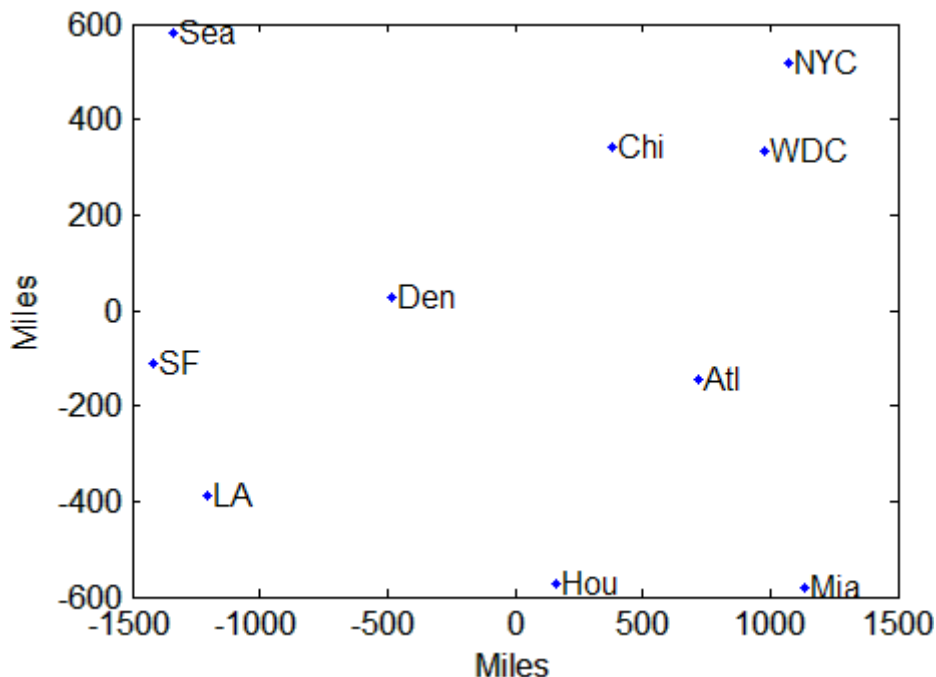
Here is a plot of the reconstructed city locations as a map. The orientation of the reconstruction is arbitrary. In this case, it happens to be close to, although not exactly, the correct orientation.

```
plot(Y(:,1),Y(:,2),'.')
text(Y(:,1)+25,Y(:,2),cities)
xlabel('Miles')
ylabel('Miles')
```

Nonclassical Multidimensional Scaling

The function mdscale performs nonclassical multidimensional scaling. As with cmdcale, you 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
```
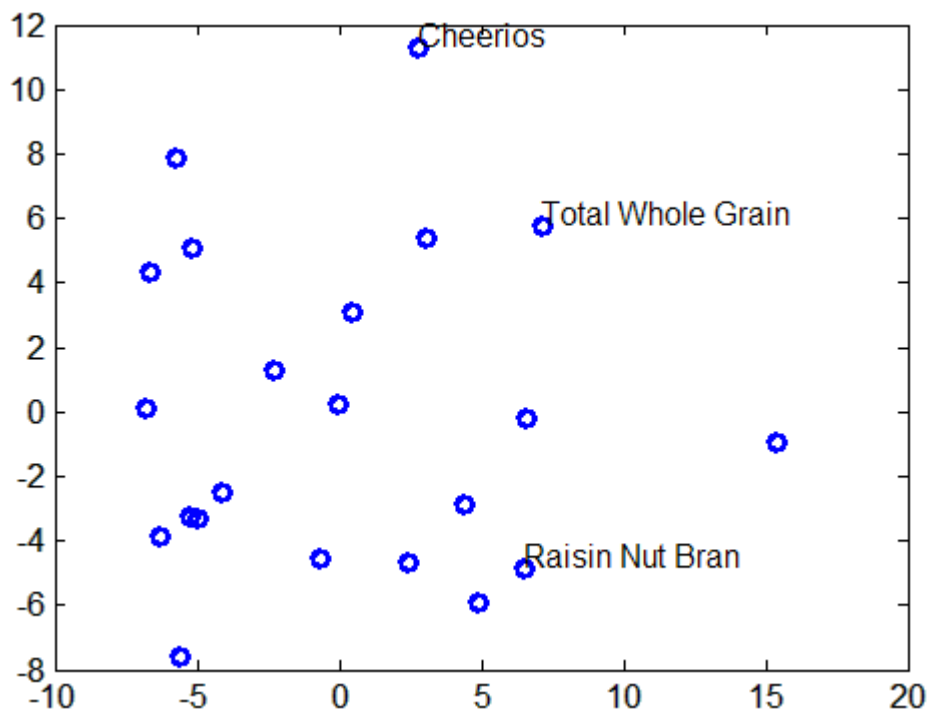
A scatterplot of the output from `mdscale` represents the original 10-dimensional data in two dimensions, and you can use the `gname` function to label selected points:

```
plot(Y(:,1),Y(:,2),'o','LineWidth',2);
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');
```
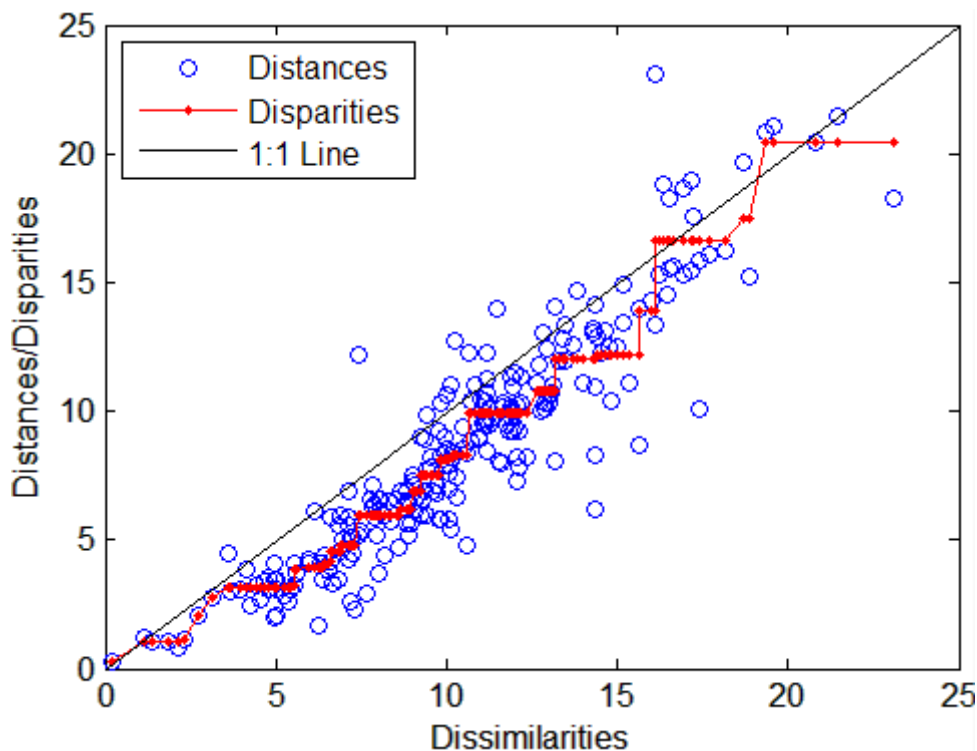
This plot shows that `mdscale` has found a configuration of points in two dimensions whose inter-point distances approximates the disparities, which in turn are a nonlinear transformation of the original dissimilarities. The concave shape of the disparities as a function of the dissimilarities indicates 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.

Procrustes Analysis

Procrustes analysis is a statistical procedure for comparing shapes. It is commonly applied to sets of *landmark data*, in which significant features in a population are measured as geometric locations. The analysis computes the best-fitting superposition of the landmarks in two data sets using a shape-preserving Euclidean transformation that minimizes variations in location, rotation and scale. It is often used as a preprocessing step for further statistical analysis.

Procrustes analysis has its origins in the biological study of animal morphometrics, but has found application in areas as diverse as archeology, astronomy, civil engineering, geography, network design, and physical chemistry. The name comes from Procrustes, a figure in Greek mythology noted for his proclivity for "shrinking" and "stretching" visitors to fit in his bed.

The Statistics Toolbox function `procrustes` carries out Procustes analysis.

Feature Selection

In this section...
"Introduction" on page 9-15
"Sequential Feature Selection" on page 9-15

Introduction

Feature selection reduces the dimensionality of data by selecting only a subset of measured features (predictor variables) to create a model. Selection criteria usually involve the minimization of a specific measure of predictive error for models fit to different subsets. Algorithms search for a subset of predictors that optimally model measured responses, subject to constraints such as required or excluded features and the size of the subset.

Feature selection is preferable to feature transformation when the original units and meaning of features are important and the modeling goal is to identify an influential subset. When categorical features are present, and numerical transformations are inappropriate, feature selection becomes the primary means of dimension reduction.

Sequential Feature Selection

- "Introduction" on page 9-15
- "Example: Sequential Feature Selection" on page 9-16

Introduction

A common method of feature selection is *sequential feature selection*. This method has two components:

- An objective function, called the *criterion*, which the method seeks to minimize over all feasible feature subsets. Common criteria are mean squared error (for regression models) and misclassification rate (for classification models).

- A sequential search algorithm, which adds or removes features from a candidate subset while evaluating the criterion. Since an exhaustive

comparison of the criterion value at all 2^n subsets of an n-feature data set is typically infeasible (depending on the size of n and the cost of objective calls), sequential searches move in only one direction, always growing or always shrinking the candidate set.

The method has two variants:

- *Sequential forward selection* (*SFS*), in which features are sequentially added to an empty candidate set until the addition of further features does not decrease the criterion.

- *Sequential backward selection* (*SBS*), in which features are sequentially removed from a full candidate set until the removal of further features increase the criterion.

Stepwise regression is a sequential feature selection technique designed specifically for least-squares fitting. The functions `stepwise` and `stepwisefit` make use of optimizations that are only possible with least-squares criteria. Unlike generalized sequential feature selection, stepwise regression may remove features that have been added or add features that have been removed.

The Statistics Toolbox function `sequentialfs` carries out sequential feature selection. Input arguments include predictor and response data and a function handle to an M-file implementing the criterion function. Optional inputs allow you to specify SFS or SBS, required or excluded features, and the size of the feature subset. The function calls `cvpartition` and `crossval` to evaluate the criterion at different candidate sets.

Example: Sequential Feature Selection

For example, consider a data set with 100 observations of 10 predictors. As described in "Example: Generalized Linear Models" on page 8-53, the following generates random data from a logistic model, with a binomial distribution of responses at each set of values for the predictors. Some coefficients are set to zero so that not all of the predictors affect the response:

```
n = 100;
m = 10;
X = rand(n,m);
b = [1 0 0 2 .5 0 0 0.1 0 1];
Xb = X*b';
```

```
p = 1./(1+exp(-Xb));
N = 50;
y = binornd(N,p);
```

The glmfit function fits a logistic model to the data:

```
Y = [y N*ones(size(y))];
[b0,dev0,stats0] = glmfit(X,Y,'binomial');

% Display coefficient estimates and their standard errors:
model0 = [b0 stats0.se]
model0 =
     0.3115    0.2596
     0.9614    0.1656
    -0.1100    0.1651
    -0.2165    0.1683
     1.9519    0.1809
     0.5683    0.2018
    -0.0062    0.1740
     0.0651    0.1641
    -0.1034    0.1685
     0.0017    0.1815
     0.7979    0.1806

% Display the deviance of the fit:
dev0
dev0 =
   101.2594
```

This is the full model, using all of the features (and an initial constant term). Sequential feature selection searches for a subset of the features in the full model with comparative predictive power.

First, you must specify a criterion for selecting the features. The following function, which calls glmfit and returns the deviance of the fit (a generalization of the residual sum of squares) is a useful criterion in this case:

```
function dev = critfun(X,Y)

[b,dev] = glmfit(X,Y,'binomial');
```

This function should be created as an M-file on the MATLAB path.

The function `sequentialfs` performs feature selection, calling the criterion function via a function handle:

```
maxdev = chi2inv(.95,1);
opt = statset('display','iter',...
              'TolFun',maxdev,...
              'TolTypeFun','abs');

inmodel = sequentialfs(@critfun,X,Y,...
                       'cv','none',...
                       'nullmodel',true,...
                       'options',opt,...
                       'direction','forward');

Start forward sequential feature selection:
Initial columns included:  none
Columns that can not be included:  none
Step 1, used initial columns, criterion value 309.118
Step 2, added column 4, criterion value 180.732
Step 3, added column 1, criterion value 138.862
Step 4, added column 10, criterion value 114.238
Step 5, added column 5, criterion value 103.503
Final columns included:  1 4 5 10
```

The iterative display shows a decrease in the criterion value as each new feature is added to the model. The final result is a reduced model with only four of the original ten features: columns 1, 4, 5, and 10 of X. These features are indicated in the logical vector `inmodel` returned by `sequentialfs`.

The deviance of the reduced model is higher than for the full model, but the addition of any other single feature would not decrease the criterion by more than the absolute tolerance, `maxdev`, set in the options structure. Adding a feature with no effect reduces the deviance by an amount that has a chi-square distribution with one degree of freedom. Adding a significant feature results in a larger change. By setting `maxdev` to `chi2inv(.95,1)`, you instruct `sequentialfs` to continue adding features so long as the change in deviance is more than would be expected by random chance.

The reduced model (also with an initial constant term) is:

```
[b,dev,stats] = glmfit(x(:,in),Y,'binomial');

% Display coefficient estimates and their standard errors:
model = [b stats.se]
model =
    0.0784    0.1642
    1.0040    0.1592
    1.9459    0.1789
    0.6134    0.1872
    0.8245    0.1730
```

Feature Transformation

Introduction

Feature transformation is a group of methods that create new features (predictor variables). The methods are useful for dimension reduction when the transformed features have a descriptive power that is more easily ordered than the original features. In this case, less descriptive features can be dropped from consideration when building models.

Feature transformation methods are contrasted with the methods presented in "Feature Selection" on page 9-15, where dimension reduction is achieved by computing an optimal subset of predictive features measured in the original data.

The methods presented in this section share some common methodology. Their goals, however, are essentially different:

- Nonnegative matrix factorization is used when model terms must represent nonnegative quantities, such as physical quantities.

- Principal component analysis is used to summarize data in fewer dimensions, for example, to visualize it.

- Factor analysis is used to build explanatory models of data correlations.

Nonnegative Matrix Factorization

- "Introduction" on page 9-21
- "Example: Nonnegative Matrix Factorization" on page 9-21

Introduction

Nonnegative matrix factorization (*NMF*) is a dimension-reduction technique based on a low-rank approximation of the feature space. Besides providing a reduction in the number of features, NMF guarantees that the features are nonnegative, producing additive models that respect, for example, the nonnegativity of physical quantities.

Given a nonnegative m-by-n matrix X and a positive integer $k < \min(m,n)$, NMF finds nonnegative m-by-k and k-by-n matrices W and H, respectively, that minimize the norm of the difference $X - WH$. W and H are thus approximate nonnegative factors of X.

The k columns of W represent transformations of the variables in X; the k rows of H represent the coefficients of the linear combinations of the original n variables in X that produce the transformed variables in W. Since k is generally smaller than the rank of X, the product WH provides a compressed approximation of the data in X. A range of possible values for k is often suggested by the modeling context.

The Statistics Toolbox function nnmf carries out nonnegative matrix factorization. nnmf uses one of two iterative algorithms that begin with random initial values for W and H. Because the norm of the residual $X - WH$ may have local minima, repeated calls to nnmf may yield different factorizations. Sometimes the algorithm converges to a solution of lower rank than k, which may indicate that the result is not optimal.

Example: Nonnegative Matrix Factorization

For example, consider the five predictors of biochemical oxygen demand in the data set moore.mat:

```
load moore
X = moore(:,1:5);
```

The following uses nnmf to compute a rank-two approximation of X with a multiplicative update algorithm that begins from five random initial values for W and H:

```
opt = statset('MaxIter',10,'Display','final');
[W0,H0] = nnmf(X,2,'replicates',5,...
                    'options',opt,...
```

```
                    'algorithm','mult');
     rep iteration     rms resid    |delta x|
       1        10      358.296   0.00190554
       2        10      78.3556  0.000351747
       3        10      230.962    0.0172839
       4        10      326.347   0.00739552
       5        10      361.547   0.00705539
Final root mean square residual = 78.3556
```

The 'mult' algorithm is sensitive to initial values, which makes it a good
choice when using 'replicates' to find W and H from multiple random
starting values.

Now perform the factorization using an alternating least-squares algorithm,
which converges faster and more consistently. Run 100 times more iterations,
beginning from the initial W0 and H0 identified above:

```
opt = statset('Maxiter',1000,'Display','final');
[W,H] = nnmf(X,2,'w0',W0,'h0',H0,...
                 'options',opt,...
                 'algorithm','als');
     rep iteration     rms resid    |delta x|
       1         3       77.5315 3.52673e-005
Final root mean square residual = 77.5315
```

The two columns of W are the transformed predictors. The two rows of H give
the relative contributions of each of the five predictors in X to the predictors
in W:

```
H
H =
    0.0835  0.0190  0.1782  0.0072  0.9802
    0.0558  0.0250  0.9969  0.0085  0.0497
```

The fifth predictor in X (weight 0.9802) strongly influences the first predictor
in W. The third predictor in X (weight 0.9969) strongly influences the second
predictor in W.

Visualize the relative contributions of the predictors in X with a biplot,
showing the data and original variables in the column space of W:

```
biplot(H','scores',W,'varlabels',{'','','v3','','v5'});
axis([0 1.1 0 1.1])
xlabel('Column 1')
ylabel('Column 2')
```
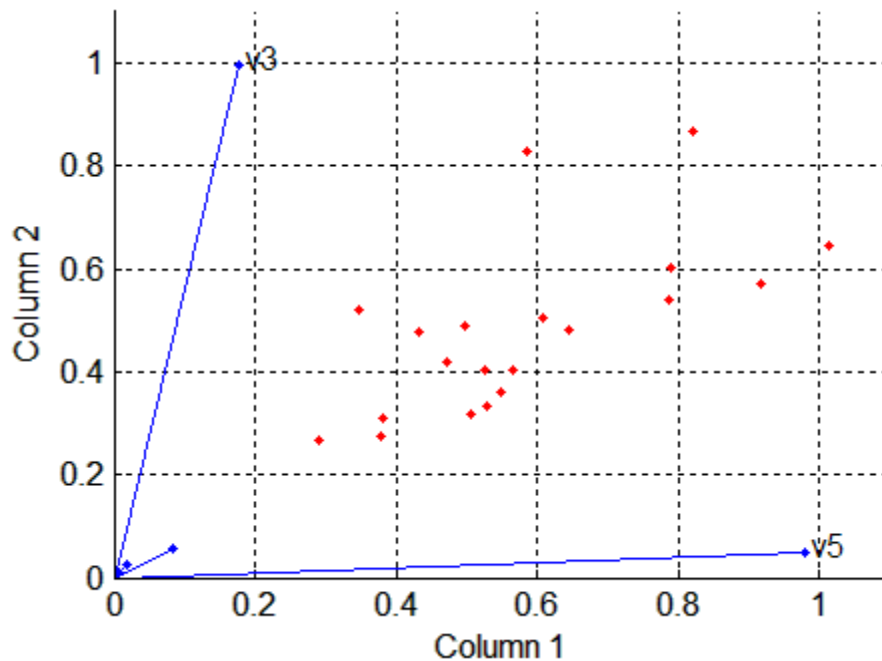

Principal Component Analysis

- "Introduction" on page 9-23
- "Example: Principal Component Analysis" on page 9-25

Introduction

One of the difficulties inherent in multivariate statistics is the problem of visualizing data that has many variables. The MATLAB function `plot` 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 component 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. 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 Component Analysis

- "Computing Components" on page 9-25
- "Component Coefficients" on page 9-28
- "Component Scores" on page 9-28
- "Component Variances" on page 9-32
- "Hotelling's T2" on page 9-34
- "Visualizing the Results" on page 9-34

Computing Components. 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.
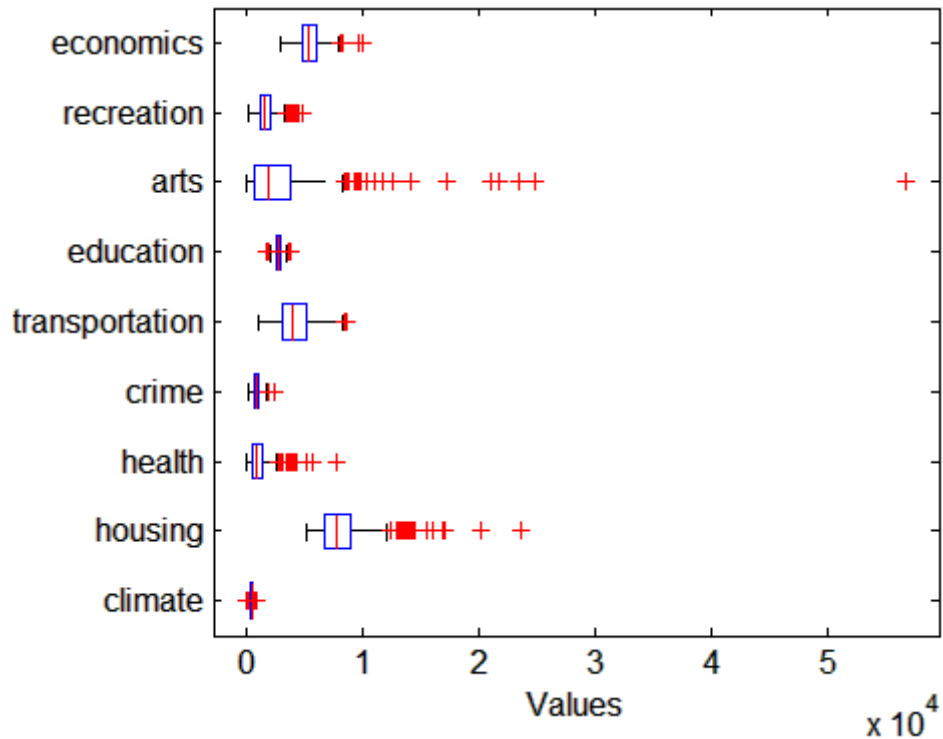
Ordinarily you might also graph pairs of the original variables, but there are 36 two-variable plots. Perhaps principal components analysis can reduce the number of variables you need to consider.

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`.

Component Coefficients. The first output of the `princomp` function, `coefs`, contains the coefficients of the linear combinations of the original variables that generate the principal components. The coefficients are also known as *loadings*.

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.

The principal components are unit length and orthogonal:

```
I = c3'*c3
I =
    1.0000   -0.0000   -0.0000
   -0.0000    1.0000   -0.0000
   -0.0000   -0.0000    1.0000
```

Component Scores. The second output, `scores`, contains the coordinates of the original data in 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')
```
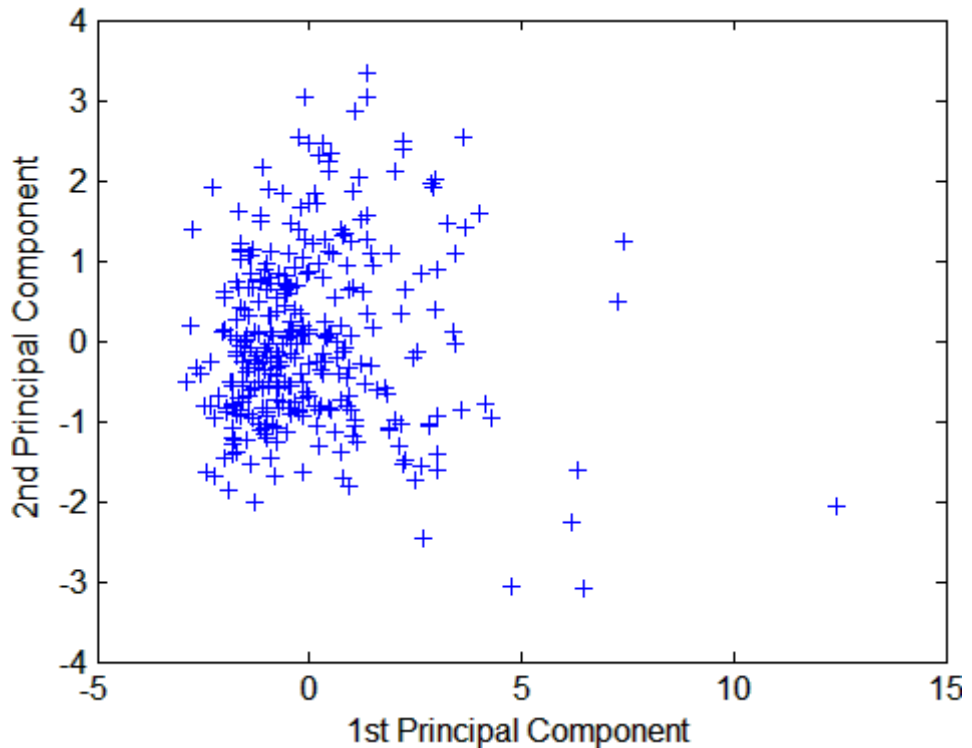

Note the outlying points in the right half of the plot.

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, it is labeled with the proper row from the `names` string matrix. Here is the plot after a few clicks:
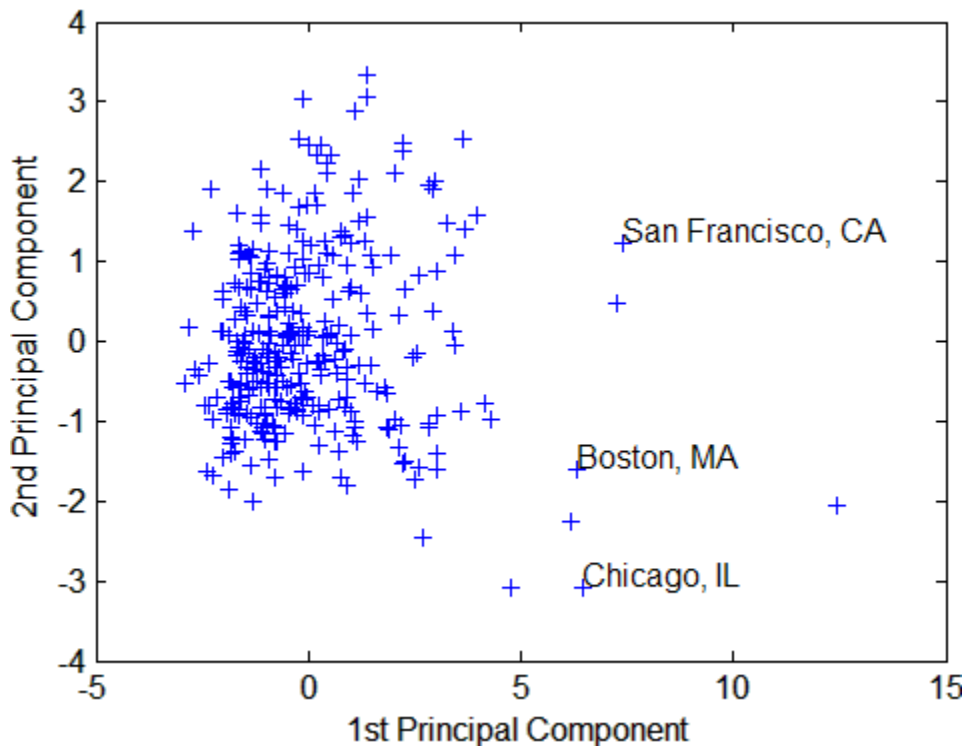
When you are finished labeling points, press the **Return** key.

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.
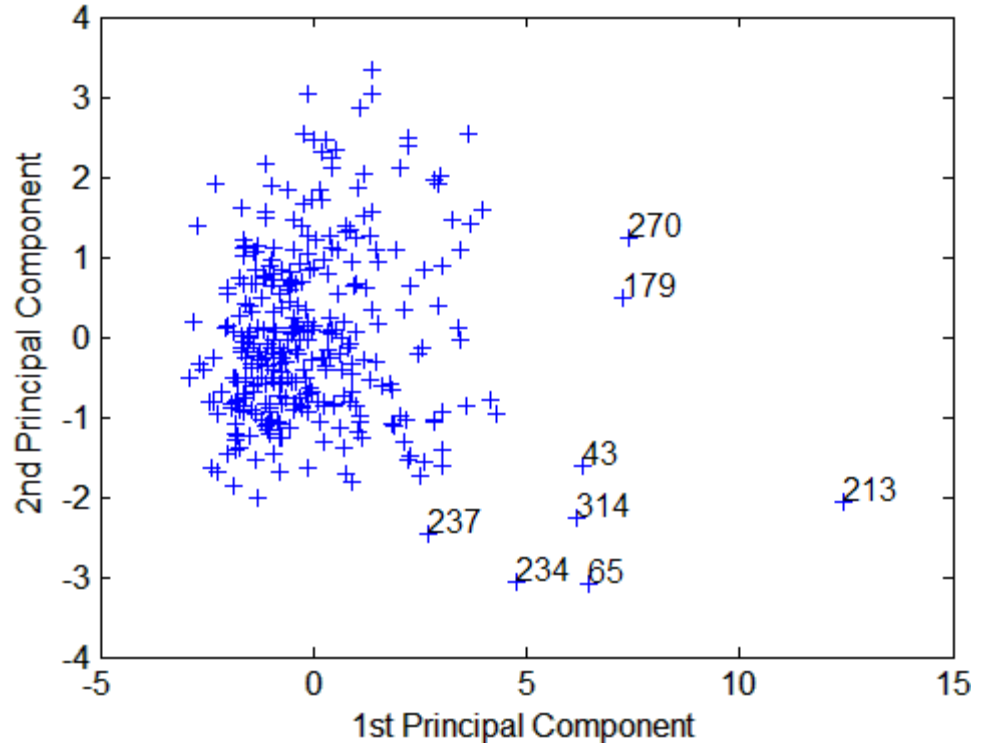
Then you can create an index variable containing the row numbers of all the metropolitan areas you choose.

```
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
```

Component Variances. 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 (%)')
```
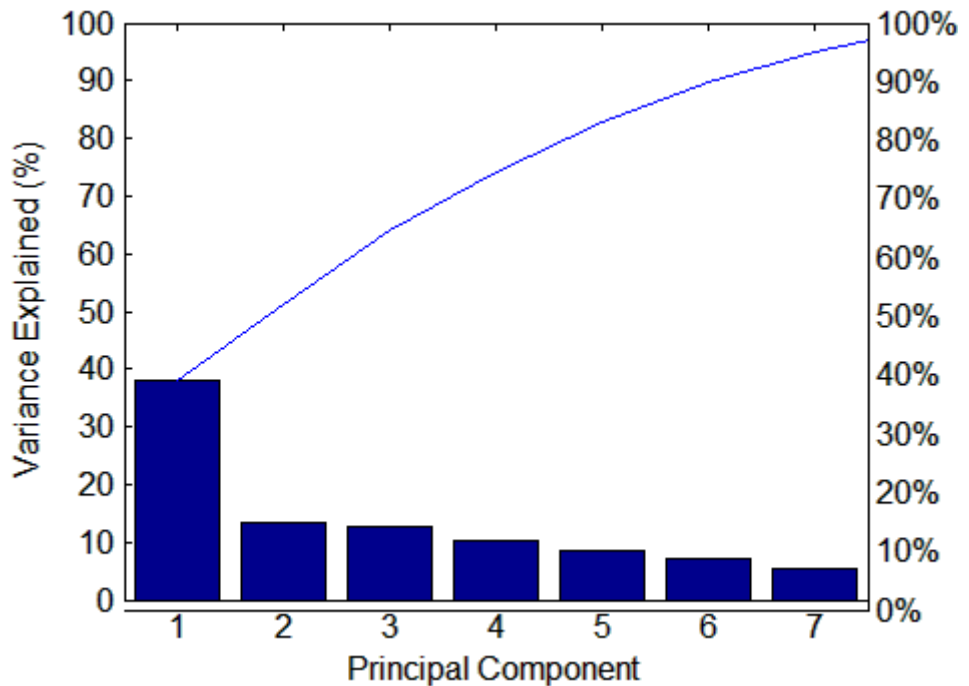
The preceding figure shows that the only clear break in the amount of variance accounted for by each component is between the first and second components. However, that component by itself explains less than 40% of the variance, so more components are probably needed. You can see that the first three 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. 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. 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]);
```
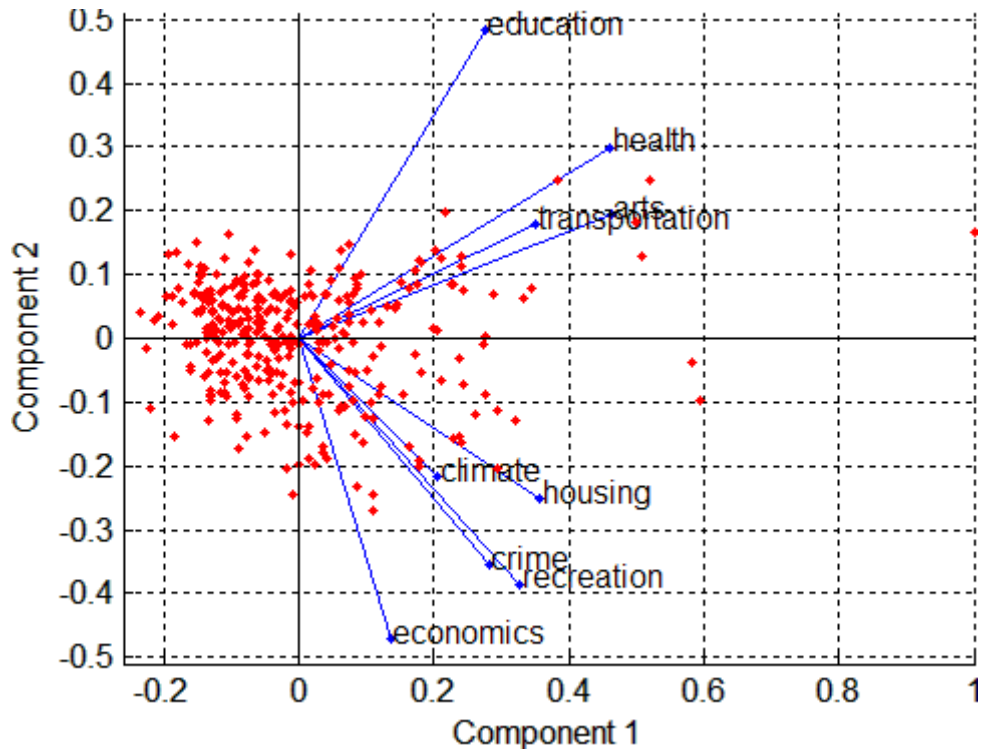
Each of the nine variables is represented in this plot by a vector, and the direction and length of the vector indicates how each variable contributes to the two principal components in the plot. For example, you have seen that the first principal component, represented in this biplot by the horizontal 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 transportation, 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]);
```
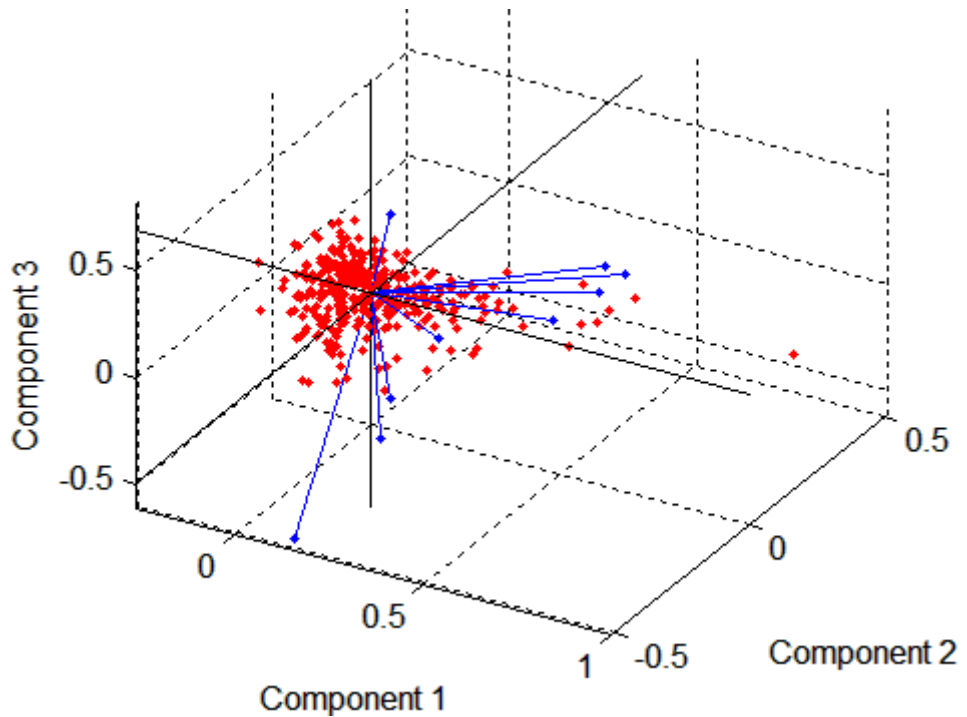
Factor Analysis

- "Introduction" on page 9-37
- "Example: Factor Analysis" on page 9-38

Introduction

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.

Example: Factor Analysis

- "Factor Loadings" on page 9-38
- "Factor Rotation" on page 9-40
- "Factor Scores" on page 9-42
- "Visualizing the Results" on page 9-44

Factor Loadings. 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 9-40 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 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);
```
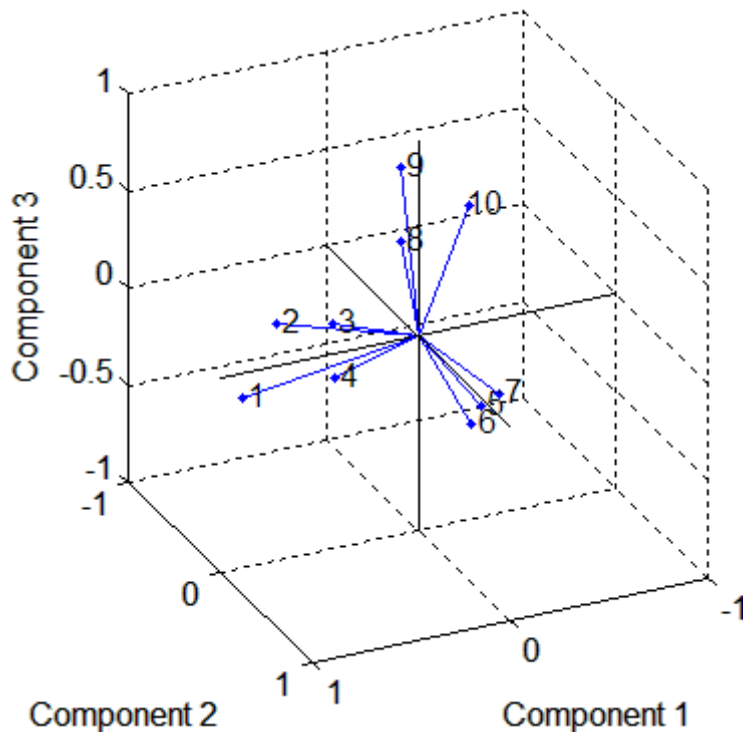
This plot shows that promax has rotated the factor loadings to a simpler structure. Each stock depends primarily on only one factor, and it is possible to describe each factor in terms of the stocks that it affects. Based on which companies are near which axes, you could reasonably conclude that the 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.

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');

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)
```
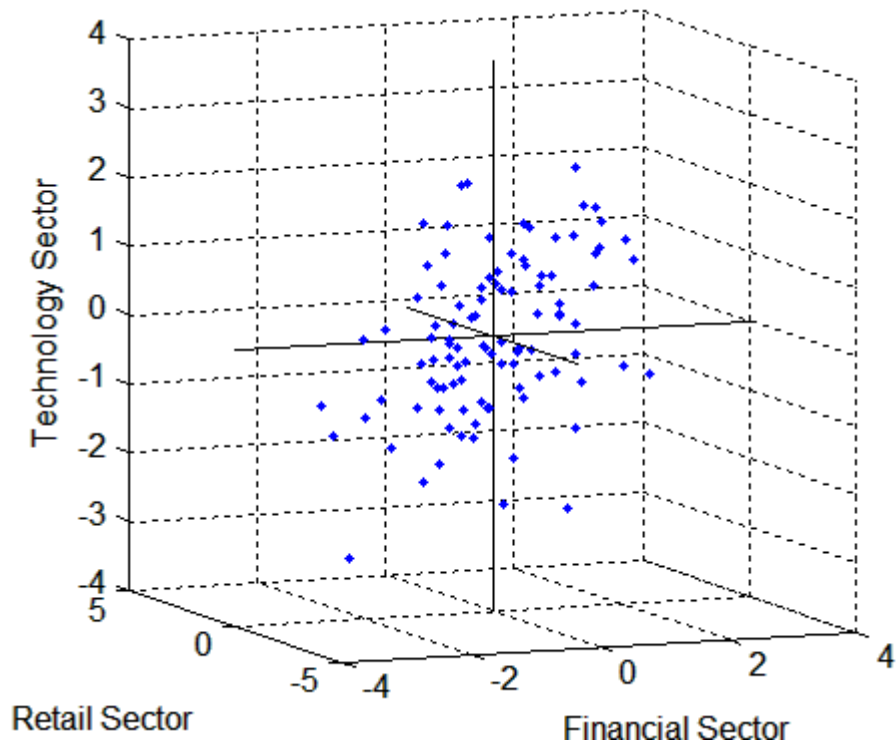
Oblique rotation often creates factors that are correlated. This plot shows some evidence of correlation between the first and third factors, and you can investigate further by computing the estimated factor correlation matrix.

```
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. 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)
```
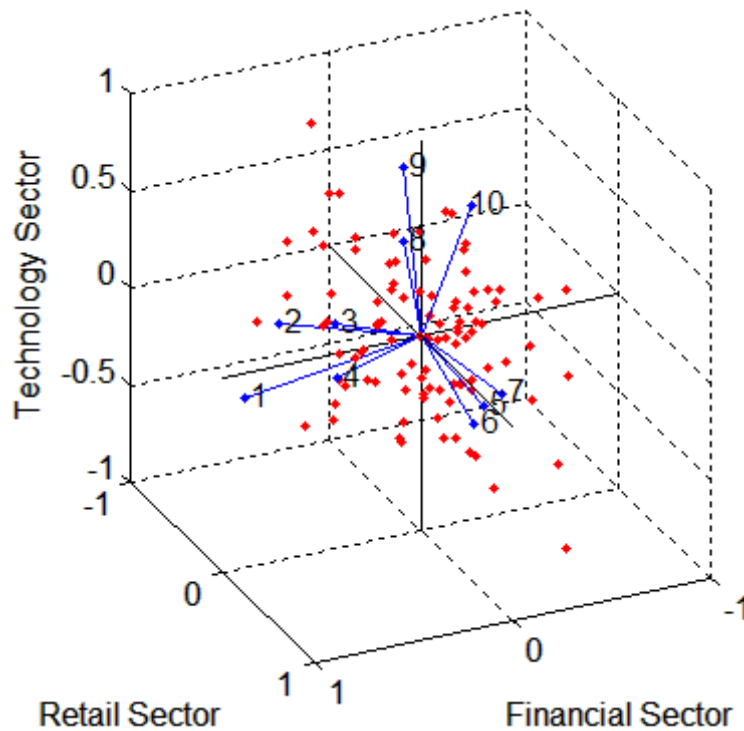
In this case, the factor analysis includes three factors, and so the biplot is three-dimensional. Each of the 10 stocks is represented in this plot by a vector, and the direction and length of the vector indicates how each stock depends on the underlying factors. For example, you have seen that after 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.

10

Cluster Analysis

Introduction

Cluster analysis, also called *segmentation analysis* or *taxonomy analysis*, creates groups, or *clusters*, of data. Clusters are formed in such a way that objects in the same cluster are very similar and objects in different clusters are very distinct. Measures of similarity depend on the application.

"Hierarchical Clustering" on page 10-3 groups data over a variety of scales by creating a cluster tree or *dendrogram*. 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 level. This allows you to decide the level or scale of clustering that is most appropriate for your application. The Statistics Toolbox function clusterdata performs all of the necessary steps for you. It incorporates the pdist, linkage, and cluster functions, which may be used separately for more detailed analysis. The dendrogram function plots the cluster tree.

"K-Means Clustering" on page 10-21 is a partitioning method. The function kmeans partitions data into k mutually exclusive clusters, and returns the index of the cluster to which it has assigned each observation. Unlike hierarchical clustering, k-means clustering operates on actual observations (rather than the larger set of dissimilarity measures), and creates a single level of clusters. The distinctions mean that k-means clustering is often more suitable than hierarchical clustering for large amounts of data.

"Gaussian Mixture Models" on page 10-28 form clusters by representing the probability density function of observed variables as a mixture of multivariate normal densities. Mixture models of the gmdistribution class are fit to data using an expectation maximization (EM) algorithm, which assigns posterior probabilities to each component density with respect to each observation. Clusters are assigned by selecting the component that maximizes the posterior probability. Like k-means clustering, Gaussian mixture modeling uses an iterative algorithm that converges to a local optimum. Gaussian mixture modeling may be more appropriate than k-means clustering when clusters have different sizes and correlation within them.

Hierarchical Clustering

In this section...
"Introduction" on page 10-3
"Algorithm Description" on page 10-3
"Similarity Measures" on page 10-4
"Linkages" on page 10-6
"Dendrograms" on page 10-8
"Verifying the Cluster Tree" on page 10-10
"Creating Clusters" on page 10-16

Introduction

Hierarchical clustering groups data over a variety of scales by creating a cluster tree or *dendrogram*. 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 level. This allows you to decide the level or scale of clustering that is most appropriate for your application. The Statistics Toolbox function `clusterdata` performs all of the necessary steps for you. It incorporates the `pdist`, `linkage`, and `cluster` functions, which may be used separately for more detailed analysis. The `dendrogram` function plots the cluster tree.

Algorithm Description

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 "Similarity Measures" on page 10-4 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 "Linkages" on page 10-6 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.

The following sections provide more information about each of these steps.

Note The Statistics Toolbox function `clusterdata` performs all of the necessary steps for you. You do not need to execute the `pdist`, `linkage`, or `cluster` functions separately.

Similarity Measures

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.
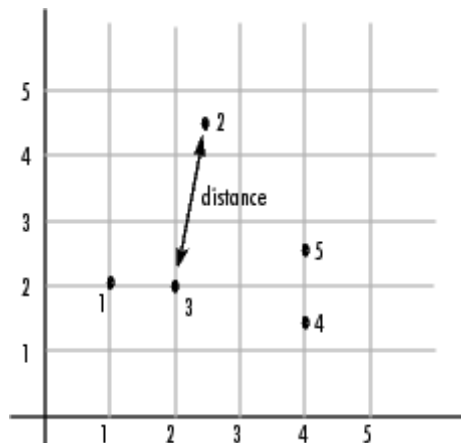
Distance Information

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
```

Linkages

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. 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 4 and 5, which have the closest proximity (distance value = 1.0000). The linkage function continues by grouping objects 1 and 3, 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 4 and 5. 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 1 and 3.

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.
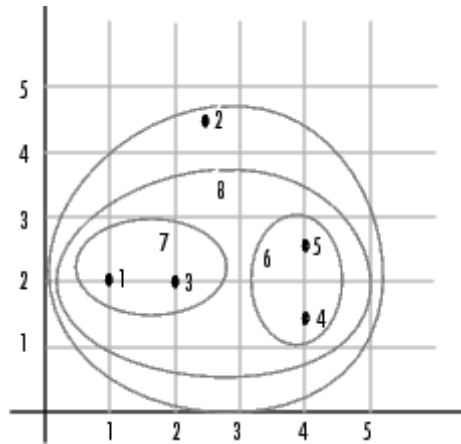
Dendrograms

The hierarchical, binary cluster tree created by the `linkage` function is most easily understood when viewed graphically. The Statistics Toolbox function `dendrogram` plots the tree, as follows:

```
dendrogram(Z)
```
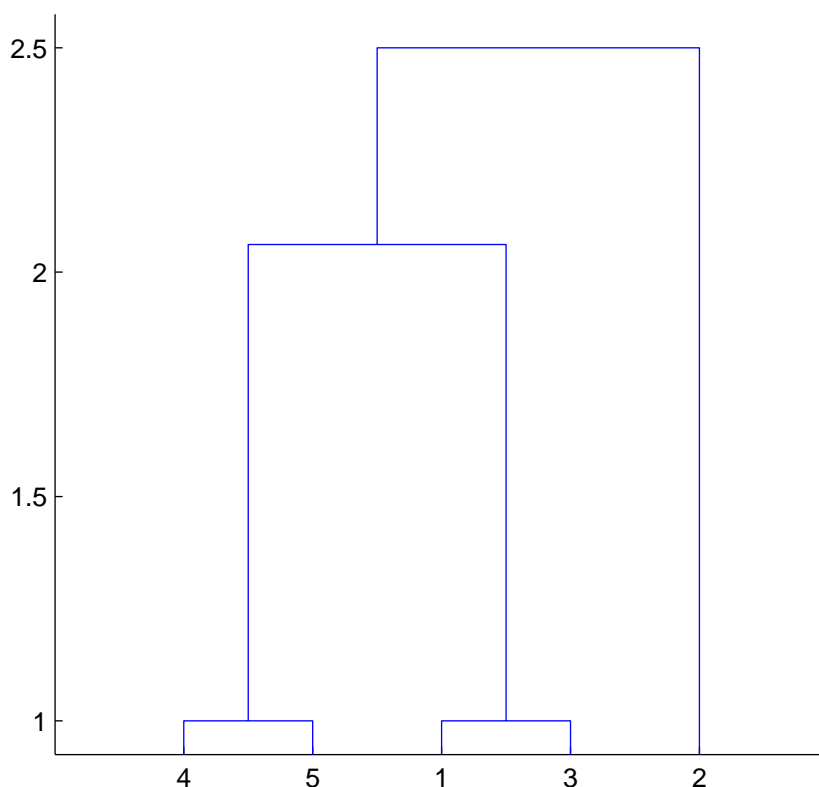
In the figure, the numbers along the horizontal axis represent the indices of the objects in the original data set. The links between objects are represented as upside-down U-shaped lines. The height of the U indicates the distance between the objects. For example, the link representing the cluster containing 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.

Verifying the Cluster Tree

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 functions are available for both of these tasks, as described in the following sections:

- "Verifying Dissimilarity" on page 10-10
- "Verifying Consistency" on page 10-11

Verifying Dissimilarity

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 =
    0.9047
```

The cophenetic correlation coefficient shows that using a different distance and linkage method creates a tree that represents the original distances slightly better.

Verifying Consistency

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.

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.

These links show inconsistency when compared
to the links below them.
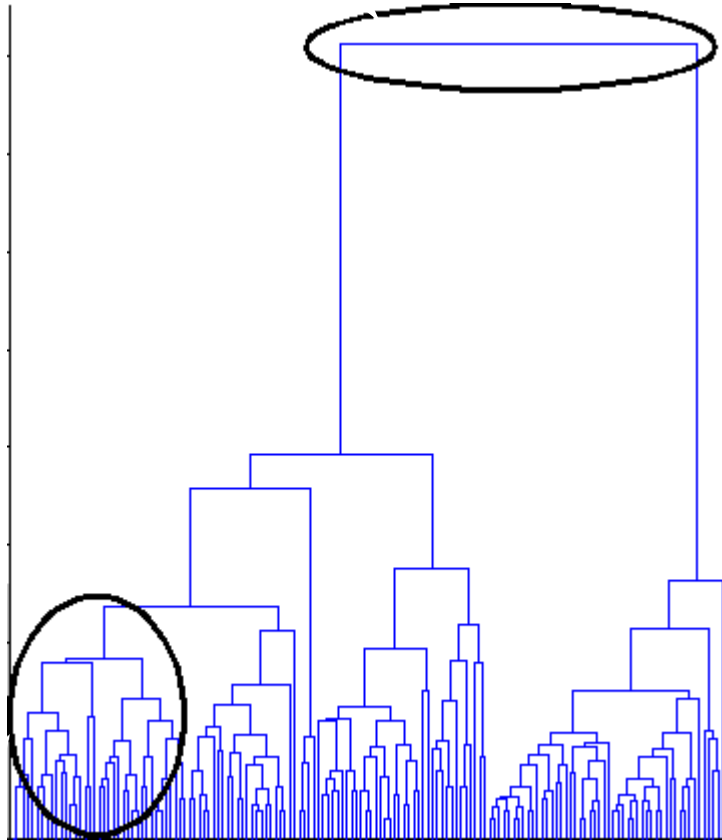
These links show consistency.

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 high inconsistency coefficient;
links that join indistinct clusters have a low 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 "Linkages" on page 10-6.

```
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.
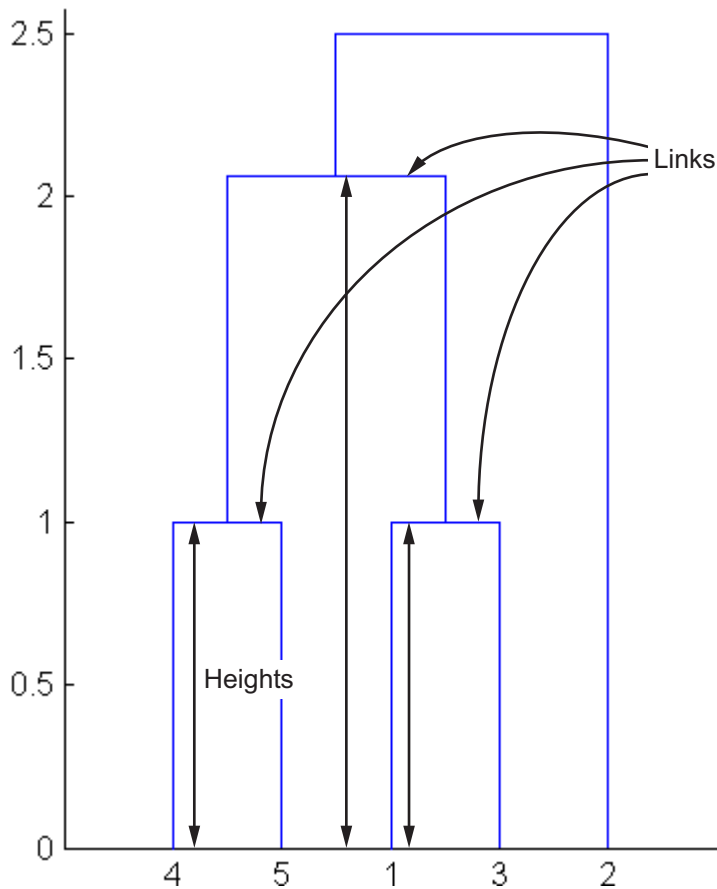
Note In the preceding figure, the lower limit on the *y*-axis is set to `0` to show the heights of the links. To set the lower limit to `0`, select `Axes Properties` from the **Edit** menu, click the **Y Axis** tab, and enter `0` in the field immediately to the right of **Y Limits**.

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.
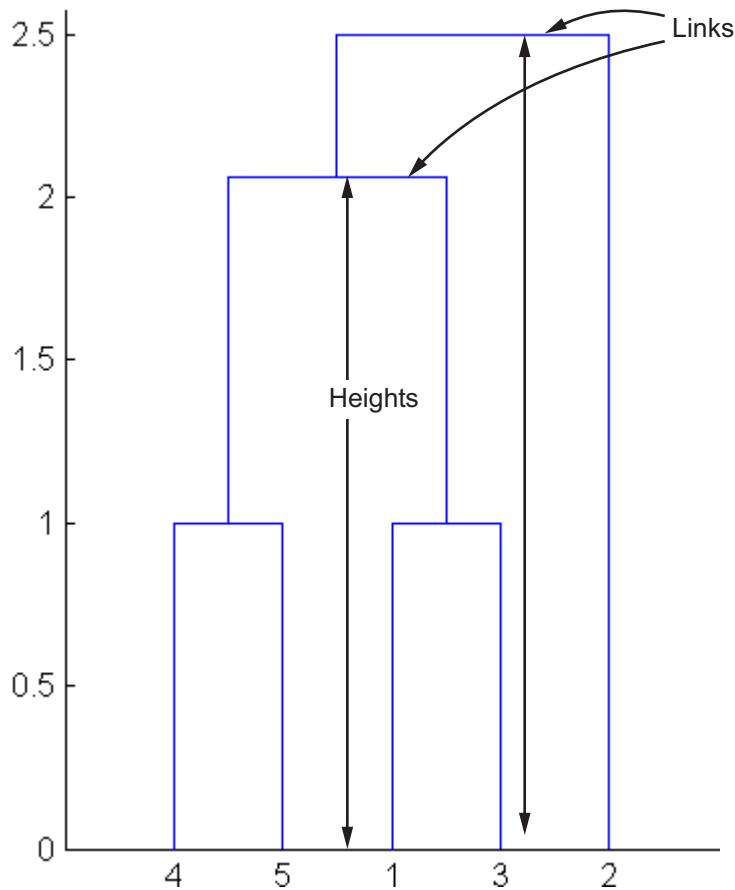
Creating Clusters

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 10-17

- "Specifying Arbitrary Clusters" on page 10-18

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 "Verifying the Cluster Tree" on page 10-10 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 =
     3
     2
     3
     1
     1
```

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.
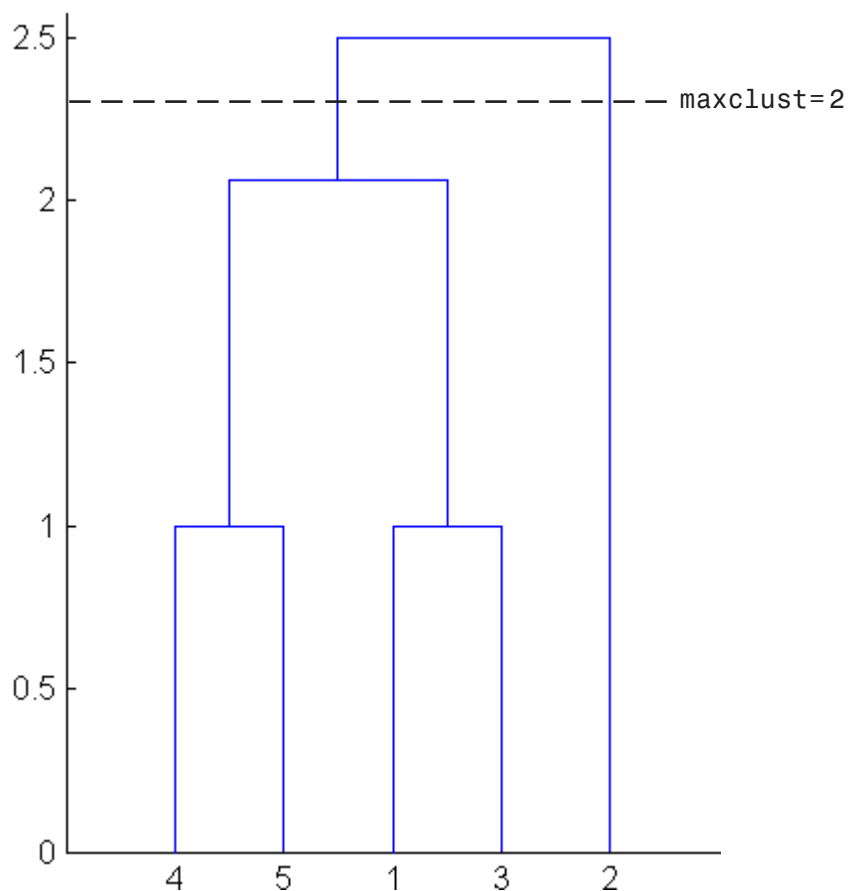
On the other hand, if you set `'maxclust'` to 3, the cluster function groups objects 4 and 5 in one cluster, objects 1 and 3 in a second cluster, and object 2 in a third cluster. The following command illustrates this.

```
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.
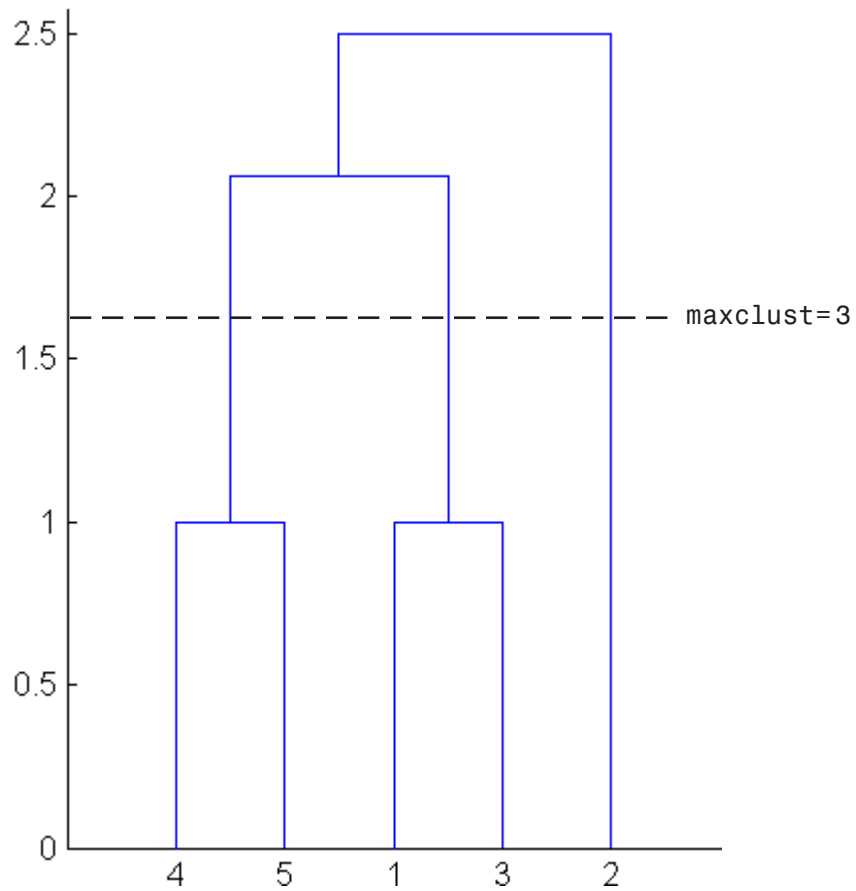
K-Means Clustering

In this section...

Introduction

K-means clustering is a partitioning method. The function `kmeans` partitions data into *k* mutually exclusive clusters, and returns the index of the cluster to which it has assigned each observation. Unlike hierarchical clustering, *k*-means clustering operates on actual observations (rather than the larger set of dissimilarity measures), and creates a single level of clusters. The distinctions mean that *k*-means clustering is often more suitable than hierarchical clustering for 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.

Creating Clusters and Determining Separation

The following 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.

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');
set(get(gca,'Children'),'FaceColor',[.8 .8 1])
xlabel('Silhouette Value')
ylabel('Cluster')
```
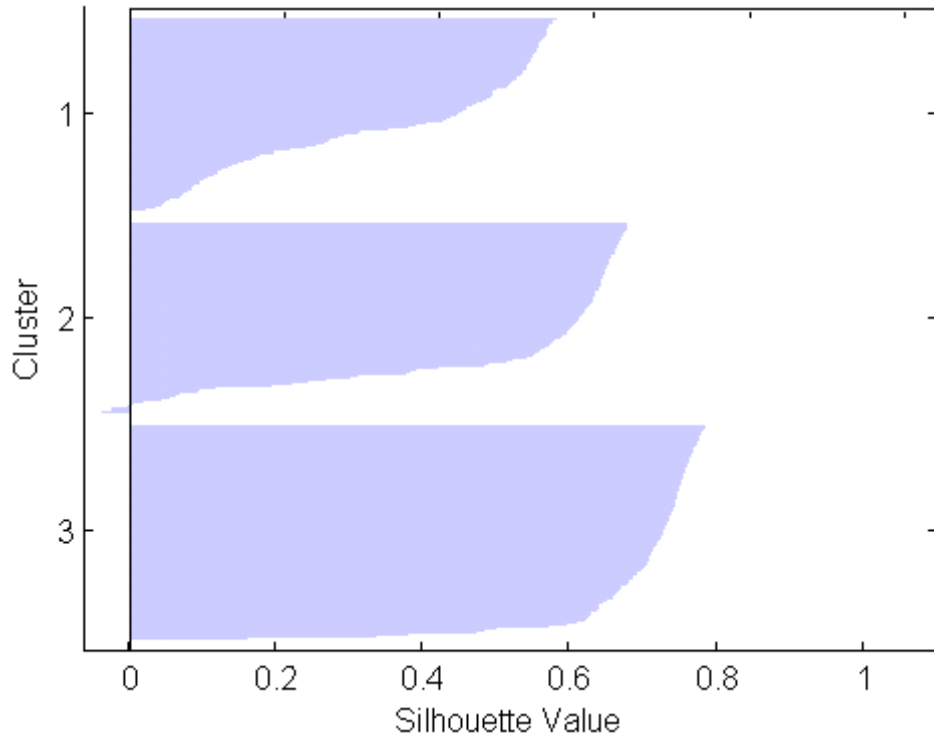
From the silhouette plot, you can see that most points in the third cluster have a large silhouette value, greater than 0.6, indicating that the cluster is somewhat separated from neighboring clusters. However, the second cluster contains many points with low silhouette values, and the first contains 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');
set(get(gca,'Children'),'FaceColor',[.8 .8 1])
xlabel('Silhouette Value')
ylabel('Cluster')
```
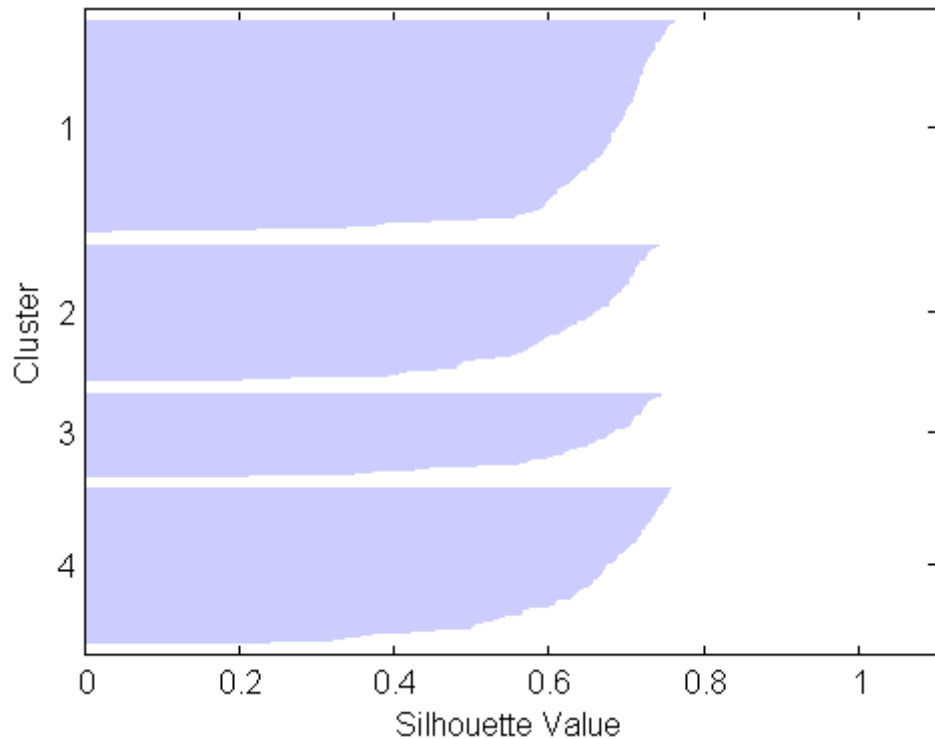
A more quantitative way to compare the two solutions is to look at the average silhouette values for the two cases:

```
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');
set(get(gca,'Children'),'FaceColor',[.8 .8 1])
xlabel('Silhouette Value')
```

```
ylabel('Cluster')
mean(silh5)
ans =
     0.52657
```
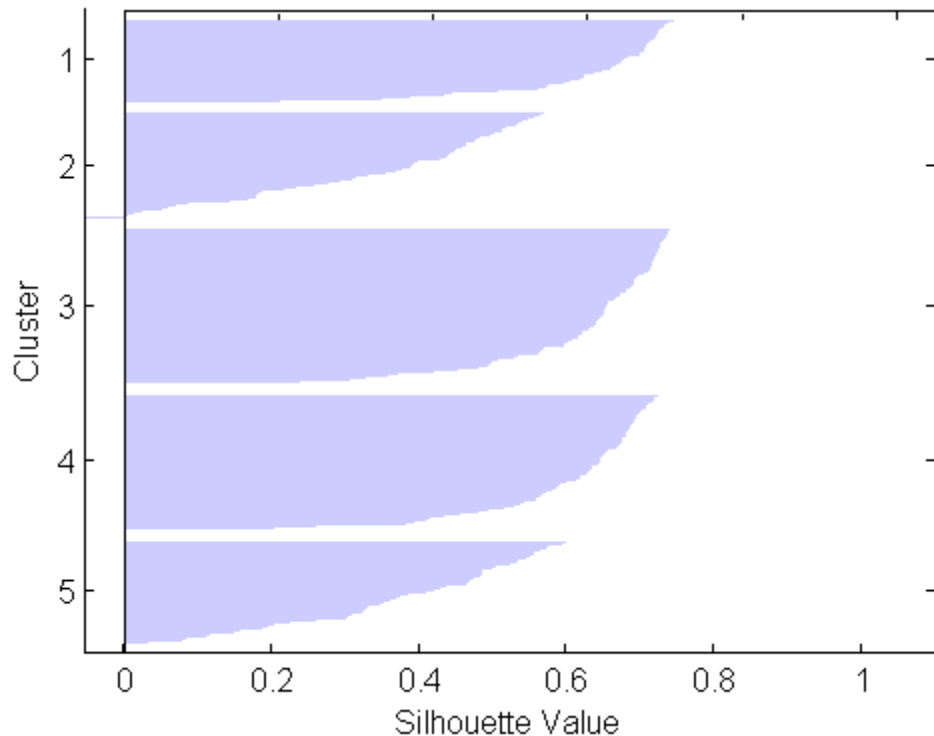

This silhouette plot indicates that this is probably not the right number of clusters, since two of the clusters contain points with mostly low silhouette values. Without some knowledge of how many clusters are really in the data, it is a good idea to experiment with a range of values for k.

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
```

Gaussian Mixture Models

In this section...
"Introduction" on page 10-28
"Clustering with Gaussian Mixtures" on page 10-28

Introduction

Gaussian mixture models are formed by combining multivariate normal density components. For information on individual multivariate normal densities, see "Multivariate Normal Distribution" on page B-53 and related distribution functions listed under "Multivariate Distributions" on page 5-8.

In Statistics Toolbox software, mixture models of the `gmdistribution` class are fit to data using an expectation maximization (EM) algorithm, which assigns posterior probabilities to each component density with respect to each observation.

Gaussian mixture models are often used for data clustering. Clusters are assigned by selecting the component that maximizes the posterior probability. Like *k*-means clustering, Gaussian mixture modeling uses an iterative algorithm that converges to a local optimum. Gaussian mixture modeling may be more appropriate than *k*-means clustering when clusters have different sizes and correlation within them.

Creation of Gaussian mixture models is described in the "Gaussian Mixture Models" on page 5-91 section of Chapter 5, "Probability Distributions". This section describes their application in cluster analysis.

Clustering with Gaussian Mixtures

Use the `cluster` method of the `gmdistribution` class to cluster data with Gaussian mixture models. The method takes as input a `gmdistribution` object `obj` and a data matrix `X`. The method assigns a cluster to each observation in `X` by choosing the component of `obj` with the largest posterior probability, weighted by the component probability.

Note The data in X is typically the same as the data used to create the Gaussian mixture distribution defined by obj. Clustering with cluster is treated as a separate step, apart from density estimation. For cluster to provide meaningful clustering with new data, X should come from the same population as the data used to create obj.

The following example illustrates this procedure.

First, generate data from a mixture of two bivariate Gaussian distributions using the mvnrnd function:

```
MU1 = [1 2];
SIGMA1 = [2 0; 0 .5];
MU2 = [-3 -5];
SIGMA2 = [1 0; 0 1];
X = [mvnrnd(MU1,SIGMA1,1000);mvnrnd(MU2,SIGMA2,1000)];

scatter(X(:,1),X(:,2),10,'.')
hold on
```
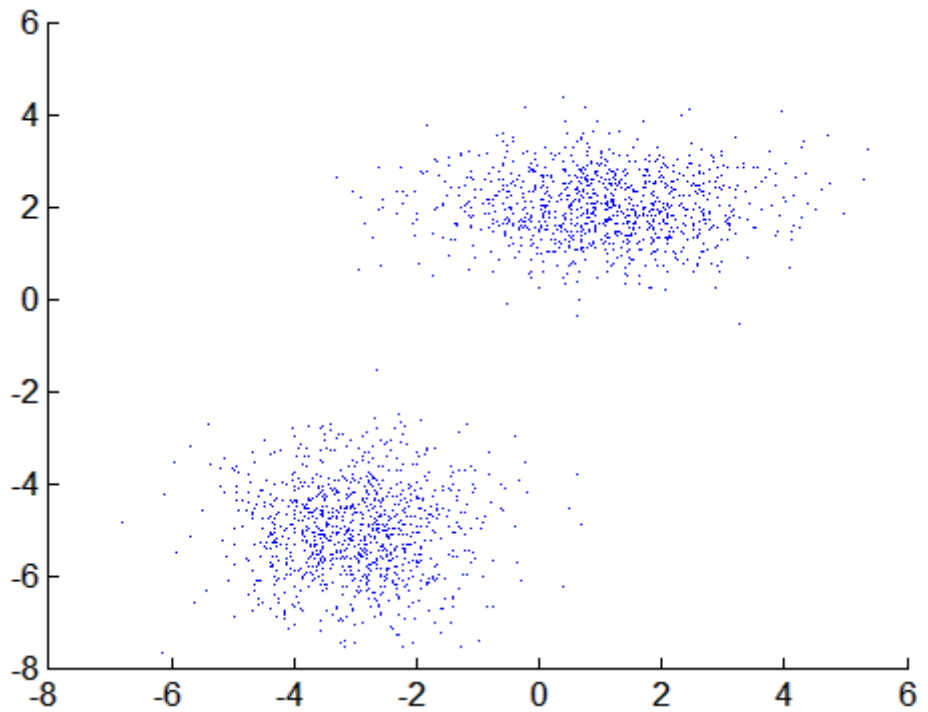
Next, fit a two-component Gaussian mixture model:

```
options = statset('Display','final');
obj = gmdistribution.fit(X,2,'Options',options);
10 iterations, log-likelihood = -7046.78

h = ezcontour(@(x,y)pdf(obj,[x y]),[-8 6],[-8 6]);
```
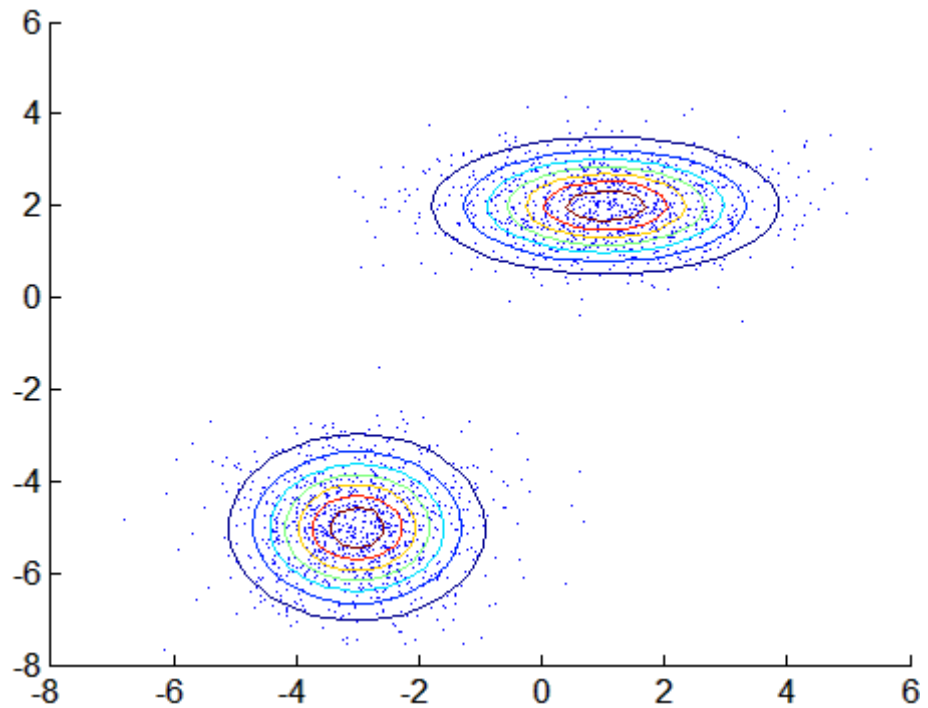
Finally, use the fit to cluster the data:

```
idx = cluster(obj,X);
cluster1 = X(idx == 1,:);
cluster2 = X(idx == 2,:);

delete(h)
h1 = scatter(cluster1(:,1),cluster1(:,2),10,'r.');
h2 = scatter(cluster2(:,1),cluster2(:,2),10,'g.');
legend([h1 h2],'Cluster 1','Cluster 2','Location','NW')
```
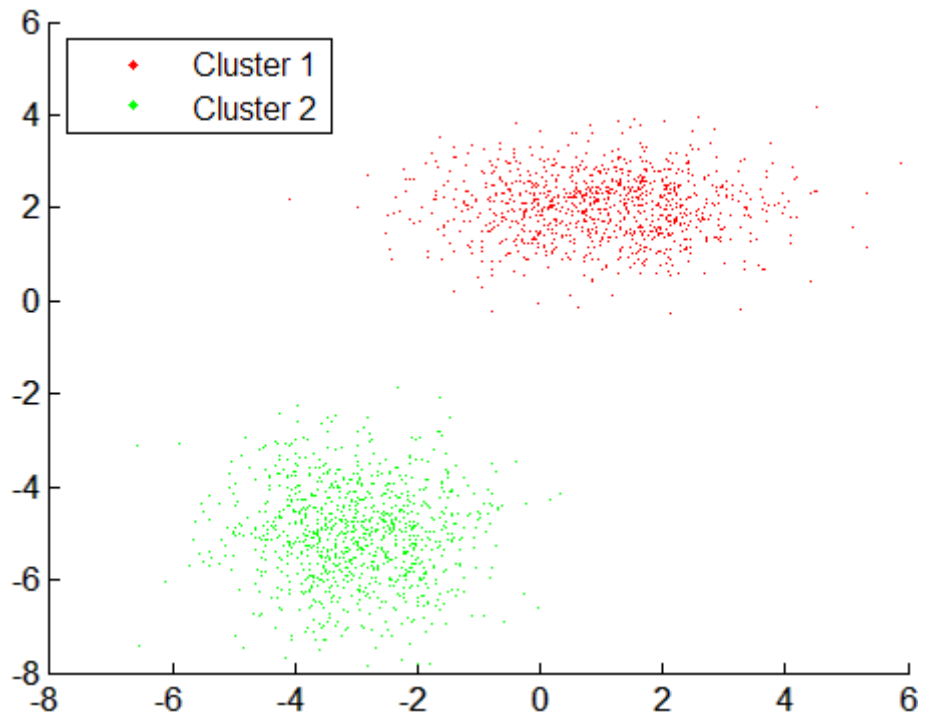
The `posterior` method of the `gmdistribution` class returns the posterior probabilities for each cluster used to cluster the data:

```
P = posterior(obj,X);

figure
scatter(X(:,1),X(:,2),10,P(:,1),'.')
hb = colorbar;
ylabel(hb,'Component 1 Probability')
```
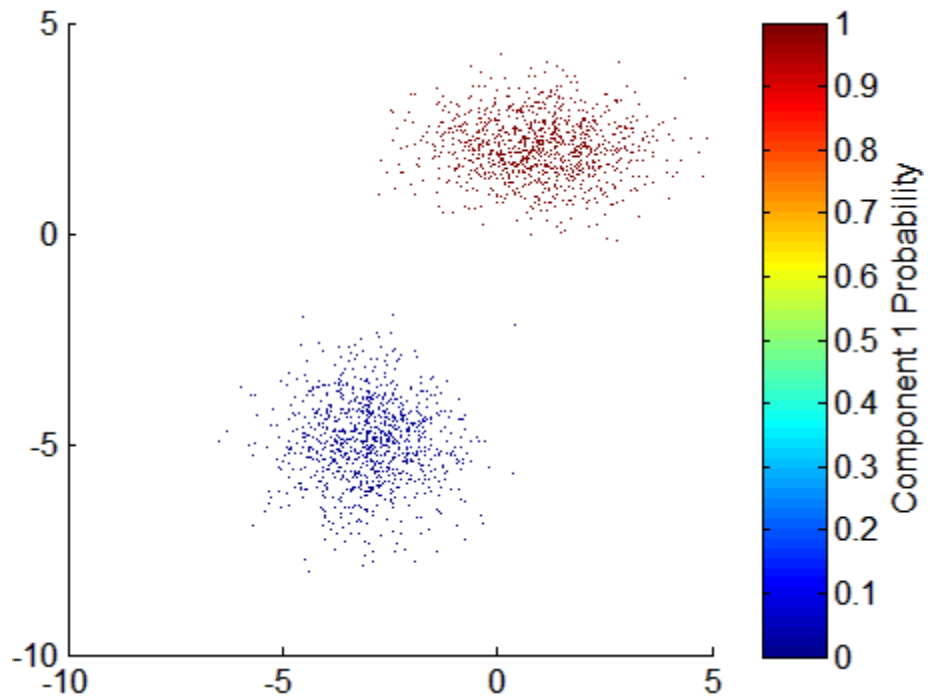
The `mahal` method of the `gmdistribution` class measures the Mahalanobis distance (in squared units) of each observation to the mean of each of the components:

```
D = mahal(obj,X);

figure
delete(h)
scatter(X(:,1),X(:,2),10,D(:,1),'.')
hb = colorbar;
ylabel(hb,'Mahalanobis Distance to Component 1')
```

11
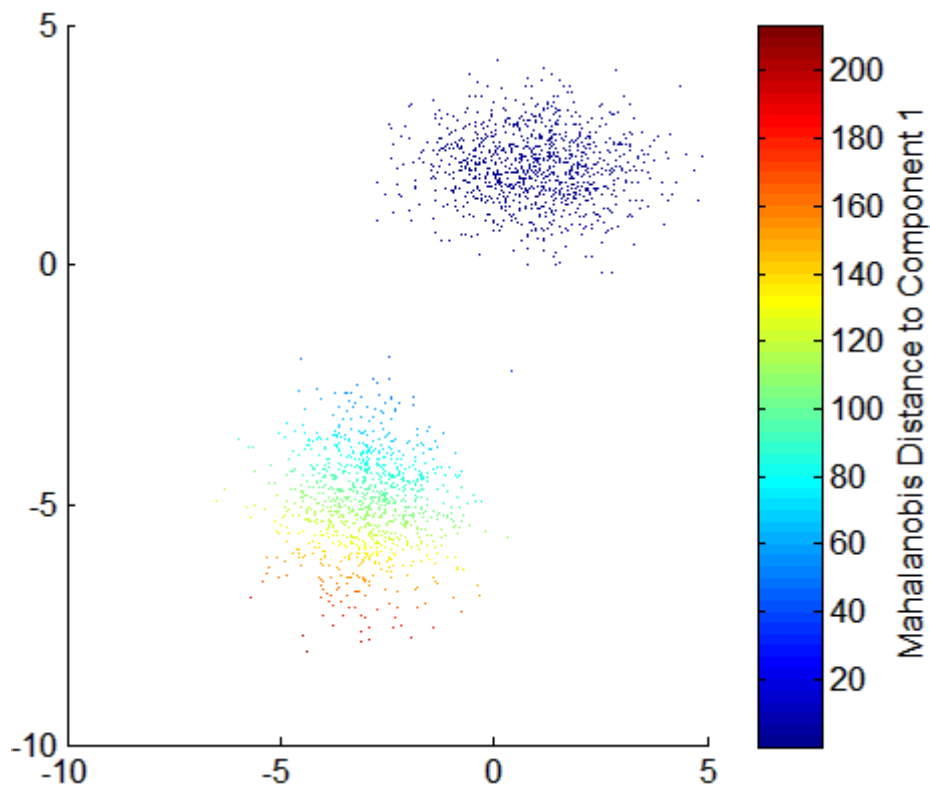
Classification

Introduction

Models of data with a categorical response are called *classifiers*. A classifier is built from *training data*, for which classifications are known. The classifier assigns new *test data* to one of the categorical levels of the response.

Parametric methods, like "Discriminant Analysis" on page 11-3, fit a parametric model to the training data and interpolate to classify test data.

Nonparametric methods, like "Classification Trees" on page 11-9, use other means to determine classifications. In this sense, classification methods are analogous to the methods discussed in "Nonlinear Regression" on page 8-58.

Discriminant Analysis

In this section...
"Introduction" on page 11-3
"Example: Discriminant Analysis" on page 11-3

Introduction

Discriminant analysis uses training data to estimate the parameters of *discriminant functions* of the predictor variables. Discriminant functions determine boundaries in predictor space between various classes. The resulting classifier discriminates among the classes (the categorical levels of the response) based on the predictor data.

The Statistics Toolbox function `classify` performs discriminant analysis.

Example: Discriminant Analysis

1 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')
```
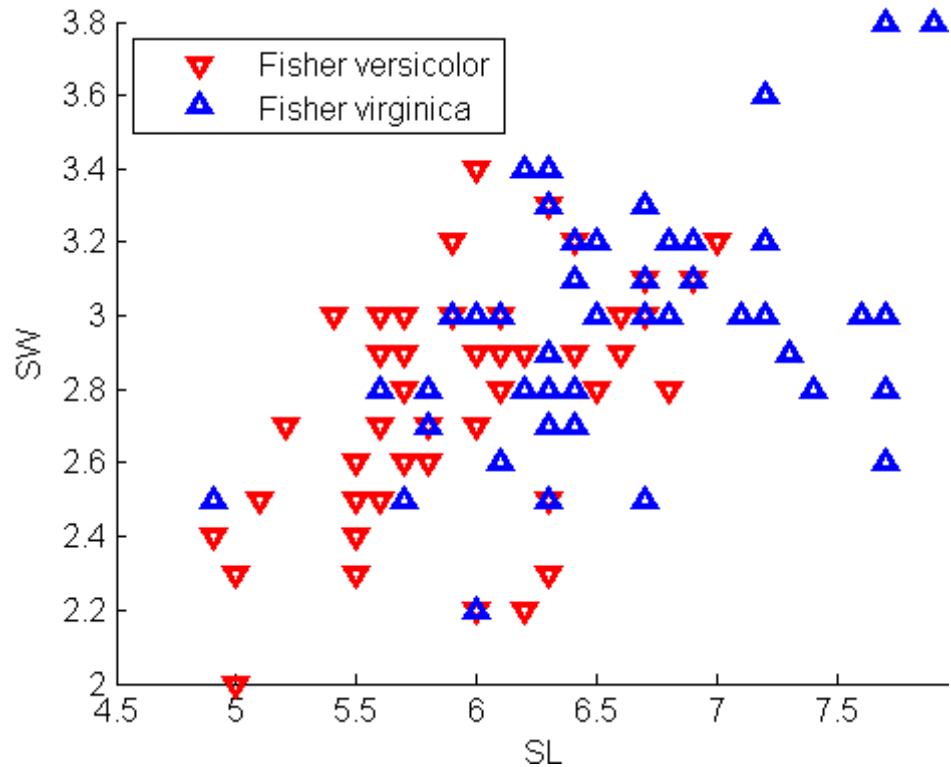
2 Classify a grid of measurements on the same scale, using `classify`:

```
[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');
```

3 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}')
```
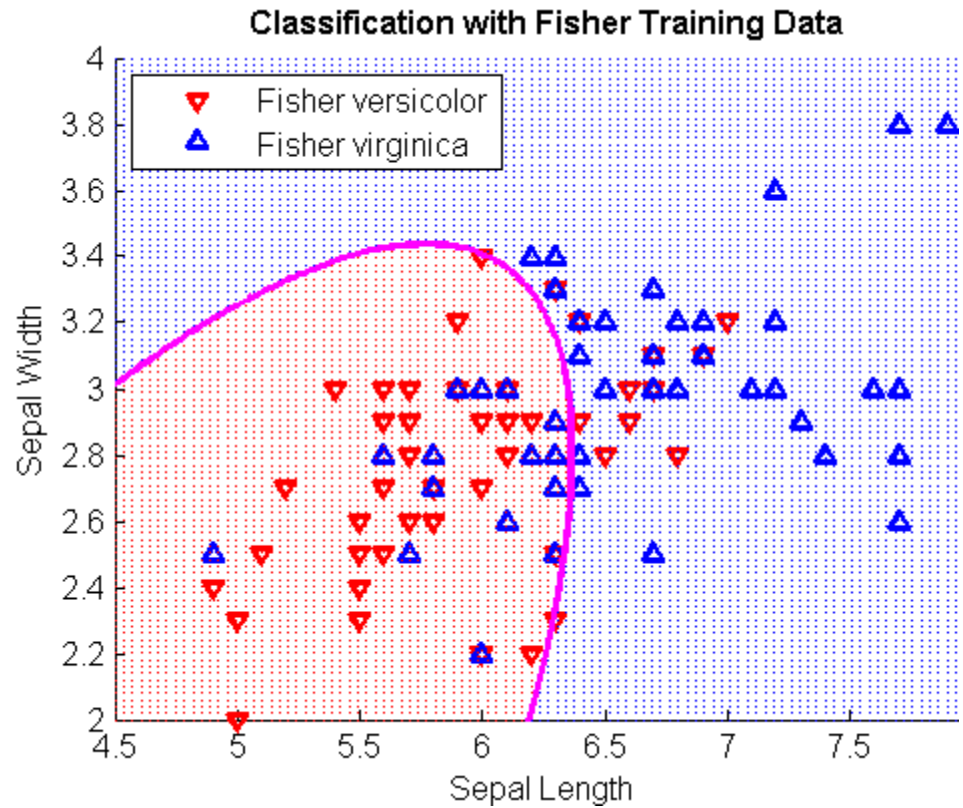
Naive Bayes Classification

The Naive Bayes classifier is designed for use when features are independent of one another within each class, but it appears to work well in practice even when that independence assumption is not valid. It classifies data in two steps:

1 Training step: Using the training samples, the method estimates the parameters of a probability distribution, assuming features are conditionally independent given the class.

2 Prediction step: For any unseen test sample, the method computes the posterior probability of that sample belonging to each class. The method then classifies the test sample according the largest posterior probability.

The class-conditional independence assumption greatly simplifies the training step since you can estimate the one-dimensional class-conditional density for each feature individually. While the class-conditional independence between features is not true in general, research shows that this optimistic assumption works well in practice. This assumption of class independence allows the Naive Bayes classifier to better estimate the parameters required for accurate classification while using less training data than many other classifiers. This makes it particularly effective for datasets containing many predictors or features.

Supported Distributions

Naive Bayes classification is based on estimating `P(X|Y)`, the probability or probability density of features `X` given class `Y`. The Naive Bayes classification object `NaiveBayes` provides support for normal (Gaussian), kernel, multinomial, and multivariate multinomial distributions. It is possible to use different distributions for different features.

Normal (Gaussian) Distribution

The `'normal'` distribution is appropriate for features that have normal distributions in each class. For each feature you model with a normal distribution, the Naive Bayes classifier estimates a separate normal

distribution for each class by computing the mean and standard deviation of the training data in that class. For more information on normal distributions, see "Normal Distribution" on page B-75.

Kernel Distribution

The `'kernel'` distribution is appropriate for features that have a continuous distribution. It does not require a strong assumption such as a normal distribution and you can use it in cases where the distribution of a feature may be skewed or have multiple peaks or modes. It requires more computing time and more memory than the normal distribution. For each feature you model with a kernel distribution, the Naive Bayes classifier computes a separate kernel density estimate for each class based on the training data for that class. By default the kernel is the normal kernel, and the classifier selects a width automatically for each class and feature. It is possible to specify different kernels for each feature, and different widths for each feature or class.

Multinomial Distribution

The multinomial distribution (specify with the `'mn'` keyword) is appropriate when all features represent counts of a set of words or tokens. This is sometimes called the "bag of words" model. For example, an e-mail spam classifier might be based on features that count the number of occurrences of various tokens in an e-mail. One feature might count the number of exclamation points, another might count the number of times the word "money" appears, and another might count the number of times the recipient's name appears. This is a Naive Bayes model under the further assumption that the total number of tokens (or the total document length) is independent of response class.

For the multinomial option, each feature represents the count of one token. The classifier counts the set of relative token probabilities separately for each class. The classifier defines the multinomial distribution for each row by the vector of probabilities for the corresponding class, and by N, the total token count for that row.

Classification is based on the relative frequencies of the tokens. For a row in which no token appears, N is 0 and no classification is possible. This classifier is not appropriate when the total number of tokens provides information about the response class.

Multivariate Multinomial Distribution

The multivariate multinomial distribution (specify with the `'mvmn'` keyword) is appropriate for categorical features. For example, you could fit a feature describing the weather in categories such as rain/sun/snow/clouds using the multivariate multinomial model. The feature categories are sometimes called the feature levels, and differ from the class levels for the response variable.

For each feature you model with a multivariate multinomial distribution, the Naive Bayes classifier computes a separate set of probabilities for the set of feature levels for each class.

Classification Trees

In this section...

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 8-59. 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 are categorical, classification trees are a natural modification.

Note This section demonstrates methods for objects of the `classregtree` class. These methods supersede the functions `treefit`, `treedisp`, `treeval`, `treeprune`, and `treetest`, which are maintained in Statistics Toolbox software only for backwards compatibility.

Example: 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.

1 Load the data and use the `classregtree` constructor of the `classregtree` class 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.

2 Use the `type` method of the `classregtree` class 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.

3 To view the tree, use the `view` method of the `classregtree` class:
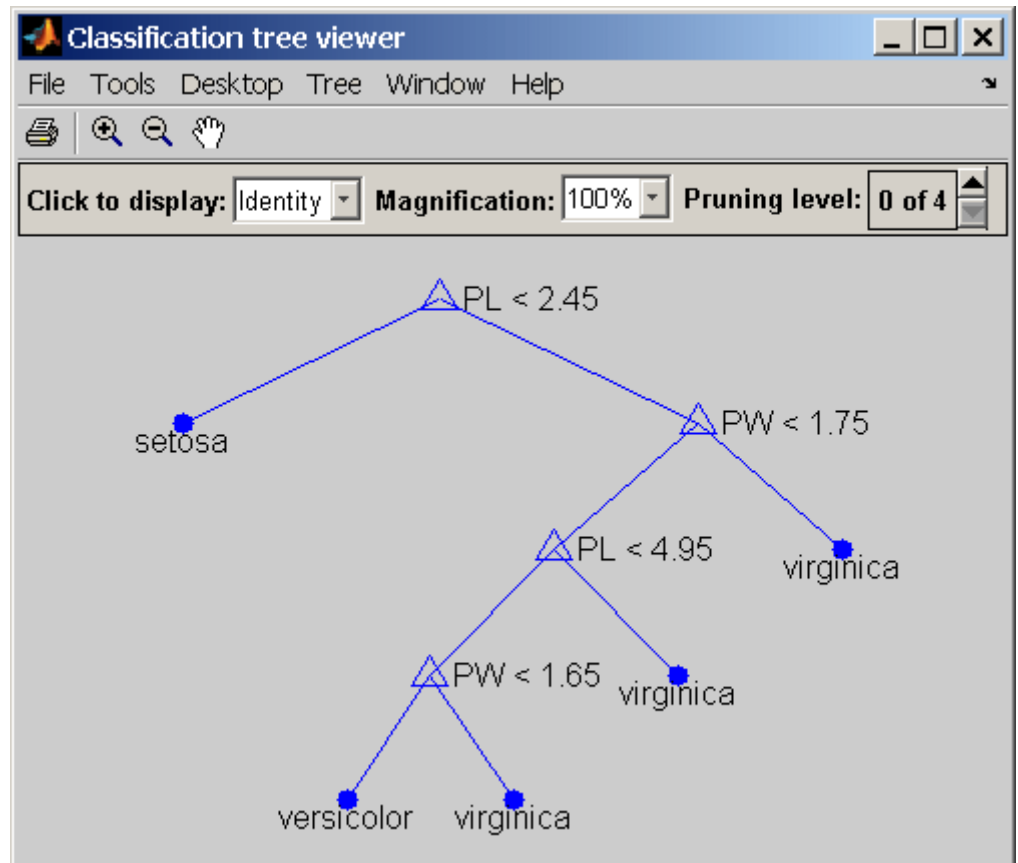
```
view(t)
```

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.

4 The tree does not use sepal measurements for predicting species. These can go unmeasured in new data, and you can enter them 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 of the `classregtree` class. The predicted species is the left leaf node at the bottom of the tree in the previous view.

5 You can use a variety of other methods of the `classregtree` class, 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'
```

6 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 overfitting. You can use the `prune` method of the `classregtree` class 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)
```
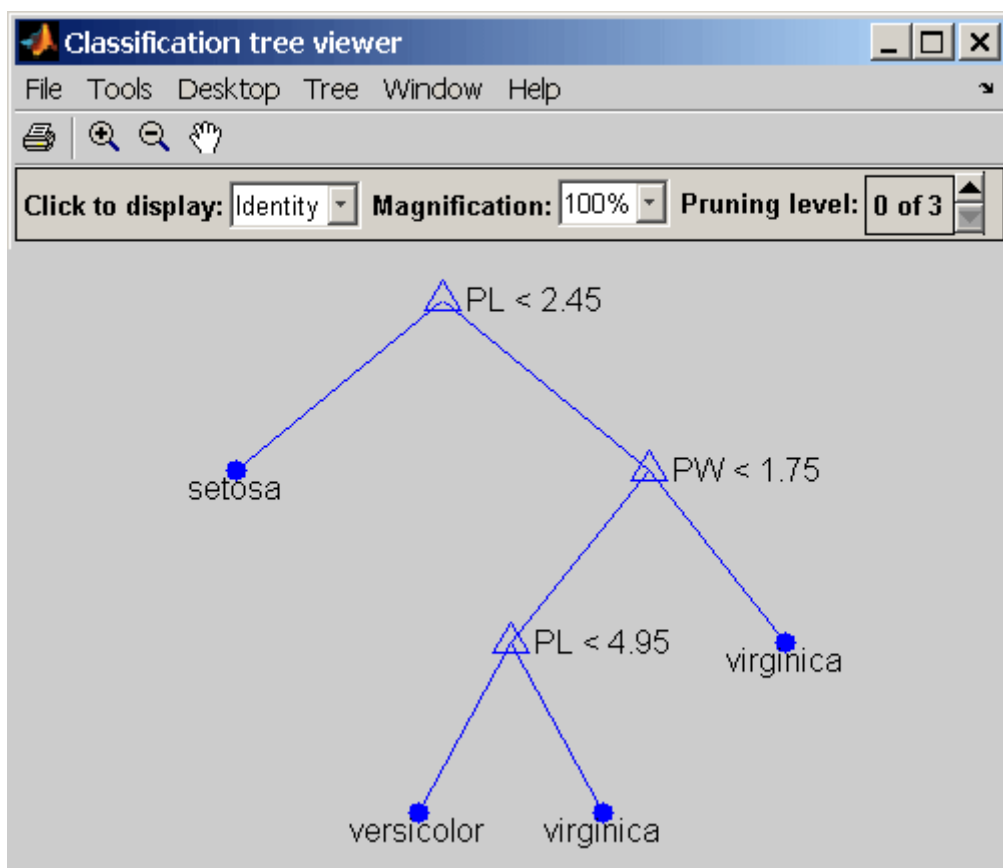
To find the best classification tree, employing the techniques of resubstitution and cross-validation, use the `test` method of the `classregtree` class.

References

[1] Breiman, L., et al., *Classification and Regression Trees*, Chapman & Hall, Boca Raton, 1993.

Regression and Classification by Bagging Decision Trees

Introduction

Bagging, which stands for "bootstrap aggregation", is a type of ensemble learning. To bag a weak learner such as a decision tree on a dataset, generate many bootstrap replicas of this dataset and grow decision trees on these replicas. Obtain each bootstrap replica by randomly selecting N observations out of N with replacement, where N is the dataset size. To find the predicted response of a trained ensemble, take an average over predictions from individual trees.

Bagging works by reducing variance of an unbiased base learner such as a decision tree. The base learner must be unstable; its configuration must vary significantly from one bootstrap replica to another. A decision tree with fine leaves satisfies this requirement.

In the process of training decision trees on bootstrap replicas of input data, you can also randomly select input variables, or features, for each decision split. Instead of attempting splits on all variables at each node of a decision tree, the algorithm only looks at feature subsets of fixed size randomly selected from input variables. For example, if data has 10 input features, you can select 3 features at random for each decision split. If the best split uses one of the omitted seven variables, it is suboptimal. Surprisingly, this technique tends to improve the predictive power of the ensemble, as random selection of features reduces correlation between trees in the ensemble and increases the overall predictive power.

Several features of bagged decision trees make them a unique algorithm. Drawing N out of N observations with replacement omits on average 37% of observations for each decision tree. These are "out-of-bag" observations. You can use them to estimate the predictive power and feature importance. For each observation, you can estimate the out-of-bag prediction by averaging over predictions from all trees in the ensemble for which this observation is out of bag. You can then compare the computed prediction against the true response for this observation. By comparing the out-of-bag predicted responses against the true responses for all observations used for training, you can estimate the average out-of-bag error. This out-of-bag average is an unbiased estimator of the true ensemble error. You can also obtain out-of-bag estimates of feature importance by randomly permuting out-of-bag data across one variable or column at a time and estimating the increase in the out-of-bag error due to this permutation. The larger the increase, the more important the feature. Thus, you do not need to supply test data for bagged ensembles because you obtain reliable estimates of the predictive power and feature importance in the process of training, which is an attractive feature of bagging.

Another attractive feature of bagged decision trees is the proximity matrix. Every time two observations land on the same leaf of a tree, their proximity increases by one. For normalization, sum these proximities over all trees in the ensemble and divided by the number of trees. The resulting matrix is symmetric with diagonal elements equal to 1 and off-diagonal elements ranging from 0 to 1. You can use this matrix for finding outlier observations and discovering clusters in the data through multidimensional scaling.

The following examples showcase the Statistics Toolbox `TreeBagger class` and `CompactTreeBagger class` functionalities.

Examples

The following examples show how to use ensembles of decision trees for regression and classification.

Regression of Insurance Risk Rating for Car Imports

In this example, use a database of 1985 car imports with 205 observations, 25 input variables, and one response variable, insurance risk rating, or "symboling". The first 15 variables are numeric and the last 10 are categorical. The symboling index takes integer values from -3 to 3.

First, load the dataset and split it into predictor and response arrays:

```
load imports-85;
Y = X(:,1);
X = X(:,2:end);
```

Because bagging uses randomized data drawings, its exact outcome depends on the initial random seed. To reproduce the exact results in this example, use the random stream settings

```
s = RandStream('mt19937ar','seed',1945);
RandStream.setDefaultStream(s);
```

Finding the Optimal Leaf Size

For regression, the general rule is to set leaf size to 5 and select one third of input features for decision splits at random. In the following step, verify the optimal leaf size by comparing mean squared errors obtained by regression for various leaf sizes. oobError computes MSE versus the number of grown trees. It is necessary to set oobpred to 'on' to obtain out-of-bag predictions later.

```
leaf = [1 5 10 20 50 100];
col = 'rgbcmy';
figure(1);
for i=1:length(leaf)
    b = TreeBagger(50,X,Y,'method','r','oobpred','on',...
    'cat',16:25,'minleaf',leaf(i));
    plot(oobError(b),col(i));
    hold on;
end
xlabel('Number of Grown Trees');
ylabel('Mean Squared Error');
legend({'1' '5' '10' '20' '50' '100'},'Location','NorthEast');
hold off;
```
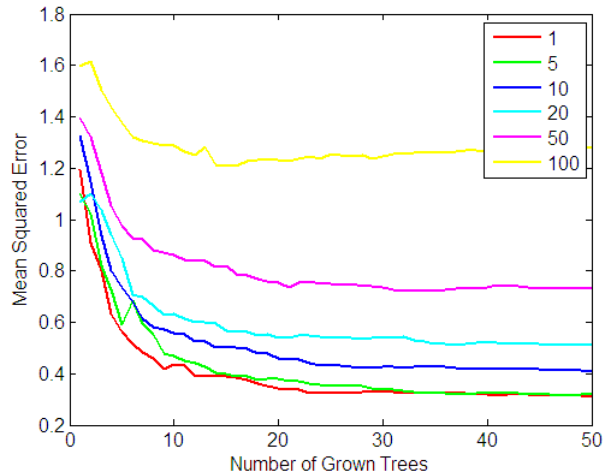
The green (leaf size 5) and red (leaf size 1) curves are competitive and give the lowest MSE values. Leaf size 5 is preferable because trees are more shallow in this case. Training and computing predicted response with shallow trees takes less CPU time and memory.

Estimating Feature Importance

In practical applications, you typically grow ensembles with hundreds of trees. Only 50 trees were used in the previous exercise for faster processing. Now that you have estimated the optimal leaf size, grow a larger ensemble with 100 trees and use it for estimation of feature importance:

```
b = TreeBagger(100,X,Y,'method','r','oobvarimp','on',...
'cat',16:25,'minleaf',5);
```

Inspect the error curve again to make sure nothing went wrong during training:

```
figure(2);
plot(oobError(b));
xlabel('Number of Grown Trees');
ylabel('Out-of-Bag Mean Squared Error');
```
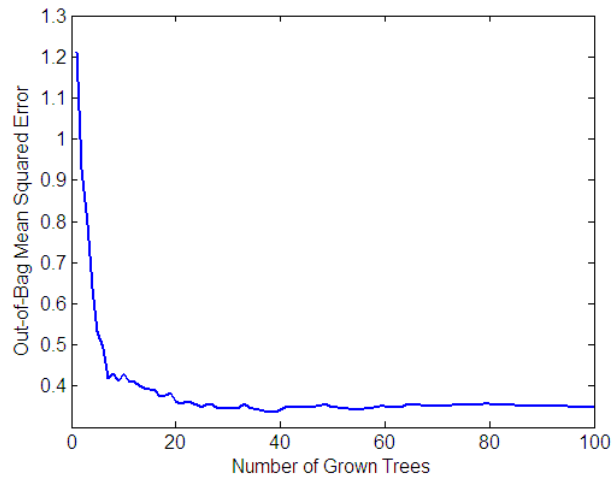
The method trains ensembles with few trees on observations that are in bag for all trees. For such observations, it is not possible to compute the true out-of-bag prediction and `TreeBagger` returns the most probable class for classification and the sample mean for regression. You can change the default value returned for in-bag observations using the `DefaultYfit` property. If you set the default value to `NaN` for regression, the method excludes in-bag observations from computation of the out-of-bag error. In this case, the curve is more variable when the number of trees is small, either because some observations are never out of bag (and are therefore excluded) or because their predictions are based on few trees.

```
b.DefaultYfit = NaN;
figure(3);
plot(oobError(b));
xlabel('Number of Grown Trees');
ylabel('Out-of-Bag Error Excluding in-Bag Observations');
```
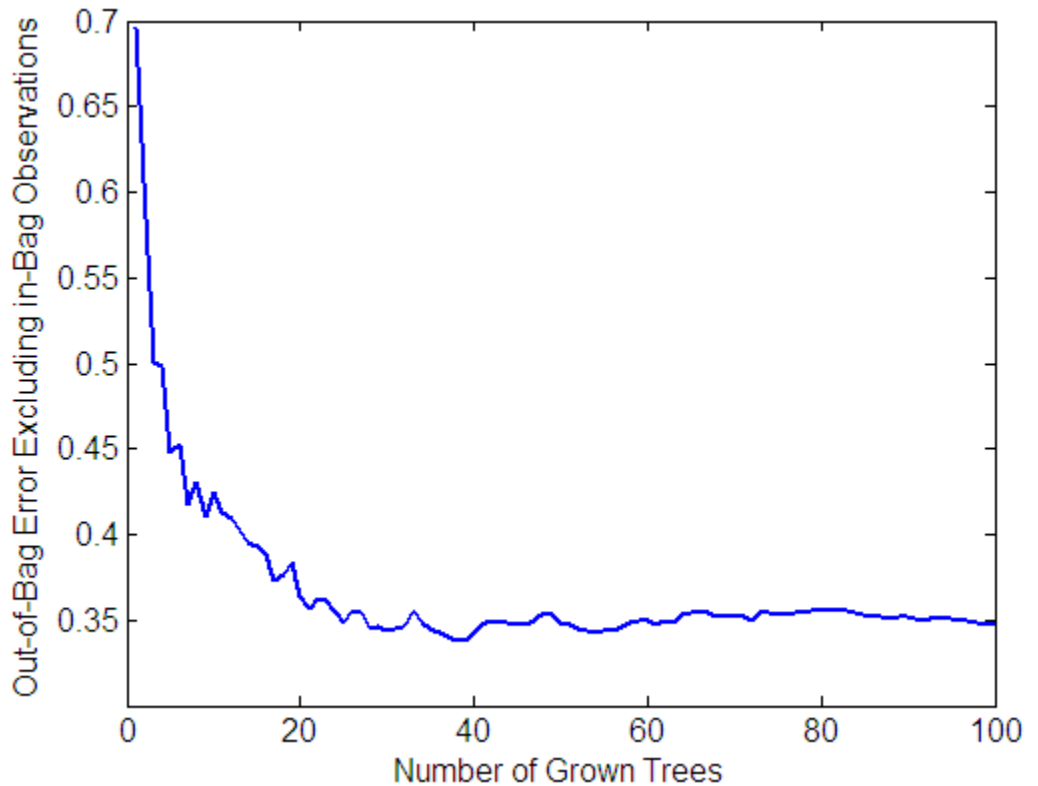
The `OOBIndices` property of `TreeBagger` keeps track of which observations are out of bag for what trees. Using this property, you can monitor the fraction of observations in the training data that are in bag for all trees. The curve starts at approximately 2/3, the fraction of unique observations selected by one bootstrap replica, and goes down to zero at approximately 10 trees.

```
finbag = zeros(1,b.NTrees);
for t=1:b.NTrees
    finbag(t) = sum(all(~b.OOBIndices(:,1:t),2));
end
finbag = finbag / size(X,1);
figure(4);
plot(finbag);
xlabel('Number of Grown Trees');
ylabel('Fraction of in-Bag Observations');
```
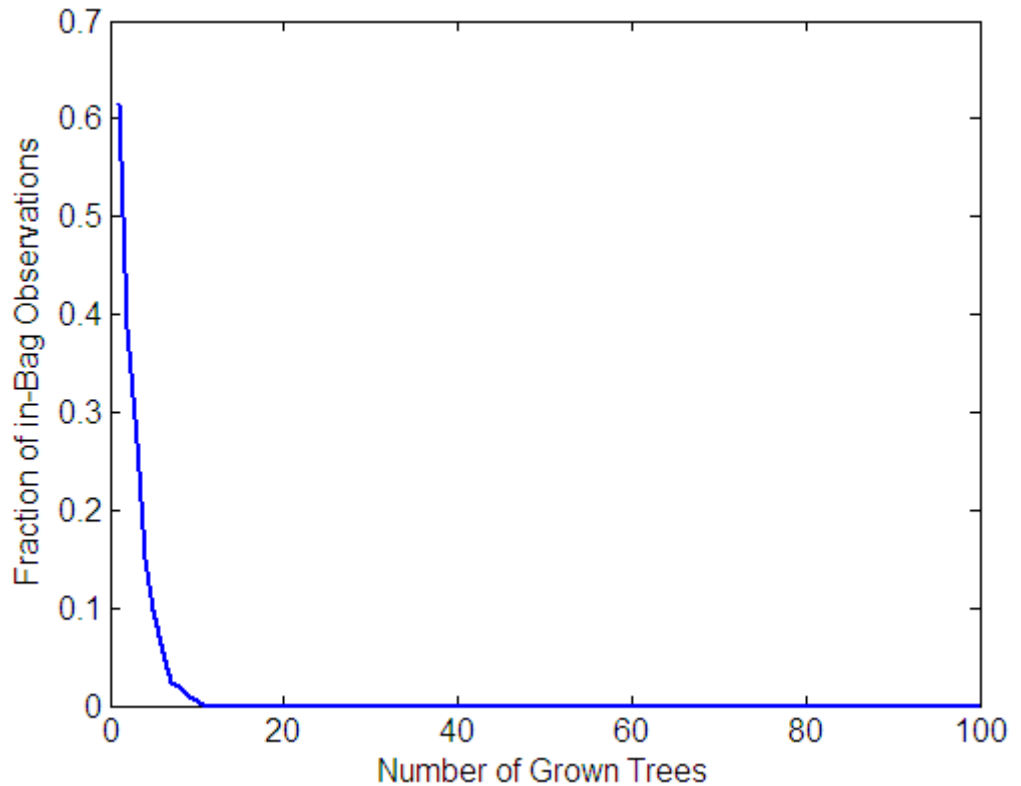
Prediction ability should depend more on important features and less on unimportant features. You can use this idea to measure feature importance.

For each feature, you can permute the values of this feature across all of the observations in the data set and measure how much worse the mean squared error (MSE) becomes after the permutation. You can repeat this for each feature.

Using the following code, plot the increase in MSE due to permuting out-of-bag observations across each input variable. The OOBPermutedVarDeltaError array stores the increase in MSE averaged over all trees in the ensemble and divided by the standard deviation taken over the trees, for each variable. The larger this value, the more important the variable. Imposing an arbitrary cutoff at 0.6, you can select the four most important features.

```
figure(5);
bar(b.OOBPermutedVarDeltaError);
xlabel('Feature Number');
ylabel('Out-Of-Bag Feature Importance');
idxvar = find(b.OOBPermutedVarDeltaError>0.6)
```

returns

```
idxvar =

      1     2    16    19
```
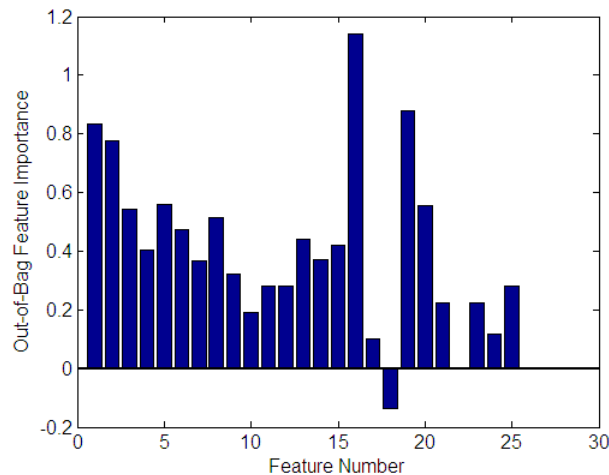

Growing Trees on a Reduced Set of Features

Using just the four most powerful features selected in "Estimating Feature Importance" on page 11-17, find out if it is possible to obtain a similar predictive power. To begin, grow 100 trees on these features only. The first two of the four selected features are numeric and the last two are categorical.

```
b4v = TreeBagger(100,X(:,idxvar),Y,'method','r',...
'oobvarimp','on','cat',3:4,'minleaf',5);
figure(6);
plot(oobError(b4v));
xlabel('Number of Grown Trees');
ylabel('Out-of-Bag Mean Squared Error');
```

```
figure(7);
bar(b4v.OOBPermutedVarDeltaError);
xlabel('Feature Index');
ylabel('Out-of-Bag Feature Importance');
```
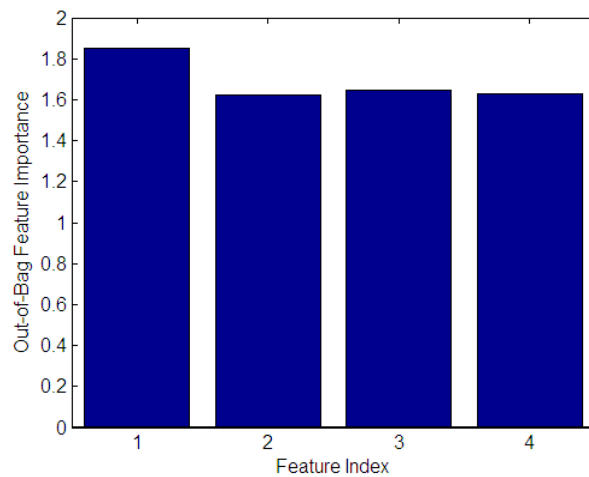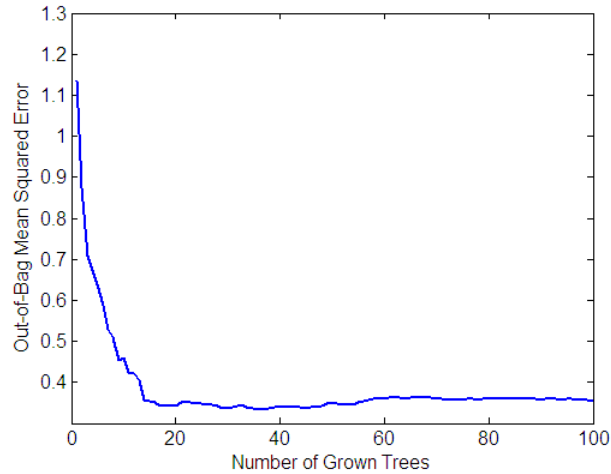
These four most powerful features give the same MSE as the full set, and the ensemble trained on the reduced set ranks these features similarly to each other.

Finding Outliers

To find outliers in the training data, compute the proximity matrix using `fillProximities`:

```
b4v = fillProximities(b4v);
```

The method normalizes this measure by subtracting the mean outlier measure for the entire sample, taking the magnitude of this difference and dividing the result by the median absolute deviation for the entire sample.

```
figure(8);
hist(b4v.OutlierMeasure);
xlabel('Outlier Measure');
ylabel('Number of Observations');
```
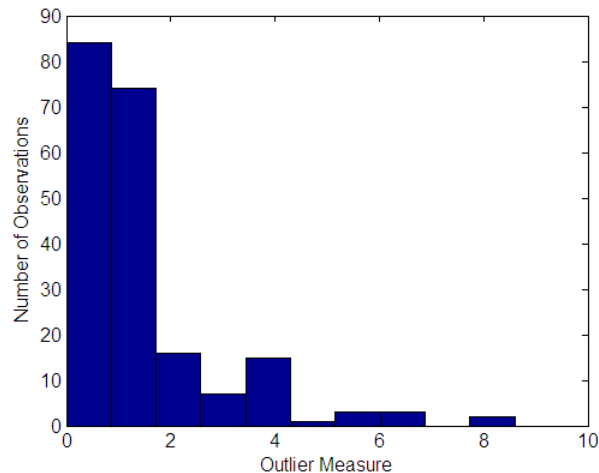

Discovering Clusters in the Data

By applying multidimensional scaling to the computed matrix of proximities, you can inspect the structure of the input data and look for possible clusters of observations. The `mdsProx` method returns scaled coordinates and eigenvalues for the computed proximity matrix. If run with the `colors` option, this method makes a scatter plot of two scaled coordinates, first and second by default.

```
figure(9);
```

```
[s,e] = mdsProx(b4v,'colors','k');
xlabel('1st Scaled Coordinate');
ylabel('2nd Scaled Coordinate');
```
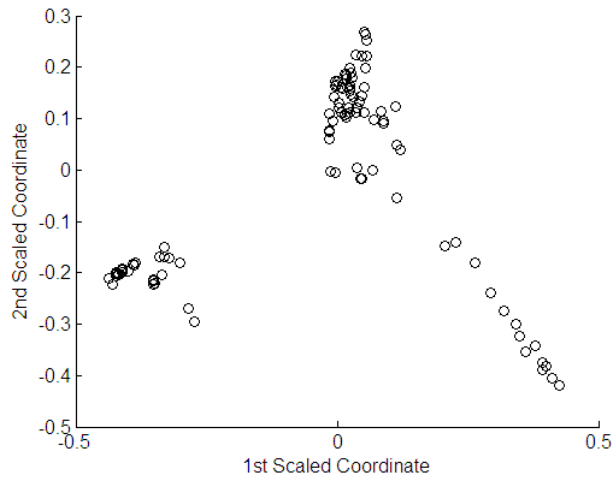

Assess the relative importance of the scaled axes by plotting the first 20 eigenvalues:

```
figure(10);
bar(e(1:20));
xlabel('Scaled Coordinate Index');
ylabel('Eigenvalue');
```
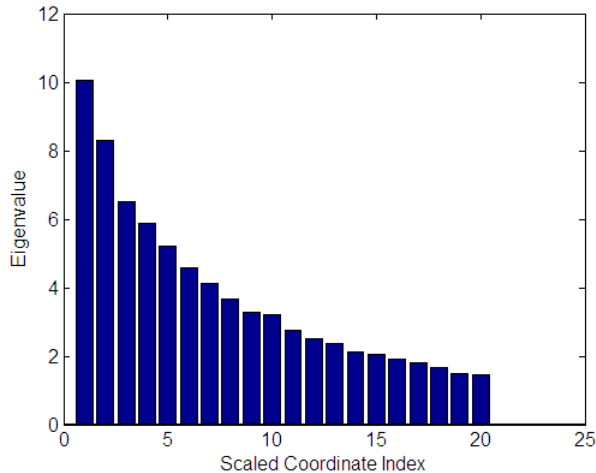
Take a closer look at the cluster of observations in the lower-right corner. You should use data stored in the ensemble instead of the original input data for this exercise because the two datasets may not match.

```
idxclus = s(:,1)>0.2;
any(~isnan(b4v.X(idxclus,1)));
```

All values of the first variable are `NaN`. You can see that the cluster at the lower right is a cluster of observations with missing values of normalized loss. In fact, the predictive power of the ensemble for these observations should be considerably worse than for the rest. To verify this, you can compute out-of-bag predictions for all observations and then evaluate MSE separately for observations with and without missing values of normalized loss:

```
Yfit = oobPredict(b4v);
mean((Yfit(idxclus)-Y(idxclus)).^2)
mean((Yfit(~idxclus)-Y(~idxclus)).^2)

ans =

    0.7046


ans =

    0.2870
```

Saving the Ensemble Configuration for Future Use

To use the trained ensemble for predicting the response on unseen data, store the ensemble to disk and retrieve it later. If you do not want to compute predictions for out-of-bag data or reuse training data in any other way, there is no need to store the ensemble object itself. Saving the compact version of the ensemble would be enough in this case. Extract the compact object from the ensemble:

```
c = compact(b4v)

c =

Ensemble with 100 decision trees:
Method:           regression
Nvars:                   4
```

This object can be now saved into a *.mat file as usual.

Classifying Radar Returns for Ionosphere Data

You can also use ensembles of decision trees for classification. For this example, use ionosphere data with 351 observations and 34 real-valued predictors. The response variable is categorical with two levels:

- `'g'` for good radar returns
- `'b'` for bad radar returns

The goal is to predict good or bad using a set of 34 measurements.

The workflow is similar to the one for regression "Regression of Insurance Risk Rating for Car Imports" on page 11-15. Again, fix the initial random seed, grow 50 trees, inspect how the ensemble error changes with accumulation of trees, and estimate feature importance. For classification, it is best to set the minimal leaf size to 1 and select thesquare root of the total number of features for each decision split at random. These are the default settings for a `TreeBagger` used for classification.

```
load ionosphere;
s = RandStream('mt19937ar','seed',1945);
RandStream.setDefaultStream(s);
```

```
b = TreeBagger(50,X,Y,'oobvarimp','on');
figure(11);
plot(oobError(b));
xlabel('Number of Grown Trees');
ylabel('Out-of-Bag Classification Error');
figure(12);
bar(b.OOBPermutedVarDeltaError);
xlabel('Feature Index');
ylabel('Out-of-Bag Feature Importance');
idxvar = find(b.OOBPermutedVarDeltaError>0.8)
```

returns

```
idxvar =
     3      5      7     14     27
```
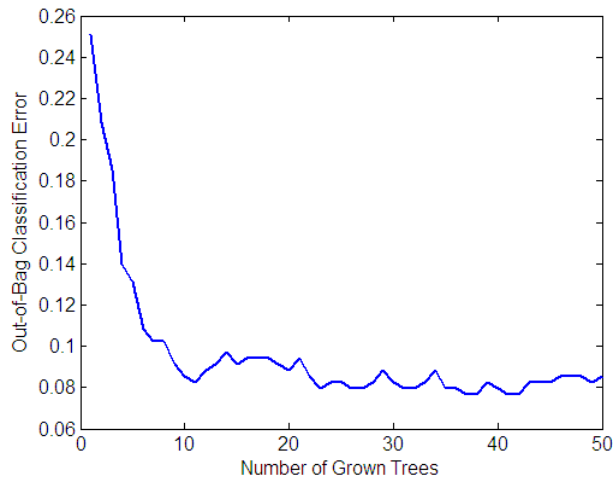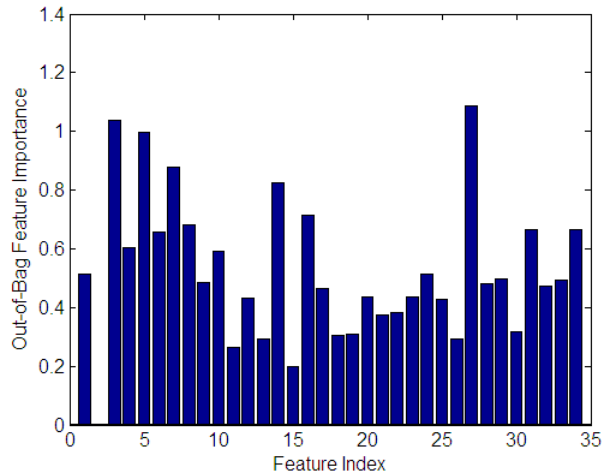
Having selected the five most important features, grow a larger ensemble on the reduced feature set. Save time by not permuting out-of-bag observations to obtain new estimates of feature importance for the reduced feature set (set `oobvarimp` to `'off'`). You would still be interested in obtaining out-of-bag estimates of classification error (set `oobpred` to `'on'`).

```
b5v = TreeBagger(100,X(:,idxvar),Y,'oobpred','on');
figure(13);
plot(oobError(b5v));
xlabel('Number of Grown Trees');
ylabel('Out-of-Bag Classification Error');
```
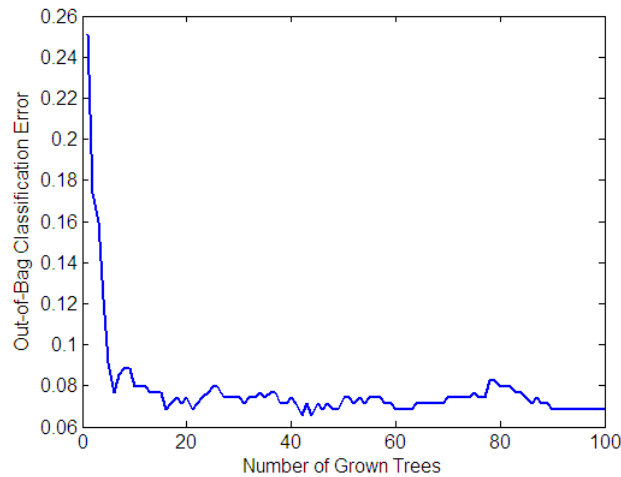
For classification ensembles, in addition to classification error (fraction of misclassified observations), you can also monitor the average classification margin. For each observation, the *margin* is defined as the difference between the score for the true class and the maximal score for other classes predicted by this tree. The cumulative classification margin uses the scores averaged over all trees and the mean cumulative classification margin is the cumulative margin averaged over all observations. The `oobMeanMargin` method with the `'mode'` argument set to `'cumulative'` (default) shows how the mean cumulative margin changes as the ensemble grows: every new element in the returned array represents the cumulative margin obtained by including a new tree in the ensemble. If training is successful, you would expect to see a gradual increase in the mean classification margin.

For decision trees, a classification score is the probability of observing an instance of this class in this tree leaf. For example, if the leaf of a grown decision tree has five `'good'` and three `'bad'` training observations in it, the scores returned by this decision tree for any observation fallen on this leaf are 5/8 for the `'good'` class and 3/8 for the `'bad'` class. These probabilities are referred to as `'scores'` for consistency with other classifiers that may not have an obvious interpretation for numeric values of returned predictions.

```
figure(14);
plot(oobMeanMargin(b5v));
xlabel('Number of Grown Trees');
```

```
ylabel('Out-of-Bag Mean Classification Margin');
```
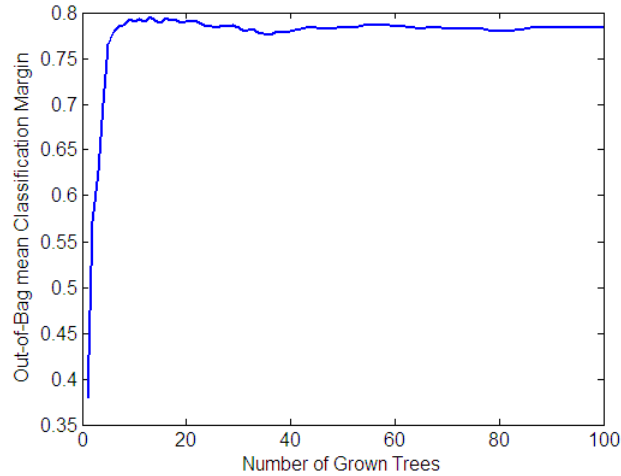

Again, compute the matrix of proximities and look at the distribution of outlier measures. Unlike regression, outlier measures for classification ensembles are computed within each class separately.

```
b5v = fillProximities(b5v);
figure(15);
hist(b5v.OutlierMeasure);
xlabel('Outlier Measure');
ylabel('Number of Observations');
```
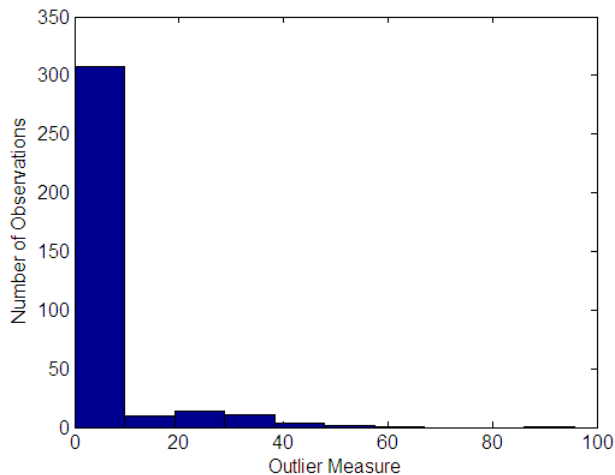
All extreme outliers for this dataset come from the `'good'` class:

```
b5v.Y(b5v.OutlierMeasure>50)

ans =

    'g'
    'g'
    'g'
    'g'
```

Just like for regression, you can plot scaled coordinates, displaying the two classes in different colors using the `colors` argument of `mdsProx`. This argument takes a string in which every character represents a color. To find out the order of classes used by the ensemble, look at the `ClassNames` property:

```
b5v.ClassNames

ans =

    'g'
    'b'
```

The 'good' class is first and the 'bad' class is second. Display scaled coordinates using red for 'good' and blue for 'bad' observations.

```
figure(16);
[s,e] = mdsProx(b5v,'colors','rb');
xlabel('1st Scaled Coordinate');
ylabel('2nd Scaled Coordinate');
legend({'good' 'bad'},'Location','NorthWest');
```
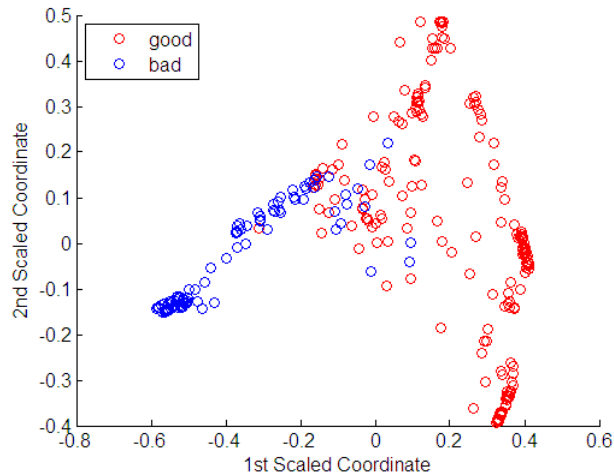

Again, plot the first 20 eigenvalues obtained by scaling. The first eigenvalue in this case clearly dominates and the first scaled coordinate is most important.

```
figure(17);
bar(e(1:20));
xlabel('Scaled Coordinate Index');
ylabel('Eigenvalue');
```
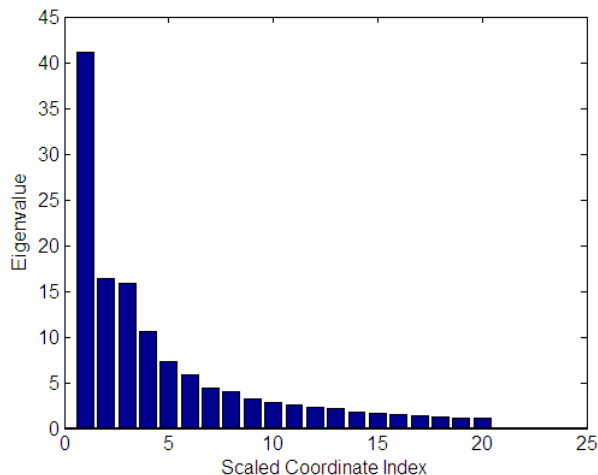
Plotting a Performance Curve

Another way of exploring the performance of a classification ensemble is to plot its Receiver Operating Characteristic (ROC) curve or another performance curve suitable for the current problem. First, you need to obtain predictions for out-of-bag observations. For a classification ensemble, the `oobPredict` method returns a cell array of classification labels (`'g'` or `'b'` for ionosphere data) as the first output argument and a numeric array of scores as the second output argument. The returned array of scores has two columns, one for each class. In this case, the first column is for the `'good'` class and the second column is for the `'bad'` class. One of the columns in the score matrix is redundant because the scores represent class probabilities in tree leaves and by definition add up to 1.

```
[Yfit,Sfit] = oobPredict(b5v);
```

You can now use the `perfcurve` utility (described in more detail in "Performance Curves" on page 11-37) to compute a performance curve. By default, `perfcurve` returns the standard ROC curve, which is the true positive rate versus false positive rate. `perfcurve` requires true class labels, scores, and the positive class label for input. In this case, choose the `'good'` class as positive. The scores for this class are in the first column of `Sfit`.

```
[fpr,tpr] = perfcurve(b5v.Y,Sfit(:,1),'g');
figure(18);
```

```
plot(fpr,tpr);
xlabel('False Positive Rate');
ylabel('True Positive Rate');
```
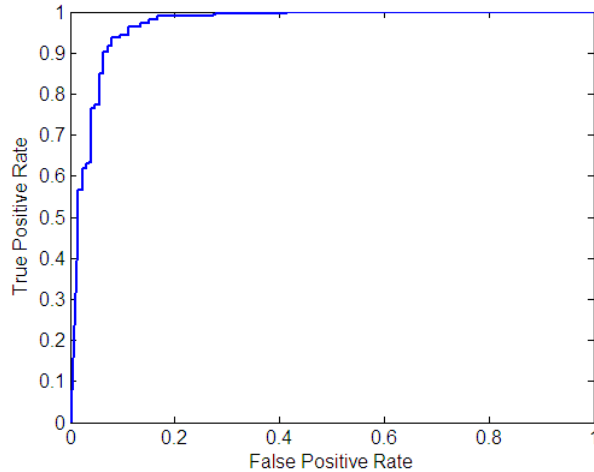

Instead of the standard ROC curve, you might want to plot, for example, ensemble accuracy versus threshold on the score for the 'good' class. The ycrit input argument of perfcurve lets you specify the criterion for the y-axis, and the third output argument of perfcurve returns an array of thresholds for the positive class score. Accuracy is the fraction of correctly classified observations, or equivalently, one minus classification error.

```
[fpr,accu,thre] = perfcurve(b5v.Y,Sfit(:,1),'g','ycrit','accu');
figure(19);
plot(thre,accu);
xlabel('Threshold for ''good'' Returns');
ylabel('Classification Accuracy');
```
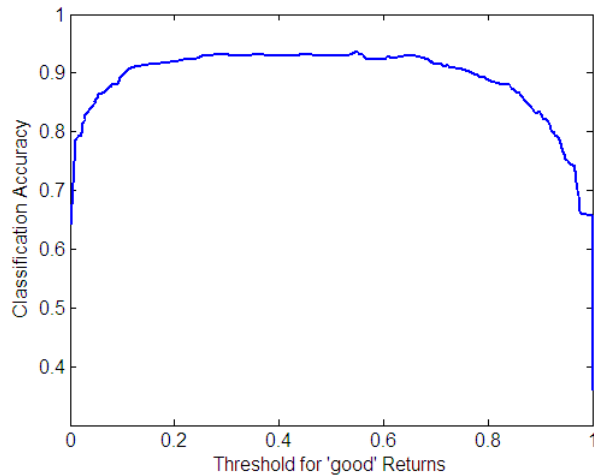
The curve shows a flat region indicating that any threshold from 0.2 to 0.6 is a reasonable choice. By default, the function assigns classification labels using 0.5 as the boundary between the two classes. You can find exactly what accuracy this corresponds to:

```
i50 = find(accu>=0.50,1,'first')
accu(abs(thre-0.5)<eps)
```

returns

```
ans =

    0.9288
```

The maximal accuracy is a little higher than the default one.

```
[maxaccu,iaccu] = max(accu)
```

returns

```
maxaccu =

    0.9373

iaccu =
```

```
    103
```

The optimal threshold is therefore:

```
thre(iaccu)

ans =

    0.5500
```

See Also

bootstrp, classify, ClassNames, CompactTreeBagger, fillProximities, perfcurve, randstream, TreeBagger, oobError, OOBPermutedVarDeltaError, perfcurve

Performance Curves

Introduction

After a classification algorithm such as `NaiveBayes` or `TreeBagger` has trained on data, you may want to examine the performance of this algorithm on a specific test dataset. One common way of doing this would be to compute a gross measure of performance such as quadratic loss, accuracy, such as quadratic loss or accuracy, averaged over the entire test dataset.

What are ROC Curves?

You may want to inspect the classifier performance more closely, for example, by plotting a Receiver Operating Characteristic (ROC) curve. By definition, a ROC curve [1,2] shows true positive rate versus false positive rate (equivalently, sensitivity versus 1–specificity) for different thresholds of the classifier output. You can use it, for example, to find the threshold that maximizes the classification accuracy or to assess, in more broad terms, how the classifier performs in the regions of high sensitivity and high specificity.

Evaluating Classifier Performance Using perfcurve

`perfcurve` computes measures for a plot of classifier performance. You can use this utility to evaluate classifier performance on test data after you train the classifier. Various measures such as mean squared error, classification error, or exponential loss can summarize the predictive power of a classifier in a single number. However, a performance curve offers more information as it lets you explore the classifier performance across a range of thresholds on its output.

You can use `perfcurve` with any classifier or, more broadly, with any method that returns a numeric score for an instance of input data. By convention adopted here,

- A high score returned by a classifier for any given instance signifies that the instance is likely from the positive class.

- A low score signifies that the instance is likely from the negative classes.

For some classifiers, you can interpret the score as the posterior probability of observing an instance of the positive class at point X. An example of such a score is the fraction of positive observations in a leaf of a decision tree. In this case, scores fall into the range from 0 to 1 and scores from positive and negative classes add up to unity. Other methods can return scores ranging between minus and plus infinity, without any obvious mapping from the score to the posterior class probability.

`perfcurve` does not impose any requirements on the input score range. Because of this lack of normalization, you can use `perfcurve` to process scores returned by any classification, regression, or fit method. `perfcurve` does not make any assumptions about the nature of input scores or relationships between the scores for different classes. As an example, consider a problem with three classes, A, B, and C, and assume that the scores returned by some classifier for two instances are as follows:

	A	B	C
instance 1	0.4	0.5	0.1
instance 2	0.4	0.1	0.5

If you want to compute a performance curve for separation of classes A and B, with C ignored, you need to address the ambiguity in selecting A over B. You could opt to use the score ratio, `s(A)/s(B)`, or score difference, `s(A)-s(B)`; this choice could depend on the nature of these scores and their normalization. `perfcurve` always takes one score per instance. If you only supply scores for class A, `perfcurve` does not distinguish between observations 1 and 2. The performance curve in this case may not be optimal.

`perfcurve` is intended for use with classifiers that return scores, not those that return only predicted classes. As a counter-example, consider a decision

tree that returns only hard classification labels, 0 or 1, for data with two classes. In this case, the performance curve reduces to a single point because classified instances can be split into positive and negative categories in one way only.

For input, `perfcurve` takes true class labels for some data and scores assigned by a classifier to these data. By default, this utility computes a Receiver Operating Characteristic (ROC) curve and returns values of 1–specificity, or false positive rate, for X and sensitivity, or true positive rate, for Y. You can choose other criteria for X and Y by selecting one out of several provided criteria or specifying an arbitrary criterion through an anonymous function. You can display the computed performance curve using `plot(X,Y)`.

`perfcurve` can compute values for various criteria to plot either on the *x*- or the *y*-axis. All such criteria are described by a 2-by-2 confusion matrix, a 2-by-2 cost matrix, and a 2-by-1 vector of scales applied to class counts.

The `confusion` matrix, `C`, is defined as

$$\begin{pmatrix} TP & FN \\ FP & TN \end{pmatrix},$$

where

- `P` stands for "positive".
- `N` stands for "negative".
- `T` stands for "true".
- `F` stands for "false".

For example, the first row of the confusion matrix defines how the classifier identifies instances of the positive class: `C(1,1)` is the count of correctly identified positive instances and `C(1,2)` is the count of positive instances misidentified as negative.

The cost matrix defines the cost of misclassification for each category:

$$\begin{pmatrix} Cost(P\,|\,P) & Cost(N\,|\,P) \\ Cost(P\,|\,N) & Cost(N\,|\,N) \end{pmatrix},$$

where `Cost(I|J)` is the cost of assigning an instance of class `J` to class `I`. Usually `Cost(I|J)=0` for `I=J`. For flexibility, `perfcurve` allows you to specify nonzero costs for correct classification as well.

The two scales include prior information about class probabilities. `perfcurve` computes these scales by taking `scale(P)=prior(P)*N` and `scale(N)=prioer(N)*P` and normalizing the sum `scale(P)+scale(N)` to 1. `P=TP+FN` and `N=TN+FP` are the total instance counts in the positive and negative class, respectively. The function then applies the scales as multiplicative factors to the counts from the corresponding class: `perfcurve` multiplies counts from the positive class by `scale(P)` and counts from the negative class by `scale(N)`. Consider, for example, computation of positive predictive value, `PPV = TP/(TP+FP)`. TP counts come from the positive class and FP counts come from the negative class. Therefore, you need to scale TP by `scale(P)` and FP by `scale(N)`, and the modified formula for PPV with prior probabilities taken into account is now:

$$PPV = \frac{scale(P)*TP}{scale(P)*TP + scale(N)*FP}$$

If all scores in the data are above a certain threshold, `perfcurve` classifies all instances as `'positive'`. This means that TP is the total number of instances in the positive class and FP is the total number of instances in the negative class. In this case, PPV is simply given by the prior:

$$PPV = \frac{prior(P)}{prior(P) + prior(N)}$$

The `perfcurve` function returns two vectors, X and Y, of performance measures. Each measure is some function of `confusion`, `cost`, and `scale` values. You can request specific measures by name or provide a function handle to compute a custom measure. The function you provide should take `confusion`, `cost`, and `scale` as its three inputs and return a vector of output values.

The criterion for X must be a monotone function of the positive classification count, or equivalently, threshold for the supplied scores. If perfcurve cannot perform a one-to-one mapping between values of the X criterion and score thresholds, it exits with an error message.

By default, perfcurve computes values of the X and Y criteria for all possible score thresholds. Alternatively, it can compute a reduced number of specific X values supplied as an input argument. In either case, for M requested values, perfcurve computes M+1 values for X and Y. The first value out of these M+1 values is special. perfcurve computes it by setting the TP instance count to zero and setting TN to the total count in the negative class. This value corresponds to the 'reject all' threshold. On a standard ROC curve, this translates into an extra point placed at (0,0).

If there are NaN values among input scores, perfcurve can process them in either of two ways:

- It can discard rows with NaN scores.
- It can add them to false classification counts in the respective class.

That is, for any threshold, instances with NaN scores from the positive class are counted as false negative (FN), and instances with NaN scores from the negative class are counted as false positive (FP). In this case, the first value of X or Y is computed by setting TP to zero and setting TN to the total count minus the NaN count in the negative class. For illustration, consider an example with two rows in the positive and two rows in the negative class, each pair having a NaN score:

Class	Score
Negative	0.2
Negative	NaN
Positive	0.7
Positive	NaN

If you discard rows with NaN scores, then as the score cutoff varies, perfcurve computes performance measures as in the following table. For example, a cutoff of 0.5 corresponds to the middle row where rows 1 and 3 are classified correctly, and rows 2 and 4 are omitted.

TP	FN	FP	TN
0	1	0	1
1	0	0	1
1	0	1	0

If you add rows with `NaN` scores to the false category in their respective classes, `perfcurve` computes performance measures as in the following table. For example, a cutoff of 0.5 corresponds to the middle row where now rows 2 and 4 are counted as incorrectly classified. Notice that only the `FN` and `FP` columns differ between these two tables.

TP	FN	FP	TN
0	2	1	1
1	1	1	1
1	1	2	0

By default `perfcurve` does not return the `'reject all'` values for `X` and `Y`. It can optionally do so upon request.

For data with three or more classes, `perfcurve` takes one positive class and a list of negative classes for input. The function computes the `X` and `Y` values using counts in the positive class to estimate `TP` and `FN`, and using counts in all negative classes to estimate `TN` and `FP`. `perfcurve` can optionally compute `Y` values for each negative class separately and, in addition to `Y`, return a matrix of size `M`-by-`C`, where `M` is the number of elements in `X` or `Y` and `C` is the number of negative classes. You can use this functionality to monitor components of the negative class contribution. For example, you can plot `TP` counts on the `X`-axis and `FP` counts on the `Y`-axis. In this case, the returned matrix shows how the `FP` component is split across negative classes.

References

[1] T. Fawcett, ROC Graphs: Notes and Practical Considerations for Researchers, 2004.

[2] M. Zweig and G. Campbell, Receiver-Operating Characteristic (ROC) Plots: A Fundamental Evaluation Tool in Clinical Medicine, Clin. Chem. 39/4, 561-577, 1993.

12

Markov Models

Introduction

Markov processes are examples of stochastic processes—processes that generate random sequences of outcomes or *states* according to certain probabilities. Markov processes are distinguished by being memoryless—their next state depends only on their current state, not on the history that led them there. Models of Markov processes are used in a wide variety of applications, from daily stock prices to the positions of genes in a chromosome.

Markov Chains

A Markov model is given visual representation with a *state diagram*, such as the one below.
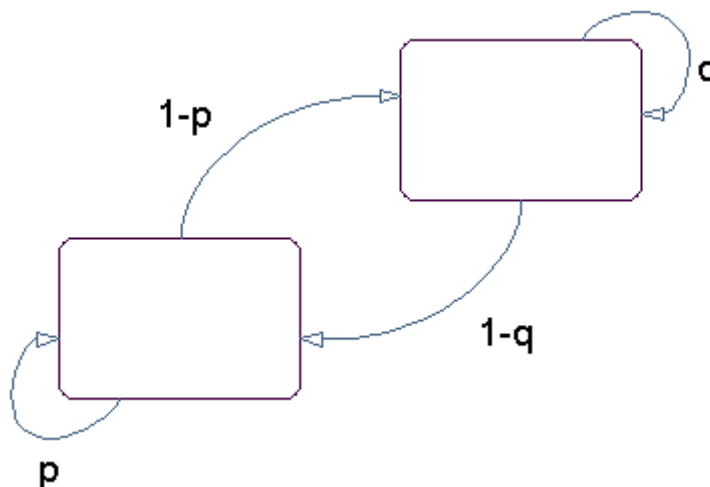
State Diagram for a Markov Model

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. At each step of the process, the model may generate an output, or *emission*, depending on which state it is in, and then make a transition to another state. An important characteristic of Markov models is that the next state depends only on the current state, and not on the history of transitions that lead to the current state.

For example, for 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. The emission might simply be the current state. In more complicated models, random processes at each state will generate emissions. You could, for example, roll a die to determine the emission at any step.

Markov chains are mathematical descriptions of Markov models with a discrete set of states. Markov chains are characterized by:

- 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 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, ... , 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.

Markov chains begin in an *initial state* i_0 at step 0. The chain then transitions to state i_1 with probability T_{1i_1}, and emits an output s_{k_1} with probability $E_{i_1 k_1}$. Consequently, the probability of observing the sequence of states $i_1 i_2 \ldots i_r$ and the sequence of emissions $s_{k_1} s_{k_2} \ldots s_{k_r}$ in the first *r* steps, is

$$T_{1i_1} E_{i_1 k_1} T_{i_1 i_2} E_{i_2 k_2} \ldots T_{i_{r-1} i_r} E_{i_r k}$$

Hidden Markov Models

In this section...
"Introduction" on page 12-5
"Analyzing Hidden Markov Models" on page 12-7

Introduction

A *hidden Markov model* is one in which you observe a sequence of emissions, but do not know the sequence of states the model went through to generate the emissions. Analyses of hidden Markov models seek to recover the sequence of states from the observed data.

As an example, consider a Markov model with two states and six possible emissions. The model uses:

- 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.

The model creates a sequence of numbers from the set {1, 2, 3, 4, 5, 6} with 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.

The state diagram for this model has two states, red and green, as shown in the following figure.
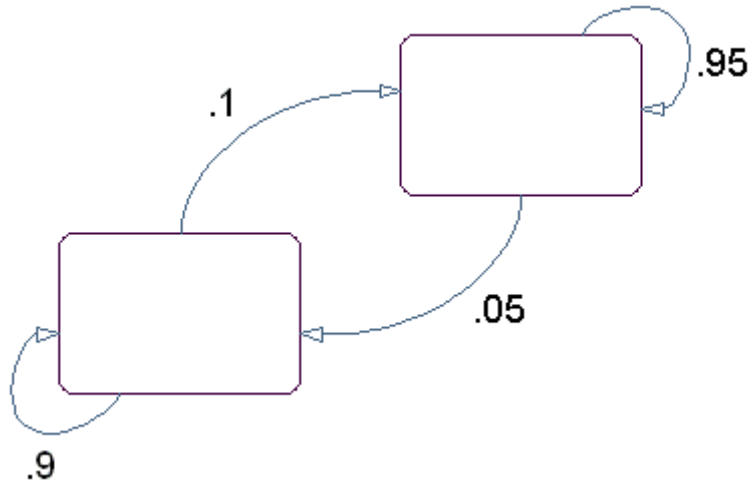
You determine the emission from a state by rolling the die with the same color as the state. You determine the transition to the next state by flipping the coin with the same color as the state.

The transition matrix is:

$$T = \begin{bmatrix} 0.9 & 0.1 \\ 0.05 & 0.95 \end{bmatrix}$$

The emissions matrix is:

$$E = \begin{bmatrix} \frac{1}{6} & \frac{1}{6} & \frac{1}{6} & \frac{1}{6} & \frac{1}{6} & \frac{1}{6} \\ \frac{7}{12} & \frac{1}{12} & \frac{1}{12} & \frac{1}{12} & \frac{1}{12} & \frac{1}{12} \end{bmatrix}$$

The model is not hidden because you know the sequence of states from the colors of the coins and dice. Suppose, however, that someone else is generating the emissions without showing you the dice or the coins. All you see is the

sequence of emissions. 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.

Hidden Markov models raise the following questions:

- Given a sequence of emissions, what is the most likely state path?
- Given a sequence of emissions, how can you estimate transition and emission probabilities of the model?
- What is the *forward probability* that the model generates a given sequence?
- What is the *posterior probability* that the model is in a particular state at any point in the sequence?

Analyzing Hidden Markov Models

- "Generating a Test Sequence" on page 12-8
- "Estimating the State Sequence" on page 12-8
- "Estimating Transition and Emission Matrices" on page 12-9
- "Estimating Posterior State Probabilities" on page 12-11
- "Changing the Initial State Distribution" on page 12-12

Statistics Toolbox functions related to hidden Markov models are:

- `hmmgenerate` — Generates a sequence of states and emissions from a Markov model
- `hmmestimate` — Calculates maximum likelihood estimates of transition and emission probabilities from a sequence of emissions and a known sequence of states
- `hmmtrain` — Calculates maximum likelihood estimates of transition and emission probabilities from a sequence of emissions
- `hmmviterbi` — Calculates the most probable state path for a hidden Markov model
- `hmmdecode` — Calculates the posterior state probabilities of a sequence of emissions

This section shows how to use these functions to analyze hidden Markov models.

Generating a Test Sequence

The following commands create the transition and emission matrices for the model described in the "Introduction" on page 12-5:

```
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];
```

To generate a random sequence of states and emissions from the model, use `hmmgenerate`:

```
[seq,states] = hmmgenerate(1000,TRANS,EMIS);
```

The output `seq` is the sequence of emissions and the output `states` is the sequence of states.

`hmmgenerate` begins in state 1 at step 0, makes the transition to state i_1 at step 1, and returns i_1 as the first entry in `states`. To change the initial state, see "Changing the Initial State Distribution" on page 12-12.

Estimating the State Sequence

Given the transition and emission matrices `TRANS` and `EMIS`, the function `hmmviterbi` uses the Viterbi algorithm to compute the most likely sequence of states the model would go through to generate a given sequence `seq` of emissions:

```
likelystates = hmmviterbi(seq, TRANS, EMIS);
```

`likelystates` is a sequence the same length as `seq`.

To test the accuracy of `hmmviterbi`, compute the percentage of the actual sequence `states` that agrees with the sequence `likelystates`.

```
sum(states==likelystates)/1000
ans =
    0.8200
```

In this case, the most likely sequence of states agrees with the random sequence 82% of the time.

Estimating Transition and Emission Matrices

- "Using hmmestimate" on page 12-9
- "Using hmmtrain" on page 12-10

The functions `hmmestimate` and `hmmtrain` estimate the transition and emission matrices `TRANS` and `EMIS` given a sequence `seq` of emissions.

Using hmmestimate. To function `hmmestimate` requires that you know the sequence of states `states` that the model went through to generate `seq`.

The following takes the emission and state sequences and returns estimates of the transition and emission matrices:

```
[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 the outputs with the original transition and emission matrices, `TRANS` and `EMIS`:

```
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 `states`, but you have initial guesses for `TRANS` and `EMIS`, you can still estimate `TRANS` and `EMIS` using `hmmtrain`.

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 estimate `TRANS` and `EMIS` as follows:

```
[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.

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, 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.

Change the 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.

Two factors reduce the reliability of the output matrices of `hmmtrain`:

- The algorithm converges to a local maximum that does not represent the true transition and emission matrices. If you suspect this, use different initial guesses for the matrices `TRANS_EST` and `EMIS_EST`.

- The sequence `seq` may be too short to properly train the matrices. If you suspect this, use a longer sequence for `seq`.

Estimating 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 compute the posterior state probabilities with `hmmdecode`:

```
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.

`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. To change the initial state, see "Changing the Initial State Distribution" on page 12-12.

To return the logarithm of the probability of the sequence `seq`, use the second output argument of `hmmdecode`:

```
[PSTATES,logpseq] = hmmdecode(seq,TRANS,EMIS)
```

The probability of a sequence tends to 0 as the length of the sequence increases, and the probability of a sufficiently long sequence becomes less than the smallest positive number your computer can represent. `hmmdecode` returns the logarithm of the probability to avoid this problem.

Changing the Initial State Distribution

By default, Statistics Toolbox hidden Markov model functions begin in state 1. In other words, the distribution of initial states has all of its probability mass concentrated at state 1. To assign a different distribution of probabilities, $p = [p_1, p_2, ..., p_M]$, to the M initial states, do the following:

1 Create an M+1-by-M+1 augmented transition matrix, $\hat{T}$ of 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. p must sum to 1.

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 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];
```

13

Design of Experiments

Introduction

Passive data collection leads to a number of problems in statistical modeling. Observed changes in a response variable may be correlated with, but not caused by, observed changes in individual *factors* (process variables). Simultaneous changes in multiple factors may produce interactions that are difficult to separate into individual effects. Observations may be dependent, while a model of the data considers them to be independent.

Designed experiments address these problems. In a designed experiment, the data-producing process is actively manipulated to improve the quality of information and to eliminate redundant data. A common goal of all experimental designs is to collect data as parsimoniously as possible while providing sufficient information to accurately estimate model parameters.

For example, a simple model of a response y in an experiment with two controlled factors x_1 and x_2 might look like this:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_1 x_2 + \varepsilon$$

Here ε includes both experimental error and the effects of any uncontrolled factors in the experiment. The terms $\beta_1 x_1$ and $\beta_2 x_2$ are *main effects* and the term $\beta_3 x_1 x_2$ is a two-way *interaction effect*. A designed experiment would systematically manipulate x_1 and x_2 while measuring y, with the objective of accurately estimating β_0, β_1, β_2, and β_3.

Full Factorial Designs

In this section...
"Multilevel Designs" on page 13-3
"Two-Level Designs" on page 13-4

Multilevel Designs

To systematically vary experimental factors, assign each factor a discrete set of *levels*. Full factorial designs measure response variables using every *treatment* (combination of the factor levels). A full factorial design for n factors with N_1, ..., N_n levels requires $N_1 \times ... \times N_n$ experimental runs—one for each treatment. While advantageous for separating individual effects, full factorial designs can make large demands on data collection.

As an example, suppose a machine shop has three machines and four operators. If the same operator always uses the same machine, it is impossible to determine if a machine or an operator is the cause of variation in production. By allowing every operator to use every machine, effects are separated. A full factorial list of treatments is generated by the Statistics Toolbox function `fullfact`:

```
dFF = fullfact([3,4])
dFF =
     1     1
     2     1
     3     1
     1     2
     2     2
     3     2
     1     3
     2     3
     3     3
     1     4
     2     4
     3     4
```

Each of the $3 \cdot 4 = 12$ rows of `dFF` represent one machine/operator combination.

Two-Level Designs

Many experiments can be conducted with two-level factors, using *two-level designs*. For example, suppose the machine shop in the previous example always keeps the same operator on the same machine, but wants to measure production effects that depend on the composition of the day and night shifts. The Statistics Toolbox function `ff2n` generates a full factorial list of treatments:

```
dFF2 = ff2n(4)
dFF2 =
     0     0     0     0
     0     0     0     1
     0     0     1     0
     0     0     1     1
     0     1     0     0
     0     1     0     1
     0     1     1     0
     0     1     1     1
     1     0     0     0
     1     0     0     1
     1     0     1     0
     1     0     1     1
     1     1     0     0
     1     1     0     1
     1     1     1     0
     1     1     1     1
```

Each of the $2^4 = 16$ rows of `dFF2` represent one schedule of operators for the day (0) and night (1) shifts.

Fractional Factorial Designs

In this section...

Introduction

Two-level designs are sufficient for evaluating many production processes. Factor levels of ±1 can indicate categorical factors, normalized factor extremes, or simply "up" and "down" from current factor settings. Experimenters evaluating process *changes* are interested primarily in the factor directions that lead to process improvement.

For experiments with many factors, two-level full factorial designs can lead to large amounts of data. For example, a two-level full factorial design with 10 factors requires $2^{10} = 1024$ runs. Often, however, individual factors or their interactions have no distinguishable effects on a response. This is especially true of higher order interactions. As a result, a well-designed experiment can use fewer runs for estimating model parameters.

Fractional factorial designs use a fraction of the runs required by full factorial designs. A subset of experimental treatments is selected based on an evaluation (or assumption) of which factors and interactions have the most significant effects. Once this selection is made, the experimental design must separate these effects. In particular, significant effects should not be *confounded*, that is, the measurement of one should not depend on the measurement of another.

Plackett-Burman Designs

Plackett-Burman designs are used when only main effects are considered significant. Two-level Plackett-Burman designs require a number of experimental runs that are a multiple of 4 rather than a power of 2. The MATLAB function `hadamard` generates these designs:

```
dPB = hadamard(8)
```

```
dPB =
     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
```

Binary factor levels are indicated by ±1. The design is for eight runs (the rows of dPB) manipulating seven two-level factors (the last seven columns of dPB). The number of runs is a fraction $8/2^7 = 0.0625$ of the runs required by a full factorial design. Economy is achieved at the expense of confounding main effects with any two-way interactions.

General Fractional Designs

At the cost of a larger fractional design, you can specify which interactions you wish to consider significant. A design of *resolution R* is one in which no *n*-factor interaction is confounded with any other effect containing less than $R - n$ factors. Thus, a resolution III design does not confound main effects with one another but may confound them with two-way interactions (as in "Plackett-Burman Designs" on page 13-5), while a resolution IV design does not confound either main effects or two-way interactions but may confound two-way interactions with each other.

Specify general fractional factorial designs using a full factorial design for a selected subset of *basic factors* and *generators* for the remaining factors. Generators are products of the basic factors, giving the levels for the remaining factors. Use the Statistics Toolbox function fracfact to generate these designs:

```
dfF = fracfact('a b c d bcd acd')
dfF =
    -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
```

This is a six-factor design in which four two-level basic factors (a, b, c, and d in the first four columns of dfF) are measured in every combination of levels, while the two remaining factors (in the last three columns of dfF) are measured only at levels defined by the generators bcd and acd, respectively. Levels in the generated columns are products of corresponding levels in the columns that make up the generator.

The challenge of creating a fractional factorial design is to choose basic factors and generators so that the design achieves a specified resolution in a specified number of runs. Use the Statistics Toolbox function fracfactgen to find appropriate generators:

```
generators = fracfactgen('a b c d e f',4,4)
generators =
    'a'
    'b'
    'c'
    'd'
    'bcd'
    'acd'
```

These are generators for a six-factor design with factors a through f, using 2^4 = 16 runs to achieve resolution IV. The fracfactgen function uses an efficient search algorithm to find generators that meet the requirements.

An optional output from fracfact displays the *confounding pattern* of the design:

```
[dfF,confounding] = fracfact(generators);
```

13-7

```
confounding
confounding =
    'Term'      'Generator'    'Confounding'
    'X1'        'a'            'X1'
    'X2'        'b'            'X2'
    'X3'        'c'            'X3'
    'X4'        'd'            'X4'
    'X5'        'bcd'          'X5'
    'X6'        'acd'          'X6'
    'X1*X2'     'ab'           'X1*X2 + X5*X6'
    'X1*X3'     'ac'           'X1*X3 + X4*X6'
    'X1*X4'     'ad'           'X1*X4 + X3*X6'
    'X1*X5'     'abcd'         'X1*X5 + X2*X6'
    'X1*X6'     'cd'           'X1*X6 + X2*X5 + X3*X4'
    'X2*X3'     'bc'           'X2*X3 + X4*X5'
    'X2*X4'     'bd'           'X2*X4 + X3*X5'
    'X2*X5'     'cd'           'X1*X6 + X2*X5 + X3*X4'
    'X2*X6'     'abcd'         'X1*X5 + X2*X6'
    'X3*X4'     'cd'           'X1*X6 + X2*X5 + X3*X4'
    'X3*X5'     'bd'           'X2*X4 + X3*X5'
    'X3*X6'     'ad'           'X1*X4 + X3*X6'
    'X4*X5'     'bc'           'X2*X3 + X4*X5'
    'X4*X6'     'ac'           'X1*X3 + X4*X6'
    'X5*X6'     'ab'           'X1*X2 + X5*X6'
```

The confounding pattern shows that main effects are effectively separated
by the design, but two-way interactions are confounded with various other
two-way interactions.

Response Surface Designs

Introduction

As discussed in "Response Surface Models" on page 8-45, quadratic response surfaces are simple models that provide a maximum or minimum without making additional assumptions about the form of the response. Quadratic models can be calibrated using full factorial designs with three or more levels for each factor, but these designs generally require more runs than necessary to accurately estimate model parameters. This section discusses designs for calibrating quadratic models that are much more efficient, using three or five levels for each factor, but not using all combinations of levels.

Central Composite Designs

Central composite designs (CCDs), also known as Box-Wilson designs, are appropriate for calibrating the full quadratic models described in "Response Surface Models" on page 8-45. There are three types of CCDs—circumscribed, inscribed, and faced—pictured below:

Faced
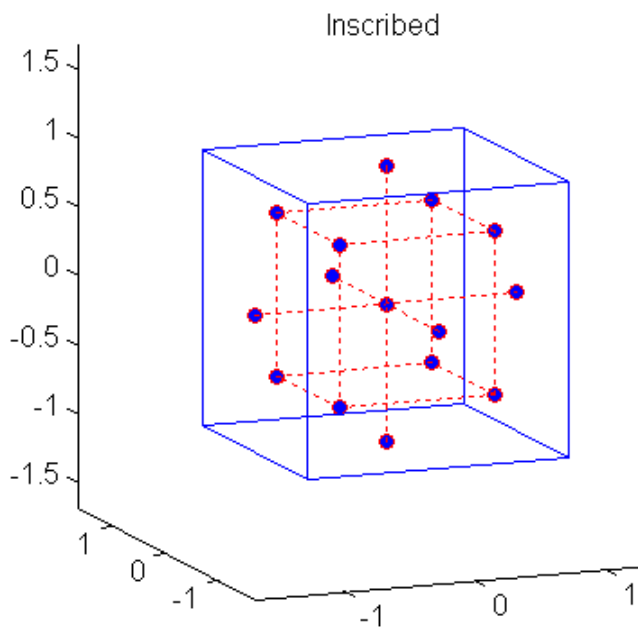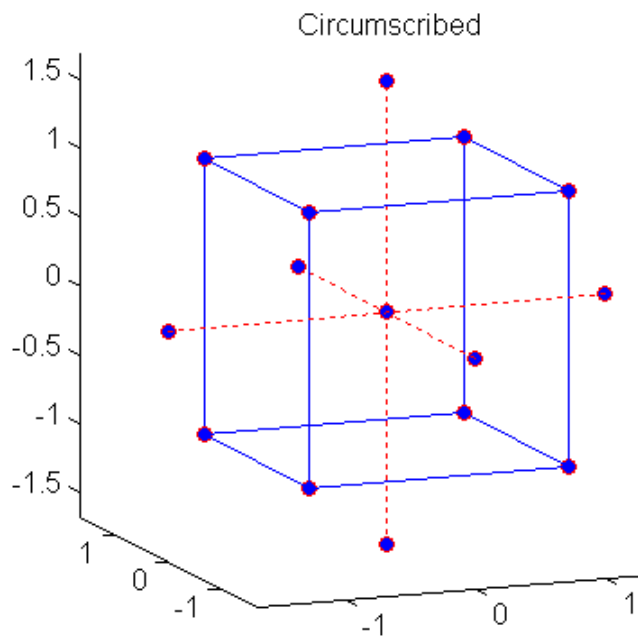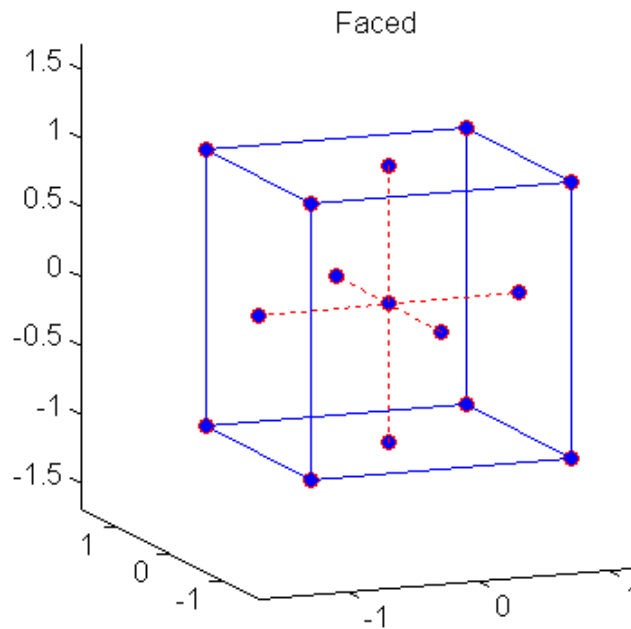
Each design consists of a factorial design (the corners of a cube) together with *center* and *star* points that allow for estimation of second-order effects. For a full quadratic model with n factors, CCDs specify $2^n + 2n + 1$ design points while estimating $(n + 2)(n + 1)/2$ coefficients.

The type of CCD used (the position of the factorial and star points) is determined by the number of factors and by the desired properties of the design. The following table summarizes some important properties. A design is *rotatable* if the prediction variance depends only on the distance of the design point from the center of the design.

Design	Rotatable	Factor Levels	Uses Points Outside ±1	Accuracy of Estimates
Circumscribed (CCC)	Yes	5	Yes	Good over entire design space

Design	Rotatable	Factor Levels	Uses Points Outside ±1	Accuracy of Estimates
Inscribed (CCI)	Yes	5	No	Good over central subset of design space
Faced (CCF)	No	3	No	Fair over entire design space; poor for pure quadratic coefficients

Generate CCDs with the Statistics Toolbox function ccdesign:

```
dCC = ccdesign(3,'type','circumscribed')
dCC =
   -1.0000   -1.0000   -1.0000
   -1.0000   -1.0000    1.0000
   -1.0000    1.0000   -1.0000
   -1.0000    1.0000    1.0000
    1.0000   -1.0000   -1.0000
    1.0000   -1.0000    1.0000
    1.0000    1.0000   -1.0000
    1.0000    1.0000    1.0000
   -1.6818         0         0
    1.6818         0         0
         0   -1.6818         0
         0    1.6818         0
         0         0   -1.6818
         0         0    1.6818
         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
```

The repeated center point runs allow for a more uniform estimate of the prediction variance over the entire design space.

Box-Behnken Designs

Like the designs described in "Central Composite Designs" on page 13-9, Box-Behnken designs are used to calibrate full quadratic models. Box-Behnken designs are rotatable and, for a small number of factors (four or less), require fewer runs than CCDs. By avoiding the corners of the design space, they allow experimenters to work around extreme factor combinations. Like an inscribed CCD, however, extremes are then poorly estimated.

The geometry of a Box-Behnken design is pictured in the following figure.
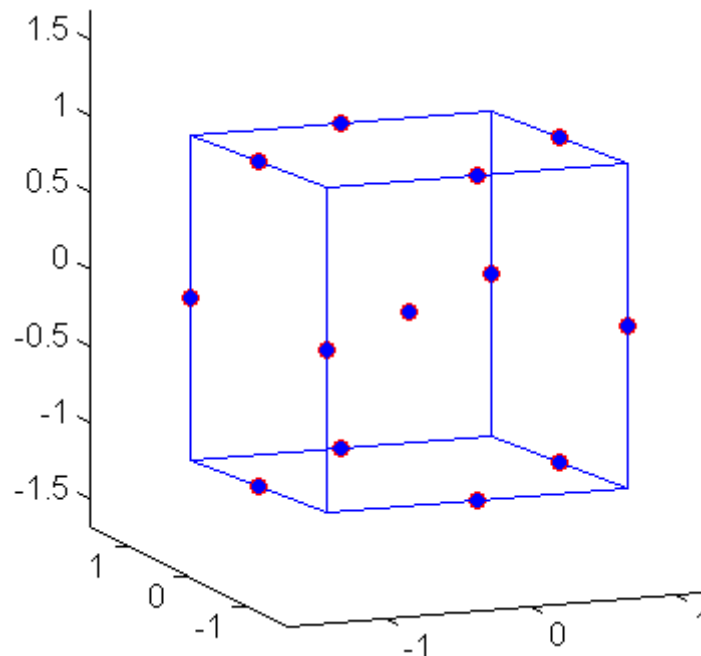
Design points are at the midpoints of edges of the design space and at the center, and do not contain an embedded factorial design.

Generate Box-Behnken designs with the Statistics Toolbox function bbdesign:

```
dBB = bbdesign(3)
dBB =
    -1    -1     0
    -1     1     0
     1    -1     0
     1     1     0
    -1     0    -1
    -1     0     1
     1     0    -1
     1     0     1
     0    -1    -1
     0    -1     1
     0     1    -1
     0     1     1
     0     0     0
     0     0     0
     0     0     0
```

Again, the repeated center point runs allow for a more uniform estimate of the prediction variance over the entire design space.

D-Optimal Designs

Introduction

Traditional experimental designs ("Full Factorial Designs" on page 13-3, "Fractional Factorial Designs" on page 13-5, and "Response Surface Designs" on page 13-9) are appropriate for calibrating linear models in experimental settings where factors are relatively unconstrained in the region of interest. In some cases, however, models are necessarily nonlinear. In other cases, certain treatments (combinations of factor levels) may be expensive or infeasible to measure. *D-optimal designs* are model-specific designs that address these limitations of traditional designs.

A *D*-optimal design is generated by an iterative search algorithm and seeks to minimize the covariance of the parameter estimates for a specified model. This is equivalent to maximizing the determinant $D = |X^TX|$, where X is the design matrix of model terms (the columns) evaluated at specific treatments in the design space (the rows). Unlike traditional designs, *D*-optimal designs do not require orthogonal design matrices, and as a result, parameter estimates may be correlated. Parameter estimates may also be locally, but not globally, *D*-optimal.

There are several Statistics Toolbox functions for generating *D*-optimal designs:

Function	Description
candexch	Uses a row-exchange algorithm to generate a *D*-optimal design with a specified number of runs for a specified model and a specified candidate set. This is the second component of the algorithm used by rowexch.
candgen	Generates a candidate set for a specified model. This is the first component of the algorithm used by rowexch.
cordexch	Uses a coordinate-exchange algorithm to generate a *D*-optimal design with a specified number of runs for a specified model.
daugment	Uses a coordinate-exchange algorithm to augment an existing *D*-optimal design with additional runs to estimate additional model terms.
dcovary	Uses a coordinate-exchange algorithm to generate a *D*-optimal design with fixed covariate factors.
rowexch	Uses a row-exchange algorithm to generate a *D*-optimal design with a specified number of runs for a specified model. The algorithm calls candgen and then candexch. (Call candexch separately to specify a candidate set.)

The following sections explain how to use these functions to generate *D*-optimal designs.

Note The Statistics Toolbox function rsmdemo generates simulated data for experimental settings specified by either the user or by a *D*-optimal design generated by cordexch. It uses the rstool interface to visualize response surface models fit to the data, and it uses the nlintool interface to visualize a nonlinear model fit to the data.

Generating D-Optimal Designs

Two Statistics Toolbox algorithms generate *D*-optimal designs:

- The cordexch function uses a coordinate-exchange algorithm
- The rowexch function uses a row-exchange algorithm

Both `cordexch` and `rowexch` use iterative search algorithms. They operate by incrementally changing an initial design matrix X to increase $D = |X^TX|$ at each step. In both algorithms, there is randomness built into the selection of the initial design and into the choice of the incremental changes. As a result, both algorithms may return locally, but not globally, D-optimal designs. Run each algorithm multiple times and select the best result for your final design. Both functions have a `'tries'` parameter that automates this repetition and comparison.

At each step, the row-exchange algorithm exchanges an entire row of X with a row from a design matrix C evaluated at a *candidate set* of feasible treatments. The `rowexch` function automatically generates a C appropriate for a specified model, operating in two steps by calling the `candgen` and `candexch` functions in sequence. Provide your own C by calling `candexch` directly. In either case, if C is large, its static presence in memory can affect computation.

The coordinate-exchange algorithm, by contrast, does not use a candidate set. (Or rather, the candidate set is the entire design space.) At each step, the coordinate-exchange algorithm exchanges a single element of X with a new element evaluated at a neighboring point in design space. The absence of a candidate set reduces demands on memory, but the smaller scale of the search means that the coordinate-exchange algorithm is more likely to become trapped in a local minimum than the row-exchange algorithm.

For example, suppose you want a design to estimate the parameters in the following three-factor, seven-term interaction model:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \beta_{12} x_1 x_2 + \beta_{13} x_1 x_3 + \beta_{23} x_2 x_3 + \varepsilon$$

Use `cordexch` to generate a D-optimal design with seven runs:

```
nfactors = 3;
nruns = 7;
[dCE,X] = cordexch(nfactors,nruns,'interaction','tries',10)
dCE =
    -1     1     1
    -1    -1    -1
     1     1     1
    -1     1    -1
     1    -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     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
```

Columns of the design matrix X are the model terms evaluated at each row of the design dCE. The terms appear in order from left to right:

1 Constant term

2 Linear terms (1, 2, 3)

3 Interaction terms (12, 13, 23)

Use X to fit the model, as described in "Linear Regression" on page 8-3, to response data measured at the design points in dCE.

Use rowexch in a similar fashion to generate an equivalent design:

```
    [dRE,X] = rowexch(nfactors,nruns,'interaction','tries',10)
    dRE =
          -1    -1     1
           1    -1     1
           1    -1    -1
           1     1     1
          -1    -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    -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
```

Augmenting D-Optimal Designs

In practice, you may want to add runs to a completed experiment to learn more about a process and estimate additional model coefficients. The daugment function uses a coordinate-exchange algorithm to augment an existing *D*-optimal design.

For example, the following eight-run design is adequate for estimating main effects in a four-factor model:

```
dCEmain = cordexch(4,8)
dCEmain =
     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
```

To estimate the six interaction terms in the model, augment the design with eight additional runs:

```
dCEinteraction = daugment(dCEmain,8,'interaction')
dCEinteraction =
     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 augmented design is full factorial, with the original eight runs in the first eight rows.

The `'start'` parameter of the `candexch` function provides the same functionality as `daugment`, but uses a row exchange algorithm rather than a coordinate-exchange algorithm.

Specifying Fixed Covariate Factors

In many experimental settings, certain factors and their covariates are constrained to a fixed set of levels or combinations of levels. These cannot be varied when searching for an optimal design. The `dcovary` function allows you to specify fixed covariate factors in the coordinate exchange algorithm.

For example, suppose you want a design to estimate the parameters in a three-factor linear additive model, with eight runs that necessarily occur at different times. If the process experiences temporal linear drift, you may want to include the run time as a variable in the model. Produce the design as follows:

```
time = linspace(-1,1,8)';
[dCV,X] = dcovary(3,time,'linear')
dCV =
   -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
X =
    1.0000   -1.0000    1.0000    1.0000   -1.0000
    1.0000    1.0000   -1.0000   -1.0000   -0.7143
    1.0000   -1.0000   -1.0000   -1.0000   -0.4286
    1.0000    1.0000   -1.0000    1.0000   -0.1429
```

```
1.0000    1.0000    1.0000   -1.0000    0.1429
1.0000   -1.0000    1.0000   -1.0000    0.4286
1.0000    1.0000    1.0000    1.0000    0.7143
1.0000   -1.0000   -1.0000    1.0000    1.0000
```

The column vector `time` is a fixed factor, normalized to values between ±1. The number of rows in the fixed factor specifies the number of runs in the design. The resulting design `dCV` gives factor settings for the three controlled model factors at each time.

Specifying Categorical Factors

Categorical factors take values in a discrete set of levels. Both `cordexch` and `rowexch` have a `'categorical'` parameter that allows you to specify the indices of categorical factors and a `'levels'` parameter that allows you to specify a number of levels for each factor.

For example, the following eight-run design is for a linear additive model with five factors in which the final factor is categorical with three levels:

```
dCEcat = cordexch(5,8,'linear','categorical',5,'levels',3)
dCEcat =
    -1    -1     1     1     2
    -1    -1    -1    -1     3
     1     1     1     1     3
     1     1    -1    -1     2
     1    -1    -1     1     3
    -1     1    -1     1     1
    -1     1     1    -1     3
     1    -1     1    -1     1
```

Specifying Candidate Sets

The row-exchange algorithm exchanges rows of an initial design matrix X with rows from a design matrix C evaluated at a candidate set of feasible treatments. The `rowexch` function automatically generates a C appropriate for a specified model, operating in two steps by calling the `candgen` and `candexch` functions in sequence. Provide your own C by calling `candexch` directly.

For example, the following uses rowexch to generate a five-run design for a two-factor pure quadratic model using a candidate set that is produced internally:

```
dRE1 = rowexch(2,5,'purequadratic','tries',10)
dRE1 =
    -1     1
     0     0
     1    -1
     1     0
     1     1
```

The same thing can be done using candgen and candexch in sequence:

```
[dC,C] = candgen(2,'purequadratic') % Candidate set, C
dC =
    -1    -1
     0    -1
     1    -1
    -1     0
     0     0
     1     0
    -1     1
     0     1
     1     1
C =
     1    -1    -1     1     1
     1     0    -1     0     1
     1     1    -1     1     1
     1    -1     0     1     0
     1     0     0     0     0
     1     1     0     1     0
     1    -1     1     1     1
     1     0     1     0     1
     1     1     1     1     1
treatments = candexch(C,5,'tries',10) % D-opt subset
treatments =
     2
     1
     7
     3
```

```
     4
dRE2 = dC(treatments,:) % Display design
dRE2 =
     0    -1
    -1    -1
    -1     1
     1    -1
    -1     0
```

You can replace C in this example with a design matrix evaluated at your own candidate set. For example, suppose your experiment is constrained so that the two factors cannot have extreme settings simultaneously. The following produces a restricted candidate set:

```
constraint = sum(abs(dC),2) < 2; % Feasible treatments
my_dC = dC(constraint,:)
my_dC =
     0    -1
    -1     0
     0     0
     1     0
     0     1
```

Use the x2fx function to convert the candidate set to a design matrix:

```
my_C = x2fx(my_dC,'purequadratic')
my_C =
     1     0    -1     0     1
     1    -1     0     1     0
     1     0     0     0     0
     1     1     0     1     0
     1     0     1     0     1
```

Find the required design in the same manner:

```
my_treatments = candexch(my_C,5,'tries',10) % D-opt subset
my_treatments =
     2
     4
     5
     1
```

```
        3
my_dRE = my_dC(my_treatments,:) % Display design
my_dRE =
    -1     0
     1     0
     0     1
     0    -1
     0     0
```

14

Statistical Process Control

Introduction

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 13, "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.

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')
```
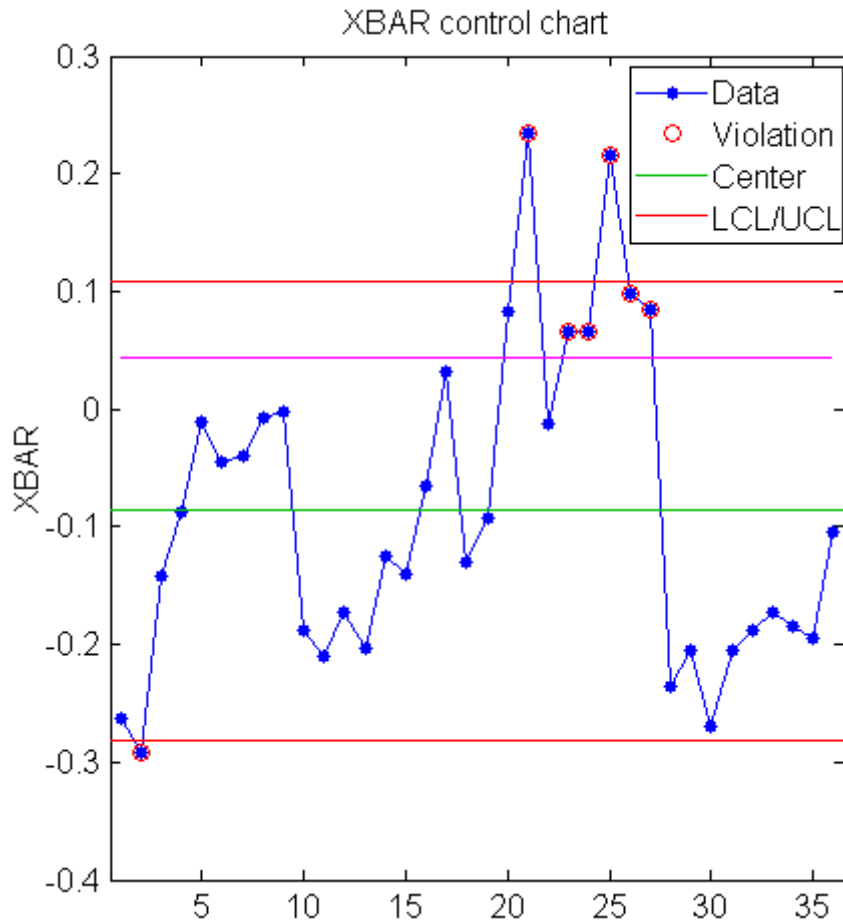

Measurements that violate the control rule can then be identified:

```
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
```

15
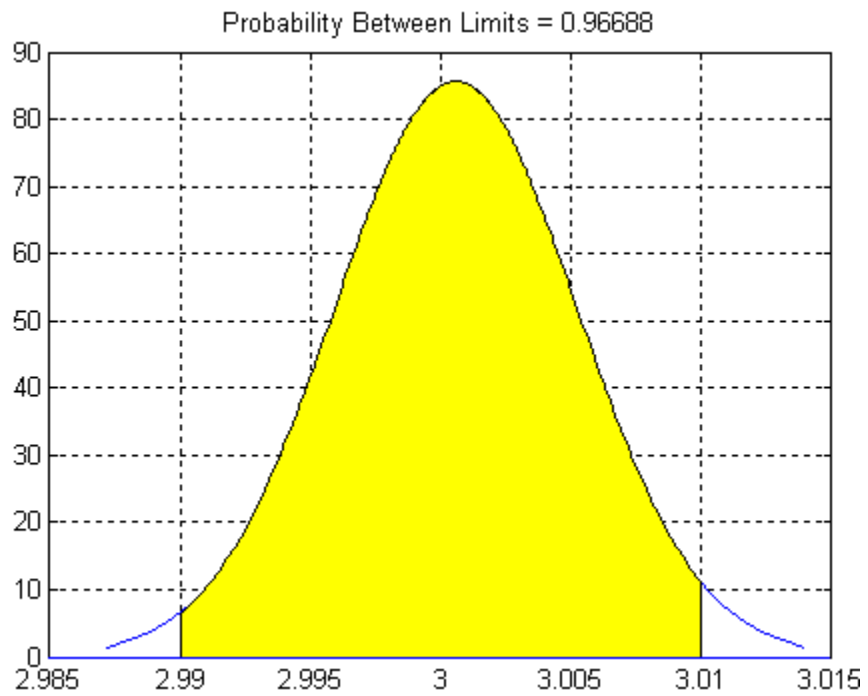
Function Reference

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 tab-delimited file

Data Organization

Categorical Arrays

addlevels (categorical)	Add levels to categorical array
cat (categorical)	Concatenate categorical arrays
categorical	Create categorical array
cellstr (categorical)	Convert categorical array to cell array of strings
char (categorical)	Convert categorical array to character array
circshift (categorical)	Shift categorical array circularly
ctranspose (categorical)	Transpose categorical matrix
double (categorical)	Convert categorical array to double array
droplevels (categorical)	Drop levels
end (categorical)	Last index in indexing expression for categorical array
flipdim (categorical)	Flip categorical array along specified dimension
fliplr (categorical)	Flip categorical matrix in left/right direction
flipud (categorical)	Flip categorical matrix in up/down direction
getlabels (categorical)	Access categorical array labels
getlevels (categorical)	Get categorical array levels

horzcat (categorical)	Horizontal concatenation for categorical arrays
int16 (categorical)	Convert categorical array to signed 16-bit integer array
int32 (categorical)	Convert categorical array to signed 32-bit integer array
int64 (categorical)	Convert categorical array to signed 64-bit integer array
int8 (categorical)	Convert categorical array to signed 8-bit integer array
intersect (categorical)	Set intersection for categorical arrays
ipermute (categorical)	Inverse permute dimensions of categorical array
isempty (categorical)	True for empty categorical array
isequal (categorical)	True if categorical arrays are equal
islevel (categorical)	Test for levels
ismember (categorical)	True for elements of categorical array in set
ismember (ordinal)	Test for membership
isscalar (categorical)	True if categorical array is scalar
isundefined (categorical)	Test for undefined elements
isvector (categorical)	True if categorical array is vector
length (categorical)	Length of categorical array
levelcounts (categorical)	Element counts by level
mergelevels (ordinal)	Merge levels
ndims (categorical)	Number of dimensions of categorical array
nominal	Construct nominal categorical array

numel (categorical)	Number of elements in categorical array
ordinal	Construct ordinal categorical array
permute (categorical)	Permute dimensions of categorical array
reorderlevels (categorical)	Reorder levels
repmat (categorical)	Replicate and tile categorical array
reshape (categorical)	Resize categorical array
rot90 (categorical)	Rotate categorical matrix 90 degrees
setdiff (categorical)	Set difference for categorical arrays
setlabels (categorical)	Label levels
setxor (categorical)	Set exclusive-or for categorical arrays
shiftdim (categorical)	Shift dimensions of categorical array
single (categorical)	Convert categorical array to single array
size (categorical)	Size of categorical array
sort (ordinal)	Sort elements of ordinal array
sortrows (ordinal)	Sort rows
squeeze (categorical)	Squeeze singleton dimensions from categorical array
summary (categorical)	Summary statistics for categorical array
times (categorical)	Product of categorical arrays
transpose (categorical)	Transpose categorical matrix
uint16 (categorical)	Convert categorical array to unsigned 16-bit integers
uint32 (categorical)	Convert categorical array to unsigned 32-bit integers

uint64 (categorical)	Convert categorical array to unsigned 64-bit integers
uint8 (categorical)	Convert categorical array to unsigned 8-bit integers
union (categorical)	Set union for categorical arrays
unique (categorical)	Unique values in categorical array
vertcat (categorical)	Vertical concatenation for categorical arrays

Dataset Arrays

cat (dataset)	Concatenate dataset arrays
dataset	Construct dataset array
datasetfun (dataset)	Apply function to dataset array variables
double (dataset)	Convert dataset variables to double array
end (dataset)	Last index in indexing expression for dataset array
export (dataset)	Write dataset array to file
get (dataset)	Access dataset array properties
grpstats (dataset)	Summary statistics by group for dataset arrays
horzcat (dataset)	Horizontal concatenation for dataset arrays
isempty (dataset)	True for empty dataset array
join (dataset)	Merge observations
length (dataset)	Length of dataset array
ndims (dataset)	Number of dimensions of dataset array

numel (dataset)	Number of elements in dataset array
replacedata (dataset)	Replace dataset variables
set (dataset)	Set and display properties
single (dataset)	Convert dataset variables to single array
size (dataset)	Size of dataset array
sortrows (dataset)	Sort rows of dataset array
summary (dataset)	Summary statistics for dataset array
unique (dataset)	Unique observations in dataset array
vertcat (dataset)	Vertical concatenation for dataset arrays

Grouped Data

gplotmatrix	Matrix of scatter plots by group
grp2idx	Create index vector from grouping variable
grpstats	Summary statistics by group
gscatter	Scatter plot by group

Descriptive Statistics

Summaries

crosstab	Cross-tabulation
grpstats	Summary statistics by group
summary (categorical)	Summary statistics for categorical array
tabulate	Frequency table

Measures of Central Tendency

geomean	Geometric mean
harmmean	Harmonic mean
trimmean	Mean excluding outliers

Measures of Dispersion

iqr	Interquartile range
mad	Mean or median absolute deviation

moment	Central moments
range	Range of values

Measures of Shape

kurtosis	Kurtosis
moment	Central moments
prctile	Percentiles
quantile	Quantiles
skewness	Skewness
zscore	Standardized z-scores

Statistics Resampling

bootci	Bootstrap confidence interval
bootstrp	Bootstrap sampling
jackknife	Jackknife sampling

Data with Missing Values

nancov	Covariance ignoring NaN values
nanmax	Maximum ignoring NaN values
nanmean	Mean ignoring NaN values
nanmedian	Median ignoring NaN values
nanmin	Minimum ignoring NaN values
nanstd	Standard deviation ignoring NaN values
nansum	Sum ignoring NaN values
nanvar	Variance, ignoring NaN values

Data Correlation

canoncorr	Canonical correlation
cholcov	Cholesky-like covariance decomposition
cophenet	Cophenetic correlation coefficient
corr	Linear or rank correlation
corrcov	Convert covariance matrix to correlation matrix
partialcorr	Linear or rank partial correlation coefficients
tiedrank	Rank adjusted for ties

Statistical Visualization

Distribution Plots

boxplot	Box plot
cdfplot	Empirical cumulative distribution function plot
dfittool	Interactive distribution fitting
disttool	Interactive density and distribution plots
ecdfhist	Empirical cumulative distribution function histogram
fsurfht	Interactive contour plot
hist3	Bivariate histogram
histfit	Histogram with normal fit
normplot	Normal probability plot
normspec	Normal density plot between specifications
pareto	Pareto chart
probplot	Probability plots

qqplot	Quantile-quantile plot
randtool	Interactive random number generation
scatterhist	Scatter plot with marginal histograms
surfht	Interactive contour plot
wblplot	Weibull probability plot

Scatter Plots

gline	Interactively add line to plot
gname	Add case names to plot
gplotmatrix	Matrix of scatter plots by group
gscatter	Scatter plot by group
lsline	Add least-squares line to scatter plot
refcurve	Add reference curve to plot
refline	Add reference line to plot
scatterhist	Scatter plot with marginal histograms

ANOVA Plots

anova1	One-way analysis of variance
aoctool	Interactive analysis of covariance
manovacluster	Dendrogram of group mean clusters following MANOVA
multcompare	Multiple comparison test

Regression Plots

addedvarplot	Added-variable plot
gline	Interactively add line to plot
lsline	Add least-squares line to scatter plot
polytool	Interactive polynomial fitting
rcoplot	Residual case order plot
refcurve	Add reference curve to plot
refline	Add reference line to plot
robustdemo	Interactive robust regression
rsmdemo	Interactive response surface demonstration
rstool	Interactive response surface modeling
view (classregtree)	Plot tree

Multivariate Plots

andrewsplot	Andrews plot
biplot	Biplot
glyphplot	Glyph plot
parallelcoords	Parallel coordinates plot

Cluster Plots

dendrogram	Dendrogram plot
manovacluster	Dendrogram of group mean clusters following MANOVA
silhouette	Silhouette plot

Classification Plots

perfcurve	Compute Receiver Operating Characteristic (ROC) curve or other performance curve for classifier output
view (classregtree)	Plot tree

DOE Plots

interactionplot	Interaction plot for grouped data
maineffectsplot	Main effects plot for grouped data
multivarichart	Multivari chart for grouped data
rsmdemo	Interactive response surface demonstration
rstool	Interactive response surface modeling

SPC Plots

capaplot	Process capability plot
controlchart	Shewhart control charts
histfit	Histogram with normal fit
normspec	Normal density plot between specifications

Probability Distributions

Distribution Objects

cdf (ProbDist)	Return cumulative distribution function (CDF) for ProbDist object
fitdist	Fit probability distribution to data
icdf (ProbDistUnivKernel)	Return inverse cumulative distribution function (ICDF) for ProbDistUnivKernel object
icdf (ProbDistUnivParam)	Return inverse cumulative distribution function (ICDF) for ProbDistUnivParam object
iqr (ProbDistUnivKernel)	Return interquartile range (IQR) for ProbDistUnivKernel object
iqr (ProbDistUnivParam)	Return interquartile range (IQR) for ProbDistUnivParam object

15-15

mean (ProbDistUnivParam)	Return mean of ProbDistUnivParam object
median (ProbDistUnivKernel)	Return median of ProbDistUnivKernel object
median (ProbDistUnivParam)	Return median of ProbDistUnivParam object
paramci (ProbDistUnivParam)	Return parameter confidence intervals of ProbDistUnivParam object
pdf (ProbDist)	Return probability density function (PDF) for ProbDist object
ProbDistUnivKernel	Construct ProbDistUnivKernel object
ProbDistUnivParam	Construct ProbDistUnivParam object
random (ProbDist)	Generate random number drawn from ProbDist object
std (ProbDistUnivParam)	Return standard deviation of ProbDistUnivParam object
var (ProbDistUnivParam)	Return variance of ProbDistUnivParam object

Distribution Plots

boxplot	Box plot
cdfplot	Empirical cumulative distribution function plot
dfittool	Interactive distribution fitting
disttool	Interactive density and distribution plots
ecdfhist	Empirical cumulative distribution function histogram

fsurfht	Interactive contour plot
hist3	Bivariate histogram
histfit	Histogram with normal fit
normplot	Normal probability plot
normspec	Normal density plot between specifications
pareto	Pareto chart
probplot	Probability plots
qqplot	Quantile-quantile plot
randtool	Interactive random number generation
scatterhist	Scatter plot with marginal histograms
surfht	Interactive contour plot
wblplot	Weibull probability plot

Probability Density

betapdf	Beta probability density function
binopdf	Binomial probability density function
chi2pdf	Chi-square probability density function
copulapdf	Copula probability density function
disttool	Interactive density and distribution 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
ksdensity	Kernel smoothing density estimate
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 functions
pdf (gmdistribution)	Probability density function for Gaussian mixture distribution

pdf (piecewisedistribution)	Probability density function for piecewise distribution
poisspdf	Poisson probability density function
random (piecewisedistribution)	Random numbers from piecewise distribution
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

betacdf	Beta cumulative distribution function
binocdf	Binomial cumulative distribution function
cdf	Cumulative distribution functions
cdf (gmdistribution)	Cumulative distribution function for Gaussian mixture distribution
cdf (piecewisedistribution)	Cumulative distribution function for piecewise distribution
cdfplot	Empirical cumulative distribution function plot
chi2cdf	Chi-square cumulative distribution function
copulacdf	Copula cumulative distribution function

disttool	Interactive density and distribution plots
ecdf	Empirical cumulative distribution function
ecdfhist	Empirical cumulative distribution function histogram
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

betainv	Beta inverse cumulative distribution function
binoinv	Binomial inverse cumulative distribution function
chi2inv	Chi-square inverse cumulative distribution function
evinv	Extreme value inverse cumulative distribution function
expinv	Exponential inverse cumulative distribution function
finv	F inverse cumulative distribution function

gaminv	Gamma inverse cumulative distribution function
geoinv	Geometric inverse cumulative distribution function
gevinv	Generalized extreme value inverse cumulative distribution function
gpinv	Generalized Pareto inverse cumulative distribution function
hygeinv	Hypergeometric inverse cumulative distribution function
icdf	Inverse cumulative distribution functions
icdf (piecewisedistribution)	Inverse cumulative distribution function for piecewise distribution
logninv	Lognormal inverse cumulative distribution function
nbininv	Negative binomial inverse cumulative distribution function
ncfinv	Noncentral F inverse cumulative distribution function
nctinv	Noncentral t inverse cumulative distribution function
ncx2inv	Noncentral chi-square inverse cumulative distribution function
norminv	Normal inverse cumulative distribution function
poissinv	Poisson inverse cumulative distribution function
raylinv	Rayleigh inverse cumulative distribution function
tinv	Student's t inverse cumulative distribution function

unidinv	Discrete uniform inverse cumulative distribution function
unifinv	Continuous uniform inverse cumulative distribution function
wblinv	Weibull inverse cumulative distribution function

Distribution Statistics

betastat	Beta mean and variance
binostat	Binomial mean and variance
chi2stat	Chi-square mean and variance
copulastat	Copula rank correlation
evstat	Extreme value mean and variance
expstat	Exponential mean and variance
fstat	F mean and variance
gamstat	Gamma mean and variance
geostat	Geometric mean and variance
gevstat	Generalized extreme value mean and variance
gpstat	Generalized Pareto mean and variance
hygestat	Hypergeometric mean and variance
lognstat	Lognormal mean and variance
nbinstat	Negative binomial mean and variance
ncfstat	Noncentral F mean and variance
nctstat	Noncentral t mean and variance
ncx2stat	Noncentral chi-square mean and variance

normstat	Normal mean and variance
poisstat	Poisson mean and variance
raylstat	Rayleigh mean and variance
tstat	Student's t mean and variance
unidstat	Discrete uniform mean of and variance
unifstat	Continuous uniform mean and variance
wblstat	Weibull mean and variance

Distribution Fitting

Supported Distributions

betafit	Beta parameter estimates
binofit	Binomial parameter estimates
copulafit	Fit copula to data
copulaparam	Copula parameters as function of rank correlation
dfittool	Interactive distribution fitting
evfit	Extreme value parameter estimates
expfit	Exponential parameter estimates
fit (gmdistribution)	Gaussian mixture parameter estimates
gamfit	Gamma parameter estimates
gevfit	Generalized extreme value parameter estimates

gpfit	Generalized Pareto parameter estimates
histfit	Histogram with normal fit
johnsrnd	Johnson system random numbers
lognfit	Lognormal parameter estimates
mle	Maximum likelihood estimates
mlecov	Asymptotic covariance of maximum likelihood estimators
nbinfit	Negative binomial parameter estimates
normfit	Normal parameter estimates
normplot	Normal probability plot
pearsrnd	Pearson system random numbers
poissfit	Poisson parameter estimates
raylfit	Rayleigh parameter estimates
unifit	Continuous uniform parameter estimates
wblfit	Weibull parameter estimates
wblplot	Weibull probability plot

Piecewise Distributions

boundary (piecewisedistribution)	Piecewise distribution boundaries
lowerparams (paretotails)	Lower Pareto tails parameters
nsegments (piecewisedistribution)	Number of segments
paretotails	Construct Pareto tails object
piecewisedistribution	Create piecewise distribution object
segment (piecewisedistribution)	Segments containing values
upperparams (paretotails)	Upper Pareto tails parameters

Negative Log-Likelihood

betalike	Beta negative log-likelihood
evlike	Extreme value negative log-likelihood
explike	Exponential negative log-likelihood
gamlike	Gamma negative log-likelihood
gevlike	Generalized extreme value negative log-likelihood
gplike	Generalized Pareto negative log-likelihood
lognlike	Lognormal negative log-likelihood
mvregresslike	Negative log-likelihood for multivariate regression
normlike	Normal negative log-likelihood
wbllike	Weibull negative log-likelihood

Random Number Generators

betarnd	Beta random numbers
binornd	Binomial random numbers
chi2rnd	Chi-square random numbers
copularnd	Copula random numbers
evrnd	Extreme value random numbers
exprnd	Exponential random numbers
frnd	*F* random numbers
gamrnd	Gamma random numbers
geornd	Geometric random numbers
gevrnd	Generalized extreme value random numbers

gprnd	Generalized Pareto random numbers
hygernd	Hypergeometric random numbers
iwishrnd	Inverse Wishart random numbers
johnsrnd	Johnson system random numbers
lhsdesign	Latin hypercube sample
lhsnorm	Latin hypercube sample from normal distribution
lognrnd	Lognormal random numbers
mhsample	Metropolis-Hastings sample
mnrnd	Multinomial random numbers
mvnrnd	Multivariate normal random numbers
mvtrnd	Multivariate t random numbers
nbinrnd	Negative binomial random numbers
ncfrnd	Noncentral F random numbers
nctrnd	Noncentral t random numbers
ncx2rnd	Noncentral chi-square random numbers
normrnd	Normal random numbers
pearsrnd	Pearson system random numbers
poissrnd	Poisson random numbers
randg	Gamma random numbers
random	Random numbers
random (gmdistribution)	Random numbers from Gaussian mixture distribution
random (piecewisedistribution)	Random numbers from piecewise distribution
randsample	Random sample

randtool	Interactive random number generation
raylrnd	Rayleigh random numbers
slicesample	Slice sampler
trnd	Student's t random numbers
unidrnd	Discrete uniform random numbers
unifrnd	Continuous uniform random numbers
wblrnd	Weibull random numbers
wishrnd	Wishart random numbers

Quasi-Random Numbers

addlistener (qrandstream)	Add listener for event
delete (qrandstream)	Delete handle object
end (qrandset)	Last index in indexing expression for point set
eq (qrandstream)	Test handle equality
findobj (qrandstream)	Find objects matching specified conditions
findprop (qrandstream)	Find property of MATLAB handle object
ge (qrandstream)	Greater than or equal relation for handles
gt (qrandstream)	Greater than relation for handles
haltonset	Construct Halton quasi-random point set
isvalid (qrandstream)	Test handle validity
le (qrandstream)	Less than or equal relation for handles

length (qrandset)	Length of point set
lt (qrandstream)	Less than relation for handles
ndims (qrandset)	Number of dimensions in matrix
ne (qrandstream)	Not equal relation for handles
net (qrandset)	Generate quasi-random point set
notify (qrandstream)	Notify listeners of event
qrand (qrandstream)	Generate quasi-random points from stream
qrandset	Create categorical array
qrandstream	Construct quasi-random number stream
rand (qrandstream)	Generate quasi-random points from stream
reset (qrandstream)	Reset state
scramble (qrandset)	Scramble quasi-random point set
size (qrandset)	Number of dimensions in matrix
sobolset	Construct Sobol quasi-random point set

Piecewise Distributions

boundary (piecewisedistribution)	Piecewise distribution boundaries
cdf (piecewisedistribution)	Cumulative distribution function for piecewise distribution
icdf (piecewisedistribution)	Inverse cumulative distribution function for piecewise distribution
lowerparams (paretotails)	Lower Pareto tails parameters
nsegments (piecewisedistribution)	Number of segments
paretotails	Construct Pareto tails object

pdf (piecewisedistribution)	Probability density function for piecewise distribution
piecewisedistribution	Create piecewise distribution object
random (piecewisedistribution)	Random numbers from piecewise distribution
segment (piecewisedistribution)	Segments containing values
upperparams (paretotails)	Upper Pareto tails parameters

Hypothesis Tests

ansaribradley	Ansari-Bradley test
barttest	Bartlett's test
canoncorr	Canonical correlation
chi2gof	Chi-square goodness-of-fit test
dwtest	Durbin-Watson test
friedman	Friedman's test
jbtest	Jarque-Bera test
kruskalwallis	Kruskal-Wallis test
kstest	One-sample Kolmogorov-Smirnov test
kstest2	Two-sample Kolmogorov-Smirnov test
lillietest	Lilliefors test
linhyptest	Linear hypothesis test
ranksum	Wilcoxon rank sum test
runstest	Run test for randomness
sampsizepwr	Sample size and power of test
signrank	Wilcoxon signed rank test
signtest	Sign test
ttest	One-sample t-test
ttest2	Two-sample t-test
vartest	Chi-square variance test
vartest2	Two-sample F-test for equal variances
vartestn	Bartlett multiple-sample test for equal variances

zscore	Standardized *z*-scores
ztest	*z*-test

Analysis of Variance

ANOVA Plots (p. 15-32)

ANOVA Operations (p. 15-32)

ANOVA Plots

anova1	One-way analysis of variance
aoctool	Interactive analysis of covariance
manovacluster	Dendrogram of group mean clusters following MANOVA
multcompare	Multiple comparison test

ANOVA Operations

anova1	One-way analysis of variance
anova2	Two-way analysis of variance
anovan	*N*-way analysis of variance
aoctool	Interactive analysis of covariance
dummyvar	Create dummy variables
friedman	Friedman's test
kruskalwallis	Kruskal-Wallis test
manova1	One-way multivariate analysis of variance

manovacluster	Dendrogram of group mean clusters following MANOVA
multcompare	Multiple comparison test

Regression Analysis

Regression Plots

addedvarplot	Added-variable plot
gline	Interactively add line to plot
lsline	Add least-squares line to scatter plot
polytool	Interactive polynomial fitting
rcoplot	Residual case order plot
refcurve	Add reference curve to plot
refline	Add reference line to plot
robustdemo	Interactive robust regression
rsmdemo	Interactive response surface demonstration
rstool	Interactive response surface modeling
view (classregtree)	Plot tree

Linear Regression

coxphfit	Cox proportional hazards regression
dummyvar	Create dummy variables
glmfit	Generalized linear model regression
glmval	Generalized linear model values
invpred	Inverse prediction
leverage	Leverage
mnrfit	Multinomial logistic regression
mnrval	Multinomial logistic regression values
mvregress	Multivariate linear regression
mvregresslike	Negative log-likelihood for multivariate regression
plsregress	Partial least-squares regression
polyconf	Polynomial confidence intervals
polytool	Interactive polynomial fitting
regress	Multiple linear regression
regstats	Regression diagnostics
ridge	Ridge regression
robustdemo	Interactive robust regression
robustfit	Robust regression
rsmdemo	Interactive response surface demonstration
rstool	Interactive response surface modeling
stepwise	Interactive stepwise regression
stepwisefit	Stepwise regression
x2fx	Convert predictor matrix to design matrix

Nonlinear Regression

dummyvar	Create dummy variables
hougen	Hougen-Watson model
nlinfit	Nonlinear regression
nlintool	Interactive nonlinear regression
nlmefit	Nonlinear mixed-effects estimation
nlparci	Nonlinear regression parameter confidence intervals
nlpredci	Nonlinear regression prediction confidence intervals

Regression Trees

catsplit (classregtree)	Categorical splits used for branches in decision tree
children (classregtree)	Child nodes
classcount (classregtree)	Class counts
classprob (classregtree)	Class probabilities
classregtree	Construct classification and regression trees
cutcategories (classregtree)	Cut categories
cutpoint (classregtree)	Returns decision tree cut point values
cuttype (classregtree)	Cut types
cutvar (classregtree)	Cut variable names
eval (classregtree)	Predicted responses
isbranch (classregtree)	Test node for branch
nodeerr (classregtree)	Return vector of node errors
nodeprob (classregtree)	Node probabilities

nodesize (classregtree)	Return node size
numnodes (classregtree)	Number of nodes
parent (classregtree)	Parent node
prune (classregtree)	Prune tree
risk (classregtree)	Node risks
test (classregtree)	Error rate
type (classregtree)	Tree type
varimportance (classregtree)	Compute embedded estimates of input feature importance
view (classregtree)	Plot tree

Ensemble Methods

append (TreeBagger)	Append new trees to ensemble
combine (CompactTreeBagger)	Combine two ensembles
compact (TreeBagger)	Compact ensemble of decision trees
CompactTreeBagger	Create CompactTreeBagger object
error (CompactTreeBagger)	Error (misclassification probability or MSE)
error (TreeBagger)	Error (misclassification probability or MSE)
fillProximities (TreeBagger)	Proximity matrix for training data
growTrees (TreeBagger)	Train additional trees and add to ensemble
margin (CompactTreeBagger)	Classification margin
margin (TreeBagger)	Classification margin
mdsProx (CompactTreeBagger)	Multidimensional scaling of proximity matrix
mdsProx (TreeBagger)	Multidimensional scaling of proximity matrix

meanMargin (CompactTreeBagger)	Mean classification margin
meanMargin (TreeBagger)	Mean classification margin
oobError (TreeBagger)	Out-of-bag error
oobMargin (TreeBagger)	Out-of-bag margins
oobMeanMargin (TreeBagger)	Out-of-bag mean margins
oobPredict (TreeBagger)	Ensemble predictions for out-of-bag observations
outlierMeasure (CompactTreeBagger)	Outlier measure for data
predict (CompactTreeBagger)	Predict response
predict (TreeBagger)	Predict response
proximity (CompactTreeBagger)	Proximity matrix for data
SetDefaultYfit (CompactTreeBagger)	Set default value for `predict`
TreeBagger	Create ensemble of bagged decision trees

Multivariate Methods

Multivariate Plots

andrewsplot	Andrews plot
biplot	Biplot
glyphplot	Glyph plot
parallelcoords	Parallel coordinates plot

Multidimensional Scaling

cmdscale	Classical multidimensional scaling
mahal	Mahalanobis distance
mdscale	Nonclassical multidimensional scaling
pdist	Pairwise distance
squareform	Format distance matrix

Procrustes Analysis

procrustes	Procrustes analysis

Feature Selection

sequentialfs   Sequential feature selection

Feature Transformation

Nonnegative Matrix Factorization
(p. 15-39)

Principal Component Analysis
(p. 15-39)

Factor Analysis (p. 15-39)

Nonnegative Matrix Factorization

nnmf   Nonnegative matrix factorization

Principal Component Analysis

barttest   Bartlett's test
pareto   Pareto chart
pcacov   Principal component analysis on
                                 covariance matrix
pcares   Residuals from principal component
                                 analysis
princomp   Principal component analysis on
                                 data

Factor Analysis

factoran   Factor analysis

Cluster Analysis

Cluster Plots

dendrogram	Dendrogram plot
manovacluster	Dendrogram of group mean clusters following MANOVA
silhouette	Silhouette plot

Hierarchical Clustering

cluster	Construct clusters from linkages
clusterdata	Construct clusters from data
cophenet	Cophenetic correlation coefficient
inconsistent	Inconsistency coefficient
linkage	Create hierarchical cluster tree
pdist	Pairwise distance
squareform	Format distance matrix

K-Means Clustering

kmeans	*K*-means clustering
mahal	Mahalanobis distance

Gaussian Mixture Models

cdf (gmdistribution)	Cumulative distribution function for Gaussian mixture distribution
cluster (gmdistribution)	Construct clusters from Gaussian mixture distribution
gmdistribution	Construct Gaussian mixture distribution
mahal (gmdistribution)	Mahalanobis distance to component means
pdf (gmdistribution)	Probability density function for Gaussian mixture distribution
posterior (gmdistribution)	Posterior probabilities of components
random (gmdistribution)	Random numbers from Gaussian mixture distribution

Model Assessment

confusionmat	Confusion matrix
crossval	Loss estimate using cross-validation
cvpartition	Create cross-validation partition for data
repartition (cvpartition)	Repartition data for cross-validation
test (cvpartition)	Test indices for cross-validation
training (cvpartition)	Training indices for cross-validation

Classification

Classification Plots

perfcurve	Compute Receiver Operating Characteristic (ROC) curve or other performance curve for classifier output
view (classregtree)	Plot tree

Discriminant Analysis

classify	Discriminant analysis
mahal	Mahalanobis distance

Classification Trees

catsplit (classregtree)	Categorical splits used for branches in decision tree
children (classregtree)	Child nodes
classcount (classregtree)	Class counts
classprob (classregtree)	Class probabilities
classregtree	Construct classification and regression trees
cutcategories (classregtree)	Cut categories

cutpoint (classregtree)	Returns decision tree cut point values
cuttype (classregtree)	Cut types
cutvar (classregtree)	Cut variable names
eval (classregtree)	Predicted responses
isbranch (classregtree)	Test node for branch
nodeerr (classregtree)	Return vector of node errors
nodeprob (classregtree)	Node probabilities
nodesize (classregtree)	Return node size
numnodes (classregtree)	Number of nodes
parent (classregtree)	Parent node
prune (classregtree)	Prune tree
risk (classregtree)	Node risks
test (classregtree)	Error rate
type (classregtree)	Tree type
varimportance (classregtree)	Compute embedded estimates of input feature importance
view (classregtree)	Plot tree

Naive Bayes Classification

fit (NaiveBayes)	Create Naive Bayes classifier object by fitting training data
NaiveBayes	Create NaiveBayes object
posterior (NaiveBayes)	Compute posterior probability of each class for test data
predict (NaiveBayes)	Predict class label for test data

Ensemble Methods

append (TreeBagger)	Append new trees to ensemble
combine (CompactTreeBagger)	Combine two ensembles
compact (TreeBagger)	Compact ensemble of decision trees
CompactTreeBagger	Create CompactTreeBagger object
error (CompactTreeBagger)	Error (misclassification probability or MSE)
error (TreeBagger)	Error (misclassification probability or MSE)
fillProximities (TreeBagger)	Proximity matrix for training data
growTrees (TreeBagger)	Train additional trees and add to ensemble
margin (CompactTreeBagger)	Classification margin
margin (TreeBagger)	Classification margin
mdsProx (CompactTreeBagger)	Multidimensional scaling of proximity matrix
mdsProx (TreeBagger)	Multidimensional scaling of proximity matrix
meanMargin (CompactTreeBagger)	Mean classification margin
meanMargin (TreeBagger)	Mean classification margin
oobError (TreeBagger)	Out-of-bag error
oobMargin (TreeBagger)	Out-of-bag margins
oobMeanMargin (TreeBagger)	Out-of-bag mean margins
oobPredict (TreeBagger)	Ensemble predictions for out-of-bag observations
outlierMeasure (CompactTreeBagger)	Outlier measure for data
predict (CompactTreeBagger)	Predict response
predict (TreeBagger)	Predict response

proximity (CompactTreeBagger)	Proximity matrix for data
SetDefaultYfit (CompactTreeBagger)	Set default value for `predict`
TreeBagger	Create ensemble of bagged decision trees

Hidden Markov Models

hmmdecode	Hidden Markov model posterior state probabilities
hmmestimate	Hidden Markov model parameter estimates from emissions and states
hmmgenerate	Hidden Markov model states and emissions
hmmtrain	Hidden Markov model parameter estimates from emissions
hmmviterbi	Hidden Markov model most probable state path

Design of Experiments

DOE Plots

interactionplot	Interaction plot for grouped data
maineffectsplot	Main effects plot for grouped data
multivarichart	Multivari chart for grouped data
rsmdemo	Interactive response surface demonstration
rstool	Interactive response surface modeling

Full Factorial Designs

ff2n	Two-level full factorial design
fullfact	Full factorial design

Fractional Factorial Designs

fracfact	Fractional factorial design
fracfactgen	Fractional factorial design generators

Response Surface Designs

bbdesign	Box-Behnken design
ccdesign	Central composite design

D-Optimal Designs

candexch	Candidate set row exchange
candgen	Candidate set generation
cordexch	Coordinate exchange
daugment	D-optimal augmentation
dcovary	D-optimal design with fixed covariates
rowexch	Row exchange
rsmdemo	Interactive response surface demonstration

Latin Hypercube Designs

lhsdesign	Latin hypercube sample
lhsnorm	Latin hypercube sample from normal distribution

Quasi-Random Designs

addlistener (qrandstream)	Add listener for event
delete (qrandstream)	Delete handle object
end (qrandset)	Last index in indexing expression for point set
eq (qrandstream)	Test handle equality
findobj (qrandstream)	Find objects matching specified conditions
findprop (qrandstream)	Find property of MATLAB handle object
ge (qrandstream)	Greater than or equal relation for handles
gt (qrandstream)	Greater than relation for handles
haltonset	Construct Halton quasi-random point set
isvalid (qrandstream)	Test handle validity
le (qrandstream)	Less than or equal relation for handles
length (qrandset)	Length of point set
lt (qrandstream)	Less than relation for handles
ndims (qrandset)	Number of dimensions in matrix
ne (qrandstream)	Not equal relation for handles
net (qrandset)	Generate quasi-random point set
notify (qrandstream)	Notify listeners of event
qrand (qrandstream)	Generate quasi-random points from stream
qrandset	Create categorical array
qrandstream	Construct quasi-random number stream

rand (qrandstream)	Generate quasi-random points from stream
reset (qrandstream)	Reset state
scramble (qrandset)	Scramble quasi-random point set
size (qrandset)	Number of dimensions in matrix
sobolset	Construct Sobol quasi-random point set

Statistical Process Control

SPC Plots

capaplot	Process capability plot
controlchart	Shewhart control charts
histfit	Histogram with normal fit
normspec	Normal density plot between specifications

SPC Functions

capability	Process capability indices
controlrules	Western Electric and Nelson control rules
gagerr	Gage repeatability and reproducibility study

GUIs

aoctool	Interactive analysis of covariance
dfittool	Interactive distribution fitting
disttool	Interactive density and distribution plots
fsurfht	Interactive contour plot
polytool	Interactive polynomial fitting
randtool	Interactive random number generation
regstats	Regression diagnostics
robustdemo	Interactive robust regression
rsmdemo	Interactive response surface demonstration
rstool	Interactive response surface modeling
surfht	Interactive contour plot

Utilities

16

Class Reference

Data Organization

In this section...
"Categorical Arrays" on page 16-2
"Dataset Arrays" on page 16-2

Categorical Arrays

categorical	Arrays for categorical data
nominal	Arrays for nominal categorical data
ordinal	Arrays for ordinal categorical data

Dataset Arrays

dataset	Arrays for statistical data

Probability Distributions

Distribution Objects

ProbDist	Object representing probability distribution
ProbDistKernel	Object representing nonparametric probability distribution defined by kernel smoothing
ProbDistParametric	Object representing parametric probability distribution
ProbDistUnivKernel	Object representing univariate kernel probability distribution
ProbDistUnivParam	Object representing univariate parametric probability distribution

Quasi-Random Numbers

haltonset	Halton quasi-random point sets
qrandset	Quasi-random point sets
qrandstream	Quasi-random number streams
sobolset	Sobol quasi-random point sets

Piecewise Distributions

paretotails   Empirical distributions with Pareto tails

piecewisedistribution   Piecewise-defined distributions

Regression Analysis

In this section...
"Regression Trees" on page 16-4
"Ensemble Method Classes" on page 16-4

Regression Trees

classregtree   Classification and regression trees

Ensemble Method Classes

CompactTreeBagger   Compact ensemble of decision trees grown by bootstrap aggregation

TreeBagger   Bootstrap aggregation for ensemble of decision trees

Gaussian Mixture Models

gmdistribution   Gaussian mixture models

Model Assessment

cvpartition	Data partitions for cross-validation

Classification

In this section...
"Classification Trees" on page 16-5
"Naive Bayes Classification " on page 16-5
"Ensemble Method Classes" on page 16-5

Classification Trees

classregtree	Classification and regression trees

Naive Bayes Classification

NaiveBayes	Naive Bayes classifier

Ensemble Method Classes

CompactTreeBagger	Compact ensemble of decision trees grown by bootstrap aggregation
TreeBagger	Bootstrap aggregation for ensemble of decision trees

Quasi-Random Design of Experiments

haltonset	Halton quasi-random point sets
qrandset	Quasi-random point sets
qrandstream	Quasi-random number streams
sobolset	Sobol quasi-random point sets

Functions — Alphabetical List

addedvarplot

Purpose	Added-variable plot

Syntax

```
addedvarplot(X,y,num,inmodel)
addedvarplot(X,y,num,inmodel,stats)
```

Description addedvarplot(X,y,num,inmodel) displays an added variable plot using the predictive terms in X, the response values in y, the added term in column num of X, and the model with current terms specified by inmodel. X is an n-by-p matrix of n observations of p predictive terms. y is vector of n response values. num is a scalar index specifying the column of X with the term to be added. inmodel is a logical vector of p elements specifying the columns of X in the current model. By default, all elements of inmodel are false.

Note addedvarplot automatically includes a constant term in all models. Do not enter a column of 1s directly into X.

---

addedvarplot(X,y,num,inmodel,stats) uses the stats output from the stepwisefit function to improve the efficiency of repeated calls to addedvarplot. Otherwise, this syntax is equivalent to the previous syntax.

Added variable plots are used to determine the unique effect of adding a new term to a multilinear model. The plot shows the relationship between the part of the response unexplained by terms already in the model and the part of the new term unexplained by terms already in the model. The "unexplained" parts are measured by the residuals of the respective regressions. A scatter of the residuals from the two regressions forms the added variable plot.

In addition to the scatter of residuals, the plot produced by addedvarplot shows 95% confidence intervals on predictions from the fitted line. The fitted line has intercept zero because, under typical linear model assumptions, both of the plotted variables have mean zero. The slope of the fitted line is the coefficient that the new term would have if it were added to the model with terms inmodel.

Added variable plots are sometimes known as partial regression leverage plots.

Examples   Load the data in `hald.mat`, which contains observations of the heat of reaction of various cement mixtures:

```
load hald
whos
  Name          Size     Bytes   Class   Attributes

  Description   22x58    2552     char
  hald          13x5      520     double
  heat          13x1      104     double
  ingredients   13x4      416     double
```

Create an added variable plot to investigate the addition of the third column of `ingredients` to a model consisting of the first two columns:

```
inmodel = [true true false false];
addedvarplot(ingredients,heat,3,inmodel)
```

addedvarplot
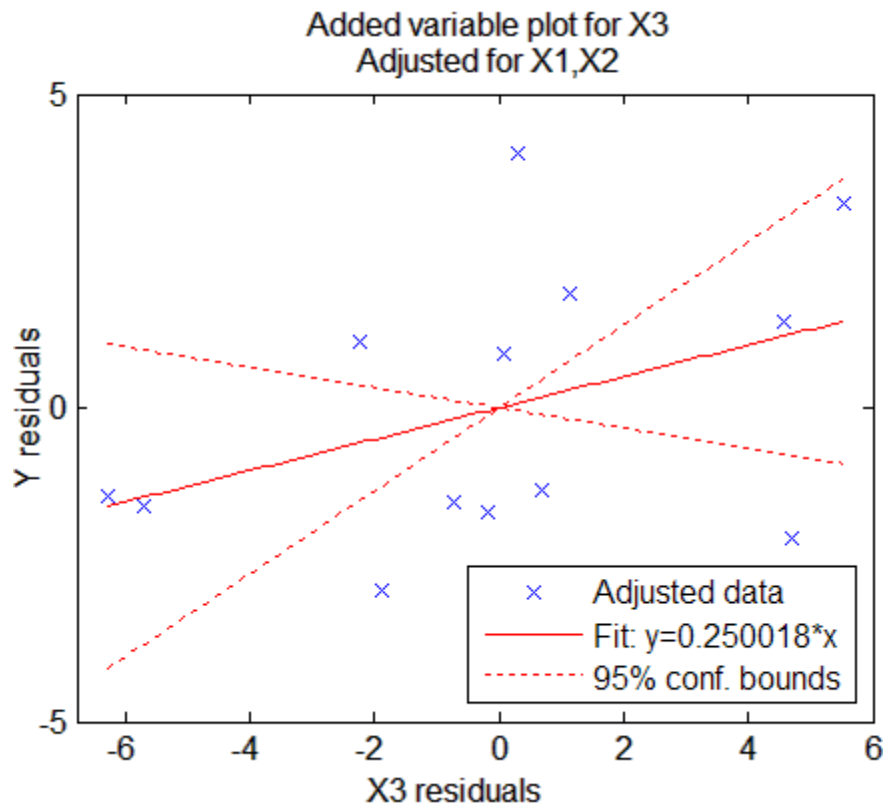
The wide scatter and the low slope of the fitted line are evidence against
the statistical significance of adding the third column to the model.

See Also stepwisefit, stepwise

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 2-D character matrix that specifies the levels to add. `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`, `getlabels`, `islevel`, `mergelevels`, `reorderlevels`

Purpose   Add listener for event

Syntax
```
el = addlistener(hsource, 'eventname', callback)
el = addlistener(hsource, property, 'eventname', callback)
```

Description   `el = addlistener(hsource, 'eventname', callback)` creates a listener for the event named `eventname`, the source of which is handle object `hsource`. If `hsource` is an array of source handles, the listener responds to the named event on any handle in the array. `callback` is a function handle that is invoked when the event is triggered.

`el = addlistener(hsource, property, 'eventname', callback)` adds a listener for a property event. `eventname` must be one of the strings `'PreGet'`, `'PostGet'`, `'PreSet'`, and `'PostSet'`. `property` must be either a property name or cell array of property names, or a `meta.property` or array of `meta.property`. The properties must belong to the class of `hsource`. If `hsource` is scalar, `property` can include dynamic properties.

For all forms, `addlistener` returns an `event.listener`. To remove a listener, delete the object returned by `addlistener`. For example, `delete(el)` calls the handle class delete method to remove the listener and delete it from the workspace.

See Also   `delete`, `dynamicprops`, `event.listener`, `events`, `meta.property`, `notify`, `qrandstream`, `reset`

gmdistribution.AIC property

Purpose   Akaike Information Criterion

Description The Akaike Information Criterion: `2*NlogL+2*m`, where `m` is the number of estimated parameters.

> **Note** This property applies only to gmdistribution objects constructed with `fit`.

Purpose   Andrews plot

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

$$f(t) = X(i,1)/\sqrt{2} + X(i,2)\sin(2\pi t) + X(i,3)\cos(2\pi t) + \ldots$$

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 2-33.)

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)
```
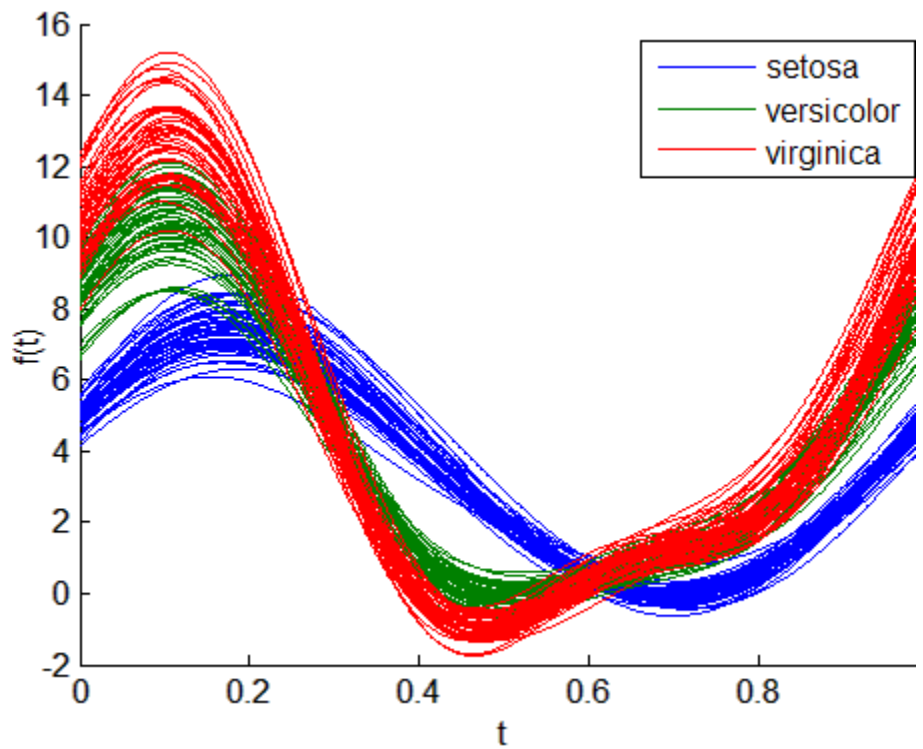
Plot only the median and quartiles of each group:

```
andrewsplot(meas,'group',species,'quantile',.25)
```

andrewsplot
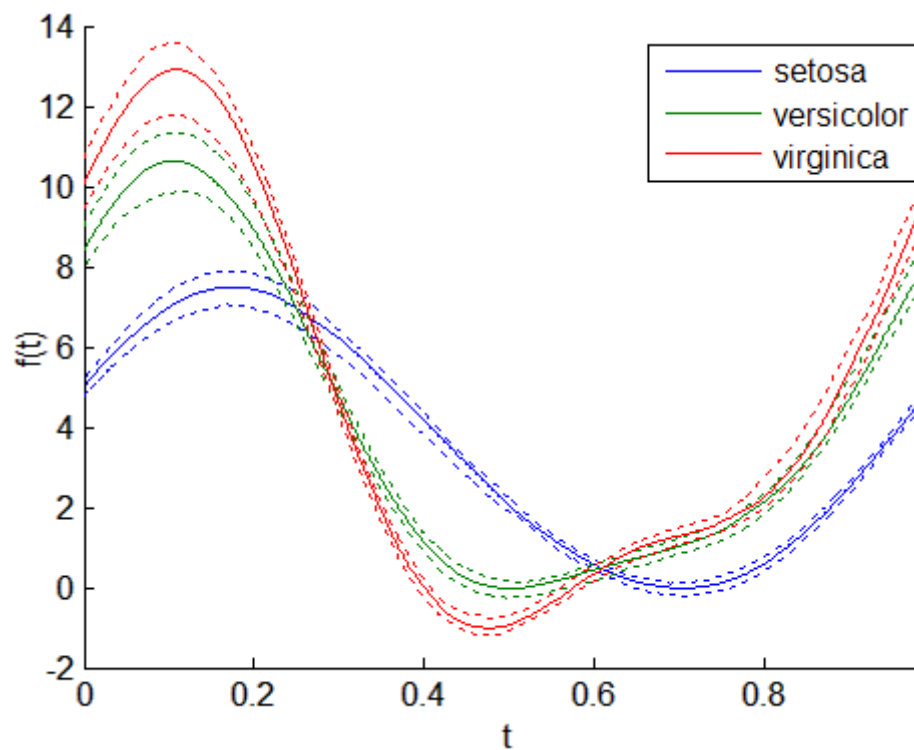
See Also   "Grouped Data" on page 2-33

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                              ▼
```
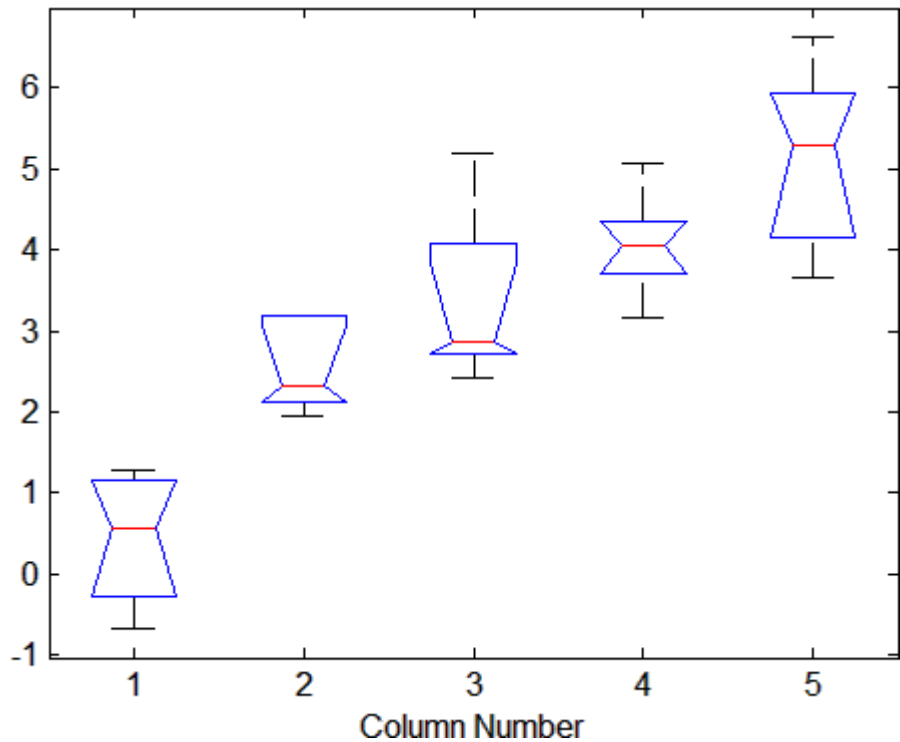
The very small *p*-value indicates that differences between column means are highly significant. The probability of this outcome under the null hypothesis (that samples drawn from the same population would have means differing by the amounts seen in X) is equal to the *p*-value.

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			
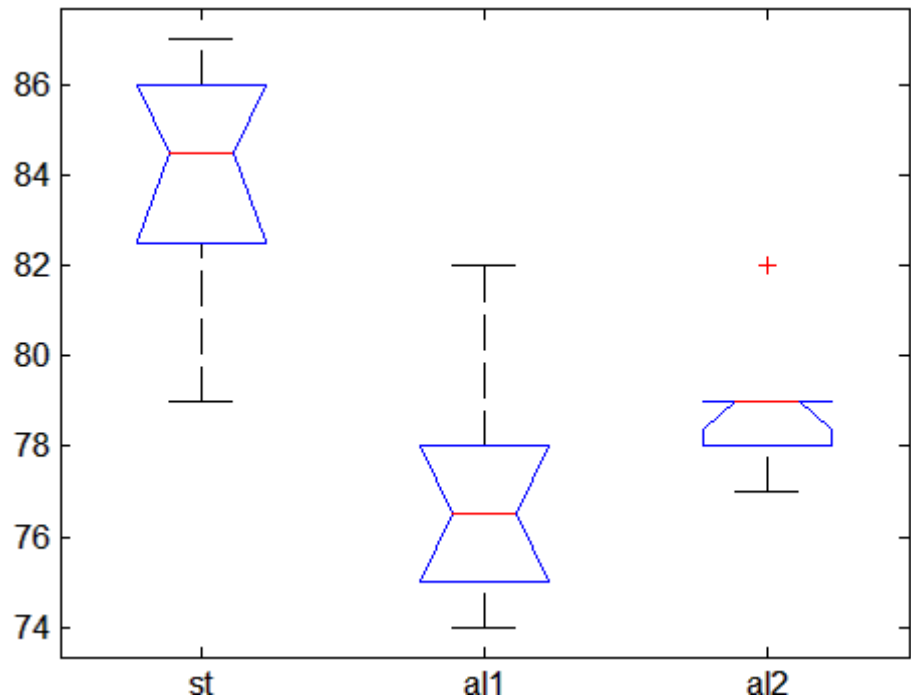
The *p*-value suggests rejection of the null hypothesis. The box plot shows that steel beams deflect more than beams made of the more expensive alloys.

References   [1] Hogg, R. V., and J. Ledolter. *Engineering Statistics*. New York: MacMillan, 1987.

See Also   "Grouped Data" on page 2-33

anova2, anovan, boxplot, manova1, multcompare

anova2

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 must 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.

$$\begin{matrix} & A=1 & A=2 \end{matrix}$$

$$\begin{bmatrix} x_{111} & x_{121} \\ x_{112} & x_{122} \\ x_{211} & x_{221} \\ x_{212} & x_{222} \\ x_{311} & x_{321} \\ x_{312} & x_{322} \end{bmatrix} \begin{matrix} \Big\} B=1 \\ \Big\} B=2 \\ \Big\} B=3 \end{matrix}$$

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)).

References [1] Hogg, R. V., and J. Ledolter. *Engineering Statistics*. New York: MacMillan, 1987.

See Also anova1, anovan

anovan

Purpose *N*-way analysis of variance

Syntax
```
p = anovan(y,group)
p = anovan(y,group,param1,val1,param2,val2,...)
[p,table] = anovan(...)
[p,table,stats] = anovan(...)
[p,table,stats,terms] = anovan(...)
```

Description p = anovan(y,group) performs multiway (n-way) analysis of variance (ANOVA) for testing the effects of multiple factors (grouping variables) on the mean of the vector y. 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 17-28Example of Three-Way ANOVA.

p = anovan(y,group,*param1*,val1,*param2*,val2,...) specifies one or more of the parameter name/value pairs described in the following table.

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	Value
`'display'`	`'on'` displays an ANOVA table (the default) `'off'` omits the display
`'model'`	The type of model used. See "Model Type" on page 17-26 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 "ANOVA with Random Effects" on page 7-19 for an example of how to use `'random'`.
`'sstype'`	`1`, `2`, `3` (default), or `h` specifies the type of sum of squares. See "Sum of Squares" on page 17-27 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 "ANOVA with Random Effects" on page 7-19 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 17-31The Stats Structure 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(y,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 $\leq$ N), `anovan` computes all interaction levels through the kth 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.

Matrix Row	ANOVA Term
[1 0 0]	Main term A

Matrix Row	ANOVA Term
[0 1 0]	Main term B
[0 0 1]	Main term C
[1 1 0]	Interaction term AB
[1 0 1]	Interaction term AC
[0 1 1]	Interaction term BC
[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(y,group,'sstype',type)
```

This syntax computes the ANOVA using the type of sum of squares specified by *type*, which can be 1, 2, 3, or h. While the numbers 1 – 3 designate Type 1, Type 2, or Type 3 sum of squares, respectively, h represents a hierarchical model similar to type 2, but with continuous as well as categorical factors used to determine the hierarchy of terms. The default value is 3. For a model containing main effects but no interactions, 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 $R(\cdot)$ 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 Sum of Squares	Type 2 Sum of Squares	Type 3 Sum of Squares
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.
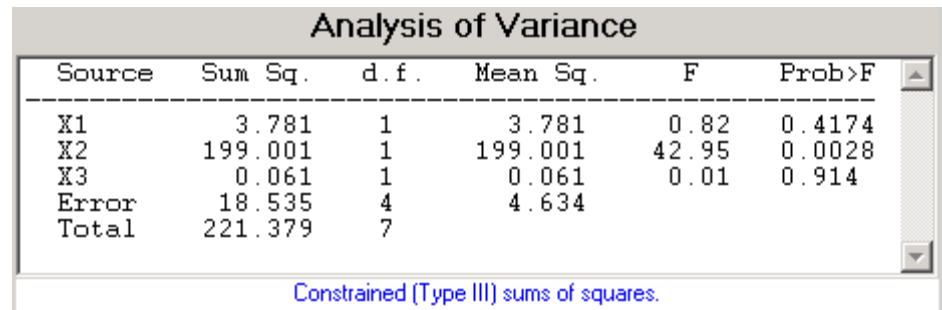
anovan also displays a figure showing the standard ANOVA table, which by default divides the variability of the data in x into

- 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:

Analysis of Variance

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.

Analysis of Variance

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:

Field	Description
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:

Field	Description
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 7-10 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.

The data in `carbig.mat` gives measurements on 406 cars. Use `anonvan` to study how the mileage depends on where and when the cars were made:

```
load carbig
```

```
p = anovan(MPG,{org when},'model',2,'sstype',3,...
           'varnames',{'Origin';'Mfg date'})
p =
      0
      0
    0.3059
```

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.

The *p*-value for the interaction term is not small, indicating little evidence that the effect of the year or manufacture (when) depends on where the car was made (org). The linear effects of those two factors, however, are significant.

References [1] Hogg, R. V., and J. Ledolter. *Engineering Statistics*. New York: MacMillan, 1987.

See Also "Grouped Data" on page 2-33

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`.

Examples   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 analysis of covariance

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 2-33.) 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 "Analysis of Covariance Tool" on page 7-27.

`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).

Examples   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   "Grouped Data" on page 2-33

anova1, multcompare, polytool

Purpose   Append new trees to ensemble

Syntax   B = append(B,other)

Description   B = append(B,other) appends the trees from the other ensemble to those in B. This method checks for consistency of the X and Y properties of the two ensembles, as well as consistency of their compact objects and out-of-bag indices, before appending the trees. The output ensemble B takes training parameters such as FBoot, Prior, Cost, and other from the B input. There is no attempt to check if these training parameters are consistent between the two objects.

See Also   CompactTreeBagger.combine

ProbDistKernel.BandWidth property

Purpose   Read-only value specifying bandwidth of kernel smoothing function
for ProbDistKernel object

Description   `BandWidth` is a read-only property of the `ProbDistKernel` class.
`BandWidth` is a value specifying the width of the kernel smoothing
function used to compute a nonparametric estimate of the probability
distribution when creating a ProbDistKernel object.

Values   For a distribution specified to cover only the positive numbers or only
a finite interval, the data are transformed before the kernel density is
applied, and the bandwidth is on the scale of the transformed data.

Use this information to view and compare the width of the kernel
smoothing function used to create distributions.

See Also   `ksdensity`

Purpose   Bartlett's test

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.

Examples   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   Box-Behnken design

Syntax
```
dBB = bbdesign(n)
[dBB,blocks] = bbdesign(n)
[...] = bbdesign(n,param1,val1,param2,val2,...)
```

Description   `dBB = bbdesign(n)` generates a Box-Behnken design for `n` factors. `n` must be an integer 3 or larger. The output matrix `dBB` is *m*-by-`n`, where *m* is the number of runs in the design. Each row represents one run, with settings for all factors represented in the columns. Factor values are normalized so that the cube points take values between `-1` and `1`.

`[dBB,blocks] = bbdesign(n)` requests a blocked design. The output `blocks` is an *m*-by-1 vector of block numbers for each run. Blocks indicate runs that are to be measured under similar conditions to minimize the effect of inter-block differences on the parameter estimates.

`[...] = bbdesign(n,param1,val1,param2,val2,...)` specifies additional parameter/value pairs for the design. Valid parameters and their values are listed in the following table.

Parameter	Description	Values
`'center'`	Number of center points.	Integer. The default depends on n.
`'blocksize'`	Maximum number of points per block.	Integer. The default is Inf.

Examples   The following creates a 3-factor Box-Behnken design:

```
dBB = bbdesign(3)
dBB =
    -1    -1     0
    -1     1     0
     1    -1     0
```

```
 1     1     0
-1     0    -1
-1     0     1
 1     0    -1
 1     0     1
 0    -1    -1
 0    -1     1
 0     1    -1
 0     1     1
 0     0     0
 0     0     0
 0     0     0
```

The center point is run 3 times to allow for a more uniform estimate of the prediction variance over the entire design space.

Visualize the design as follows:

```
plot3(dBB(:,1),dBB(:,2),dBB(:,3),'ro',...
        'MarkerFaceColor','b')
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];
Y = [-1 -1 1 -1 -1 -1 1 -1 1 -1 1 -1; ...
     1 -1 1 1 1 -1 1 1 1 -1 1 -1];
Z = [1 1 1 1 -1 -1 -1 -1 -1 -1 -1 -1; ...
     1 1 1 1 -1 -1 -1 -1 1 1 1 1];
line(X,Y,Z,'Color','b')
axis square equal
```

bbdesign
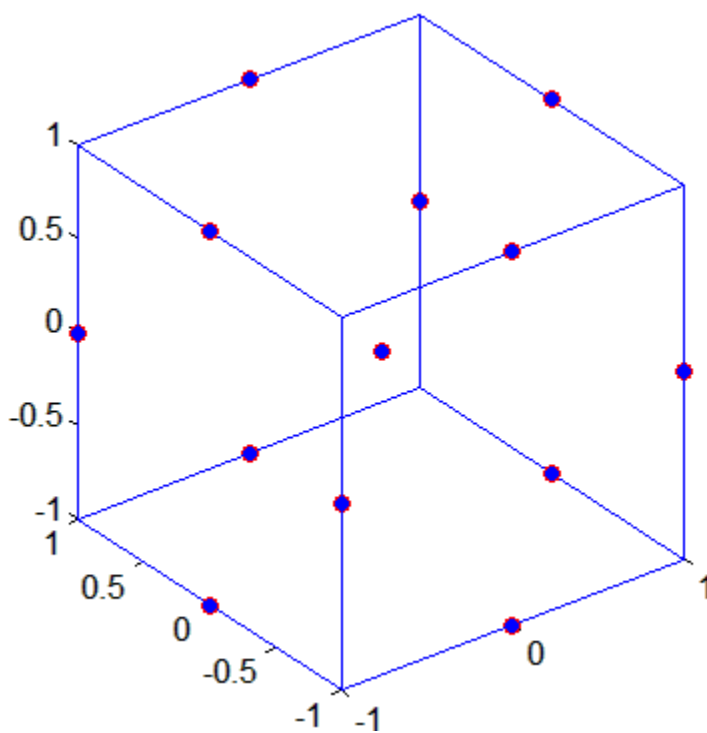
See Also   ccdesign

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   cdf, betapdf, betainv, betastat, betafit, betalike, betarnd

betafit

Purpose   Beta parameter estimates

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.

Examples   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
```

References   [1] Hahn, Gerald J., and S. S. Shapiro. *Statistical Models in Engineering.* Hoboken, NJ: John Wiley & Sons, Inc., 1994, p. 95.

See Also   mle, betalike, betapdf, betacdf, betainv, betastat, betarnd

betainv

Purpose   Beta inverse 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 are
all 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   `icdf`, `betacdf`, `betapdf`, `betastat`, `betafit`, `betalike`, `betarnd`

betalike

Purpose	Beta negative log-likelihood
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.

Examples 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, betapdf, betacdf, betainv, betastat, betarnd

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 `pdf`, `betacdf`, `betainv`, `betastat`, `betafit`, `betalike`, `betarnd`

betarnd

Purpose	Beta random numbers

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.

Examples
```
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   random, betapdf, betacdf, betainv, betastat, betafit, betalike

betastat

Purpose	Beta mean and variance
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   `betapdf`, `betacdf`, `betainv`, `betafit`, `betalike`, `betarnd`

Purpose   Bayes Information Criterion

Description   The Bayes Information Criterion: `2*NlogL+2*m*log(n)`, where `n` is the number of observations and `m` is the number of estimated parameters.

binocdf

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 are all 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 a given pair of parameters n and p is

$$y = F(x|n, p) = \sum_{i=0}^{x} \binom{n}{i} p^i q^{(n-i)} I_{(0, 1, ..., n)}(i)$$

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 $I_{(0, 1, ..., n)}(i)$ ensures that x only adopts values of $0, 1, ..., n$.

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

cdf, binopdf, binoinv, binostat, binofit, binornd

Purpose   Binomial parameter estimates

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 Statistics Toolbox
                functions 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.

binofit

Examples
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.

References
[1] Johnson, N. L., S. Kotz, and A. W. Kemp. *Univariate Discrete Distributions*. Hoboken, NJ: Wiley-Interscience, 1993.

See Also
mle, binopdf, binocdf, binoinv, binostat, binornd

17-60

Purpose   Binomial inverse 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   `icdf`, `binocdf`, `binopdf`, `binostat`, `binofit`, `binornd`

binopdf

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

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

where $q = 1 - p$. 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 $I_{(0,1,...,n)}(x)$ 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   pdf, binocdf, binoinv, binostat, binofit, binornd

binornd

Purpose	Binomial random numbers

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.

Examples
```
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   random, binopdf, binocdf, binoinv, binostat, binofit

binostat

Purpose	Binomial mean and variance

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   binopdf, binocdf, binoinv, binofit, binornd

Purpose   Biplot

Syntax
```
biplot(coefs)
biplot(coefs,...,'Scores',scores)
biplot(coefs,...,'VarLabels',varlabels)
biplot(coefs,...,'Scores',scores,'ObsLabels',obslabels)
biplot(coefs,...,'Positive',true)
biplot(coeffs,...,PropertyName,PropertyValue,...)
h = biplot(coefs,...)
```

Description   `biplot(coefs)` creates a biplot of the coefficients in the matrix `coefs`. The biplot is 2-D if `coefs` has two columns or 3-D 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(coefs,...,'Positive',true)` restricts the biplot to the positive quadrant (in 2-D) or octant (in 3-D).

biplot

biplot(coefs,...,'Positive',false) (the default) makes
the biplot over the range +/- max(coefs(:)) for all coordinates.

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.

Examples   Perform a principal component analysis of the data in carsmall.mat:

```
load carsmall
x = [Acceleration Displacement Horsepower MPG Weight];
x = x(all(~isnan(x),2),:);

[coefs,score] = princomp(zscore(x));
```

View the data and the original variables in the space of the first three
principal components:

```
vbls = {'Accel','Disp','HP','MPG','Wgt'};
biplot(coefs(:,1:3),'scores',score(:,1:3),...
                    'varlabels',vbls);
```
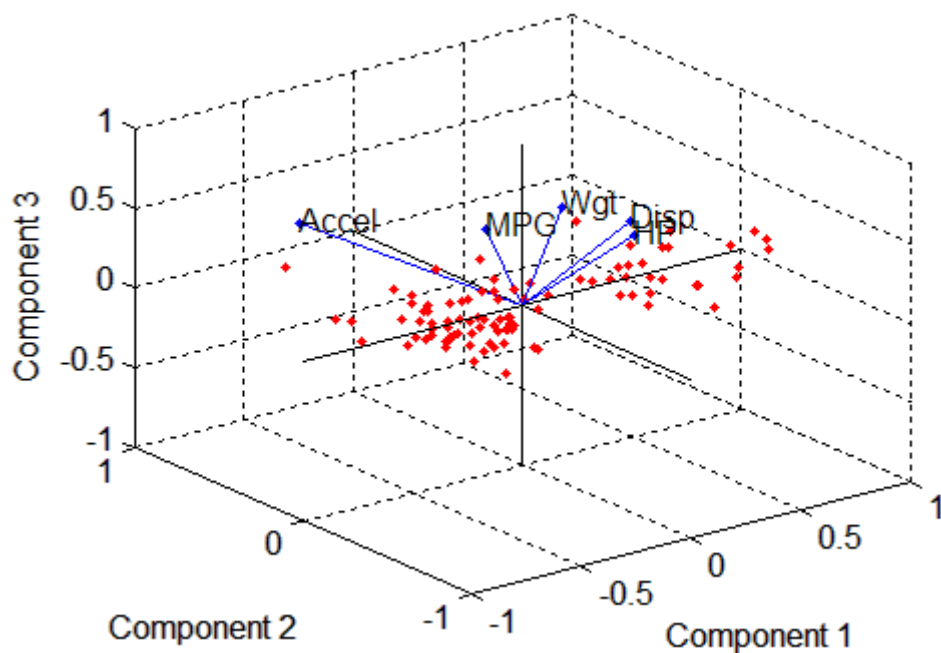
See Also   factoran, nnmf, princomp, pcacov, rotatefactors

bootci

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','student',
    'nbootstd',nbootstd)
ci = bootci(nboot,{bootfun,...},...,'type','student','stderr',
    stderr)
```

Description   `ci = bootci(nboot,bootfun,...)` computes the 95% BCa bootstrap
confidence interval of the statistic computed by the function `bootfun`.
`nboot` is a positive integer indicating the number of bootstrap samples
used in the computation. `bootfun` is a function handle to a function
returning a scalar. Additional input arguments to `bootci` are passed
as data inputs to `bootfun`. Data inputs are scalars, column vectors,
or matrices; vectors and matrices must have the same number of
rows. Scalar data inputs are passed to `bootfun` unchanged. Rows of
non-scalar data inputs are used to create bootstrap samples. `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`. `bootfun` and the data that `bootci`
passes to it are contained in a single cell array. `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, chosen from among the
following strings:

- `'normal'` — Normal approximated interval with bootstrapped bias
  and standard error.

- `'per'` — Basic percentile method.

- `'cper'` — Bias corrected percentile method.

- 'bca' — Bias corrected and accelerated percentile method. This is the default.

- 'student' — Studentized confidence interval.

```
ci =
bootci(nboot,{bootfun,...},...,'type','student','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','student','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. stderr takes the same arguments as bootfun and returns the standard error of the statistic computed by bootfun.

Examples   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
ci = bootci(2000,capable,y)           % BCa confidence interval
ci =
    0.8122
    1.2657

sci = bootci(2000,{capable,y},'type','student') % Studentized ci
sci =
    0.7739
    1.2707
```

See Also   bootstrp, jackknife

bootstrp

Purpose	Bootstrap sampling
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);
```
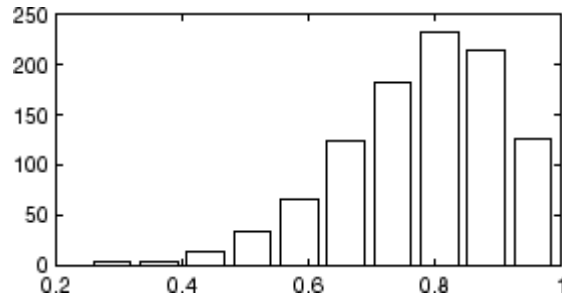

The histogram shows the variation of the correlation coefficient across all the bootstrap samples. The sample minimum is positive, indicating that the relationship between LSAT score and GPA is not accidental.

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);
```

17-74
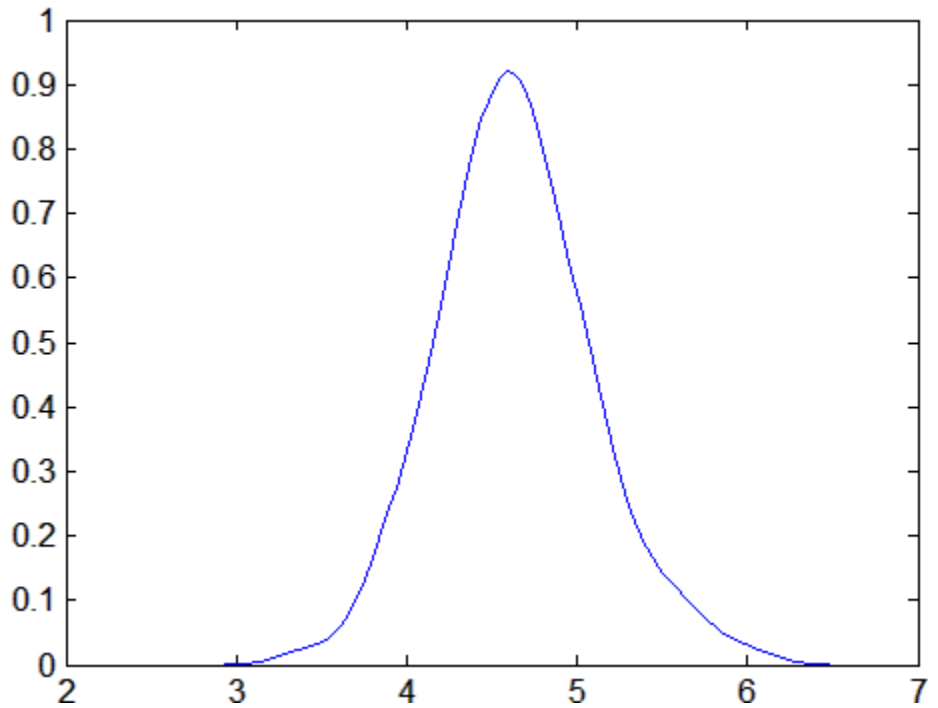
Bootstrapping More Than One Statistic

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')
```
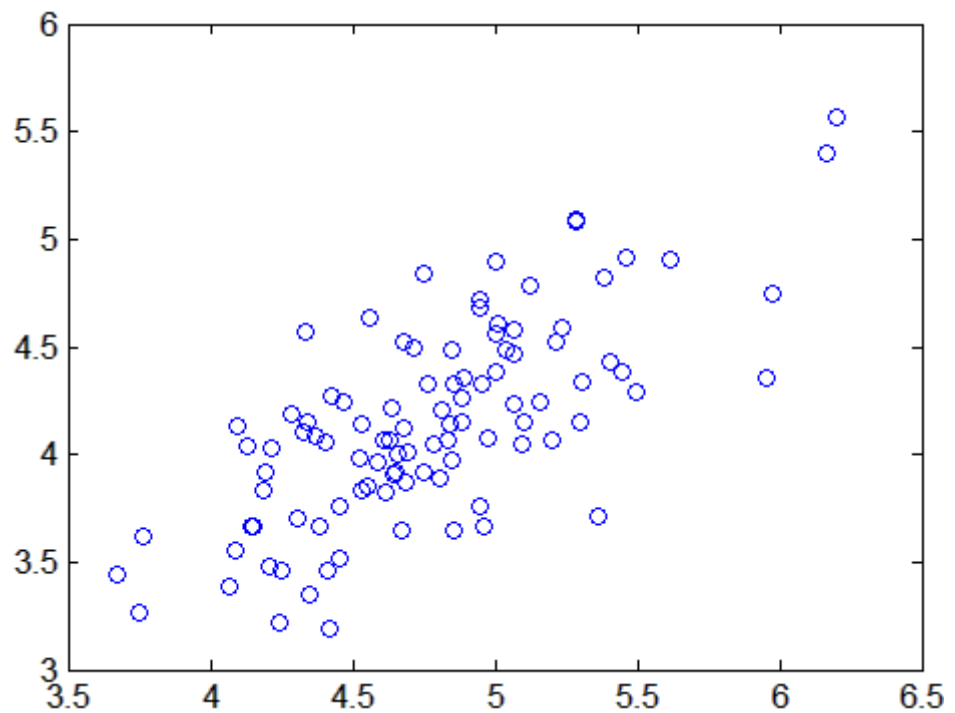
Bootstrapping a Regression Model

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));
```

Purpose   Box plot

Syntax   ```
                     boxplot(X)
                     boxplot(X,G)
                     boxplot(axes,X,...)
                     boxplot(...,param1,val1,param2,val2,...)
                     ```

**Description**      `boxplot(X)` produces a box plot of the data in X. If X is a matrix, there is
                     one box per column; if X is a vector, there is just one box. On each box,
                     the central mark is the median, the edges of the box are the 25th and
                     75th percentiles, the whiskers extend to the most extreme data points
                     not considered outliers, and outliers are plotted individually.

                     `boxplot(X,G)` specifies one or more grouping variables G, producing
                     a separate box for each set of X values sharing the same G value or
                     values (see "Grouped Data" on page 2-33). Grouping variables must
                     have one row per element of X, or one row per column of X. Specify a
                     single grouping variable in G using a vector, a character array, a cell
                     array of strings, or a vector categorical array; specify multiple grouping
                     variables in G using a cell array of these variable types, such as {G1 G2
                     G3}, or by using a matrix. If multiple grouping variables are used, they
                     must all be the same length. Groups that contain a NaN value or an
                     empty string in a grouping variable are omitted, and are not counted in
                     the number of groups considered by other parameters.

                     By default, character and string grouping variables are sorted in the
                     order they initially appear in the data, categorical grouping variables
                     are sorted by the order of their levels, and numeric grouping variables
                     are sorted in numeric order. To control the order of groups, do one of
                     the following:

                     • Use categorical variables in G and specify the order of their levels.

                     • Use the 'grouporder' parameter described below.

                     • Pre-sort your data.

                     `boxplot(axes,X,...)` creates the plot in the axes with handle axes.

boxplot(...,*param1*,*val1*,*param2*,*val2*,...) specifies optional parameter name/value pairs, as described in the following table.

| Parameter | Values |
|---|---|
| 'plotstyle' | • 'traditional' — Traditional box style. This is the default.<br><br>• 'compact' — Box style designed for plots with many groups. This style changes the defaults for some other parameters, as described in the following table. |
| 'boxstyle' | • 'outline' — Draws an unfilled box with dashed whiskers. This is the default.<br><br>• 'filled' — Draws a narrow filled box with lines for whiskers. |
| 'colorgroup' | One or more grouping variables, of the same type as permitted for G, specifying that the box color should change when the specified variables change. The default is [] for no box color change. |
| 'colors' | Colors for boxes, specified as a single color (such as 'r' or [1 0 0]) or multiple colors (such as 'rgbm' or a three-column matrix of RGB values). The sequence is replicated or truncated as required, so for example 'rb' gives boxes that alternate in color. The default when no 'colorgroup' is specified is to use the same color scheme for all boxes. The default when 'colorgroup' is specified is a modified hsv colormap. |
| 'datalim' | A two-element vector containing lower and upper limits, used by 'extrememode' to determine which points are extreme. The default is [-Inf Inf]. |

| Parameter | Values |
|---|---|
| `'extrememode'` | • `'clip'` — Moves data outside the `'datalim'` limits to the limit. This is the default. |
| | • `'compress'` — Evenly distributes data outside the `'datalim'` limits in a region just outside the limit, retaining the relative order of the points. |
| | A dotted line marks the limit if any points are outside it, and two gray lines mark the compression region if any points are compressed. Values at +/–`Inf` can be clipped or compressed, but `NaN` values still do not appear on the plot. Box notches are drawn to scale and may extend beyond the bounds if the median is inside the limit; they are not drawn if the median is outside the limits. |
| `'factordirection'` | • `'data'` — Arranges factors with the first value next to the origin. This is the default. |
| | • `'list'` — Arranges factors left-to-right if on the *x* axis or top-to-bottom if on the *y* axis. |
| | • `'auto'` — Uses `'data'` for numeric grouping variables and `'list'` for strings. |
| `'fullfactors'` | • `'off'` — One group for each unique row of `G`. This is the default. |
| | • `'on'` — Create a group for each possible combination of group variable values, including combinations that do not appear in the data. |

| Parameter | Values |
|---|---|
| 'factorseparator' | Specifies which factors should have their values separated by a grid line. The value may be 'auto' or a vector of grouping variable numbers. For example, [1 2] adds a separator line when the first or second grouping variable changes value. 'auto' is [] for one grouping variable and [1] for two or more grouping variables. The default is []. |
| 'factorgap' | Specifies an extra gap to leave between boxes when the corresponding grouping factor changes value, expressed as a percentage of the width of the plot. For example, with [3 1], the gap is 3% of the width of the plot between groups with different values of the first grouping variable, and 1% between groups with the same value of the first grouping variable but different values for the second. 'auto' specifies that boxplot should choose a gap automatically. The default is []. |
| 'grouporder' | Order of groups for plotting, specified as a cell array of strings. With multiple grouping variables, separate values within each string with a comma. Using categorical arrays as grouping variables is an easier way to control the order of the boxes. The default is [], which does not reorder the boxes. |
| 'jitter' | Maximum distance $d$ to displace outliers along the factor axis by a uniform random amount, in order to make duplicate points visible. A $d$ of 1 makes the jitter regions just touch between the closest adjacent groups. The default is 0. |

| Parameter | Values |
|---|---|
| `'labels'` | A character array, cell array of strings, or numeric vector of box labels. There may be one label per group or one label per X value. Multiple label variables may be specified via a numeric matrix or a cell array containing any of these types. |
| `'labelorientation'` | • `'inline'` — Rotates the labels to be vertical. This is the default when `'plotstyle'` is `'compact'`.<br><br>• `'horizontal'` — Leaves the labels horizontal. This is the default when `'plotstyle'` has the default value of `'traditional'`.<br><br>When the labels are on the *y* axis, both settings leave the labels horizontal. |
| `'labelverbosity'` | • `'all'` — Displays every label. This is the default.<br><br>• `'minor'` — Displays a label for a factor only when that factor has a different value from the previous group.<br><br>• `'majorminor'` — Displays a label for a factor when that factor or any factor major to it has a different value from the previous group. |
| `'medianstyle'` | • `'line'` — Draws a line for the median. This is the default.<br><br>• `'target'` — Draws a black dot inside a white circle for the median. |

| Parameter | Values |
|---|---|
| 'notch' | • 'on' — Draws comparison intervals using notches when 'plotstyle' is 'traditional', or triangular markers when 'plotstyle' is 'compact'.<br><br>• 'marker' — Draws comparison intervals using triangular markers.<br><br>• 'off' — Omits notches. This is the default.<br><br>Two medians are significantly different at the 5% significance level if their intervals do not overlap. Interval endpoints are the extremes of the notches or the centers of the triangular markers. When the sample size is small, notches may extend beyond the end of the box. |
| 'orientation' | • 'vertical' — Plots X on the *y* axis. This is the default.<br><br>• 'horizontal' — Plots X on the *x* axis. |
| 'outliersize' | Size of the marker used for outliers, in points. The default is 6 (6/72 inch). |
| 'positions' | Box positions specified as a numeric vector with one entry per group or X value. The default is 1:numGroups, where numGroups is the number of groups. |
| 'symbol' | Symbol and color to use for outliers, using the same values as the LineSpec parameter in plot. The default is 'r+'. If the symbol is omitted then the outliers are invisible; if the color is omitted then the outliers have the same color as their corresponding box. |

| Parameter | Values |
|-----------|--------|
| 'whisker' | Maximum whisker length $w$. The default is a $w$ of 1.5. Points are drawn as outliers if they are larger than $q_3 + w(q_3 - q_1)$ or smaller than $q_1 - w(q_3 - q_1)$, where $q_1$ and $q_3$ are the 25th and 75th percentiles, respectively. The default of 1.5 corresponds to approximately +/–2.7$\sigma$ and 99.3 coverage if the data are normally distributed. The plotted whisker extends to the *adjacent value*, which is the most extreme data value that is not an outlier. Set 'whisker' to 0 to give no whiskers and to make every point outside of $q_1$ and $q_3$ an outlier. |
| 'widths' | A scalar or vector of box widths for when 'boxstyle' is 'outline'. The default is half of the minimum separation between boxes, which is 0.5 when the 'positions' argument takes its default value. The list of values is replicated or truncated as necessary. |

When the 'plotstyle' parameter takes the value 'compact', the default values for other parameters are the listed in the following table.

| Parameter | Default when plotstyle is compact |
|-----------|-----------------------------------|
| 'boxstyle' | 'filled' |
| 'factorseparator' | 'auto' |
| 'factorgap' | 'auto' |
| 'jitter' | 0.5 |
| 'labelorientation' | 'inline' |
| 'labelverbosity' | 'majorminor' |
| 'medianstyle' | 'target' |

| Parameter | Default when `plotstyle` is `compact` |
|---|---|
| `'outliersize'` | 4 |
| `'symbol'` | `'o'` |

You can see data values and group names using the data cursor in the figure window. The cursor shows the original values of any points affected by the `'datalim'` parameter. You can label the group to which an outlier belongs using the `gname` function.

To modify graphics properties of a box plot component, use `findobj` with the `'Tag'` property to find the component's handle. `'Tag'` values for box plot components depend on parameter settings, and are listed in the table below.

| Parameter Settings | `Tag` Values |
|---|---|
| All settings | • `'Box'`<br>• `'Outliers'` |
| When `'plotstyle'` is `'traditional'` | • `'Median'`<br>• `'Upper Whisker'`<br>• `'Lower Whisker'`<br>• `'Upper Adjacent Value'`<br>• `'Lower Adjacent Value'` |
| When `'plotstyle'` is `'compact'` | • `'Whisker'`<br>• `'MedianOuter'`<br>• `'MedianInner'` |
| When `'notch'` is `'marker'` | • `'NotchLo'`<br>• `'NotchHi'` |

**Examples**      ## Example 1

Create a box plot of car mileage, grouped by country:

```
load carsmall
boxplot(MPG,Origin)
```


Example 2

Create 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')
```

boxplot

The difference between the medians of the two groups is approximately 1. Since the notches in the box plot do not overlap, you can conclude, with 95% confidence, that the true medians do differ.

The following figure shows the box plot for the 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)
```

Example 3

A `'plotstyle'` of `'compact'` is useful for large numbers of groups:

```
X = randn(100,25);
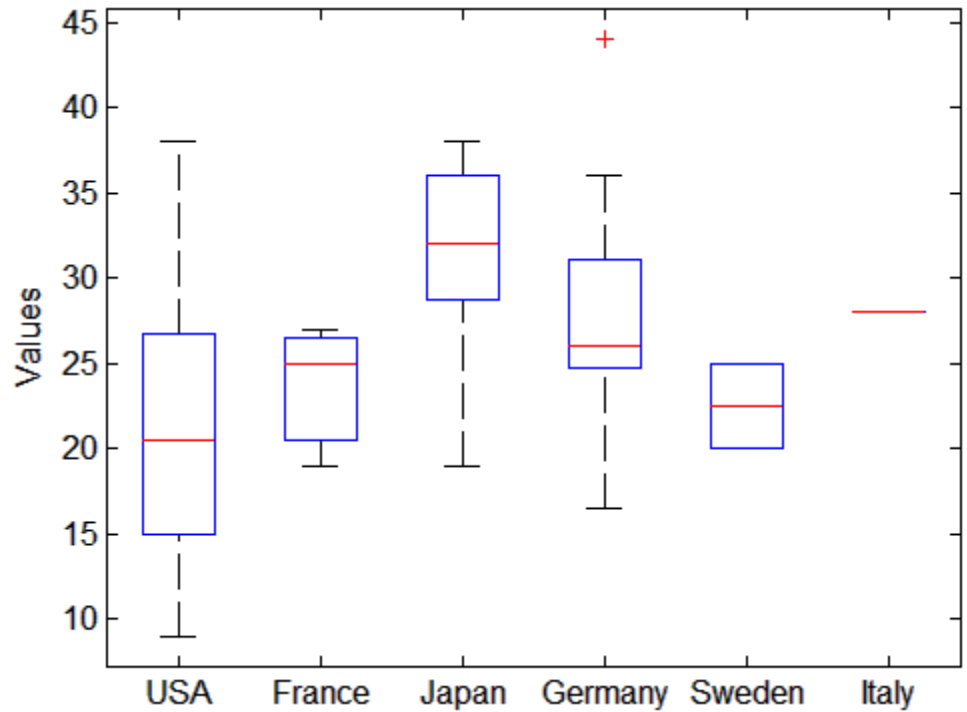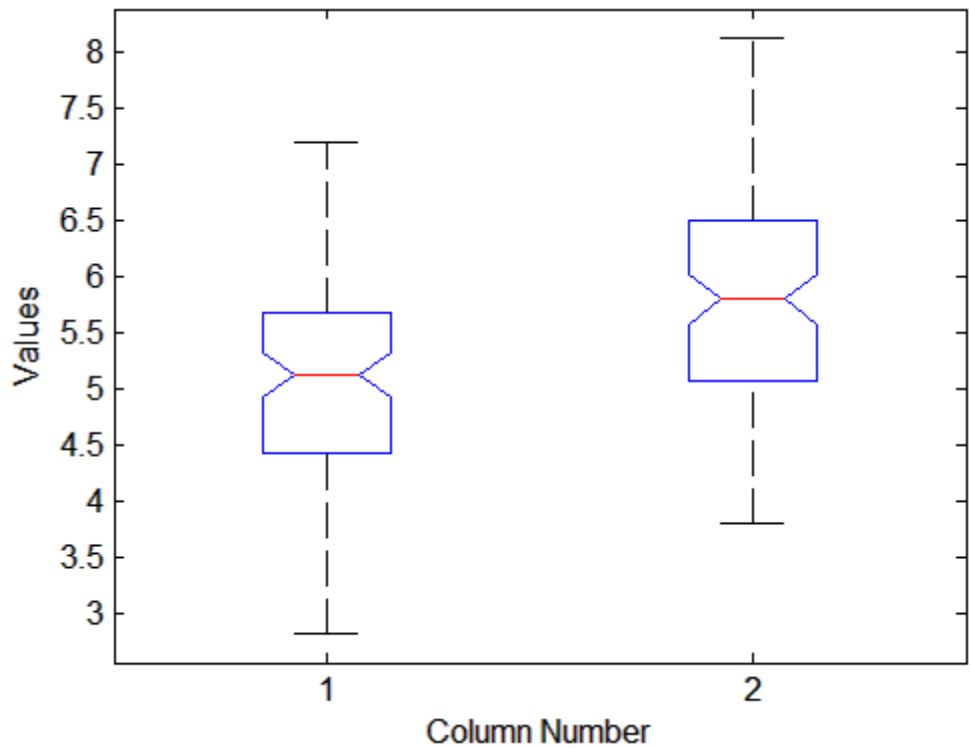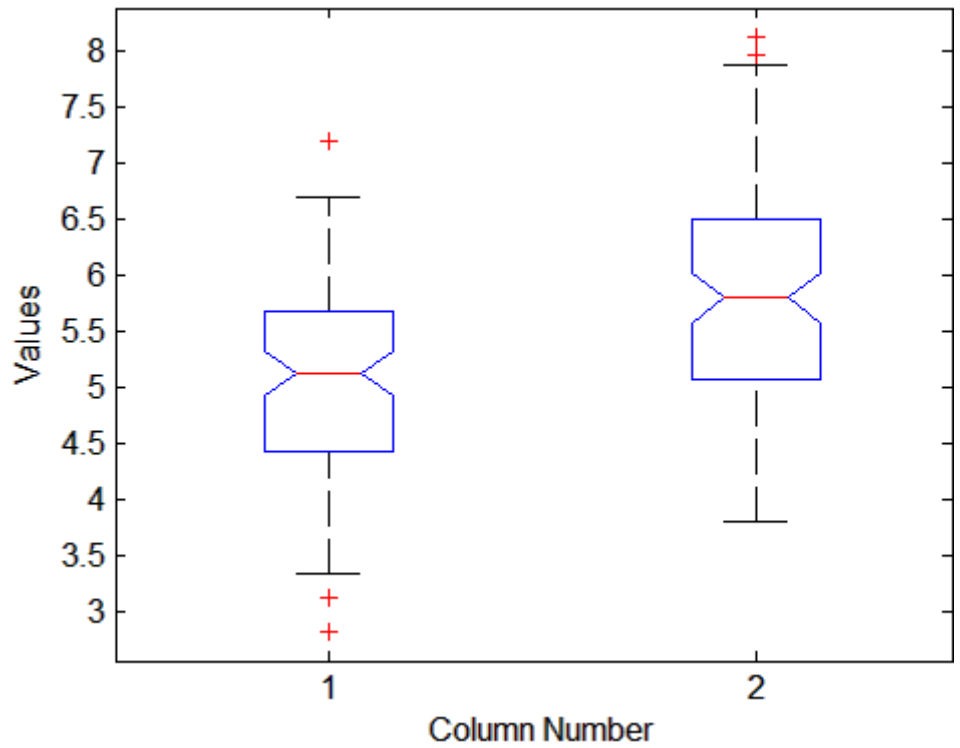
subplot(2,1,1)
boxplot(X)

subplot(2,1,2)
boxplot(X,'plotstyle','compact')
```

boxplot

References   [1] McGill, R., J. W. Tukey, and W. A. Larsen. "Variations of Boxplots." *The American Statistician*. Vol. 32, No. 1, 1978, pp. 12–16.

[2] Velleman, P.F., and D.C. Hoaglin. *Applications, Basics, and Computing of Exploratory Data Analysis*. Pacific Grove, CA: Duxbury Press, 1981.

[3] Nelson, L. S. "Evaluating Overlapping Confidence Intervals." *Journal of Quality Technology*. Vol. 21, 1989, pp. 140–141.

See Also   "Grouped Data" on page 2-33

anova1, kruskalwallis, multcompare

piecewisedistribution.boundary

Purpose   Piecewise distribution boundaries

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.

Examples   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, icdf, nsegments

candexch

Purpose Candidate set row exchange

Syntax

```
treatments = candexch(C,nruns)
treatments = candexch(...,param1,val1,param2,val2,...)
```

Description `treatments = candexch(C,nruns)` uses a row-exchange algorithm to select `treatments` from a candidate design matrix `C` to produce a *D*-optimal design with `nruns` runs. The columns of `C` represent model terms evaluated at candidate treatments. `treatments` is a vector of length `nruns` giving indices of the rows in `C` used in the *D*-optimal design. The function selects a starting design at random.

`treatments = candexch(...,param1,val1,param2,val2,...)` specifies additional parameter/value pairs for the design. Valid parameters and their values are listed in the following table.

Parameter	Value
`'display'`	Either `'on'` or `'off'` to control display of the iteration counter. The default is `'on'`.
`'init'`	Initial design as an `nruns`-by-*p* matrix, where *p* is the number of model terms. The default is a random subset of the rows of `C`.
`'maxiter'`	Maximum number of iterations. The default is `10`.
`'start'`	A matrix of treatments as a `nobs`-by-*p* matrix, where *p* is the number of model terms, specifying a set of `nobs` fixed treatments to include in the design. The default matrix is empty. `candexch` finds `nruns`-`nobs` additional rows to add to the `'start'` design. The parameter provides the same functionality as the `daugment` function, using a row-exchange algorithm rather than a coordinate-exchange algorithm.
`'tries'`	Number of times to try to generate a design from a new starting point. The algorithm uses random points for each try, except possibly the first. The default is `1`.

Note The rowexch function automatically generates a candidate set using candgen, and then creates a *D*-optimal design from that candidate set using candexch. Call candexch separately to specify your own candidate set to the row-exchange algorithm.

Examples The following example uses rowexch to generate a five-run design for a two-factor pure quadratic model using a candidate set that is produced internally:

```
dRE1 = rowexch(2,5,'purequadratic','tries',10)
dRE1 =
    -1     1
     0     0
     1    -1
     1     0
     1     1
```

The same thing can be done using candgen and candexch in sequence:

```
[dC,C] = candgen(2,'purequadratic') % Candidate set
dC =
    -1    -1
     0    -1
     1    -1
    -1     0
     0     0
     1     0
    -1     1
     0     1
     1     1
C =
     1    -1    -1     1     1
     1     0    -1     0     1
     1     1    -1     1     1
     1    -1     0     1     0
```

```
       1    0    0    0    0
       1    1    0    1    0
       1   -1    1    1    1
       1    0    1    0    1
       1    1    1    1    1
treatments = candexch(C,5,'tries',10) % D-opt subset
treatments =
     2
     1
     7
     3
     4
dRE2 = dC(treatments,:) % Display design
dRE2 =
     0   -1
    -1   -1
    -1    1
     1   -1
    -1    0
```

You can replace C in this example with a design matrix evaluated at your own candidate set. For example, suppose your experiment is constrained so that the two factors cannot have extreme settings simultaneously. The following produces a restricted candidate set:

```
constraint = sum(abs(dC),2) < 2; % Feasible treatments
my_dC = dC(constraint,:)
my_dC =
     0   -1
    -1    0
     0    0
     1    0
     0    1
```

Use the x2fx function to convert the candidate set to a design matrix:

```
my_C = x2fx(my_dC,'purequadratic')
my_C =
```

```
1    0   -1    0    1
1   -1    0    1    0
1    0    0    0    0
1    1    0    1    0
1    0    1    0    1
```

Find the required design in the same manner:

```
my_treatments = candexch(my_C,5,'tries',10) % D-opt subset
my_treatments =
     2
     4
     5
     1
     3
my_dRE = my_dC(my_treatments,:) % Display design
my_dRE =
    -1     0
     1     0
     0     1
     0    -1
     0     0
```

See Also   candgen, rowexch, cordexch, daugment, x2fx

candgen

Purpose   Candidate set generation

Syntax   dC = candgen(nfactors,*model*)
                 [dC,C] = candgen(nfactors,*model*)
                 [...] = candgen(...,*param1*,*val1*,*param2*,*val2*,...)

Description dC = candgen(nfactors,*model*) generates a candidate set dC of
                 treatments appropriate for estimating the parameters in the *model*
                 with nfactors factors. dC has nfactors columns and one row for each
                 candidate treatment. *model* is one of the following strings:

- 'linear' — Constant and linear terms. This is the default.

- 'interaction' — Constant, linear, and interaction terms

- 'quadratic' — Constant, linear, interaction, and squared terms

- 'purequadratic' — Constant, linear, and squared terms

Alternatively, *model* can be a matrix specifying polynomial terms of
arbitrary order. In this case, *model* should have one column for each
factor and one row for each term in the model. The entries in any row
of *model* are powers for the factors in the columns. For example, if a
model has factors X1, X2, and X3, then a row [0 1 2] in *model* specifies
the term (X1.^0).*(X2.^1).*(X3.^2). A row of all zeros in *model*
specifies a constant term, which can be omitted.

[dC,C] = candgen(nfactors,*model*) also returns the design matrix C
evaluated at the treatments in dC. The order of the columns of C for a
full quadratic model with *n* terms is:

1 The constant term

2 The linear terms in order 1, 2, ..., *n*

3 The interaction terms 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.

Pass C to candexch to generate a *D*-optimal design using a coordinate-exchange algorithm.

[...] = candgen(...,*param1*,*val1*,*param2*,*val2*,...) specifies additional parameter/value pairs for the design. Valid parameters and their values are listed in the following table.

Parameter	Value
'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.
'levels'	Vector of number of levels for each factor.

Note The rowexch function automatically generates a candidate set using candgen, and then creates a *D*-optimal design from that candidate set using candexch. Call candexch separately to specify your own candidate set to the row-exchange algorithm.

Examples The following example uses rowexch to generate a five-run design for a two-factor pure quadratic model using a candidate set that is produced internally:

```
dRE1 = rowexch(2,5,'purequadratic','tries',10)
dRE1 =
    -1     1
     0     0
     1    -1
     1     0
     1     1
```

The same thing can be done using candgen and candexch in sequence:

```
[dC,C] = candgen(2,'purequadratic') % Candidate set, C
dC =
    -1    -1
     0    -1
     1    -1
    -1     0
     0     0
     1     0
    -1     1
     0     1
     1     1
C =
     1    -1    -1     1     1
     1     0    -1     0     1
     1     1    -1     1     1
     1    -1     0     1     0
     1     0     0     0     0
     1     1     0     1     0
     1    -1     1     1     1
     1     0     1     0     1
     1     1     1     1     1
treatments = candexch(C,5,'tries',10) % Find D-opt subset
treatments =
     2
     1
     7
     3
     4
dRE2 = dC(treatments,:) % Display design
dRE2 =
     0    -1
    -1    -1
    -1     1
     1    -1
    -1     0
```

See Also   candexch, rowexch

canoncorr

Purpose   Canonical correlation

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:

Field	Description
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: A User's Perspective.* New York: Oxford University Press, 1988.

[2] Seber, G. A. F. *Multivariate Observations.* Hoboken, NJ: John Wiley & Sons, Inc., 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 a
process with fewer measurements outside of specification.

capability

Examples   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
```

Probability Between Limits = 0.96688

References   [1] Montgomery, D. *Introduction to Statistical Quality Control*. Hoboken, NJ: John Wiley & Sons, 1991, pp. 369–374.

See Also   capaplot, histfit

capaplot

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`.

Examples   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
```

Probability Between Limits = 0.96688

See Also   capability, histfit

caseread

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.
Examples	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.

Examples
```
strmat = char('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

categorical.cat

Purpose   Concatenate categorical arrays

Syntax   `c = cat(dim,A,B,...)`

Description   `c = cat(dim,A,B,...)` concatenates the categorical arrays `A,B,...`
along dimension `dim`. All inputs must have the same size except along
dimension `dim`. The set of categorical levels for `C` is the sorted union of
the sets of levels of the inputs, as determined by their labels.

See Also   `cat` (MATLAB function), `horzcat` (`categorical` class method), `vertcat`
(`categorical` class method)

Purpose   Arrays for categorical data

Description   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 you do not encode ordering in the levels, the data and the array are nominal. If you do encode an ordering, the data and the array are ordinal.

Construction

categorical	Create categorical array

Methods

addlevels	Add levels to categorical array
cat	Concatenate categorical arrays
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
disp	Display categorical array
display	Display categorical array
double	Convert categorical array to double array
droplevels	Drop levels
end	Last index in indexing expression for categorical array
flipdim	Flip categorical array along specified dimension

categorical class

fliplr	Flip categorical matrix in left/right direction
flipud	Flip categorical matrix in up/down direction
getlabels	Access categorical array labels
getlevels	Get categorical array levels
horzcat	Horizontal concatenation for categorical arrays
int16	Convert categorical array to signed 16-bit integer array
int32	Convert categorical array to signed 32-bit integer array
int64	Convert categorical array to signed 64-bit integer array
int8	Convert categorical array to signed 8-bit integer array
intersect	Set intersection for categorical arrays
ipermute	Inverse permute dimensions of categorical array
isempty	True for empty categorical array
isequal	True if categorical arrays are equal
islevel	Test for levels
ismember	True for elements of categorical array in set
isscalar	True if categorical array is scalar
isundefined	Test for undefined elements
isvector	True if categorical array is vector

length	Length of categorical array
levelcounts	Element counts by level
ndims	Number of dimensions of categorical array
numel	Number of elements in categorical array
permute	Permute dimensions of categorical array
reorderlevels	Reorder levels
repmat	Replicate and tile categorical array
reshape	Resize categorical array
rot90	Rotate categorical matrix 90 degrees
setdiff	Set difference for categorical arrays
setlabels	Label levels
setxor	Set exclusive-or for categorical arrays
shiftdim	Shift dimensions of categorical array
single	Convert categorical array to single array
size	Size of categorical array
squeeze	Squeeze singleton dimensions from categorical array
subsasgn	Subscripted assignment for categorical array

subsindex	Subscript index for categorical array
subsref	Subscripted reference for categorical array
summary	Summary statistics for categorical array
times	Product of categorical arrays
transpose	Transpose categorical matrix
uint16	Convert categorical array to unsigned 16-bit integers
uint32	Convert categorical array to unsigned 32-bit integers
uint64	Convert categorical array to unsigned 64-bit integers
uint8	Convert categorical array to unsigned 8-bit integers
union	Set union for categorical arrays
unique	Unique values in categorical array
vertcat	Vertical concatenation for categorical arrays

Properties

labels	Text labels for levels
undeflabel	Text label for undefined levels

Copy Semantics

Value. To learn how this affects your use of the class, see Comparing Handle and Value Classes in the MATLAB Object-Oriented Programming documentation.

See Also   "Categorical Arrays" on page 2-13

categorical

Purpose	Create categorical array
Description	`categorical` is an abstract class, and you cannot create instances of it directly. You must create `nominal` or `ordinal` arrays.
See Also	`nominal`, `ordinal`

Purpose   Concatenate dataset arrays

Syntax   ds = cat(dim, ds1, ds2, ...)

Description ds = cat(dim, ds1, ds2, ...) concatenates the dataset arrays ds1, ds2, ... along dimension dim by calling the dataset/horzcat or dataset/vertcat method. dim must be 1 or 2.

See Also   horzcat, vertcat

classregtree.catsplit

Purpose   Categorical splits used for branches in decision tree

Syntax   v=catsplit(t)
                 v=catsplit(t,j)

Description v=catsplit(t) returns an n-by-2 cell array v. Each row in v gives left
                 and right values for a categorical split. For each branch node j based on
                 a categorical predictor variable z, the left child is chosen if z is in v(j,1)
                 and the right child is chosen if z is in v(j,2). The splits are in the
                 same order as nodes of the tree. Nodes for these splits can be found by
                 running cuttype and selecting 'categorical' cuts from top to bottom.

                 v=catsplit(t,j) takes an array j of rows and returns the splits for
                 the specified rows.

See Also   classregtree

Purpose   Cumulative distribution function for Gaussian mixture distribution

Syntax   y = cdf(obj,X)

Description   y = cdf(obj,X) returns a vector y of length *n* containing the values of
the cumulative distribution function (cdf) for the gmdistribution object
obj, evaluated at the *n*-by-*d* data matrix X, where *n* is the number of
observations and *d* is the dimension of the data. obj is an object created
by gmdistribution or fit. y(I) is the cdf of observation I.

Examples   Create a gmdistribution object defining a two-component mixture of
bivariate Gaussian distributions:

```
MU = [1 2;-3 -5];
SIGMA = cat(3,[2 0;0 .5],[1 0;0 1]);
p = ones(1,2)/2;
obj = gmdistribution(MU,SIGMA,p);

ezsurf(@(x,y)cdf(obj,[x y]),[-10 10],[-10 10])
```

gmdistribution.cdf

cdf(obj,[x,y])

See Also   gmdistribution, fit, pdf, mvncdf

Purpose   Central composite design

Syntax
```
dCC = ccdesign(n)
[dCC,blocks] = ccdesign(n)
[...] = ccdesign(n,param1,val1,param2,val2,...)
```

Description   dCC = ccdesign(n) generates a central composite design for n factors.
n must be an integer 2 or larger. The output matrix dCC is *m*-by-n, where
m is the number of runs in the design. Each row represents one run,
with settings for all factors represented in the columns. Factor values
are normalized so that the cube points take values between -1 and 1.

[dCC,blocks] = ccdesign(n) requests a blocked design. The output
blocks is an *m*-by-1 vector of block numbers for each run. Blocks
indicate runs that are to be measured under similar conditions
to minimize the effect of inter-block differences on the parameter
estimates.

[...] = ccdesign(n,*param1*,*val1*,*param2*,*val2*,...) specifies
additional parameter/value pairs for the design. Valid parameters and
their values are listed in the following table.

Parameter	Description	Values
'center'	Number of center points.	• Integer — Number of center points to include. • 'uniform' — Select number of center points to give uniform precision. • 'orthogonal' — Select number of center points to give an orthogonal design. This is the default.

Parameter	Description	Values
'fraction'	Fraction of full-factorial cube, expressed as an exponent of 1/2.	• 0 — Whole design. This is the default. • 1 — 1/2 fraction. • 2 — 1/4 fraction.
'type'	Type of CCD.	• 'circumscribed' — Circumscribed (CCC). This is the default. • 'inscribed' — Inscribed (CCI). • 'faced' — Faced (CCF).
'blocksize'	Maximum number of points per block.	Integer. The default is Inf.

Examples The following creates a 2-factor CCC:

```
dCC = ccdesign(2,'type','circumscribed')
dCC =
   -1.0000   -1.0000
   -1.0000    1.0000
    1.0000   -1.0000
    1.0000    1.0000
   -1.4142         0
    1.4142         0
         0   -1.4142
         0    1.4142
         0         0
         0         0
         0         0
         0         0
         0         0
         0         0
```

```
            0          0
            0          0
```

The center point is run 8 times to allow for a more uniform estimate of the prediction variance over the entire design space.

Visualize the design as follows:

```
plot(dCC(:,1),dCC(:,2),'ro','MarkerFaceColor','b')
X = [1 -1 -1 -1; 1 1 1 -1];
Y = [-1 -1 1 -1; 1 -1 1 1];
line(X,Y,'Color','b')
axis square equal
```


See Also   bbdesign

cdf

Purpose	Cumulative distribution functions

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`. `A` contains parameter values for the distribution. 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 Compute the cdf of the normal distribution with mean 0 and standard deviation 1 at inputs –2, –1, 0, 1, 2:

```
p1 = cdf('Normal',-2:2,0,1)
p1 =
  0.0228  0.1587  0.5000  0.8413  0.9772
```

The order of the parameters is the same as for `normcdf`.

Compute the cdfs of Poisson distributions with rate parameters 0, 1, ..., 4 at inputs 1, 2, ..., 5, respectively:

```
p2 = cdf('Poisson',0:4,1:5)
p2 =
  0.3679  0.4060  0.4232  0.4335  0.4405
```

The order of the parameters is the same as for `poisscdf`.

See Also   `pdf`, `icdf`

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.

Examples   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, icdf

ProbDist.cdf

Purpose	Return cumulative distribution function (CDF) for ProbDist object
Syntax	`Y = cdf(PD, X)`
Description	`Y = cdf(PD, X)` returns `Y`, an array containing the cumulative distribution function (CDF) for the ProbDist object `PD`, evaluated at values in `X`.

Inputs

`PD`	An object of the class `ProbDistUnivParam` or `ProbDistUnivKernel`.
`X`	A numeric array of values where you want to evaluate the CDF.

Outputs

`Y`	An array containing the cumulative distribution function (CDF) for the ProbDist object `PD`.

See Also   cdf

Purpose   Empirical cumulative distribution function plot

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	Description
stats.min	Minimum value
stats.max	Maximum value
stats.mean	Sample mean
stats.median	Sample median (50th percentile)
stats.std	Sample standard deviation

Examples   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

Purpose   Convert categorical array to cell array of strings

Syntax   B = cellstr(A)

Description   B = cellstr(A) converts the categorical array A to a cell array of
                     strings. Each element of B contains the categorical level label for the
                     corresponding element of A.

See Also   char, getlabels

categorical.char

Purpose	Convert categorical array to character array
Syntax	`B = char(A)`
Description	`B = char(A)` converts the categorical array `A` to a 2-D character matrix. `char` does not preserve the shape of `A`. `B` contains `numel(A)` rows, and each row of `B` contains the categorical level label for the corresponding element of `A(:)`.
See Also	`cellstr`, `getlabels`

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 v is

                 $$p = F(x|v) = \int_0^x \frac{t^{(v-2)/2}e^{-t/2}}{2^{v/2}\,\Gamma(v/2)}dt$$

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

                 The chi-square density function with v degrees-of-freedom is the same
                 as the gamma density function with parameters v/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, chi2pdf, chi2inv, chi2stat, chi2rnd

chi2gof

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. Do not specify both 'cdf' and 'expected'.

- 'cdf' — A fully specified cumulative distribution function. This can be a function name, a function handle, or a ProbDist object of the ProbDistUnivParam class or ProbDistUnivKernel class. When 'cdf' is a function name or handle, the distribution 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   Chi-square inverse 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 v degrees of
freedom is

$$x = F^{-1}(p|v) = \{x : F(x|v) = p\}$$

where

$$p = F(x|v) = \int_0^x \frac{t^{(v-2)/2} e^{-t/2}}{2^{v/2}\, \Gamma(v/2)} dt$$

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.

chi2inv

See Also   icdf, chi2cdf, chi2pdf, chi2stat, chi2rnd

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 v degrees of freedom is

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

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 v/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   `pdf`, `chi2cdf`, `chi2inv`, `chi2stat`, `chi2rnd`

chi2rnd

Purpose	Chi-square random numbers

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.

Examples   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   random, chi2pdf, chi2cdf, chi2inv, chi2stat

Purpose   Chi-square mean and variance

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 v, the degrees of freedom parameter, and the variance is 2v.

Examples
```
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
```

chi2stat

See Also   chi2pdf, chi2cdf, chi2inv, chi2rnd

Purpose   Child nodes

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.

Examples   Create a classification tree for Fisher's iris data:

```
load fisheriris;
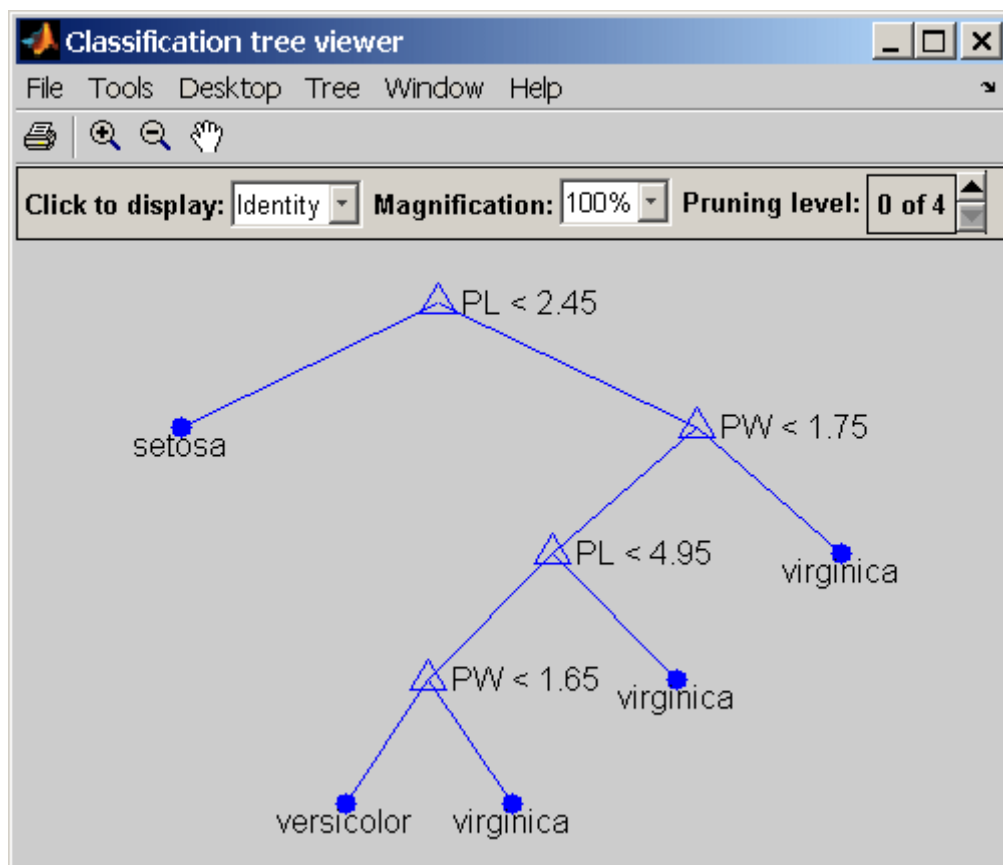
t = classregtree(meas,species,...
                 'names',{'SL' 'SW' 'PL' 'PW'})
t =
Decision tree for classification
1  if PL<2.45 then node 2 else node 3
2  class = setosa
3  if PW<1.75 then node 4 else node 5
4  if PL<4.95 then node 6 else node 7
5  class = virginica
6  if PW<1.65 then node 8 else node 9
7  class = virginica
8  class = versicolor
9  class = virginica

view(t)
```

```
C = children(t)
C =
     2     3
     0     0
     4     5
     6     7
     0     0
     8     9
     0     0
```

```
        0      0
        0      0
```

References   [1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees.* Boca Raton, FL: CRC Press, 1984.

See Also   classregtree, numnodes, parent

cholcov

Purpose   Cholesky-like covariance decomposition

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)`.

Examples   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, cov

categorical.circshift

Purpose   Shift categorical array circularly

Syntax   B = circshift(A,shiftsize)

Description B = circshift(A,shiftsize) circularly shifts the values in the
categorical array A by `shiftsize` elements. `shiftsize` is a vector of
integer scalars where the n-th element specifies the shift amount for the
n-th dimension of array A. If an element in `shiftsize` is positive, the
values of A are shifted down (or to the right). If it is negative, the values
of A are shifted up (or to the left).

See Also   permute, shiftdim

Purpose   Flag for non-empty classes

Description   The `CIsNonEmpty` property is a logical vector of length `NClasses` specifying which classes are not empty. When the grouping variable is categorical, it may contain categorical levels that don't appear in the elements of the grouping variable. Those levels are empty and `NaiveBayes` ignores them for the purposes of training the classifier.

classregtree.classcount

Purpose   Class counts

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.

Examples   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
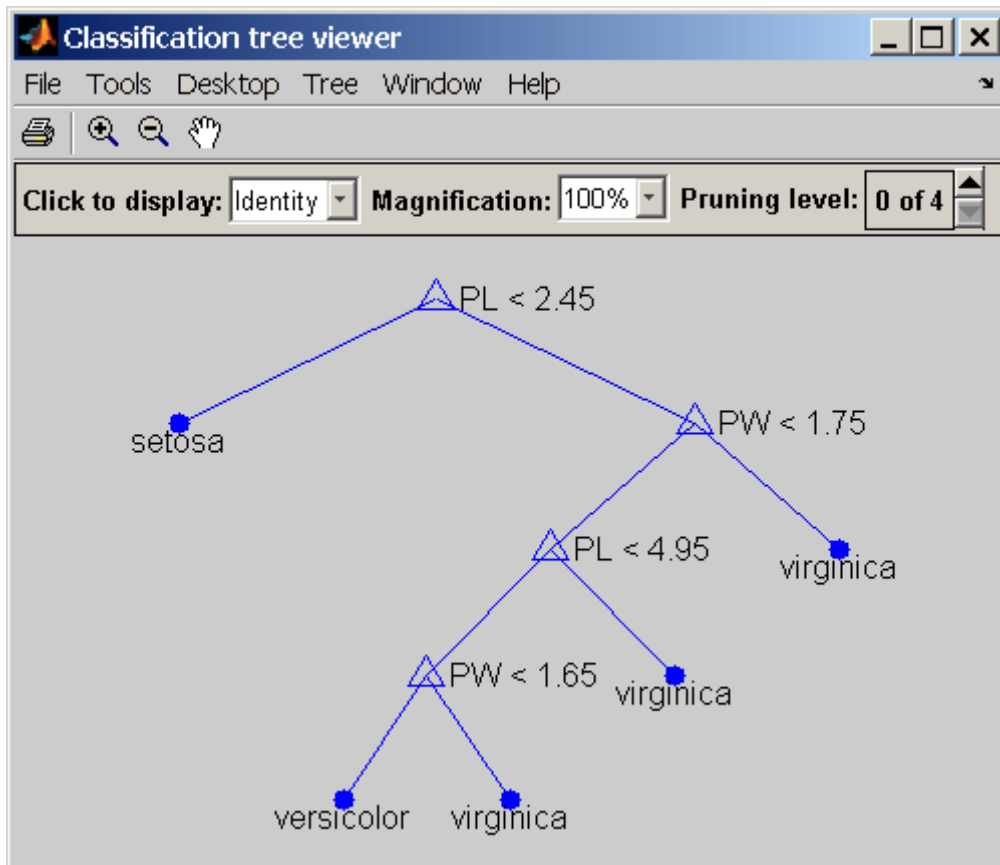
view(t)
```

```
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
```

References   [1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

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 2-33.) `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 *j*th training group was the source of the *i*th 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'.

Examples   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');
```

classify

```
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}')
```

classify

Classification with Fisher Training Data

References
[1] Krzanowski, W. J. *Principles of Multivariate Analysis: A User's Perspective.* New York: Oxford University Press, 1988.

[2] Seber, G. A. F. *Multivariate Observations.* Hoboken, NJ: John Wiley & Sons, Inc., 1984.

See Also
"Grouped Data" on page 2-33

classregtree, mahal, NaiveBayes

Purpose   Names of classes

Description The `ClassNames` property is a cell array containing the class names for the response variable `Y` supplied to `TreeBagger`. This property is empty for regression trees.

TreeBagger.ClassNames property

Purpose   Names of classes

Description   The `ClassNames` property is a cell array containing the class names for the response variable `Y`. This property is empty for regression trees.

Purpose   Class probabilities

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.

Examples   Create a classification tree for Fisher's iris data:

```
load fisheriris;
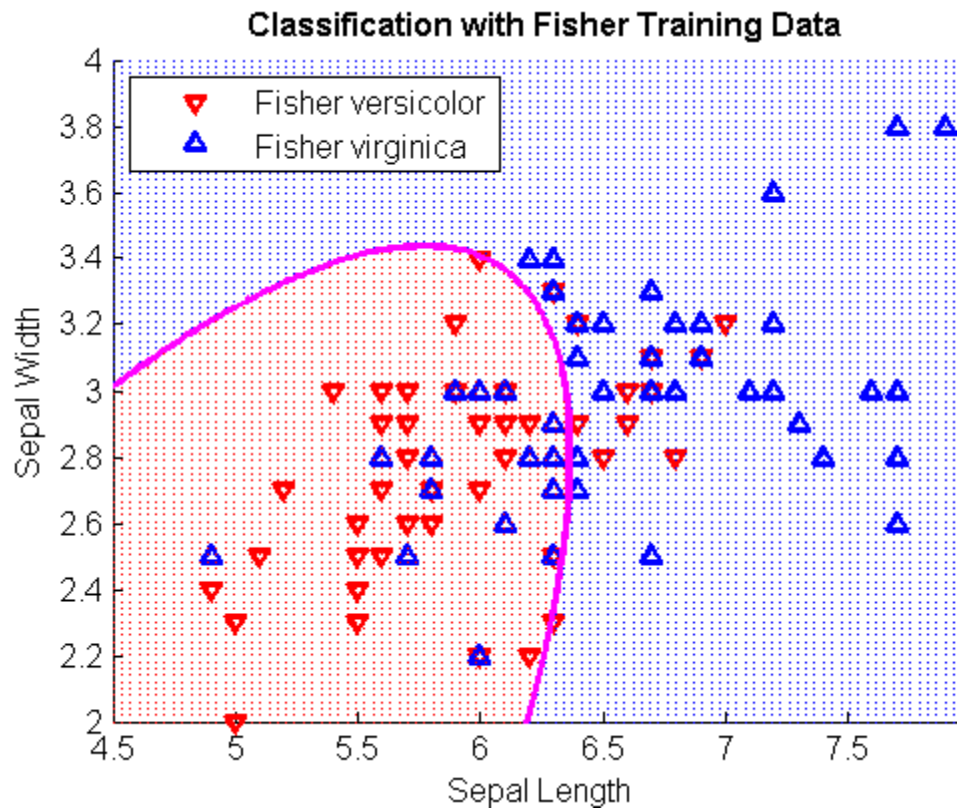
t = classregtree(meas,species,...
                 'names',{'SL' 'SW' 'PL' 'PW'})
t =
Decision tree for classification
1  if PL<2.45 then node 2 else node 3
2  class = setosa
3  if PW<1.75 then node 4 else node 5
4  if PL<4.95 then node 6 else node 7
5  class = virginica
6  if PW<1.65 then node 8 else node 9
7  class = virginica
8  class = versicolor
9  class = virginica

view(t)
```

classregtree.classprob


```
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
```

References   [1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees.* Boca Raton, FL: CRC Press, 1984.

See Also   classregtree, numnodes

classregtree class

Purpose   Classification and regression trees

Construction

classregtree	Construct classification and regression trees

Methods

catsplit	Categorical splits used for branches in decision tree
children	Child nodes
classcount	Class counts
classprob	Class probabilities
cutcategories	Cut categories
cutpoint	Returns decision tree cut point values
cuttype	Cut types
cutvar	Cut variable names
disp	Display `classregtree` object
display	Display `classregtree` object
eval	Predicted responses
isbranch	Test node for branch
nodeerr	Return vector of node errors
nodeprob	Node probabilities
nodesize	Return node size
numnodes	Number of nodes
parent	Parent node
prune	Prune tree
risk	Node risks

subsasgn	Subscripted reference for `classregtree` object
subsref	Subscripted reference for `classregtree` object
test	Error rate
type	Tree type
varimportance	Compute embedded estimates of input feature importance
view	Plot tree

Properties   Objects of the `classregtree` class have no properties accessible by dot indexing, `get` methods, or `set` methods. To obtain information about a `classregtree` object, use the appropriate method.

Copy Semantics   Value. To learn how this affects your use of the class, see Comparing Handle and Value Classes in the MATLAB Object-Oriented Programming documentation.

See Also   "Regression and Classification by Bagging Decision Trees" on page 11-14, "Classification Trees" on page 11-9

"Regression Trees" on page 8-84

"Grouped Data" on page 2-33

classregtree

Purpose	Construct classification and regression trees

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 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.

- 'prune' — 'on' (default) to compute the full tree and the optimal sequence of pruned subtrees, or 'off' for the full tree without pruning.

- 'minparent' — A number *k* such that impure nodes must have *k* or more observations to be split (default is 10).

- 'minleaf' — A minimal number of observations per tree leaf (default is 1). If you supply both 'minparent' and 'minleaf', classregtree uses the setting which results in larger leaves: minparent = max(minparent,2*minleaf)

- `'nvartosample'` — Number of predictor variables randomly selected for each split. By default all variables are considered for each decision split.

- `'mergeleaves'` — `'on'` (default) to merge leaves that originate from the same parent node and give the sum of risk values greater or equal to the risk associated with the parent node. If `'off'`, `classregtree` does not merge leaves.

For regression trees only:

- `'qetoler'` — Defines tolerance on quadratic error per node for regression trees. Splitting nodes stops when quadratic error per node drops below `qetoler*qed`, where `qed` is the quadratic error for the entire data computed before the decision tree is grown: `qed = norm(y-ybar)` with `ybar` estimated as the average of the input array `Y`. Default value is 1e-6.

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 (default has `C(i,j)=1` if `i~=j`, and `C(i,j)=0` if `i=j`). Alternatively, this value can be a structure S having two fields: `S.group`containing the group names as a categorical variable, character array, or cell array of strings; and S.cost containing the cost matrix C.

- `'splitcriterion'` — Criterion for choosing a split. One of `'gdi'` (default) or 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 groupnames as a categorical variable, character array, or cell array of strings; and `S.prob` containing a vector of corresponding probabilities.

Examples   Create a classification tree for Fisher's iris data:

```
load fisheriris;
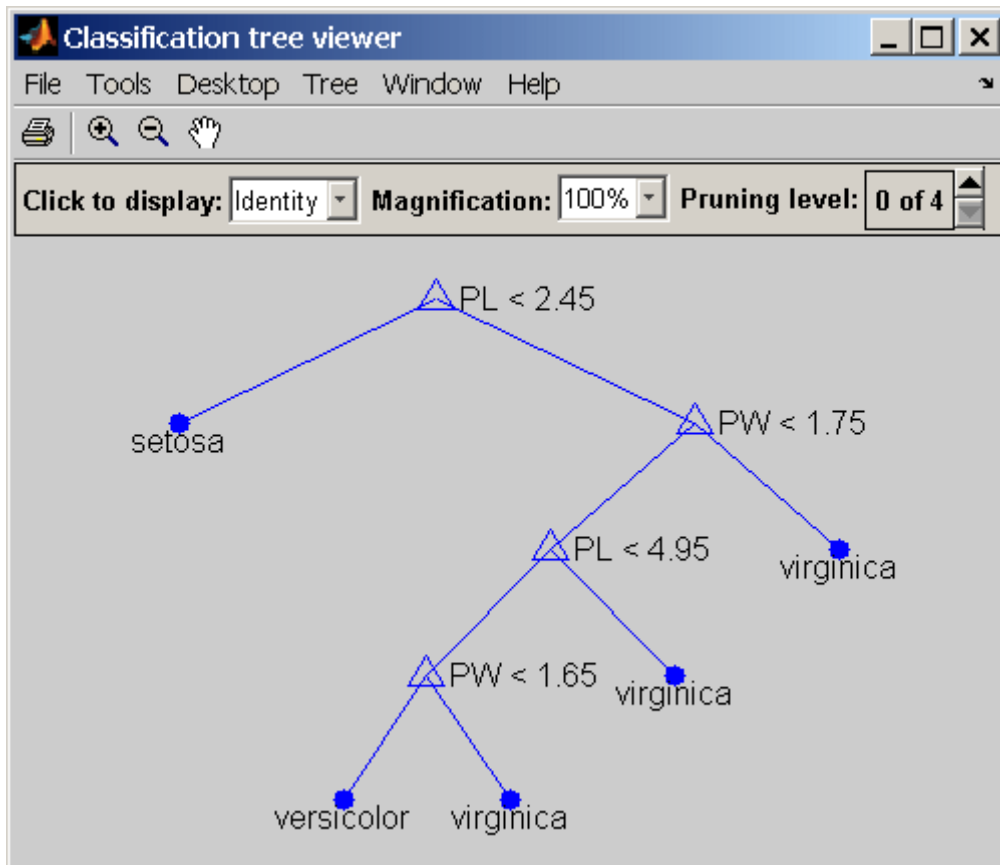
t = classregtree(meas,species,...
                 'names',{'SL' 'SW' 'PL' 'PW'})
t =
Decision tree for classification
1  if PL<2.45 then node 2 else node 3
2  class = setosa
```

```
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)
```

classregtree

References   [1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

See Also   "Regression and Classification by Bagging Decision Trees" on page 11-14

Grouped Data

`eval`, `prune`, `test`, `view`

Purpose   Class levels

Description The CLevels property is a vector of the same type as the grouping variable, containing the unique levels of the grouping variable.

cluster

Purpose	Construct clusters from linkages

Syntax

```
T = cluster(Z,'cutoff',c)
T = cluster(Z,'cutoff',c,'depth',d)
T = cluster(Z,'cutoff',c,'criterion',criterion)
T = cluster(Z,'maxclust',n)
```

Description

$T = cluster(Z, 'cutoff', c)$ constructs clusters from the hierarchical cluster tree, Z, as 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 a node and all of its subnodes have inconsistent value less than c. All leaves at or below the node are grouped into a cluster. t is a vector of size m containing the cluster assignments of each observation.

If c is a vector, T is a matrix of cluster assignments with one column per cutoff value.

$T = cluster(Z, 'cutoff', c, 'depth', d)$ evaluates inconsistent values by looking to a depth d below each node. The default depth is 2.

$T = cluster(Z, 'cutoff', c, 'criterion', criterion)$ uses the specified criterion for forming clusters, where *criterion* is one of the strings 'inconsistent' (default) or 'distance'. The 'distance' criterion uses the distance between the two subnodes merged at a node to measure node height. All leaves at or below a node with height less than c are grouped into a cluster.

$T = cluster(Z, 'maxclust', n)$ constructs a maximum of n clusters using the 'distance' criterion. cluster finds the smallest height at which a horizontal cut through the tree leaves n or fewer clusters.

If n is a vector, T is a matrix of cluster assignments with one column per maximum value.

Examples

Compare clusters from Fisher iris data with species:

```
load fisheriris
d = pdist(meas);
```

```
Z = linkage(d);
c = cluster(Z,'maxclust',3:5);

crosstab(c(:,1),species)
ans =
     0     0     2
     0    50    48
    50     0     0

crosstab(c(:,2),species)
ans =
     0     0     1
     0    50    47
     0     0     2
    50     0     0

crosstab(c(:,3),species)
ans =
     0     4     0
     0    46    47
     0     0     1
     0     0     2
    50     0     0
```

See Also   clusterdata, cophenet, inconsistent, linkage, pdist

gmdistribution.cluster

Purpose   Construct clusters from Gaussian mixture distribution

Syntax   idx = cluster(obj,X)
                 [idx,nlogl] = cluster(obj,X)
                 [idx,nlogl,P] = cluster(obj,X)
                 [idx,nlogl,P,logpdf] = cluster(obj,X)
                 [idx,nlogl,P,logpdf,M] = cluster(obj,X)

Description idx = cluster(obj,X) partitions data in the *n*-by-*d* matrix X, where *n* is the number of observations and *d* is the dimension of the data, into *k* clusters determined by the *k* components of the Gaussian mixture distribution defined by obj. obj is an object created by gmdistribution or fit. idx is an *n*-by-1 vector, where idx(I) is the cluster index of observation I. The cluster index gives the component with the largest posterior probability for the observation, weighted by the component probability.

> **Note** The data in X is typically the same as the data used to create the Gaussian mixture distribution defined by obj. Clustering with cluster is treated as a separate step, apart from density estimation. For cluster to provide meaningful clustering with new data, X should come from the same population as the data used to create obj.

cluster treats NaN values as missing data. Rows of X with NaN values are excluded from the partition.

[idx,nlogl] = cluster(obj,X) also returns nlogl, the negative log-likelihood of the data.

[idx,nlogl,P] = cluster(obj,X) also returns the posterior probabilities of each component for each observation in the *n*-by-*k* matrix P. P(I,J) is the probability of component J given observation I.

[idx,nlogl,P,logpdf] = cluster(obj,X) also returns the *n*-by-1 vector logpdf containing the logarithm of the estimated probability density function for each observation. The density estimate for

observation I is a sum over all components of the component density at I times the component probability.

[idx,nlogl,P,logpdf,M] = cluster(obj,X) also returns an *n*-by-*k* matrix M containing Mahalanobis distances in squared units. M(I,J) is the Mahalanobis distance of observation I from the mean of component J.

Examples   Generate data from a mixture of two bivariate Gaussian distributions using the mvnrnd function:

```
MU1 = [1 2];
SIGMA1 = [2 0; 0 .5];
MU2 = [-3 -5];
SIGMA2 = [1 0; 0 1];
X = [mvnrnd(MU1,SIGMA1,1000);mvnrnd(MU2,SIGMA2,1000)];
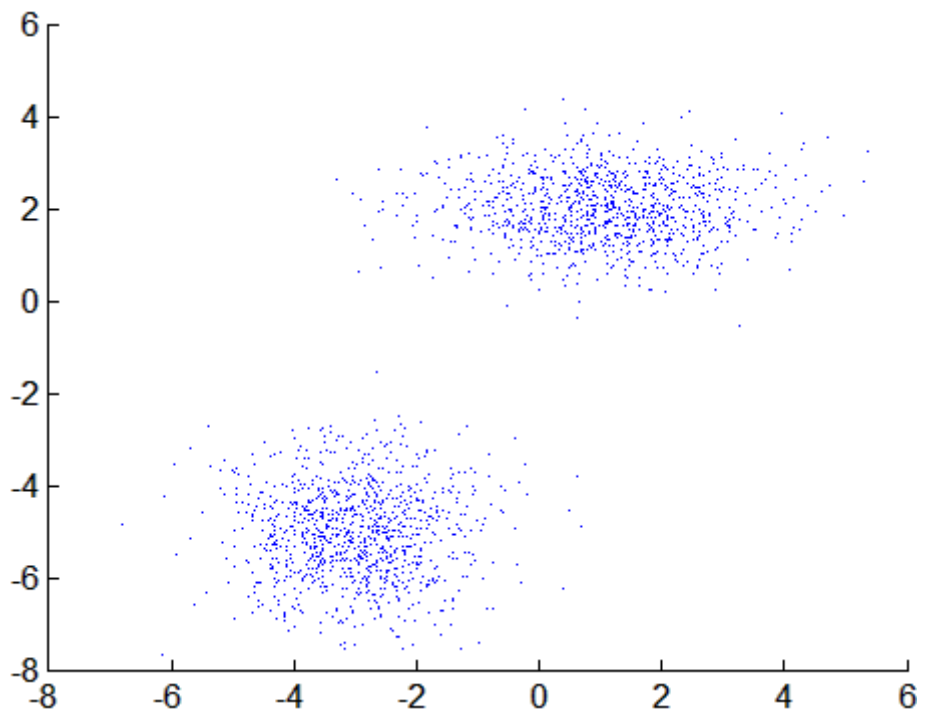
scatter(X(:,1),X(:,2),10,'.')
hold on
```

Fit a two-component Gaussian mixture model:

```
obj = gmdistribution.fit(X,2);
h = ezcontour(@(x,y)pdf(obj,[x y]),[-8 6],[-8 6]);
```

Use the fit to cluster the data:

```
idx = cluster(obj,X);
cluster1 = X(idx == 1,:);
cluster2 = X(idx == 2,:);
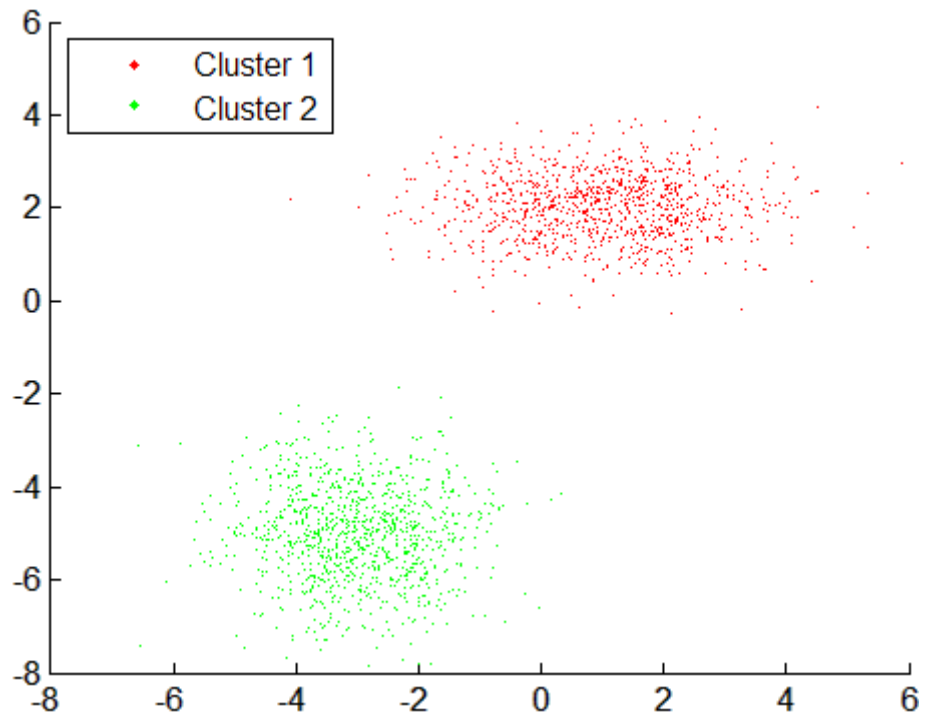
delete(h)
h1 = scatter(cluster1(:,1),cluster1(:,2),10,'r.');
h2 = scatter(cluster2(:,1),cluster2(:,2),10,'g.');
legend([h1 h2],'Cluster 1','Cluster 2','Location','NW')
```

See Also   gmdistribution, fit, posterior, mahal

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 inconsistent). When cutoff is an integer
≥ 2, 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.

When 0 < cutoff < 2, T = clusterdata(X,cutoff) is equivalent to:

```
Y = pdist(X,'euclid');
Z = linkage(Y,'single');
T = cluster(Z,'cutoff',cutoff);
```

When cutoff is an integer ≥ 2, T = clusterdata(X,cutoff) is
equivalent to:

```
Y = pdist(X,'euclid');
Z = linkage(Y,'single');
T = cluster(Z,'maxclust',cutoff);
```

T = clusterdata(X,*param1*,*val1*,*param2*,*val2*,...) provides more
control over the clustering through a set of parameter/value pairs.
Valid parameters are:

Parameter	Value
'distance'	Any of the distance metric names allowed by pdist (follow the 'minkowski' option by the value of the exponent p)

clusterdata

Parameter	Value
'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

Examples

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

cmdscale

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.

cmdscale

Examples

Generate some points in 4-D space, but close to 3-D 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*. Hoboken, NJ: John Wiley & Sons, Inc., 1984.

See Also

mdscale, pdist, procrustes

17-183

NaiveBayes.CNames property

Purpose   Class names

Description   The `CNames` property is an `NClasses`-by-1 cell array containing the group names, where `NClasses` number of groups in the grouping variable used to create the Naive Bayes classifier.

Purpose   Combine two ensembles

Syntax   `B1 = combine(B1,B2)`

Description   `B1 = combine(B1,B2)` appends decision trees from ensemble `B2` to those stored in `B1` and returns ensemble `B1`. This method requires that the class and variable names be identical in both ensembles.

See Also   `TreeBagger.append`

combnk

Purpose	Enumeration of combinations

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 $n! / k!(n\text{-}k)!$ 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.

Examples   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
```

See Also

perms

Purpose   Compact ensemble of decision trees

Description   Return an object of class `CompactTreeBagger` holding the structure of the trained ensemble. The class is more compact than the full `TreeBagger` class because it does not contain information for growing more trees for the ensemble. In particular, it does not contain X and Y used for training.

See Also   `CompactTreeBagger`

CompactTreeBagger class

Purpose   Compact ensemble of decision trees grown by bootstrap aggregation

Description   `CompactTreeBagger` class is a lightweight class that contains the trees grown using `TreeBagger`. `CompactTreeBagger` does not preserve any information about how `TreeBagger` grew the decision trees. It does not contain the input data used for growing trees, nor does it contain training parameters such as minimal leaf size or number of variables sampled for each decision split at random. You can only use `CompactTreeBagger` for predicting the response of the trained ensemble given new data X, and other related functions.

`CompactTreeBagger` lets you save the trained ensemble to disk, or use it in any other way, while discarding training data and various parameters of the training configuration irrelevant for predicting response of the fully grown ensemble. This reduces storage and memory requirements, especially for ensembles trained on large datasets.

Construction

CompactTreeBagger	Create CompactTreeBagger object

Methods

combine	Combine two ensembles
error	Error (misclassification probability or MSE)
margin	Classification margin
mdsProx	Multidimensional scaling of proximity matrix
meanMargin	Mean classification margin
outlierMeasure	Outlier measure for data
predict	Predict response

proximity	Proximity matrix for data
SetDefaultYfit	Set default value for `predict`

Properties

ClassNames	Names of classes
DefaultYfit	Default value returned by `predict`
DeltaCritDecisionSplit	Split criterion contributions for each predictor
Method	Method used by trees (classification or regression)
NTrees	Number of decision trees in ensemble
Trees	Decision trees in ensemble
VarNames	Variable names

Copy Semantics
Value. To learn how this affects your use of the class, see Comparing Handle and Value Classes in the MATLAB Object-Oriented Programming documentation.

See Also
`classregtree`

"Regression and Classification by Bagging Decision Trees" on page 11-14

"Classification Trees" on page 11-9

"Regression Trees" on page 8-84

"Grouped Data" on page 2-33

CompactTreeBagger

Purpose　　　Create CompactTreeBagger object

Description　When you use the `TreeBagger` constructor to grow trees, it creates a `CompactTreeBagger` object. You can obtain the compact object from the full `TreeBagger` object using the `TreeBagger/compact` method. You do not create an instance of `CompactTreeBagger` directly.

See Also　　"Regression and Classification by Bagging Decision Trees" on page 11-14, Grouped Data

　　　　　　　`TreeBagger`

TreeBagger.ComputeOOBPrediction property

Purpose   Flag to compute out-of-bag predictions

Description   The `ComputeOOBPrediction` property is a logical flag specifying whether out-of-bag predictions for training observations should be computed. The default is `false`.

If this flag is true, the following properties are available:

- `OOBIndices`
- `OOBInstanceWeight`

If this flag is true, the following methods can be called:

- `oobError`
- `oobMargin`
- `oobMeanMargin`

See Also   `oobError`, `OOBIndices`, `OOBInstanceWeight`, `oobMargin`, `oobMeanMargin`

TreeBagger.ComputeOOBVarImp property

Purpose   Flag to compute out-of-bag variable importance

Description   The `ComputeOOBVarImp` property is a logical flag specifying whether
                  `TreeBagger` should compute out-of-bag estimates of variable
                  importance. The default is false.

                  If this flag is true, the following properties are available:

                  • `OOBPermutedVarDeltaError`

                  • `OOBPermutedVarDeltaMeanMargin`

                  • `OOBPermutedVarCountRaiseMargin`

See Also   `ComputeOOBPrediction`, `OOBPermutedVarDeltaError`,
                  `OOBPermutedVarDeltaMeanMargin`,`OOBPermutedVarCountRaiseMargin`,
                  `oobMeanMargin`, `TreeBagger`.

Purpose   Confusion matrix

Syntax
```
C = confusionmat(group,grouphat)
C = confusionmat(group,grouphat,'order',grouporder)
[C,order] = confusionmat(...)
```

Description   `C = confusionmat(group,grouphat)` returns the confusion matrix `C` determined by the known and predicted groups in `group` and `grouphat`, respectively. `group` and `grouphat` are grouping variables with the same number of observations, as described in "Grouped Data" on page 2-33. Input vectors must be of the same type. `C` is a square matrix with size equal to the total number of distinct elements in `group` and `grouphat`. `C(i,j)` is a count of observations known to be in group `i` but predicted to be in group `j`. Group indices and their order are the same for the rows and columns of `C`, computed by `grp2idx` using `grp2idx(group;grouphat)`. NaN, empty, or `'undefined'` groups are not counted.

`C = confusionmat(group,grouphat,'order',grouporder)` uses `grouporder` to order the rows and columns of `C`. `grouporder` is a grouping variable containing all of the distinct elements in `group` and `grouphat`. If `grouporder` contains elements that are not in `group` or `grouphat`, the corresponding entries in `C` will be 0.

`[C,order] = confusionmat(...)` also returns the order of the rows and columns of `C` in a variable `order` the same type as `group` and `grouphat`.

Examples   **Example 1**

Display the confusion matrix for data with two misclassifications and one missing classification:

```
g1 = [1 1 2 2 3 3]';  % Known groups
g2 = [1 1 2 3 4 NaN]'; % Predicted groups

[C,order] = confusionmat(g1,g2)
C =
```

```
     2    0    0    0
     0    1    1    0
     0    0    0    1
     0    0    0    0
order =
     1
     2
     3
     4
```

Example 2

Randomize the measurements and groups in Fisher's iris data:

```
load fisheriris
numObs = length(species);
p = randperm(numObs);
meas = meas(p,:);
species = species(p);
```

Use `classify` to classify measurements in the second half of the data, using the first half of the data for training:

```
half = floor(numObs/2);
training = meas(1:half,:);
trainingSpecies = species(1:half);
sample = meas(half+1:end,:);
grouphat = classify(sample,training,trainingSpecies);
```

Display the confusion matrix for the resulting classification:

```
group = species(half+1:end);
[C,order] = confusionmat(group,grouphat)
C =
    22    0    0
     2   22    0
     0    0   29
order =
    'virginica'
```

```
'versicolor'
'setosa'
```

See Also   "Grouped Data" on page 2-33

crosstab, grp2idx

controlchart

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 2-33.) `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 $1 \leq n \leq$ `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.

Examples 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   "Grouped Data" on page 2-33

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.

Examples   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')
```

controlrules

XBAR control chart

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

gmdistribution.Converged property

Purpose   Determine if algorithm converged

Description   Logical `true` if the algorithm has converged; logical `false` if the
                   algorithm has not converged.

> **Note** This property applies only to gmdistribution objects constructed
> with `fit`.

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:

cophenet

- 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.

Examples
```
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

copulacdf

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].

Examples
```
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')
```

17-209

See Also   copulapdf, copularnd, copulastat, copulaparam

copulafit

Purpose   Fit copula to data

Syntax
```
RHOHAT = copulafit('Gaussian',U)
[RHOHAT,nuhat] = copulafit('t',U)
[RHOHAT,nuhat,nuci] = copulafit('t',U)
paramhat = copulafit(family,U)
[paramhat,paramci] = copulafit(family,U)
[...] = copulafit(...,'alpha',alpha)
[...] = copulafit('t',U,'Method','ApproximateML')
[...] = copulafit(...,'Options',options)
```

Description   RHOHAT = copulafit('Gaussian',U) returns an estimate RHOHAT of
the matrix of linear correlation parameters for a Gaussian copula, given
data in U. U is an *n*-by-*p* matrix of values in the open interval (0,1)
representing *n* points in the *p*-dimensional unit hypercube.

[RHOHAT,nuhat] = copulafit('t',U) returns an estimate RHOHAT
of the matrix of linear correlation parameters for a *t* copula and an
estimate nuhat of the degrees of freedom parameter, given data in U.
U is an *n*-by-*p* matrix of values in the open interval (0,1) representing
n points in the *p*-dimensional unit hypercube.

[RHOHAT,nuhat,nuci] = copulafit('t',U) also returns an
approximate 95% confidence interval nuci for the degrees of freedom
parameter estimated in nuhat.

paramhat = copulafit(*family*,U) returns an estimate paramhat of
the copula parameter for an Archimedean copula specified by *family*,
given data in U. U is an *n*-by-2 matrix of values in the open interval (0,1)
representing *n* points in the unit square. *family* is one of 'Clayton',
'Frank', or 'Gumbel'.

[paramhat,paramci] = copulafit(*family*,U) also returns an
approximate 95% confidence interval paramci for the copula parameter
estimated in paramhat.

[...] = copulafit(...,'alpha',alpha) returns approximate
100*(1-alpha)% confidence intervals in nuci or paramci.

copulafit

`[...] = copulafit('t',U,'Method','ApproximateML')` fits a *t*
copula for large samples U by maximizing an objective function that
approximates the profile log-likelihood for the degrees of freedom
parameter (see [1]). This method can be significantly faster than
maximum likelihood, but the estimates and confidence limits may not
be accurate for small to moderate sample sizes.

`[...] = copulafit(...,'Options',options)` specifies control
parameters for the iterative parameter estimation algorithm
using an options structure `options` as created by `statset`. Type
`statset('copulafit')` at the command prompt for fields and default
values used by `copulafit`. This argument is not applicable to the
`'Gaussian'` family.

References

[1] Bouye, E., V. Durrleman, A. Nikeghbali, G. Riboulet, and T. Roncalli.
"Copulas for Finance: A Reading Guide and Some Applications."
Working Paper. Groupe de Recherche Operationnelle, Credit Lyonnais,
2000.

Examples

Load and plot simulated stock return data:

```
load stockreturns
x = stocks(:,1);
y = stocks(:,2);

scatterhist(x,y)
```

Transform the data to the copula scale (unit square) using a kernel estimator of the cumulative distribution function:

```
u = ksdensity(x,x,'function','cdf');
v = ksdensity(y,y,'function','cdf');

scatterhist(u,v)
xlabel('u')
ylabel('v')
```

Fit a *t* copula:

```
[Rho,nu] = copulafit('t',[u v],'Method','ApproximateML')
Rho =
    1.0000    0.7220
    0.7220    1.0000
nu =
  2.8934e+006
```

Generate a random sample from the *t* copula:

```
r = copularnd('t',Rho,nu,1000);
```

```
u1 = r(:,1);
v1 = r(:,2);
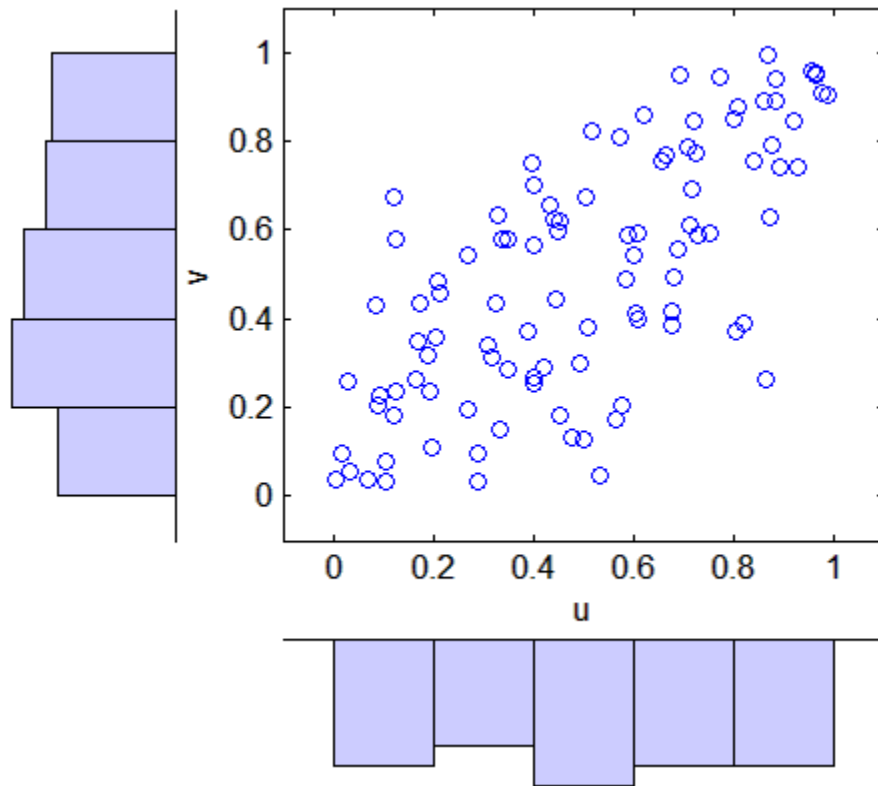
scatterhist(u1,v1)
xlabel('u')
ylabel('v')
set(get(gca,'children'),'marker','.')
```


Transform the random sample back to the original scale of the data:

```
x1 = ksdensity(x,u1,'function','icdf');
y1 = ksdensity(y,v1,'function','icdf');
```

copulafit

```
scatterhist(x1,y1)
set(get(gca,'children'),'marker','.')
```


See Also   ecdf, copulacdf, copulaparam, copulapdf, copularnd, copulastat

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.

Examples   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].

Examples

```
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')
```

copulapdf

See Also   copulacdf, copulaparam, copularnd, copulastat

copulastat

Purpose   Copula rank correlation

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.

Examples   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

copularnd

Purpose	Copula random numbers

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.

Examples

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   Coordinate exchange

Syntax   ```
dCE = cordexch(nfactors,nruns)
[dCE,X] = cordexch(nfactors,nruns)
[dCE,X] = cordexch(nfactors,nruns,model)
[dCE,X] = cordexch(...,param1,val1,param2,val2,...)
```

Description   dCE = cordexch(nfactors,nruns) uses a coordinate-exchange
algorithm to generate a *D*-optimal design dCE with nruns runs (the rows
of dCE) for a linear additive model with nfactors factors (the columns
of dCE). The model includes a constant term.

[dCE,X] = cordexch(nfactors,nruns) also returns the associated
design matrix X, whose columns are the model terms evaluated at each
treatment (row) of dCE.

[dCE,X] = cordexch(nfactors,nruns,*model*) uses the linear
regression model specified in *model*. *model* is one of the following
strings:

- 'linear' — Constant and linear terms. This is the default.

- 'interaction' — Constant, linear, and interaction terms

- 'quadratic' — Constant, linear, interaction, and squared terms

- 'purequadratic' — Constant, linear, and squared terms

The order of the columns of X for a full quadratic model with *n* terms is:

1 The constant term

2 The linear terms in order 1, 2, ..., *n*

3 The interaction terms 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 factor and one row for each term in the model. The entries in any row of *model* are powers for the factors in the columns. For example, if a model has factors X1, X2, and X3, then a row [0 1 2] in *model* specifies the term (X1.^0).*(X2.^1).*(X3.^2). A row of all zeros in *model* specifies a constant term, which can be omitted.

[dCE,X] = cordexch(...,*param1*,*val1*,*param2*,*val2*,...) specifies additional parameter/value pairs for the design. Valid parameters and their values are listed in the following table.

Parameter	Value
'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 the iteration counter. The default is 'on'.
'excludefun'	Handle to a function that excludes undesirable runs. If the function is *f*, it must support the syntax *b* = *f*(*S*), where *S* is a matrix of treatments with nfactors columns and *b* is a vector of Boolean values with the same number of rows as *S*. *b*(*i*) is true if the *i*th row *S* should be excluded.
'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.

Parameter	Value
`'maxiter'`	Maximum number of iterations. The default is 10.
`'tries'`	Number of times to try to generate a design from a new starting point. The algorithm uses random points for each try, except possibly the first. The default is 1.

Algorithm   Both `cordexch` and `rowexch` use iterative search algorithms. They operate by incrementally changing an initial design matrix X to increase $D = |X^TX|$ at each step. In both algorithms, there is randomness built into the selection of the initial design and into the choice of the incremental changes. As a result, both algorithms may return locally, but not globally, D-optimal designs. Run each algorithm multiple times and select the best result for your final design. Both functions have a `'tries'` parameter that automates this repetition and comparison.

Unlike the row-exchange algorithm used by `rowexch`, `cordexch` does not use a candidate set. (Or rather, the candidate set is the entire design space.) At each step, the coordinate-exchange algorithm exchanges a single element of X with a new element evaluated at a neighboring point in design space. The absence of a candidate set reduces demands on memory, but the smaller scale of the search means that the coordinate-exchange algorithm is more likely to become trapped in a local minimum than the row-exchange algorithm.

Examples   Suppose you want a design to estimate the parameters in the following three-factor, seven-term interaction model:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \beta_{12} x_1 x_2 + \beta_{13} x_1 x_3 + \beta_{23} x_2 x_3 + \varepsilon$$

Use `cordexch` to generate a D-optimal design with seven runs:

```
nfactors = 3;
nruns = 7;
[dCE,X] = cordexch(nfactors,nruns,'interaction','tries',10)
dCE =
```

```
        -1     1     1
        -1    -1    -1
         1     1     1
        -1     1    -1
         1    -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     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
```

Columns of the design matrix X are the model terms evaluated at each row of the design dCE. The terms appear in order from left to right: constant term, linear terms (1, 2, 3), interaction terms (12, 13, 23). Use X to fit the model, as described in "Linear Regression" on page 8-3, to response data measured at the design points in dCE.

See Also   rowexch, daugment, dcovary

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 (`NaNs`) • `'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, corrcov, tiedrank

Purpose   Convert covariance matrix to correlation matrix

Syntax   R = corrcov(C)
                     [R,sigma] = corrcov(C)

Description   R = corrcov(C) computes the correlation matrix R corresponding to
                     the covariance matrix C. C must be square, symmetric, and positive
                     semi-definite.

                     [R,sigma] = corrcov(C) also computes the vector of standard
                     deviations sigma.

Examples   Use cov and corrcoef to compute covariances and correlations,
                     respectively, for sample data on weight and blood pressure (systolic,
                     diastolic) in hospital.mat:

```
load hospital
X = [hospital.Weight hospital.BloodPressure];
C = cov(X)
C =
  706.0404   27.7879   41.0202
   27.7879   45.0622   23.8194
   41.0202   23.8194   48.0590
R = corrcoef(X)
R =
    1.0000    0.1558    0.2227
    0.1558    1.0000    0.5118
    0.2227    0.5118    1.0000
```

Compare R with the correlation matrix computed from C by corrcov:

```
corrcov(C)
ans =
    1.0000    0.1558    0.2227
    0.1558    1.0000    0.5118
    0.2227    0.5118    1.0000
```

See Also   cov, corrcoef, corr, cholcov

TreeBagger.Cost property

Purpose Misclassification costs

Description The `Cost` property is a matrix with misclassification costs. This property is empty for ensembles of regression trees.

See Also `classregtree`

Purpose   Type of covariance matrices

Description   The string `'diagonal'` if the covariance matrices are restricted to be diagonal; the string `'full'` otherwise.

coxphfit

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 1s.

                   [...] = 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 jth element of this vector gives the number of times the jth element of y and the jth 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`

Examples 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])
legend('Survivor Function','Weibull Function')
```
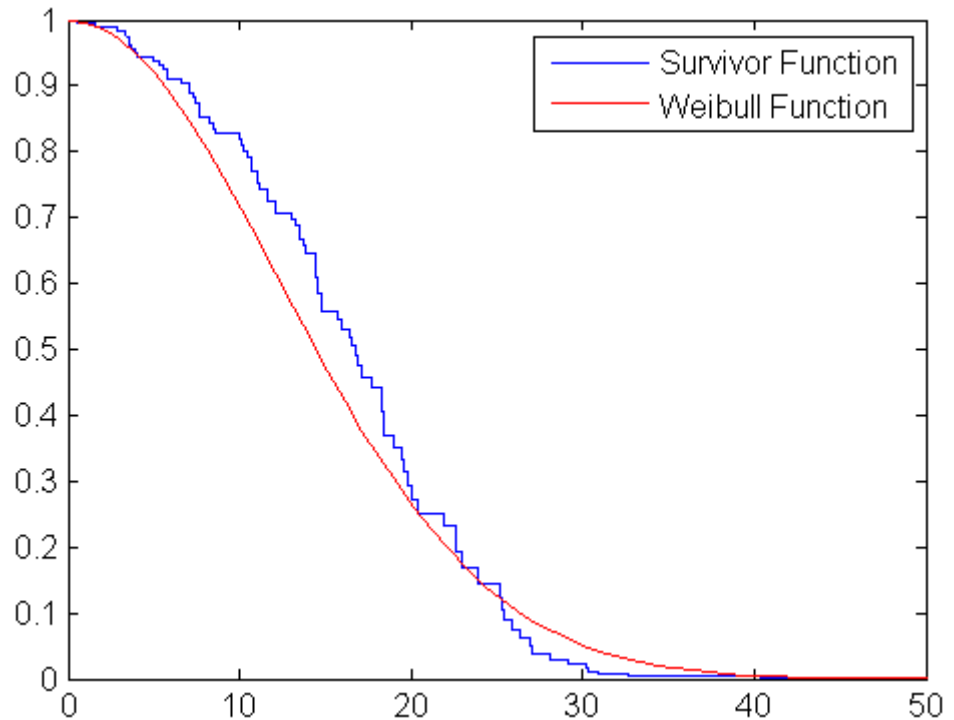

References   [1] Cox, D.R., and D. Oakes. *Analysis of Survival Data.* London: Chapman & Hall, 1984.

[2] Lawless, J. F. *Statistical Models and Methods for Lifetime Data.* Hoboken, NJ: Wiley-Interscience, 2002.

See Also   ecdf, statset, wblfit

Purpose   Class priors

Description The CPrior property is a vector of length NClasses containing the class priors. The priors for empty classes are zero.

crosstab

Purpose   Cross-tabulation

Syntax
```
table = crosstab(x1,x2)
table = crosstab(x1,x2,x3,...)
[table,chi2,p] = crosstab(...)
[table,chi2,p,labels] = crosstab(...)
```

Description   `table = crosstab(x1,x2)` returns a cross-tabulation `table` of two vectors of the same length `x1` and `x2`. `table` is *m*-by-*n*, where *m* is the number of distinct values in `x1` and *n* is the number of distinct values in `x2`.

x1 and x2 are grouping variables, as described in "Grouped Data" on page 2-33. `crosstab` uses `grp2idx` to assign a positive integer to each distinct value. `table(i,j)` is a count of indices where `grp2idx(x1)` is i and `grp2idx(x2)` is j. Rows and columns of `table` are ordered by the numerical order of `grp2idx(x1)` and `grp2idx(x2)`, respectively.

`table = crosstab(x1,x2,x3,...)` returns a multi-dimensional `table` where `table(i,j,k,...)` is a count of indices where `grp2idx(x1)` is i, `grp2idx(x2)` is j, `grp2idx(x3)` is k, and so on.

`[table,chi2,p] = crosstab(...)` also returns the chi-square statistic `chi2` and its *p*-value `p` for a test that `table` is independent in each dimension. The null hypothesis is that the proportion in any entry of `table` is the product of the proportions in each dimension.

`[table,chi2,p,labels] = crosstab(...)` also returns a cell array `labels` with one column for each input argument. The entries in the first column are labels for the rows of `table`, the entries in the second column are labels for the columns, and so on for a multi-dimensional `table`.

Examples

Example 1

Cross-tabulate two vectors with three and four distinct values, respectively:

```
x = [1 1 2 3 1]; y = [1 2 5 3 1];
```

```
table = crosstab(x,y)
table =
     2     1     0     0
     0     0     0     1
     0     0     1     0
```

Example 2

Generate two independent vectors, each containing 50 discrete uniform random numbers in the range 1:3:

```
x1 = unidrnd(3,50,1);
x2 = unidrnd(3,50,1);
[table,chi2,p] = crosstab(x1,x2)
table =
      1     6     7
      5     5     2
     11     7     6
chi2 =
    7.5449
p =
    0.1097
```

At the 95% confidence level, the *p*-value fails to reject the null hypothesis that table is independent in each dimension.

Example 3

The file carbig.mat contains measurements of large model cars during the years 1970-1982:

```
load carbig
[table,chi2,p,labels] = crosstab(cyl4,when,org)
table(:,:,1) =
    82    75    25
    12    22    38
table(:,:,2) =
     0     4     3
    23    26    17
table(:,:,3) =
```

```
         3      3      4
        12     25     32

chi2 =
  207.7689

p =
     0

label =
    'Other'    'Early'    'USA'
    'Four'     'Mid'      'Europe'
        []     'Late'     'Japan'
```

table and label together show that the number of four-cylinder cars made in the USA during the late period of the data was table(2,3,1) or 38 cars.

See Also   "Grouped Data" on page 2-33

tabulate, grp2idx

Purpose	Loss estimate using cross-validation

Syntax	`vals = crossval(fun,X)`
	`vals = crossval(fun,X,Y,...)`
	`mse = crossval('mse',X,y,'Predfun',predfun)`
	`mcr = crossval('mcr',X,y,'Predfun',predfun)`
	`val = crossval(`*criterion*`,X1,X2,...,y,'Predfun',predfun)`
	`vals = crossval(...,`*param1*`,`*val1*`,`*param2*`,`*val2*`,...)`

Description `vals = crossval(fun,X)` performs 10-fold cross-validation for the function `fun`, applied to the data in X.

`fun` is a function handle to a function with two inputs, the training subset of X, XTRAIN, and the test subset of X, XTEST, as follows:

```
testval = fun(XTRAIN,XTEST)
```

Each time it is called, `fun` should use XTRAIN to fit a model, then return some criterion `testval` computed on XTEST using that fitted model.

X can be a column vector or a matrix. Rows of X correspond to observations; columns correspond to variables or features. Each row of `vals` contains the result of applying `fun` to one test set. If `testval` is a non-scalar value, it is converted to a row vector using linear indexing and stored in one row of `vals`.

`vals = crossval(fun,X,Y,...)` is used when data are stored in separate variables X, Y, ... All variables (column vectors, matrices, or arrays) must have the same number of rows. `fun` is called with the training subsets of X, Y, ... , followed by the test subsets of X, Y, ... , as follows:

```
testvals = fun(XTRAIN,YTRAIN,...,XTEST,YTEST,...)
```

`mse = crossval('mse',X,y,'Predfun',predfun)` returns `mse`, a scalar containing a 10-fold cross-validation estimate of mean-squared error for the function `predfun`. X can be a column vector, matrix, or array of predictors. y is a column vector of response values. X and y must have the same number of rows.

predfun is a function handle called with the training subset of X, the training subset of y, and the test subset of X as follows:

```
yfit = predfun(XTRAIN,ytrain,XTEST)
```

Each time it is called, predfun should use XTRAIN and ytrain to fit a regression model and then return fitted values in a column vector yfit. Each row of yfit contains the predicted values for the corresponding row of XTEST. crossval computes the squared errors between yfit and the corresponding response test set, and returns the overall mean across all test sets.

mcr = crossval('mcr',X,y,'Predfun',predfun) returns mcr, a scalar containing a 10-fold cross-validation estimate of misclassification rate (the proportion of misclassified samples) for the function predfun. The matrix X contains predictor values and the vector y contains class labels. predfun should use XTRAIN and YTRAIN to fit a classification model and return yfit as the predicted class labels for XTEST. crossval computes the number of misclassifications between yfit and the corresponding response test set, and returns the overall misclassification rate across all test sets.

val = crossval(*criterion*,X1,X2,...,y,'Predfun',predfun), where *criterion* is 'mse' or 'mcr', returns a cross-validation estimate of mean-squared error (for a regression model) or misclassification rate (for a classification model) with predictor values in X1, X2, ... and, respectively, response values or class labels in y. X1, X2, ... and y must have the same number of rows. predfun is a function handle called with the training subsets of X1, X2, ..., the training subset of y, and the test subsets of X1, X2, ..., as follows:

```
yfit = predfun(X1TRAIN,X2TRAIN,...,ytrain,X1TEST,X2TEST,...)
```

yfit should be a column vector containing the fitted values.

vals = crossval(...,*param1*,*val1*,*param2*,*val2*,...) specifies optional parameter name/value pairs from the following table:

Name	Value
`'holdout'`	A scalar specifying the ratio or the number of observations p for holdout cross-validation. When $0 < p < 1$, approximately p*n observations for the test set are randomly selected. When p is an integer, p observations for the test set are randomly selected.
`'kfold'`	A scalar specifying the number of folds k for k-fold cross-validation.
`'leaveout'`	Specifies leave-one-out cross-validation. The value must be 1.
`'mcreps'`	A positive integer specifying the number of Monte-Carlo repetitions for validation. If the first input of `crossval` is `'mse'` or `'mcr'`, `crossval` returns the mean of mean-squared error or misclassification rate across all of the Monte-Carlo repetitions. Otherwise, `crossval` concatenates the values vals from all of the Monte-Carlo repetitions along the first dimension.
`'partition'`	An object c of the `cvpartition` class, specifying the cross-validation type and partition.
`'stratify'`	A column vector group specifying groups for stratification. Both training and test sets have roughly the same class proportions as in group. NaNs or empty strings in group are treated as missing values, and the corresponding rows of the data are ignored.

Only one of `'kfold'`, `'holdout'`, `'leaveout'`, or `'partition'` can be specified, and `'partition'` cannot be specified with `'stratify'`. If both `'partition'` and `'mcreps'` are specified, the first Monte-Carlo repetition uses the partition information in the `cvpartition` object, and the `repartition` method is called to generate new partitions for each of

the remaining repetitions. If no cross-validation type is specified, the default is 10-fold cross-validation.

Note When using cross-validation with classification algorithms, stratification is preferred. Otherwise, some test sets may not include observations from all classes.

Examples   **Example 1**

Compute mean-squared error for regression using 10-fold cross-validation:

```
load('fisheriris');
y = meas(:,1);
X = [ones(size(y,1),1),meas(:,2:4)];

regf = @(XTRAIN,ytrain,XTEST)(XTEST*regress(ytrain,XTRAIN));

cvMse = crossval('mse',X,y,'predfun',regf)
cvMse =
    0.1015
```

Example 2

Compute misclassification rate using stratified 10-fold cross-validation:

```
load('fisheriris');
y = species;
X = meas;
cp = cvpartition(y,'k',10); % Stratified cross-validation

classf = @(XTRAIN, ytrain,XTEST)(classify(XTEST,XTRAIN,...
ytrain));

cvMCR = crossval('mcr',X,y,'predfun',classf,'partition',cp)
cvMCR =
    0.0200
```

17-244

Example 3

Compute the confusion matrix using stratified 10-fold cross-validation:

```
load('fisheriris');
y = species;
X = meas;
order = unique(y); % Order of the group labels
cp = cvpartition(y,'k',10); % Stratified cross-validation

f = @(xtr,ytr,xte,yte)confusionmat(yte,...
classify(xte,xtr,ytr),'order',order);

cfMat = crossval(f,X,y,'partition',cp);
cfMat = reshape(sum(cfMat),3,3)
cfMat =
    50     0     0
     0    48     2
     0     1    49
```

cfMat is the summation of 10 confusion matrices from 10 test sets.

References   [1] Hastie, T., R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning.* New York: Springer, 2001.

See Also   "Grouped Data" on page 2-33

cvpartition

categorical.ctranspose

Purpose   Transpose categorical matrix

Syntax   B = ctranspose(A)

Description   B = ctranspose(A) returns the transpose of the 2-D categorical matrix A. Note that ctranspose is identical to transpose for categorical arrays.

See Also   transpose, permute

Purpose   Cut categories

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.

Examples   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)
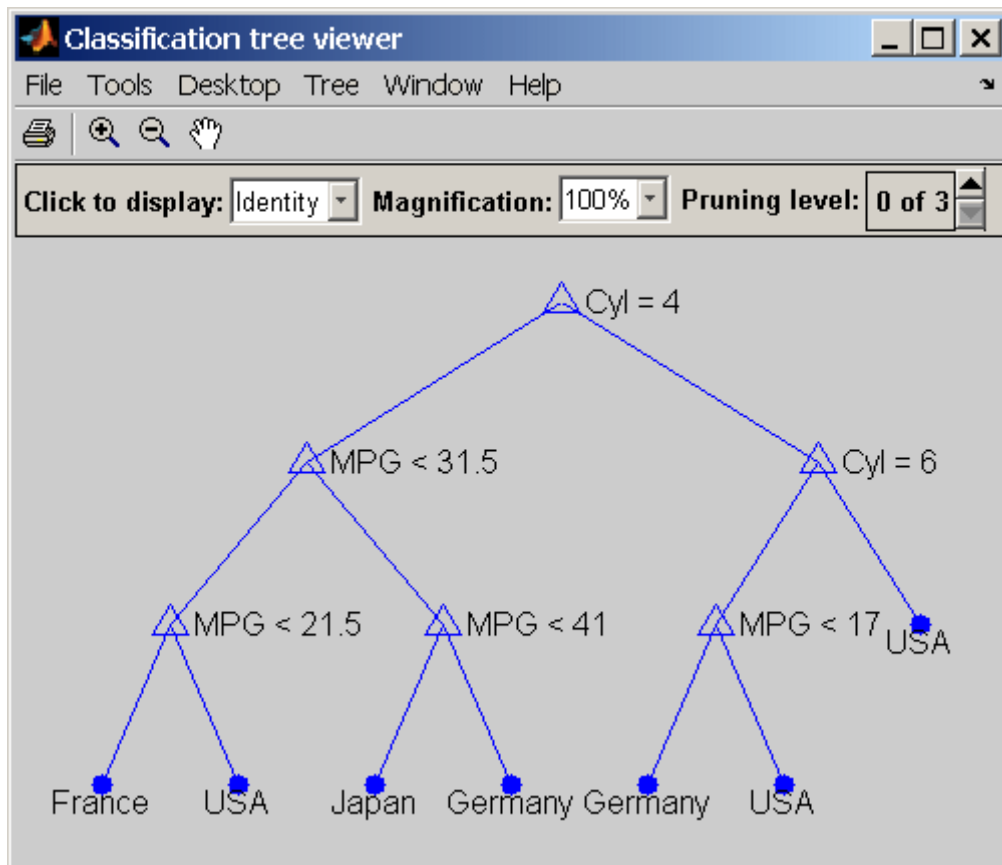

```
C = cutcategories(t)
C =
    [4]      [1x2 double]
    []                []
    [6]      [        8]
    []                []
    []                []
```

```
          [ ]              [ ]
          [ ]              [ ]
          [ ]              [ ]
          [ ]              [ ]
          [ ]              [ ]
          [ ]              [ ]
          [ ]              [ ]
          [ ]              [ ]
      C{1,2}
      ans =
           6      8
```

References   [1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees.* Boca Raton, FL: CRC Press, 1984.

See Also   classregtree, cutpoint, cuttype, cutvar

classregtree.cutpoint

Purpose	Returns decision tree cut point values
Syntax	v = cutpoint(t) v = cutpoint(t,nodes)
Description	v = cutpoint(t) returns an *n*-element vector v of the values used as cut points 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 cut points for the specified nodes.
Examples	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)
```
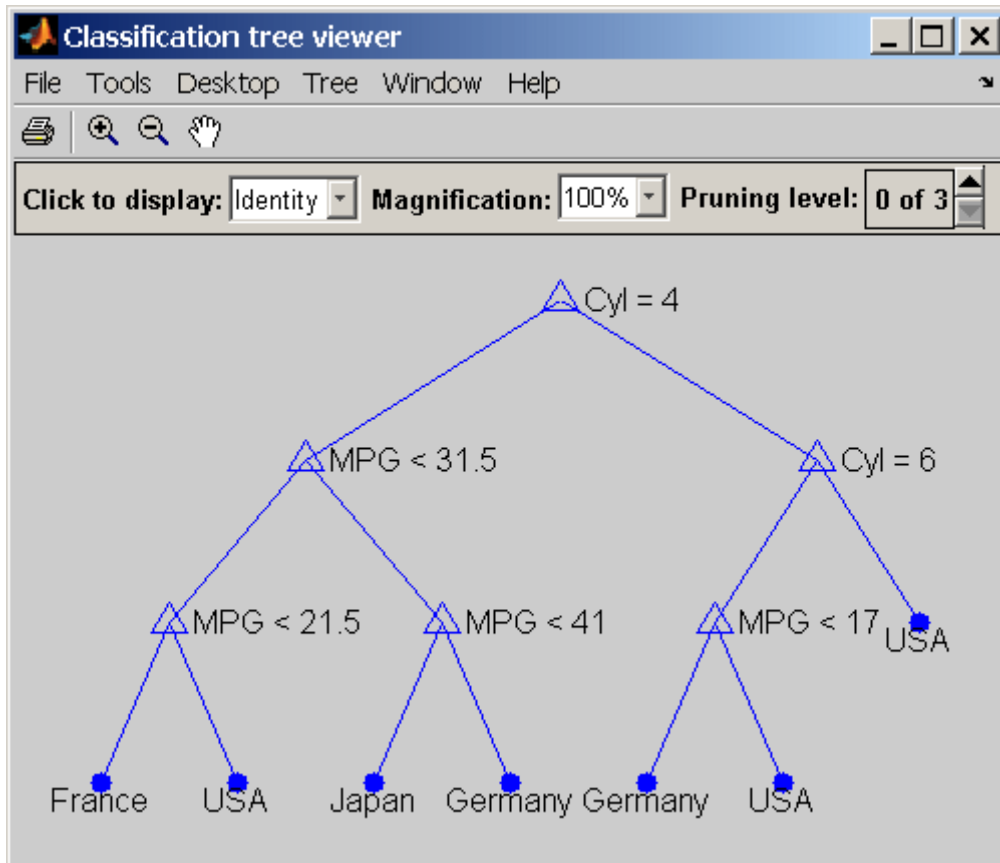


```
v = cutpoint(t)
v =
        NaN
    31.5000
        NaN
    21.5000
    41.0000
```

classregtree.cutpoint

```
   17.0000
      NaN
      NaN
      NaN
      NaN
      NaN
      NaN
      NaN
```

References [1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

See Also classregtree, cutcategories, cuttype, cutvar

Purpose   Cut types

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 cut point 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 cut points 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.

Examples   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)
```
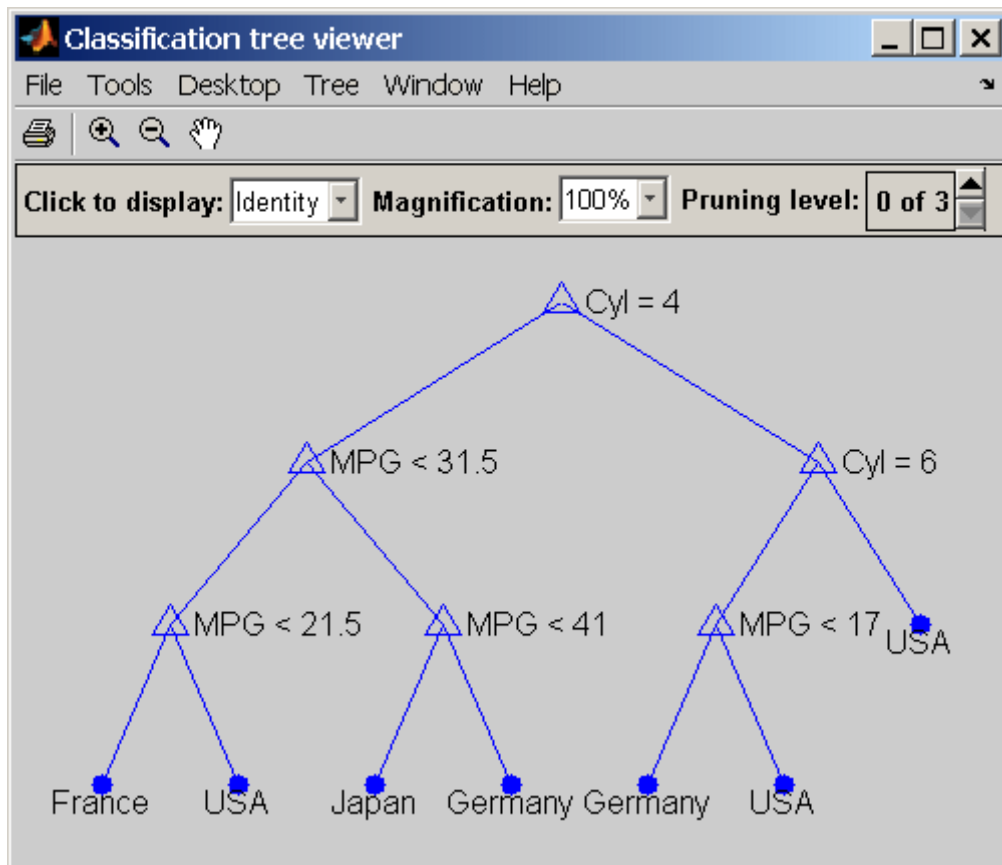


```
c = cuttype(t)
```

```
c =
    'categorical'
    'continuous'
    'categorical'
    'continuous'
    'continuous'
    'continuous'
    ''
    ''
    ''
    ''
    ''
    ''
    ''
```

References   [1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

See Also   classregtree, numnodes, cutvar, cutcategories

classregtree.cutvar

Purpose   Cut variable names

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.

Examples   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)
```
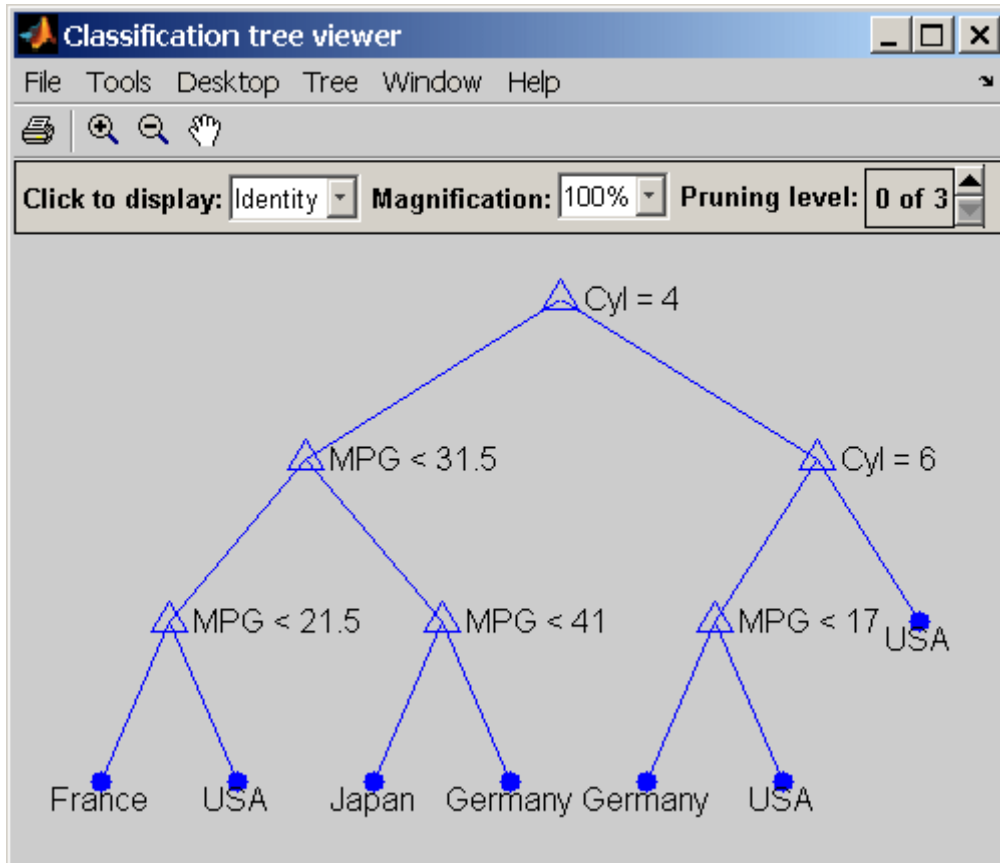


```
[v,num] = cutvar(t)
v =
    'Cyl'
    'MPG'
    'Cyl'
    'MPG'
```

classregtree.cutvar

```
                        'MPG'
                        'MPG'
                        ''
                        ''
                        ''
                        ''
                        ''
                        ''
                        ''
           num =
                2
                1
                2
                1
                1
                1
                0
                0
                0
                0
                0
                0
                0
```

References   [1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

See Also   classregtree, numnodes, children

Purpose Data partitions for cross-validation

Description An object of the `cvpartition` class defines a random partition on a set of data of a specified size. You can be use this partition to define test and training sets for validating a statistical model using cross-validation.

Construction

cvpartition	Create cross-validation partition for data

Methods

disp	Display `cvpartition` object
display	Display `cvpartition` object
repartition	Repartition data for cross-validation
test	Test indices for cross-validation
training	Training indices for cross-validation

Properties

N	Number of observations (including observations with missing `group` values)
NumTestSets	Number of test sets
TestSize	Size of each test set
TrainSize	Size of each training set
Type	Type of partition

Copy Semantics Value. To learn how this affects your use of the class, see Comparing Handle and Value Classes in the MATLAB Object-Oriented Programming documentation.

cvpartition class

Examples   Use a 10-fold stratified cross-validation to compute the misclassification error for `classify` on iris data.

```
load('fisheriris');
CVO = cvpartition(species,'k',10);
err = zeros(CVO.NumTestSets,1);
for i = 1:CVO.NumTestSets
    trIdx = CVO.training(i);
    teIdx = CVO.test(i);
    ytest = classify(meas(teIdx,:),meas(trIdx,:),...
  species(trIdx,:));
    err(i) = sum(~strcmp(ytest,species(teIdx)));
end
cvErr = sum(err)/sum(CVO.TestSize);
```

See Also   "Grouped Data" on page 2-33

crossval

Purpose   Create cross-validation partition for data

Syntax
```
c = cvpartition(n,'kfold',k)
c = cvpartition(group,'kfold',k)
c = cvpartition(n,'holdout',p)
c = cvpartition(group,'holdout',p)
c = cvpartition(n,'leaveout')
c = cvpartition(n,'resubstitution')
```

Description   c = cvpartition(n,'kfold',k) constructs an object c of the cvpartition class defining a random partition for k-fold cross-validation on n observations. The partition divides the observations into k disjoint subsamples (or *folds*), chosen randomly but with roughly equal size. The default value of k is 10.

c = cvpartition(group,'kfold',k) creates a random partition for a stratified k-fold cross-validation. group is a numeric vector, categorical array, string array, or cell array of strings indicating the class of each observation. Each subsample has roughly equal size and roughly the same class proportions as in group. cvpartition treats NaNs or empty strings in group as missing values.

c = cvpartition(n,'holdout',p) creates a random partition for holdout validation on n observations. This partition divides the observations into a training set and a test (or *holdout*) set. The parameter p must be a scalar. When $0 < p < 1$, cvpartition randomly selects approximately p*n observations for the test set. When p is an integer, cvpartition randomly selects p observations for the test set. The default value of p is 1/10.

c = cvpartition(group,'holdout',p) randomly partitions observations into a training set and a test set with stratification, using the class information in group; that is, both training and test sets have roughly the same class proportions as in group.

c = cvpartition(n,'leaveout') creates a random partition for leave-one-out cross-validation on n observations. Leave-one-out is a special case of 'kfold', in which the number of folds equals the number of observations.

cvpartition

c = cvpartition(n,'resubstitution') creates an object c that does not partition the data. Both the training set and the test set contain all of the original n observations.

Examples   Use stratified 10-fold cross-validation to compute misclassification rate:

```
load fisheriris;
y = species;
c = cvpartition(y,'k',10);

fun = @(xT,yT,xt,yt)(sum(~strcmp(yt,classify(xt,xT,yT))));

rate = sum(crossval(fun,meas,y,'partition',c))...
           /sum(c.TestSize)
rate =
    0.0200
```

See Also   Grouped Data

crossval, repartition

Purpose   Arrays for statistical data

Description   Dataset arrays are used to collect heterogeneous data and metadata including into a single container variable. Dataset arrays can be viewed as tables of values, with rows representing different observations and columns representing different measured variables. Dataset arrays can accommodate variables of different types, sizes, units, etc.

Construction

dataset	Construct dataset array

Methods

cat	Concatenate dataset arrays
datasetfun	Apply function to dataset array variables
disp	Display dataset array
display	Display dataset array
double	Convert dataset variables to double array
end	Last index in indexing expression for dataset array
export	Write dataset array to file
get	Access dataset array properties
grpstats	Summary statistics by group for dataset arrays
horzcat	Horizontal concatenation for dataset arrays
isempty	True for empty dataset array
join	Merge observations
length	Length of dataset array

ndims	Number of dimensions of dataset array
numel	Number of elements in dataset array
replacedata	Replace dataset variables
set	Set and display properties
single	Convert dataset variables to single array
size	Size of dataset array
sortrows	Sort rows of dataset array
subsasgn	Subscripted assignment to dataset array
subsref	Subscripted reference for dataset array
summary	Summary statistics for dataset array
unique	Unique observations in dataset array
vertcat	Vertical concatenation for dataset arrays

Properties

description	String describing data set
DimNames	Cell array of strings giving units of variables in data set
ObsNames	Cell array of nonempty, distinct strings giving names of observations in data set
units	Units of variables in data set

| UserData | Variable containing additional information associated with data set |
| VarNames | Cell array giving names of variables in data set |

Copy Semantics
Value. To learn how this affects your use of the class, see Comparing Handle and Value Classes in the MATLAB Object-Oriented Programming documentation.

See Also
"Dataset Arrays" on page 2-23

genvarname, tdfread, textscan, xlsread

dataset

Purpose   Construct 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,param1,val1,param2,val2,...)
A = dataset('xlsfile',filename,param1,val1,param2,val2,...)
```

Description   `A = dataset(VAR1,VAR2,...)` creates dataset array `A` from workspace variables `VAR1`, `VAR2`, `...` using the workspace variable names for the names of the variables in `A`. Variables can be arrays of any size, but all variables must be the same size along dimension 1 (rows).

`A = dataset(...,{VAR,name},...)` creates a variable in dataset `A` from the workspace variable `VAR` and assigns it the name `name` in `A`. Names must be valid, unique MATLAB identifier strings.

`A = dataset(...,{VAR,name_1,...,name_m},...)`, where `VAR` is an array with size `m` along dimension 2 (columns), creates `m` variables in dataset `A` from the columns of the workspace variable `VAR` and assigns them the names `name_1`, ..., `name_m` in `A`.

`A = dataset(...,'VarNames',{name_1,...,name_m},...)` names the `m` variables in `A` with the specified variable names. Names must be valid, unique MATLAB identifier strings. The number of names must equal the number of variables in `A`. You cannot use the `'VarNames'` parameter if you provide names for individual variables using `{VAR,name}` pairs.

`A = dataset(...,'ObsNames',{name_1,...,name_n},...)` names the `n` observations in `A` with the specified observation names. The names need not be valid MATLAB identifier strings, but must be unique. The number of names must equal the number of observations (rows) in `A`.

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,*param1*,*val1*,*param2*,*val2*,...) creates dataset array A from column-oriented data in the text file specified by the string filename. Variables in A are of type `double` if data in the corresponding column of the file, following the column header, are entirely numeric; otherwise the variables in A are cell arrays of strings. `dataset` converts empty fields to either NaN (for a numeric variable) or the empty string (for a string-valued variable). `dataset` ignores insignificant white space in the file.

The following optional parameter name/value pairs are all those `textscan` allows when you use the `'format'` parameter.

`'delimiter'`	A string indicating the character separating columns in the file. Values are `'\t'` (tab, the default), `' '` (space), `','` (comma), `';'` (semicolon), and `'|'` (bar).
`'format'`	A string indicating how `dataset` reads data from the file into the variables in A. Values are conversion specifiers for `textscan`. You may use additional `textscan` parameter/value pairs when using this parameter.
`'ReadVarNames'`	A logical value indicating whether (`true`) or not (`false`) to read variable names from the first row of the file. The default is `true`. If `'ReadVarNames'` is `true`, variable names in the column headers of the file cannot be empty.

dataset

'ReadObsNames' A logical value indicating whether (`true`) or not (`false`)
                to read observation names from the first column of
                the file. The default is `false`. If `'ReadObsNames'` and
                `'ReadVarNames'` are both `true`, `dataset` saves the
                header of the first column in the file as the name of the
                first dimension in `A.Properties.DimNames`.

'TreatAsEmpty' Specifies strings to treat as the empty string in a
                numeric column. Values may be a character string or
                a cell array of strings. The parameter applies only to
                numeric columns in the file; `dataset` does not accept
                numeric literals such as `'-99'`.

`A = dataset('xlsfile',filename,`*`param1`*`,`*`val1`*`,`*`param2`*`,`*`val2`*`,...)`
creates dataset array `A` from column-oriented data in the Excel®
spreadsheet specified by the string `filename`. Variables in `A` are of
type `double` if data in the corresponding column of the spreadsheet,
following the column header, are entirely numeric; otherwise the
variables in `A` are cell arrays of strings. Optional parameter name/value
pairs are as follows:

'sheet'   A positive scalar value of type `double` indicating the
          sheet number, or a quoted string indicating the sheet
          name.

'range'   A string of the form `'C1:C2'` where `C1` and `C2` are the
          names of cells at opposing corners of a rectangular
          region to be read, as for `xlsread`. By default, the
          rectangular region extends to the right-most column
          containing data. If the spreadsheet contains empty
          columns between columns of data, or if the spreadsheet
          contains figures or other non-tabular information,
          specify a range that contains only data.

'ReadVarNames' A logical value indicating whether (`true`) or not (`false`) to read variable names from the first row of the range. The default is `true`. If `'ReadVarNames'` is `true`, variable names in the column headers of the range cannot be empty.

'ReadObsNames' A logical value indicating whether (`true`) or not (`false`) to read observation names from the first column of the range. The default is `false`. If `'ReadObsNames'` and `'ReadVarNames'` are both `true`, the header of the first column in the range is saved as the name of the first dimension in `A.Properties.DimNames`.

Examples

Example 1

Create a dataset array to contain Fisher's iris data:

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

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 "Determining Which Function Gets Called".)

[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` (the default) 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 MATLAB function 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.

Examples   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')
```

dataset.datasetfun
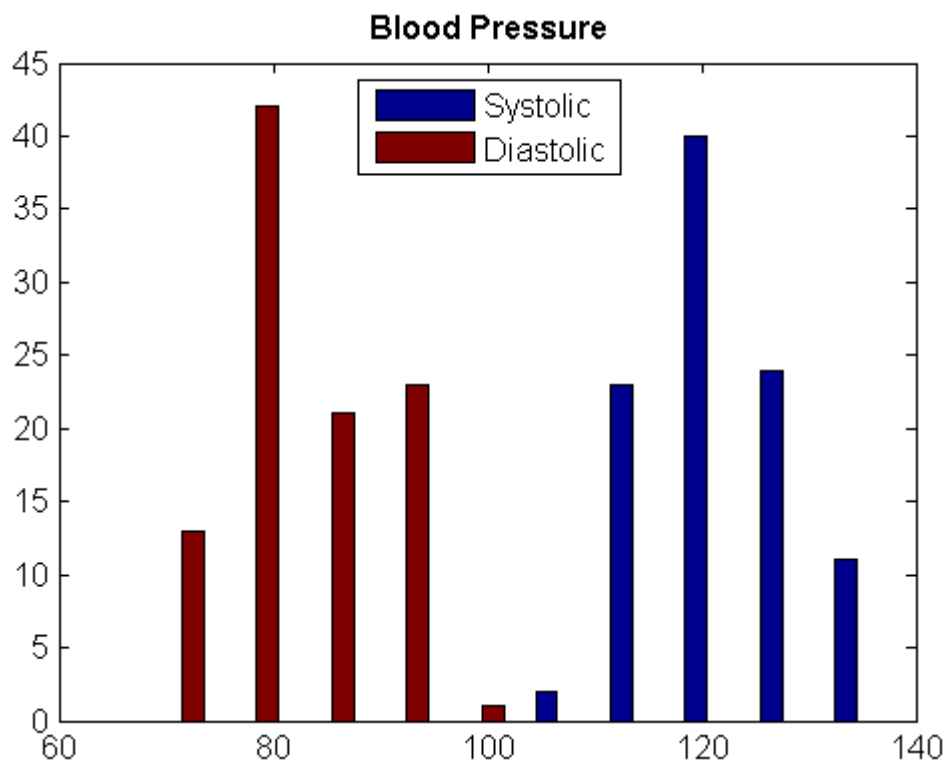
See Also   grpstats

Purpose   *D*-optimal augmentation

Syntax   dCE2 = daugment(dCE,mruns)
                   [dCE2,X] = daugment(dCE,mruns)
                   [dCE2,X] = daugment(dCE,mruns,*model*)
                   [dCE2,X] = daugment(...,*param1*,*val1*,*param2*,*val2*,...)

Description   dCE2 = daugment(dCE,mruns) uses a coordinate-exchange algorithm
                   to *D*-optimally add mruns runs to an existing experimental design dCE
                   for a linear additive model.

                   [dCE2,X] = daugment(dCE,mruns) also returns the design matrix X
                   associated with the augmented design.

                   [dCE2,X] = daugment(dCE,mruns,*model*) uses the linear regression
                   model specified in *model*. *model* is one of the following strings:

                   • 'linear' — Constant and linear terms. This is the default.

                   • 'interaction' — Constant, linear, and interaction terms

                   • 'quadratic' — Constant, linear, interaction, and squared terms

                   • 'purequadratic' — Constant, linear, and squared terms

                   The order of the columns of X for a full quadratic model with *n* terms is:

                   1 The constant term

                   2 The linear terms in order 1, 2, ..., *n*

                   3 The interaction terms 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
                   factor and one row for each term in the model. The entries in any row

of *model* are powers for the factors in the columns. For example, if a model has factors X1, X2, and X3, then a row [0 1 2] in *model* specifies the term (X1.^0).*(X2.^1).*(X3.^2). A row of all zeros in *model* specifies a constant term, which can be omitted.

[dCE2,X] = daugment(...,*param1*,*val1*,*param2*,*val2*,...) specifies additional parameter/value pairs for the design. Valid parameters and their values are listed in the following table.

Parameter	Value
'bounds'	Lower and upper bounds for each factor, specified as a 2-by-nfactors matrix, where nfactors is the number of factors. 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 the iteration counter. The default is 'on'.
'excludefun'	Handle to a function that excludes undesirable runs. If the function is *f*, it must support the syntax $b = f(S)$, where *S* is a matrix of treatments with nfactors columns, where nfactors is the number of factors, and *b* is a vector of Boolean values with the same number of rows as *S*. $b(i)$ is true if the *i*th row *S* should be excluded.
'init'	Initial design as an mruns-by-nfactors matrix, where nfactors is the number of factors. The default is a randomly selected set of points.
'levels'	Vector of number of levels for each factor.

Parameter	Value
`'maxiter'`	Maximum number of iterations. The default is 10.
`'tries'`	Number of times to try to generate a design from a new starting point. The algorithm uses random points for each try, except possibly the first. The default is 1.

Note The daugment function augments an existing design using a coordinate-exchange algorithm; the `'start'` parameter of the candexch function provides the same functionality using a row-exchange algorithm.

Examples   The following eight-run design is adequate for estimating main effects in a four-factor model:

```
dCEmain = cordexch(4,8)
dCEmain =
     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
```

To estimate the six interaction terms in the model, augment the design with eight additional runs:

```
dCEinteraction = daugment(dCEmain,8,'interaction')
dCEinteraction =
     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 augmented design is full factorial, with the original eight runs in the first eight rows.

See Also   dcovary, cordexch, candexch

Purpose   *D*-optimal design with fixed covariates

Syntax
```
dCV = dcovary(nfactors,fixed)
[dCV,X] = dcovary(nfactors,fixed)
[dCV,X] = dcovary(nfactors,fixed,model)
[dCV,X] = daugment(...,param1,val1,param2,val2,...)
```

Description   dCV = dcovary(nfactors,fixed) uses a coordinate-exchange algorithm to generate a *D*-optimal design for a linear additive model with nfactors factors, subject to the constraint that the model include the fixed covariate factors in fixed. The number of runs in the design is the number of rows in fixed. The design dCV augments fixed with initial columns for treatments of the model terms.

[dCV,X] = dcovary(nfactors,fixed) also returns the design matrix X associated with the design.

[dCV,X] = dcovary(nfactors,fixed,*model*) uses the linear regression model specified in *model*. *model* is one of the following strings:

- 'linear' — Constant and linear terms. This is the default.

- 'interaction' — Constant, linear, and interaction terms

- 'quadratic' — Constant, linear, interaction, and squared terms

- 'purequadratic' — Constant, linear, and squared terms

The order of the columns of X for a full quadratic model with *n* terms is:

1 The constant term

2 The linear terms in order 1, 2, ..., *n*

3 The interaction terms 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 factor and one row for each term in the model. The entries in any row of *model* are powers for the factors in the columns. For example, if a model has factors X1, X2, and X3, then a row [0 1 2] in *model* specifies the term (X1.^0).*(X2.^1).*(X3.^2). A row of all zeros in *model* specifies a constant term, which can be omitted.

[dCV,X] = daugment(...,*param1*,*val1*,*param2*,*val2*,...) specifies additional parameter/value pairs for the design. Valid parameters and their values are listed in the following table.

Parameter	Value
'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 the iteration counter. The default is 'on'.
'excludefun'	Handle to a function that excludes undesirable runs. If the function is *f*, it must support the syntax *b = f(S)*, where *S* is a matrix of treatments with nfactors columns and *b* is a vector of Boolean values with the same number of rows as *S*. *b*(*i*) is true if the *i*th row *S* should be excluded.
'init'	Initial design as an mruns-by-nfactors matrix. The default is a randomly selected set of points.
'levels'	Vector of number of levels for each factor.

Parameter	Value
`'maxiter'`	Maximum number of iterations. The default is 10.
`'tries'`	Number of times to try to generate a design from a new starting point. The algorithm uses random points for each try, except possibly the first. The default is 1.

Examples   **Example 1**

Suppose you want a design to estimate the parameters in a three-factor linear additive model, with eight runs that necessarily occur at different times. If the process experiences temporal linear drift, you may want to include the run time as a variable in the model. Produce the design as follows:

```
time = linspace(-1,1,8)';
[dCV1,X] = dcovary(3,time,'linear')
dCV1 =
    -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
X =
     1.0000   -1.0000    1.0000    1.0000   -1.0000
     1.0000    1.0000   -1.0000   -1.0000   -0.7143
     1.0000   -1.0000   -1.0000   -1.0000   -0.4286
     1.0000    1.0000   -1.0000    1.0000   -0.1429
     1.0000    1.0000    1.0000   -1.0000    0.1429
     1.0000   -1.0000    1.0000   -1.0000    0.4286
     1.0000    1.0000    1.0000    1.0000    0.7143
     1.0000   -1.0000   -1.0000    1.0000    1.0000
```

The column vector `time` is a fixed factor, normalized to values between ±1. The number of rows in the fixed factor specifies the number of runs in the design. The resulting design `dCV` gives factor settings for the three controlled model factors at each time.

Example 2

The following example uses the `dummyvar` function to block an eight-run experiment into 4 blocks of size 2 for estimating a linear additive model with two factors:

```
fixed = dummyvar([1 1 2 2 3 3 4 4]);
dCV2 = dcovary(2,fixed(:,1:3),'linear')
dCV2 =
    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 `dCV2` contain the settings for the two factors; the last three columns are dummy variable codings for the four blocks.

See Also   daugment, cordexch, dummyvar

Purpose   Default value returned by `predict`

Description   The `DefaultYfit` property controls what predicted value
                   `CompactTreeBagger` returns when no prediction is possible, for example
                   when the `predict` method needs to predict for an observation which has
                   only false values in the matrix supplied through `'useifort'` argument.

                   For classification, you can set this property to either `''` or
                   `'MostPopular'`. If you choose `'MostPopular'` (default), the property
                   value becomes the name of the most probable class in the training data.

                   For regression, you can set this property to any numeric scalar. The
                   default is the mean of the response for the training data.

See Also   `predict`, `setDefaultYfit`, `TreeBagger.DefaultYfit`.

TreeBagger.DefaultYfit property

Purpose   Default value returned by `predict` and `oobPredict`

Description   The `DefaultYfit` property controls what predicted value `TreeBagger` returns when no prediction is possible, for example when the `oobPredict` method needs to predict for an observation that is in-bag for all trees in the ensemble.

For classification, you can set this property to either `''` or `'MostPopular'`. If you choose `'MostPopular'` (default), the property value becomes the name of the most probable class in the training data.

For regression, you can set this property to any numeric scalar. The default is the mean of the response for the training data.

If you set this property to `''` for classification or `NaN` for regression, `TreeBagger` excludes the in-bag observations from computation of the out-of-bagerror and margin.

See Also   `oobPredict`, `Predict`, `OOBIndices`, `CompactTreeBagger.DefaultYfit`.

Purpose   Delete handle object

Syntax   `delete(h)`

Description   `delete(h)` deletes the handle object h, where h is a scalar handle. The `delete` method deletes a handle object but does not clear the handle from the workspace. A deleted handle is no longer valid.

See Also   `clear`, `isvalid`, `qrandstream`

CompactTreeBagger.DeltaCritDecisionSplit property

Purpose   Split criterion contributions for each predictor

Description The `DeltaCritDecisionSplit` property is a numeric array of size
1-by-`Nvars` of changes in the split criterion summed over splits on each
variable, summed across the entire ensemble of grown trees.

See Also   `TreeBagger.DeltaCritDecisionSplit`, `classregtree.varimportance`

Purpose   Split criterion contributions for each predictor

Description The `DeltaCritDecisionSplit` property is a numeric array of size
1-by-`Nvars` of changes in the split criterion summed over splits on each
variable, summed across the entire ensemble of grown trees.

See Also   `CompactTreeBagger.DeltaCritDecisionSplit`,
`classregtree.varimportance`

dendrogram

Purpose   Dendrogram plot

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. Acceptable values for '*orient*' are:

Value	Description
'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.

Examples

```
X = rand(100,2);
Y = pdist(X,'cityblock');
Z = linkage(Y,'average');
[H,T] = dendrogram(Z,'colorthreshold','default');
set(H,'LineWidth',2)
```

dendrogram
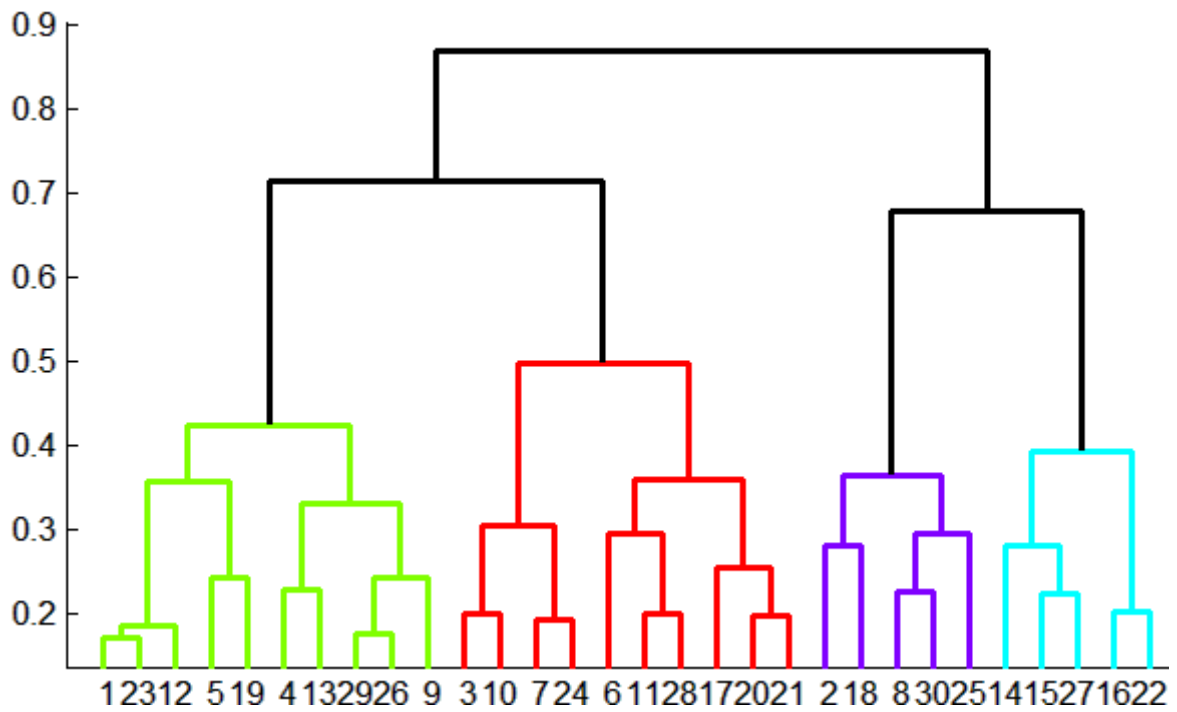

```
find(T==20)
ans =
      20
      49
      62
      65
      73
      96
```

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   String describing data set

Description   `description` is astring describing the data set. The default is an empty
                   string.

dfittool

Purpose   Interactive distribution fitting

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 5-44.

See Also   `mle,randtool,disttool`

Purpose   Number of dimensions

Description   Number of dimensions (positive integer). Read only. Set at construction.

dataset.DimNames property

Purpose　　　Cell array of strings giving units of variables in data set

Description　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.

Purpose   Display categorical array

Syntax   disp(A)

Description   disp(A) prints the categorical array A without printing the array
                   name. In all other ways it's the same as leaving the semicolon off an
                   expression, except that empty arrays don't display.

See Also   categorical, display

classregtree.disp

Purpose   Display classregtree object

Syntax   display(t)

Description   display(t) prints the classregtree object t.

See Also   classregtree, view

Purpose   Display cvpartition object

Syntax   disp(c)

Description   disp(c) prints the cvpartition object c.

See Also   cvpartition

dataset.disp

Purpose   Display dataset array

Syntax   disp(ds)

Description disp(ds) prints the dataset array ds, including variable names and observation names (if present), without printing the dataset name. In all other ways it's the same as leaving the semicolon off an expression.

For numeric or categorical variables that are 2-D and have three or fewer columns, disp prints the actual data using either short g or long g format, depending on the current command line setting. Otherwise, disp prints the size and type of each dataset element.

For character variables that are 2-D and 10 or fewer characters wide, disp prints quoted strings. Otherwise, disp prints the size and type of each dataset element.

For cell variables that are 2-D and have three or fewer columns, disp prints the contents of each cell (or its size and type if too large). Otherwise, disp prints the size of each dataset element.

For time series variables, disp prints columns for both the time and the data. If the variable is 2-D and has three or fewer columns, disp prints the actual data Otherwise, disp prints the size and type of each dataset element.

For other types of variables, disp prints the size and type of each dataset element.

See Also   dataset, display, format

Purpose   Display Gaussian mixture distribution object

Syntax   `disp(obj)`

Description   `disp(obj)` prints a text representation of the `gmdistribution` object, `obj`, without printing the object name. In all other ways it's the same as leaving the semicolon off an expression.

See Also   `gmdistribution`, `display`

NaiveBayes.disp

Purpose   Display NaiveBayes classifier object

Syntax   disp(nb)

Description disp(nb) prints a text representation of the NaiveBayes object nb, without printing the object name. In all other ways it's the same as leaving the semicolon off an expression.

See Also   NaiveBayes, display

Purpose   Display piecewisedistribution object

Syntax   disp(A)

Description   disp(A) prints a text representation of the piecewisedistribution object
                  A, without printing the object name. In all other ways it's the same as
                  leaving the semicolon off an expression.

See Also   piecewisedistribution

qrandset.disp

Purpose   Display qrandset object

Syntax   disp(p)

Description disp(p) displays the properties of the quasi-random point set s, without printing the variable name. disp prints out the number of dimensions and points in the point-set, and follows this with the list of all property values for the object.

See Also   qrandset

Purpose   Display qrandstream object

Syntax   disp(q)

Description   disp(q) displays the quasi-random stream q, without printing the variable name. disp prints the type and number of dimensions in the stream, and follows it with the list of point set properties.

See Also   qrandstream

categorical.display

Purpose   Display categorical array

Syntax   `display(A)`

Description   `display(A)` prints the categorical array A. `categorical` calls`display` when a you do not use a semicolon to terminate a statement.

See Also   `categorical`, `disp`

Purpose	Display `classregtree` object
Syntax	`display(t)` `display(A)`
Description	`display(t)` prints the `classregtree` object `t`. `classregtree` calls`display` when a you do not use a semicolon to terminate a statement.
	`display(A)` prints the categorical array A. `categorical` calls`display` when a you do not use a semicolon to terminate a statement.
See Also	`classregtree`, `eval`, `prune`, `test`

cvpartition.display

Purpose   Display cvpartition object

Syntax   display(c)

Description   display(c) prints the cvpartition object c. cvpartition callsdisplay
when a you do not use a semicolon to terminate a statement.

See Also   cvpartition

Purpose   Display dataset array

Syntax   `display(ds)`

Description   `display(ds)` prints the dataset array `ds`, including variable names and observation names (if present). `dataset` calls `display` when a you do not use a semicolon to terminate a statement

For numeric or categorical variables that are 2-D and have three or fewer columns, `display` prints the actual data. Otherwise, `display` prints the size and type of each dataset element.

For character variables that are 2-D and 10 or fewer characters wide, `display` prints quoted strings. Otherwise, `display` prints the size and type of each dataset element.

For cell variables that are 2-D and have three or fewer columns, `display` prints the contents of each cell (or its size and type if too large). Otherwise, `display` prints the size of each dataset element.

For time series variables, `display` prints columns for both the time and the data. If the variable is 2-D and has three or fewer columns, `display` prints the actual data. Otherwise, `display` prints the size and type of each dataset element.

For other types of variables, `display` prints the size and type of each dataset element.

See Also   `dataset`, `display`, `format`

gmdistribution.display

Purpose   Display Gaussian mixture distribution object

Syntax   display(obj)

Description   display(obj) prints a text representation of the gmdistribution
object obj. gmdistribution calls display when a you do not use a
semicolon to terminate a statement.

See Also   gmdistribution, disp

Purpose   Display NaiveBayes classifier object

Syntax   display(nb)

Description   display(nb) prints a text representation of the NaiveBayes object
nb. NaiveBayes callsdisplay when a you do not use a semicolon to
terminate a statement.

See Also   NaiveBayes, display

piecewisedistribution.display

Purpose   Display piecewisedistribution object

Syntax   display(A)

Description display(A) prints a text representation of the piecewisedistribution object A, without printing the object name. piecewisedistribution calls display when a you do not use a semicolon to terminate a statement.

See Also   piecewisedistribution

Purpose   Read-only string containing probability distribution name of ProbDist object

Description   DistName is a read-only property of the ProbDist class. DistName is a string containing the type of distribution used to create the object.

Values   Possible values are:

- 'kernel'
- 'beta'
- 'binomial'
- 'birnbaumsaunders'
- 'exponential'
- 'extreme value'
- 'gamma'
- 'generalized extreme value'
- 'generalized pareto'
- 'inversegaussian'
- 'logistic'
- 'loglogistic'
- 'lognormal'
- 'nakagami'
- 'negative binomial'
- 'normal'
- 'poisson'
- 'rayleigh'
- 'rician'

- `'tlocationscale'`

- `'weibull'`

Use this information to view and compare the type of distribution used to create distribution objects.

Purpose   Distribution names

Description   The `Dist` property is a string or a 1-by-`NDims` cell array of strings indicating the types of distributions for all the features. If all the features use the same type of distribution, `Dist` is a single string. Otherwise `Dist(j)` indicates the distribution type used for the `j`th feature.

The valid strings for this property are the following:

`'normal'`	Normal distribution.
`'kernel'`	Kernel smoothing density estimate.
`'mvmn'`	Multivariate multinomial distribution.
`'mn'`	Multinomial bag-of-tokens model.

gmdistribution.DistName property

Purpose   Type of distribution

Description   The string `'gaussian mixture distribution'`.

Purpose   Interactive density and distribution 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

categorical.double

Purpose   Convert categorical array to double array

Syntax   B = double(A)

Description B = double(A) converts the categorical array A to a double array. Each element of B contains the internal categorical level code for the corresponding element of A.

See Also   single

Purpose   Convert dataset variables to double array

Syntax   b = double(A)
b = double(a,vars)

Description   b = double(A) returns the contents of the dataset A, converted to one double array. The classes of the variables in the dataset must support the conversion.

b = double(a,vars) returns the contents of the dataset variables 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.

See Also   dataset, single, replacedata

categorical.droplevels

Purpose	Drop levels
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 2-D 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, getlabels, islevel, mergelevels, reorderlevels

dummyvar

Purpose   Create dummy variables

Syntax   D = dummyvar(group)

Description   D = dummyvar(group) creates {0,1}-valued dummy variables for each
of the unique values in group. Columns of group represent categorical
predictor variables, with values indicating categorical levels. Rows of
group represent observations across variables. Each column of D is a
dummy variable for one categorical level of one of the variables in group.

group can be a numeric vector or categorical column vector, representing
levels within a single variable, or a numeric matrix or cell array
of categorical column vectors, representing levels within multiple
variables. If group is a numeric vector or matrix, values in any column
must be positive integers in the range from 1 to the number of levels for
the corresponding variable.

If group is n-by-p, D is n-by-S, where S is the sum of the number of
levels in each of the columns of group. The number of levels s in any
column of group is the maximum positive integer in the column or the
number of categorical levels. Levels are considered distinct if they
appear in different columns of group, even if they have the same value.
Columns of D are, from left to right, dummy variables created from
the first column of group, followed by dummy variables created from
the second column of group, etc.

dummyvar treats NaN values or undefined categorical levels in group as
missing data and returns NaN values in D.

Dummy variables are used in regression analysis and ANOVA to
indicate values of categorical predictors.

Note If a column of 1s is introduced in the matrix D, the resulting matrix X = [ones(size(D,1),1) D] will be rank deficient. The matrix D itself will be rank deficient if group has multiple columns. This is because dummy variables produced from any column of group always sum to a column of 1s. Regression and ANOVA calculations often address this issue by eliminating one dummy variable (implicitly setting the coefficients for dropped columns to zero) from each group of dummy variables produced by a column of group.

Examples   Suppose you are studying the effects of two machines and three operators on a process. Use group to organize predictor data on machine-operator combinations:

```
machine = [1 1 1 1 2 2 2 2]';
operator = [1 2 3 1 2 3 1 2]';
group = [machine operator]
group =
     1     1
     1     2
     1     3
     1     1
     2     2
     2     3
     2     1
     2     2
```

Use dummyvar to create dummy variables for a regression or ANOVA calculation:

```
D = dummyvar(group)
D =
     1     0     1     0     0
     1     0     0     1     0
     1     0     0     0     1
     1     0     1     0     0
```

```
0    1    0    1    0
0    1    0    0    1
0    1    1    0    0
0    1    0    1    0
```

The first two columns of D represent observations of machine 1 and machine 2, respectively; the remaining columns represent observations of the three operators.

See Also   "Grouped Data" on page 2-33

regress, anova1

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 one of the following alternative hypotheses, specified by *tail*:

Tail	Alternative Hypothesis
`'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).

Examples   Fit a straight line to the census data and note the autocorrelation in the residuals load census

```
n = length(cdate);
X = [ones(n,1),cdate];
[b,bint,r1] = regress(pop,X);
```

```
p1 = dwtest(r1,X)
plot(cdate,r1,'b-',cdate,zeros(n,1),'k:')
```

Adding a squared term reduces the autocorrelation but it is still significantly different from zero

```
X = [ones(n,1),cdate,cdate.^2];
[b,bint,r2] = regress(pop,X);
p2 = dwtest(r2,X)
line(cdate,r2,'color','r')
```

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:

| Parameter | Value |
|-----------|-------|
| `'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. |

| Parameter | Value |
|---|---|
| `'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,'LineWidth',2)
hold on
stairs(x,flo,'r:','LineWidth',2)
stairs(x,fup,'r:','LineWidth',2)

% Superimpose a plot of the known population cdf
xx = 0:.1:max(t);
yy = 1-exp(-xx/10);
plot(xx,yy,'g-','LineWidth',2)
legend('Empirical','LCB','UCB','Population',...
       'Location','SE')
hold off
```

References   [1] Cox, D. R., and D. Oakes. *Analysis of Survival Data.* London: Chapman & Hall, 1984.

See Also   cdfplot, ecdfhist

ecdfhist

Purpose	Empirical cumulative distribution function histogram

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.

Examples 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
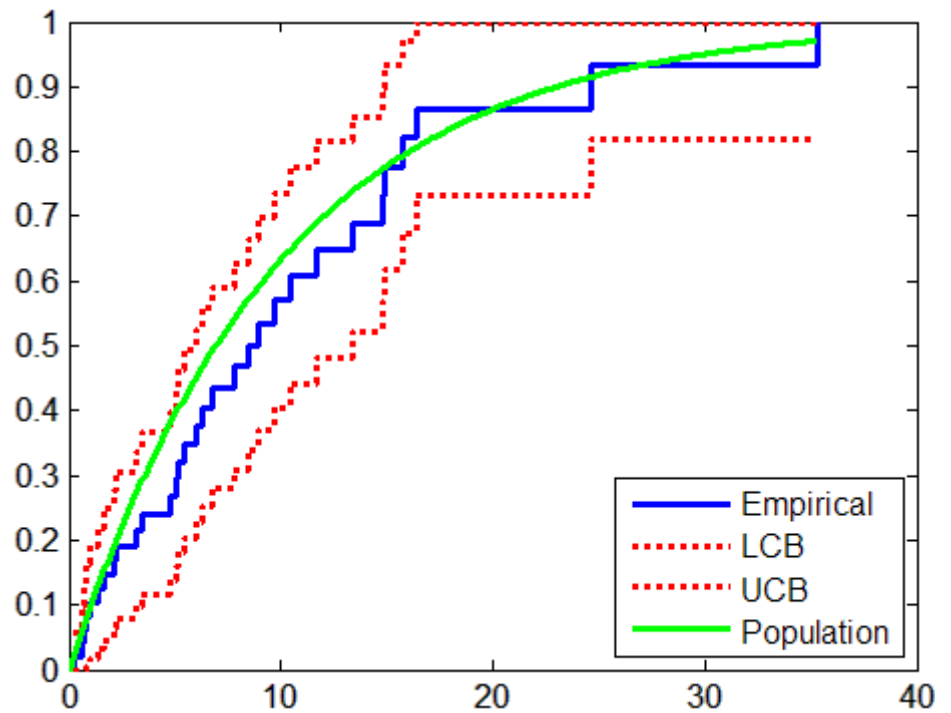
% Calculate the empirical cdf and plot a histogram from it
[f,x] = ecdf(t,'censoring',censored);
ecdfhist(f,x)
```

```
set(get(gca,'Children'),'FaceColor',[.8 .8 1])
hold on

% Superimpose a plot of the known population pdf
xx = 0:.1:max(t);
yy = exp(-xx/10)/10;
plot(xx,yy,'r-','LineWidth',2)
hold off
```


See Also   ecdf, hist, histc

categorical.end

Purpose   Last index in indexing expression for categorical array

Syntax   end(A,k,n)

Description   end(A,k,n) indexes expressions involving the categorical array A
                  when end is part of the k-th index out of n indices. For example, the
                  expression A(end-1,:) calls A's end method with end(A,1,2).

See Also   single

Purpose   Last index in indexing expression for dataset array

Syntax   end(A,k,n)

Description   end(A,k,n) is called for indexing expressions involving the dataset A
when end is part of the k-th index out of n indices. For example, the
expression A(end-1,:) calls A's end method with end(A,1,2).

See Also   size

qrandset.end

Purpose   Last index in indexing expression for point set

Syntax   `end(p,k,n)`

Description   `end(p,k,n)` is called for indexing expressions involving the point set
                   `p` when `end` is part of the k-th index out of n indices. For example, the
                   expression `p(end-1,:)` calls p's `end` method with `end(p,1,2)`.

See Also   `qrandset`

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, evpdf, evinv, evstat, evfit, evlike, evrnd

evfit

Purpose	Extreme value parameter estimates

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   `mle`, `evlike`, `evpdf`, `evcdf`, `evinv`, `evstat`, `evrnd`

evinv

Purpose   Extreme value inverse 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   `icdf`, `evcdf`, `evpdf`, `evstat`, `evfit`, `evlike`, `evrnd`

Purpose   Test handle equality

Syntax   h1 == h2
                 tf = eq(h1, h2)

Description h1 == h2 performs element-wise comparisons between handle arrays
                 h1 and h2. h1 and h2 must be of the same dimensions unless one is a
                 scalar. The result is a logical array of the same dimensions, where each
                 element is an element-wise equality result. If one of h1 or h2 is scalar,
                 scalar expansion is performed and the result will match the dimensions
                 of the array that is not scalar.

                 tf = eq(h1, h2) stores the result in a logical array of the same
                 dimensions.

See Also   qrandstream, ge, gt, le, lt, ne

CompactTreeBagger.error

Purpose   Error (misclassification probability or MSE)

Syntax   err = error(B,X,Y)
                     err = error(B,X,Y,'param1',val1,'param2',val2,...)

Description   err = error(B,X,Y) computes the misclassification probability (for
                     classification trees) or mean squared error (MSE, for regression trees)
                     for each tree, for predictors X given true response Y. For classification, Y
                     can be either a numeric vector, character matrix, cell array of strings,
                     categorical vector or logical vector. For regression, Y must be a numeric
                     vector. err is a vector with one error measure for each of the NTrees
                     trees in the ensemble B.

                     err = error(B,X,Y,'param1',val1,'param2',val2,...) specifies
                     optional parameter name/value pairs:

   'mode'   String indicating how the method computes errors.
                     If set to 'cumulative' (default), error computes
                     cumulative errors and err is a vector of length NTrees,
                     where the first element gives error from trees(1),
                     second element gives error from trees(1:2) etc, up
                     to trees(1:NTrees). If set to 'individual', err
                     is a vector of length NTrees, where each element is
                     an error from each tree in the ensemble. If set to
                     'ensemble', err is a scalar showing the cumulative
                     error for the entire ensemble.

   'trees'   Vector of indices indicating what trees to include
                     in this calculation. By default, this argument is set
                     to 'all' and the method uses all trees. If 'trees'
                     is a numeric vector, the method returns a vector of
                     length NTrees for 'cumulative' and 'individual'
                     modes, where NTrees is the number of elements in the
                     input vector, and a scalar for 'ensemble' mode. For
                     example, in the 'cumulative' mode, the first element
                     gives error from trees(1), the second element gives
                     error from trees(1:2) etc.

'treeweights' Vector of tree weights. This vector must have the same length as the 'trees' vector. The method uses these weights to combine output from the specified trees by taking a weighted average instead of the simple non-weighted majority vote. You cannot use this argument in the 'individual' mode.

'useifort' Logical matrix of size Nobs-by-NTrees indicating which trees should be used to make predictions for each observation. By default the method uses all trees for all observations.

See Also   TreeBagger.error

TreeBagger.error

Purpose   Error (misclassification probability or MSE)

Syntax
```
err = error(B,X,Y)
err = error(B,X,Y,'param1',val1,'param2',val2,...)
```

Description   `err = error(B,X,Y)` computes the misclassification probability for classification trees or mean squared error (MSE) for regression trees for each tree, for predictors `X` given true response `Y`. For classification, `Y` can be either a numeric vector, character matrix, cell array of strings, categorical vector or logical vector. For regression, `Y` must be a numeric vector. `err` is a vector with one error measure for each of the `NTrees` trees in the ensemble `B`.

`err = error(B,X,Y,'param1',val1,'param2',val2,...)` specifies optional parameter name/value pairs:

`'mode'`	String indicating how the method computes errors. If set to `'cumulative'` (default), `error` computes cumulative errors and `err` is a vector of length `NTrees`, where the first element gives error from `trees(1)`, second element gives error from `trees(1:2)` etc, up to `trees(1:NTrees)`. If set to `'individual'`, `err` is a vector of length `NTrees`, where each element is an error from each tree in the ensemble. If set to `'ensemble'`, `err` is a scalar showing the cumulative error for the entire ensemble.
`'trees'`	Vector of indices indicating what trees to include in this calculation. By default, this argument is set to `'all'` and the method uses all trees. If `'trees'` is a numeric vector, the method returns a vector of length `NTrees` for `'cumulative'` and `'individual'` modes, where `NTrees` is the number of elements in the input vector, and a scalar for `'ensemble'` mode. For example, in the `'cumulative'` mode, the first element gives error from `trees(1)`, the second element gives error from `trees(1:2)` etc.

'treeweights'	Vector of tree weights. This vector must have the same length as the 'trees' vector. The method uses these weights to combine output from the specified trees by taking a weighted average instead of the simple non-weighted majority vote. You cannot use this argument in the 'individual' mode.
'useifort'	Logical matrix of size Nobs-by-NTrees indicating which trees should be used to make predictions for each observation. By default the method uses all trees for all observations.

See Also   CompactTreeBagger.error

classregtree.eval

Purpose   Predicted responses

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 jth
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.

Examples 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
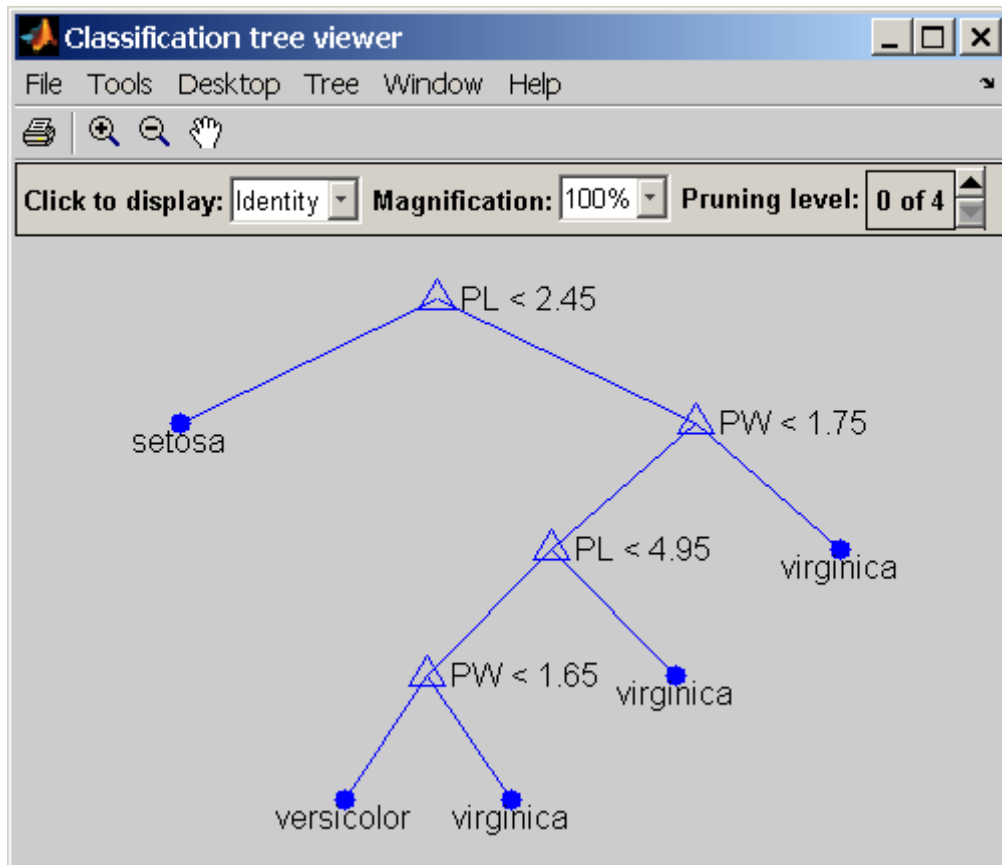
view(t)
```

Find assigned class names:

```
sfit = eval(t,meas);
```

Compute that proportion is correctly classified:

```
pct = mean(strcmp(sfit,species))
pct =
    0.9800
```

References   [1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees.* Boca Raton, FL: CRC Press, 1984.

See Also   classregtree, prune, test, view

evlike

Purpose	Extreme value negative log-likelihood

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	`evfit`, `evpdf`, `evcdf`, `evinv`, `evstat`, `evrnd`

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** | `pdf`, `evcdf`, `evinv`, `evstat`, `evfit`, `evlike`, `evrnd` |

evrnd

Purpose	Extreme value random numbers

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	`random`, `evpdf`, `evcdf`, `evinv`, `evstat`, `evfit`, `evlike`

Purpose   Extreme value mean and variance

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   `evpdf`, `evcdf`, `evinv`, `evfit`, `evlike`, `evrnd`

expcdf

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, exppdf, expinv, expstat, expfit, explike, exprnd

expfit

Purpose	Exponential parameter estimates

Syntax

```
muhat = expfit(data)
[muhat,muci] = expfit(data)
[muhat,muci] = expfit(data,alpha)
[...] = expfit(data,alpha,censoring)
[...] = expfit(data,alpha,censoring,freq)
```

Description

muhat = expfit(data) estimates the mean of an exponentially distributed sample data. Each entry of muhat corresponds to the data in a column of data.

[muhat,muci] = expfit(data) returns 95% confidence intervals for the parameter estimates in matrix muci. The first row of muci contains the lower bounds of the confidence intervals, and the second row contains the upper bounds.

[muhat,muci] = 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.

Examples

The following estimates the mean mu of exponentially distributed data, and returns a 95% confidence interval for the estimate:

```
mu = 3;
data = exprnd(mu,100,1); % Simulated data
```

```
[muhat,muci] = expfit(data)
muhat =
    2.7511
muci =
    2.2826
    3.3813
```

See Also   mle, explike, exppdf, expcdf, expinv, expstat, exprnd

expinv

Purpose

Exponential inverse 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 $\mu = 700$ 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   `icdf`, `expcdf`, `exppdf`, `expstat`, `expfit`, `explike`, `exprnd`

explike

Purpose	Exponential negative log-likelihood

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

expfit, exppdf, expcdf, expinv, expstat, exprnd

Purpose   Write dataset array to file

Syntax
```
export(DS,'file',filename)
export(DS)
export(DS,'file',filename,'Delimiter',DELIM)
export(DS,'XLSfile',filename)
export(DS,...,'WriteVarNames',false)
export(DS,...,'WriteObsNames',false)
```

Description   `export(DS,'file',filename)` writes the dataset array DS to a tab-delimited text file, including variable names and observation names, if present. If the observation names exist, the name in the first column of the first line of the file is the first dimension name for the dataset (by default, `'Observations'`). `export` overwrites any existing file named filename.

`export(DS)` writes to a text file whose default name is the name of the dataset array DS appended by `'.txt'`. If `export` cannot construct the file name from the dataset array input, it writes to the file `'dataset.txt'`. `export` overwrites any existing file.

`export(DS,'file',filename,'Delimiter',DELIM)` writes the dataset array DS to a text file using the delimiter DELIM. DELIM must be one of the following:

- `' '` or `'space'`

- `'\t'` or `'tab'`

- `','` or `'comma'`

- `';'` or `'semi'`

- `'|'` or `'bar'`

`export(DS,'XLSfile',filename)` writes the dataset array DS to an Microsoft® Excel® spreadsheet file, including variable names and observation names (if present). You may also specify the `'Sheet'` and `'Range'` parameter name/value pairs, with parameter values as accepted by the `XLSREAD` function.

dataset.export

export(DS,...,'WriteVarNames',false) does not write the variable names to the text file. export(DS,...,'WriteVarNames',true) is the default, writing the names as column headings in the first line of the file.

export(DS,...,'WriteObsNames',false) does not write the observation names to the text file. export(DS,...,'WriteVarNames',true) is the default, writing the names as the first column of the file.

In some cases, export creates a text file that does not represent A exactly, as described below. If you use dataset to read the file back into MATLAB, the new dataset array may not have exactly the same contents as the original dataset array. Save A as a MAT-file if you need to import it again as a dataset array.

export writes out numeric variables using long g format, and categorical or character variables as unquoted strings. For non-character variables with more than one column, export writes out multiple delimiter-separated fields on each line, and constructs suitable column headings for the first line of the file. export writes out both the time and the data fields of timeseries variables, as separate columns. export writes out variables that have more than two dimensions as a single empty field in each line of the file. For cell-valued variables, export writes out the contents of each cell only when the cell contains a single row, and writes out a single empty field otherwise.

In some cases, export may create a file that cannot be read back into MATLAB using dataset. Writing a dataset array that contains a cell-valued variable whose cell contents are not scalars will result in a mismatch in the file between the number of fields on each line and the number of column headings on the first line. Writing a dataset array that contains a cell-valued variable whose cell contents are not all the same length will result in a different number of fields on each line in the file.

Examples   Move data between external text files and dataset arrays in the MATLAB workspace:

```
A = dataset('file','sat2.dat','delimiter',',')
```

17-358

```
A =
    Test                    Gender          Score
    'Verbal'                'Male'          470
    'Verbal'                'Female'        530
    'Quantitative'          'Male'          520
    'Quantitative'          'Female'        480

export(A(A.Score > 500,:),'file','HighScores.txt')

B = dataset('file','HighScores.txt','delimiter','\t')
B =
    Test                    Gender          Score
    'Verbal'                'Female'        530
    'Quantitative'          'Male'          520
```

See Also   dataset

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   pdf, expcdf, expinv, expstat, expfit, explike, exprnd

Purpose   Exponential random numbers

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   random, exppdf, expcdf, expinv, expstat, expfit, explike

expstat

Purpose	Exponential mean and variance
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	`exppdf`, `expcdf`, `expinv`, `expfit`, `explike`, `exprnd`

Purpose   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

$$\text{cov}(x) = \Lambda \Lambda^T + \Psi$$

where $\Psi = \text{cov}(e)$ 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 jth factor for the ith
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.

factoran

`[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:

Field	Description
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 17-370 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][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.

factoran

Parameter	Value	
'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.
	'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.

Parameter	Value	
	'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.
	'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.

factoran

Parameter	Value	
	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" on page 17-375Example 4.
`'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'`.	

Parameter	Value
'power'	Exponent for creating the target matrix in the 'promax' rotation. Must be ≥ 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" on page 17-375.
'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.
'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 "Example 1" on page 17-3701 and "Example 3" on page 17-3723.

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
       'LineWidth',2,...
       'MarkerSize',20)
```

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
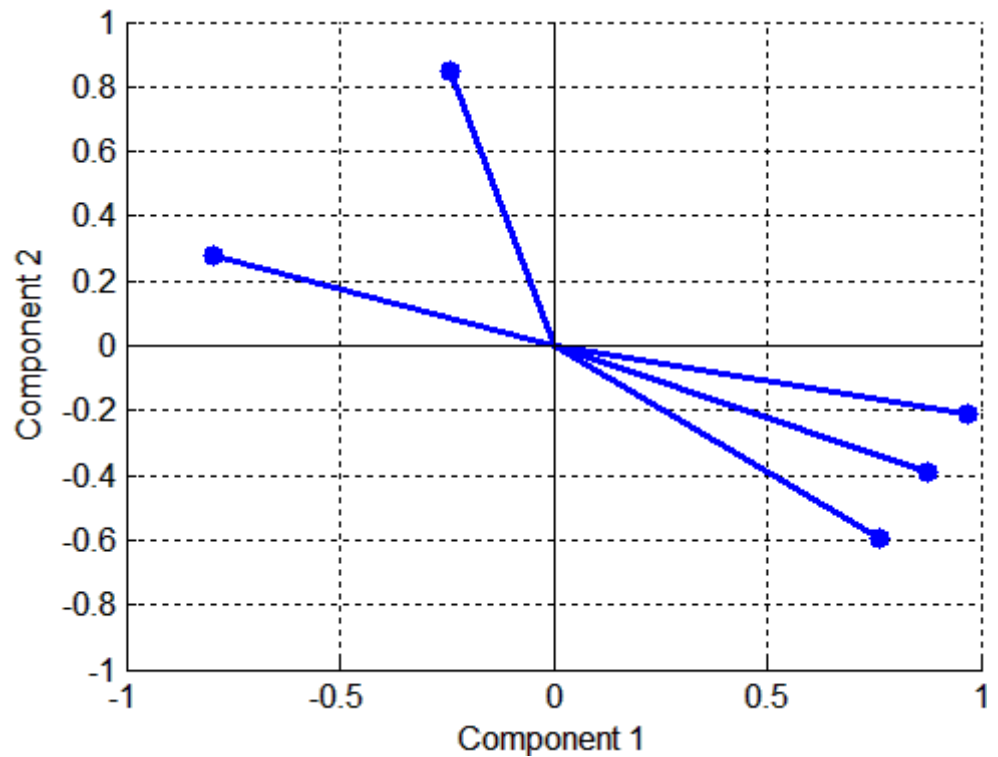
line([-invT(1,1) invT(1,1) NaN -invT(2,1) invT(2,1)], ...
     [-invT(1,2) invT(1,2) NaN -invT(2,2) invT(2,2)], ...
     'Color','r','linewidth',2)
hold on
biplot(Lambda0,...
       'LineWidth',2,...
       'MarkerSize',20)
xlabel('Loadings for unrotated Factor 1')
ylabel('Loadings for unrotated Factor 2')
```

factoran

Plot the rotated variables against the oblique axes:

```
biplot(Lambda,'LineWidth',2,'MarkerSize',20)
```

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. Chicago: University of Chicago Press, 1976.

[2] Jöreskog, K. G. "Some Contributions to Maximum Likelihood Factor Analysis." *Psychometrika*. Vol. 32, Issue 4, 1967, pp. 443–482.

factoran

[3] Lawley, D. N., and A. E. Maxwell. *Factor Analysis as a Statistical Method*. 2nd Ed. New York: American Elsevier Publishing Co., 1971.

See Also   biplot, princomp, procrustes, pcacov, rotatefactors, statset

Purpose　　　Fraction of in-bag observations

Description　　The FBoot property is the fraction of observations to be randomly selected with replacement for each bootstrap replica. The size of each replica is given by n*FBoot, where n is the number of observations in the training set. The default value is 1.

fcdf

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 are all 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

$$p = F(x|v_1,v_2) = \int_0^x \frac{\Gamma\left[\frac{(v_1+v_2)}{2}\right]}{\Gamma\left(\frac{v_1}{2}\right)\Gamma\left(\frac{v_2}{2}\right)}\left(\frac{v_1}{v_2}\right)^{\frac{v_1}{2}}\frac{t^{\frac{v_1-2}{2}}}{\left[1+\left(\frac{v_1}{v_2}\right)t\right]^{\frac{v_1+v_2}{2}}}dt$$

The result, *p*, is the probability that a single observation from an *F*
distribution with parameters v_1 and v_2 will fall in the interval [0 *x*].

Examples   The following illustrates a 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`, `fpdf`, `finv`, `fstat`, `frnd`

ff2n

Purpose   Two-level full factorial design

Syntax   dFF2 = ff2n(n)

Description   dFF2 = ff2n(n) gives factor settings dFF2 for a two-level full factorial
design with n factors. dFF2 is *m*-by-n, where *m* is the number of
treatments in the full-factorial design. Each row of dFF2 corresponds
to a single treatment. Each column contains the settings for a single
factor, with values of 0 and 1 for the two levels.

Examples
```
dFF2 = ff2n(3)
dFF2 =
   0   0   0
   0   0   1
   0   1   0
   0   1   1
   1   0   0
   1   0   1
   1   1   0
   1   1   1
```

See Also   fullfact

TreeBagger.fillProximities

Purpose	Proximity matrix for training data
Syntax	`B = fillProximities(B)` `B = fillProximities(B,'param1',val1,'param2',val2,...`
Description	`B = fillProximities(B)` computes a proximity matrix for the training data and stores it in the Properties field of B. `B = fillProximities(B,'param1',val1,'param2',val2,...)` specifies optional parameter name/value pairs:

`'trees'`	Either `'all'` or a vector of indices of the trees in the ensemble to be used in computing the proximity matrix. Default is `'all'`.
`'nprint'`	Number of training cycles (grown trees) after which `TreeBagger` displays a diagnostic message showing training progress. Default is no diagnostic messages.

See Also	`CompactTreeBagger.outlierMeasure`, `CompactTreeBagger.proximity`

Purpose   Find objects matching specified conditions

Syntax   hm = findobj(h, 'conditions')

Description   The findobj method of the handle class follows the same syntax as
                the MATLAB findobj command, except that the first argument must
                be an array of handles to objects.

                hm = findobj(h, 'conditions') searches the handle object array
                h and returns an array of handle objects matching the specified
                conditions. Only the public members of the objects of h are considered
                when evaluating the conditions.

See Also   findobj, qrandstream

qrandstream.findprop

Purpose	Find property of MATLAB handle object
Syntax	`p = findprop(h,'propname')`
Description	`p = findprop(h,'propname')` finds and returns the `meta.property` object associated with property name `propname` of scalar handle object `h`. `propname` must be a string. It can be the name of a property defined by the class of `h` or a dynamic property added to scalar object `h`. If no property named `propname` exists for object `h`, an empty `meta.property` array is returned.
See Also	`dynamicprops`, `findobj`, `meta.property`, `qrandstream`

finv

Purpose   *F* inverse 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   icdf, fcdf, fpdf, fstat, frnd

gmdistribution.fit

Purpose   Gaussian mixture parameter estimates

Syntax   obj = gmdistribution.fit(X,k)
                obj = gmdistribution.fit(...,*param1*,*val1*,*param2*,*val2*,...)

Description obj = gmdistribution.fit(X,k) uses the Expectation Maximization
                (EM) algorithm to construct an object obj of the gmdistribution
                class containing maximum likelihood estimates of the parameters in
                a Gaussian mixture model with k components for data in the *n*-by-*d*
                matrix X, where *n* is the number of observations and *d* is the dimension
                of the data.

                gmdistribution treats NaN values as missing data. Rows of X with NaN
                values are excluded from the fit.

                obj = gmdistribution.fit(...,*param1*,*val1*,*param2*,*val2*,...)
                provides control over the iterative EM algorithm. Parameters and
                values are listed below.

Parameter	Value
'Start'	Method used to choose initial component parameters. One of the following:
	• 'randSample' — To select k observations from X at random as initial component means. The mixing proportions are uniform. The initial covariance matrices for all components are diagonal, where the element j on the diagonal is the variance of X(:,j). This is the default.
	• S — A structure array with fields mu, Sigma, and PComponents. See gmdistribution for descriptions of values.
	• s — A vector of length *n* containing an initial guess of the component index for each point.

Parameter	Value
'Replicates'	A positive integer giving the number of times to repeat the EM algorithm, each time with a new set of parameters. The solution with the largest likelihood is returned. A value larger than 1 requires the 'randSample' start method. The default is 1.
'CovType'	'diagonal' if the covariance matrices are restricted to be diagonal; 'full' otherwise. The default is 'full'.
'SharedCov'	Logical true if all the covariance matrices are restricted to be the same (pooled estimate); logical false otherwise.
'Regularize'	A nonnegative regularization number added to the diagonal of covariance matrices to make them positive-definite. The default is 0.
'Options'	Options structure for the iterative EM algorithm, as created by statset. gmdistribution.fit uses the parameters 'Display' with a default value of 'off', 'MaxIter' with a default value of 100, and 'TolFun' with a default value of 1e6.

Examples   Generate data from a mixture of two bivariate Gaussian distributions using the mvnrnd function:

```
MU1 = [1 2];
SIGMA1 = [2 0; 0 .5];
MU2 = [-3 -5];
SIGMA2 = [1 0; 0 1];
X = [mvnrnd(MU1,SIGMA1,1000);mvnrnd(MU2,SIGMA2,1000)];

scatter(X(:,1),X(:,2),10,'.')
hold on
```

Next, fit a two-component Gaussian mixture model:

```
options = statset('Display','final');
obj = gmdistribution.fit(X,2,'Options',options);
10 iterations, log-likelihood = -7046.78

h = ezcontour(@(x,y)pdf(obj,[x y]),[-8 6],[-8 6]);
```

Among the properties of the fit are the parameter estimates:

```
ComponentMeans = obj.mu
ComponentMeans =
    0.9391    2.0322
   -2.9823   -4.9737

ComponentCovariances = obj.Sigma
ComponentCovariances(:,:,1) =
    1.7786   -0.0528
   -0.0528    0.5312
ComponentCovariances(:,:,2) =
    1.0491   -0.0150
   -0.0150    0.9816
```

```
MixtureProportions = obj.PComponents
MixtureProportions =
    0.5000    0.5000
```

The Akaike information is minimized by the two-component model:

```
AIC = zeros(1,4);
obj = cell(1,4);
for k = 1:4
    obj{k} = gmdistribution.fit(X,k);
    AIC(k)= obj{k}.AIC;
end

[minAIC,numComponents] = min(AIC);
numComponents
numComponents =
     2

model = obj{2}
model =
Gaussian mixture distribution
with 2 components in 2 dimensions
Component 1:
Mixing proportion: 0.500000
Mean:     0.9391    2.0322
Component 2:
Mixing proportion: 0.500000
Mean:    -2.9823   -4.9737
```

Both the Akaike and Bayes information are negative log-likelihoods for the data with penalty terms for the number of estimated parameters. They are often used to determine an appropriate number of components for a model when the number of components is unspecified.

References   [1] McLachlan, G., and D. Peel. *Finite Mixture Models.* Hoboken, NJ: John Wiley & Sons, Inc., 2000.

See Also   gmdistribution, cluster

NaiveBayes.fit

Purpose	Create Naive Bayes classifier object by fitting training data
Syntax	`nb = NaiveBayes.fit(training, class)` `nb = NaiveBayes.fit(..., 'param1',val1, 'param2',val2, ...)`

Description   `nb = NaiveBayes.fit(training, class)` builds a `NaiveBayes` classifier object `nb`. `training` is an N-by-D numeric matrix of training data. Rows of `training` correspond to observations; columns correspond to features. `class` is a classing variable for `training` (see "Grouped Data" on page 2-33) taking K distinct levels. Each element of `class` defines which class the corresponding row of `training` belongs to. `training` and `class` must have the same number of rows.

`nb = NaiveBayes.fit(..., 'param1',val1, 'param2',val2, ...)` specifies one or more of the following name/value pairs:

- `'Distribution'` – a string or a 1-by-D cell vector of strings, specifying which distributions `fit` uses to model the data. If the value is a string, `fit` models all the features using one type of distribution. `fit` can also model different features using different types of distributions. If the value is a cell vector, its jth element specifies the distribution `fit` uses for the jth feature. The available types of distributions are:

`'normal'` (default)	Normal (Gaussian) distribution.
`'kernel'`	Kernel smoothing density estimate.

'mvmn'	Multivariate multinomial distribution for discrete data. fit assumes each individual feature follows a multinomial model within a class. The parameters for a feature include the probabilities of all possible values that the corresponding feature can take.
'mn'	Multinomial distribution for classifying the count-based data such as the bag-of-tokens model. In the bag-of-tokens model, the value of the jth feature is the number of occurrences of the jth token in this observation, so it must be a non-negative integer. When 'mn' is used, fit considers each observation as multiple trials of a multinomial distribution, and considers each occurrence of a token as one trial. The number of categories (bins) in this multinomial model is the number of distinct tokens, i.e., the number of columns of training.

- 'Prior' – The prior probabilities for the classes, specified as one of the following:

'empirical' (default)	fit estimates the prior probabilities from the relative frequencies of the classes in training.
'uniform'	The prior probabilities are equal for all classes.

vector	A numeric vector of length K specifying the prior probabilities in the class order of class.
structure	A structure S containing class levels and their prior probabilities. S must have two fields:

- S.prob: A numeric vector of prior probabilities.

- S.class: A vector of the same type as class, containing unique class levels indicating the class for the corresponding element of prob. S.class must contain all the K levels in class. It can also contain classes that do not appear in class. This can be useful if training is a subset of a larger training set. fit ignores any classes that appear in S.class but not in class.

If the prior probabilities don't sum to one, fit will normalize them.

- 'KSWidth' – The bandwidth of the kernel smoothing window. The default is to select a default bandwidth automatically for each combination of feature and class, using a value that is optimal for a Gaussian distribution. You can specify the value as one of the following:

scalar	Width for all features in all classes.
row vector	1-by-D vector where the jth element is the bandwidth for the jth feature in all classes.
column vector	K-by-1 vector where the ith element specifies the bandwidth for all features in the ith class. K represents the number of class levels.

matrix   K-by-D matrix M where M(i,j) specifies the bandwidth for the jth feature in the ith class.

structure   A structure S containing class levels and their bandwidths. S must have two fields:

- S.width – A numeric array of bandwidths specified as a row vector, or a matrix with D columns.

- S.class – A vector of the same type as class, containing unique class levels indicating the class for the corresponding row of width.

- 'KSSupport' – The regions where the density can be applied. It can be a string, a two-element vector as shown below, or a 1-by-D cell array of these values:

'unbounded' (default)   The density can extend over the whole real line.

'positive'   The density is restricted to positive values.

[L,U]   A two-element vector specifying the finite lower bound L and upper bound U for the support of the density.

- 'KSType' – The type of kernel smoother to use. It can be a string or a 1-by-D cell array of strings. Each string can be 'normal' (default), 'box', 'triangle', or 'epanechnikov'.

See Also   "Naive Bayes Classification" on page 11-6, "Grouped Data" on page 2-33

fitdist

Purpose	Fit probability distribution to data

Syntax

PD = fitdist(*X*, *DistName*)
[*PDCA*, *GN*, *GL*] = fitdist(*X*, *DistName*, 'By', *GroupVar*)
... = fitdist(..., *param1*, *val1*, *param2*, *val2*, ...)

Description

PD = fitdist(*X*, *DistName*) fits the probability distribution specified by *DistName* to the data in the column vector *X*, and returns *PD*, an object representing the fitted distribution.

[*PDCA*, *GN*, *GL*] = fitdist(*X*, *DistName*, 'By', *GroupVar*) takes a grouping variable, *GroupVar*, fits the specified distribution to the data in *X* from each group, and returns *PDCA*, a cell array of the fitted probability distribution objects. *GroupVar* can also be a cell array of multiple grouping variables. *GN* is a cell array of group labels. *GL* is a cell array of grouping variable levels, with one column for each grouping variable.

... = fitdist(..., *param1*, *val1*, *param2*, *val2*, ...) specifies optional parameter name/value pairs, as described in the Parameter/Values table. Parameter and value names are case insensitive.

Inputs

X A column vector of data.

> **Note** Any NaN values in *X* are ignored by the fitting calculations. Additionally, any NaN values in the censoring vector or frequency vector will cause the corresponding values in *X* to be ignored by the fitting calculations.

DistName A string specifying a distribution. Choices are:

- 'kernel' — To fit a nonparametric kernel-smoothing distribution.

- Any of the following to fit a parametric distribution:

  - `'beta'`
  - `'binomial'`
  - `'birnbaumsaunders'`
  - `'exponential'`
  - `'extreme value'` or `'ev'`
  - `'gamma'`
  - `'generalized extreme value'` or `'gev'`
  - `'generalized pareto'` or `'gp'`
  - `'inversegaussian'`
  - `'logistic'`
  - `'loglogistic'`
  - `'lognormal'`
  - `'nakagami'`
  - `'negative binomial'` or `'nbin'`
  - `'normal'`
  - `'poisson'`
  - `'rayleigh'`
  - `'rician'`
  - `'tlocationscale'`
  - `'weibull'` or `'wbl'`

For more information on these parametric distributions, see Appendix B, "Distribution Reference".

GroupVar	A grouping variable or a cell array of multiple grouping variables. For more information on grouping variables, see "Grouped Data" on page 2-33.

Parameter	Values
'censoring'	A Boolean vector the same size as *X*, containing 1s when the corresponding elements in *X* are right-censored observations and 0s when the corresponding elements are exact observations. Default is a vector of 0s. **Note** Any NaN values in this censoring vector are ignored by the fitting calculations. Additionally, any NaN values in *X* or the frequency vector will cause the corresponding values in the censoring vector to be ignored by the fitting calculations.
'frequency'	A vector the same size as *X*, containing nonnegative integers specifying the frequencies for the corresponding elements in *X*. Default is a vector of 1s. **Note** Any NaN values in this frequency vector are ignored by the fitting calculations. Additionally, any NaN values in *X* or the censoring vector will cause the corresponding values in the frequency vector to be ignored by the fitting calculations.

Parameter	Values
'options'	A structure created by the statset function to specify control parameters for the iterative fitting algorithm.
'n'	For 'binomial' distributions only, a positive integer specifying the N parameter (number of trials).
'theta'	For 'generalized pareto' distributions only, value specifying the theta (threshold) parameter for the generalized Pareto distribution. Default is 0.
'kernel'	For 'kernel' distributions only, a string specifying the type of kernel smoother to use. Choices are: • 'normal' (default) • 'box' • 'triangle' • 'epanechnikov'
'support'	For 'kernel' distributions only, any of the following to specify the support: • 'unbounded' — Default. If the density can extend over the whole real line. • 'positive' — To restrict it to positive values. • A two-element vector giving finite lower and upper limits for the support of the density.
'width'	For 'kernel' distributions only, a value specifying 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.

fitdist

Outputs

PD	An object in either the `ProbDistUnivKernel` class or the `ProbDistUnivParam` class, which are derived from the `ProbDist` class.
PDCA	A cell array of the fitted probability distribution objects.
GN	A cell array of group labels.
GL	A cell array of grouping variable levels, with one column for each grouping variable.

Examples

Creating a ProbDistUnivKernel Object

1 Load a MAT-file, included with the Statistics Toolbox software, which contains MPG, a column vector of data.

```
load carsmall
```

2 Create a ProbDistUnivKernel object by fitting a nonparametric kernel-smoothing distribution to the data:

```
ksd = fitdist(MPG,'kernel')

ksd =

kernel distribution

    Kernel = normal
    Bandwidth = 4.11428
    Support = unbounded
```

Creating a ProbDistUnivParam Object

1 Load a MAT-file, included with the Statistics Toolbox software, which contains MPG, a column vector of data, and Origin, a cell array of seven grouping variables.

```
load carsmall
```

2 Create a cell array of ProbDistUnivParam objects by fitting a parametric distribution, namely a Weibull distribution, to the data, and also grouping the data.

```
wd = fitdist(MPG,'weibull','by',Origin)
```

Algorithm   The fitdist function fits most distributions using maximum likelihood. Two exceptions are the normal and lognormal distributions with uncensored data. For the uncensored normal distribution, the estimated value of the sigma parameter is the square root of the unbiased estimate of the variance. For the uncensored lognormal distribution, the estimated value of the sigma parameter is the square root of the unbiased estimate of the variance of the log of the data.

References   [1] Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*. Vol. 1, Hoboken, NJ: Wiley-Interscience, 1993.

[2] Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*. Vol. 2, Hoboken, NJ: Wiley-Interscience, 1994.

[3] Bowman, A. W., and A. Azzalini. *Applied Smoothing Techniques for Data Analysis*. New York: Oxford University Press, 1997.

Alternatives   dfittool — Opens a graphical user interface for displaying fit distributions to data, or for fitting distributions to your data and displaying them over plots of the empirical distributions, by importing data from the workspace.

See Also   disttool
randtool
statset — Function that creates a structure that specifies control parameters for the iterative fitting algorithm

`ProbDist` class

`ProbDistUnivKernel` class

`ProbDistUnivParam` class

Appendix B, "Distribution Reference" — For more information on parametric distributions

"Grouped Data" on page 2-33 — For more information on grouping variables

Purpose Flip categorical array along specified dimension

Syntax `B = flipdim(A,dim)`

Description `B = flipdim(A,dim)` returns the categorical array A with dimension `dim` flipped.

See Also `fliplr`, `flipud`, `permute`, `rot90`

categorical.fliplr

Purpose   Flip categorical matrix in left/right direction

Syntax   B = fliplr(A)

Description   B = fliplr(A) returns the 2-D categorical matrix A with rows
                   preserved and columns flipped in the left/right direction.

See Also   flipdim, flipud, permute, rot90

Purpose   Flip categorical matrix in up/down direction

Syntax   B = flipud(A)

Description   B = flipud(A) returns the 2-D categorical matrix A with rows preserved and columns flipped in the up/down direction.

See Also   flipdim, fliplr, permute, rot90

fpdf

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 $[0\ \infty)$.

The probability density function for the *F* distribution is

$$y = f(x|v_1,v_2) = \frac{\Gamma\left[\dfrac{(v_1+v_2)}{2}\right]}{\Gamma\left(\dfrac{v_1}{2}\right)\Gamma\left(\dfrac{v_2}{2}\right)}\left(\dfrac{v_1}{v_2}\right)^{\frac{v_1}{2}}\frac{x^{\frac{v_1-2}{2}}}{\left[1+\left(\dfrac{v_1}{v_2}\right)x\right]^{\frac{v_1+v_2}{2}}}$$

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   pdf, fcdf, finv, fstat, frnd

Purpose   Fractional factorial design

Syntax   dfF = fracfact(generators)
                  [dfF,confounding] = fracfact(generators)

Description   dfF = fracfact(generators) gives factor settings dfF for a two-level
                  Box-Hunter-Hunter fractional factorial design specified by the
                  generators in generators. generators is a string consisting of
                  words formed from the letters a-z, separated by spaces. For example,
                  generators = 'a b c ab ac'. Alternatively, generators is a cell
                  array of strings with one word per cell, as returned by fracfactgen.
                  Single-character words indicate basic factors, for which the design
                  includes all full-factorial treatments. Multiple-character words indicate
                  factors whose levels are limited by the design to products of the levels
                  of component basic factors. dfF is *m*-by-*n*, where *m* is the number
                  of treatments in the design and *n* is the number factors specified by
                  generators.

                  [dfF,confounding] = fracfact(generators) also returns a cell
                  array confounding that shows the confounding pattern among the
                  main effects and the two-factor interactions.

Examples   Suppose you wish to determine the effects of four two-level factors, for
                  which there may be two-way interactions. A full-factorial design would
                  require $2^4 = 16$ runs. The fracfactgen function finds generators for a
                  resolution IV (separating main effects) fractional-factorial design that
                  requires only $2^3 = 8$ runs:

```
generators = fracfactgen('a b c d',3,4)
generators =
    'a'
    'b'
    'c'
    'abc'
```

                  The more economical design and the corresponding confounding pattern
                  are returned by fracfact:

```
[dfF,confounding] = fracfact(generators)
dfF =
    -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
confounding =
    '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 confounding pattern shows, for example, that the two-way interaction between X1 and X2 is confounded by the two-way interaction between X3 and X4.

References [1] Box, G. E. P., W. G. Hunter, and J. S. Hunter. *Statistics for Experimenters*. Hoboken, NJ: Wiley-Interscience, 1978.

See Also fracfactgen, hadamard

Purpose   Fractional factorial design generators

Syntax   generators = fracfactgen(terms)
                generators = fracfactgen(terms,k)
                generators = fracfactgen(terms,k,R)
                generators = fracfactgen(terms,k,R,basic)

Description generators = fracfactgen(terms) uses the Franklin-Bailey
                algorithm to find generators for the smallest two-level
                fractional-factorial design for estimating linear model terms specified
                by terms. terms is a string consisting of words formed from the
                letters a-z, separated by spaces. For example, terms = 'a b c ab
                ac'. Single-character words indicate main effects to be estimated;
                multiple-character words indicate interactions. Alternatively, terms
                is an *m*-by-*n* matrix of 0s and 1s where *m* is the number of model
                terms to be estimated and *n* is the number of factors. For example,
                if terms contains rows [0 1 0 0] and [1 0 0 1], then the factor b
                and the interaction between factors a and d are included in the model.
                generators is a cell array of strings with one generator per cell. Pass
                generators to fracfact to produce the fractional-factorial design and
                corresponding confounding pattern.

                generators = fracfactgen(terms,k) returns generators for a
                two-level fractional-factorial design with 2^k-runs, if possible. If k is [],
                fracfactgen finds the smallest design.

                generators = fracfactgen(terms,k,R) finds a design with resolution
                R, if possible. The default resolution is 3.

                A design of *resolution R* is one in which no *n*-factor interaction is
                confounded with any other effect containing less than $R - n$ factors.
                Thus a resolution III design does not confound main effects with one
                another but may confound them with two-way interactions, while a
                resolution IV design does not confound either main effects or two-way
                interactions but may confound two-way interactions with each other.

                If fracfactgen is unable to find a design at the requested resolution, it
                tries to find a lower-resolution design sufficient to calibrate the model.

fracfactgen

If it is successful, it returns the generators for the lower-resolution design along with a warning. If it fails, it returns an error.

generators = fracfactgen(terms,k,R,basic) also accepts a vector basic specifying the indices of factors that are to be treated as basic. These factors receive full-factorial treatments in the design. The default includes factors that are part of the highest-order interaction in terms.

Examples Suppose you wish to determine the effects of four two-level factors, for which there may be two-way interactions. A full-factorial design would require $2^4 = 16$ runs. The fracfactgen function finds generators for a resolution IV (separating main effects) fractional-factorial design that requires only $2^3 = 8$ runs:

```
generators = fracfactgen('a b c d',3,4)
generators =
    'a'
    'b'
    'c'
    'abc'
```

The more economical design and the corresponding confounding pattern are returned by fracfact:

```
[dfF,confounding] = fracfact(generators)
dfF =
    -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
confounding =
    '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 confounding pattern shows, for example, that the two-way interaction between X1 and X2 is confounded by the two-way interaction between X3 and X4.

References [1] Box, G. E. P., W. G. Hunter, and J. S. Hunter. *Statistics for Experimenters*. Hoboken, NJ: Wiley-Interscience, 1978.

See Also fracfact, hadamard

friedman

Purpose   Friedman's test

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
```

Friedman's ANOVA Table

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			

Test for column effects after row effects are removed

The small *p*-value of `0.001` indicates the popcorn brand affects the yield of popcorn. This is consistent with the results from `anova2`.

References   [1] Hogg, R. V., and J. Ledolter. *Engineering Statistics*. New York: MacMillan, 1987.

[2] Hollander, M., and D. A. Wolfe. *Nonparametric Statistical Methods*. Hoboken, NJ: John Wiley & Sons, Inc., 1999.

See Also   `anova2`, `multcompare`, `kruskalwallis`

frnd

Purpose	*F* random numbers

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.

Examples
```
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   random, fpdf, fcdf, finv, fstat

Purpose   *F* mean and variance

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 v_2 greater than 2 is

$$\frac{v_2}{v_2-2}$$

The variance of the *F* distribution for values of v_2 greater than 4 is

$$\frac{2{v_2}^2(v_1+v_2-2)}{v_1(v_2-2)^2(v_2-4)}$$

The mean of the *F* distribution is undefined if v_2 is less than 3. The variance is undefined for v_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   fpdf, fcdf, finv, frnd

fsurfht

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.

Examples   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)
```

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
```

fullfact

Purpose   Full factorial design

Syntax   dFF = fullfact(levels)

Description dFF = fullfact(levels) gives factor settings dFF for a full factorial design with *n* factors, where the number of levels for each factor is given by the vector levels of length *n*. dFF is *m*-by-*n*, where *m* is the number of treatments in the full-factorial design. Each row of dFF corresponds to a single treatment. Each column contains the settings for a single factor, with integer values from one to the number of levels.

Examples   The following generates an eight-run full-factorial design with two levels in the first factor and four levels in the second factor:

```
dFF = fullfact([2 4])
dFF =
    1    1
    2    1
    1    2
    2    2
    1    3
    2    3
    1    4
    2    4
```

See Also   ff2n

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 2-33.)

`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.

Examples   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%
```

See Also   "Grouped Data" on page 2-33

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).

gamcdf

See Also   cdf, gampdf, gaminv, gamstat, gamfit, gamlike, gamrnd, gamma

Purpose   Gamma parameter estimates

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.

Examples   Fit a gamma distribution to random data generated from a specified
gamma distribution:

```
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
```

References   [1] Hahn, Gerald J., and S. S. Shapiro. *Statistical Models in Engineering.* Hoboken, NJ: John Wiley & Sons, Inc., 1994, p. 88.

See Also   mle, gamlike, gampdf, gamcdf, gaminv, gamstat, gamrnd

Purpose   Gamma inverse 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;

gaminv

```
x = gaminv(gamcdf(1:5,a,b),a,b)
x =
  1.0000  2.0000  3.0000  4.0000  5.0000
```

See Also   icdf, gamcdf, gampdf, gamstat, gamfit, gamlike, gamrnd

Purpose   Gamma negative log-likelihood

Syntax   nlogL = gamlike(params,data)
                [nlogL,AVAR] = gamlike(params,data)

Description nlogL = gamlike(params,data) returns the negative of the gamma
                log-likelihood of the parameters, params, given 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.

Examples   Compute the negative log-likelihood of parameter estimates computed
                by the gamfit function:

```
a = 2; b = 3;
r = gamrnd(a,b,100,1);

[nlogL,AVAR] = gamlike(gamfit(r),r)
nlogL =
  267.5648
AVAR =
  0.0788  -0.1104
```

-0.1104   0.1955

See Also   gamfit, gampdf, gamcdf, gaminv, gamstat, gamrnd

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   pdf, gamcdf, gaminv, gamstat, gamfit, gamlike, gamrnd

gamrnd

Purpose	Gamma random numbers

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.

Examples

```
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   randg, random, gampdf, gamcdf, gaminv, gamstat, gamfit, gamlike

Purpose   Gamma mean and variance

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   gampdf, gamcdf, gaminv, gamfit, gamlike, gamrnd

qrandstream.ge

Purpose   Greater than or equal relation for handles

Syntax   `h1 >= h2`

Description   `h1 >= h2` performs element-wise comparisons between handle arrays `h1` and `h2`. `h1` and `h2` must be of the same dimensions unless one is a scalar. The result is a logical array of the same dimensions, where each element is an element-wise `>=` result.

If one of `h1` or `h2` is scalar, scalar expansion is performed and the result will match the dimensions of the array that is not scalar.

`tf = ge(h1, h2)` stores the result in a logical array of the same dimensions.

See Also   `qrandstream`, `eq`, `gt`, `le`, `lt`, `ne`

geocdf

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, geopdf, geoinv, geostat, geornd, mle

geoinv

Purpose	Geometric inverse 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   icdf, geocdf, geopdf, geostat, geornd

Purpose   Geometric mean

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

geopdf

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 `pdf`, `geocdf`, `geoinv`, `geostat`, `geornd`

Purpose	Geometric random numbers

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].

Examples
```
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

random, geopdf, geocdf, geoinv, geostat

geostat

Purpose	Geometric mean and variance
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 $q = 1-p$. 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 geopdf, geocdf, geoinv, geornd

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.

Examples   Create a dataset array from Fisher's iris data and access the information:

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

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

Purpose   Access categorical array labels

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   getlevels, setlabels

categorical.getlevels

Purpose   Get categorical array levels

Syntax   S = getlevels(A)

Description S = getlevels(A) returns the levels for the categorical array A. S is
a vector with the same type as A.

See Also   getlabels

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 $K \geq 1$, and the
                   variance is not finite when $K \geq 1/2$. 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. *Modelling Extremal
                   Events for Insurance and Finance.* New York: Springer, 1997.

                   [2] Kotz, S., and S. Nadarajah. *Extreme Value Distributions: Theory
                   and Applications.* London: Imperial College Press, 2000.

See Also   cdf, gevpdf, gevinv, gevstat, gevfit, gevlike, gevrnd

gevfit

Purpose   Generalized extreme value parameter estimates

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 $K \geq 1$, and the
                variance is not finite when $K \geq 1/2$. The GEV distribution is defined
                for K*(X-mu)/sigma > -1.

References [1] Embrechts, P., C. Klüppelberg, and T. Mikosch. *Modelling Extremal
                Events for Insurance and Finance.* New York: Springer, 1997.

[2] Kotz, S., and S. Nadarajah. *Extreme Value Distributions: Theory and Applications.* London: Imperial College Press, 2000.

See Also mle, gevlike, gevpdf, gevcdf, gevinv, gevstat, gevrnd

gevinv

Purpose   Generalized extreme value inverse 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 $K \geq 1$, and the variance is not finite when $K \geq 1/2$. 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. *Modelling Extremal Events for Insurance and Finance.* New York: Springer, 1997.

[2] Kotz, S., and S. Nadarajah. *Extreme Value Distributions: Theory and Applications.* London: Imperial College Press, 2000.

See Also   icdf, gevcdf, gevpdf, gevstat, gevfit, gevlike, gevrnd

Purpose	Generalized extreme value negative log-likelihood
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 $K \geq 1$, and the variance is not finite when $K \geq 1/2$. 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. *Modelling Extremal Events for Insurance and Finance*. New York: Springer, 1997.

[2] Kotz, S., and S. Nadarajah.*Extreme Value Distributions: Theory and Applications*. London: Imperial College Press, 2000.

See Also   gevfit, gevpdf, gevcdf, gevinv, gevstat, gevrnd

gevpdf

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 $K \geq 1$, and the variance is not finite when $K \geq 1/2$. 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. *Modelling Extremal Events for Insurance and Finance.* New York: Springer, 1997.

[2] Kotz, S., and S. Nadarajah. *Extreme Value Distributions: Theory and Applications.* London: Imperial College Press, 2000.

See Also   pdf, gevcdf, gevinv, gevstat, gevfit, gevlike, gevrnd

Purpose	Generalized extreme value random numbers

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 $K \geq 1$, and the variance is not finite when $K \geq 1/2$. 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. *Modelling Extremal Events for Insurance and Finance*. New York: Springer, 1997.

[2] Kotz, S., and S. Nadarajah. *Extreme Value Distributions: Theory and Applications*. London: Imperial College Press, 2000.

See Also random, gevpdf, gevcdf, gevinv, gevstat, gevfit, gevlike

gevstat

Purpose   Generalized extreme value mean and variance

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 $K \geq 1$, and the variance is not finite when $K \geq 1/2$. 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. *Modelling Extremal Events for Insurance and Finance.* New York: Springer, 1997.

[2] Kotz, S., and S. Nadarajah. *Extreme Value Distributions: Theory and Applications.* London: Imperial College Press, 2000.

See Also   gevpdf, gevcdf, gevinv, gevfit, gevlike, gevrnd

Purpose   Interactively add line to plot

Syntax   gline(h)
                  gline
                  hline = gline(...)

Description   gline(h) allows you to draw a line segment in the figure with handle h
                  by clicking the pointer at the two endpoints. A rubber-band line tracks
                  the pointer movement.

                  gline with no input arguments defaults to h = gcf and draws in the
                  current figure.

                  hline = gline(...) returns the handle hline to the line.

Examples   Use gline to connect two points in a plot:

```
x = 1:10;

y = x + randn(1,10);
scatter(x,y,25,'b','*')

lsline

mu = mean(y);
hold on
plot([1 10],[mu mu],'ro')

hline = gline; % Connect circles
set(hline,'Color','r')
```

gline

See Also   refline, refcurve, lsline

Purpose	Generalized linear model regression
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 1s to X, corresponding to a constant term in the model. Do not enter a column of 1s directly into X. You can change the default behavior of glmfit using the 'constant' parameter, below.

glmfit treats NaNs 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	Description
'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	Description
'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.* New York: Chapman & Hall, 1990.

[2] McCullagh, P., and J. A. Nelder. *Generalized Linear Models.* New York: Chapman & Hall, 1990.

[3] Collett, D. *Modeling Binary Data.* New York: Chapman & Hall, 2002.

See Also   glmval, regress, regstats

glmval

Purpose	Generalized linear model values
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 1s to X, corresponding to a constant term in the model. Do not enter a column of 1s 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

Examples

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)
```

glmval

References   [1] Dobson, A. J. *An Introduction to Generalized Linear Models.* New York: Chapman & Hall, 1990.

[2] McCullagh, P., and J. A. Nelder. *Generalized Linear Models.* New York: Chapman & Hall, 1990.

[3] Collett, D. *Modeling Binary Data.* New York: Chapman & Hall, 2002.

See Also   glmfit

glyphplot

Purpose   Glyph plot

Syntax
```
glyphplot(X)
glyphplot(X,'glyph','face')
glyphplot(X,'glyph','face','features',f)
glyphplot(X,...,'grid',[rows,cols])
glyphplot(X,...,'grid',[rows,cols],'page',p)
glyphplot(X,...,'centers',C)
glyphplot,...,'centers',C,'radius',r)
glyphplot(X,...,'obslabels',labels)
glyphplot(X,...,'standardize',method)
glyphplot(X,...,prop1,val1,...)
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 *i*th spoke is proportional in length to the *i*th 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 NaNs 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 *i*th 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
17-467Face Features.

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 *i*th column of X. f must contain integers from 0 to
17, where 0 indicate that the corresponding column of X should not be
plotted. See "Face Features" on page 17-467Face Features for more
information.

glyphplot(X,...,'grid',[rows,cols]) organizes the glyphs into a rows-by-cols grid.

glyphplot(X,...,'grid',[rows,cols],'page',p) organizes the glyph into one or more pages of a rows-by-cols grid, and displays the page p. If p is a vector, glyphplot displays multiple pages in succession. If p is 'all', glyphplot displays all pages. If p 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. Use '' for blank 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,...,*prop1*,*val1*,...) 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

glyphplot

Examples

```
load carsmall
X = [Acceleration Displacement Horsepower MPG Weight];

glyphplot(X,'standardize','column',...
          'obslabels',Model,...
          'grid',[2 2],...
          'page','scroll');
```


chevrolet chevelle malibu

buick skylark 320

plymouth satellite

amc rebel sst

```
glyphplot(X,'glyph','face',...
          'obslabels',Model,...
          'grid',[2 3],...
          'page',9);
```

pontiac ventura sj   amc pacer d/l   volkswagen rabbit

datsun b-210   toyota corolla   ford pinto

See Also   andrewsplot, parallelcoords

gmdistribution class

Purpose   Gaussian mixture models

Description   An object of the `gmdistribution` class defines a Gaussian mixture distribution, which is a multivariate distribution that consists of a mixture of one or more multivariate Gaussian distribution components. The number of components for a given `gmdistribution` object is fixed. Each multivariate Gaussian component is defined by its mean and covariance, and the mixture is defined by a vector of mixing proportions.

Construction   To create a Gaussian mixture distribution by specifying the distribution parameters, use the `gmdistribution` constructor. To fit a Gaussian mixture distribution model to data, use `gmdistribution.fit`.

fit	Gaussian mixture parameter estimates
gmdistribution	Construct Gaussian mixture distribution

Methods

cdf	Cumulative distribution function for Gaussian mixture distribution
cluster	Construct clusters from Gaussian mixture distribution
disp	Display Gaussian mixture distribution object
display	Display Gaussian mixture distribution object
fit	Gaussian mixture parameter estimates
mahal	Mahalanobis distance to component means

pdf	Probability density function for Gaussian mixture distribution
posterior	Posterior probabilities of components
random	Random numbers from Gaussian mixture distribution
subsasgn	Subscripted reference for Gaussian mixture distribution object
subsref	Subscripted reference for Gaussian mixture distribution object

Properties   All objects of the class have the properties listed in the following table.

CovType	Type of covariance matrices
DistName	Type of distribution
Mu	Input matrix of means `MU`
NComponents	Number k of mixture components
NDimensions	Dimension d of multivariate Gaussian distributions
PComponents	Input vector of mixing proportions
SharedCov	`true` if all covariance matrices are restricted to be the same
Sigma	Input array of covariances

Objects constructed with `fit` have the additional properties listed in the following table.

gmdistribution class

AIC	Akaike Information Criterion
BIC	Bayes Information Criterion
Converged	Determine if algorithm converged
Iters	Number of iterations
NlogL	Negative of log-likelihood
RegV	Value of `Regularize` parameter

Copy Semantics   Value. To learn how this affects your use of the class, see Comparing Handle and Value Classes in the MATLAB Object-Oriented Programming documentation.

References   McLachlan, G., and D. Peel, Finite Mixture Models, John Wiley & Sons, New York, 2000.

See Also   "Normal Distribution" on page B-75

Purpose   Construct Gaussian mixture distribution

Syntax   `obj = gmdistribution(mu,sigma,p)`

Description   `obj = gmdistribution(mu,sigma,p)` constructs an object `obj` of the `gmdistribution` class defining a Gaussian mixture distribution.

mu is a *k*-by-*d* matrix specifying the *d*-dimensional mean of each of the *k* components.

sigma specifies the covariance of each component. The size of sigma is:

- *d*-by-*d*-by-*k* if there are no restrictions on the form of the covariance. In this case, `sigma(:,:,I)` is the covariance of component `I`.

- 1-by-*d*-by-*k* if the covariance matrices are restricted to be diagonal, but not restricted to be same across components. In this case, `sigma(:,:,I)` contains the diagonal elements of the covariance of component I.

- *d*-by-*d* matrix if the covariance matrices are restricted to be the same across components, but not restricted to be diagonal. In this case, sigma is the pooled estimate of covariance.

- 1-by-*d* if the covariance matrices are restricted to be diagonal and the same across components. In this case, sigma contains the diagonal elements of the pooled estimate of covariance.

p is an optional 1-by-*k* vector specifying the mixing proportions of each component. If p does not sum to 1, gmdistribution normalizes it. The default is equal proportions.

Examples   Create a `gmdistribution` object defining a two-component mixture of bivariate Gaussian distributions:

```
mu = [1 2;-3 -5];
sigma = cat(3,[2 0;0 .5],[1 0;0 1]);
p = ones(1,2)/2;
obj = gmdistribution(mu,sigma,p);
```

ezsurf(@(x,y)pdf(obj,[x y]),[-10 10],[-10 10])

pdf(obj,[x,y])

References   [1] McLachlan, G., and D. Peel. *Finite Mixture Models*. Hoboken, NJ: John Wiley & Sons, Inc., 2000.

See Also   fit, pdf, cdf, random, cluster, posterior, mahal

Purpose   Add case names to plot

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.

Examples   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 gtext, gscatter, gplotmatrix

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 ≥ 1, and the variance is not finite when K ≥ 1/2. When K ≥ 0, the GP has positive density for

X > theta, or, when

$$K < 0, \ 0 \leq \frac{X - \theta}{\sigma} \leq -\frac{1}{K}.$$

References   [1] Embrechts, P., C. Klüppelberg, and T. Mikosch. *Modelling Extremal Events for Insurance and Finance.* New York: Springer, 1997.

[2] Kotz, S., and S. Nadarajah. *Extreme Value Distributions: Theory and Applications.* London: Imperial College Press, 2000.

See Also   cdf, gppdf, gpinv, gpstat, gpfit, gplike, gprnd

gpfit

Purpose	Generalized Pareto parameter estimates

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/k, 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

$$0 \leq \frac{X - \theta}{\sigma} \leq -\frac{1}{K}$$

References   [1] Embrechts, P., C. Klüppelberg, and T. Mikosch. *Modelling Extremal Events for Insurance and Finance.* New York: Springer, 1997.

[2] Kotz, S., and S. Nadarajah. *Extreme Value Distributions: Theory and Applications.* London: Imperial College Press, 2000.

See Also   mle, gplike, gppdf, gpcdf, gpinv, gpstat, gprnd

gpinv

Purpose Generalized Pareto inverse 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 ≥ 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. *Modelling Extremal Events for Insurance and Finance.* New York: Springer, 1997.

[2] Kotz, S., and S. Nadarajah. *Extreme Value Distributions: Theory and Applications.* London: Imperial College Press, 2000.

See Also `icdf, gpcdf, gppdf, gpstat, gpfit, gplike, gprnd`

Purpose   Generalized Pareto negative log-likelihood

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. *Modelling Extremal
                  Events for Insurance and Finance.* New York: Springer, 1997.

                  [2] Kotz, S., and S. Nadarajah. *Extreme Value Distributions: Theory
                  and Applications.* London: Imperial College Press, 2000.

See Also   gpfit, gppdf, gpcdf, gpinv, gpstat, gprnd

gppdf

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 $K \geq 1$, 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. *Modelling Extremal Events for Insurance and Finance.* New York: Springer, 1997.

[2] Kotz, S., and S. Nadarajah. *Extreme Value Distributions: Theory and Applications.* London: Imperial College Press, 2000.

See Also   pdf, gpcdf, gpinv, gpstat, gpfit, gplike, gprnd

Purpose   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 2-33.)

                   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

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.

Examples   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(:,1:2),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(:,1:2),ratings(:,[4 7]),group,...
            'br','.o',[],'on','',categories(1:2,:),...
             categories([4 7],:))
```

See Also   "Grouped Data" on page 2-33

grpstats, gscatter, plotmatrix

gprnd

Purpose	Generalized Pareto random numbers

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 $K \geq 1$, 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

$$0 \leq \frac{X - \theta}{\sigma} \leq -\frac{1}{K}$$

References   [1] Embrechts, P., C. Klüppelberg, and T. Mikosch. *Modelling Extremal Events for Insurance and Finance.* New York: Springer, 1997.

[2] Kotz, S., and S. Nadarajah. *Extreme Value Distributions: Theory and Applications.* London: Imperial College Press, 2000.

See Also   random, gppdf, gpcdf, gpinv, gpstat, gpfit, gplike

Purpose	Generalized Pareto mean and variance
Syntax	[M,V] = gpstat(K,sigma,theta)

Description   [M,V] = gpstat(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

$$K < 0, \ 0 \leq \frac{X - \theta}{\sigma} \leq -\frac{1}{K} \, .$$

References   [1] Embrechts, P., C. Klüppelberg, and T. Mikosch. *Modelling Extremal Events for Insurance and Finance.* New York: Springer, 1997.

[2] Kotz, S., and S. Nadarajah. *Extreme Value Distributions: Theory and Applications.* London: Imperial College Press, 2000.

See Also   gppdf, gpcdf, gpinv, gpfit, gplike, gprnd

TreeBagger.growTrees

Purpose	Train additional trees and add to ensemble
Syntax	`B = growTrees(B,ntrees)` `B = growTrees(B,ntrees,'nprint',N)`
Description	`B = growTrees(B,ntrees)` grows `ntrees` new trees and appends them to those already stored in the ensemble `B`.
	`B = growTrees(B,ntrees,'nprint',N)` specifies that a diagnostic message showing training progress should display after every `N` training cycles (grown trees). Default is no diagnostic messages.
See Also	`classregtree`

Purpose   Create index vector from grouping variable

Syntax   ```
[G,GN]=grp2idx(S)
[indices,names] = grp2idx(group)
[G,GN,GL] = grp2idx(S)
```

Description   `[G,GN]=grp2idx(S)` creates an index vector `G` from the grouping variable `S`. `S` can be a categorical, numeric, or logical vector; a cell vector of strings; or a character matrix with each row representing a group label. The result `G` is a vector taking integer values from 1 up to the number `K` of distinct groups. `GN` is a cell array of strings representing group labels. If `S` is a character matrix, `GN(G,:)` reproduces `S`, otherwise `GN(G)` reproduces `S` (aside from any differences in type).

The order of `indices` depends on the grouping variable:

- For numeric and logical grouping variables, the order is the sorted order of `group`.

- For categorical grouping variables, the order is the order of `getlabels(group)`.

- For string grouping variables, the order is the order of first appearance in `group`.

`[indices,names] = grp2idx(group)` also returns a cell array of group `names`, so that `names(indices)` reproduces `group`, apart from differences in type.

`[G,GN,GL] = grp2idx(S)` returns a column vector `GL` representing the group levels. The set of groups and their order in `GL` and `GN` are the same, except that `GL` has the same type as `S`. If `S` is a character matrix, `GL(G,:)` reproduces `S`, otherwise `GL(G)` reproduces `S`.

`grp2idx` treats NaNs (numeric or logical), empty strings (char or cell array of strings), or values (categorical) in `S` as missing values and returns NaNs in the corresponding rows of `G`. `GN` and `GL` don't include entries for missing values.

grp2idx

Examples Load the data in `hospital.mat` and create a categorical grouping variable:

```
load hospital
edges = 0:10:100;
labels = strcat(num2str((0:10:90)','%d'),{'s'});
AgeGroup = ordinal(hospital.Age,labels,[],edges);

ages = hospital.Age(1:5)
ages =
    38
    43
    38
    40
    49

group = AgeGroup(1:5)
group =
    30s
    40s
    30s
    40s
    40s

indices = grp2idx(group)
indices =
    4
    5
    4
    5
    5
```

See Also "Grouped Data" on page 2-33

crosstab, getlabels, grpstats, gscatter

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 2-33.) 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

- `'min'` — minimum

- `'max'` — maximum

- `'range'` — maximum - minimum

- `'meanci'` — 95% confidence interval for the mean

- `'predci'` — 95% prediction interval for a new observation

Each function included in *whichstats* must accept a column vector of data and compute a descriptive statistic for it. For example, `@median` and `@skewness` are suitable functions to apply to a numeric input. A function must return the same size output each time `grpstats` calls it, even if the input for some groups is empty. The function typically returns a scalar value, but may return an *nvals*-by-1 column vector if the descriptive statistic is not a scalar (a confidence interval, for example). The size of each output A, B, ... is *ngroups*-by-*ncols*-by-*nvals*, where *ngroups* is the number of groups, *ncols* is the number of columns in the data X, and *nvals* is the number of values returned by the function for data from a single group in one column of X. If X is a vector of data, then the size of each output A, B, ... is *ngroups*-by-*nvals*.

A function included in *whichstats* may also be written to accept a matrix of data and compute a descriptive statistic for each column. The function should return either a row vector, or an *nvals*-by-*ncols* matrix if the descriptive statistic is not a scalar.

`[...] = 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.

Examples

```
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   "Grouped Data" on page 2-33

gscatter, grp2idx, grpstats

dataset.grpstats

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 2-33.)

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 NaNs 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*.

dataset.grpstats

Examples   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

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 2-33.) 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.

gscatter

Examples   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')
```
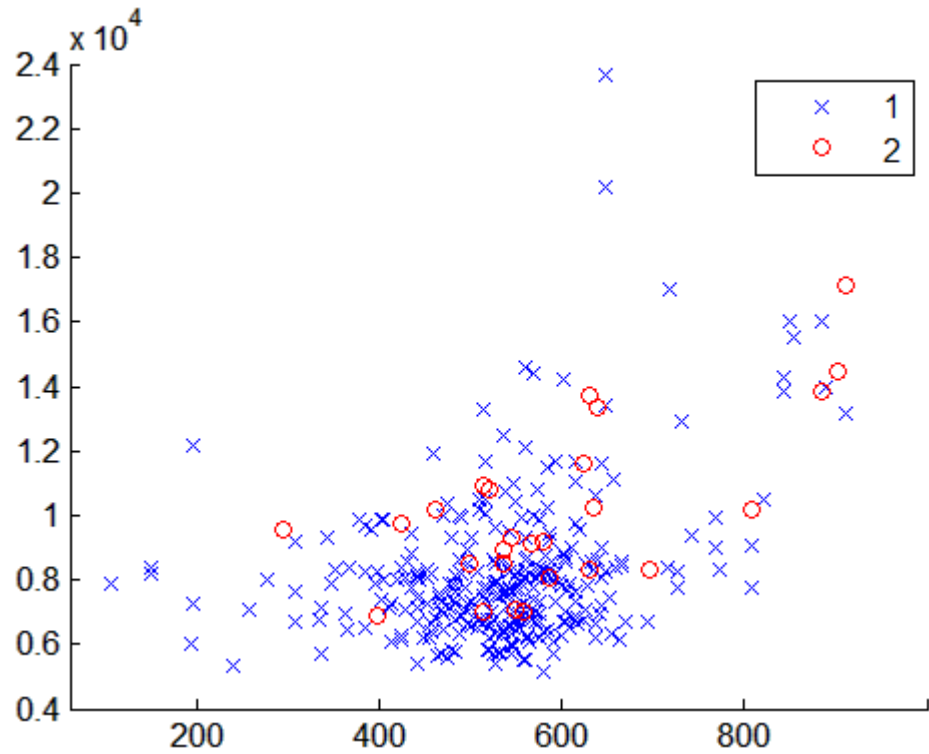

See Also   "Grouped Data" on page 2-33

gplotmatrix, grpstats, scatter

Purpose   Greater than relation for handles

Syntax   `h1 > h2`

Description `h1 > h2` performs element-wise comparisons between handle arrays `h1` and `h2`. `h1` and `h2` must be of the same dimensions unless one is a scalar. The result is a logical array of the same dimensions, where each element is an element-wise > result.

If one of `h1` or `h2` is scalar, scalar expansion is performed and the result will match the dimensions of the array that is not scalar.

`tf = gt(h1, h2)` stores the result in a logical array of the same dimensions.

See Also   `qrandstream, eq, ge, le, lt, ne`

haltonset class

Superclasses   qrandset

Purpose   Halton quasi-random point sets

Description   haltonset is a quasi-random point set class that produces points from
the Halton sequence.

Construction

haltonset	Construct Halton quasi-random point set

Methods   **Inherited Methods**

Methods in the following table are inherited from qrandset.

disp	Display qrandset object
end	Last index in indexing expression for point set
length	Length of point set
ndims	Number of dimensions in matrix
net	Generate quasi-random point set
scramble	Scramble quasi-random point set
size	Number of dimensions in matrix
subsref	Subscripted reference for qrandset

Properties   **Inherited Properties**

Properties in the following table are inherited from qrandset.

Dimensions	Number of dimensions
Leap	Interval between points

ScrambleMethod	Scramble settings
Skip	Number of initial points omitted
Type	Subclass

Copy Semantics   Handle. To learn how this affects your use of the class, see Comparing Handle and Value Classes in the MATLAB Object-Oriented Programming documentation.

References   [1] Kocis, L., and W. J. Whiten, "Computational Investigations of Low-Discrepancy Sequences," ACM Transactions on Mathematical Software, Vol. 23, No. 2, pp. 266-294, 1997.

See Also   "Quasi-Random Point Sets" on page 5-144

sobolset

haltonset

Purpose	Construct Halton quasi-random point set
Syntax	`p = haltonset(d)` `p = haltonset(d,`*`prop1`*`,`*`val1`*`,`*`prop2`*`,`*`val2`*`,...)`
Description	`p = haltonset(d)` constructs a d-dimensional point set `p` of the `haltonset` class, with default property settings. `p = haltonset(d,`*`prop1`*`,`*`val1`*`,`*`prop2`*`,`*`val2`*`,...)` specifies property name/value pairs used to construct `p`. The object `p` returned by `haltonset` encapsulates properties of a specified quasi-random sequence. The point set is finite, with a length determined by the `Skip` and `Leap` properties and by limits on the size of point set indices (maximum value of 2^{53}). Values of the point set are not generated and stored in memory until you access `p` using `net` or parenthesis indexing.
Examples	Generate a 3-D Halton point set, skip the first 1000 values, and then retain every 101st point:

```
p = haltonset(3,'Skip',1e3,'Leap',1e2)
p =
    Halton point set in 3 dimensions (8.918019e+013 points)
    Properties:
              Skip : 1000
              Leap : 100
    ScrambleMethod : none
```

Use `scramble` to apply reverse-radix scrambling:

```
p = scramble(p,'RR2')
p =
    Halton point set in 3 dimensions (8.918019e+013 points)
    Properties:
              Skip : 1000
              Leap : 100
    ScrambleMethod : RR2
```

Use net to generate the first four points:

```
X0 = net(p,4)
X0 =
    0.0928    0.6950    0.0029
    0.6958    0.2958    0.8269
    0.3013    0.6497    0.4141
    0.9087    0.7883    0.2166
```

Use parenthesis indexing to generate every third point, up to the 11th point:

```
X = p(1:3:11,:)
X =
    0.0928    0.6950    0.0029
    0.9087    0.7883    0.2166
    0.3843    0.9840    0.9878
    0.6831    0.7357    0.7923
```

References   [1] Kocis, L., and W. J. Whiten. "Computational Investigations of Low-Discrepancy Sequences." *ACM Transactions on Mathematical Software.* Vol. 23, No. 2, 1997, pp. 266–294.

See Also   net, scramble, sobolset

harmmean

Purpose   Harmonic mean

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

$$m = \frac{n}{\displaystyle\sum_{i=1}^{n} \frac{1}{x_i}}$$

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   Bivariate histogram

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 2-D 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 NaNs. 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 2-D 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.

hist3

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.

Examples

Example 1

Make a 3-D figure using a histogram with a density plot underneath:

```
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);
```

Seamount: Data Point Density Histogram and Intensity Map
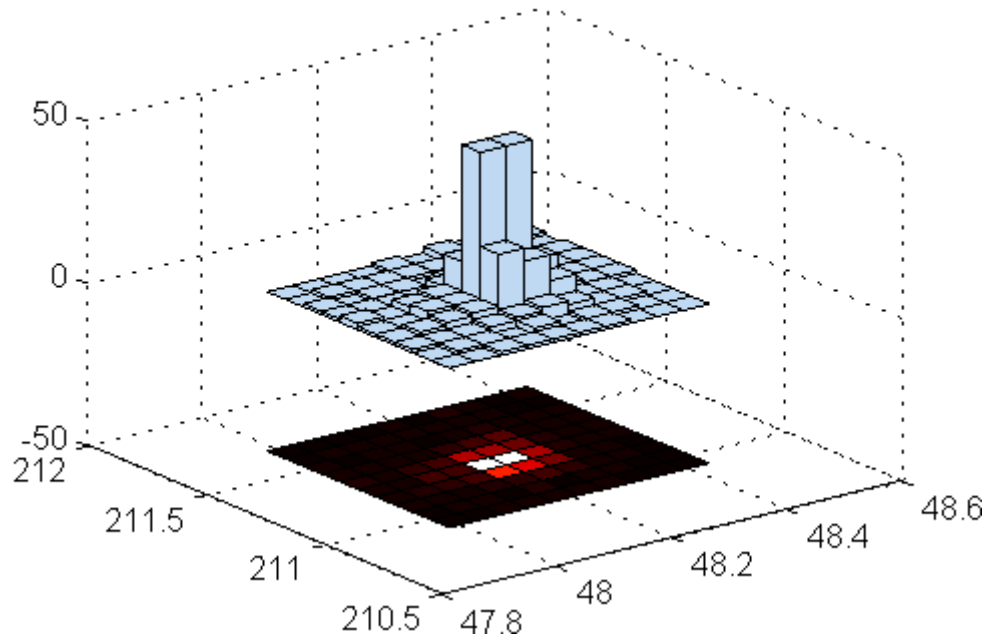
Example 2

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');
```
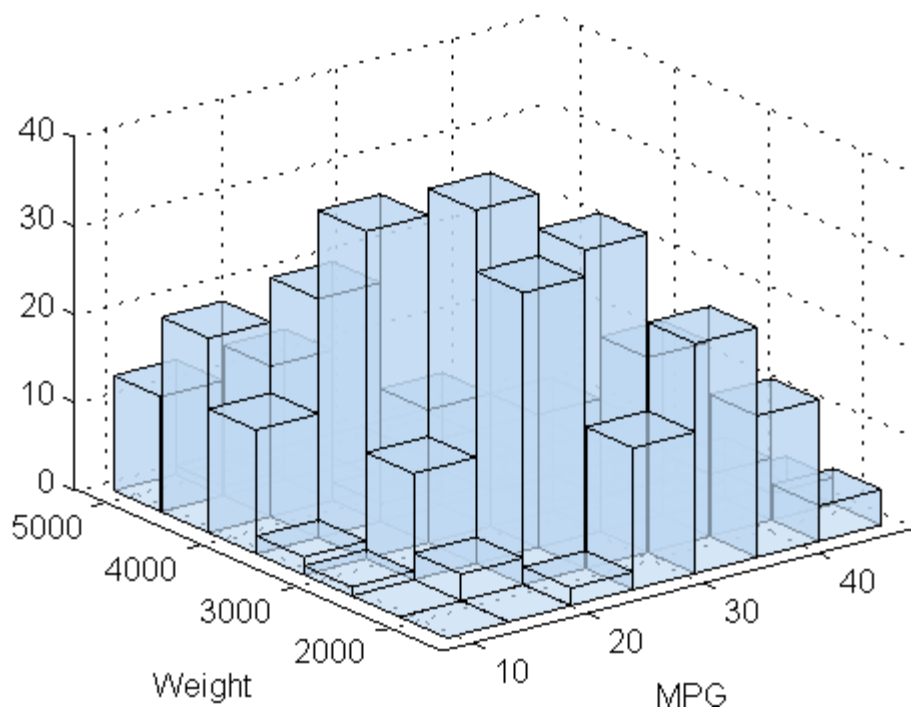


```
% Specify bin centers, different in each direction.
% Get back counts, but don't make the plot.
cnt = hist3(X, {0:10:50 2000:500:5000});
```

See Also   accumarray, bar, bar3, hist, histc

Purpose   Histogram with normal fit

Syntax   ```
histfit(data)
histfit(data,nbins)
histfit(data,nbins,dist)
h = histfit(...)
```

**Description**     `histfit(data)` plots a histogram of the values in the vector `data` using
a number of bins equal to the square root of the number of elements in
`data`, then superimposes a fitted normal distribution.

`histfit(data,nbins)` uses `nbins` bins for the histogram.

`histfit(data,nbins,dist)` plots a histogram with a density from the
distribution specified by `dist`, one of the following strings:

- `'beta'`
- `'birnbaumsaunders'`
- `'exponential'`
- `'extreme value'` or `ev'`
- `'gamma'`
- `'generalized extreme value'` or `'gev'`
- `'generalized pareto'` or `'gp'`
- `'inversegaussian'`
- `'logistic'`
- `'loglogistic'`
- `'lognormal'`
- `'nakagami'`
- `'negative binomial'` or `'nbin'`
- `'normal'` (default)
- `'poisson'`

# histfit

- `'rayleigh'`
- `'rician'`
- `'tlocationscale'`
- `'weibull'` or `'wbl'`

`h = histfit(...)` returns a vector of handles h, where h(1) is the handle to the histogram and h(2) is the handle to the normal curve.

**Examples**

```
r = normrnd(10,1,100,1);
histfit(r)
h = get(gca,'Children');
set(h(2),'FaceColor',[.8 .8 1])
```

histfit

See Also   hist, normfit

hmmdecode

Purpose	Hidden Markov model posterior state probabilities

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 jth 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.

`[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`.

`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.

References   [1] Durbin, R., S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis.* Cambridge, UK: Cambridge University Press, 1998.

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'});
```

See Also   hmmgenerate, hmmestimate, hmmviterbi, hmmtrain

hmmestimate

Purpose   Hidden Markov model parameter estimates from emissions and states

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.

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 $i \to j$ 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 $i \to j$ in a sequence of the same length as states, and the actual number of transitions $i \to j$ 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.

References [1] Durbin, R., S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis*. Cambridge, UK: Cambridge University Press, 1998.

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

hmmgenerate

Purpose   Hidden Markov model states and emissions

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 emission a_{k_1} with
probability $E_{i_1 k_{1_1}}$. hmmgenerate returns i_1 as the first entry of states,
and a_{k_1} as the first entry of seq.

---

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.

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

hmmtrain

Purpose   Hidden Markov model parameter estimates from emissions

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.

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.

References [1] Durbin, R., S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis*. Cambridge, UK: Cambridge University Press, 1998.

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

Purpose   Hidden Markov model most probable state path

Syntax   STATES = hmmviterbi(seq,TRANS,EMIS)
hmmviterbi(...,'Symbols',SYMBOLS)
hmmviterbi(...,'Statenames',STATENAMES)

Description   STATES = hmmviterbi(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.

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.

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);
```

hmmviterbi

```
[seq,states] = ...
   hmmgenerate(100,trans,emis,...
               'Statenames',{'fair';'loaded'});
estimatesStates = ...
   hmmviterbi(seq,trans,eemis,...
               'Statenames',{'fair';'loaded'});
```

References

[1] Durbin, R., S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis.* Cambridge, UK: Cambridge University Press, 1998.

See Also

hmmgenerate, hmmdecode, hmmestimate, hmmtrain

Purpose   Horizontal concatenation for categorical arrays

Syntax
```
C = horzcat(dim,A,B,...)
C = horzcat(A,B)
```

Description   `C = horzcat(dim,A,B,...)` horizontally concatenates the categorical arrays `A,B,...` . For matrices, all inputs must have the same number of rows. For n-D arrays, all inputs must have the same sizes except in the second dimension. The set of categorical levels for `C` is the sorted union of the sets of levels of the inputs, as determined by their labels.

`C = horzcat(A,B)` is called for the syntax `[A B]`.

See Also   `cat`, `vertcat`

dataset.horzcat

Purpose   Horizontal concatenation for dataset arrays

Syntax   `ds = horzcat(ds1, ds2, ...)`

Description   `ds = horzcat(ds1, ds2, ...)` horizontally concatenates the dataset arrays `ds1, ds2, ...`. You may concatenate dataset arrays that have duplicate variable names, however, the variables must contain identical data, and `horzcat` includes only one copy of the variable in the output dataset.

Observation names for all dataset arrays that have them must be identical except for order. `horzcat` concatenates by matching observation names when present, or by position for datasets that do not have observation names.

See Also   `cat`, `vertcat`

Purpose   Hougen-Watson model

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:

$$\hat{y} = \frac{\beta_1 x_2 - x_3 / \beta_5}{1 + \beta_2 x_1 + \beta_3 x_2 + \beta_4 x_3}$$

References   [1] Bates, D. M., and D. G. Watts. *Nonlinear Regression Analysis and Its Applications.* Hoboken, NJ: John Wiley & Sons, Inc., 1988.

See Also   rsmdemo

hygecdf

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　　　`cdf`, `hygepdf`, `hygeinv`, `hygestat`, `hygernd`

Purpose   Hypergeometric inverse 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   icdf, hygecdf, hygepdf, hygestat, hygernd

hygepdf

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 $N \leq M$. 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   pdf, hygecdf, hygeinv, hygestat, hygernd

hygernd

Purpose	Hypergeometric random numbers

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.

Examples
```
numbers = hygernd(1000,40,50)
numbers =
    1
```

See Also   random, hygepdf, hygecdf, hygeinv, hygestat

hygestat

Purpose	Hypergeometric mean and variance
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 `NK(M-K)(M-N)/[M^2(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`, `hygecdf`, `hygeinv`, `hygernd`

Purpose   Inverse cumulative distribution functions

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   Compute the icdf of the normal distribution with mean 0 and standard deviation 1 at inputs 0.1, 0.3, ..., 0.9:

```
x1 = icdf('Normal',0.1:0.2:0.9,0,1)
x1 =
  -1.2816  -0.5244  0  0.5244  1.2816
```

The order of the parameters is the same as for norminv.

Compute the icdfs of Poisson distributions with rate parameters 0, 1, ..., 4 at inputs 0.1, 0.3, ..., 0.9, respectively:

```
x2 = icdf('Poisson',0.1:0.2:0.9,0:4)
x2 =
   NaN    0    2    4    7
```

The order of the parameters is the same as for poissinv.

See Also   cdf, mle, pdf, random

ProbDistUnivKernel.icdf

Purpose	Return inverse cumulative distribution function (ICDF) for ProbDistUnivKernel object
Syntax	`Y = icdf(PD, P)`
Description	`Y = icdf(PD, P)` returns `Y`, an array containing the inverse cumulative distribution function (ICDF) for the ProbDistUnivKernel object `PD`, evaluated at values in `P`.

Inputs

PD	An object of the class `ProbDistUnivKernel`.
P	A numeric array of values from `0` to `1` where you want to evaluate the ICDF.

Outputs

Y	An array containing the inverse cumulative distribution function (ICDF) for the ProbDistUnivKernel object *PD*.

See Also   `icdf`

Purpose	Return inverse cumulative distribution function (ICDF) for ProbDistUnivParam object
Syntax	*Y* = icdf(*PD*, *P*)
Description	*Y* = icdf(*PD*, *P*) returns *Y*, an array containing the inverse cumulative distribution function (ICDF) for the ProbDistUnivParam object *PD*, evaluated at values in *P*.

Inputs

PD	An object of the class ProbDistUnivParam.
P	A numeric array of values from 0 to 1 where you want to evaluate the ICDF.

Outputs

Y	An array containing the inverse cumulative distribution function (ICDF) for the ProbDistUnivParam object *PD*.

See Also   icdf

piecewisedistribution.icdf

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.
Examples	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

Purpose   Inconsistency coefficient

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.

inconsistent

Examples

```
rand('state',12);
X = rand(10,2);
Y = pdist(X);
Z = linkage(Y,'single');
dendrogram(Z)
```



```
W = inconsistent(Z,3)
W =
    0.1313        0   1.0000        0
    0.1386        0   1.0000        0
    0.1463   0.0109   2.0000   0.7071
    0.2391        0   1.0000        0
    0.1951   0.0568   4.0000   0.9425
    0.2308   0.0543   4.0000   0.9320
    0.2395   0.0748   4.0000   0.7636
    0.2654   0.0945   4.0000   0.9203
    0.3769   0.0950   3.0000   1.1040
```

References   [1] Jain, A., and R. Dubes. *Algorithms for Clustering Data.* Upper Saddle River, NJ: Prentice-Hall, 1988.

[2] Zahn, C. T. "Graph-theoretical methods for detecting and describing Gestalt clusters." *IEEE Transactions on Computers.* Vol. C-20, Issue 1, 1971, pp. 68–86.

See Also   cluster, cophenet, clusterdata, dendrogram, linkage, pdist, squareform

ProbDist.InputData property

Purpose   Read-only structure containing information about input data to ProbDist object

Description   InputData is a read-only property of the ProbDist class. InputData is a structure containing information about input data to a ProbDist object. It includes the following fields:

- data

- cens

- freq

Values   Possible values for the three fields in the structure are any data supplied to the fitdist function:

- data — Data passed to the fitdist function when creating the ProbDist object. This field is empty if the ProbDist object was created without fitting to data, that is by using the ProbDistUnivParam.ProbDistUnivParam constructor.

- cens — The vector supplied with the 'censoring' parameter when creating the ProbDist object using the fitdist function. This field is empty if the ProbDist object was created without fitting to data, that is by using the ProbDistUnivParam.ProbDistUnivParam constructor.

- freq — The vector supplied with the 'frequency' parameter when creating the ProbDist object using the fitdist function. This field is empty if the ProbDist object was created without fitting to data, that is by using the ProbDistUnivParam.ProbDistUnivParam constructor.

Use this information to view and compare the data supplied to create distributions.

Purpose   Convert categorical array to signed 8-bit integer array

Syntax   `B = int8(A)`

Description   `B = int8(A)` converts the categorical array `A` to a signed 8-bit integer array. Each element of `B` contains the internal categorical level code for the corresponding element of `A`.

Undefined elements of `A` are assigned the value 0 in `B`. If `A` contains more than `intmax('int8')` levels, the internal codes will saturate to `intmax('int8')` when cast to `int8`.

See Also   For more information on signed integers, see "Integers" in the MATLAB documentation.

`double`, `uint8`

categorical.int16

Purpose   Convert categorical array to signed 16-bit integer array

Syntax   B = int16(A)

Description   B = int16(A) converts the categorical array A to a signed 16-bit integer array. Each element of B contains the internal categorical level code for the corresponding element of A.

Undefined elements of A are assigned the value 0 in B.

See Also   For more information on signed integers, see "Integers" in the MATLAB documentation.

double, uint16

Purpose   Convert categorical array to signed 32-bit integer array

Syntax   B = int32(A)

Description B = int32(A) converts the categorical array A to a signed 32-bit integer array. Each element of B contains the internal categorical level code for the corresponding element of A.

Undefined elements of A are assigned the value 0 in B.

See Also   For more information on signed integers, see "Integers" in the MATLAB documentation.

double, uint32

categorical.int64

Purpose	Convert categorical array to signed 64-bit integer array
Syntax	B = int64(A)
Description	B = int64(A) converts the categorical array A to a signed 64-bit integer array. Each element of B contains the internal categorical level code for the corresponding element of A.
	Undefined elements of A are assigned the value 0 in B.
See Also	For more information on signed integers, see "Integers" in the MATLAB documentation.
	double, uint64

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 2-33.) `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.

Examples   Display interaction plots for data with four 3-level factors named `'A'`,
`'B'`,`'C'`, and `'D'`:

interactionplot

```
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   "Grouped Data" on page 2-33

maineffectsplot, multivarichart

categorical.intersect

Purpose   Set intersection for categorical arrays

Syntax   C = intersect(A,B)

Description C = intersect(A,B) when A and B are categorical arrays returns a categorical vector C containing the values common to both A and B. The result C is sorted. The set of categorical levels for C is the sorted union of the sets of levels of the inputs, as determined by their labels.

[C,IA,IB] = UNION(A,B) also returns index vectors IA and IB such that C = A(IA) and C = B(IB).

See Also   ismember, setdiff, setxor, union, unique

invpred

Purpose	Inverse prediction

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`.

Examples

```
x = 4*rand(25,1);
y = 10 + 5*x + randn(size(x));
scatter(x,y)
x0 = invpred(x,y,20)
```

See Also polyfit, polyval, polyconf, polytool

categorical.ipermute

Purpose	Inverse permute dimensions of categorical array
Syntax	A = ipermute(B,order)
Description	A = ipermute(B,order) is the inverse of permute. ipermute rearranges the dimensions of the categorical array B so that permute(A,order) will produce B. The array produced has the same values of A but the order of the subscripts needed to access any particular element are rearranged as specified by order. The elements of order must be a rearrangement of the numbers from 1 to n.
See Also	permute

Purpose   Interquartile range

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

ProbDistUnivKernel.iqr

Purpose   Return interquartile range (IQR) for ProbDistUnivKernel object

Syntax   `Y = iqr(PD)`

Description   `Y = iqr(PD)` returns `Y`, the interquartile range for the ProbDistUnivKernel object `PD`. The interquartile range is the distance between the 75th and 25th percentiles.

Inputs

`PD`	An object of the class `ProbDistUnivKernel`.

Outputs

`Y`	The value of the interquartile range for the ProbDistUnivKernel object `PD`.

See Also   `iqr`
`ProbDistUnivKernel.icdf`

Purpose	Return interquartile range (IQR) for ProbDistUnivParam object
Syntax	Y = iqr(*PD*)
Description	Y = iqr(*PD*) returns Y, the interquartile range for the ProbDistUnivParam object *PD*. The interquartile range is the distance between the 75th and 25th percentiles.
Inputs	*PD* An object of the class ProbDistUnivParam.
Outputs	Y The value of the interquartile range for the ProbDistUnivParam object *PD*.
See Also	iqr ProbDistUnivParam.icdf

classregtree.isbranch

Purpose Test 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.

Examples 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
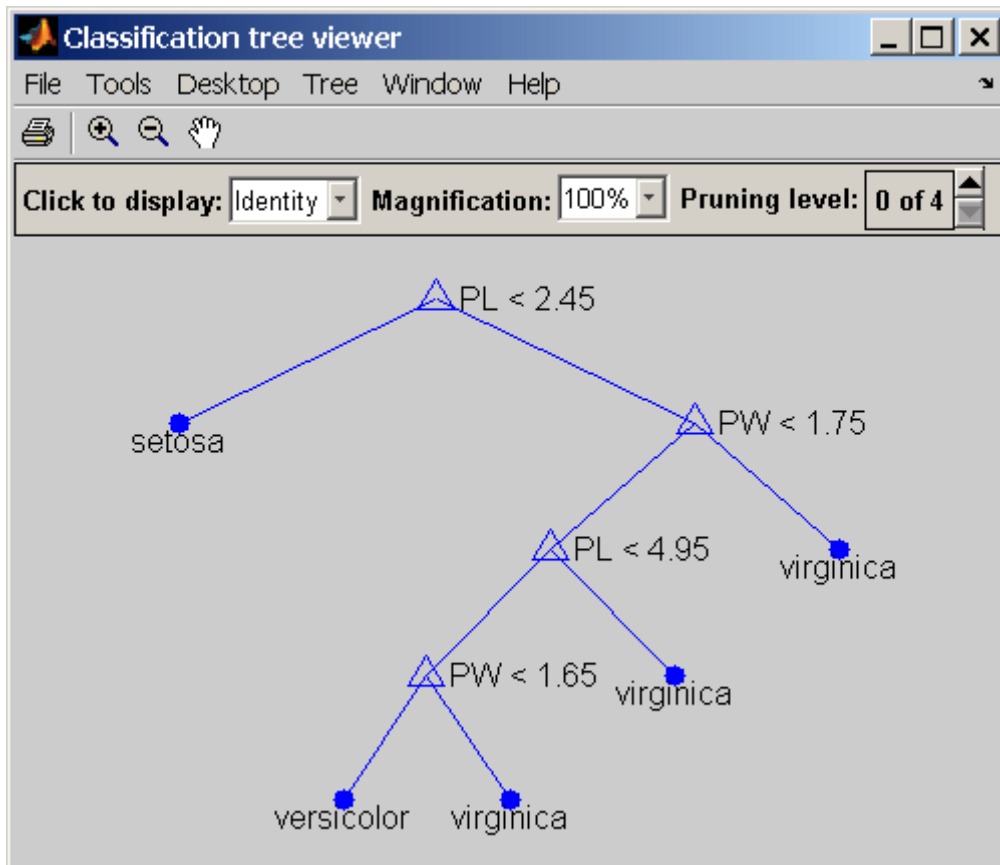
view(t)
```

```
ib = isbranch(t)
ib =
     1
     0
     1
     1
     0
     1
     0
```

classregtree.isbranch

```
           0
           0
```

References   [1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees.* Boca Raton, FL: CRC Press, 1984.

See Also   classregtree, cutvar, numnodes

Purpose   True for empty categorical array

Syntax   TF = isempty(A)

Description   TF = isempty(A) returns true (1) if A is an empty categorical array
                     and false (0) otherwise. An empty array has no elements, that is
                     numel(A)==0.

See Also   numel, size

dataset.isempty

Purpose   True for empty dataset array

Syntax   tf = isempty(A)

Description   tf = isempty(A) returns true (1) if A is an empty dataset and false (0) otherwise. An empty array has no elements, that is prod(size(A))==0.

See Also   size

Purpose   True if categorical arrays are equal

Syntax   TF = isequal(A,B)
                  TF = isequal(A,B,C,...)

Description   TF = isequal(A,B) is true (1) if the categorical arrays A and B are the
                  same class, have the same size and the same sets of levels, and contain
                  the same values, and false (0) otherwise.

                  TF = isequal(A,B,C,...) is true (1) if all the input arguments are
                  equal.

                  Elements with undefined levels are not considered equal to each other.

See Also   getlabels

categorical.islevel

Purpose	Test for 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 2-D 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.
Examples	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 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 2-D 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 True for elements of categorical array in set

Syntax
```
TF = ismember(A,levels)
[TF,LOC] = ismember(A,levels)
```

Description `TF = ismember(A,levels)` returns a logical array the same size as the categorical array `A`, containing true (1) where the level of the corresponding element of `A` is equal to one of the levels specified in `levels`, and false (0) otherwise. `levels` is a categorical array, or a cell array of strings or 2-D character array containing level labels.

`[TF,LOC] = ismember(A,levels)` also returns an index array `LOC` 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.

See Also intersect, islevel, setdiff, setxor, union, unique

categorical.isscalar

Purpose   True if categorical array is scalar

Syntax   TF = isscalar(A)

Description TF = isscalar(A) returns true (1) if the categorical array A is a 1-by-1 matrix, and false (0) otherwise.

See Also   isempty, isvector, size

Purpose   Test for undefined elements

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.

Examples   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

qrandstream.isvalid

Purpose   Test handle validity

Syntax   tf = isvalid(h)

Description tf = isvalid(h) performs an element-wise check for validity on the handle elements of h. The result is a logical array of the same dimensions as h, where each element is the element-wise validity result.

A handle is invalid if it has been deleted or if it is an element of a handle array and has not yet been initialized.

See Also   delete, qrandstream

Purpose   True if categorical array is vector

Syntax   TF = isvector(A)

Description   TF = isvector(A) returns true (1) if the categorical array A is a 1-by-n
                  or n-by-1 vector, where n >= 0, and false (0) otherwise.

See Also   isempty, isscalar, size

gmdistribution.Iters property

Purpose   Number of iterations

Description   The number of iterations of the algorithm.

> **Note** This property applies only to gmdistribution objects constructed with `fit`.

Purpose   Inverse Wishart random numbers

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 transpose of the inverse of the Cholesky factor of `sigma`, so that `DI'*DI = inv(sigma)`, where `inv` is the MATLAB inverse function. `DI` is lower-triangular and 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 an input in future calls to `iwishrnd`.

Note that different sources use different parametrizations 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`

jackknife

Purpose	Jackknife sampling
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.

Examples 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 "Pearson Systems" on page 5-85 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

$$JB = \frac{n}{6}(s^2 + \frac{(k-3)^2}{4})$$

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.

jbtest

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.

Examples   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. 163–172.

[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.

johnsrnd

Purpose	Johnson system random numbers

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 `coefs` 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`.

Examples   Generate random values with longer tails than a standard normal:

```
r = johnsrnd([-1.7 -.5 .5 1.7],1000,1);
qqplot(r);
```

johnsrnd

QQ Plot of Sample Data versus Standard Normal

Generate random values skewed to the right:

```
r = johnsrnd([-1.3 -.5 .5 1.7],1000,1);
qqplot(r);
```

QQ Plot of Sample Data versus Standard Normal

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

Syntax
```
C = join(A,B)
C = join(A,B,key)
C = join(A,B,param1,val1,param2,val2,...)
[C,IB] = join(...)
C = join(A,B,'Type',TYPE,...)
[C,IA,IB] = join(A,B,'Type',TYPE,...)
```

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 = 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 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). If A and B contain
variables with identical names, join adds the suffix '_left' and
'_right' to the corresponding variables in C.

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:

- 'Keys' — Specifies the variable to use as a keys in both A and B.

- 'LeftKeys' — Specifies the variable to use as a keys in A.

- 'RightKeys' — Specifies the variable to use as a keys in B.

dataset.join

You may provide either the `'Keys'` parameter, or both the `'LeftKeys'` and `'RightKeys'` 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. `'LeftKeys'` or `'RightKeys'` must both specify the same number of key variables, and `join` pairs the left and right keys are paired in the order specified.

- `'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,IB] = join(...)` returns an index vector `IB`, where `join` constructs `C` by horizontally concatenating `A(:,LeftVars)` and `B(IB,RightVars)`. `join` can also perform more complicated inner and outer join operations that allow a many-to-many correspondence between `A` and `B`, and allow unmatched observations in either `A` or `B`.

`C = join(A,B,'Type',TYPE,...)` performs the join operation specified by TYPE. TYPE is one of `'inner'`, `'leftouter'`, `'rightouter'`, `'fullouter'`, or `'outer'` (which is a synonym for `'fullouter'`). For an inner join, `C` only contains observations corresponding to a combination of key values that occurred in both `A` and `B`. For a left (or right) outer join, `C` also contains observations corresponding to keys in `A` (or `B`) that did not match any in `B` (or `A`). Variables in `C` taken from `A` (or `B`) contain null values in those observations. A full outer join is equivalent to a left and right outer join. `C` contains variables corresponding to the key variables from both `A` and `B`, and `join` sorts the observations in `C` by the key values.

For inner and outer joins, `C` contains variables corresponding to the key variables from both `A` and `B` by default, as well as all the remaining variables. `join` sorts the observations in the result `C` by the key values.

[C,IA,IB] = join(A,B,'Type',TYPE,...) returns index vectors IA and IB indicating the correspondence between observations in C and those in A and B. For an inner join, join constructs C by horizontally concatenating A(IA,LeftVars) and B(IB,RightVars). For an outer join, IA or IB may also contain zeros, indicating the observations in C that do not correspond to observations in A or B, respectively.

Examples   Create a dataset array from Fisher's iris data:

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

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

Purpose   Read-only string specifying name of kernel smoothing function for
ProbDistKernel object

Description   Kernel is a read-only property of the ProbDistKernel class. Kernel is
a string specifying the name of the kernel smoothing function used
to create a ProbDistKernel object.

Values   'normal'
'box'
'triangle'
'epanechnikov'

Use this information to view and compare the kernel smoothing
function used to create distributions.

See Also   ksdensity

kmeans

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 listed in the following table.

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.

Parameter	Value	
	`'sqEuclidean'`	Squared Euclidean distance (default). Each centroid is the mean of the points in that cluster.
	`'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.

kmeans

Parameter	Value	
`'emptyaction'`	Action to take if a cluster loses all its member observations.	
	`'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.
`'onlinephase'`	Flag indicating whether kmeans should perform an online update phase in addition to a batch update phase. The online phase can be time consuming for large data sets, but guarantees a solution that is a local minimum of the distance criterion, that is, a partition of the data where moving any single point to a different cluster increases the total sum of distances.	
	`'on'`	Perform online update (default).
	`'off'`	Do not perform online update.
`'options'`	Options for the iterative algorithm used to minimize the fitting criterion, as created by statset.	
`'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 3D array as the value for the `'start'` parameter.	

Parameter	Value	
'start'	Method used to choose the initial cluster centroid positions, sometimes known as *seeds*.	
	'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-D array, implying a value for the 'replicates' parameter from the array's third dimension.

Algorithm   kmeans uses a two-phase iterative algorithm to minimize the sum of point-to-centroid distances, summed over all k clusters:

1 The first phase uses *batch updates*, where each iteration consists of reassigning points to their nearest cluster centroid, all at once, followed by recalculation of cluster centroids. This phase occasionally does not converge to solution that is a local minimum, that is, a partition of the data where moving any single point to a different cluster increases the total sum of distances. This is more likely for small data sets. The batch phase is fast, but potentially only approximates a solution as a starting point for the second phase.

kmeans

2 The second phase uses *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 the second phase consists of one pass though all the points. The second phase will converge to a local minimum, although there may be other local minima with lower total sum of distances. The problem of finding the global minimum can only be solved in general by an exhaustive (or clever, or lucky) choice of starting points, but using several replicates with random starting points typically results in a solution that is a global minimum.

References

[1] Seber, G. A. F. *Multivariate Observations*. Hoboken, NJ: John Wiley & Sons, Inc., 1984.

[2] Spath, H. *Cluster Dissection and Analysis: Theory, FORTRAN Programs, Examples*. Translated by J. Goldschmidt. New York: Halsted Press, 1985.

Examples

The following creates two clusters from separated random data:

```
X = [randn(100,2)+ones(100,2);...
     randn(100,2)-ones(100,2)];
opts = statset('Display','final');

[idx,ctrs] = kmeans(X,2,...
                    'Distance','city',...
                    'Replicates',5,...
                    'Options',opts);
5 iterations, total sum of distances = 284.671
4 iterations, total sum of distances = 284.671
4 iterations, total sum of distances = 284.671
3 iterations, total sum of distances = 284.671
3 iterations, total sum of distances = 284.671

plot(X(idx==1,1),X(idx==1,2),'r.','MarkerSize',12)
hold on
plot(X(idx==2,1),X(idx==2,2),'b.','MarkerSize',12)
```

```
plot(ctrs(:,1),ctrs(:,2),'kx',...
     'MarkerSize',12,'LineWidth',2)
plot(ctrs(:,1),ctrs(:,2),'ko',...
     'MarkerSize',12,'LineWidth',2)
legend('Cluster 1','Cluster 2','Centroids',...
       'Location','NW')
```


See Also   linkage, clusterdata, silhouette

kruskalwallis

Purpose   Kruskal-Wallis test

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 2-33.)

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. NaNs 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.

Examples

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.

References

[1] Gibbons, J. D. *Nonparametric Statistical Inference.* New York: Marcel Dekker, 1985.

[2] Hollander, M., and D. A. Wolfe. *Nonparametric Statistical Methods.* Hoboken, NJ: John Wiley & Sons, Inc., 1999.

See Also

"Grouped Data" on page 2-33

anova1, boxplot, friedman, multcompare, ranksum

ksdensity

Purpose   Kernel smoothing density estimate

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:

Parameter	Value
`'censoring'`	A logical vector of the same length as `x`, indicating which entries are censoring times. Default is no censoring.

Parameter	Value
`'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.
`'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.

ksdensity

In place of the kernel functions listed above, you can specify another kernel function by using @ (such as @normpdf) or quotes (such as 'normpdf'). ksdensity calls the function with a single argument that is an array containing distances between data values in x and locations in xi where the density is evaluated. The function must return an array of the same size containing corresponding values of the kernel function. When the 'function' parameter value is 'pdf', this kernel function returns density values, otherwise it returns cumulative probability values. Specifying a custom kernel when the 'function' parameter value is 'icdf' returns 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);
```
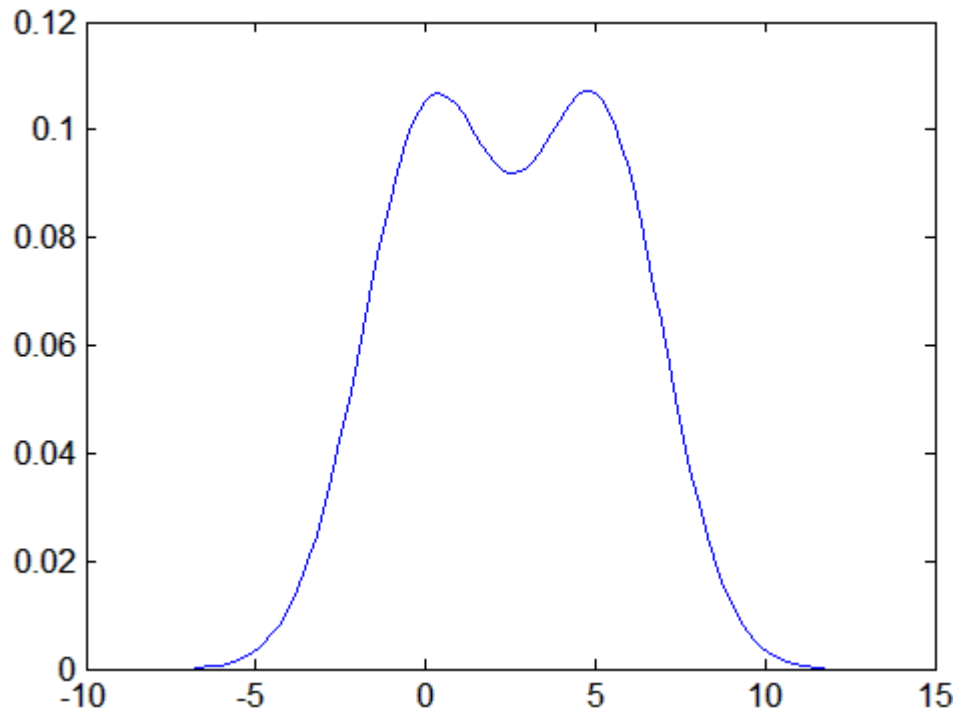
References   [1] Bowman, A. W., and A. Azzalini. *Applied Smoothing Techniques for Data Analysis*. New York: Oxford University Press, 1997.

See Also   hist, @ (function handle)

kstest

Purpose   One-sample Kolmogorov-Smirnov test

Syntax
```
h = kstest(x)
h = kstest(x,CDF)
h = kstest(x,CDF,alpha)
h = kstest(x,CDF,alpha,type)
[h,p,ksstat,cv] = kstest(...)
```

Description   h = kstest(x) performs a Kolmogorov-Smirnov test to compare the values in the data vector x to a standard normal distribution. The null hypothesis is that x has a standard normal distribution. The alternative hypothesis is that x does not have that distribution. The result h is 1 if the test rejects the null hypothesis at the 5% significance level, 0 otherwise.

The test statistic is:

$$\max(|F(x) - G(x)|)$$

where $F(x)$ is the empirical cdf and $G(x)$ is the standard normal cdf.

h = kstest(x,CDF) compares the distribution of x to the hypothesized continuous distribution defined by CDF, which is either a two-column matrix or a ProbDist object of the ProbDistUnivParam class or ProbDistUnivKernel class. When CDF is a matrix, column 1 contains a set of possible x values, and column 2 contains the corresponding hypothesized cumulative distribution function values $G(x)$. If possible, define CDF so that column 1 contains the values in x. If there are values in x not found in column 1 of CDF, kstest approximates $G(X)$ 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 ([ ]), kstest uses the standard normal distribution.

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,*type*) specifies the type of test using one of the following values for the string *type*:

- 'unequal' — Tests the alternative hypothesis that the population cdf is unequal to the specified CDF. This is the default.

- 'larger' — Tests the alternative hypothesis that the population cdf is larger than the specified CDF. The test statistic does not use the absolute value.

- 'smaller' — Tests the alternative hypothesis that the population cdf is smaller than the specified CDF. The test statistic does not use the absolute value.

[h,p,ksstat,cv] = kstest(...) also returns the *p*-value p, the test statistic ksstat, and the cutoff value cv for determining if ksstat is significant.

Examples   Generate evenly spaced numbers and perform a Kolmogorov-Smirnov test to see if they come from 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
```

The test fails to reject the null hypothesis that the values come from a standard normal distribution. This illustrates the difficulty of testing normality in small samples. (The Lilliefors test, implemented by the Statistics Toolbox function `lillietest`, may be more appropriate.)

The following figure illustrates the test statistic:

```
xx = -3:.1:5;
F = cdfplot(x);
hold on
G = plot(xx,normcdf(xx),'r-');
set(F,'LineWidth',2)
set(G,'LineWidth',2)
legend([F G],...
        'Empirical','Standard Normal',...
        'Location','NW')
```
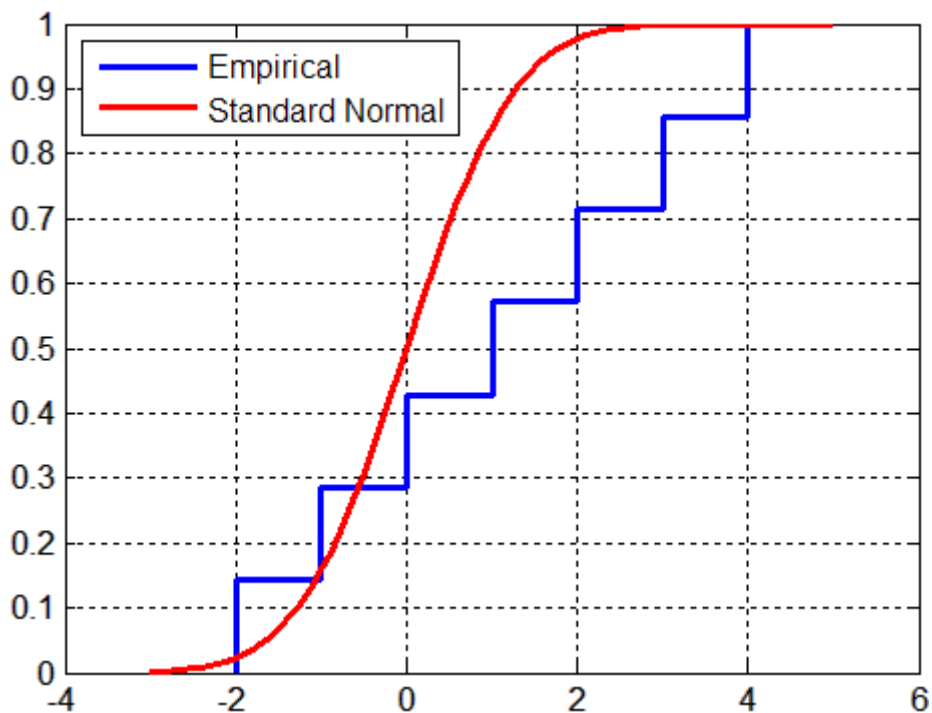
The test statistic k is the maximum difference between the curves.

Setting *type* to 'smaller' tests the alternative that the population cdf is smaller than the normal cdf:

```
[h,p,ksstat] = kstest(x,[],0.05,'smaller')
h =
    0
p =
    0.068181
k =
    0.41277
```

The test statistic is the same as before, but the *p*-value is smaller.

kstest

Setting *type* to `'larger'` changes the test statistic:

```
[h,p,k] = kstest(x,[],0.05,'larger')
h =
    0
p =
    0.77533
k =
    0.12706
```

References

[1]-[3] are references.

[1] Massey, F. J. "The Kolmogorov-Smirnov Test for Goodness of Fit." *Journal of the American Statistical Association.* Vol. 46, No. 253, 1951, pp. 68–78.

[2] Miller, L. H. "Table of Percentage Points of Kolmogorov Statistics." *Journal of the American Statistical Association.* Vol. 51, No. 273, 1956, pp. 111–121.

[3] Marsaglia, G., W. Tsang, and J. Wang. "Evaluating Kolmogorov's Distribution." *Journal of Statistical Software.* Vol. 8, Issue 18, 2003.

See Also   kstest2, lillietest

Purpose   Two-sample Kolmogorov-Smirnov test

Syntax
```
h = kstest2(x1,x2)
h = kstest2(x1,x2,alpha,type)
[h,p] = kstest2(...)
[h,p,ks2stat] = kstest2(...)
```

Description   `h = kstest2(x1,x2)` performs a two-sample Kolmogorov-Smirnov test to compare the distributions of the values in the two data vectors `x1` and `x2`. The null hypothesis is that `x1` and `x2` are from the same continuous distribution. The alternative hypothesis is that they are from different continuous distributions. The result `h` is `1` if the test rejects the null hypothesis at the 5% significance level; `0` otherwise.

The test statistic is:

$$\max(|F1(x) - F2(x)|)$$

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.

`h = kstest2(x1,x2,alpha)` specifies the significance level `alpha` for the test. The default is `0.05`.

`h = kstest2(x1,x2,alpha,type)` specifies the type of test using one of the following values for the string *type*:

- `'unequal'` — Tests the alternative hypothesis that the population cdfs are unequal. This is the default.

- `'larger'` — Tests the alternative hypothesis that the first population cdf is larger than the second population cdf. The test statistic does not use the absolute value.

- `'smaller'` — Tests the alternative hypothesis that the first population cdf is smaller than the second population cdf. The test statistic does not use the absolute value.

`[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,ks2stat] = kstest2(...)` also returns the *p*-value p and the
test statistic `ks2stat`.

Examples   The following test compares 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 =
     0
p =
    0.0774
k =
    0.5214
```

The following figure illustrates the test statistic:

```
F1 = cdfplot(x);
hold on
F2 = cdfplot(y)
set(F1,'LineWidth',2,'Color','r')
set(F2,'LineWidth',2)
legend([F1 F2],'F1(x)','F2(x)','Location','NW')
```
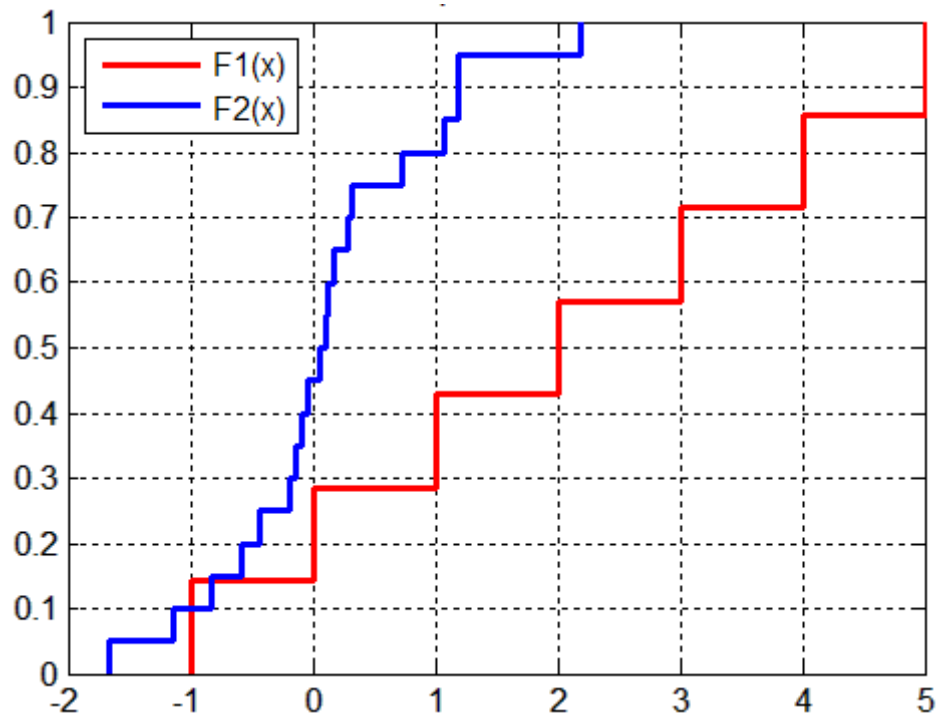
The test statistic k is the maximum difference between the curves.

References

[1] Massey, F. J. "The Kolmogorov-Smirnov Test for Goodness of Fit." *Journal of the American Statistical Association*. Vol. 46, No. 253, 1951, pp. 68–78.

[2] Miller, L. H. "Table of Percentage Points of Kolmogorov Statistics." *Journal of the American Statistical Association*. Vol. 51, No. 273, 1956, pp. 111–121.

[3] Marsaglia, G., W. Tsang, and J. Wang. "Evaluating Kolmogorov's Distribution." *Journal of Statistical Software*. Vol. 8, Issue 18, 2003.

[4] Stephens, M. A. "Use of the Kolmogorov-Smirnov, Cramer-Von Mises and Related Statistics Without Extensive Tables." *Journal of the Royal Statistical Society*. Series B, Vol. 32, No. 1, 1970, pp. 115–122.

See Also   kstest, lillietest

Purpose   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 NaNs 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 $\mu$ is the mean of $x$, $\sigma$ is the standard deviation of $x$, and $E(t)$ represents the expected value of the quantity $t$.

# kurtosis

> **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.

**Examples**
```
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**     Text labels for levels

**Description**     Text labels for levels. Access labels with `getlabels`.

# qrandstream.le

**Purpose**     Less than or equal relation for handles

**Syntax**      h1 <= h2

**Description**  Handles are equal if they are handles for the same object. All comparisons use a number associated with each handle object. Nothing can be assumed about the result of a handle comparison except that the repeated comparison of two handles in the same MATLAB session will yield the same result. The order of handle values is purely arbitrary and has no connection to the state of the handle objects being compared.

h1 <= h2 performs element-wise comparisons between handle arrays h1 and h2. h1 and h2 must be of the same dimensions unless one is a scalar. The result is a logical array of the same dimensions, where each element is an element-wise <= result.

If one of h1 or h2 is scalar, scalar expansion is performed and the result will match the dimensions of the array that is not scalar.

tf = le(h1, h2) stores the result in a logical array of the same dimensions.

**See Also**    qrandstream, eq, ge, gt, lt, ne

**Purpose**          Interval between points

**Description**      Interval between points (positive integer).

# dataset.length

**Purpose**     Length of dataset array

**Syntax**      n = length(A)

**Description**  n = length(A) returns the number of observations in the dataset A. length is equivalent to size(A,1).

**See Also**    size

# qrandset.length

**Purpose**      Length of point set

**Syntax**       length(p)

**Description**  length(p) returns the number of points in the point set p. It is equivalent to size(p, 1).

**See Also**     qrandset, size

# categorical.length

**Purpose**      Length of categorical array

**Syntax**       n = length(A)

**Description**  n = length(A) returns the size of the longest dimension of the
                 categorical array A when A is not empty. If A is a vector, this is the
                 same as its length. length is equivalent to max(size(x)) for nonempty
                 arrays, and 0 for empty arrays.

**See Also**     isempty, isscalar, size

**Purpose**     Element counts by level

**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.

**Examples**    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
```

categorical.levelcounts

See Also   islevel, ismember, summary

Purpose   Leverage

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.

Algorithm   [Q,R] = qr(x2fx(data,'model'));

leverage = (sum(Q'.*Q'))'

Examples   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.

leverage

References   [1] Goodall, C. R. "Computation Using the QR Decomposition."
Handbook in Statistics. Vol. 9, Amsterdam: Elsevier/North-Holland,
1993.

See Also   regstats

Purpose   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.

Criterion	Description
'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. The
                   default is k = 5.

See Also   haltonset, sobolset, lhsnorm, unifrnd

lhsnorm

Purpose	Latin hypercube sample from normal distribution

Syntax	`X = lhsnorm(mu,sigma,n)`
	`X = lhsnorm(mu,sigma,n,`*`flag`*`)`
	`[X,Z] = lhsnorm(...)`

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)`.

`[X,Z] = lhsnorm(...)` also returns Z, the original multivariate normal sample before the marginals are adjusted to obtain X.

References　　[1] Stein, M. "Large sample properties of simulations using latin hypercube sampling." *Technometrics*. Vol. 29, No. 2, 1987, pp. 143–151. Correction, Vol. 32, p. 367.

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 Lilliefors 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.

lillietest

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.

Examples   Use lillietest to determine if car mileage, in miles per gallon (MPG),
follows a normal distribution across different makes of cars:

```
load carbig.mat
[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*. Hoboken, NJ: John Wiley & Sons, Inc., 1980.

[2] 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.

[3] Lilliefors, H. W. "On the Kolmogorov-Smirnov test for normality with mean and variance unknown." *Journal of the American Statistical Association*. Vol. 62, 1967, pp. 399–402.

See Also   jbtest, kstest, kstest2, cdfplot

linhyptest

Purpose   Linear hypothesis test

Syntax
```
p = linhyptest(beta,COVB,c,H,dfe)
[p,t,r] = linhyptest(...)
```

Description   `p = linhyptest(beta,COVB,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. `COVB` 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 `COVB` estimate, or `Inf` if `COVB` is known rather than estimated.

`beta` is required. The remaining arguments have default values:

- `COVB = 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 following functions return outputs suitable for use as the `COVB` input argument to `linhyptest`: `nlinfit`, `coxphfit`, `glmfit`, `mnrfit`, `regstats`, `robustfit`. `nlinfit` returns `COVB` directly; the other functions return `COVB` in `stats.covb`.

`[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.

Examples   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)
Z = linkage(X,method,metric)
Z = linkage(X,method,inputs)
```

Description Z = linkage(y) creates a hierarchical cluster tree from the distances in y. y is a Euclidean distance matrix or a more general dissimilarity matrix, formatted as a vector, as returned by pdist.

Z is a (m–1)-by-3 matrix, where m is the number of observations in the original data. Columns 1 and 2 of Z contain cluster indices linked in pairs to form a binary tree. The leaf nodes are numbered from 1 to m. Leaf nodes are the singleton clusters from which all higher clusters are built. Each newly-formed cluster, corresponding to row Z(I,:), is assigned the index m+I. Z(I,1:2) contains the indices of the two component clusters that form cluster m+I. There are m-1 higher clusters which correspond to the interior nodes of the clustering tree. Z(I,3) contains the linkage distances between the two clusters merged in row Z(I,:).

For example, suppose there are 30 initial nodes and at step 12 cluster 5 and cluster 7 are combined. Suppose their distance at that time is 1.5. Then Z(12,:) will be [5, 7, 1.5]. The newly formed cluster will have index 12 + 30 = 42. If cluster 42 appears in a later row, it means the cluster created at step 12 is being combined into some larger cluster.

Z = linkage(y,*method*) creates the tree using the specified *method*. Methods differ from one another in how they measure the distance between clusters. Available methods are listed in the following table.

Method	Description
'average'	Unweighted average distance (UPGMA).
'centroid'	Centroid distance (UPGMC). Y must contain Euclidean distances.

Method	Description
'complete'	Furthest distance.
'median'	Weighted center of mass distance (WPGMC). Y must contain Euclidean distances.
'single'	Shortest distance. This is the default.
'ward'	Inner squared distance (minimum variance algorithm). Y must contain Euclidean distances.
'weighted'	Weighted average distance (WPGMA).

Note 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.

Z = linkage(X,*method*,*metric*) creates a hierarchical cluster tree from the observations in X. Rows in X correspond to observations and columns to variables. Pairwise distances are computed internally by calling pdist. *metric* is one of the distance metrics accepted by pdist.

Z = linkage(X,*method*,inputs) allows you to pass extra input arguments to pdist. inputs is a cell array containing input arguments.

Linkages

The following notation is used to describe the linkages used by the various methods:

- Cluster *r* is formed from clusters *p* and *q*.

- n_r is the number of objects in cluster *r*.

- x_{ri} is the *i*th object in cluster *r*.

linkage

- *Single linkage*, also called *nearest neighbor*, uses the smallest distance between objects in the two clusters:

$$d(r,s) = min(dist(x_{ri}, x_{sj})), i \in (i, ..., n_r), j \in (1, ..., n_s)$$

- *Complete linkage*, also called *furthest neighbor*, uses the largest distance between objects in the two clusters:

$$d(r,s) = max(dist(x_{ri}, x_{sj})), i \in (1, ..., n_r), j \in (1, ..., n_s)$$

- *Average linkage* uses the average distance between all pairs of objects in any two clusters:

$$d(r,s) = \frac{1}{n_r n_s} \sum_{i=1}^{n_r} \sum_{j=1}^{n_s} dist(x_{ri}, x_{sj})$$

- *Centroid linkage* uses the Euclidean distance between the centroids of the two clusters:

$$d(r,s) = \left\| \bar{x}_r - \bar{x}_s \right\|_2$$

where

$$\bar{x}_r = \frac{1}{n_r} \sum_{i=1}^{n} x_{ri}$$

- Median linkage uses the Euclidean distance between weighted centroids of the two clusters,

$$d(r,s) = \left\| \tilde{x}_r - \tilde{x}_s \right\|_2$$

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

$$\tilde{x}_r = \frac{1}{2}(\tilde{x}_p + \tilde{x}_q)$$

- *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 two clusters. 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:

$$d^2(r,s) = n_r n_s \frac{\|\bar{x}_r - \bar{x}_s\|_2^2}{(n_r + n_s)}$$

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.

Examples

```
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

$$p = F(x|\mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \int_0^x \frac{e^{\frac{-(\ln(t)-\mu)^2}{2\sigma^2}}}{t} dt$$

Examples

```
x = (0:0.2:10);
y = logncdf(x,0,1);
plot(x,y); grid;
xlabel('x'); ylabel('p');
```


References   [1] Evans, M., N. Hastings, and B. Peacock. *Statistical Distributions.* 2nd ed., Hoboken, NJ: John Wiley & Sons, Inc., 1993, pp. 102–105.

See Also   cdf, lognpdf, logninv, lognstat, lognfit, lognlike, lognrnd

lognfit

Purpose   Lognormal parameter estimates

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.

Examples   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   `mle`, `lognlike`, `lognpdf`, `logncdf`, `logninv`, `lognstat`, `lognrnd`

logninv

Purpose   Lognormal inverse 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 Pth 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

$$p = F(x|\mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}}\int_0^x \frac{e^{\frac{-(\ln(t)-\mu)^2}{2\sigma^2}}}{t}dt$$

Examples

```
p = (0.005:0.01:0.995);
crit = logninv(p,1,0.5);
plot(p,crit)
xlabel('Probability'); ylabel('Critical Value'); grid
```


References

[1] Evans, M., N. Hastings, and B. Peacock. *Statistical Distributions*. Hoboken, NJ: Wiley-Interscience, 2000. pp. 102–105.

See Also   icdf, logncdf, lognpdf, lognstat, lognfit, lognlike, lognrnd

lognlike

Purpose	Lognormal negative log-likelihood
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	lognfit, lognpdf, logncdf, logninv, lognstat, 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.

lognpdf

Examples

```
x = (0:0.02:10);
y = lognpdf(x,0,1);
plot(x,y); grid;
xlabel('x'); ylabel('p')
```


References   [1] Mood, A. M., F. A. Graybill, and D. C. Boes. *Introduction to the Theory of Statistics.* 3rd ed., New York: McGraw-Hill, 1974. pp. 540–541.

See Also   pdf, logncdf, logninv, lognstat, lognfit, lognlike, lognrnd

Purpose   Lognormal random numbers

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

lognrnd

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

Examples   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
```

References   [1] Evans, M., N. Hastings, and B. Peacock. *Statistical Distributions*. Hoboken, NJ: Wiley-Interscience, 2000. pp. 102–105.

See Also   random, lognpdf, logncdf, logninv, lognstat, lognfit, lognlike

Purpose   Lognormal mean and variance

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

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

Examples   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));
```

lognstat

```
[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
```

References   [1] Mood, A. M., F. A. Graybill, and D. C. Boes. *Introduction to the Theory of Statistics*. 3rd ed., New York: McGraw-Hill, 1974. pp. 540–541.

See Also   lognpdf, logncdf, logninv, lognfit, lognlike, lognrnd

Purpose   Lower Pareto tails parameters

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.

Examples   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

qrandstream.lt

Purpose   Less than relation for handles

Syntax   h1 < h2

Description h1 < h2 performs element-wise comparisons between handle arrays
h1 and h2. h1 and h2 must be of the same dimensions unless one is a
scalar. The result is a logical array of the same dimensions, where each
element is an element-wise < result.

If one of h1 or h2 is scalar, scalar expansion is performed and the result
will match the dimensions of the array that is not scalar.

tf = lt(h1, h2) stores the result in a logical array of the same
dimensions.

See Also   qrandstream, eq, ge, gt, le, ne

Purpose   Add least-squares line to scatter plot

Syntax   lsline
                 h = lsline

Description lsline superimposes a least-squares line on each scatter plot in the
                 current axes. Scatter plots are produced by the MATLAB scatter and
                 plot functions. Data points connected with solid, dashed, or dash-dot
                 lines (LineStyle '-', '--', or '.-') are not considered to be scatter
                 plots by lsline, and are ignored.

                 h = lsline returns a column vector of handles h to the least-squares
                 lines.

Examples   Use lsline together with scatter plots produced by scatter and
                 various line styles of plot:

```
x = 1:10;

y1 = x + randn(1,10);
scatter(x,y1,25,'b','*')
hold on

y2 = 2*x + randn(1,10);
plot(x,y2,'mo')

y3 = 3*x + randn(1,10);
plot(x,y3,'rx:')

y4 = 4*x + randn(1,10);
plot(x,y4,'g+--')
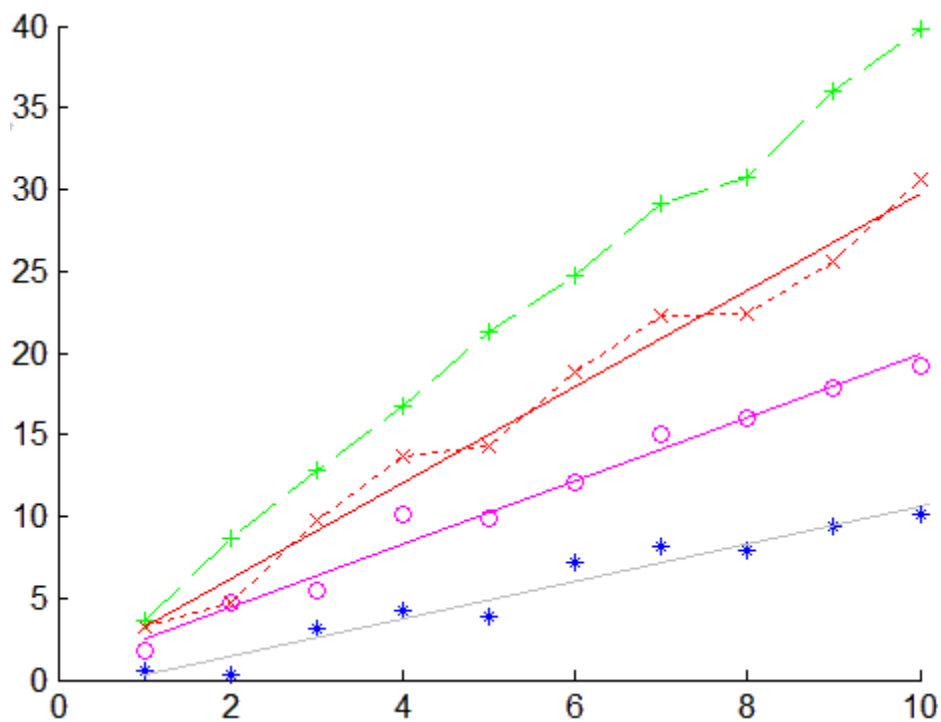
lsline
```

lsline

See Also scatter, plot, refline, refcurve, gline

Purpose   Mean or median absolute deviation

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) returns the median absolute deviation of the values in X. For vector input, y is median(abs(X-median(X))). For a matrix input, y is a row vector containing the median absolute deviation of each column of X. For *N*-dimensional arrays, mad operates along the first nonsingleton dimension of X.

Y = mad(X,0) is the same as mad(X), and returns the mean absolute deviation of the values in X.

mad(X,flag,dim) computes absolute deviations along the dimension dim of X. flag is 0 or 1 to indicate mean or median absolute deviation, respectively.

mad treats NaNs as missing values and removes them.

For normally distributed data, multiply mad by one of the following factors to obtain an estimate of the normal scale parameter *σ*:

- sigma = 1.253*mad(X,0) — For mean absolute deviation

- sigma = 1.4785*mad(X,1) — For median absolute deviation

Examples   The following compares the robustness of different scale estimates for normally distributed data in the presence of outliers:

```
x = normrnd(0,1,1,50);
xo = [x 10]; % Add outlier
```

```
r1 = std(xo)/std(x)
r1 =
    1.7385

r2 = mad(xo,0)/mad(x,0)
r2 =
    1.2306

r3 = mad(xo,1)/mad(x,1)
r3 =
    1.0602
```

References   [1] Mosteller, F., and J. Tukey. *Data Analysis and Regression.* Upper Saddle River, NJ: Addison-Wesley, 1977.

[2] Sachs, L. *Applied Statistics: A Handbook of Techniques.* New York: Springer-Verlag, 1984, p. 253.

See Also   std, range, iqr

Purpose   Mahalanobis distance

Syntax   d = mahal(Y,X)

Description   d = mahal(Y,X) computes the Mahalanobis distance (in squared units) of each observation in Y from the reference sample in matrix X. If Y is *n*-by-*m*, where *n* is the number of observations and *m* is the dimension of the data, d is *n*-by-1. X and Y must have the same number of columns, but can have different numbers of rows. X must have more rows than columns.

For observation I, the Mahalanobis distance is defined by d(I) = (Y(I,:)-mu)*inv(SIGMA)*(Y(I,:)-mu)', where mu and SIGMA are the sample mean and covariance of the data in X. mahal performs an equivalent, but more efficient, computation.

Examples   Generate some correlated bivariate data in X and compare the Mahalanobis and squared Euclidean distances of observations in Y:

```
X = mvnrnd([0;0],[1 .9;.9 1],100);
Y = [1 1;1 -1;-1 1;-1 -1];

d1 = mahal(Y,X) % Mahalanobis
d1 =
    1.3592
   21.1013
   23.8086
    1.4727

d2 = sum((Y-repmat(mean(X),4,1)).^2, 2) % Squared Euclidean
d2 =
    1.9310
    1.8821
    2.1228
    2.0739

scatter(X(:,1),X(:,2))
```

mahal

```
hold on
scatter(Y(:,1),Y(:,2),100,d1,'*','LineWidth',2)
hb = colorbar;
ylabel(hb,'Mahalanobis Distance')
legend('X','Y','Location','NW')
```


The observations in Y with equal coordinate values are much closer to X in Mahalanobis distance than observations with opposite coordinate values, even though all observations are approximately equidistant from the mean of X in Euclidean distance. The Mahalanobis distance, by considering the covariance of the data and the scales of the different variables, is useful for detecting outliers in such cases.

See Also   pdist, mahal

Purpose   Mahalanobis distance to component means

Syntax   D = mahal(obj,X)

Description D = mahal(obj,X) computes the Mahalanobis distance (in squared units) of each observation in X to the mean of each of the *k* components of the Gaussian mixture distribution defined by obj. obj is an object created by gmdistribution or fit. X is an *n*-by-*d* matrix, where *n* is the number of observations and *d* is the dimension of the data. D is *n*-by-*k*, with D(I,J) the distance of observation I from the mean of component J.

Examples   Generate data from a mixture of two bivariate Gaussian distributions using the mvnrnd function:

```
MU1 = [1 2];
SIGMA1 = [2 0; 0 .5];
MU2 = [-3 -5];
SIGMA2 = [1 0; 0 1];
X = [mvnrnd(MU1,SIGMA1,1000);mvnrnd(MU2,SIGMA2,1000)];
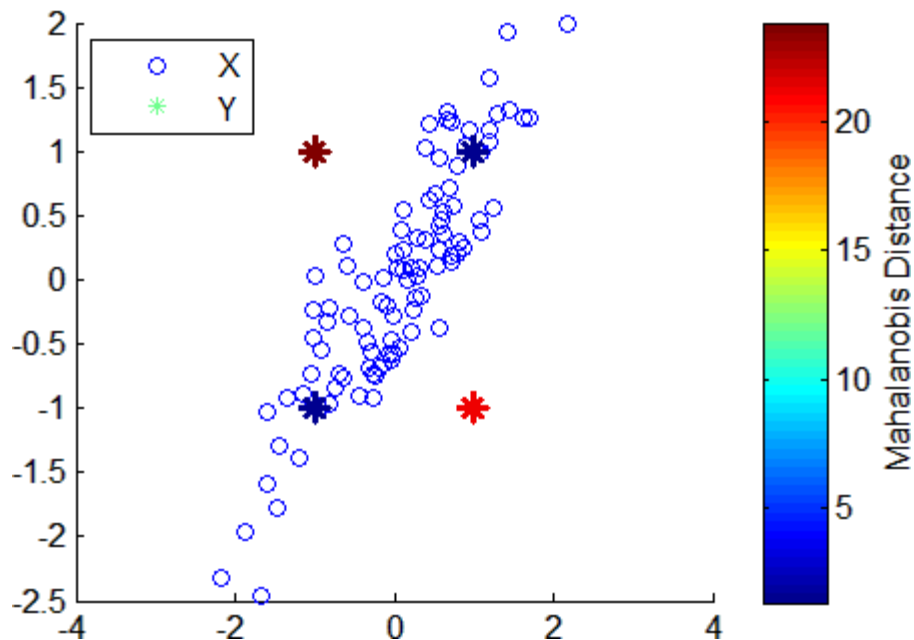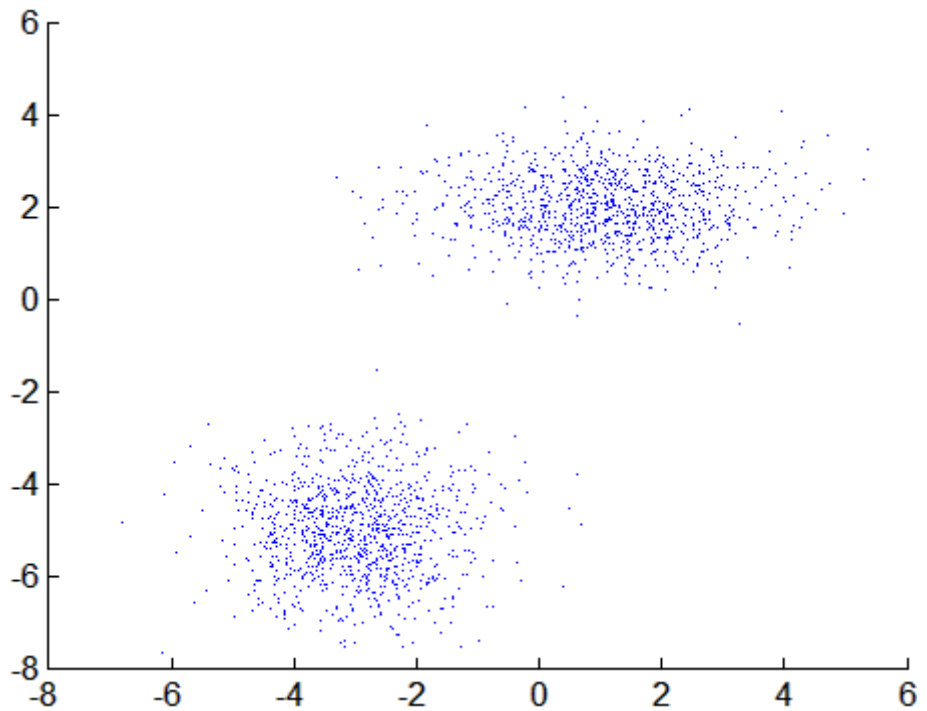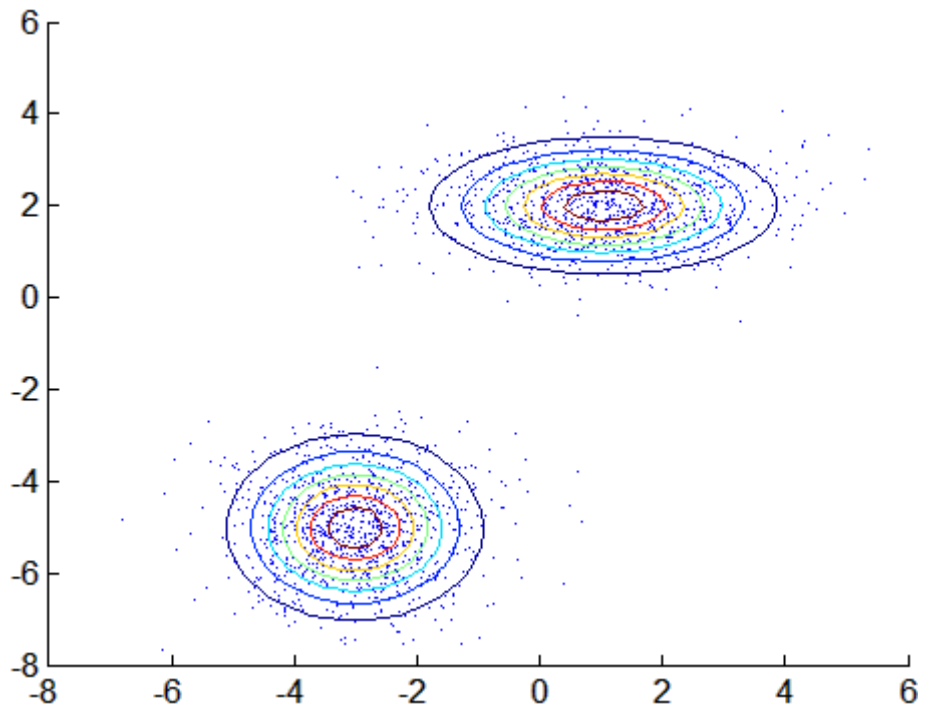
scatter(X(:,1),X(:,2),10,'.')
hold on
```

Fit a two-component Gaussian mixture model:

```
obj = gmdistribution.fit(X,2);
h = ezcontour(@(x,y)pdf(obj,[x y]),[-8 6],[-8 6]);
```

Compute the Mahalanobis distance of each point in X to the mean of each component of obj:

```
D = mahal(obj,X);

delete(h)
scatter(X(:,1),X(:,2),10,D(:,1),'.')
hb = colorbar;
ylabel(hb,'Mahalanobis Distance to Component 1')
```

gmdistribution.mahal

See Also   gmdistribution, cluster, posterior, mahal

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 2-33.)
                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.

maineffectsplot

Examples   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   "Grouped Data" on page 2-33

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 2-33.) 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.

Examples   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 3-D subspace.

References   [1] Krzanowski, W. J. *Principles of Multivariate Analysis: A User's Perspective.* New York: Oxford University Press, 1988.

See Also   "Grouped Data" on page 2-33

anova1, canoncorr, gscatter, gplotmatrix, 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 additional information.)

Method	Description
'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.

Examples   Let's analyze the larger car data set to determine which countries produce cars with the most similar characteristics.

```
load carbig
```

manovacluster

```
X = [MPG Acceleration Weight Displacement];
[d,p,stats] = manova1(X,Origin);
manovacluster(stats)
```


See Also   cluster, dendrogram, linkage, manova1

Purpose   Classification margin

Syntax   mar = margin(B,X,Y)
                mar = margin(B,X,Y,'param1',val1,'param2',val2,...)

Description mar = margin(B,X,Y) computes the classification margins for
                predictors X given true response Y. The Y can be either a numeric vector,
                character matrix, cell array of strings, categorical vector or logical
                vector. mar is a numeric array of size Nobs-by-NTrees, where Nobs is
                the number of rows of X and Y, and NTrees is the number of trees in the
                ensemble B. For observation I and tree J, mar(I,J) is the difference
                between the score for the true class and the largest score for other
                classes. This method is available for classification ensembles only.

                mar = margin(B,X,Y,'param1',val1,'param2',val2,...) specifies
                optional parameter name/value pairs:

                'mode'   String indicating how the method computes errors.
                            If set to 'cumulative' (default), margin computes
                            cumulative errors and mar is an Nobs-by-NTrees
                            matrix, where the first column gives error from
                            trees(1), second column gives error fromtrees(1:2)
                            etc, up to trees(1:NTrees). If set to 'individual',
                            mar is a Nobs-by-NTrees matrix, where each element
                            is an error from each tree in the ensemble. If set
                            to 'ensemble', mar a single column of length Nobs
                            showing the cumulative margins for the entire
                            ensemble.

                'trees'   Vector of indices indicating what trees to include
                            in this calculation. By default, this argument is set
                            to 'all' and the method uses all trees. If 'trees'
                            is a numeric vector, the method returns a vector of
                            length NTrees for 'cumulative' and 'individual'
                            modes, where NTrees is the number of elements in the
                            input vector, and a scalar for 'ensemble' mode. For
                            example, in the 'cumulative' mode, the first element

	gives error from `trees(1)`, the second element gives error from `trees(1:2)` etc.
`'treeweights'`	Vector of tree weights. This vector must have the same length as the `'trees'` vector. The method uses these weights to combine output from the specified trees by taking a weighted average instead of the simple non-weighted majority vote. You cannot use this argument in the `'individual'` mode.
`'useifort'`	Logical matrix of size Nobs-by-NTrees indicating which trees should be used to make predictions for each observation. By default the method uses all trees for all observations.

See Also TreeBagger.margin

Purpose   Classification margin

Syntax   mar = margin(B,X,Y)
                 mar = margin(B,X,Y,'param1',val1,'param2',val2,...)

Description mar = margin(B,X,Y) computes the classification margins for
predictors X given true response Y. The Y can be either a numeric vector,
character matrix, cell array of strings, categorical vector or logical
vector. mar is a numeric array of size Nobs-by-NTrees, where Nobs is
the number of rows of X and Y, and NTrees is the number of trees in the
ensemble B. For observation I and tree J, mar(I,J) is the difference
between the score for the true class and the largest score for other
classes. This method is available for classification ensembles only.

mar = margin(B,X,Y,'param1',val1,'param2',val2,...) specifies
optional parameter name/value pairs:

'mode'	String indicating how the method computes errors. If set to 'cumulative' (default), margin computes cumulative errors and mar is an Nobs-by-NTrees matrix, where the first column gives error from trees(1), second column gives error from trees(1:2) etc, up to trees(1:NTrees). If set to 'individual', mar is a Nobs-by-NTrees matrix, where each element is an error from each tree in the ensemble. If set to 'ensemble', mar a single column of length Nobs showing the cumulative margins for the entire ensemble.
'trees'	Vector of indices indicating what trees to include in this calculation. By default, this argument is set to 'all' and the method uses all trees. If 'trees' is a numeric vector, the method returns a vector of length NTrees for 'cumulative' and 'individual' modes, where NTrees is the number of elements in the input vector, and a scalar for 'ensemble' mode. For example, in the 'cumulative' mode, the first element

gives error from trees(1), the second element gives error from trees(1:2) etc.

'treeweights'   Vector of tree weights. This vector must have the same length as the 'trees' vector. The method uses these weights to combine output from the specified trees by taking a weighted average instead of the simple non-weighted majority vote. You cannot use this argument in the 'individual' mode.

'useifort'   Logical matrix of size Nobs-by-NTrees indicating which trees should be used to make predictions for each observation. By default the method uses all trees for all observations.

See Also   CompactTreeBagger.margin

Purpose   Nonclassical 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 3-D 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.

Examples
```
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   Multidimensional scaling of proximity matrix

Syntax   `[SC,EIGEN] = mdsProx(B,X)`
`[SC,EIGEN] = mdsProx(B,X,'param1',val1,'param2',val2,...)`

Description   `[SC,EIGEN] = mdsProx(B,X)` applies classical multidimensional scaling to the proximity matrix computed for the data in the matrix X, and returns scaled coordinates SC and eigenvalues EIGEN of the scaling transformation. The method applies multidimensional scaling to the matrix of distances defined as 1-`prox`, where `prox` is the proximity matrix returned by the `proximity` method.

You can supply the proximity matrix directly by using the `'data'` parameter.

`[SC,EIGEN] = mdsProx(B,X,'param1',val1,'param2',val2,...)` specifies optional parameter name/value pairs:

  `'data'`   Flag indicating how the method treats the X input argument. If set to `'predictors'` (default), `mdsProx` assumes X to be a matrix of predictors and used for computation of the proximity matrix. If set to `'proximity'`, the method treats X as a proximity matrix returned by the `proximity` method.

  `'colors'`   If you supply this argument, `mdsProx` makes overlaid scatter plots of two scaled coordinates using specified colors for different classes. You must supply the colors as a string with one character for each color. If there are more classes in the data than characters in the supplied string, `mdsProx` only plots the first C classes, where C is the length of the string. For regression or if you do not provide the vector of true class labels, the method uses the first color for all observations in X.

`'labels'`	Vector of true class labels for a classification ensemble. True class labels can be either a numeric vector, character matrix, or cell array of strings. If supplied, this vector must have as many elements as there are observations (rows) in X. This argument has no effect unless you also supply the `'colors'` argument.
`'mdscoords'`	Indices of the two scaled coordinates to plot. By default, `mdsProx` makes a scatter plot of the first and second scaled coordinates which correspond to the two largest eigenvalues. You can specify any other two indices not exceeding the dimensionality of the scaled data. This argument has no effect unless you also supply the `'colors'` argument.

See Also   cmdscale, TreeBagger.mdsProx, proximity

Purpose	Multidimensional scaling of proximity matrix

Syntax
```
[S,E] = mdsProx(B)
[S,E] = mdsProx(B,'param1',val1,'param2',val2,...)
```

Description   `[S,E] = mdsProx(B)` returns scaled coordinates, `S`, and eigenvalues, `E`, for the proximity matrix in the ensemble `B`. An earlier call to `fillProximities(B)` must create the proximity matrix.

`[S,E] = mdsProx(B,'param1',val1,'param2',val2,...)` specifies optional parameter name/value pairs:

`'keep'`	Array of indices of observations in the training data to use for multidimensional scaling. By default, this argument is set to `'all'`. If you provide numeric or logical indices, the method uses only the subset of the training data specified by these indices to compute the scaled coordinates and eigenvalues.
`'colors'`	If you supply this argument, `mdsProx` makes overlaid scatter plots of two scaled coordinates using specified colors for different classes. You must supply the colors as a string with one character for each color. If there are more classes in the data than characters in the supplied string, `mdsProx` only plots the first `C` classes, where `C` is the length of the string. For regression or if you do not provide the vector of true class labels, the method uses the first color for all observations in `X`.
`'mdscoords'`	Indices of the two scaled coordinates to plot. By default, `mdsProx` makes a scatter plot of the first and second scaled coordinates which correspond to the two largest eigenvalues. You can specify any other two indices not exceeding the dimensionality of the scaled data. This argument has no effect unless you also supply the `'colors'` argument.

See Also   cmdscale, CompactTreeBagger.mdsProx, fillProximities

ProbDistUnivKernel.median

Purpose	Return median of ProbDistUnivKernel object
Syntax	*M* = median(*PD*)
Description	*M* = median(*PD*) returns *M*, the median of the ProbDistUnivKernel object *PD*.
Inputs	*PD* An object of the class ProbDistUnivKernel.
Outputs	*M* The median of the ProbDistUnivKernel object *PD*.
See Also	median

Purpose   Return median of ProbDistUnivParam object

Syntax   *M* = median(*PD*)

Description   *M* = median(*PD*) returns *M*, the median of the ProbDistUnivParam
                     object *PD*.

Inputs   *PD*   An object of the class ProbDistUnivParam.

Outputs   *M*   The median of the ProbDistUnivParam object
                                                 PD.

See Also   median

ProbDistUnivParam.mean

Purpose	Return mean of ProbDistUnivParam object
Syntax	*M* = mean(*PD*)
Description	*M* = mean(*PD*) returns *M*, the mean of the ProbDistUnivParam object *PD*.
Inputs	*PD* An object of the class ProbDistUnivParam.
Outputs	*M* The mean of the ProbDistUnivParam object *PD*.
See Also	mean

Purpose   Mean classification margin

Syntax   mar = meanMargin(B,X,Y)
                  mar = meanMargin(B,X,Y,'param1',val1,'param2',val2,...)

Description   mar = meanMargin(B,X,Y) computes average classification margins for
                  predictors X given true response Y. The Y can be either a numeric vector,
                  character matrix, cell array of strings, categorical vector or logical
                  vector. meanMargin averages the margins over all observations (rows)
                  in X for each tree. mar is a matrix of size 1-by-NTrees, where NTrees
                  is the number of trees in the ensemble B. This method is available for
                  classification ensembles only.

                  mar = meanMargin(B,X,Y,'param1',val1,'param2',val2,...)
                  specifies optional parameter name/value pairs:

'mode'	String indicating how meanMargin computes errors. If set to 'cumulative' (default), is a vector of length NTrees where the first element gives mean margin from trees(1), second column gives mean margins from trees(1:2) etc, up to trees(1:NTrees). If set to 'individual', mar is a vector of length NTrees, where each element is a mean margin from each tree in the ensemble . If set to 'ensemble', mar is a scalar showing the cumulative mean margin for the entire ensemble.
'trees'	Vector of indices indicating what trees to include in this calculation. By default, this argument is set to 'all' and the method uses all trees. If 'trees' is a numeric vector, the method returns a vector of length NTrees for 'cumulative' and 'individual' modes, where NTrees is the number of elements in the input vector, and a scalar for 'ensemble' mode. For example, in the 'cumulative' mode, the first element

gives mean margin from `trees(1)`, the second element gives mean margin from `trees(1:2)` etc.

'treeweights' Vector of tree weights. This vector must have the same length as the `'trees'` vector. `meanMargin` uses these weights to combine output from the specified trees by taking a weighted average instead of the simple nonweighted majority vote. You cannot use this argument in the `'individual'` mode.

See Also   TreeBagger.meanMargin

Purpose   Mean classification margin

Syntax   mar = meanMargin(B,X,Y)
                mar = meanMargin(B,X,Y,'param1',val1,'param2',val2,...)

Description mar = meanMargin(B,X,Y) computes average classification margins for
                predictors X given true response Y. The Y can be either a numeric vector,
                character matrix, cell array of strings, categorical vector or logical
                vector. meanMargin averages the margins over all observations (rows)
                in X for each tree. mar is a matrix of size 1-by-NTrees, where NTrees
                is the number of trees in the ensemble B. This method is available for
                classification ensembles only.

                mar = meanMargin(B,X,Y,'param1',val1,'param2',val2,...)
                specifies optional parameter name/value pairs:

  'mode'   String indicating how meanMargin computes errors.
                If set to 'cumulative' (default), is a vector of length
                NTrees where the first element gives mean margin
                from trees(1), second column gives mean margins
                from trees(1:2) etc, up to trees(1:NTrees). If set
                to 'individual', mar is a vector of length NTrees,
                where each element is a mean margin from each tree
                in the ensemble . If set to 'ensemble', mar is a scalar
                showing the cumulative mean margin for the entire
                ensemble .

  'trees'   Vector of indices indicating what trees to include
                in this calculation. By default, this argument is set
                to 'all' and the method uses all trees. If 'trees'
                is a numeric vector, the method returns a vector of
                length NTrees for 'cumulative' and 'individual'
                modes, where NTrees is the number of elements in the
                input vector, and a scalar for 'ensemble' mode. For
                example, in the 'cumulative' mode, the first element

gives mean margin from `trees(1)`, the second element gives mean margin from `trees(1:2)` etc.

`'treeweights'`	Vector of tree weights. This vector must have the same length as the `'trees'` vector. `meanMargin` uses these weights to combine output from the specified trees by taking a weighted average instead of the simple nonweighted majority vote. You cannot use this argument in the `'individual'` mode.

See Also   CompactTreeBagger.meanMargin

Purpose   Flag to merge leaves that do not improve risk

Description   The `MergeLeaves` property is true if decision trees have their leaves with the same parent merged for splits that do not decrease the total risk, and false otherwise. The default value is false.

See Also   `classregtree`

ordinal.mergelevels

Purpose   Merge levels

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 2-D 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

CompactTreeBagger.Method property

Purpose   Method used by trees (classification or regression)

Description   The Method property is 'classification' for classification ensembles and 'regression' for regression ensembles.

Purpose   Method used by trees (classification or regression)

Description   The Method property is `'classification'` for classification ensembles and `'regression'` for regression ensembles.

mhsample

Purpose	Metropolis-Hastings sample

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 'logproppdf'. The density functions used in
Metropolis-Hastings algorithm are not necessarily normalized.

The proposal distribution $q(x,y)$ gives the probability density for
choosing x as the next point when y is the current point. It is sometimes
written as $q(x|y)$.

If the proppdf or logproppdf satisfies $q(x,y) = q(y,x)$, that is, the
proposal distribution is symmetric, mhsample implements Random
Walk Metropolis-Hastings sampling. If the proppdf or logproppdf
satisfies $q(x,y) = q(x)$, that is, the proposal distribution is independent of
current values, mhsample implements 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 kth point omitted in the generated sequence. Values beyond the kth 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)
h = get(gca,'Children');
set(h(2),'FaceColor',[.8 .8 1])
```

See Also   slicesample, rand

TreeBagger.MinLeaf property

Purpose   Minimum number of observations per tree leaf

Description   The `MinLeaf` property specifies the minimum number of observations per tree leaf. The default values are 1 for classification and 5 for regression. For `classregtree` training, the `'minparent'` value is set to `2*MinLeaf`.

See Also   `classregtree`

Purpose Maximum likelihood estimates

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*. Acceptable strings for *dist* are:

- 'beta'
- 'bernoulli'
- 'binomial'
- 'discrete uniform'
- 'exponential'
- 'extreme value'
- 'gamma'
- 'generalized extreme value'
- 'generalized pareto'
- 'geometric'
- 'lognormal'

- `'negative binomial'`

- `'normal'`

- `'poisson'`

- `'rayleigh'`

- `'uniform'`

- `'weibull'`

`[...] = 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 nonnegative 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.
`'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 Optimization Toolbox software 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 NaNs, 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.

Examples The following returns an MLE and a 95% confidence interval for the success probability of a binomial distribution with 20 trials:

```
data = binornd(20,0.75,100,1); % Simulated data, p = 0.75

[phat,pci] = mle(data,'distribution','binomial',...
                'alpha',.05,'ntrials',20)
phat =
    0.7370
```

```
pci =
    0.7171
    0.7562
```

See Also   betafit, binofit, evfit, expfit, gamfit, gevfit, gpfit, lognfit, nbinfit, normfit, mlecov, poissfit, raylfit, statset, unifit, wblfit

Purpose	Asymptotic covariance 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.

mlecov

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 table.

Parameter	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.

Parameter	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'`.

Examples   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`

mnpdf

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.

Examples

```
% 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')
title('Trinomial Distribution')
```

Trinomial Distribution

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

See Also   mnrnd

mnrfit

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 kth category are taken to be zero.

Note mnrfit automatically includes a constant term in all models. Do not enter a column of 1s directly into X.

---

mnrfit treats NaNs 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 ith observation in the jth 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., and J. A. Nelder. *Generalized Linear Models.* New York: Chapman & Hall, 1990.

See Also mnrval, glmfit, glmval, regress, regstats

Purpose	Multinomial random numbers

Syntax

```
r = mnrnd(n,p)
R = mnrnd(n,p,m)
R = mnrnd(N,P)
```

Description   `r = mnrnd(n,p)` returns random values `r` from the multinomial distribution with parameters `n` and `p`. `n` is a positive integer specifying the number of trials (sample size) for each multinomial outcome. `p` is a 1-by-k vector of multinomial probabilities, where k is the number of multinomial bins or categories. `p` must sum to one. (If `p` does not sum to one, `r` consists entirely of `NaN` values.) `r` is a 1-by-k vector, containing counts for each of the k multinomial bins.

`R = mnrnd(n,p,m)` returns `m` random vectors from the multinomial distribution with parameters `n` and `p`. `R` is a m-by-k matrix, where k is the number of multinomial bins or categories. Each row of `R` corresponds to one multinomial outcome.

`R = mnrnd(N,P)` generates outcomes from different multinomial distributions. `P` is a m-by-k matrix, where k is the number of multinomial bins or categories and each of the m rows contains a different set of multinomial probabilities. Each row of `P` must sum to one. (If any row of `P` does not sum to one, the corresponding row of `R` consists entirely of `NaN` values.) `N` is a m-by-1 vector of positive integers or a single positive integer (replicated by `mnrnd` to a m-by-1 vector). `R` is a m-by-k matrix. Each row of `R` is generated using the corresponding rows of `N` and `P`.

Examples   Generate 2 random vectors with the same probabilities:

```
n = 1e3;
p = [0.2,0.3,0.5];
R = mnrnd(n,p,2)
R =
   215    282    503
   194    303    503
```

Generate 2 random vectors with different probabilities:

```
n = 1e3;
P = [0.2, 0.3, 0.5; ...
     0.3, 0.4, 0.3;];
R = mnrnd(n,P)
R =
    186    290    524
    290    389    321
```

See Also mnpdf

Purpose	Multinomial logistic regression values
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 1s 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., and J. A. Nelder. *Generalized Linear Models.* New York: Chapman & Hall, 1990.

See Also mnrfit, glmfit, glmval

Purpose   Central moments

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.

Examples
```
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
```

moment

See Also   kurtosis, mean, skewness, std, var

Purpose   Input matrix of means MU

Description   Input matrix of means mu.

multcompare

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 17-717.

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	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)`%.
`'display'`	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 17-717.
`ctype`	Specifies the type of critical value to use for the multiple comparison. "Values of `ctype`" on page 17-714 describes the allowed values for `ctype`.

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 17-716 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 17-716

`[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`.

Value	Description
`'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	Values
`'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.

Click on the group you want to test

The means of groups USA and Japan are significantly different

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]
```

Click on the group you want to test

2 groups have means significantly different from st

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*. Hoboken, NJ: John Wiley & Sons, 1987.

[2] Milliken, G. A., and D. E. Johnson. *Analysis of Messy Data, Volume 1: Designed Experiments*. Boca Raton, FL: Chapman & Hall/CRC Press, 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 2-33.) 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.

Examples   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)
```

multivarichart

See Also   "Grouped Data" on page 2-33

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`. If the covariance
matrix is diagonal, containing variances along the diagonal and zero
covariances off the diagonal, `SIGMA` may also be specified as a 1-by-*d*
vector containing just the diagonal. 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, $\Pr\{V(1) \le X(1), V(2) \le X(2), \ldots, V(d) \le X(d)\}$.

`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 $d < 4$.

Examples

```
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));
```

References   [1] Drezner, Z. "Computation of the Trivariate Normal Integral."
Mathematics of Computation. Vol. 63, 1994, pp. 289–294.

[2] Drezner, Z., and G. O. Wesolowsky. "On the Computation of the
Bivariate Normal Integral." *Journal of Statistical Computation and
Simulation*. Vol. 35, 1989, pp. 101–107.

[3] Genz, A. "Numerical Computation of Rectangular Bivariate and
Trivariate Normal and t Probabilities." *Statistics and Computing*. Vol.
14, No. 3, 2004, pp. 251–260.

[4] Genz, A., and F. Bretz. "Numerical Computation of Multivariate
t Probabilities with Application to Power Calculation of Multiple

mvncdf

Contrasts." *Journal of Statistical Computation and Simulation*. Vol. 63, 1999, pp. 361–378.

[5] Genz, A., and F. Bretz. "Comparison of Methods for the Computation of Multivariate t Probabilities." *Journal of Computational and Graphical Statistics*. Vol. 11, No. 4, 2002, pp. 950–971.

See Also   mvnpdf, mvnrnd

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 a *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). If the covariance matrix is diagonal, containing
                variances along the diagonal and zero covariances off the diagonal,
                SIGMA may also be specified as a 1-by-*d* vector or a 1-by-*d*-by-*n* array,
                containing just the diagonal. 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.

Examples   mu = [1 -1];
                   SIGMA = [.9 .4; .4 .3];
                   X = mvnrnd(mu,SIGMA,10);
                   p = mvnpdf(X,mu,SIGMA);

mvnpdf

See Also   mvncdf, mvnrnd

Purpose	Multivariate linear regression

Syntax	`b = mvregress(X,Y)`
	`[b,SIGMA] = mvregress(X,Y)`
	`[b,SIGMA,RESID] = mvregress(X,Y)`
	`[b,SIGMA,RESID,COVB] = mvregress(...)`
	`[b,SIGMA,RESID,objective] = mvregress(...)`
	`[...] = mvregress(X,Y,`*param1*`,`*val1*`,`*param2*`,`*val2*`,...)`

Description `b = mvregress(X,Y)` returns a vector `b` of coefficient estimates for a multivariate regression of the d-dimensional responses in `Y` on the predictors in `X`. If $d = 1$, `X` can be an n-by-p matrix of p predictors at each of n observations. If $d \geq 1$, `X` can 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. `Y` is n-by-d. `b` is p-by-1.

Note To include a constant term in a model, `X` should contain a column of 1s.

`mvregress` treats NaNs 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.

`[b,SIGMA] = mvregress(X,Y)` also returns a d-by-d matrix `SIGMA` for the estimated covariance of `Y`.

`[b,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.

`[b,SIGMA,RESID,COVB] = mvregress(...)` also returns a matrix `COVB` for the estimated covariance of the coefficients. By default, or if the

'varformat' parameter is 'beta' (see below), COVB is the estimated covariance matrix of b. If the 'varformat' parameter is 'full', COVB is the combined estimated covariance matrix for beta and SIGMA.

[b,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. The default is 'mvn' for complete data, 'ecm' for missing data when the sample size is sufficient to estimate all parameters, and 'cwls' otherwise.

- 'covar0' — A *d*-by-*d* matrix to be used as the initial estimate for SIGMA. The 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.

- 'covtype' — Either 'full', to allow a full covariance matrix, or 'diagonal', to restrict the covariance matrix to be diagonal. The default is 'full'.

- 'maxiter' — Maximum number of iterations. The default is 100.

- '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.

The function should return logical `true` if the iterations should stop, or logical `false` if they should continue.

- `'param0'` — A vector of *p* elements to be used as the initial estimate for b. Default is a zero vector. Not used for the `'mvn'` algorithm.

- `'tolbeta'` — Convergence tolerance for b. The default is `sqrt(eps)`. Iterations continue until the `tolbeta` and `tolobj` conditions are met. The test for convergence at iteration k is

      norm(b(k)-b(k-1)) < sqrt(p)*tolbeta*(1+norm(b(k)))

  where `p = length(b)`.

- `'tolobj'` — Convergence tolerance for changes in the objective function. The 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.

- `'varformat'` — Either `'beta'` to compute COVB for b only (default), or `'full'` to compute COVB for both b and SIGMA.

- `'vartype'` — Either `'hessian'` to compute COVB using the Hessian or observed information (default), or `'fisher'` to compute COVB 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, Roderick J. A., and Donald B. Rubin. *Statistical Analysis with Missing Data.* 2nd ed., Hoboken, NJ: John Wiley & Sons, Inc., 2002.

[2] Meng, Xiao-Li, and Donald B. Rubin. "Maximum Likelihood Estimation via the ECM Algorithm." *Biometrika.* Vol. 80, No. 2, 1993, pp. 267–278.

[3] Sexton, Joe, and 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, 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.

See Also   mvregresslike, manova1

Purpose   Negative log-likelihood for multivariate regression

Syntax
```
nlogL = mvregresslike(X,Y,b,SIGMA,alg)
[nlogL,COVB] = mvregresslike(...)
[nlogL,COVB] = mvregresslike(...,type,format)
```

Description   `nlogL = mvregresslike(X,Y,b,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 `b` 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 `b`, 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,COVB] = mvregresslike(...)` also returns an estimated covariance matrix `COVB` of the parameter estimates `b`.

`[nlogL,COVB] = mvregresslike(...,type,format)` specifies the type and format of `COVB`.

type is either:

mvregresslike

- `'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 COVB for b only. This is the default.

- `'full'` — To compute COVB for both b and SIGMA.

See Also   mvregress, manova1

Purpose   Multivariate normal random numbers

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 the covariance matrix is diagonal, containing variances along the
                diagonal and zero covariances off the diagonal, SIGMA may also be
                specified as a 1-by-*d* vector or a 1-by-*d*-by-*n* array, containing just the
                diagonal. 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.

Examples   mu = [2 3];
                    SIGMA = [1 1.5; 1.5 3];
                    r = mvnrnd(mu,SIGMA,100);
                    plot(r(:,1),r(:,2),'+')

mvnrnd

See Also   mvnpdf, mvncdf

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:

mvtcdf

- `'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 $d < 4$.

Examples

```
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));
```

References

[1] Genz, A. "Numerical Computation of Rectangular Bivariate and Trivariate Normal and t Probabilities." *Statistics and Computing.* Vol. 14, No. 3, 2004, pp. 251–260.

[2] Genz, A., and F. Bretz. "Numerical Computation of Multivariate t Probabilities with Application to Power Calculation of Multiple Contrasts." *Journal of Statistical Computation and Simulation.* Vol. 63, 1999, pp. 361–378.

[3] Genz, A., and F. Bretz. "Comparison of Methods for the Computation of Multivariate t Probabilities." *Journal of Computational and Graphical Statistics.* Vol. 11, No. 4, 2002, pp. 950–971.

mvtcdf

See Also   mvtpdf, mvtrnd

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.

Examples   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))
```

mvtpdf

See Also   mvtcdf, mvtrnd

Purpose	Multivariate *t* random numbers
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.

Examples
```
SIGMA = [1 0.8;0.8 1];
R = mvtrnd(SIGMA,3,100);
plot(R(:,1),R(:,2),'+')
```

mvtrnd

See Also   `mvtpdf`, `mvtcdf`

Purpose   Number of observations (including observations with missing `group` values)

Description   Number of observations (including observations with missing `group` values).

NaiveBayes class

Purpose   Naive Bayes classifier

Description   A `NaiveBayes` object defines a Naive Bayes classifier. A Naive Bayes classifier assigns a new observation to the most probable class, assuming the features are conditionally independent given the class value.

Construction

NaiveBayes	Create NaiveBayes object

Methods

disp	Display `NaiveBayes` classifier object
display	Display `NaiveBayes` classifier object
fit	Create Naive Bayes classifier object by fitting training data
posterior	Compute posterior probability of each class for test data
predict	Predict class label for test data
subsasgn	Subscripted reference for `NaiveBayes` object
subsref	Subscripted reference for `NaiveBayes` object

Properties

CIsNonEmpty	Flag for non-empty classes
CLevels	Class levels
CNames	Class names
CPrior	Class priors
Dist	Distribution names

NClasses	Number of classes
NDims	Number of dimensions
Params	Parameter estimates

Copy Semantics

Value. To learn how this affects your use of the class, see Comparing Handle and Value Classes in the MATLAB Object-Oriented Programming documentation.

Examples

Predict the class label using the Naive Bayes classifier

```
load fisheriris
```

Use the default Gaussian distribution and a confusion matrix:

```
O1 = NaiveBayes.fit(meas,species);
C1 = O1.predict(meas);
cMat1 = confusionmat(species,C1)
```

This returns:

```
cMat1 =

    50     0     0
     0    47     3
     0     3    47
```

Use the Gaussian distribution for features 1 and 3 and use the kernel density estimation for features 2 and 4:

```
O2 = NaiveBayes.fit(meas,species,'dist',...
{'normal','kernel','normal','kernel'});
C2 = O2.predict(meas);
cMat2 = confusionmat(species,C2)
```

This returns:

```
cMat2 =
```

NaiveBayes class

```
    50     0     0
     0    47     3
     0     3    47
```

References   [1] Mitchell, T. (1997) Machine Learning, McGraw Hill.

[2] Vangelis M., Ion A., and Geogios P. Spam Filtering with Naive Bayes - Which Naive Bayes? (2006) Third Conference on Email and Anti-Spam.

[3] George H. John and Pat Langley. Estimating continuous distributions in bayesian classifiers (1995) the Eleventh Conference on Uncertainty in Artificial Intelligence.

See Also   "Naive Bayes Classification" on page 11-6

"Grouped Data" on page 2-33

Purpose   Create NaiveBayes object

Description   You cannot create a `NaiveBayes` classifier by calling the constructor.
                     Use `NaiveBayes.fit` to create a `NaiveBayes` classifier by fitting the
                     object to training data.

See Also   fit

Purpose	Covariance ignoring NaN values

Syntax
```
Y = nancov(X)
Y = nancov(X1,X2)
Y = nancov(...,1)
Y = nancov(...,'pairwise')
```

Description   `Y = nancov(X)` is the covariance `cov` of X, computed after removing observations with NaN values.

For vectors x, `nancov(x)` is the sample variance of the remaining elements, once NaN values are removed. For matrices X, `nancov(X)` is the sample covariance of the remaining observations, once observations (rows) containing any NaN values are removed.

`Y = nancov(X1,X2)`, where X1 and X2 are matrices with the same number of elements, is equivalent to `nancov(X)`, where X = [X1(:) X2(:)].

`nancov` removes the mean from each variable (column for matrix X) before calculating Y. If n is the number of remaining observations after removing observations with NaN values, `nancov` normalizes Y by either $n - 1$ or n, depending on whether $n > 1$ or $n = 1$, respectively. To specify normalization by n, use `Y = nancov(...,1)`.

`Y = nancov(...,'pairwise')` computes Y(i,j) using rows with no NaN values in columns i or j. The result Y may not be a positive definite matrix.

Examples   Generate random data for two variables (columns) with random missing values:

```
X = rand(10,2);
p = randperm(numel(X));
X(p(1:5)) = NaN
X =
    0.8147    0.1576
       NaN       NaN
    0.1270    0.9572
```

```
    0.9134      NaN
    0.6324      NaN
    0.0975   0.1419
    0.2785   0.4218
    0.5469   0.9157
    0.9575   0.7922
    0.9649      NaN
```

Establish a correlation between a third variable and the other two variables:

```
X(:,3) = sum(X,2)
X =
    0.8147   0.1576   0.9723
       NaN      NaN      NaN
    0.1270   0.9572   1.0842
    0.9134      NaN      NaN
    0.6324      NaN      NaN
    0.0975   0.1419   0.2394
    0.2785   0.4218   0.7003
    0.5469   0.9157   1.4626
    0.9575   0.7922   1.7497
    0.9649      NaN      NaN
```

Compute the covariance matrix for the three variables after removing observations (rows) with NaN values:

```
Y = nancov(X)
Y =
    0.1311   0.0096   0.1407
    0.0096   0.1388   0.1483
    0.1407   0.1483   0.2890
```

See Also   NaN, cov, var, nanvar

nanmax

Purpose   Maximum ignoring NaN values

Syntax
```
y = nanmax(X)
Y = nanmax(X1,X2)
y = nanmax(X,[],dim)
[y,indices] = nanmax(...)
```

Description   `y = nanmax(X)` is the maximum `max` of `X`, computed after removing NaN values.

For vectors `x`, `nanmax(x)` is the maximum of the remaining elements, once NaN values are removed. For matrices `X`, `nanmax(X)` is a row vector of column maxima, once NaN values are removed. For multidimensional arrays `X`, `nanmax` operates along the first nonsingleton dimension.

`Y = nanmax(X1,X2)` returns an array `Y` the same size as `X1` and `X2` with `Y(i,j) = nanmax(X1(i,j),X2(i,j))`. Scalar inputs are expanded to an array of the same size as the other input.

`y = nanmax(X,[],dim)` operates along the dimension `dim` of `X`.

`[y,indices] = nanmax(...)` also returns the row indices of the maximum values for each column in the vector `indices`.

Examples   Find column maxima and their indices for data with missing values:

```
X = magic(3);
X([1 6:9]) = repmat(NaN,1,5)
X =
   NaN     1   NaN
     3     5   NaN
     4   NaN   NaN
[y,indices] = nanmax(X)
y =
     4     5   NaN
indices =
     3     2     1
```

See Also   NaN, max, nanmin

Purpose   Mean ignoring NaN values

Syntax   y = nanmean(X)
                 y = nanmean(X,dim)

Description   y = nanmean(X) is the mean of X, computed after removing NaN values.

For vectors x, nanmean(x) is the mean of the remaining elements, once NaN values are removed. For matrices X, nanmean(X) is a row vector of column means, once NaN values are removed. For multidimensional arrays X, nanmean operates along the first nonsingleton dimension.

y = nanmean(X,dim) takes the mean along dimension dim of X.

Note If X contains a vector of all NaN values along some dimension, the vector is empty once the NaN values are removed, so the sum of the remaining elements is 0. Since the mean involves division by 0, its value is NaN. The output NaN is not a mean of NaN values.

Examples   Find column means for data with missing values:

```
X = magic(3);
X([1 6:9]) = repmat(NaN,1,5)
X =
   NaN     1   NaN
     3     5   NaN
     4   NaN   NaN
y = nanmean(X)
y =
   3.5000  3.0000   NaN
```

See Also   NaN, mean, nanmedian

nanmedian

Purpose	Median ignoring NaN values
Syntax	`y = nanmedian(X)` `y = nanmedian(X,dim)`

Description   `y = nanmedian(X)` is the median of X, computed after removing NaN values.

For vectors x, `nanmedian(x)` is the median of the remaining elements, once NaN values are removed. For matrices X, `nanmedian(X)` is a row vector of column medians, once NaN values are removed. For multidimensional arrays X, `nanmedian` operates along the first nonsingleton dimension.

`y = nanmedian(X,dim)` takes the mean along dimension dim of X.

Examples   Find column medians for data with missing values:

```
X = magic(3);
X([1 6:9]) = repmat(NaN,1,5)
X =
   NaN     1   NaN
     3     5   NaN
     4   NaN   NaN
y = nanmedian(X)
y =
   3.5000  3.0000   NaN
```

See Also   NaN, median, nanmean

Purpose   Minimum ignoring NaN values

Syntax
```
y = nanmin(X)
Y = nanmin(X1,X2)
y = nanmin(X,[],dim)
[y,indices] = nanmin(...)
```

Description   `y = nanmin(X)` is the minimum `min` of X, computed after removing NaN values.

For vectors `x`, `nanmin(x)` is the minimum of the remaining elements, once NaN values are removed. For matrices X, `nanmin(X)` is a row vector of column minima, once NaN values are removed. For multidimensional arrays X, `nanmin` operates along the first nonsingleton dimension.

`Y = nanmin(X1,X2)` returns an array Y the same size as X1 and X2 with `Y(i,j) = nanmin(X1(i,j),X2(i,j))`. Scalar inputs are expanded to an array of the same size as the other input.

`y = nanmin(X,[],dim)` operates along the dimension `dim` of X.

`[y,indices] = nanmin(...)` also returns the row indices of the minimum values for each column in the vector `indices`.

Examples   Find column minima and their indices for data with missing values:

```
X = magic(3);
X([1 6:9]) = repmat(NaN,1,5)
X =
   NaN     1   NaN
     3     5   NaN
     4   NaN   NaN
[y,indices] = nanmin(X)
y =
     3     1   NaN
indices =
     2     1     1
```

See Also   NaN, min, nanmax

nanstd

Purpose	Standard deviation ignoring NaN values

Syntax	`y = nanstd(X)`
	`y = nanstd(X,1)`
	`y = nanstd(X,flag,dim)`

Description   `y = nanstd(X)` is the standard deviation `std` of X, computed after removing NaN values.

For vectors x, `nanstd(x)` is the sample standard deviation of the remaining elements, once NaN values are removed. For matrices X, `nanstd(X)` is a row vector of column sample standard deviations, once NaN values are removed. For multidimensional arrays X, `nanstd` operates along the first nonsingleton dimension.

If n is the number of remaining observations after removing observations with NaN values, `nanstd` normalizes y by $n - 1$. To specify normalization by n, use `y = nanstd(X,1)`.

`y = nanstd(X,flag,dim)` takes the standard deviation along the dimension dim of X. The flag is 0 or 1 to specify normalization by $n - 1$ or n, respectively, where n is the number of remaining observations after removing observations with NaN values.

Examples   Find column standard deviations for data with missing values:

```
X = magic(3);
X([1 6:9]) = repmat(NaN,1,5)
X =
   NaN     1   NaN
     3     5   NaN
     4   NaN   NaN
y = nanstd(X)
y =
   0.7071  2.8284   NaN
```

See Also   NaN, std, nanvar, nanmean

Purpose   Sum ignoring NaN values

Syntax   y = nansum(X)
                   y = nansum(X,dim)

Description   y = nansum(X) is the sum of X, computed after removing NaN values.

For vectors x, nansum(x) is the sum of the remaining elements, once NaN values are removed. For matrices X, nansum(X) is a row vector of column sums, once NaN values are removed. For multidimensional arrays X, nansum operates along the first nonsingleton dimension.

y = nansum(X,dim) takes the sum along dimension dim of X.

Note If X contains a vector of all NaN values along some dimension, the vector is empty once the NaN values are removed, so the sum of the remaining elements is 0. The output 0 is not a sum of NaN values.

Examples   Find column sums for data with missing values:

```
X = magic(3);
X([1 6:9]) = repmat(NaN,1,5)
X =
   NaN     1   NaN
     3     5   NaN
     4   NaN   NaN
y = nansum(X)
y =
     7     6     0
```

See Also   NaN, sum

nanvar

Purpose   Variance, ignoring NaN values

Syntax
```
y = nanvar(X)
y = nanvar(X,1)
y = nanvar(X,w)
y = nanvar(X,w,dim)
```

Description   `y = nanvar(X)` is the variance `var` of X, computed after removing NaN values.

For vectors x, `nanvar(x)` is the sample variance of the remaining elements, once NaN values are removed. For matrices X, `nanvar(X)` is a row vector of column sample variances, once NaN values are removed. For multidimensional arrays X, `nanvar` operates along the first nonsingleton dimension.

`nancov` removes the mean from each variable (column for matrix X) before calculating Y. If n is the number of remaining observations after removing observations with NaN values, `nanvar` normalizes y by either $n - 1$ or n, depending on whether $n > 1$ or $n = 1$, respectively. To specify normalization by n, use `y = nanvar(X,1)`.

`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 values 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 $n - 1$.

Examples   Find column standard deviations for data with missing values:

```
X = magic(3);
X([1 6:9]) = repmat(NaN,1,5)
X =
   NaN     1   NaN
     3     5   NaN
     4   NaN   NaN
y = nanvar(X)
```

```
y =
   0.5000  8.0000  NaN
```

See Also   NaN, var, nanstd, nanmean

nbincdf

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

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

Examples
```
x = (0:15);
p = nbincdf(x,3,0.5);
stairs(x,p)
```

See Also   cdf, nbinpdf, nbininv, nbinstat, nbinfit, nbinrnd

nbinfit

Purpose	Negative binomial parameter estimates

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`

Purpose   Negative binomial inverse 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.

Examples   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, nbinpdf, nbinstat, nbinfit, nbinrnd

nbinpdf

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

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

Examples
```
x = (0:10);
y = nbinpdf(x,3,0.5);
plot(x,y,'+')
set(gca,'Xlim',[-0.5,10.5])
```

See Also   pdf, nbincdf, nbininv, nbinstat, nbinfit, nbinrnd

nbinrnd

Purpose	Negative binomial random numbers

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.

Examples   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   random, nbinpdf, nbincdf, nbininv, nbinstat, nbinfit

Purpose   Negative binomial mean and variance

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.

Examples
```
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 nbinpdf, nbincdf, nbininv, nbinfit, nbinrnd

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

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

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

Examples Compare the noncentral F cdf with $\delta = 10$ 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.* Hoboken, NJ: John Wiley & Sons, Inc., 1970, pp. 189–200.

See Also   cdf, ncfpdf, ncfinv, ncfstat, ncfrnd

ncfinv

Purpose	Noncentral *F* inverse 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.

Examples 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
```

ncfinv

References   [1] Evans, M., N. Hastings, and B. Peacock. *Statistical Distributions*. Hoboken, NJ: Wiley-Interscience, 2000.

[2] Johnson, N., and S. Kotz. *Distributions in Statistics: Continuous Univariate Distributions-2*. Hoboken, NJ: John Wiley & Sons, Inc., 1970, pp. 189–200.

See Also   icdf, ncfcdf, ncfpdf, ncfstat, , ncfrnd

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.

Examples   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.* Hoboken, NJ: John Wiley & Sons, Inc.,
1970, pp. 189–200.

See Also   `pdf`, `ncfcdf`, `ncfinv`, `ncfstat`, `ncfrnd`

ncfrnd

Purpose   Noncentral *F* random numbers

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.

Examples   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
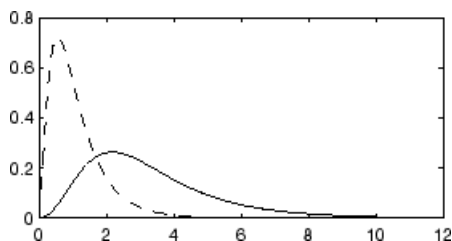
r1 = frnd(10,100,1,6)
r1 =
  0.9826  0.5911  1.0967  0.9681  2.0096  0.6598
```

References   [1] Johnson, N., and S. Kotz. *Distributions in Statistics: Continuous Univariate Distributions-2.* Hoboken, NJ: John Wiley & Sons, Inc., 1970, pp. 189–200.

See Also   random, ncfpdf, ncfcdf, ncfinv, ncfstat

ncfstat

Purpose	Noncentral *F* mean and variance

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 v_1, v_2, and δ is

$$\frac{v_2(\delta + v_1)}{v_1(v_2 - 2)}$$

where $v_2 > 2$.

The variance is

$$2\left(\frac{v_2}{v_1}\right)^2 \left[\frac{(\delta + v_1)^2 + (2\delta + v_1)(v_2 - 2)}{(v_2 - 2)^2(v_2 - 4)}\right]$$

where $v_2 > 4$.

Examples

```
[m,v]= ncfstat(10,100,4)
m =
  1.4286
v =
  0.4252
```

References   [1] Evans, M., N. Hastings, and B. Peacock. *Statistical Distributions*. 2nd ed., Hoboken, NJ: John Wiley & Sons, Inc., 1993, pp. 73–74.

[2] Johnson, N., and S. Kotz. *Distributions in Statistics: Continuous Univariate Distributions-2.* Hoboken, NJ: John Wiley & Sons, Inc., 1970, pp. 189–200.

See Also   ncfpdf, ncfcdf, ncfinv, ncfrnd

NaiveBayes.NClasses property

Purpose   Number of classes

Description   The `NClasses` property specifies the number of classes in the grouping
variable used to create the Naive Bayes classifier.

Purpose Number k of mixture components

Description The number k of mixture components.

nctcdf

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.

Examples   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 ed., Hoboken, NJ: John Wiley & Sons, Inc., 1993, pp. 147–148.

                 [2] Johnson, N., and S. Kotz. *Distributions in Statistics: Continuous
                 Univariate Distributions-2*. Hoboken, NJ: John Wiley & Sons, Inc.,
                 1970, pp. 201–219.

See Also   cdf, nctpdf, nctinv, nctstat, nctrnd

Purpose   Noncentral *t* inverse cumulative distribution function

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.

Examples
```
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 ed., Hoboken, NJ: John Wiley & Sons, Inc., 1993, pp. 147–148.

                   [2] Johnson, N., and S. Kotz. *Distributions in Statistics: Continuous
                   Univariate Distributions-2*. Hoboken, NJ: John Wiley & Sons, Inc.,
                   1970, pp. 201–219.

See Also   icdf, nctcdf, nctpdf, nctstat, nctrnd

nctpdf

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.

Examples   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)';
nct = nctpdf(x,10,1);
t = tpdf(x,10);

plot(x,nct,'b-','LineWidth',2)
hold on
plot(x,t,'g--','LineWidth',2)
legend('nct','t')
```

References [1] Evans, M., N. Hastings, and B. Peacock. *Statistical Distributions*. 2nd ed., Hoboken, NJ: John Wiley & Sons, Inc., 1993, pp. 147–148.

[2] Johnson, N., and S. Kotz. *Distributions in Statistics: Continuous Univariate Distributions-2*. Hoboken, NJ: John Wiley & Sons, Inc., 1970, pp. 201–219.

See Also pdf, nctcdf, nctinv, nctstat, nctrnd

nctrnd

Purpose   Noncentral *t* random numbers

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`.

Examples
```
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 ed., Hoboken, NJ: John Wiley & Sons, Inc., 1993, pp. 147–148.

[2] Johnson, N., and S. Kotz. *Distributions in Statistics: Continuous Univariate Distributions-2.* Hoboken, NJ: John Wiley & Sons, Inc., 1970, pp. 201–219.

See Also   `random`, `nctpdf`, `nctcdf`, `nctinv`, `nctstat`

Purpose   Noncentral *t* mean and variance

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 v and δ is

$$\frac{\delta(v/2)^{1/2}\Gamma((v-1)/2)}{\Gamma(v/2)}$$

where $v > 1$.

The variance is

$$\frac{v}{(v-2)}(1+\delta^2)-\frac{v}{2}\delta^2\left[\frac{\Gamma((v-1)/2)}{\Gamma(v/2)}\right]^2$$

where $v > 2$.

Examples   ```
[m,v] = nctstat(10,1)

m =
  1.0837

v =
  1.3255
```

References   [1] Evans, M., N. Hastings, and B. Peacock. *Statistical Distributions*. 2nd ed., Hoboken, NJ: John Wiley & Sons, Inc., 1993, pp. 147–148.

nctstat

[2] Johnson, N., and S. Kotz. *Distributions in Statistics: Continuous Univariate Distributions-2.* Hoboken, NJ: John Wiley & Sons, Inc., 1970, pp. 201–219.

See Also   nctpdf, nctcdf, nctinv, 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

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

Examples   Compare the noncentral chi-square cdf with DELTA = 2 to the
chi-square cdf with the same number of degrees of freedom (4):

```
x = (0:0.1:10)';
ncx2 = ncx2cdf(x,4,2);
chi2 = chi2cdf(x,4);

plot(x,ncx2,'b-','LineWidth',2)
hold on
plot(x,chi2,'g--','LineWidth',2)
legend('ncx2','chi2','Location','NW')
```
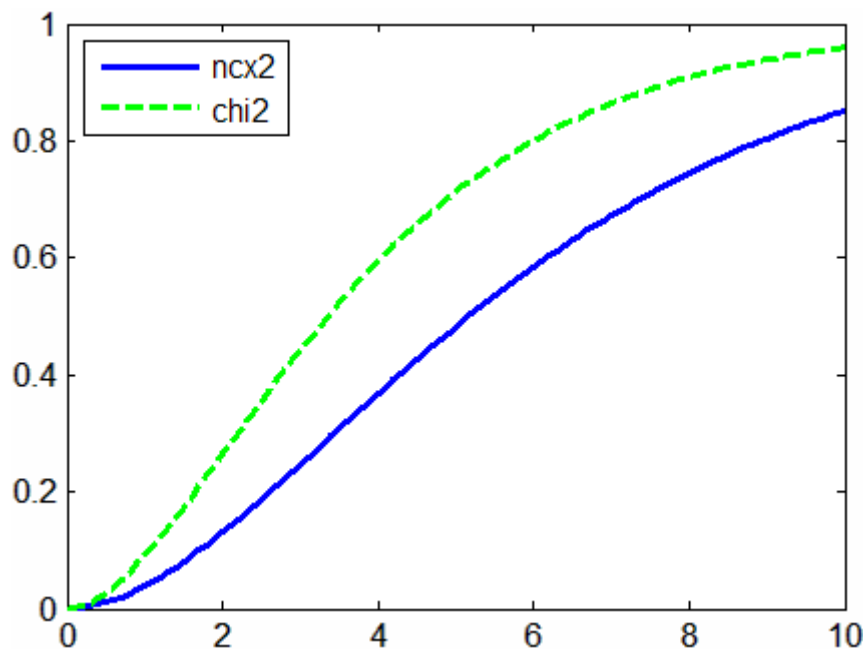
References   [1] Johnson, N., and S. Kotz. *Distributions in Statistics: Continuous Univariate Distributions-2.* Hoboken, NJ: John Wiley & Sons, Inc., 1970, pp. 130–148.

See Also   cdf, ncx2pdf, ncx2inv, ncx2stat, ncx2rnd

Purpose	Noncentral chi-square inverse 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.
Examples	``` 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 ed., Hoboken, NJ: John Wiley & Sons, Inc., 1993, pp. 50–52.

[2] Johnson, N., and S. Kotz. *Distributions in Statistics: Continuous Univariate Distributions-2*. Hoboken, NJ: John Wiley & Sons, Inc., 1970, pp. 130–148. |
| **See Also** | icdf, ncx2cdf, ncx2pdf, ncx2stat, ncx2rnd |

ncx2pdf

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.

Examples   Compare the noncentral chi-square pdf with `DELTA = 2` to the
chi-square pdf with the same number of degrees of freedom (4):

```
x = (0:0.1:10)';
ncx2 = ncx2pdf(x,4,2);
chi2 = chi2pdf(x,4);

plot(x,ncx2,'b-','LineWidth',2)
hold on
plot(x,chi2,'g--','LineWidth',2)
legend('ncx2','chi2')
```
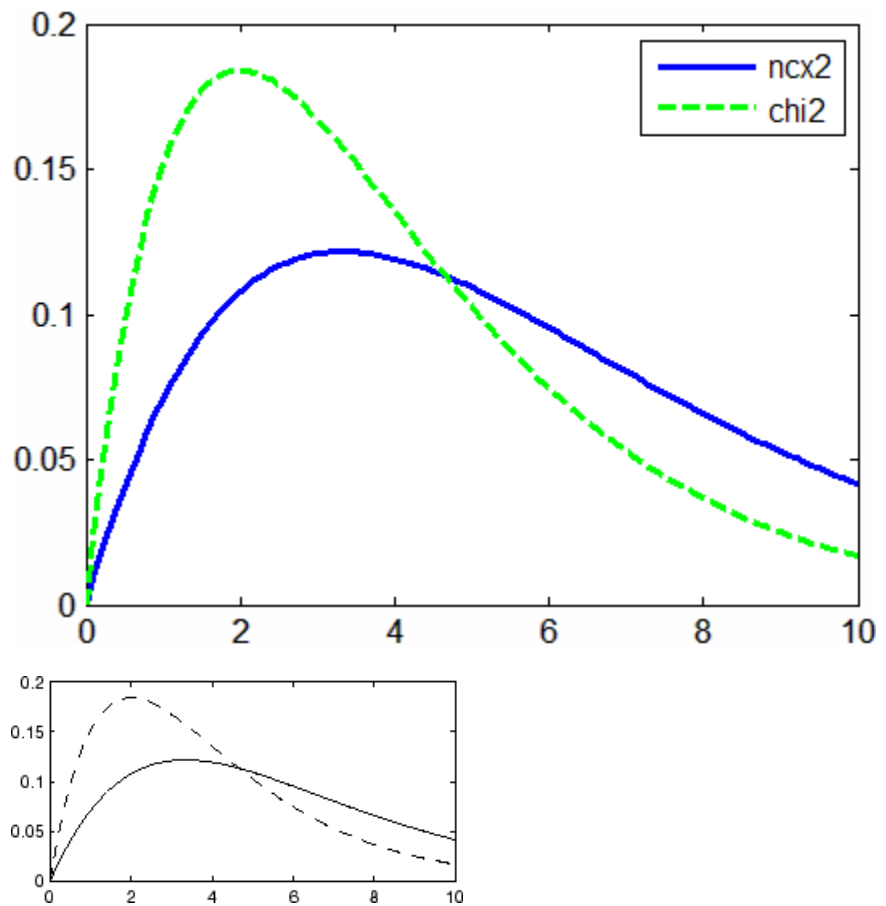
References   [1] Johnson, N., and S. Kotz. *Distributions in Statistics: Continuous Univariate Distributions-2.* Hoboken, NJ: John Wiley & Sons, Inc., 1970, pp. 130–148.

See Also   pdf, ncx2cdf, ncx2inv, ncx2stat, ncx2rnd

ncx2rnd

Purpose   Noncentral chi-square random numbers

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.

Examples
```
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 ed., Hoboken, NJ: John Wiley & Sons, Inc., 1993, pp. 50–52.

[2] Johnson, N., and S. Kotz. *Distributions in Statistics: Continuous Univariate Distributions-2.* Hoboken, NJ: John Wiley & Sons, Inc., 1970, pp. 130–148.

See Also   random, ncx2pdf, ncx2cdf, ncx2inv, ncx2stat

ncx2stat

Purpose	Noncentral chi-square mean and variance
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 v and δ is $v + \delta$, and the variance is $2(v + 2\delta)$.

Examples
```
[m,v] = ncx2stat(4,2)
m =
    6
v =
   16
```

References   [1] Evans, M., N. Hastings, and B. Peacock. *Statistical Distributions*. 2nd ed., Hoboken, NJ: John Wiley & Sons, Inc., 1993, pp. 50–52.

[2] Johnson, N., and S. Kotz. *Distributions in Statistics: Continuous Univariate Distributions-2*. Hoboken, NJ: John Wiley & Sons, Inc., 1970, pp. 130–148.

See Also   `ncx2pdf`, `ncx2cdf`, `ncx2inv`, `ncx2rnd`

Purpose   Number of dimensions of categorical array

Syntax   n = ndims(A)

Description   n = ndims(A) returns the number of dimensions in the categorical
array A. The number of dimensions in an array is always greater than
or equal to 2. Trailing singleton dimensions are ignored. Put simply,
ndims(A) is length(size(A)).

See Also   size

gmdistribution.NDimensions property

Purpose   Dimension d of multivariate Gaussian distributions

Description   The dimension d of the multivariate Gaussian distributions.

Purpose   Number of dimensions of dataset array

Syntax   n = ndims(A)

Description   n = ndims(A) returns the number of dimensions in the dataset A. The number of dimensions in an array is always 2.

See Also   size

qrandset.ndims

Purpose   Number of dimensions in matrix

Syntax   n = ndims(p)

Description n = ndims(p) returns the number of dimensions in the matrix that
                 is created by the syntax p(:,:). Since this is always a 2-D matrix, n
                 is always equal to 2.

See Also   qrandset, size

Purpose   Number of dimensions

Description   The `NDims` property specifies the number of dimensions, which is equal to the number of features in the training data used to create the Naive Bayes classifier.

qrandstream.ne

Purpose   Not equal relation for handles

Syntax   h1 ~= h2

Description Handles are equal if they are handles for the same object and are unequal otherwise.

h1 ~= h2 performs element-wise comparisons between handle arrays h1 and h2. h1 and h2 must be of the same dimensions unless one is a scalar. The result is a logical array of the same dimensions, where each element is an element-wise ~= result.

If one of h1 or h2 is scalar, scalar expansion is performed and the result will match the dimensions of the array that is not scalar.

tf = ne(h1, h2) stores the result in a logical array of the same dimensions.

See Also   qrandstream, eq, ge, gt, le, lt

Purpose   Generate quasi-random point set

Syntax   X = net(p,n)

Description   X = net(p,n) returns the first n points X from the point set p of the qrandset class. X is n-by-*d*, where *d* is the dimension of the point set.

Objects p of the @qrandset class encapsulate properties of a specified quasi-random sequence. Values of the point set are not generated and stored in memory until p is accessed using net or parenthesis indexing.

Examples   Use haltonset to generate a 3-D Halton point set, skip the first 1000 values, and then retain every 101st point:

```
p = haltonset(3,'Skip',1e3,'Leap',1e2)
p =
    Halton point set in 3 dimensions (8.918019e+013 points)
    Properties:
                Skip : 1000
                Leap : 100
    ScrambleMethod : none
```

Use scramble to apply reverse-radix scrambling:

```
p = scramble(p,'RR2')
p =
    Halton point set in 3 dimensions (8.918019e+013 points)
    Properties:
                Skip : 1000
                Leap : 100
    ScrambleMethod : RR2
```

Use net to generate the first four points:

```
X0 = net(p,4)
X0 =
    0.0928    0.6950    0.0029
    0.6958    0.2958    0.8269
```

```
    0.3013    0.6497    0.4141
    0.9087    0.7883    0.2166
```

Use parenthesis indexing to generate every third point, up to the 11th point:

```
X = p(1:3:11,:)
X =
    0.0928    0.6950    0.0029
    0.9087    0.7883    0.2166
    0.3843    0.9840    0.9878
    0.6831    0.7357    0.7923
```

See Also   haltonset, sobolset, qrandstream

Purpose   Nonlinear regression

Syntax
```
beta = nlinfit(X,y,fun,beta0)
[beta,r,J,COVB,mse] = nlinfit(X,y,fun,beta0)
[...] = nlinfit(X,y,fun,beta0,options)
```

Description   `beta = nlinfit(X,y,fun,beta0)` returns a vector `beta` of coefficient estimates for a nonlinear regression of the responses in `y` on the predictors in `X` using the model specified by `fun`. `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. `fun` is a function handle to a function of the form:

```
yhat = modelfun(b,X)
```

where `b` is a coefficient vector. `beta0` is a vector containing initial values for the coefficients. `beta` is the same length as `beta0`.

`[beta,r,J,COVB,mse] = nlinfit(X,y,fun,beta0)` returns the fitted coefficients `b`, the residuals `r`, the Jacobian `J` of `fun`, the estimated covariance matrix `COVB` 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 `COVB` and may need `mse` as input to `nlpredci` or `nlparci` to insure that the confidence intervals take the robust fit properly into account.

`[...] = 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 b0. 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'` — A weight function for robust fitting. Valid only when `'Robust'` is `'on'`. It can be `'bisquare'` (the default), `'andrews'`, `'cauchy'`, `'fair'`, `'huber'`, `'logistic'`, `'talwar'`, or `'welsch'`. 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 in robust fitting to normalize the residuals before applying the weight function. The value is a positive scalar, with the default value dependent on the weight function. This parameter is required if the weight function is specified as a function handle.

Note Robust nonlinear fitting uses an algorithm that iteratively reweights response values and recomputes a least-squares fit. The least-squares component of the algorithm differs from linear least squares, but the reweighting loop is identical to that for robust linear methods. In particular, the weight function and its tuning parameter are the same as described for robustfit. Note that nlinfit counts iterations of both the reweighting loop and the least-squares fit toward the maximum number of iterations.

---

nlinfit treats NaNs in y or modelfun(beta0,X) as missing data and ignores the corresponding rows.

nlintool is a graphical user interface to nlinfit.

Examples
The data in reaction.mat are partial pressures of three chemical reactants and the corresponding reaction rates. The function hougen implements the nonlinear Hougen-Watson model for reaction rates. The following fits the model to the data:

```
load reaction

beta = nlinfit(reactants,rate,@hougen,beta)
beta =
    1.2526
    0.0628
    0.0400
    0.1124
    1.1914
```

References
[1] Seber, G. A. F., and C. J. Wild. *Nonlinear Regression*. Hoboken, NJ: Wiley-Interscience, 2003.

See Also
nlparci, nlpredci, nlintool

nlintool

Purpose	Interactive nonlinear regression

Syntax

```
nlintool(X,y,fun,beta0)
nlintool(X,y,fun,beta0,alpha)
nlintool(X,y,fun,beta0,alpha,'xname','yname')
```

Description nlintool(X,y,fun,beta0) is a graphical user interface to the nlinfit function, and uses the same input arguments. The interface displays plots of the fitted response against each predictor, with the other predictors held fixed. 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 plots update to display the model at the new point in predictor space. Dashed red curves show 95% global confidence intervals for predictions.

nlintool(X,y,fun,beta0,alpha) shows 100(1-alpha)% confidence intervals for predictions.

nlintool(X,y,fun,beta0,alpha,'xname','yname') labels the plots using the string matrix 'xname' for the predictors and the string 'yname' for the response.

Examples The data in reaction.mat are partial pressures of three chemical reactants and the corresponding reaction rates. The function hougen implements the nonlinear Hougen-Watson model for reaction rates. The following fits the model to the data:

```
load reaction
nlintool(reactants,rate,@hougen,beta,0.01,xn,yn)
```
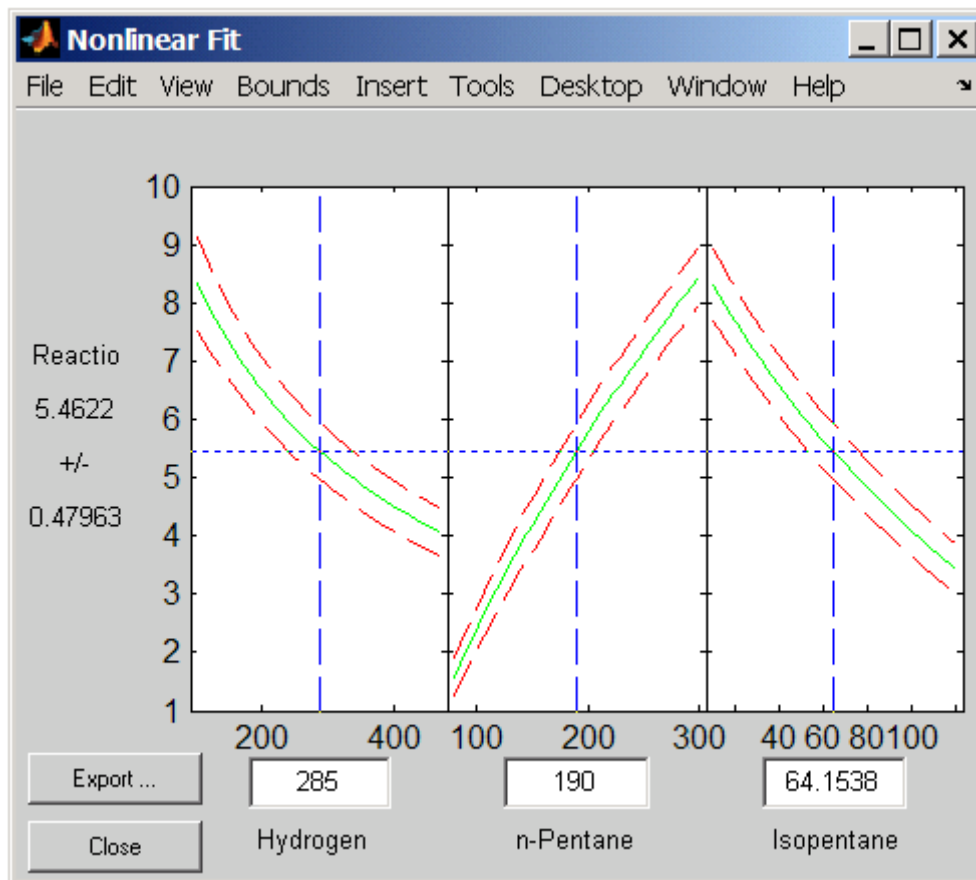
See Also   nlinfit, polytool, rstool

nlmefit

Purpose   Nonlinear mixed-effects estimation

Syntax
```
beta = nlmefit(X,y,group,V,fun,beta0)
[beta,PSI] = nlmefit(...)
[beta,PSI,stats] = nlmefit(...)
[beta,PSI,stats,B] = nlmefit(...)
[...] = nlmefit(...,param1,val1,param2,val2,...)
```

Description   `beta = nlmefit(X,y,group,V,fun,beta0)` fits a nonlinear mixed-effects regression model and returns estimates of the fixed effects in `beta`. By default, `nlmefit` fits a model in which each parameter is the sum of a fixed and a random effect, and the random effects are uncorrelated (their covariance matrix is diagonal).

X is an n-by-h matrix of n observations on h predictors. y is an n-by-1 vector of responses.

`group` is a grouping variable indicating m groups in the observations. `group` is a categorical variable, a numeric vector, a character matrix with rows for group names, or a cell array of strings.

V is an m-by-g matrix or cell array of g group-specific predictors. These are predictors that take the same value for all observations in a group. The rows of V are assigned to groups using `grp2idx`, according to the order specified by `grp2idx(group)`. Use a cell array for V if group predictors vary in size across groups. Use `[]` for V if there are no group-specific predictors.

`fun` is a function handle to a function that accepts predictor values and model parameters and returns fitted values. `fun` has the form

```
yfit = modelfun(PHI,XFUN,VFUN)
```

The arguments are:

- PHI — A 1-by-p vector of model parameters.

- XFUN — A k-by-h array of predictors, where:

  - $k = 1$ if XFUN is a single row of X.

- $k = n_i$ if XFUN contains the rows of X for a single group of size n_i.

- $k = n$ if XFUN contains all rows of X.

- VFUN — Group-specific predictors given by one of:

  - A 1-by-g vector corresponding to a single group and a single row of V.

  - An n-by-g array, where the jth row is V(I,:) if the jth observation is in group I.

  If V is empty, nlmefit calls modelfun with only two inputs.

- yfit — A k-by-1 vector of fitted values

When either PHI or VFUN contains a single row, it corresponds to all rows in the other two input arguments.

Note If modelfun can compute yfit for more than one vector of model parameters per call, use the 'Vectorization' parameter (described later) for improved performance.

---

beta0 is a q-by-1 vector with initial estimates for q fixed effects. By default, q is the number of model parameters p.

nlmefit fits the model by maximizing an approximation to the marginal likelihood with random effects integrated out, assuming that:

- Random effects are multivariate normally distributed and independent between groups.

- Observation errors are independent, identically normally distributed, and independent of the random effects.

[beta,PSI] = nlmefit(...) also returns PSI, an r-by-r estimated covariance matrix for the random effects. By default, r is equal to the number of model parameters p.

[beta,PSI,stats] = nlmefit(...) also returns stats, a structure with fields:

- logl — The maximized log-likelihood for the fitted model

- mse — The estimated error variance for the fitted model

- aic — The Akaike information criterion for the fitted model

- bic — The Bayesian information criterion for the fitted model

- sebeta — The standard errors for beta

- dfe — The error degrees of freedom for the model

[beta,PSI,stats,B] = nlmefit(...) also returns B, an *r*-by-*m* matrix of estimated random effects for the *m* groups. By default, *r* is equal to the number of model parameters *p*.

[...] = nlmefit(...,*param1*,*val1*,*param2*,*val2*,...) specifies additional parameter name/value pairs.

Use the following parameters to fit a model different from the default. (The default model is obtained by setting both 'FEConstDesign' and 'REConstDesign' to eye(p), or by setting both 'FEParamsSelect' and 'REParamsSelect' to 1:p.) Use at most one parameter with an 'FE' prefix and one parameter with an 'RE' prefix. The nlmefit function requires you to specify at least one fixed effect and one random effect.

Parameter	Value
`'FEParamsSelect'`	A vector specifying which elements of the parameter vector PHI include a fixed effect, given as a numeric vector of indices between 1 and p or as a 1-by-p logical vector. If q is the specified number of elements, then the model includes q fixed effects.
`'FEConstDesign'`	A p-by-q design matrix ADESIGN, where ADESIGN*beta are the fixed components of the p elements of PHI.
`'FEGroupDesign'`	A p-by-q-by-m array specifying a different p-by-q fixed-effects design matrix for each of the m groups.
`'FEObsDesign'`	A p-by-q-by-n array specifying a different p-by-q fixed-effects design matrix for each of the n observations.
`'REParamsSelect'`	A vector specifying which elements of the parameter vector PHI include a random effect, given as a numeric vector of indices between 1 and p or as a 1-by-p logical vector. The model includes r random effects, where r is the specified number of elements.
`'REConstDesign'`	A p-by-r design matrix BDESIGN, where BDESIGN*B are the random components of the p elements of PHI.
`'REGroupDesign'`	A p-by-r-by-m array specifying a different p-by-r random-effects design matrix for each of m groups.
`'REObsDesign'`	A p-by-r-by-n array specifying a different p-by-r random-effects design matrix for each of n observations.

Use the following parameters to control the iterative algorithm for maximizing the likelihood:

Parameter	Value
`'RefineBeta0'`	Determines whether `nlmefit` makes an initial refinement of `beta0` by first fitting `modelfun` without random effects and replacing `beta0` with `beta`. Choices are `'on'` and `'off'`. The default value is `'on'`.
`'ApproximationType'`	The method used to approximate the likelihood of the model. Choices are: • `'LME'` — Use the likelihood for the linear mixed-effects model at the current conditional estimates of `beta` and `B`. This is the default. • `'RELME'` — Use the restricted likelihood for the linear mixed-effects model at the current conditional estimates of `beta` and `B`. • `'FO'` — First-order Laplacian approximation without random effects. • `'FOCE'` — First-order Laplacian approximation at the conditional estimates of `B`.

Parameter	Value
`'Vectorization'`	Indicates acceptable sizes for the `PHI`, `XFUN`, and `VFUN` input arguments to `modelfun`. Choices are: • `'SinglePhi'` — `modelfun` can only accept a single set of model parameters at a time, so `PHI` must be a single row vector in each call. `nlmefit` calls `modelfun` in a loop, if necessary, with a single `PHI` vector and with `XFUN` containing rows for a single observation or group at a time. `VFUN` may be a single row that applies to all rows of `XFUN`, or a matrix with rows corresponding to rows in `XFUN`. This is the default. • `'SingleGroup'` — `modelfun` can only accept inputs corresponding to a single group in the data, so `XFUN` must contain rows of `X` from a single group in each call. Depending on the model, `PHI` is a single row that applies to the entire group or a matrix with one row for each observation. `VFUN` is a single row. • `'Full'` — `modelfun` can accept inputs for multiple parameter vectors and multiple groups in the data. Either `PHI` or `VFUN` may be a single row that applies to all rows of `XFUN` or a matrix with rows corresponding to rows in `XFUN`. This option can improve performance by reducing the number of calls to `modelfun`, but may require `modelfun` to perform singleton expansion on `PHI` or `V`.

Parameter	Value
`'CovParameterization'`	Specifies the parameterization used internally for the scaled covariance matrix. Choices are `'chol'` for the Cholesky factorization or `'logm'` the matrix logarithm. The default is `'logm'`.
`'CovPattern'`	Specifies an *r*-by-*r* logical or numeric matrix P that defines the pattern of the random-effects covariance matrix PSI. `nlmefit` estimates the variances along the diagonal of PSI and the covariances specified by nonzeroes in the off-diagonal elements of P. Covariances corresponding to zero off-diagonal elements in P are constrained to be zero. If P does not specify a row-column permutation of a block diagonal matrix, `nlmefit` adds nonzero elements to P as needed. The default value of P is `eye(r)`, corresponding to uncorrelated random effects.

Alternatively, P may be a 1-by-*r* vector containing values in `1:r`, with equal values specifying groups of random effects. In this case, `nlmefit` estimates covariances only within groups, and constrains covariances across groups to be zero. |

Parameter	Value
'Options'	A structure of the form returned by statset. nlmefit uses the following statset parameters: • 'TolX' — Termination tolerance on the estimated fixed and random effects. The default is 1e-4. • 'TolFun' — Termination tolerance on the log-likelihood function. The default is 1e-4. • 'MaxIter' — Maximum number of iterations allowed. The default is 200. • 'Display' — Level of iterative display during estimation. Choices are: ▪ 'off' — Displays no information. This is the default. ▪ 'iter' — Displays iterative output to the command window. ▪ 'final' — Displays the final output. • 'FunValCheck' — Check for invalid values, such as NaN or Inf, from modelfun. Choices are 'on' and 'off'. The default is 'on'.
'OptimFun'	Specifies the optimization function used in maximizing the likelihood. Choices are 'fminsearch' to use fminsearch or 'fminunc' to use fminunc. The default is 'fminsearch'. You can only specify 'fminunc' if Optimization Toolbox software is installed.

nlmefit

Examples   Display data on the growth of five orange trees:

```
CIRC = [30 58 87 115 120 142 145;
        33 69 111 156 172 203 203;
        30 51 75 108 115 139 140;
        32 62 112 167 179 209 214;
        30 49 81 125 142 174 177];
time = [118 484 664 1004 1231 1372 1582];

h = plot(time,CIRC','o','LineWidth',2);
xlabel('Time (days)')
ylabel('Circumference (mm)')
title('{\bf Orange Tree Growth}')
legend([repmat('Tree ',5,1),num2str((1:5)')],...
        'Location','NW')
grid on
hold on
```

Orange Tree Growth
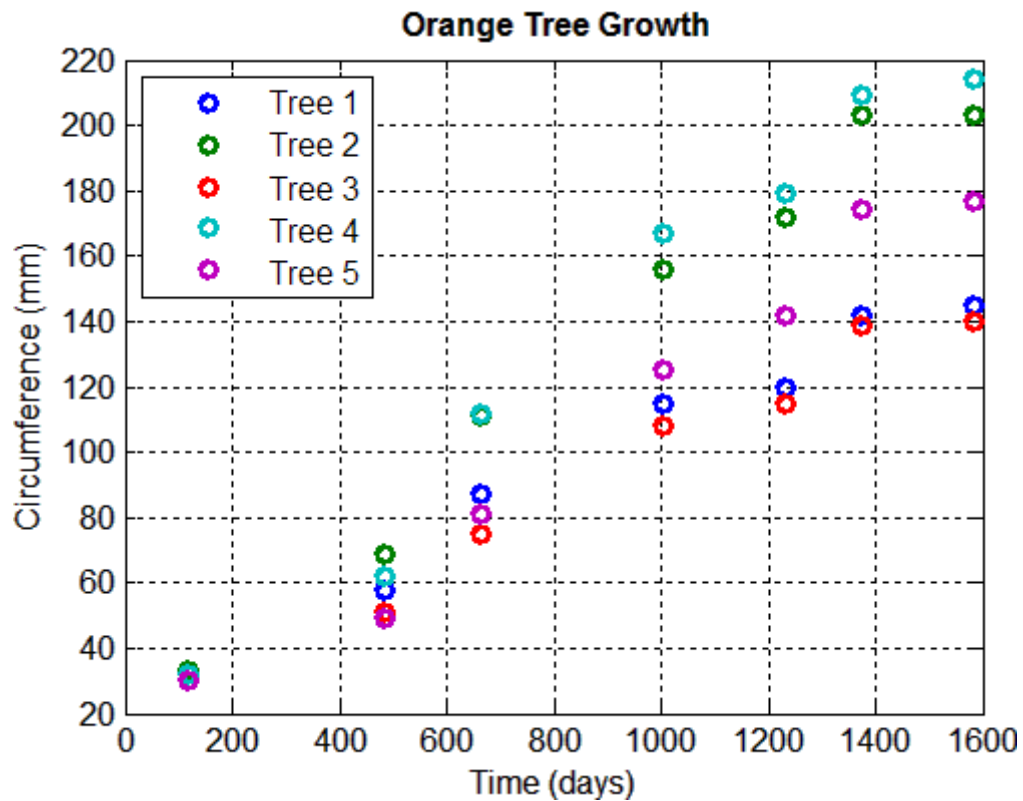
Use an anonymous function to specify a logistic growth model:

```
model = @(PHI,t)(PHI(:,1))./(1+exp(-(t-PHI(:,2))./PHI(:,3)));
```

Fit the model using `nlmefit` with default settings (that is, assuming each parameter is the sum of a fixed and a random effect, with no correlation among the random effects):

```
TIME = repmat(time,5,1);
NUMS = repmat((1:5)',size(time));

beta0 = [100 100 100];
```

```
[beta1,PSI1,stats1] = nlmefit(TIME(:),CIRC(:),NUMS(:),...
                              [],model,beta0)
beta1 =
  191.3189
  723.7609
  346.2518
PSI1 =
  962.1533         0         0
        0    0.0000         0
        0         0  297.9931
stats1 =
      logl: -131.5457
       mse: 59.7881
       aic: 277.0913
       bic: 287.9788
    sebeta: NaN
       dfe: 28
```

The negligible variance of the second random effect, PSI1(2,2), suggests that it can be removed to simplify the model:

```
[beta2,PSI2,stats2,b2] = nlmefit(TIME(:),CIRC(:),...
NUMS(:),[],model,beta0,'REParamsSelect',[1 3])
beta2 =
  191.3190
  723.7610
  346.2527
PSI2 =
  962.0491         0
        0  298.1869
stats2 =
      logl: -131.5457
       mse: 59.7881
       aic: 275.0913
       bic: 284.4234
    sebeta: NaN
       dfe: 29
```

```
b2 =
  -28.5254   31.6061   -36.5071   39.0738   -5.6475
   10.0034   -0.7633     6.0080   -9.4630   -5.7853
```

The log-likelihood logl is unaffected, and both the Akaike and Bayesian information criteria (aic and bic) are reduced, supporting the decision to drop the second random effect from the model.

Use the estimated fixed effects in beta2 and the estimated random effects for each tree in b2 to plot the model through the data:

```
PHI = repmat(beta2,1,5) + ...              % Fixed effects
      [b2(1,:);zeros(1,5);b2(2,:)];        % Random effects

colors = get(h,'Color');
tplot = 0:0.1:1600;
for I = 1:5
    fitted_model = @(t)(PHI(1,I))./(1+exp(-(t-PHI(2,I))./PHI(3,I)))
    plot(tplot,fitted_model(tplot),'Color',colors{I},...
    'LineWidth',2)
end
```

Orange Tree Growth
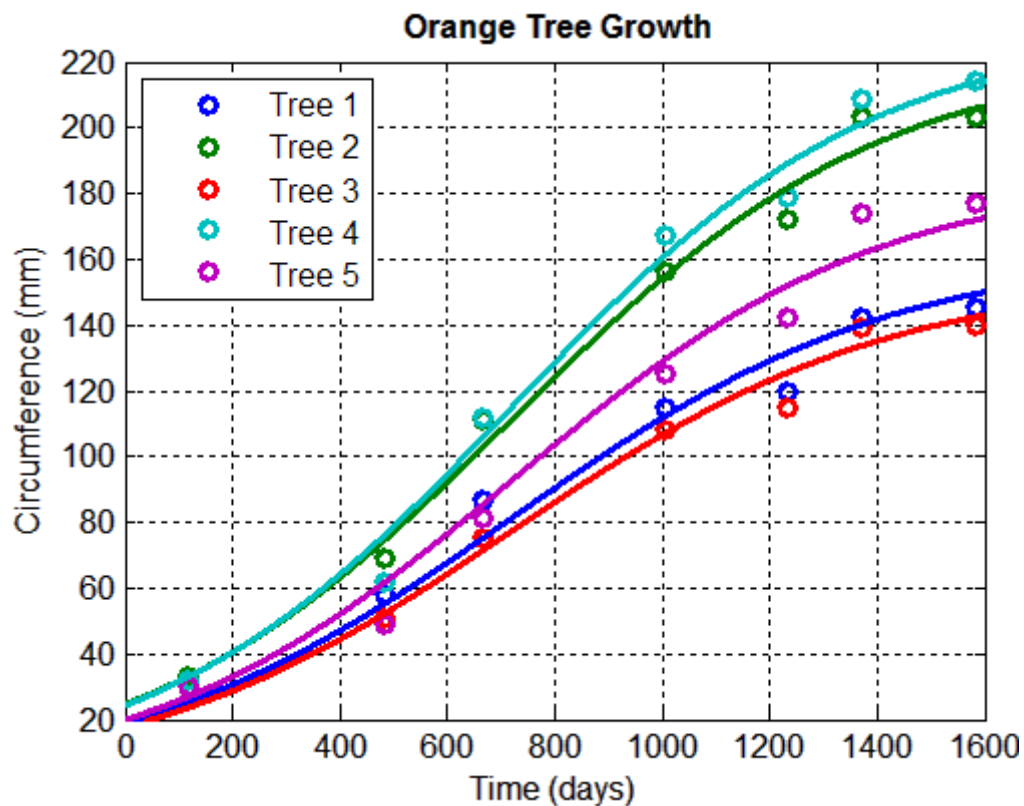
References

[1] Lindstrom, M. J., and D. M. Bates. "Nonlinear mixed-effects models for repeated measures data." *Biometrics*. Vol. 46, 1990, pp. 673–687.

[2] Davidian, M., and D. M. Giltinan. *Nonlinear Models for Repeated Measurements Data*. New York: Chapman & Hall, 1995.

[3] Pinheiro, J. C., and D. M. Bates. "Approximations to the log-likelihood function in the nonlinear mixed-effects model." *Journal of Computational and Graphical Statistics*. Vol. 4, 1995, pp. 12–35.

[4] Demidenko, E. *Mixed Models: Theory and Applications.* Hoboken, NJ: John Wiley & Sons, Inc., 2004.

See Also   "Grouped Data" on page 2-33

nlinfit, nlpredci

gmdistribution.NlogL property

Purpose   Negative of log-likelihood

Description   The negative of the log-likelihood of the data.

Note This property applies only to gmdistribution objects constructed with `fit`.

Purpose   Read-only value specifying negative log likelihood for input data to ProbDistParametric object

Description   NLogL is a read-only property of the `ProbDistParametric` class. NLogL is a value specifying the negative log likelihood for input data used to fit a distribution represented by a ProbDistParametric object.

Values   The value is a numeric scalar for a distribution fit to input data, that is, a distribution created using the `fitdist` function. This property is empty for distributions created without fitting to data, that is, by using the `ProbDistUnivParam.ProbDistUnivParam` constructor. Use this information to view and compare the negative log likelihood for input data supplied to create distributions.

ProbDistUnivKernel.NLogL property

Purpose   Read-only value specifying negative log likelihood for input data to
                ProbDistUnivKernel object

Description NLogL is a read-only property of the `ProbDistUnivKernel` class. `NLogL`
                is a value specifying the negative log likelihood for input data used to fit
                a distribution represented by a ProbDistUnivKernel object.

Values   The value is a numeric scalar for a distribution fit to input data, that is,
                a distribution created using the `fitdist` function. Use this information
                to view and compare the negative log likelihood for input data used to
                create distributions.

| **Purpose** | Nonlinear regression parameter confidence intervals |

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 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. When `J` is ill-conditioned, confidence intervals may be inaccurate.

Examples

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, nlpredci

Purpose   Nonlinear regression prediction confidence intervals

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.

Parameter	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.

Parameter	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.

Examples 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,Sigma] = nlinfit(reactants,rate,@hougen,...
beta);

newX = reactants(1:2,:);
[ypred, delta] = nlpredci(@hougen,newX,beta,resid,...
'Covar',Sigma);
ypred =
    8.4179
    3.9542
delta =
    0.2805
```

         0.2474

See Also   nlinfit, nlparci

Purpose	Nonnegative matrix factorization

Syntax	`[W,H] = nnmf(A,k)`
	`[W,H] = nnmf(A,k,`*param1*`,`*val1*`,`*param2*`,`*val2*`,...)`
	`[W,H,D] = nnmf(...)`

Description   `[W,H] = nnmf(A,k)` factors the nonnegative *n*-by-*m* matrix A into nonnegative factors W (*n*-by-k) and H (k-by-*m*). The factorization is not exact; W*H is a lower-rank approximation to A. The factors W and H are chosen to minimize the root-mean-squared residual D between A and W*H:

```
D = sqrt(norm(A-W*H,'fro')/(N*M))
```

The factorization uses an iterative method starting with random initial values for W and H. Because the root-mean-squared residual D may have local minima, repeated factorizations may yield different W and H. Sometimes the algorithm converges to a solution of lower rank than *k*, which may indicate that the result is not optimal.

W and H are normalized so that the rows of H have unit length. The columns of W are ordered by decreasing length.

`[W,H] = nnmf(A,k,`*param1*`,`*val1*`,`*param2*`,`*val2*`,...)` specifies optional parameter name/value pairs from the following table.

Parameter	Value
`'algorithm'`	Either `'als'` (the default) to use an alternating least-squares algorithm, or `'mult'` to use a multiplicative update algorithm.
	In general, the `'als'` algorithm converges faster and more consistently. The `'mult'` algorithm is more sensitive to initial values, which makes it a good choice when using `'replicates'` to find W and H from multiple random starting values.

Parameter	Value
'w0'	An *n*-by-k matrix to be used as the initial value for W.
'h0'	A k-by-*m* matrix to be used as the initial value for H.
'options'	An options structure as created by the statset function. nnmf uses the following fields of the options structure: Display, TolX, TolFun, and MaxIter. Unlike in optimization settings, reaching MaxIter iterations is treated as convergence.
'replicates'	The number of times to repeat the factorization, using new random starting values for W and H, except at the first replication if 'w0' and 'h0' are given. This is most beneficial with the 'mult' algorithm. The default is 1.

[W,H,D] = nnmf(...) also returns D, the root mean square residual.

Examples

Example 1

Compute a nonnegative rank-two approximation of the measurements of the four variables in Fisher's iris data:

```
load fisheriris
[W,H] = nnmf(meas,2);
H
H =
    0.6852    0.2719    0.6357    0.2288
    0.8011    0.5740    0.1694    0.0087
```

The first and third variables in meas (sepal length and petal length, with coefficients 0.6852 and 0.6357, respectively) provide relatively strong weights to the first column of W. The first and second variables in meas (sepal length and sepal width, with coefficients 0.8011and 0.5740) provide relatively strong weights to the second column of W.

Create a `biplot` of the data and the variables in `meas` in the column space of `W`:

```
biplot(H','scores',W,'varlabels',{'sl','sw','pl','pw'});
axis([O 1.1 O 1.1])
xlabel('Column 1')
ylabel('Column 2')
```
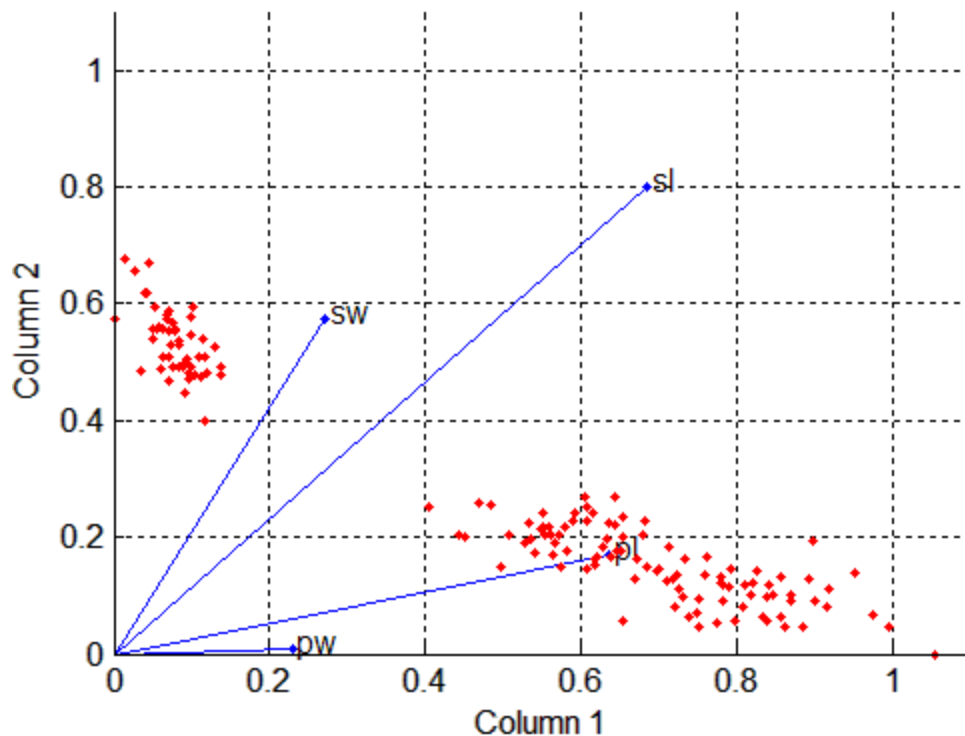

Example 2

Starting from a random array X with rank 20, try a few iterations at several replicates using the multiplicative algorithm:

```
X = rand(100,20)*rand(20,50);
opt = statset('MaxIter',5,'Display','final');
```

```
[WO,HO] = nnmf(X,5,'replicates',10,...
                 'options',opt,...
                 'algorithm','mult');
    rep iteration    rms resid    |delta x|
     1       5       0.560887     0.0245182
     2       5        0.66418     0.0364471
     3       5       0.609125     0.0358355
     4       5       0.608894     0.0415491
     5       5       0.619291     0.0455135
     6       5       0.621549     0.0299965
     7       5       0.640549     0.0438758
     8       5       0.673015     0.0366856
     9       5       0.606835     0.0318931
    10       5       0.633526     0.0319591
Final root mean square residual = 0.560887
```

Continue with more iterations from the best of these results using alternating least squares:

```
opt = statset('Maxiter',1000,'Display','final');
[W,H] = nnmf(X,5,'wO',WO,'hO',HO,...
                 'options',opt,...
                 'algorithm','als');
    rep iteration    rms resid    |delta x|
     1      80       0.256914 9.78625e-005
Final root mean square residual = 0.256914
```

References   [1] Berry, M. W., et al. "Algorithms and Applications for Approximate Nonnegative Matrix Factorization." *Computational Statistics and Data Analysis*. Vol. 52, No. 1, 2007, pp. 155–173.

See Also   princomp, factoran, statset

classregtree.nodeerr

Purpose   Return vector of node errors

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.

Examples   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)
```
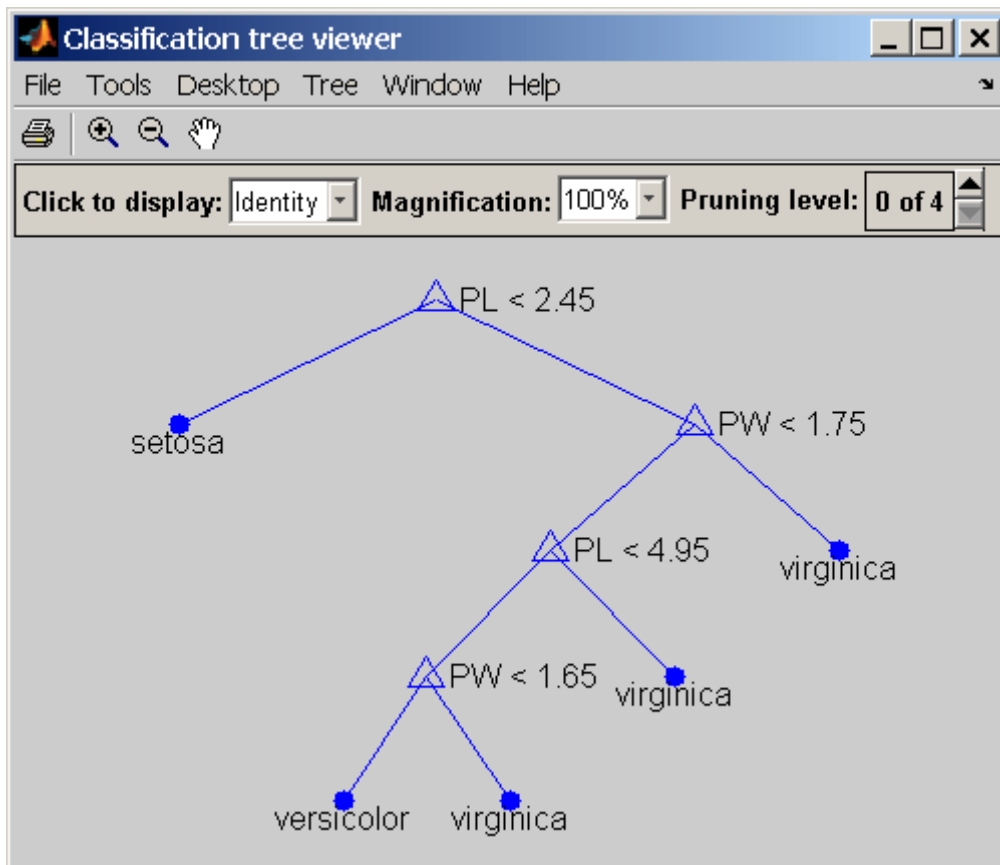


```
e = nodeerr(t)
e =
    0.6667
         0
    0.5000
    0.0926
    0.0217
```

```
                      0.0208
                      0.3333
                           0
                           0
```

References   [1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

See Also   classregtree, numnodes, test

Purpose   Node probabilities

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.

Examples   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
```

References   [1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

See Also   classregtree, nodesize, numnodes

classregtree.nodesize

Purpose	Return node size
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.

Examples 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)
```
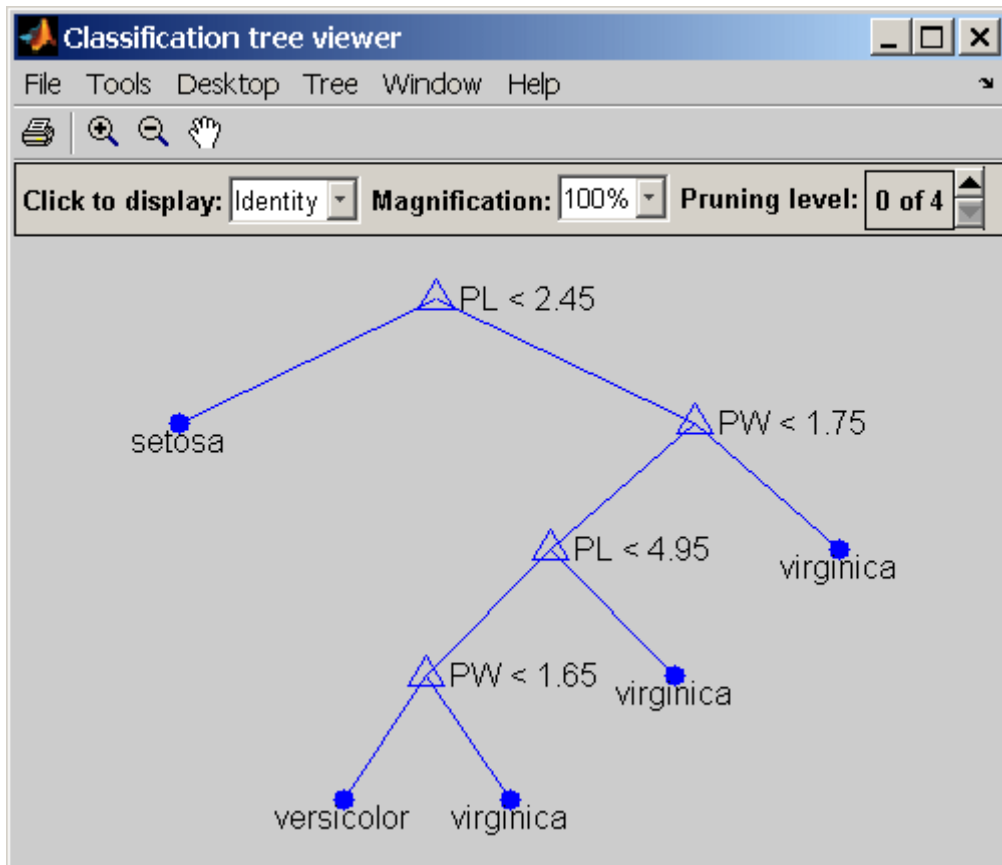
```
sizes = nodesize(t)
sizes =
   150
    50
   100
    54
    46
    48
     6
```

classregtree.nodesize

```
    47
     1
```

References   [1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

See Also   classregtree, numnodes

Purpose   Notify listeners of event

Syntax   `notify(h,'eventname')`
`notify(h,'eventname',data)`

Description   `notify(h,'eventname')` notifies listeners added to the event named `eventname` on handle object array `h` that the event is taking place. `h` is the array of handles to objects triggering the event, and `eventname` must be a string.

`notify(h,'eventname',data)` provides a way of encapsulating information about an event which can then be accessed by each registered listener. `data` must belong to the `event.eventdata` class.

See Also   `addlistener`, `event.EventData`, `events`, `qrandstream`

nominal class

Superclasses	categorical
Purpose	Arrays for nominal categorical data

Construction	nominal	Construct nominal categorical array

Methods **Inherited Methods**

Methods in the following table are inherited from `categorical`.

addlevels	Add levels to categorical array
cat	Concatenate categorical arrays
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
disp	Display categorical array
display	Display categorical array
double	Convert categorical array to double array
droplevels	Drop levels
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
getlabels	Access categorical array labels
getlevels	Get categorical array levels
horzcat	Horizontal concatenation for categorical arrays
int16	Convert categorical array to signed 16-bit integer array
int32	Convert categorical array to signed 32-bit integer array
int64	Convert categorical array to signed 64-bit integer array
int8	Convert categorical array to signed 8-bit integer array
intersect	Set intersection for categorical arrays
ipermute	Inverse permute dimensions of categorical array
isempty	True for empty categorical array
isequal	True if categorical arrays are equal
islevel	Test for levels
ismember	True for elements of categorical array in set
isscalar	True if categorical array is scalar
isundefined	Test for undefined elements
isvector	True if categorical array is vector

length	Length of categorical array
levelcounts	Element counts by level
ndims	Number of dimensions of categorical array
numel	Number of elements in categorical array
permute	Permute dimensions of categorical array
reorderlevels	Reorder levels
repmat	Replicate and tile categorical array
reshape	Resize categorical array
rot90	Rotate categorical matrix 90 degrees
setdiff	Set difference for categorical arrays
setlabels	Label levels
setxor	Set exclusive-or for categorical arrays
shiftdim	Shift dimensions of categorical array
single	Convert categorical array to single array
size	Size of categorical array
squeeze	Squeeze singleton dimensions from categorical array
subsasgn	Subscripted assignment for categorical array

subsindex	Subscript index for categorical array
subsref	Subscripted reference for categorical array
summary	Summary statistics for categorical array
times	Product of categorical arrays
transpose	Transpose categorical matrix
uint16	Convert categorical array to unsigned 16-bit integers
uint32	Convert categorical array to unsigned 32-bit integers
uint64	Convert categorical array to unsigned 64-bit integers
uint8	Convert categorical array to unsigned 8-bit integers
union	Set union for categorical arrays
unique	Unique values in categorical array
vertcat	Vertical concatenation for categorical arrays

Properties

Inherited Properties

Properties in the following table are inherited from `categorical`.

labels	Text labels for levels
undeflabel	Text label for undefined levels

nominal class

Copy Semantics   Value. To learn how this affects your use of the class, see Comparing Handle and Value Classes in the MATLAB Object-Oriented Programming documentation.

See Also   histc, ordinal

Purpose   Construct nominal categorical 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`.

# nominal

**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   `histc`, `ordinal`

normcdf

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 ($-\infty$ 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, normpdf, norminv, normstat, normfit, normlike, normrnd

normfit

Purpose	Normal parameter estimates

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.

Examples   In this example the data is a two-column random normal matrix. Both
columns have μ = 10 and σ = 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, normlike, normpdf, normcdf, norminv, normstat, normrnd

norminv

Purpose   Normal inverse 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 Pth 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, normcdf, normpdf, normstat, normfit, normlike, normrnd

normlike

Purpose	Normal negative log-likelihood

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	`normfit`, `normpdf`, `normcdf`, `norminv`, `normstat`, `normrnd`

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
deviationsigma. 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, normcdf, norminv, normstat, normfit, normlike, normrnd

normplot

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)
```

N

See Also cdfplot, wblplot, probplot, hist, normfit, norminv, normpdf, normspec, normstat, normcdf, normrnd, normlike

normrnd

Purpose Normal random numbers

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.

Examples
```
n1 = normrnd(1:6,1./(1:6))
n1 =
  2.1650  2.3134  3.0250  4.0879  4.8607  6.2827
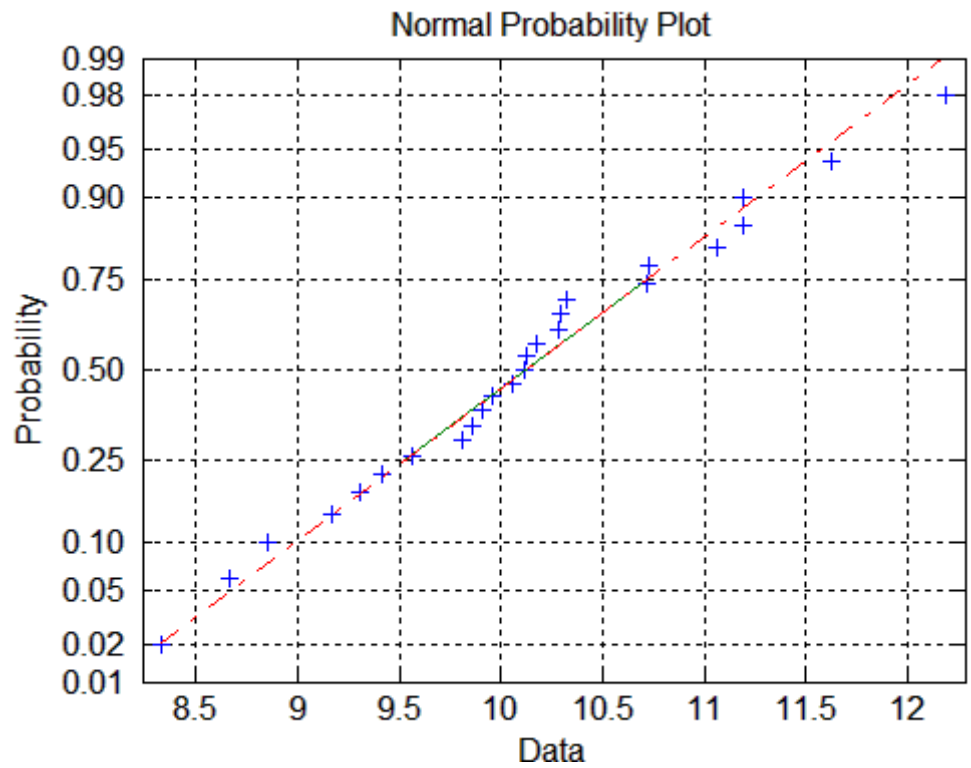
n2 = normrnd(0,1,[1 5])
n2 =
  0.0591  1.7971  0.2641  0.8717  -1.4462

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   random, normpdf, normcdf, norminv, normstat, normfit, normlike

normspec

Purpose	Normal density plot between specifications
Syntax	normspec(specs) normspec(specs,mu,sigma) normspec(specs,mu,sigma,*region*) p = normspec(...) [p,h] = normspec(...)

Description

normspec(specs) plots the standard normal density, shading the portion inside the specification limits given by the two-element vector specs. Set specs(1) to -Inf if there is no lower limit; set specs(2) to Inf if there is no upper limit.

normspec(specs,mu,sigma) shades the portion inside the specification limits of a normal density with parameters mu and sigma. The defaults are mu = 0 and sigma = 1.

normspec(specs,mu,sigma,*region*) shades the *region* either 'inside' or 'outside' the specification limits. The default is 'inside'.

p = normspec(...) also returns the probability, p, of the shaded area.

[p,h] = normspec(...) also returns a handle h to the line objects.

Examples

A production process fills cans of paint. The average amount of paint in any can is 1 gallon, but variability in the process produces a standard deviation of 2 ounces (2/128 gallons). What is the probability that cans will be filled under specification by 3 or more ounces?

```
p = normspec([1-3/128,Inf],1,2/128,'outside')
p =
    0.0668
```

See Also   capaplot, histfit

normstat

Purpose	Normal mean and variance
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
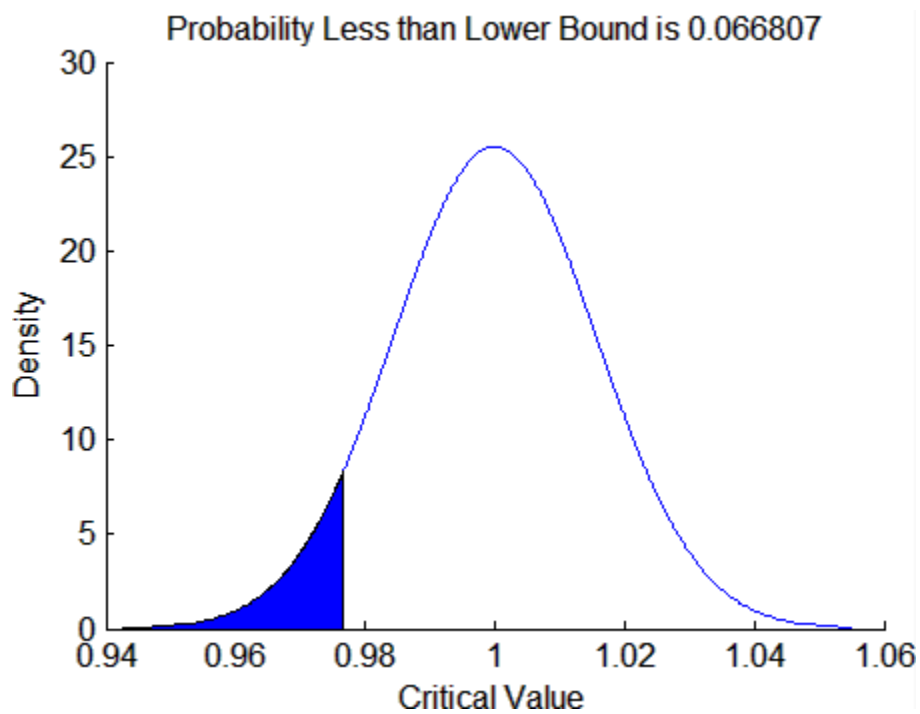
v =
     1    4    9   16   25
     4   16   36   64  100
     9   36   81  144  225
    16   64  144  256  400
    25  100  225  400  625
```

See Also   `normpdf, normcdf, norminv, normfit, normlike, normrnd`

piecewisedistribution.nsegments

Purpose   Number of segments

Syntax   n = nsegments(obj)

Description   n = nsegments(obj) returns the number of segments n in the
                   piecewise distribution object obj.

Examples   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

CompactTreeBagger.NTrees property

Purpose   Number of decision trees in ensemble

Description The NTrees property is a scalar equal to the number of decision trees in the ensemble.

See Also   Trees

Purpose   Number of decision trees in ensemble

Description The NTrees property is a scalar equal to the number of decision trees in the ensemble.

See Also   Trees

ProbDistParametric.NumParams property

Purpose   Read-only value specifying number of parameters of ProbDistParametric object

Description NumParams is a read-only property of the `ProbDistParametric` class. NumParams is a value specifying the number of parameters of a distribution represented by a ProbDistParametric object.

Values   This value is an integer that counts both the specified parameters and parameters that are fit to the data. Use this information to view and compare the number of parameters supplied to create distributions.

Purpose   Number of elements in dataset array

Syntax   n = numel(A)
                n = numel(A, varargin)

Description n = numel(A) returns 1. To find the number of elements, n, in the
                dataset array A, use prod(size(A)) or numel(A,':',':').

                n = numel(A, varargin) returns the number of subscripted elements,
                n, in A(index1, index2, ..., indexn), where varargin is a cell
                array whose elements are index1, index2, ... indexn.

See Also   length, size

categorical.numel

Purpose	Number of elements in categorical array
Syntax	`n = numel(A)` `n = numel(A, varargin)`
Description	`n = numel(A)` returns the number of elements in the categorical array A. `n = numel(A, varargin)` returns the number of subscripted elements, n, in `A(index1, index2, ..., indexN)`, where `varargin` is a cell array whose elements are `index1, index2, ... indexN`.
See Also	`size`

Purpose   Number of nodes

Syntax   n = numnodes(t)

Description n = numnodes(t) returns the number of nodes n in the tree t.

Examples   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
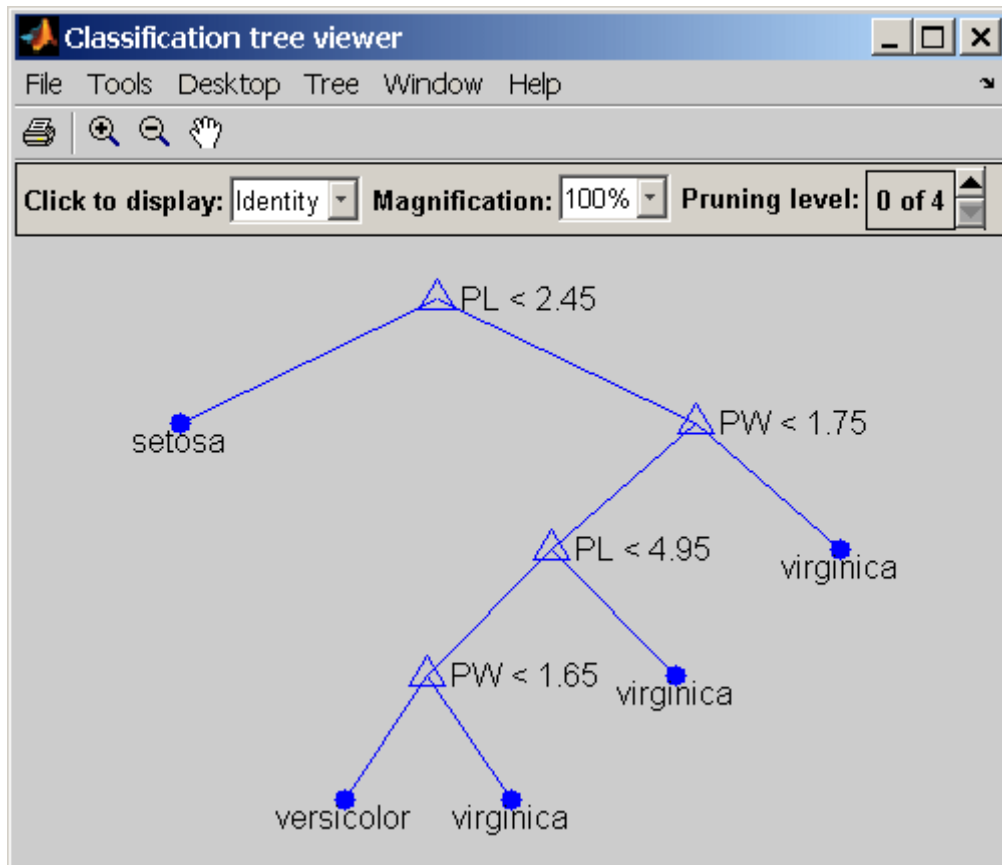
view(t)
```

```
n = numnodes(t)
n =
    9
```

References   [1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

See Also   classregtree

Purpose   Number of test sets

Description   Value is the number of folds in partitions of type `'kfold'` and
                   `'leaveout'`.

                   Value is 1 in partitions of type `'holdout'` and `'resubstitution'`.

TreeBagger.NVarToSample property

Purpose   Number of variables for random feature selection

Description   The `NVarToSample` property specifies the number of predictor or feature variables to select at random for each decision split. By default, it is set to the square root of the total number of variables for classification and one third of the total number of variables for regression.

See Also   `classregtree`

Purpose　　Cell array of nonempty, distinct strings giving names of observations in data set

Description　　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.

TreeBagger.oobError

Purpose	Out-of-bag error

Syntax

```
err = oobError(B)
err = oobError(B,'param1',val1,'param2',val2,...)
```

Description err = oobError(B) computes the misclassification probability (for classification trees) or mean squared error (for regression trees) for out-of-bag observations in the training data, using the trained bagger B. err is a vector of length NTrees, where NTrees is the number of trees in the ensemble.

err = oobError(B,'param1',val1,'param2',val2,...) specifies optional parameter name/value pairs:

'mode'	String indicating how oobError computes errors. If set to 'cumulative' (default), the method computes cumulative errors and err is a vector of length NTrees, where the first element gives error from trees(1), second element gives error from trees(1:2) etc, up to trees(1:NTrees). If set to 'individual', err is a vector of length NTrees, where each element is an error from each tree in the ensemble. If set to 'ensemble', err is a scalar showing the cumulative error for the entire ensemble.
'trees'	Vector of indices indicating what trees to include in this calculation. By default, this argument is set to 'all' and the method uses all trees. If 'trees' is a numeric vector, the method returns a vector of length NTrees for 'cumulative' and 'individual' modes, where NTrees is the number of elements in the input vector, and a scalar for 'ensemble' mode. For example, in the 'cumulative' mode, the

first element gives error from `trees(1)`, the second element gives error from `trees(1:2)` etc.

`'treeweights'`	Vector of tree weights. This vector must have the same length as the `'trees'` vector. `oobError` uses these weights to combine output from the specified trees by taking a weighted average instead of the simple nonweighted majority vote. You cannot use this argument in the `'individual'` mode.

See Also   `CompactTreeBagger.error`

TreeBagger.OOBIndices property

Purpose   Indicator matrix for out-of-bag observations

Description The `OOBIndices` property is a logical array of size `Nobs`-by-`NTrees`
where `Nobs` is the number of observations in the training data and
`NTrees` is the number of trees in the ensemble. The `(I,J)` element is
true if observation `I` is out-of-bag for tree `J` and false otherwise. In
other words, a true value means observation `I` was not selected for the
training data used to grow tree `J`.

See Also   `classregtree`

Purpose   Count of out-of-bag trees for each observation

Description   The `OOBInstanceWeight` property is a numeric array of size `Nobs`-by-1
containing the number of trees used for computing out-of-bag response
for each observation. `Nobs` is the number of observations in the training
data used to create the ensemble.

TreeBagger.oobMargin

Purpose	Out-of-bag margins

Syntax

```
mar = oobMargin(B)
mar = oobMargin(B,'param1',val1,'param2',val2,...)
```

Description

mar = oobMargin(B) computes an Nobs-by-NTrees matrix of classification margins for out-of-bag observations in the training data, using the trained bagger B.

mar = oobMargin(B,'param1',val1,'param2',val2,...) specifies optional parameter name/value pairs:

'mode'	String indicating how oobMargin computes errors. If set to 'cumulative' (default), the method computes cumulative margins and mar is an Nobs-by-NTrees matrix, where the first column gives margins from trees(1), second column gives margins from trees(1:2) etc, up to trees(1:NTrees). If set to 'individual', mar is an Nobs-by-NTrees matrix, where each column gives margins from each tree in the ensemble. If set to 'ensemble', mar is a single column of length Nobs showing the cumulative margins for the entire ensemble.
'trees'	Vector of indices indicating what trees to include in this calculation. By default, this argument is set to 'all' and the method uses all trees. If 'trees' is a numeric vector, the method returns an Nobs-by-NTrees matrix for 'cumulative' and 'individual' modes, where NTrees is the number of elements in the input vector, and a single column for 'ensemble' mode. For example, in the 'cumulative' mode, the first column gives

margins from `trees(1)`, the second column gives margins from `trees(1:2)` etc.

'treeweights'   Vector of tree weights. This vector must have the same length as the `'trees'` vector. `oobMargin` uses these weights to combine output from the specified trees by taking a weighted average instead of the simple nonweighted majority vote. You cannot use this argument in the `'individual'` mode.

See Also   CompactTreeBagger.margin

Purpose   Out-of-bag mean margins

Syntax   mar = oobMeanMargin(B)
                 mar = oobMeanMargin(B,'param1',val1,'param2',val2,...)

Description mar = oobMeanMargin(B) computes average classification margins for
                 out-of-bag observations in the training data, using the trained bagger B.
                 oobMeanMargin averages the margins over all out-of-bag observations.
                 mar is a row-vector of length NTrees, where NTrees is the number
                 of trees in the ensemble.

                 mar = oobMeanMargin(B,'param1',val1,'param2',val2,...)
                 specifies optional parameter name/value pairs:

'mode'	String indicating how oobMargin computes errors. If set to 'cumulative' (default), is a vector of length NTrees where the first element gives mean margin from trees(1), second column gives mean margins from trees(1:2) etc, up to trees(1:NTrees). If set to 'individual', mar is a vector of length NTrees, where each element is a mean margin from each tree in the ensemble . If set to 'ensemble', mar is a scalar showing the cumulative mean margin for the entire ensemble .
'trees'	Vector of indices indicating what trees to include in this calculation. By default, this argument is set to 'all' and the method uses all trees. If 'trees' is a numeric vector, the method returns a vector of length NTrees for 'cumulative' and 'individual' modes, where NTrees is the number of elements in the input vector, and a scalar for 'ensemble' mode. For example, in the 'cumulative' mode, the first element gives mean margin from trees(1), the second element gives mean margin from trees(1:2) etc.
'treeweights'	Vector of tree weights. This vector must have the same length as the 'trees' vector. oobMeanMargin uses these weights to combine output from the specified trees by taking a weighted average instead of the simple nonweighted majority vote. You cannot use this argument in the 'individual' mode.

See Also   CompactTreeBagger.meanMargin

TreeBagger.OOBPermutedVarCountRaiseMargin property

Purpose   Variable importance for raising margin

Description   The `OOBPermutedVarCountRaiseMargin` property is a numeric array of size 1-by-`Nvars` containing a measure of variable importance for each predictor. For any variable, the measure is the difference between the number of raised margins and the number of lowered margins if the values of that variable are permuted across the out-of-bag observations. This property is empty for regression trees.

Purpose   Variable importance for classification error

Description   The `OOBPermutedVarDeltaError` property is a numeric array of size
1-by-`Nvars` containing a measure of importance for each predictor
variable (feature). For any variable, the measure is the increase in
classification error averaged over the entire ensemble of trees if the
values of that variable are permuted across the out-of-bag observations.

TreeBagger.OOBPermutedVarDeltaMeanMargin property

Purpose   Variable importance for classification margin

Description   The `OOBPermutedVarDeltaMeanMargin` property is a numeric array of size 1-by-`Nvars` containing a measure of importance for each predictor variable (feature). For any variable, the measure is the decrease in the classification margin if the values of that variable are permuted across the out-of-bag observations. This property is empty for regression trees.

Purpose	Ensemble predictions for out-of-bag observations

Syntax

```
Y = oobPredict(B)
Y = oobPredict(B,'param1',val1,'param2',val2,...)
```

Description Y = oobPredict(B) computes predicted responses using the trained bagger B for out-of-bag observations in the training data. The output has one prediction for each observation in the training data. The returned Y is a cell array of strings for classification and a numeric array for regression.

Y = oobPredict(B,'param1',val1,'param2',val2,...) specifies optional parameter name/value pairs:

'trees'	Array of tree indices to use for computation of responses. Default is 'all'.
'treeweights'	Array of NTrees weights for weighting votes from the specified trees.

See Also CompactTreeBagger.predict, OOBIndices

ordinal class

Superclasses	categorical
Purpose	Arrays for ordinal categorical data
Description	An ordinal array is a kind of categorical array in which an ordering is encoded in the categories. More generally, categorical arrays are designed to store, manipulate, and process sets of data that assume values only in a finite, discrete set of categories or levels. Each ordinal array has its own set of levels and optional labels. Like a numerical array, an ordinal array can have any size or dimension.

Construction

ordinal	Construct ordinal categorical array

Methods

ismember	Test for membership
mergelevels	Merge levels
sort	Sort elements of ordinal array
sortrows	Sort rows

Inherited Methods

Methods in the following table are inherited from `categorical`.

addlevels	Add levels to categorical array
cat	Concatenate categorical arrays
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
disp	Display categorical array
display	Display categorical array
double	Convert categorical array to double array
droplevels	Drop levels
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
getlabels	Access categorical array labels
getlevels	Get categorical array levels
horzcat	Horizontal concatenation for categorical arrays
int16	Convert categorical array to signed 16-bit integer array
int32	Convert categorical array to signed 32-bit integer array
int64	Convert categorical array to signed 64-bit integer array
int8	Convert categorical array to signed 8-bit integer array
intersect	Set intersection for categorical arrays

ipermute	Inverse permute dimensions of categorical array
isempty	True for empty categorical array
isequal	True if categorical arrays are equal
islevel	Test for levels
ismember	True for elements of categorical array in set
isscalar	True if categorical array is scalar
isundefined	Test for undefined elements
isvector	True if categorical array is vector
length	Length of categorical array
levelcounts	Element counts by level
ndims	Number of dimensions of categorical array
numel	Number of elements in categorical array
permute	Permute dimensions of categorical array
reorderlevels	Reorder levels
repmat	Replicate and tile categorical array
reshape	Resize categorical array
rot90	Rotate categorical matrix 90 degrees
setdiff	Set difference for categorical arrays
setlabels	Label levels

setxor	Set exclusive-or for categorical arrays
shiftdim	Shift dimensions of categorical array
single	Convert categorical array to single array
size	Size of categorical array
squeeze	Squeeze singleton dimensions from categorical array
subsasgn	Subscripted assignment for categorical array
subsindex	Subscript index for categorical array
subsref	Subscripted reference for categorical array
summary	Summary statistics for categorical array
times	Product of categorical arrays
transpose	Transpose categorical matrix
uint16	Convert categorical array to unsigned 16-bit integers
uint32	Convert categorical array to unsigned 32-bit integers
uint64	Convert categorical array to unsigned 64-bit integers
uint8	Convert categorical array to unsigned 8-bit integers
union	Set union for categorical arrays

ordinal class

unique	Unique values in categorical array
vertcat	Vertical concatenation for categorical arrays

Properties

Inherited Properties

Properties in the following table are inherited from `categorical`.

labels	Text labels for levels
undeflabel	Text label for undefined levels

Copy Semantics

Value. To learn how this affects your use of the class, see Comparing Handle and Value Classes in the MATLAB Object-Oriented Programming documentation.

References

[1] Johnson, N. L., S. Kotz, and A. W. Kemp, Univariate Discrete Distributions, 2nd edition, Wiley, 1992, pp. 124-130.

See Also

`histc, nominal`

Purpose   Construct ordinal categorical 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.

ordinal

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   histc, nominal

CompactTreeBagger.outlierMeasure

Purpose   Outlier measure for data

Syntax   out = outlierMeasure(B,X)
               out = outlierMeasure(B,X,'param1',val1,'param2',val2,...)

Description   out = outlierMeasure(B,X) computes outlier measures for predictors
                   X using trees in the ensemble B. The method computes the outlier
                   measure for a given observation by taking an inverse of the average
                   squared proximity between this observation and other observations.
                   outlierMeasure then normalizes these outlier measures by subtracting
                   the median of their distribution, taking the absolute value of this
                   difference, and dividing by the median absolute deviation. A high value
                   of the outlier measure indicates that this observation is an outlier.

                   You can supply the proximity matrix directly by using the 'data'
                   parameter.

                   out = outlierMeasure(B,X,'param1',val1,'param2',val2,...)
                   specifies optional parameter name/value pairs:

   'data'   Flag indicating how to treat the X input argument. If
                set to 'predictors' (default), the method assumes X
                is a matrix of predictors and uses it for computation
                of the proximity matrix. If set to 'proximity', the
                method treats X as a proximity matrix returned
                by the proximity method. If you do not supply
                the proximity matrix, outlierMeasure computes
                it internally. If you use the proximity method to
                compute a proximity matrix, supplying it as input to
                outlierMeasure reduces computing time.

   'labels'   Vector of true class labels. True class labels can be
                either a numeric vector, character matrix, or cell
                array of strings. When you supply this parameter,
                the method performs the outlier calculation for any
                observations using only other observations from the
                same class. This parameter must specify one label for
                each observation (row) in X.

See Also   proximity

TreeBagger.OutlierMeasure property

Purpose   Measure for determining outliers

Description   The `OutlierMeasure` property is a numeric array of size `Nobs`-by-1, where `Nobs` is the number of observations in the training data, containing outlier measures for each observation.

See Also   `CompactTreeBagger.outlierMeasure`

Purpose   Parallel coordinates plot

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,...)
parallelcoords(axes,...)
```

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 NaNs 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 (NaNs) 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

parallelcoords

array containing a group index for each observation. (See "Grouped Data" on page 2-33.) 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.

parallelcoords(axes,...) plots into the axes with handle axes.

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 "Grouped Data" on page 2-33

andrewsplot, glyphplot

Purpose   Return parameter confidence intervals of ProbDistUnivParam object

Syntax   *CI* = paramci(*PD*)
                 CI = paramci(*PD*, *Alpha*)

Description *CI* = paramci(*PD*) returns *CI*, a 2-by-*N* array containing 95%
                 confidence intervals for the parameters of the ProbDistUnivParam
                 object *PD*. *N* is the number of parameters in the distribution.
                 When you create *PD* by specifying parameters (such as using the
                 ProbDistUnivParam.ProbDistUnivParam constructor or using the
                 fitdist function and specifying a 'binomial' or 'generalized
                 pareto' distribution) rather than by fitting to data, the confidence
                 intervals have a width of 0 because the parameters are viewed as
                 estimates of an unknown parameter.

                 CI = paramci(*PD*, *Alpha*) returns 100*(1 - *Alpha*)% confidence
                 intervals. Default *Alpha* is 0.05, which specifies 95% confidence
                 intervals.

                 Note If you create *PD* with a distribution that does not support
                 confidence intervals, then *CI* contains NaN values.

Inputs   *PD*   An object of the class ProbDistUnivParam.

                 Alpha   A value between 0 and 1 that specifies a
                                     confidence interval. Default is 0.05, which
                                     specifies 95% confidence intervals.

Outputs   *CI*   A 2-by-*N* array containing 95% confidence
                                     intervals for the parameters of the
                                     ProbDistUnivParam object *PD*. *N* is the
                                     number of parameters in the distribution.

ProbDistUnivParam.paramci

See Also   fitdist

Purpose   Read-only covariance matrix of parameter estimates of
ProbDistParametric object

Description   `ParamCov` is a read-only property of the `ProbDistParametric` class.
`ParamCov` is a covariance matrix containing the parameter estimates of
a distribution represented by a ProbDistParametric object. `ParamCov`
has a size of `NumParams`-by-`NumParams`.

Values   This covariance matrix includes estimates for both the specified
parameters and parameters that are fit to the data. For specified
parameters, the covariance is `0`, indicating the parameter is known
exactly. Use this information to view and compare the descriptions of
parameters supplied to create distributions.

ProbDistParametric.ParamDescription property

Purpose   Read-only cell array specifying descriptions of parameters of ProbDistParametric object

Description   `ParamDescription` is a read-only property of the `ProbDistParametric` class. `ParamDescription` is a cell array of strings specifying the descriptions or meanings of the parameters of a distribution represented by a ProbDistParametric object. `ParamDescription` has a length of `NumParams`.

Values   This cell array includes a brief description of the meaning of both the specified parameters and parameters that are fit to the data. The description is the same as the parameter name when no further description information is available. Use this information to view and compare the descriptions of parameters used to create distributions.

ProbDistParametric.ParamIsFixed property

Purpose   Read-only logical array specifying fixed parameters of ProbDistParametric object

Description   `ParamIsFixed` is a read-only property of the `ProbDistParametric` class. `ParamIsFixed` is a logical array specifying the fixed parameters of a distribution represented by a ProbDistParametric object. `ParamIsFixed` has a length of `NumParams`.

Values   This array specifies a `1` (`true`) for fixed parameters, and a `0` (`false`) for parameters that are estimated from the input data. Use this information to view and compare the fixed parameters used to create distributions.

ProbDistParametric.ParamNames property

Purpose   Read-only cell array specifying names of parameters of
ProbDistParametric object

Description   `ParamNames` is a read-only property of the `ProbDistParametric` class.
`ParamNames` is a cell array of strings specifying the names of the
parameters of a distribution represented by a ProbDistParametric
object. `ParamNames` has a length of `NumParams`.

Values   This cell array includes the names of both the specified parameters and
parameters that are fit to the data. Use this information to view and
compare the names of parameters used to create distributions.

Purpose   Parameter estimates

Description   The `Params` property is an `NClasses`-by-`NDims` cell array containing the parameter estimates, excluding the class priors. `Params(i,j)` contains the parameter estimates for the `j`th feature in the `i`th class. `Params(i,j)` is an empty cell if the `i`th class is empty.

The entry in `Params(i,j)` depends on the distribution type used for the `j`th feature, as follows:

`'normal'`	A vector of length two. The first element is the mean, and the second element is standard deviation.
`'kernel'`	A `ProbDistUnivKernel` object
`'mvmn'`	A vector containing the probability for each possible value of the `j`th feature in the `i`th class. The order of the probabilities is decided by the sorted order of all the unique values of the `j`th feature.
`'mn'`	A scalar representing the probability the `j`th token appearing in the `i`th class, `Prob(token j | class i)`. It is estimated as `(1 + the number of occurrence of token J in class I)/(NDims + the total number of token occurrence in class I)`.

ProbDistParametric.Params property

Purpose   Read-only array specifying values of parameters of ProbDistParametric object

Description   `Params` is a read-only property of the `ProbDistParametric` class. `Params` is an array of values specifying the values of the parameters of a distribution represented by a ProbDistParametric object. `Params` has a length of `NumParams`.

Values   This array includes the values of both the specified parameters and parameters that are fit to the data. Use this information to view and compare the values of parameters used to create distributions.

Purpose   Parent 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.

Examples   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
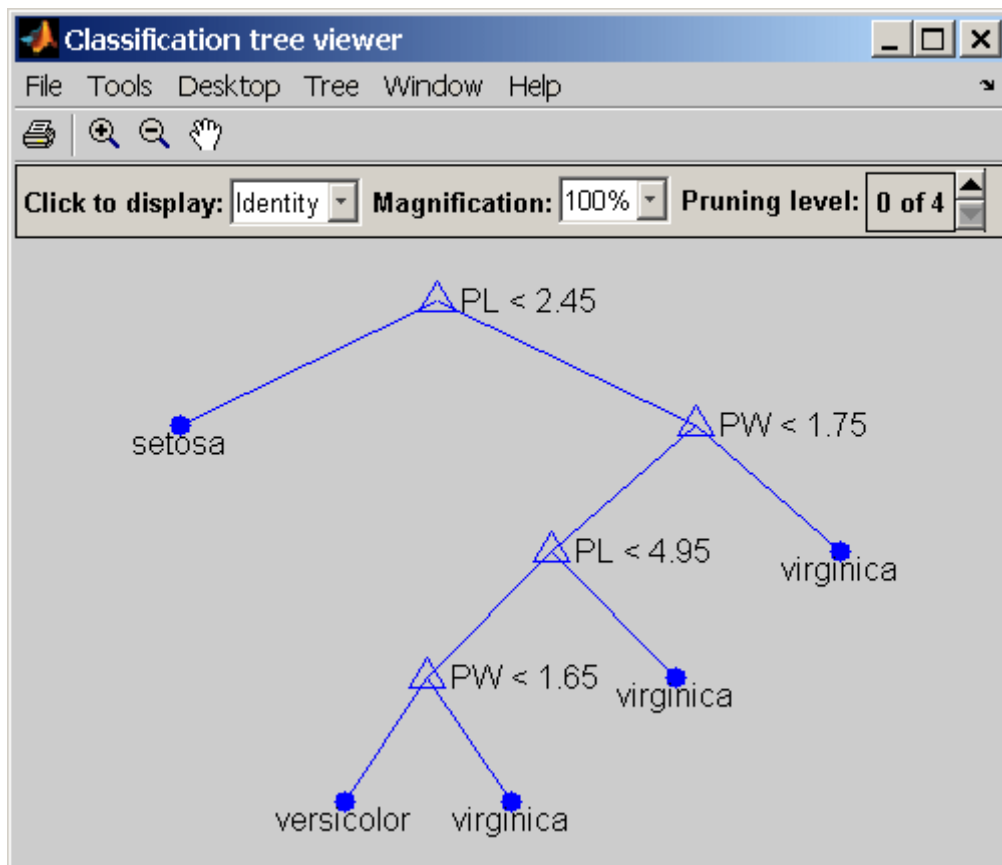
view(t)
```

```
p = parent(t)
p =
    0
    1
    1
    3
    3
    4
    4
```

```
        6
        6
```

References   [1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees.* Boca Raton, FL: CRC Press, 1984.

See Also   classregtree, children, numnodes

pareto

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`.

Examples	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

paretotails class

Superclasses	piecewisedistribution
Purpose	Empirical distributions with Pareto tails

Construction

paretotails	Construct Pareto tails object

Methods

lowerparams	Lower Pareto tails parameters
upperparams	Upper Pareto tails parameters

Inherited Methods

Methods in the following table are inherited from piecewisedistribution.

boundary	Piecewise distribution boundaries
cdf	Cumulative distribution function for piecewise distribution
disp	Display piecewisedistribution object
display	Display piecewisedistribution object
icdf	Inverse cumulative distribution function for piecewise distribution
nsegments	Number of segments
pdf	Probability density function for piecewise distribution
random	Random numbers from piecewise distribution
segment	Segments containing values

Properties   Objects of the `paretotails` class have no properties accessible by dot indexing, `get` methods, or `set` methods. To obtain information about a `paretotails` object, use the appropriate method.

Copy Semantics   Value. To learn how this affects your use of the class, see Comparing Handle and Value Classes in the MATLAB Object-Oriented Programming documentation.

See Also   "Generalized Pareto Distribution" on page B-34

paretotails

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.

Examples 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,cdf(obj,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   cdf, ecdf, gpfit, icdf, ksdensity

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-p_1 matrix, Y an n-by-p_2 matrix, and Z is an n-by-q matrix, with rows corresponding to observations, and columns corresponding to variables. RHO is a p_1-by-p_2 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 parameter/value
pars are listed in the following table.

Parameter	Values
'type'	• 'Pearson' — To compute Pearson (linear) partial correlations. This is the default. • 'Spearman' — To compute Spearman (rank) partial correlations.
'rows'	• 'all' — To use all rows regardless of missing (NaN) values. This is the default. • 'complete' — To use only rows with no missing values. • '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.

A 'pairwise' value for the rows parameter can produce a RHO that is
not positive definite. A 'complete' value always produces a positive
definite RHO, but when data is missing, the estimates will be based on
fewer observations, in general.

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

pcacov

Purpose	Principal component analysis on 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.

Examples
```
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

latent =
  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*. Hoboken, NJ: John Wiley and Sons, 1991.

[2] Jolliffe, I. T. *Principal Component Analysis*. 2nd ed., New York: Springer-Verlag, 2002.

[3] Krzanowski, W. J. *Principles of Multivariate Analysis: A User's Perspective*. New York: Oxford University Press, 1988.

[4] Seber, G. A. F., *Multivariate Observations*, Wiley, 1984.

See Also   barttest, biplot, factoran, pcares, princomp , rotatefactors

pcares

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.

Examples   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: A User's Perspective.* New York: Oxford University Press, 1988.

[4] Seber, G. A. F. *Multivariate Observations.* Hoboken, NJ: John Wiley & Sons, Inc., 1984.

See Also   factoran, pcacov, princomp

gmdistribution.PComponents property

Purpose   Input vector of mixing proportions

Description   Optional input vector of mixing proportions p, or its default value.

Purpose   Probability density functions

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   Compute the pdf of the normal distribution with mean 0 and standard deviation 1 at inputs –2, –1, 0, 1, 2:

gmdistribution.pdf

Purpose   Probability density function for Gaussian mixture distribution

Syntax   `y = pdf(obj,X)`

Description   `y = pdf(obj,X)` returns a vector `y` of length *n* containing the values of the probability density function (pdf) for the `gmdistribution` object `obj`, evaluated at the *n*-by-*d* data matrix `X`, where *n* is the number of observations and *d* is the dimension of the data. `obj` is an object created by `gmdistribution` or `fit`. `y(I)` is the cdf of observation `I`.

Examples   Create a `gmdistribution` object defining a two-component mixture of bivariate Gaussian distributions:

```
MU = [1 2;-3 -5];
SIGMA = cat(3,[2 0;0 .5],[1 0;0 1]);
p = ones(1,2)/2;
obj = gmdistribution(MU,SIGMA,p);
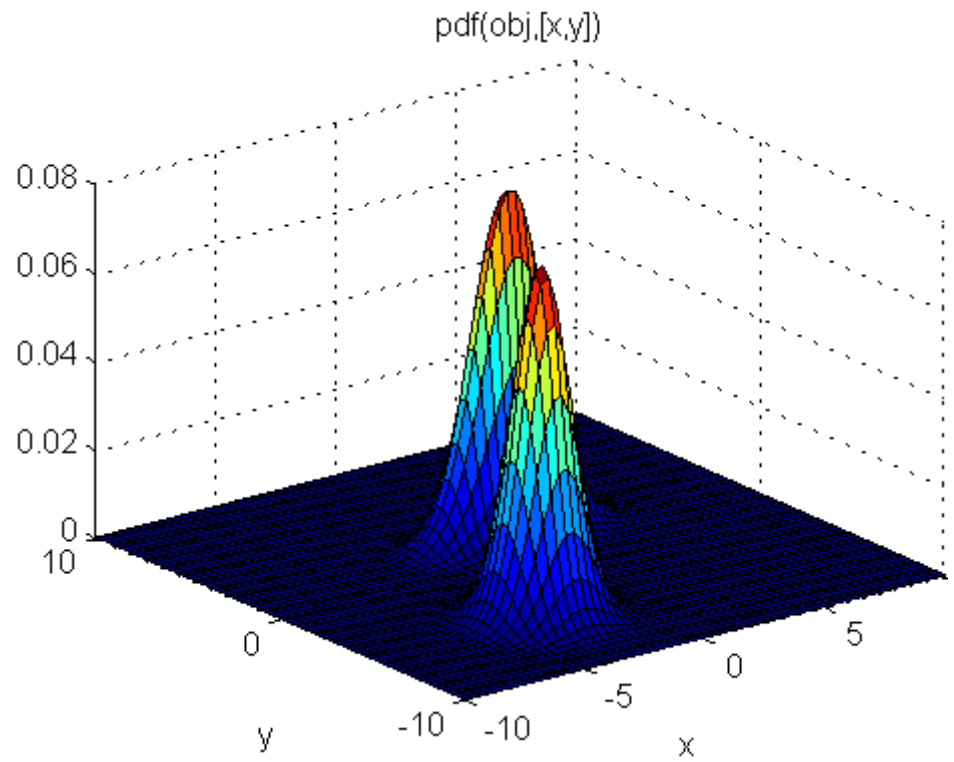
ezsurf(@(x,y)pdf(obj,[x y]),[-10 10],[-10 10])
```

pdf(obj,[x,y])

See Also   gmdistribution, fit, cdf, mvnpdf

piecewisedistribution.pdf

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.

Examples　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

ProbDist.pdf

Purpose	Return probability density function (PDF) for ProbDist object
Syntax	`Y = pdf(PD, X)`
Description	`Y = pdf(PD, X)` returns `Y`, an array containing the probability density function (PDF) for the ProbDist object `PD`, evaluated at values in `X`.

Inputs

`PD`	An object of the class `ProbDistUnivParam` or `ProbDistUnivKernel`.
`X`	A numeric array of values where you want to evaluate the PDF.

Outputs

`Y`	An array containing the probability density function (PDF) for the ProbDist object `PD`.

See Also   `pdf`

Purpose　　　Pairwise distance

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.

Metric	Description
'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

Metric	Description
'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 *k*th element is the distance between `u` and `V(k,:)`.

`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.

Metrics

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

$$\bar{x}_r = \frac{1}{n}\sum_j x_{rj} \quad \text{and} \quad \bar{x}_s = \frac{1}{n}\sum_j x_{sj}$$

- 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	Pearson system random numbers

Syntax	`r = pearsrnd(mu,sigma,skew,kurt,m,n)`
	`[r,type] = pearsrnd(...)`
	`[r,type,coefs] = 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 distribution

- 2 — Symmetric four-parameter beta distribution

- 3 — Three-parameter gamma distribution

- 4 — Not related to any standard distribution. The density is proportional to:

  $(1+((x–a)/b)^2)^{-c} \exp(–d \arctan((x–a)/b))$.

- 5 — Inverse gamma location-scale distribution

- 6 — *F* location-scale distribution

- 7 — Student's *t* location-scale distribution

`[r,type,coefs] = pearsrnd(...)` returns the coefficients `coefs` of the quadratic polynomial that defines the distribution via the

differential equation $\dfrac{\mathrm{d}(log(\mathrm{p(x)}))}{\mathrm{dx}} = \dfrac{(\mathrm{a} + \mathrm{x})}{(\mathrm{c(0)} + \mathrm{c(1)} \cdot \mathrm{x} + \mathrm{c(2)} \cdot \mathrm{x}^2)}$.

Examples   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

perfcurve

Purpose	Compute Receiver Operating Characteristic (ROC) curve or other performance curve for classifier output

Syntax

```
[X,Y] = perfcurve(labels,scores,posclass)
[X,Y] = perfcurve(labels,scores,posclass,'param1', val1,
    'param2',val2,...)
[X,Y,THRE,AUC,OPTROCPT,SUBY,SUBYNAMES] = perfcurve(labels,
    scores,posclass)
```

Description   [X,Y] = perfcurve(labels,scores,posclass) computes a ROC curve for a vector of classifier predictions scores given true class labels, labels. labels can be a numeric vector, logical vector, character matrix, cell array of strings or categorical vector. scores is a numeric vector of scores returned by a classifier for some data. posclass is the positive class label (scalar), either numeric (for numeric labels) or char. The returned values X and Y are coordinates for the performance curve and can be visualized with plot(X,Y). For more information on labels, scores, and posclass, see "Inputs" on page 17-947 . For more information on X and Y, see "Outputs" on page 17-950.

[X,Y] = perfcurve(labels,scores,posclass,'param1', val1,'param2',val2,...) specifies optional parameter name/value pairs. See "Inputs" on page 17-947 for a list of inputs, parameter name/value pairs, and respective explanations.

[X,Y,THRE,AUC,OPTROCPT,SUBY,SUBYNAMES] = perfcurve(labels,scores,posclass) returns:

- An array of thresholds on classifier scores for the computed values of X and Y (THRE).

- The area under curve (AUC) for the computed values of X and Y.

- The optimal operating point of the ROC curve (OPTROCPT).

- An array of Y values for negative subclasses (SUBY).

- A cell array of negative class names (SUBYNAMES).

For more information on each output, see "Outputs" on page 17-950.

Inputs

labels	labels can be a numeric vector, logical vector, character matrix, cell array of strings or categorical vector.
scores	scores is a numeric vector of scores returned by a classifier for some data. This vector must have as many elements as labels does.
posclass	posclass is the positive class label (scalar), either numeric (for numeric labels) or char. The specified positive class must be in the array of input labels.

The following name/value pairs are available for the syntax
[X,Y] = perfcurve(labels,scores,posclass,'param1', val1,'param2',val2,...)

Name ('param')	Value (val) and Description
'negClass'	List of negative classes. Can be either a numeric array or an array of chars or a cell array of strings. By default, negClass is set to 'all' and all classes found in the input array of labels that are not the positive class are considered negative. If negClass is a subset of the classes found in the input array of labels, instances with labels that do not belong to either positive or negative classes are discarded.

Name ('param')	Value (val) and Description
'xCrit'	Criterion to be computed for X. perfcurve supports the following criteria:

- TP — Number of true positive instances.

- FN — Number of false negative instances.

- FP — Number of false positive instances.

- TN — Number of true negative instances.

- TP+FP — Sum of TP and FP.

- RPP — Rate of positive predictions.
  RPP=(TP+FP)/(TP+FN+FP+TN)

- RNP — Rate of negative predictions.
  RNP=(TN+FN)/(TP+FN+FP+TN)

- accu — Accuracy. accu = (TP+TN)/(TP+FN+FP+TN)

- TPR, sens, reca — True positive rate, sensitivity, recall. TPR, sens, reca = TP/(TP+FN)

- FNR, miss — False negative rate, miss. FNR,miss=FN/(TP+FN)

- FPR, fall — False positive rate, fallout.
  FPR,fall=FP/(TN+FP)

- TNR, spec — True negative rate, specificity.
  TNR,spec=TN/(TN+FP)

- PPV, prec — Positive predictive value, precision.
  PPV,prec=TP/(TP+FP)

- NPV — Negative predictive value. NPV=TN/(TN+FN)

- ecost — Expected cost.
  ecost=(TP*COST(P|P)+FN*COST(N|P)+FP*
  COST(P|N)+TN*COST(N|N))/(TP+FN+FP+TN)

Name ('param')	Value (val) and Description
	In addition, you can define an arbitrary criterion by supplying an anonymous function of three arguments, (C,scale,cost), where C is a 2-by-2 confusion matrix, scale is a 2-by-1 array of class scales, and cost is a 2-by-2 misclassification cost matrix.
	Caution Some of these criteria return NaN values at one of the two special thresholds, 'reject all' and 'accept all'.
'yCrit'	Criterion to compute for Y. perfcurve supports the same criteria as for X.
'xVals'	Values for the X criterion. The default value for xVals is 'all' and perfcurve computes X and Y values for all scores. If the value for xVals is not 'all', it must be a numeric array. In this case, perfcurve computes X and Y only for the specified xVals.
'processNaN'	Specifies how perfcurve processes NaN scores. The default value is 'ignore' and perfcurve removes instances with NaN scores from the data. If you set the parameter to 'addtofalse', perfcurve adds instances with NaN scores to false classification counts in the respective class. That is, perfcurve always counts instances from the positive class as false negative (FN), and always counts instances from the negative class as false positive (FP).
'prior'	Either string or array with two elements. It represents prior probabilities for the positive and negative class, respectively. Default is 'empirical', that is, perfcurve derives prior probabilities from class frequencies. If set to 'uniform', perfcurve sets all prior probabilities equal.
'cost'	A 2-by-2 matrix of misclassification costs [C(P|P) C(N|P); C(P|N) C(N|N)], where C(I|J) is the cost of misclassifying class J as class I. The default value is [0 0.5; 0.5 0].

perfcurve

Outputs

X   *x*-coordinates for the performance curve. By default, X is false positive rate, FPR, (equivalently, fallout, or 1–specificity). To change this output, use the `'xCrit'` name/value input. For accepted criterion, see `'xCrit'` in "Inputs" on page 17-947 for more information.

Y   *y*-coordinates for the performance curve. By default, Y is true positive rate, TPR, (equivalently, recall, or sensitivity). To change this output, use the `'yCrit'` input. For accepted criterion, see `'xCrit'` in "Inputs" on page 17-947 for more information.

THRE   An array of thresholds on classifier scores for the computed values of X and Y. It has the same number of rows as X and Y. For each threshold, TP is the count of true positive observations with scores greater or equal to this threshold, and FP is the count of false positive observations with scores greater or equal to this threshold. perfcurve defines negative counts, TN and FN, in a similar way then sorts the thresholds in the descending order which corresponds to the ascending order of positive counts.

For the M distinct thresholds found in the array of scores, perfcurve returns the X, Y and THRE arrays with M+1 rows. perfcurve sets elements THRE(2:M+1) to the distinct thresholds, and THRE(1) replicates THRE(2). By convention, THRE(1) represents the highest `'reject all'` threshold and perfcurve computes the corresponding values of X and Y for TP=0 and FP=0. THRE(end) is the lowest `'accept all'` threshold for which TN=0 and FN=0.

AUC   The area under curve (AUC) for the computed values of X and Y. If you set xVals to 'all' (the default), perfcurve computes AUC using the returned X and Y values. If xVals is a numeric array, perfcurve computes AUC using X and Y values found from all distinct scores in the interval specified by the smallest and largest elements of xVals. More precisely, perfcurve finds X values for all distinct thresholds as if xVals were set to 'all', then uses a subset of these (with corresponding Y values) between min(xVals) and max(xVals) to compute AUC. The function uses trapezoidal approximation to estimate the area. If the first or last value of X or Y are NaNs, perfcurve removes them to allow calculation of AUC. This takes care of criteria that produce NaNs for the special 'reject all' or 'accept all' thresholds, for example, positive predictive value (PPV) or negative predictive value (NPV).

OPTROCPT   The optimal operating point of the ROC curve as an array of size 1-by-2 with FPR and TPR values for the optimal ROC operating point. perfcurve computes optrocpt only for the standard ROC curve and sets to NaNs otherwise. To obtain the optimal operating point for the ROC curve, perfcurve first finds the slope, S, using

$$S = \frac{\cos t(P \mid N) - \cos t(N \mid N)}{\cos t(N \mid P) - \cos t(P \mid P)} * \frac{N}{P}$$

where cost(I|J) is the cost of assigning an instance of class J to class I, and P=TP+FN and N=TN+FP are the total instance counts in the positive and negative class, respectively. perfcurve then finds the optimal operating point by moving the straight line with slope M from the upper left corner of the ROC plot (FPR=0, TPR=1) down and to the right until it intersects the ROC curve.

SUBY   An array of Y values for negative subclasses. If you
        only specify one negative class, SUBY is identical to Y.
        Otherwise SUBY is a matrix of size M-by-K, where M is
        the number of returned values for X and Y, and K is
        the number of negative classes. perfcurve computes
        Y values by summing counts over all negative classes.
        SUBY gives values of the Y criterion for each negative
        class separately. For each negative class, perfcurve
        places a new column in SUBY and fills it with Y values for
        TN and FP counted just for this class.

SUBYNAMES A cell array of negative class names. If you provide
           an input array, negClass, of negative class names,
           perfcurve copies it into SUBYNAMES. If you do not provide
           negClass, perfcurve extracts SUBYNAMES from input
           labels. The order of SUBYNAMES is the same as the order
           of columns in SUBY, that is, SUBY(:,1) is for negative
           class SUBYNAMES{1} etc.

Examples   Plot the ROC curve for classification by logistic regression:

```
load fisheriris
x = meas(51:end,1:2);
% iris data, 2 classes and 2 features
y = (1:100)'>50;
% versicolor=0, virginica=1
b = glmfit(x,y,'binomial');
% logistic regression
p = glmval(b,x,'logit');
% fit probabilities for scores
[X,Y] = perfcurve(species(51:end,:),p,'virginica');
plot(X,Y)
xlabel('False positive rate'); ylabel('True positive rate')
title('ROC for classification by logistic regression')
```


References   [1] T. Fawcett, ROC Graphs: Notes and Practical Considerations for Researchers, 2004.

[2] M. Zweig and G. Campbell, Receiver-Operating Characteristic (ROC) Plots: A Fundamental Evaluation Tool in Clinical Medicine, Clin. Chem. 39/4, 561-577, 1993.

[3] J. Davis and M. Goadrich, The relationship between precision-recall and ROC curves, in Proceedings of ICML '06, 233-240, 2006.

[4] C. Moskowitz and M. Pepe, Quantifying and comparing the predictive accuracy of continuous prognostic factors for binary outcomes, Biostatistics 5, 113-127, 2004.

[5] Y. Huang, M. Pepe and Z. Feng, Evaluating the Predictiveness of a Continuous Marker, U. Washington Biostatistics Paper Series, 282, 2006.

[6] W. Briggs and R. Zaretzki, The Skill Plot: A Graphical Technique for Evaluating Continuous Diagnostic Tests, Biometrics 63, 250-261, 2008.

[7] http://www2.cs.uregina.ca/~hamilton/courses/831/notes/lift_chart/lift_chart.html; http://www.dmreview.com/news/5329-1.html.

[8] R. Bettinger, Cost-Sensitive Classifier Selection Using the ROC Convex Hull Method, SAS Institute.

[9] http://www.stata.com/statalist/archive/2003-02/msg00060.html

See Also   "Performance Curves" on page 11-37, "Plotting a Performance Curve" on page 11-33, "Grouped Data" on page 2-33

`glmfit`, `mnrfit`

Purpose   Enumeration of 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.

Examples
```
perms([2 4 6])

ans =

    6    4    2
    6    2    4
    4    6    2
    4    2    6
    2    4    6
    2    6    4
```

See Also

combnk

categorical.permute

Purpose	Permute dimensions of categorical array
Syntax	B = permute(A,order)
Description	B = permute(A,order) rearranges the dimensions of the categorical array A so that they are in the order specified by the vector order. The array produced has the same values as A but the order of the subscripts needed to access any particular element are rearranged as specified by order. The elements of order must be a rearrangement of the numbers from 1 to n.
See Also	circshift, ipermute

Purpose   Piecewise-defined distributions

Construction `piecewisedistribution` is an abstract class. To construct a
`piecewisedistribution` object, use the subclass constructor,
`paretotails`.

Methods

boundary	Piecewise distribution boundaries
cdf	Cumulative distribution function for piecewise distribution
disp	Display piecewisedistribution object
display	Display piecewisedistribution object
icdf	Inverse cumulative distribution function for piecewise distribution
nsegments	Number of segments
pdf	Probability density function for piecewise distribution
random	Random numbers from piecewise distribution
segment	Segments containing values

Properties   Objects of the `piecewisedistribution` class have no properties
accessible by dot indexing, `get` methods, or `set` methods. To
obtain information about a `piecewisedistribution` object, use the
appropriate method.

**Copy
Semantics**   Value. To learn how this affects your use of the class, see Comparing
Handle and Value Classes in the MATLAB Object-Oriented
Programming documentation.

piecewisedistribution

Purpose   Create piecewise distribution object

Description   `piecewisedistribution` is an abstract class, and you cannot create
                  instances of it directly. You can create `paretotails` objects that are
                  derived from this class.

See Also   `paretotails`

Purpose   Partial least-squares regression

Syntax
```
[XL,YL] = plsregress(X,Y,ncomp)
[XL,YL,XS] = plsregress(X,Y,ncomp)
[XL,YL,XS,YS] = plsregress(X,Y,ncomp)
[XL,YL,XS,YS,BETA] = PLSREGRESS(X,Y,ncomp,...)
[XL,YL,XS,YS,BETA,PCTVAR] = plsregress(X,Y,ncomp)
[XL,YL,XS,YS,BETA,PCTVAR,MSE] = plsregress(X,Y,ncomp)
[XL,YL,XS,YS,BETA,PCTVAR,MSE] = plsregress(...,param1,val1,
    param2,val2,...)
[XL,YL,XS,YS,BETA,PCTVAR,MSE,stats] = PLSREGRESS(X,Y,ncomp,
    ...)
```

Description   [XL,YL] = plsregress(X,Y,ncomp) computes a partial least-squares
(PLS) regression of Y on X, using ncomp PLS components, and returns
the predictor and response loadings in XL and YL, respectively. X is
an n-by-p matrix of predictor variables, with rows corresponding to
observations and columns to variables. Y is an n-by-m response matrix.
XL is a p-by-ncomp matrix of predictor loadings, where each row contains
coefficients that define a linear combination of PLS components that
approximate the original predictor variables. YL is an m-by-ncomp
matrix of response loadings, where each row contains coefficients that
define a linear combination of PLS components that approximate the
original response variables.

[XL,YL,XS] = plsregress(X,Y,ncomp) returns the predictor scores
XS, that is, the PLS components that are linear combinations of the
variables in X. XS is an n-by-ncomp orthonormal matrix with rows
corresponding to observations and columns to components.

[XL,YL,XS,YS] = plsregress(X,Y,ncomp) returns the response
scores YS, that is, the linear combinations of the responses with which
the PLS components XS have maximum covariance. YS is an n-by-ncomp
matrix with rows corresponding to observations and columns to
components. YS is neither orthogonal nor normalized.

plsregress uses the SIMPLS algorithm, first centering X and Y by
subtracting off column means to get centered variables X0 and Y0.

However, it does not rescale the columns. To perform PLS with standardized variables, use `zscore` to normalize X and Y.

If `ncomp` is omitted, its default value is `min(size(X,1)-1,size(X,2))`.

The relationships between the scores, loadings, and centered variables X0 and Y0 are:

`XL = (XS\X0)' = X0'*XS,`

`YL = (XS\Y0)' = Y0'*XS,`

XL and YL are the coefficients from regressing X0 and Y0 on XS, and `XS*XL'` and `XS*YL'` are the PLS approximations to X0 and Y0.

`plsregress` initially computes YS as:

`YS = Y0*YL = Y0*Y0'*XS,`

By convention, however, `plsregress` then orthogonalizes each column of YS with respect to preceding columns of XS, so that `XS'*YS` is lower triangular.

`[XL,YL,XS,YS,BETA] = PLSREGRESS(X,Y,ncomp,...)` returns the PLS regression coefficients BETA. BETA is a $(p+1)$-by-m matrix, containing intercept terms in the first row:

`Y = [ones(n,1),X]*BETA + RESIDUALS,`

`Y0 = X0*BETA(2:end,:) + RESIDUALS.`

`[XL,YL,XS,YS,BETA,PCTVAR] = plsregress(X,Y,ncomp)` returns a 2-by-ncomp matrix PCTVAR containing the percentage of variance explained by the model. The first row of PCTVAR contains the percentage of variance explained in X by each PLS component, and the second row contains the percentage of variance explained in Y.

`[XL,YL,XS,YS,BETA,PCTVAR,MSE] = plsregress(X,Y,ncomp)` returns a 2-by-(ncomp+1) matrix MSE containing estimated mean-squared errors for PLS models with `0:ncomp` components. The first row of MSE contains mean-squared errors for the predictor variables in X, and the second row contains mean-squared errors for the response variable(s) in Y.

[XL,YL,XS,YS,BETA,PCTVAR,MSE] =
plsregress(...,*param1*,*val1*,*param2*,*val2*,...) specifies optional
parameter name/value pairs from the following table to control the
calculation of MSE.

Parameter	Value
`'cv'`	The method used to compute MSE. • When the value is a positive integer k, `plsregress` uses k-fold cross-validation. • When the value is an object of the `cvpartition` class, other forms of cross-validation can be specified. • When the value is `'resubstitution'`, `plsregress` uses X and Y both to fit the model and to estimate the mean-squared errors, without cross-validation. The default is `'resubstitution'`.
`'mcreps'`	A positive integer indicating the number of Monte-Carlo repetitions for cross-validation. The default value is 1. The value must be 1 if the value of `'cv'` is `'resubstitution'`.

[XL,YL,XS,YS,BETA,PCTVAR,MSE,stats] =
PLSREGRESS(X,Y,ncomp,...) returns a structure stats with the
following fields:

• W — A *p*-by-ncomp matrix of PLS weights so that XS = X0*W.

• T2 — The T^2 statistic for each point in XS.

• Xresiduals — The predictor residuals, that is, X0-XS*XL'.

• Yresiduals — The response residuals, that is, Y0-XS*YL'.

Examples   Load data on near infrared (NIR) spectral intensities of 60 samples of
gasoline at 401 wavelengths, and their octane ratings:

```
load spectra
X = NIR;
y = octane;
```

Perform PLS regression with ten components:

```
[XL,yl,XS,YS,beta,PCTVAR] = plsregress(X,y,10);
```

Plot the percent of variance explained in the response variable as a function of the number of components:

```
plot(1:10,cumsum(100*PCTVAR(2,:)),'-bo');
xlabel('Number of PLS components');
ylabel('Percent Variance Explained in y');
```


Compute the fitted response and display the residuals:

```
yfit = [ones(size(X,1),1) X]*beta;
residuals = y-yfit;
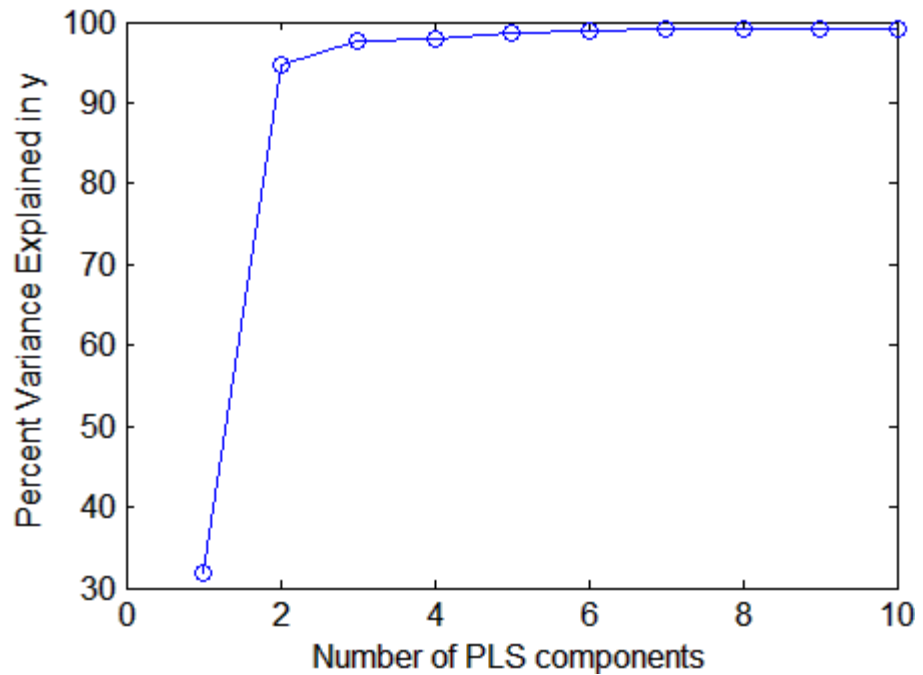
stem(residuals)
xlabel('Observation');
ylabel('Residual');
```


References [1] de Jong, S. "SIMPLS: An Alternative Approach to Partial Least Squares Regression." *Chemometrics and Intelligent Laboratory Systems.* Vol. 18, 1993, pp. 251–263.

[2] Rosipal, R., and N. Kramer. "Overview and Recent Advances in Partial Least Squares." *Subspace, Latent Structure and Feature Selection: Statistical and Optimization Perspectives Workshop (SLSFS*

2005), Revised Selected Papers (Lecture Notes in Computer Science 3940). Berlin, Germany: Springer-Verlag, 2006, pp. 34–51.

See Also   regress, sequentialfs

Purpose   Point generation method

Description   The `PointOrder` property contains a string that specifies the order in which the Sobol sequence points are produced. The property value must be one of `'standard'` or `'graycode'`. When set to `'standard'` the points produced match the original Sobol sequence implementation. When set to `'graycode'`, the sequence is generated using an implementation that uses the Gray code of the index instead of the index itself.

qrandstream.PointSet property

Purpose   Point set from which stream is drawn

Description   Point set from which the stream is drawn. Read only.

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 produces 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, poisspdf, poissinv, poisstat, poissfit, poissrnd

Purpose   Poisson parameter estimates

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.

                 $$\hat{\lambda} = \frac{1}{n} \sum_{i=1}^{n} x_i$$

Examples   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   mle, poisspdf, poisscdf, poissinv, poisstat, poissrnd

poissinv

Purpose	Poisson inverse 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, poisspdf, poisstat, poissfit, poissrnd

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, poissinv, poisstat, poissfit, poissrnd

poissrnd

Purpose
Poisson random numbers

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`.

Examples
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
random, poisspdf, poisscdf, poissinv, poisstat, poissfit

Purpose   Poisson mean and variance

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   poisspdf, poisscdf, poissinv, poissfit, poissrnd

polyconf

Purpose	Polynomial confidence intervals

Syntax	`Y = polyconf(p,X)`
	`[Y,DELTA] = polyconf(p,X,S)`
	`[Y,DELTA] = polyconf(p,X,S,`*`param1`*`,`*`val1`*`,`*`param2`*`,`*`val2`*`,...)`

Description   `Y = polyconf(p,X)` evaluates the polynomial `p` at the values in `X`. `p` is a vector of coefficients in descending powers.

`[Y,DELTA] = polyconf(p,X,S)` takes outputs `p` and `S` from `polyfit` and generates 95% prediction intervals `Y` − `DELTA` for new observations at the values in `X`.

`[Y,DELTA] = polyconf(p,X,S,`*`param1`*`,`*`val1`*`,`*`param2`*`,`*`val2`*`,...)` specifies optional parameter name/value pairs chosen from the following list.

Parameter	Value
`'alpha'`	A value between 0 and 1 specifying a confidence level of `100*(1-alpha)`%. The default is `0.05`.
`'mu'`	A two-element vector containing centering and scaling parameters. With this option, `polyconf` uses `(X-mu(1))/mu(2)` in place of `X`.
`'predopt'`	Either `'observation'` (the default) to compute prediction intervals for new observations at the values in `X`, or `'curve'` to compute confidence intervals for the fit evaluated at the values in `X`. See below.
`'simopt'`	Either `'off'` (the default) for nonsimultaneous bounds, or `'on'` for simultaneous bounds. See below.

The `'predopt'` and `'simopt'` parameters can be understood in terms of the following functions:

- $p(x)$ — the unknown mean function estimated by the fit

- $l(x)$ — the lower confidence bound

- $u(x)$ — the upper confidence bound

Suppose you make a new observation y_{n+1} at x_{n+1}, so that

$$y_{n+1}(x_{n+1}) = p(x_{n+1}) + \varepsilon_{n+1}$$

By default, the interval $[l_{n+1}(x_{n+1}), u_{n+1}(x_{n+1})]$ is a 95% confidence bound on $y_{n+1}(x_{n+1})$.

The following combinations of the `'predopt'` and `'simopt'` parameters allow you to specify other bounds.

simopt	predopt	Bounded Quantity
`'off'`	`'observation'`	$y_{n+1}(x_{n+1})$ (default)
`'off'`	`'curve'`	$p(x_{n+1})$
`'on'`	`'observation'`	$y_{n+1}(x)$, for all x
`'on'`	`'curve'`	$p(x)$, for all x

In general, `'observation'` intervals are wider than `'curve'` intervals, because of the additional uncertainty of predicting a new response value (the curve plus random errors). Likewise, simultaneous intervals are wider than nonsimultaneous intervals, because of the additional uncertainty of bounding values for all predictors x.

polyconf

Examples   This example uses code from the documentation example function
polydemo, and calls the documentation example function polystr
to convert the coefficient vector p into a string for the polynomial
expression displayed in the figure title. It combines the functions
polyfit, polyval, roots, and polyconf to produce a formatted display
of data with a polynomial fit.

Display simulated data with a quadratic trend, a fitted quadratic
polynomial, and 95% prediction intervals for new observations:

```
xdata = -5:5;
ydata = x.^2 - 5*x - 3 + 5*randn(size(x));

degree = 2;  % Degree of the fit
alpha = 0.05; % Significance level

% Compute the fit and return the structure used by POLYCONF.
[p,S] = polyfit(xdata,ydata,degree);

% Compute the real roots and determine the extent of the data.
r = roots(p)';        % Roots as a row vector.
real_r = r(imag(r) == 0);  % Real roots.

% Assure that the data are row vectors.
xdata = reshape(xdata,1,length(xdata));
ydata = reshape(ydata,1,length(ydata));

% Extent of the data.
mx = min([real_r,xdata]);
Mx = max([real_r,xdata]);
my = min([ydata,0]);
My = max([ydata,0]);

% Scale factors for plotting.
sx = 0.05*(Mx-mx);
```

```
sy = 0.05*(My-my);

% Plot the data, the fit, and the roots.
hdata = plot(xdata,ydata,'md','MarkerSize',5,'LineWidth',2);
hold on
xfit = mx-sx:0.01:Mx+sx;
yfit = polyval(p,xfit);
hfit = plot(xfit,yfit,'b-','LineWidth',2);
hroots = plot(real_r,zeros(size(real_r)),...
              'bo','MarkerSize',5,...
              'LineWidth',2,...
              'MarkerFaceColor','b');
grid on
plot(xfit,zeros(size(xfit)),'k-','LineWidth',2)
axis([mx-sx Mx+sx my-sy My+sy])

% Add prediction intervals to the plot.
[Y,DELTA] = polyconf(p,xfit,S,'alpha',alpha);
hconf = plot(xfit,Y+DELTA,'b--');
plot(xfit,Y-DELTA,'b--')

% Display the polynomial fit and the real roots.
approx_p = round(100*p)/100; % Round for display.
htitle = title(['{\bf Fit:   }',texlabel(polystr(approx_p))]);
set(htitle,'Color','b')
approx_real_r = round(100*real_r)/100; % Round for display.
hxlabel = xlabel(['{\bf Real Roots:    }',...
    num2str(approx_real_r)]);
set(hxlabel,'Color','b')

% Add a legend.
legend([hdata,hfit,hroots,hconf],...
        'Data','Fit','Real Roots of Fit',...
        '95% Prediction Intervals')
```

Fit: $0.81\,x^2 - 4.51\,x - 1.19$

Real Roots: 5.81   -0.25

See Also   `polyfit, polyval, polytool`

polytool

Purpose	Interactive polynomial fitting

Syntax
```
polytool
polytool(x,y)
polytool(x,y,n)
polytool(x,y,n,alpha)
polytool(x,y,n,alpha,xname,yname)
h = polytool(...)
```

Description   polytool

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.

See Also   polyfit, polyval, polyconf, invpred

Purpose   Posterior probabilities of components

Syntax   P = posterior(obj,X)
                [P,nlogl] = posterior(obj,X)

Description P = posterior(obj,X) returns the posterior probabilities of each of
                the *k* components in the Gaussian mixture distribution defined by obj
                for each observation in the data matrix X. X is *n*-by-*d*, where *n* is the
                number of observations and *d* is the dimension of the data. obj is an
                object created by gmdistribution or fit. P is *n*-by-*k*, with P(I,J) the
                probability of component J given observation I.

                posterior treats NaN values as missing data. Rows of X with NaN values
                are excluded from the computation.

                [P,nlogl] = posterior(obj,X) also returns nlogl, the negative
                log-likelihood of the data.

Examples   Generate data from a mixture of two bivariate Gaussian distributions
                using the mvnrnd function:

```
MU1 = [1 2];
SIGMA1 = [2 0; 0 .5];
MU2 = [-3 -5];
SIGMA2 = [1 0; 0 1];
X = [mvnrnd(MU1,SIGMA1,1000);mvnrnd(MU2,SIGMA2,1000)];
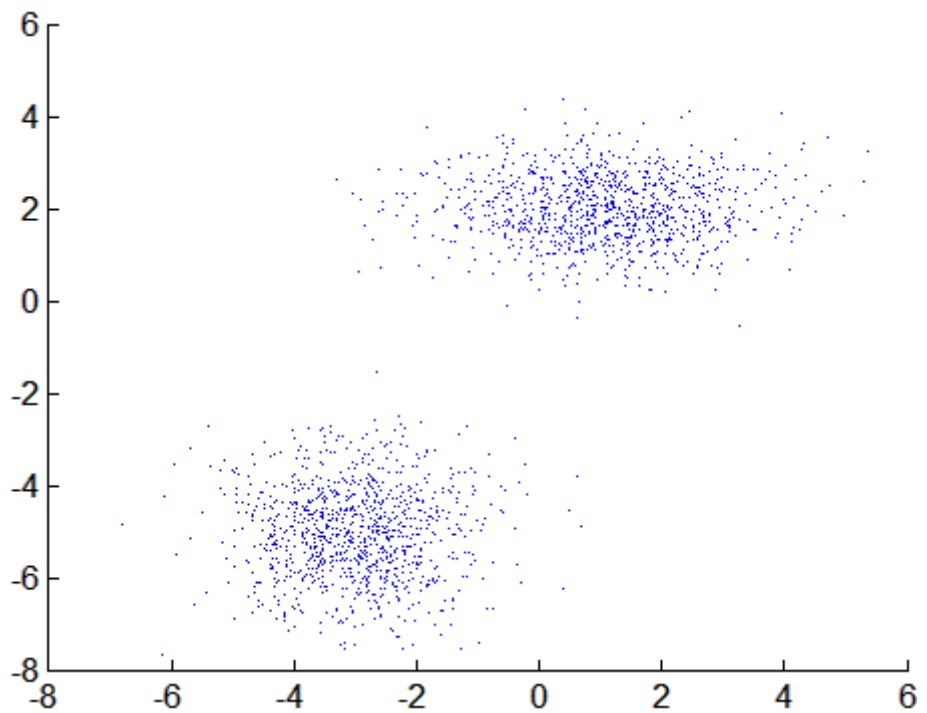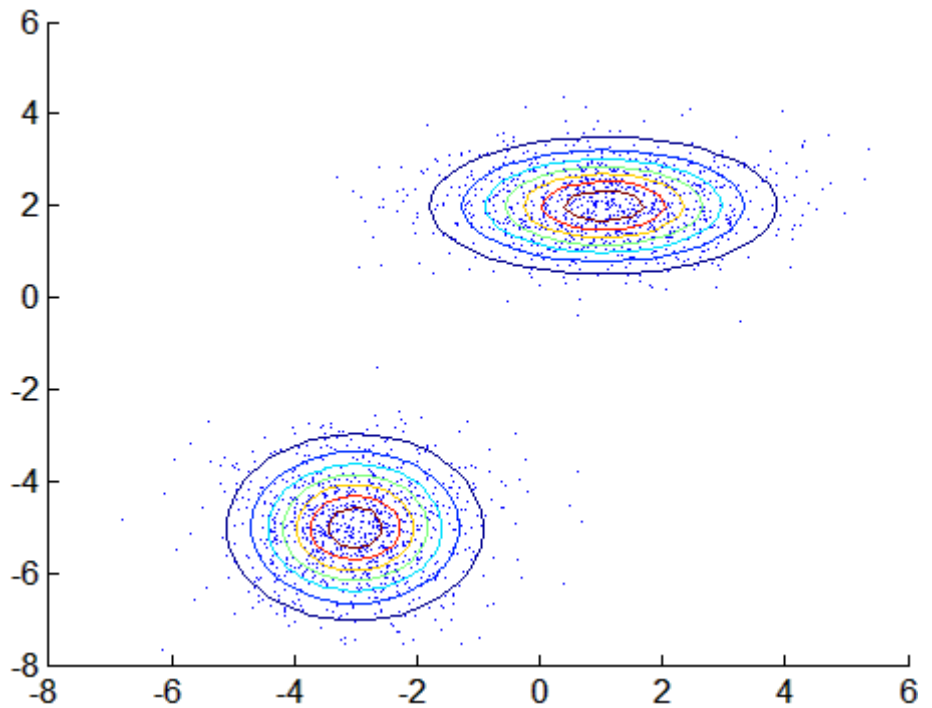
scatter(X(:,1),X(:,2),10,'.')
hold on
```

Fit a two-component Gaussian mixture model:

```
obj = gmdistribution.fit(X,2);
h = ezcontour(@(x,y)pdf(obj,[x y]),[-8 6],[-8 6]);
```

Compute posterior probabilities of the components:

```
P = posterior(obj,X);

delete(h)
scatter(X(:,1),X(:,2),10,P(:,1),'.')
hb = colorbar;
ylabel(hb,'Component 1 Probability')
```

gmdistribution.posterior

See Also   gmdistribution, fit, cluster, mahal

Purpose   Compute posterior probability of each class for test data

Syntax
```
post = posterior(nb,test)
[post,cpre] = posterior(nb,test)
[post,cpre,logp] = posterior(nb,test)
[...] = posterior(..., 'HandleMissing',val)
```

Description   `post = posterior(nb,test)` returns the posterior probability of the observations in `test` according to the `NaiveBayes` object `nb`. `test` is a N-by-`nb.ndims` matrix, where `N` is the number of observations in the test data. Rows of `test` correspond to points, columns of `test` correspond to features. `post` is a N-by-`nb.nclasses` matrix containing the posterior probability of each observation for each class. `post(i,j)` is the posterior probability of point I belonging to class j. Classes are ordered the same as `nb.clevels`, i.e., column j of `post` corresponds to the jth class in `nb.clevels`. The posterior probabilities corresponding to any empty classes are NaN.

`[post,cpre] = posterior(nb,test)` returns `cpre`, an N-by-1 vector, containing the class to which each row of `test` has been assigned. `cpre` has the same type as `nb.CLevels`.

`[post,cpre,logp] = posterior(nb,test)` returns `logp`, an N-by-1 vector containing estimates of the log of the probability density function (PDF). `logp(i)` is the log of the PDF of point `i`. The PDF value of point i is the sum of `Prob(point I | class J) * Pr{class J}` taken over all classes.

`[...] = posterior(..., 'HandleMissing',val)` specifies how `posterior` treats NaN (missing values). `val` can be one of the following:

NaiveBayes.posterior

'off' (default)	Observations with NaN in any of the columns are not classified into any class. The corresponding rows in post and logp are NaN. The corresponding rows in cpre are NaN (if obj.clevels is numeric or logical), empty strings (if obj.clevels is char or cell array of strings) or (if obj.clevels is categorical).
'on'	For observations having NaN in some (but not all) columns, post and cpre are computed using the columns with non-NaN values. Corresponding logp values are NaN.

See Also   NaiveBayes, fit, predict

17-986

Purpose   Percentiles

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
ith 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 NaNs 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   Predict response

Syntax
```
YFIT = predict(B,X)
[YFIT,stdevs] = predict(B,X)
[YFIT,scores] = predict(B,X)
[YFIT,scores,stdevs] = predict(B,X)
Y = predict(B,X,'param1',val1,'param2',val2,...)
```

Description   `YFIT = predict(B,X)` computes the predicted response of the trained ensemble `B` for predictors `X`. By default, `predict` takes a democratic (nonweighted) average vote from all trees in the ensemble. In `X`, rows represent observations and columns represent variables. `YFIT` is a cell array of strings for classification and a numeric array for regression.

For regression, `[YFIT,stdevs] = predict(B,X)` also returns standard deviations of the computed responses over the ensemble of the grown trees.

For classification, `[YFIT,scores] = predict(B,X)` returns scores for all classes. `scores` is a matrix with one row per observation and one column per class. For each observation and each class, the score generated by each tree is the probability of this observation originating from this class computed as the fraction of observations of this class in a tree leaf. `predict` averages these scores over all trees in the ensemble.

`[YFIT,scores,stdevs] = predict(B,X)` also returns standard deviations of the computed scores for classification. `stdevs` is a matrix with one row per observation and one column per class, with standard deviations taken over the ensemble of the grown trees.

`Y = predict(B,X,'param1',val1,'param2',val2,...)` specifies optional parameter name/value pairs:

'trees'	Array of tree indices to use for computation of responses. Default is 'all'.
'treeweights'	Array of NTrees weights for weighting votes from the specified trees.
'useifort'	Logical matrix of size Nobs-by-NTrees indicating which trees to use to make predictions for each observation. By default all trees are used for all observations.

See Also   classregtree.eval, TreeBagger.predict

NaiveBayes.predict

Purpose   Predict class label for test data

Syntax
```
cpre = predict(nb,test)
cpre = predict(...,'HandleMissing',val)
```

Description   cpre = predict(nb,test) classifies each row of data in test into one
of the classes according to the NaiveBayes classifier nb, and returns the
predicted class level cpre. test is an N-by-nb.ndims matrix, where N is
the number of observations in the test data. Rows of test correspond to
points, columns of test correspond to features. cpre is an N-by-1 vector
of the same type as nb.CLevels, and it indicates the class to which
each row of test has been assigned.

cpre = predict(...,'HandleMissing',val) specifies how predict
treats NaN (missing values). val can be one of the following:

'off' (default)	Observations with NaN in any of the columns are not classified into any class. The corresponding rows in post and logp are NaN. The corresponding rows in cpre are NaN (if obj.clevels is numeric or logical), empty strings (if obj.clevels is char or cell array of strings) or (if obj.clevels is categorical).
'on'	For observations having NaN in some (but not all) columns, post and predict computes cpre using the columns with non-NaN values. Corresponding logp values are NaN.

See Also   NaiveBayes, fit, posterior

TreeBagger.predict

Purpose	Predict response

Syntax
```
Y = predict(B,X)
Y = predict(B,X,'param1',val1,'param2',val2,...)
```

Description   `Y = predict(B,X)` computes predicted response of the trained ensemble `B` for data `X`. The output has one prediction for each row of `X`. The returned `Y` is a cell array of strings for classification and a numeric array for regression.

`Y = predict(B,X,'param1',val1,'param2',val2,...)` specifies optional parameter name/value pairs:

`'trees'`	Array of tree indices to use for computation of responses. Default is `'all'`.
`'treeweights'`	Array of `NTrees` weights for weighting votes from the specified trees.
`'useifort'`	Logical matrix of size `Nobs`-by-`NTrees` indicating which trees to use to make predictions for each observation. By default all trees are used for all observations.

See Also   `CompactTreeBagger.predict`

Purpose   Principal component analysis on data

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

[...] = 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.

Examples

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: A User's Perspective.* New York: Oxford University Press, 1988.

[4] Seber, G. A. F., *Multivariate Observations*, Wiley, 1984.

See Also

barttest, biplot, canoncorr, factoran, pcacov, pcares , rotatefactors

Purpose   Prior class probabilities

Description   The `Prior` property is a vector with prior probabilities for classes. This property is empty for ensembles of regression trees.

See Also   `classregtree`

ProbDist class

Purpose   Object representing probability distribution

Description ProbDist is an abstract class representing a probability distribution.

Construction ProbDist is an abstract class. You cannot create instances of this
class directly. You can construct an object in a subclass, such as
`ProbDistUnivParam` or `ProbDistUnivKernel`, either by calling the
subclass constructors (`ProbDistUnivParam.ProbDistUnivParam` or
`ProbDistUnivKernel.ProbDistUnivKernel`) or by using the `fitdist`
function.

Methods

cdf	Return cumulative distribution function (CDF) for ProbDist object
pdf	Return probability density function (PDF) for ProbDist object
random	Generate random number drawn from ProbDist object

Properties

DistName	Read-only string containing probability distribution name of ProbDist object
InputData	Read-only structure containing information about input data to ProbDist object
Support	Read-only structure containing information about support of ProbDist object

Copy Semantics   Value. To learn how this affects your use of the class, see Copying
Objects in the MATLAB Programming Fundamentals documentation.

See Also   `fitdist`
`ProbDistParametric` class
`ProbDistKernel` class
`ProbDistUnivParam` class
`ProbDistUnivKernel` class
`ProbDistUnivParam.ProbDistUnivParam` constructor
`ProbDistUnivKernel.ProbDistUnivKernel` constructor

ProbDistKernel class

Superclasses	ProbDist
Purpose	Object representing nonparametric probability distribution defined by kernel smoothing
Description	ProbDistKernel is an abstract class defining the properties and methods of a nonparametric distribution defined by a kernel smoothing function.
Construction	ProbDistKernel is an abstract class. You cannot create instances of this class directly. You can construct an object in a subclass, ProbDistUnivKernel, either by calling the subclass constructor, ProbDistUnivKernel.ProbDistUnivKernel, or by using the fitdist function.

Methods

cdf	Return cumulative distribution function (CDF) for ProbDist object
pdf	Return probability density function (PDF) for ProbDist object
random	Generate random number drawn from ProbDist object

Note The above methods are inherited from the ProbDist class.

Properties

BandWidth	Read-only value specifying bandwidth of kernel smoothing function for ProbDistKernel object
DistName	Read-only string containing probability distribution name of ProbDist object

InputData	Read-only structure containing information about input data to ProbDist object
Kernel	Read-only string specifying name of kernel smoothing function for ProbDistKernel object
Support	Read-only structure containing information about support of ProbDist object

Note Some of the above properties are inherited from the `ProbDist` class.

Copy Semantics

Value. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

See Also

`fitdist`
`ProbDist` class
`ProbDistUnivKernel` class
`ProbDistUnivKernel.ProbDistUnivKernel` constructor

ProbDistParametric class

Superclasses	ProbDist
Purpose	Object representing parametric probability distribution
Description	ProbDistParametric is an abstract class defining the properties and methods of a parametric probability distribution.
Construction	ProbDistParametric is an abstract class. You cannot create instances of this class directly. You can construct an object in its subclass, ProbDistUnivParam, either by calling the subclass constructor, ProbDistUnivParam.ProbDistUnivParam, or by using the fitdist function.

Methods

cdf	Return cumulative distribution function (CDF) for ProbDist object
pdf	Return probability density function (PDF) for ProbDist object
random	Generate random number drawn from ProbDist object

Note The above methods are inherited from the ProbDist class.

Properties

DistName	Read-only string containing probability distribution name of ProbDist object
InputData	Read-only structure containing information about input data to ProbDist object

NLogL	Read-only value specifying negative log likelihood for input data to ProbDistParametric object
NumParams	Read-only value specifying number of parameters of ProbDistParametric object
ParamCov	Read-only covariance matrix of parameter estimates of ProbDistParametric object
ParamDescription	Read-only cell array specifying descriptions of parameters of ProbDistParametric object
ParamIsFixed	Read-only logical array specifying fixed parameters of ProbDistParametric object
ParamNames	Read-only cell array specifying names of parameters of ProbDistParametric object
Params	Read-only array specifying values of parameters of ProbDistParametric object
Support	Read-only structure containing information about support of ProbDist object

Note Some of the above properties are inherited from the ProbDist class.

Copy Semantics

Value. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

ProbDistParametric class

See Also
fitdist
ProbDist class
ProbDistUnivParam class
ProbDistUnivParam.ProbDistUnivParam constructor

Superclasses   ProbDistKernel

Purpose   Object representing univariate kernel probability distribution

Description   A ProbDistUnivKernel object represents a univariate nonparametric probability distribution defined by kernel smoothing. You create this object using the fitdist function to fit the distribution to data.

Construction

fitdist	Fit probability distribution to data

Methods

cdf	Return cumulative distribution function (CDF) for ProbDist object
icdf	Return inverse cumulative distribution function (ICDF) for ProbDistUnivKernel object
iqr	Return interquartile range (IQR) for ProbDistUnivKernel object
median	Return median of ProbDistUnivKernel object
pdf	Return probability density function (PDF) for ProbDist object
random	Generate random number drawn from ProbDist object

Note Some of the above methods are inherited from the ProbDistKernel class.

ProbDistUnivKernel class

Properties

BandWidth	Read-only value specifying bandwidth of kernel smoothing function for ProbDistKernel object
DistName	Read-only string containing probability distribution name of ProbDist object
InputData	Read-only structure containing information about input data to ProbDist object
Kernel	Read-only string specifying name of kernel smoothing function for ProbDistKernel object
NLogL	Read-only value specifying negative log likelihood for input data to ProbDistUnivKernel object
Support	Read-only structure containing information about support of ProbDist object

Note Some of the above properties are inherited from the `ProbDistKernel` class.

Copy Semantics

Value. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

References

[1] Bowman, A. W., and A. Azzalini. *Applied Smoothing Techniques for Data Analysis*. New York: Oxford University Press, 1997.

See Also   `fitdist`
                      `ksdensity`
                      `ProbDist` class
                      `ProbDistKernel` class
                      `ProbDistUnivKernel.ProbDistUnivKernel` constructor

ProbDistUnivKernel

Purpose	Construct ProbDistUnivKernel object

Syntax

PD = ProbDistUnivKernel(*X*)
PD = ProbDistUnivKernel(*X*, *param1*, *val1*, *param2*, *val2*, ...)

Description

Tip Although you can use this constructor function to create a ProbDistUnivKernel object, using the fitdist function is an easier way to create the ProbDistUnivKernel object.

PD = ProbDistUnivKernel(*X*) creates *PD*, a ProbDistUnivKernel object, which represents a nonparametric probability distribution, based on a normal kernel smoothing function.

PD = ProbDistUnivKernel(*X*, *param1*, *val1*, *param2*, *val2*, ...) specifies optional parameter name/value pairs, as described in the Parameter/Values table. Parameter and value names are case insensitive.

Inputs

X	A column vector of data.

Note Any NaN values in *X* are ignored by the fitting calculations.

Parameter	Values
'censoring'	A Boolean vector the same size as *X*, containing 1s when the corresponding elements in *X* are right-censored observations and 0s when the corresponding elements are exact observations. Default is a vector of 0s. **Note** Any NaN values in this censoring vector are ignored by the fitting calculations.
'frequency'	A vector the same size as *X*, containing nonnegative integers specifying the frequencies for the corresponding elements in *X*. Default is a vector of 1s. **Note** Any NaN values in this frequency vector are ignored by the fitting calculations.
'kernel'	A string specifying the type of kernel smoother to use. Choices are: • 'normal' (default) • 'box' • 'triangle' • 'epanechnikov'

ProbDistUnivKernel

Parameter	Values
`'support'`	Any of the following to specify the support: • `'unbounded'` — Default. If the density can extend over the whole real line. • `'positive'` — To restrict it to positive values. • A two-element vector giving finite lower and upper limits for the support of the density.
`'width'`	A value specifying 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.

Outputs *PD*   An object in the `ProbDistUnivKernel` class, which is derived from the `ProbDist` class. It represents a nonparametric probability distribution.

References [1] Bowman, A. W., and A. Azzalini. *Applied Smoothing Techniques for Data Analysis.* New York: Oxford University Press, 1997.

See Also   fitdist
ksdensity

Superclasses   ProbDistParametric

Purpose   Object representing univariate parametric probability distribution

Description   A ProbDistUnivParam object represents a univariate parametric probability distribution. You create this object by using the constructor (`ProbDistUnivParam.ProbDistUnivParam`) and supplying parameter values, or by using the `fitdist` function to fit the distribution to data.

Construction

fitdist	Fit probability distribution to data
ProbDistUnivParam	Construct ProbDistUnivParam object

Methods

cdf	Return cumulative distribution function (CDF) for ProbDist object
icdf	Return inverse cumulative distribution function (ICDF) for ProbDistUnivParam object
iqr	Return interquartile range (IQR) for ProbDistUnivParam object
mean	Return mean of ProbDistUnivParam object
median	Return median of ProbDistUnivParam object
paramci	Return parameter confidence intervals of ProbDistUnivParam object
pdf	Return probability density function (PDF) for ProbDist object

ProbDistUnivParam class

random	Generate random number drawn from ProbDist object
std	Return standard deviation of ProbDistUnivParam object
var	Return variance of ProbDistUnivParam object

Note Some of the above methods are inherited from the `ProbDistParametric` class.

Properties

DistName	Read-only string containing probability distribution name of ProbDist object
InputData	Read-only structure containing information about input data to ProbDist object
NLogL	Read-only value specifying negative log likelihood for input data to ProbDistParametric object
NumParams	Read-only value specifying number of parameters of ProbDistParametric object
ParamCov	Read-only covariance matrix of parameter estimates of ProbDistParametric object
ParamDescription	Read-only cell array specifying descriptions of parameters of ProbDistParametric object

ParamIsFixed	Read-only logical array specifying fixed parameters of ProbDistParametric object
ParamNames	Read-only cell array specifying names of parameters of ProbDistParametric object
Params	Read-only array specifying values of parameters of ProbDistParametric object
Support	Read-only structure containing information about support of ProbDist object

Note The above properties are inherited from the `ProbDistParametric` class.

Note Parameter values are also properties. For example, if you create PD, a univariate parametric probability distribution object that represents a normal distribution, then `PD.mu` and `PD.sigma` are properties that give the values of the mu and sigma parameters.

Copy Semantics

Value. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

References

[1] Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*. Vol. 1, Hoboken, NJ: Wiley-Interscience, 1993.

ProbDistUnivParam class

[2] Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*. Vol. 2, Hoboken, NJ: Wiley-Interscience, 1994.

See Also   fitdist
ProbDist class
ProbDistParametric class
ProbDistUnivParam.ProbDistUnivParam constructor
Appendix B, "Distribution Reference" — For more information on parametric distributions

Purpose   Construct ProbDistUnivParam object

Syntax   *PD* = ProbDistUnivParam(*DistName*, *Params*)

Description   *PD* = ProbDistUnivParam(*DistName*, *Params*) creates *PD*, a
ProbDistUnivParam object, which represents a probability distribution.
This distribution is defined by the parametric distribution specified by
DistName, with parameters specified by the numeric vector *Params*.

Inputs   *DistName*   A string specifying a distribution. Choices are:

- 'beta'
- 'binomial'
- 'birnbaumsaunders'
- 'exponential'
- 'extreme value' or ev'
- 'gamma'
- 'generalized extreme value' or 'gev'
- 'generalized pareto' or 'gp'
- 'inversegaussian'
- 'logistic'
- 'loglogistic'
- 'lognormal'
- 'nakagami'
- 'negative binomial' or 'nbin'
- 'normal'
- 'poisson'

- `'rayleigh'`

- `'rician'`

- `'tlocationscale'`

- `'weibull'` or `'wbl'`

  For more information on these parametric distributions, see Appendix B, "Distribution Reference".

Params	Numeric vector of distribution parameters. The number and type of parameters depends on the distribution you specify with *DistName*. For information on parameters for each distribution type, see Appendix B, "Distribution Reference".

Outputs

PD	An object in the ProbDistUnivParam class, which is derived from the ProbDist class. It represents a parametric probability distribution.

Examples

1 Create an object representing a normal distribution with a mean of 100 and a standard deviation of 10.

```
pd = ProbDistUnivParam('normal',[100 10])

pd =

normal distribution

    mu = 100
    sigma = 10
```

2 Generate a 4-by-5 matrix of random values from this distribution.

```
random(pd,4,5)

ans =

  105.3767   103.1877   135.7840   107.2540    98.7586
  118.3389    86.9231   127.6944    99.3695   114.8970
   77.4115    95.6641    86.5011   107.1474   114.0903
  108.6217   103.4262   130.3492    97.9503   114.1719
```

References   [1] Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*. Vol. 1, Hoboken, NJ: Wiley-Interscience, 1993.

[2] Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*. Vol. 2, Hoboken, NJ: Wiley-Interscience, 1994.

See Also   Appendix B, "Distribution Reference" — For more information on parametric distributions
fitdist

probplot

Purpose	Probability plots

Syntax

```
probplot(Y)
probplot(distribution,Y)
probplot(Y,cens,freq)
probplot(ax,Y)
probplot(...,'noref')
probplot(ax,PD)
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. The ith sorted value from a sample of size N is plotted against the midpoint in the jump of the empirical CDF on the y axis. With uncensored data, that midpoint is $(i–0.5)/N$. With censored data (see below), the y value is more complicated to compute.

`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)

The *y* axis scale is based on the selected distribution. The *x* axis has a log scale for the Weibull and lognormal distributions, and a linear scale for the others.

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,PD)` takes a probability distribution object, PD, and adds a fitted line to the axes specified by `ax` to represent the probability distribution specified by PD. PD is a ProbDist object of the `ProbDistUnivParam` class or `ProbDistUnivKernel` class.

`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, `@wblcdf`). `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.

Examples ### Example 1

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')
```


Probability plot for Weibull distribution

Example 2

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');
t = @(data,mu,sig,df)cdf('tlocationscale',data,mu,sig,df);
h = probplot(gca,t,p);
set(h,'color','r','linestyle','-')
title('{\bf Probability Plot}')
legend('Data','Normal','t','Location','NW')
```

probplot

See Also   normplot, ecdf, wblplot

Purpose   Procrustes analysis

Syntax
```
d = procrustes(X,Y)
[d,Z] = procrustes(X,Y)
[d,Z,transform] = procrustes(X,Y)
[...] = procrustes(...,'scaling',flag)
[...] = procrustes(...,'reflection',flag)
```

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))
```

That is, 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 `Y(i)` to `X(i)`. 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 array with fields:

- `c` — Translation component

- `T` — Orthogonal rotation and reflection component

- `b` — Scale component

That is:

```
c = transform.c;
```

```
T = transform.T;
b = transform.b;

Z = b*Y*T + c;
```

[...] = procrustes(...,'scaling',*flag*), when *flag* is false, allows you to compute the transformation without a scale component (that is, with b equal to 1). The default *flag* is true.

[...] = procrustes(...,'reflection',*flag*), when *flag* is false, allows you to compute the transformation without a reflection component (that is, with det(T) equal to 1). The default *flag* is 'best', which computes the best-fitting transformation, whether or not it includes a reflection component. A *flag* of true forces the transformation to be computed with a reflection component (that is, with det(T) equal to -1)

Examples   This example creates some random points in two dimensions, then rotates, scales, translates, and adds some noise to those points. It uses procrustes to conform Y to X, then plots the original X and Y with 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');
```

References

[1] Kendall, David G. "A Survey of the Statistical Theory of Shape." *Statistical Science.* Vol. 4, No. 2, 1989, pp. 87–99.

[2] Bookstein, Fred L. *Morphometric Tools for Landmark Data.* Cambridge, UK: Cambridge University Press, 1991.

[3] Seber, G. A. F. *Multivariate Observations.* Hoboken, NJ: John Wiley & Sons, Inc., 1984.

See Also   cmdscale, factoran

CompactTreeBagger.proximity

Purpose   Proximity matrix for data

Syntax   prox = proximity(B,X)

Description prox = proximity(B,X) computes a numeric matrix of size
Nobs-by-Nobs of proximities for data X, where Nobs is the number of
observations (rows) in X. Proximity between any two observations in the
input data is defined as a fraction of trees in the ensemble B for which
these two observations land on the same leaf. This is a symmetric
matrix with ones on the diagonal and off-diagonal elements ranging
from 0 to 1.

Purpose   Proximity matrix for observations

Description   The Proximity property is a numeric matrix of size Nobs-by-Nobs, where Nobs is the number of observations in the training data, containing measures of the proximity between observations. For any two observations, their proximity is defined as the fraction of trees for which these observations land on the same leaf. This is a symmetric matrix with 1s on the diagonal and off-diagonal elements ranging from 0 to 1.

See Also   CompactTreeBagger.proximity, classregtree.varimportance

classregtree.prune

Purpose   Prune tree

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.

Examples   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
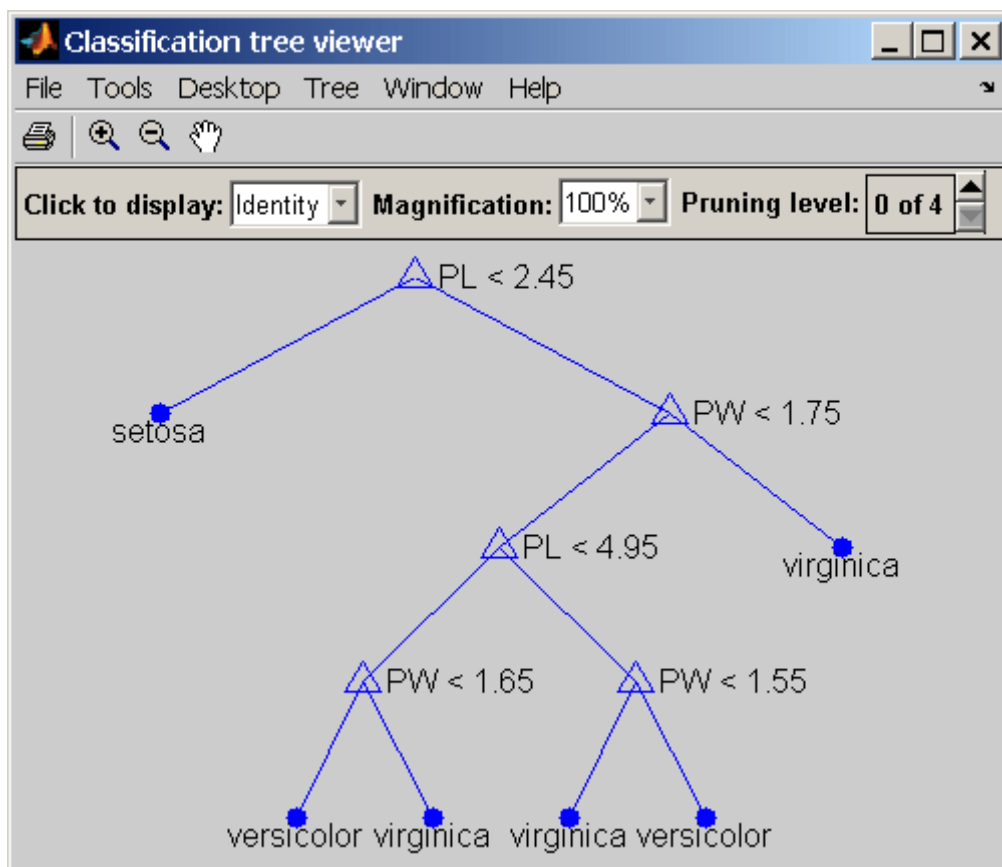
view(t1)
```

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
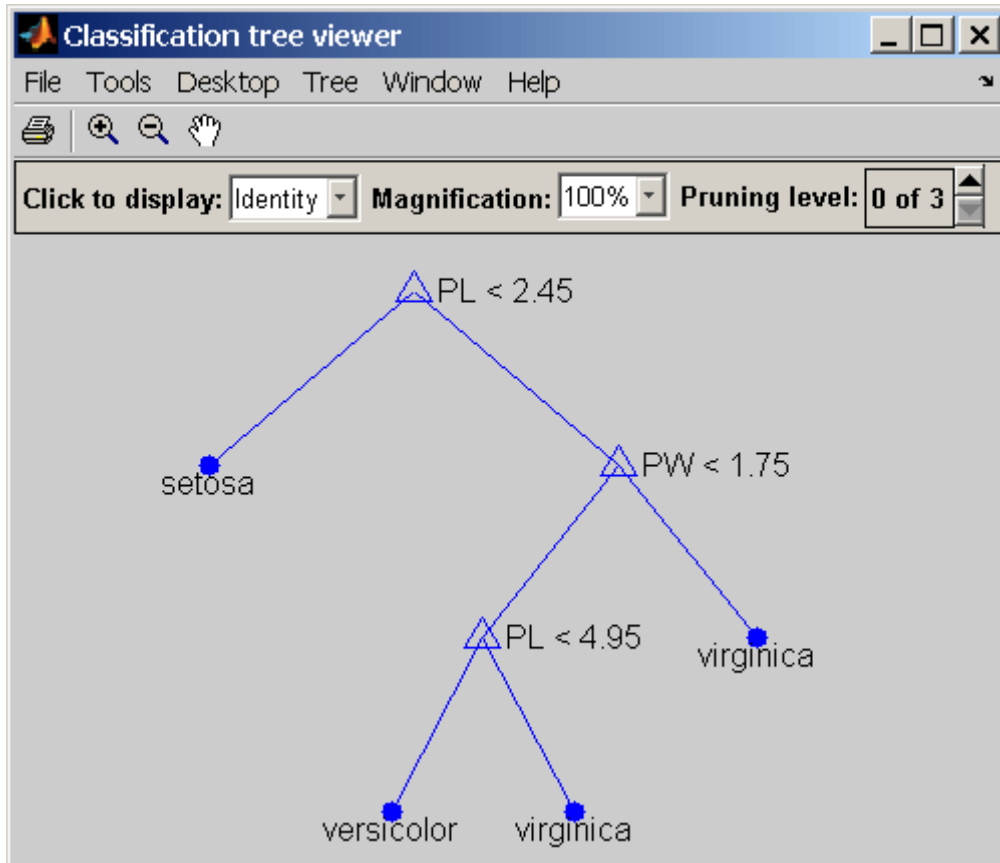
view(t2)
```


References   [1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees.* Boca Raton, FL: CRC Press, 1984.

classregtree.prune

Purpose   Flag to prune trees

Description   The Prune property is true if decision trees are pruned and false if they are not. Pruning decision trees is not recommended for ensembles. The default value is false.

See Also   classregtree.prune

qrandstream.qrand

Purpose   Generate quasi-random points from stream

Syntax
```
x = qrand(q)
X = qrand(q,n)
```

Description   `x = qrand(q)` returns the next value `x` in the quasi-random number stream `q` of the `qrandstream` class. `x` is a 1-by-d vector, where d is the dimension of the stream. The command sets `q.State` to the index in the underlying point set of the next value to be returned.

`X = qrand(q,n)` returns the next `n` values `X` in an n-by-d matrix.

Objects `q` of the `qrandstream` class encapsulate properties of a specified quasi-random number stream. Values of the stream are not generated and stored in memory until `q` is accessed using `qrand`.

Examples   Use `qrandstream` to construct a 3-D Halton stream, based on a point set that skips the first 1000 values and then retains every 101st point:

```
q = qrandstream('halton',3,'Skip',1e3,'Leap',1e2)
q =
   Halton quasi-random stream in 3 dimensions
   Point set properties:
               Skip : 1000
               Leap : 100
     ScrambleMethod : none

nextIdx = q.State
nextIdx =
      1
```

Use `qrand` to generate two samples of size four:

```
X1 = qrand(q,4)
X1 =
    0.0928    0.3475    0.0051
    0.6958    0.2035    0.2371
    0.3013    0.8496    0.4307
```

```
        0.9087    0.5629    0.6166
nextIdx = q.State
nextIdx =
     5

X2 = qrand(q,4)
X2 =
    0.2446    0.0238    0.8102
    0.5298    0.7540    0.0438
    0.3843    0.5112    0.2758
    0.8335    0.2245    0.4694
nextIdx = q.State
nextIdx =
     9
```

Use reset to reset the stream, then generate another sample:

```
reset(q)
nextIdx = q.State
nextIdx =
     1

X = qrand(q,4)
X =
    0.0928    0.3475    0.0051
    0.6958    0.2035    0.2371
    0.3013    0.8496    0.4307
    0.9087    0.5629    0.6166
```

See Also   qrandstream, reset

qrandset class

Purpose   Quasi-random point sets

Description   `qrandset` is a base class that encapsulates a sequence of multi-dimensional quasi-random numbers. This base class is abstract and cannot be instantiated directly. Concrete subclasses include `sobolset` and `haltonset`.

Construction

qrandset	Create categorical array

Methods

disp	Display qrandset object
end	Last index in indexing expression for point set
length	Length of point set
ndims	Number of dimensions in matrix
net	Generate quasi-random point set
scramble	Scramble quasi-random point set
size	Number of dimensions in matrix
subsref	Subscripted reference for qrandset

Properties

Dimensions	Number of dimensions
Leap	Interval between points
ScrambleMethod	Scramble settings
Skip	Number of initial points omitted
Type	Subclass

Copy Semantics   Value. To learn how this affects your use of the class, see Comparing Handle and Value Classes in the MATLAB Object-Oriented Programming documentation.

See Also   "Quasi-Random Point Sets" on page 5-144

qrandset

Purpose   Create categorical array

Description   qrandset is an abstract class, and you cannot create instances of it
                     directly. You must use haltonset or sobolset to create a qrandset
                     object.

See Also   haltonset, sobolset

Purpose   Quasi-random number streams

Construction   qrandstream   Construct quasi-random number
                                                    stream

Methods   addlistener   Add listener for event

                     delete   Delete handle object

                     disp   Display `qrandstream` object

                     eq   Test handle equality

                     findobj   Find objects matching specified
                                                    conditions

                     findprop   Find property of MATLAB handle
                                                    object

                     ge   Greater than or equal relation for
                                                    handles

                     gt   Greater than relation for handles

                     isvalid   Test handle validity

                     le   Less than or equal relation for
                                                    handles

                     lt   Less than relation for handles

                     ne   Not equal relation for handles

                     notify   Notify listeners of event

                     qrand   Generate quasi-random points
                                                    from stream

                     rand   Generate quasi-random points
                                                    from stream

                     reset   Reset state

qrandstream class

Properties

PointSet	Point set from which stream is drawn
State	Point set index

Copy Semantics

Handle. To learn how this affects your use of the class, see Comparing Handle and Value Classes in the MATLAB Object-Oriented Programming documentation.

Purpose	Construct quasi-random number stream
Syntax	q = qrandstream(*type*,d) q = qrandstream(*type*,d,*prop1*,*val1*,*prop2*,*val2*,...) q = qrandstream(p)
Description	q = qrandstream(*type*,d) constructs a d-dimensional quasi-random number stream q of the qrandstream class, of type specified by the string *type*. *type* is either 'halton' or 'sobol', and q is based on a point set from either the haltonset class or sobolset class, respectively, with default property settings. q = qrandstream(*type*,d,*prop1*,*val1*,*prop2*,*val2*,...) specifies property name/value pairs for the point set on which the stream is based. Applicable properties depend on *type*. q = qrandstream(p) constructs a stream based on the specified point set p. p must be a point set from either the haltonset class or sobolset class.
Examples	Construct a 3-D Halton stream, based on a point set that skips the first 1000 values and then retains every 101st point:

```
q = qrandstream('halton',3,'Skip',1e3,'Leap',1e2)
q =
   Halton quasi-random stream in 3 dimensions
   Point set properties:
              Skip : 1000
              Leap : 100
     ScrambleMethod : none

nextIdx = q.State
nextIdx =
     1
```

Use qrand to generate two samples of size four:

```
X1 = qrand(q,4)
```

```
X1 =
    0.0928    0.3475    0.0051
    0.6958    0.2035    0.2371
    0.3013    0.8496    0.4307
    0.9087    0.5629    0.6166
nextIdx = q.State
nextIdx =
     5

X2 = qrand(q,4)
X2 =
    0.2446    0.0238    0.8102
    0.5298    0.7540    0.0438
    0.3843    0.5112    0.2758
    0.8335    0.2245    0.4694
nextIdx = q.State
nextIdx =
     9
```

Use reset to reset the stream, and then generate another sample:

```
reset(q)
nextIdx = q.State
nextIdx =
     1

X = qrand(q,4)
X =
    0.0928    0.3475    0.0051
    0.6958    0.2035    0.2371
    0.3013    0.8496    0.4307
    0.9087    0.5629    0.6166
```

See Also   haltonset,qrand, reset, sobolset

Purpose   Quantile-quantile plot

Syntax   qqplot(X)
                  qqplot(X,Y)
                  qqplot(X,PD)
                  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.

                  qqplot(X,PD) makes an empirical quantile-quantile plot of
                  the quantiles of the data in the vector X versus the quantiles
                  of the distribution specified by PD, a ProbDist object of the
                  ProbDistUnivParam class or ProbDistUnivKernel class.

                  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.

Examples   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);
```

qqplot

See Also   `normplot`

quantile

Purpose   Quantiles

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
                 dimth 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 NaNs 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

qrandstream.rand

Purpose   Generate quasi-random points from stream

Syntax
```
rand
rand(q,n)
rand(q)
rand(q,m,n)
rand(q,[m,n])
rand(q,m,n,p,...)
rand(q,[m,n,p,...])
```

Description   rand returns a matrix of quasi-random values and is intended to allow objects of the qrandstream class to be used in code that contains calls to the rand method of the MATLAB pseudo-random randstream class. Due to the multidimensional nature of quasi-random numbers, only some syntaxes of rand are supported by the qrandstream class.

rand(q,n) returns an n-by-n matrix only when n is equal to the number of dimensions. Any other value of n produces an error.

rand(q) returns a scalar only when the stream is in one dimension. Having more than one dimension in q produces an error.

rand(q,m,n) or rand(q,[m,n]) returns an m-by-n matrix only when n is equal to the number of dimensions in the stream. Any other value of n produces an error.

rand(q,m,n,p,...) or rand(q,[m,n,p,...]) produces an error unless p and all following dimensions sizes are equal to one.

Examples   Generate the first 256 points from a 5-D Sobol sequence:

```
q = qrandstream('sobol',5);
X = rand(q,256,5);
```

See Also   qrandstream, qrand, rand

Purpose   Gamma random numbers

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.

randg

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 W. W. Tsang. "A Simple Method for Generating Gamma Variables." *ACM Transactions on Mathematical Software.* Vol. 26, 2000, pp. 363–372.

See Also   gamrnd

Purpose	Random numbers

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   Generate a 2-by-4 array of random values from the normal distribution with mean 0 and standard deviation 1:

```
x1 = random('Normal',0,1,2,4)
x1 =
  1.1650  0.0751  -0.6965  0.0591
```

```
   0.6268  0.3516  1.6961  1.7971
```

The order of the parameters is the same as for normrnd.

Generate a single random value from Poisson distributions with rate parameters 1, 2, ..., 6, respectively:

```
x2 = random('Poisson',1:6,1,6)
x2 =
   0   0   1   2   5   7
```

See Also   cdf, pdf, icdf, mle

gmdistribution.random

Purpose	Random numbers from Gaussian mixture distribution

Syntax	`y = random(obj)`
	`Y = random(obj,n)`
	`[Y,idx] = random(obj,n)`

Description	`y = random(obj)` generates a 1-by-d vector `y` drawn at random from the d-dimensional Gaussian mixture distribution defined by `obj`. `obj` is an object created by `gmdistribution` or `fit`.
	`Y = random(obj,n)` generates an n-by-d matrix `Y` of n d-dimensional random samples.
	`[Y,idx] = random(obj,n)` also returns an n-by-1 vector `idx`, where `idx(I)` is the index of the component used to generate `Y(I,:)`.
Examples	Create a `gmdistribution` object defining a two-component mixture of bivariate Gaussian distributions:

```
MU = [1 2;-3 -5];
SIGMA = cat(3,[2 0;0 .5],[1 0;0 1]);
p = ones(1,2)/2;
obj = gmdistribution(MU,SIGMA,p);
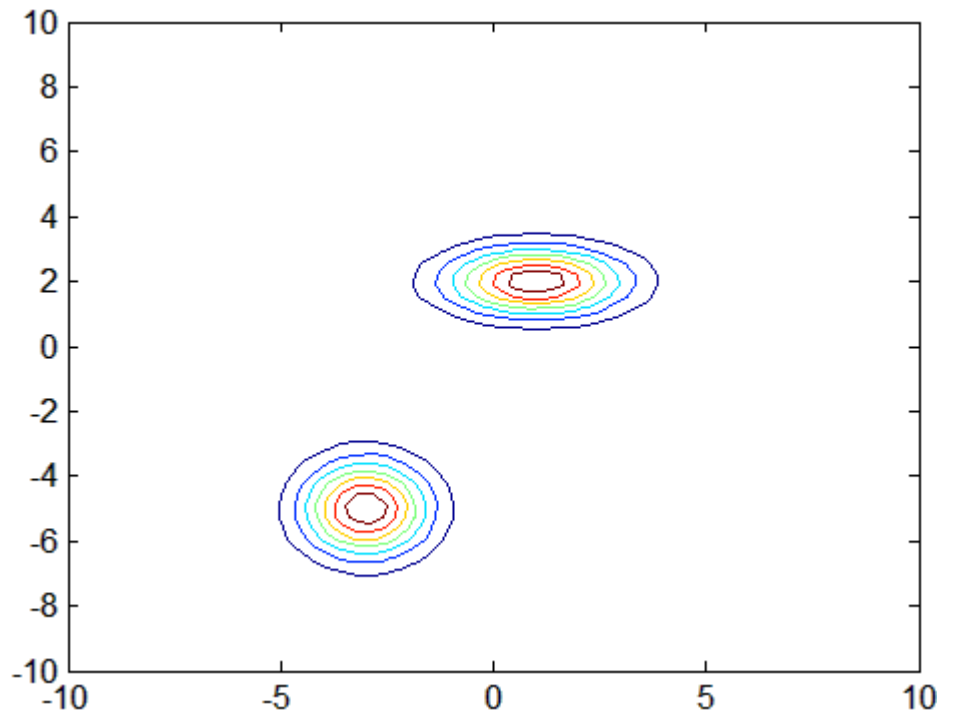
ezcontour(@(x,y)pdf(obj,[x y]),[-10 10],[-10 10])
hold on
```

Generate 1000 random values:

```
Y = random(obj,1000);

scatter(Y(:,1),Y(:,2),10,'.')
```

gmdistribution.random

See Also   gmdistribution, fit, mvnrnd

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.

Examples   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);
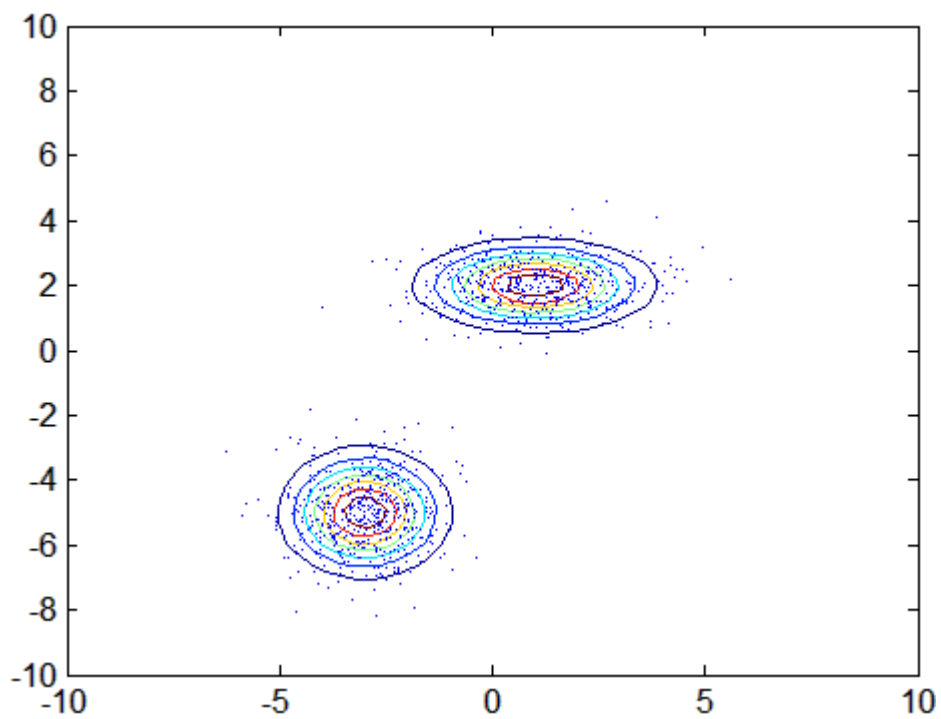
r = random(obj)
r =
    0.8285
```

See Also   paretotails, cdf, icdf

ProbDist.random

Purpose	Generate random number drawn from ProbDist object

Syntax	`Y = random(PD)`
	`Y = random(PD, N)`
	`Y = random(PD, N, M, ...)`

Description	`Y = random(PD)` generates a random number drawn from the distribution specified by `PD`, a ProbDist object. `Y = random(PD, N)` generates an `N`-by-`N` array of random numbers drawn from the distribution specified by `PD`, a ProbDist object. `Y = random(PD, N, M, ...)` generates an `N`-by-`M`-by... array of random numbers drawn from the distribution specified by `PD`, a ProbDist object.

Inputs

`PD`	An object of the class `ProbDistUnivParam` or `ProbDistUnivKernel`.
`N`	A positive integer.
`M`	A positive integer.

Outputs

`Y`	A random number drawn from the distribution specified by `PD`.

See Also   `random`

Purpose   Random sample

Syntax
```
y = randsample(n,k)
y = randsample(population,k)
y = randsample(...,replace)
y = randsample(...,true,w)
```

Description   `y = randsample(n,k)` returns a k-by-1 vector y of values sampled uniformly at random, without replacement, from the integers 1 to n.

`y = randsample(population,k)` returns a vector of values sampled uniformly at random, without replacement, from the values in the vector population. The orientation of y (row or column) is the same as 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.

Examples   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`

randtool

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.

randtool

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   Range of values

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 NaNs 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.

Examples   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

ranksum

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)

Examples   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.* New York: Marcel Dekker, 1985.

[2] Hollander, M., and D. A. Wolfe. *Nonparametric Statistical Methods.* Hoboken, NJ: John Wiley & Sons, Inc., 1999.

See Also   kruskalwallis, signrank, signtest, ttest2

raylcdf

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

$$y = F(x|b) = \int_0^x \frac{t}{b^2} e^{\left(\frac{-t^2}{2b^2}\right)} dt$$

Examples

```
x = 0:0.1:3;
p = raylcdf(x,1);
plot(x,p)
```


References [1] Evans, M., N. Hastings, and B. Peacock. *Statistical Distributions*.
Hoboken, NJ: Wiley-Interscience, 2000. pp. 134–136.

See Also `cdf`, `raylpdf`, `raylinv`, `raylstat`, `raylfit`, `raylrnd`

Purpose   Rayleigh parameter estimates

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   mle, raylpdf, raylcdf, raylinv, raylstat, raylrnd

raylinv

Purpose	Rayleigh inverse 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.

Examples

```
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

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

Examples
```
x = 0:0.1:3;
p = raylpdf(x,1);
plot(x,p)
```


See Also   pdf, raylcdf, raylinv, raylstat, raylfit, raylrnd

raylrnd

Purpose	Rayleigh random numbers

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`.

Examples
```
r = raylrnd(1:5)
r =
   1.7986   0.8795   3.3473   8.9159   3.5182
```

See Also   `random`, `raylpdf`, `raylcdf`, `raylinv`, `raylstat`, `raylfit`

Purpose   Rayleigh mean and variance

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

                    $$\frac{4-\pi}{2}b^2$$

Examples   ```
                    [mn,v] = raylstat(1)
                    mn =
                      1.2533
                    v =
                      0.4292
                    ```
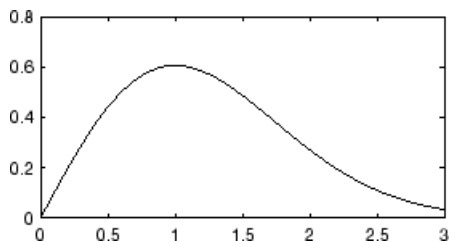
See Also   raylpdf, raylcdf, raylinv, raylfit, raylrnd

rcoplot

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.

Examples   The following plots residuals and prediction intervals from a regression
                    of a linearly additive model to the data in moore.mat:

```
load moore
X = [ones(size(moore,1),1) moore(:,1:5)];
y = moore(:,6);
alpha = 0.05;
[betahat,Ibeta,res,Ires,stats] = regress(y,X,alpha);
rcoplot(res,Ires)
```
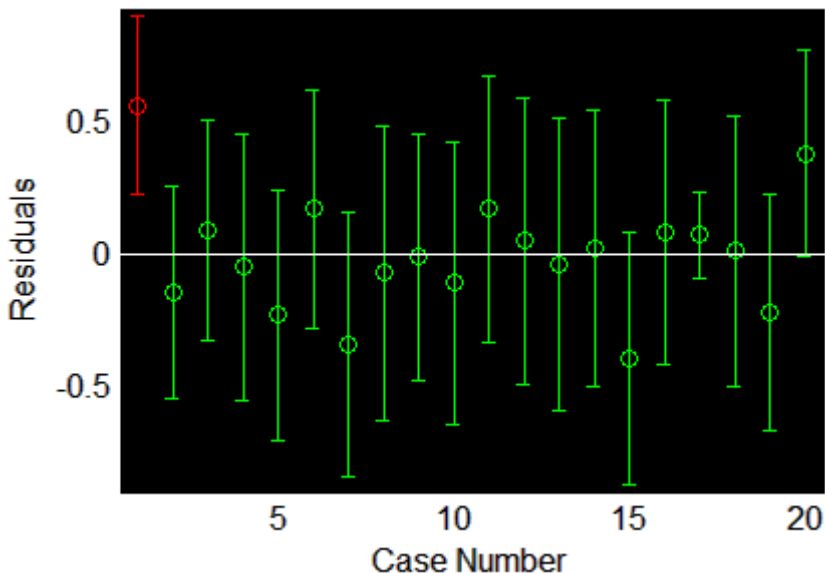
The interval around the first residual, shown in red, does not contain zero. This indicates that the residual is larger than expected in 95% of new observations, and suggests the data point is an outlier.

See Also   `regress`

refcurve

Purpose	Add reference curve to plot

Syntax
```
refcurve(p)
refcurve
hcurve = refcurve(...)
```

Description   refcurve(p) adds a polynomial reference curve with coefficients p to the current axes. If p is a vector with n+1 elements, the curve is:

```
y = p(1)*x^n + p(2)*x^(n-1) + ... + p(n)*x + p(n+1)
```

refcurve with no input arguments adds a line along the *x* axis.

hcurve = refcurve(...) returns the handle hcurve to the curve.

Examples   **Example 1**

Plot data from a population with a polynomial trend and use refcurve to add both the population and fitted mean functions:

```
p = [1 -2 -1 0];
t = 0:0.1:3;
y = polyval(p,t) + 0.5*randn(size(t));

plot(t,y,'ro')
h = refcurve(p);
set(h,'Color','r')

q = polyfit(t,y,3);
refcurve(q)
legend('Data','Population Mean','Fitted Mean','Location','NW')
```
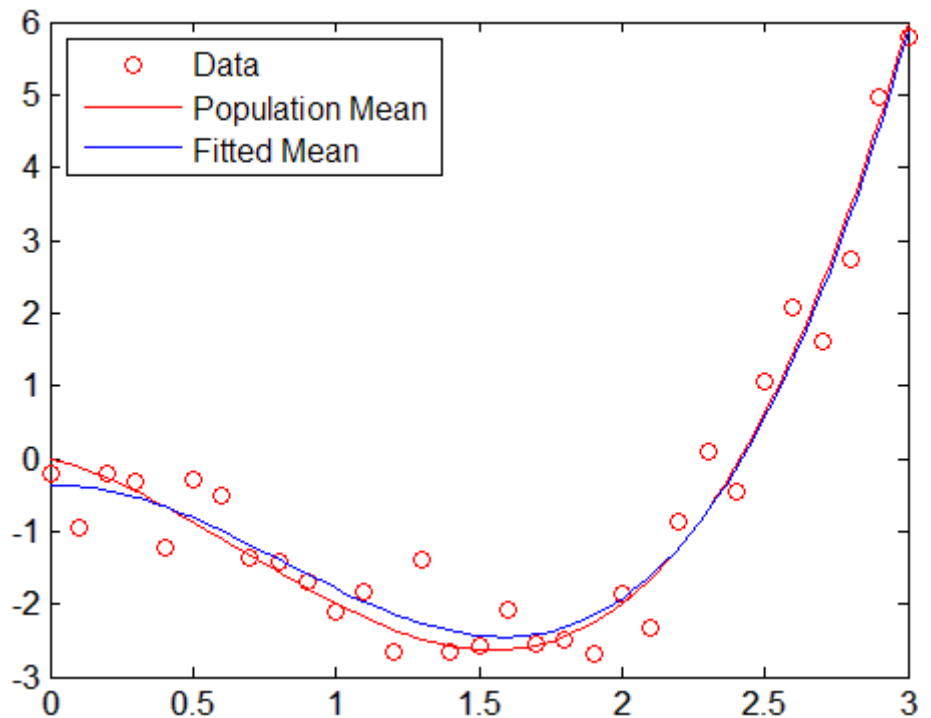
Example 2

Plot trajectories of a batted baseball, with and without air resistance.

Relevant physical constants are:

```
M = 0.145;      % Mass (kg)
R = 0.0366;     % Radius (m)
A = pi*R^2;     % Area (m^2)
rho = 1.2;      % Density of air (kg/m^3)
C = 0.5;        % Drag coefficient
D = rho*C*A/2;  % Drag proportional to the square of the speed
g = 9.8;        % Acceleration due to gravity (m/s^2)
```

refcurve

First, simulate the trajectory with drag proportional to the square of the speed, assuming constant acceleration in each time interval:

```
dt = 1e-2;       % Simulation time interval (s)
r0 = [0 1];      % Initial position (m)
s0 = 50;         % Initial speed (m/s)
alpha0 = 35;     % Initial angle (deg)
v0 = s0*[cosd(alpha0) sind(alpha0)]; % Initial velocity (m/s)

r = r0;
v = v0;
trajectory = r0;
while r(2) > 0
    a = [0 -g]-(D/M)*norm(v)*v;
    v = v + a*dt;
    r = r + v*dt + (1/2)*a*(dt^2);
    trajectory = [trajectory;r];
end
```

Second, use `refcurve` to add the drag-free parabolic trajectory (found analytically) to a plot of `trajectory`:

```
plot(trajectory(:,1),trajectory(:,2),'m','LineWidth',2)
xlim([0,250])
h = refcurve([-g/(2*v0(1)^2),...
              (g*r0(1)/v0(1)^2)+(v0(2)/v0(1)),...
              (-g*r0(1)^2/(2*v0(1)^2))-(v0(2)*r0(1)/v0(1))+r0(2)]);
set(h,'Color','c','LineWidth',2)
axis equal
ylim([0,50])
grid on
xlabel('Distance (m)')
ylabel('Height (m)')
title('{\bf Baseball Trajectories}')
legend('With Drag','Without Drag')
```

Baseball Trajectories
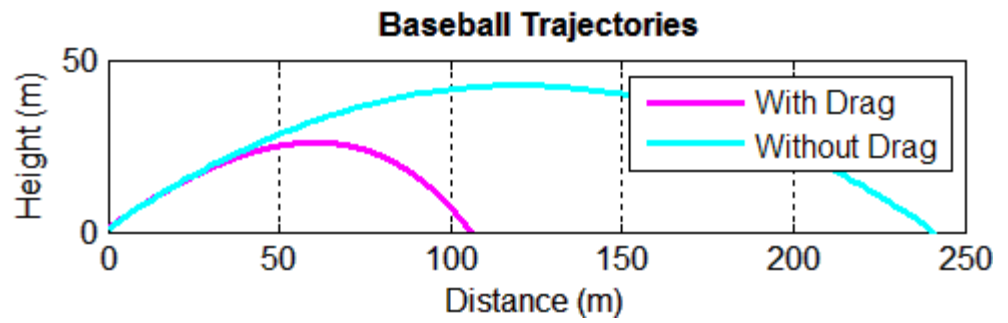
See Also   refline, lsline, gline, polyfit

refline

Purpose	Add reference line to plot

Syntax	`refline(m,b)`
	`refline(coeffs)`
	`refline`
	`hline = refline(...)`

Description	`refline(m,b)` adds a reference line with slope m and intercept b to the current axes.

`refline(coeffs)`, where coeffs is a two-element coefficient vector, adds the line

```
y = coeffs(1)*x + coeffs(2)
```

to the figure.

`refline` with no input arguments is equivalent to `lsline`.

`hline = refline(...)` returns the handle hline to the line.

Examples   Add a reference line at the mean of a data scatter and its least-squares line:

```
x = 1:10;

y = x + randn(1,10);
scatter(x,y,25,'b','*')

lsline

mu = mean(y);
hline = refline([0 mu]);
set(hline,'Color','r')
```
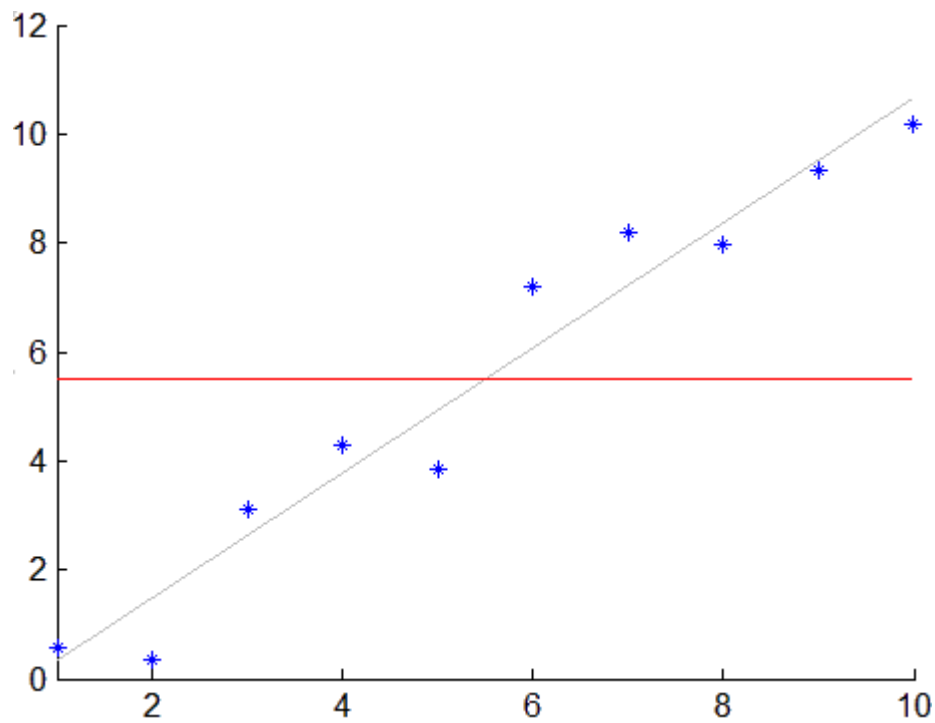
See Also   refcurve, lsline, gline

regress

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 NaNs 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, the corresponding residual is larger than expected in 95% of new observations, suggesting an outlier.

In a linear model, observed values of `y` are random variables, and so are their residuals. Residuals have normal distributions with zero mean but with different variances at different values of the predictors. To put residuals on a comparable scale, they are "Studentized," that is, they are divided by an estimate of their standard deviation that is independent of their value. Studentized residuals have t distributions with known

degrees of freedom. The intervals returned in rint are shifts of the 95% confidence intervals of these t distributions, centered at the residuals.

[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 1s 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 rint.

Examples   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)
```
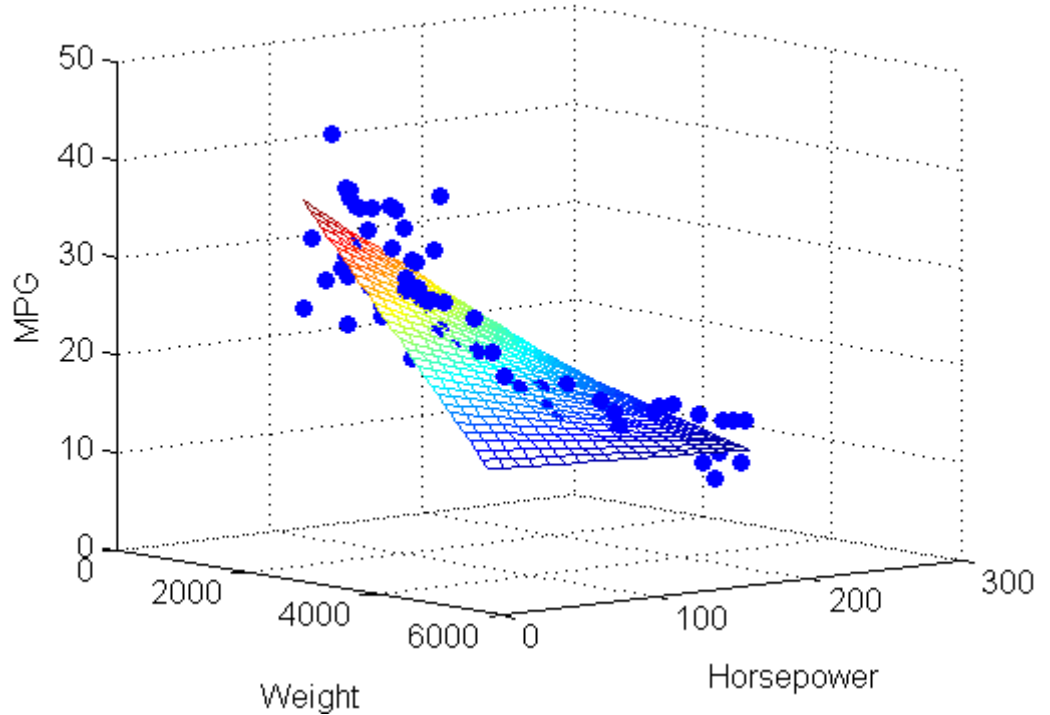
References   [1] Chatterjee, S., and A. S. Hadi. "Influential Observations, High Leverage Points, and Outliers in Linear Regression." *Statistical Science*. Vol. 1, 1986, pp. 379–416.

See Also   `regstats`, `mvregress`, `robustfit`, `stepwisefit`, `rcoplot`

regstats

Purpose	Regression diagnostics

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.

Note By default, regstats adds a first column of 1s to X, corresponding to a constant term in the model. Do not enter a column of 1s directly into X.

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.

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 listed in the following table.

Field	Description
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

Field	Description
dffits	Scaled change in fitted values
covratio	Change in covariance
cookd	Cook's distance
tstat	t statistics for coefficients
fstat	F statistic
dwstat	Durbin-Watson 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.

Examples　　Open the regstats GUI using data from hald.mat:

```
load hald
regstats(heat,ingredients,'linear');
```

Select **Fitted Values** and **Residuals** in the GUI:

regstats

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;
```

References
[1] Belsley, D. A., E. Kuh, and R. E. Welsch. *Regression Diagnostics.* Hoboken, NJ: John Wiley & Sons, Inc., 1980.

[2] Chatterjee, S., and A. S. Hadi. "Influential Observations, High Leverage Points, and Outliers in Linear Regression." *Statistical Science.* Vol. 1, 1986, pp. 379–416.

[3] Cook, R. D., and S. Weisberg. *Residuals and Influence in Regression.* New York: Chapman & Hall/CRC Press, 1983.

[4] Goodall, C. R. "Computation Using the QR Decomposition." *Handbook in Statistics.* Vol. 9, Amsterdam: Elsevier/North-Holland, 1993.

See Also   x2fx, regress, stepwise, leverage

Purpose   Value of `'Regularize'` parameter

Description   The value of the parameter `'Regularize'`.

Note This property applies only to gmdistribution objects constructed with `fit`.

---

categorical.reorderlevels

Purpose   Reorder levels

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 2-D
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.

Examples   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, getlabels, islevel,mergelevels

Purpose   Repartition data for cross-validation

Syntax   `cnew = repartition(c)`

Description   `cnew = repartition(c)` constructs an object `cnew` of the `cvpartition` class defining a random partition of the same type as `c`, where `c` is also an object of the `cvpartition` class.

Repartitioning is useful for Monte-Carlo repetitions of cross-validation analyses. `repartition` is called by `crossval` when the `'mcreps'` parameter is specified.

Examples   Partition and repartition 100 observations for 3-fold cross-validation:

```
c = cvpartition(100,'kfold',3)
c =
K-fold cross validation partition
             N: 100
   NumTestSets: 3
     TrainSize: 67  66  67
      TestSize: 33  34  33

cnew = repartition(c)
cnew =
K-fold cross validation partition
             N: 100
   NumTestSets: 3
     TrainSize: 67  66  67
      TestSize: 33  34  33
```

Check for equality of the test data in the first fold:

```
isequal(test(c,1),test(cnew,1))
ans =
     0
```

See Also   `cvpartition`

dataset.replacedata

Purpose   Replace 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`.

Examples   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

categorical.repmat

Purpose	Replicate and tile categorical array

Syntax

```
B = repmat(A,m,n)
B = repmat(A,[m n p ...])
```

Description   B = repmat(A,m,n) creates a large array B consisting of an m-by-n tiling of copies of the categorical array A. The size of B is [size(A,1)*M size(A,2)*N, size(A,3), ...]. repmat(A,n) creates an n-by-n tiling.

B = repmat(A,[m n p ...]) tiles the categorical array A to produce a multidimensional array B composed of copies of A. The size of B is [size(A,1)*M, size(A,2)*N, size(A,3)*P, ...].

See Also   ndims, size

Purpose   Reset state

Syntax   reset(q)

Description   reset(q) resets the state of the quasi-random number stream q of the
                  qrandstream class back to its initial state, 1. Subsequent points drawn
                  from the stream will be the same as those drawn from a new stream.
                  The command is equivalent to q.State = 1.

Examples   Use qrandstream to construct a 3-D Halton stream, based on a point
                  set that skips the first 1000 values and then retains every 101st point:

```
q = qrandstream('halton',3,'Skip',1e3,'Leap',1e2)
q =
   Halton quasi-random stream in 3 dimensions
   Point set properties:
               Skip : 1000
               Leap : 100
     ScrambleMethod : none

nextIdx = q.State
nextIdx =
     1
```

Use qrand to generate two samples of size four:

```
X1 = qrand(q,4)
X1 =
    0.0928    0.3475    0.0051
    0.6958    0.2035    0.2371
    0.3013    0.8496    0.4307
    0.9087    0.5629    0.6166
nextIdx = q.State
nextIdx =
     5

X2 = qrand(q,4)
```

```
X2 =
    0.2446    0.0238    0.8102
    0.5298    0.7540    0.0438
    0.3843    0.5112    0.2758
    0.8335    0.2245    0.4694
nextIdx = q.State
nextIdx =
     9
```

Use reset to reset the stream, then generate another sample:

```
reset(q)
nextIdx = q.State
nextIdx =
     1

X = qrand(q,4)
X =
    0.0928    0.3475    0.0051
    0.6958    0.2035    0.2371
    0.3013    0.8496    0.4307
    0.9087    0.5629    0.6166
```

See Also   qrandstream, qrand

Purpose   Resize categorical array

Syntax   B = reshape(A,M,N)
                   B = reshape(A,m,n,p,...)
                   reshape(A,[m n p ...])
                   B = reshape(A,...,[],...)

Description   B = reshape(A,M,N) returns an m-by-n categorical matrix whose
                   elements are taken columnwise from the categorical array A. An error
                   results if A does not have m*n elements.

                   B = reshape(A,m,n,p,...) or reshape(A,[m n p ...]) returns
                   an array with the same elements as A but reshaped to have the size
                   m-by-n-by-p-by-... . m*n*p*... must be the same as numel(A).

                   B = reshape(A,...,[],...) calculates the length of the dimension
                   represented by[], such that the product of the dimensions equals
                   numel(A). numel(A) must be evenly divisible by the product of the
                   known dimensions. You can use only one occurrence of [].

                   In general, reshape(A,siz) returns an array with the same elements
                   as A but reshaped to the size siz. prod(siz) must be the same as
                   numel(A).

See Also   shiftdim, squeeze

ridge

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 1s.

`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*b1_scaled; b1_scaled]
```

This can be seen by replacing the x_i ($i = 1, ..., n$) in the multilinear model $y = b_0{}^0 + b_1{}^0 x_1 + ... + b_n{}^0 x_n$ 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.

Coefficient estimates for multiple linear regression models rely on the independence of the model terms. When terms are correlated and the columns of the design matrix X have an approximate linear dependence, the matrix $(X^TX)^{-1}$ becomes close to singular. As a result, the least-squares estimate

$$\hat{\beta} = (X^TX)^{-1}X^Ty$$

becomes highly sensitive to random errors in the observed response y, producing a large variance. This situation of *multicollinearity* can arise, for example, when data are collected without an experimental design.

Ridge regression addresses the problem by estimating regression coefficients using

$$\hat{\beta} = (X^TX + kI)^{-1}X^Ty$$

where k is the *ridge parameter* and I is the identity matrix. Small positive values of k 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.

Examples   Load the data in `acetylene.mat`, with observations of the predictor variables `x1`, `x2`, `x3`, and the response variable `y`:

```
load acetylene
```

Plot the predictor variables 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
```
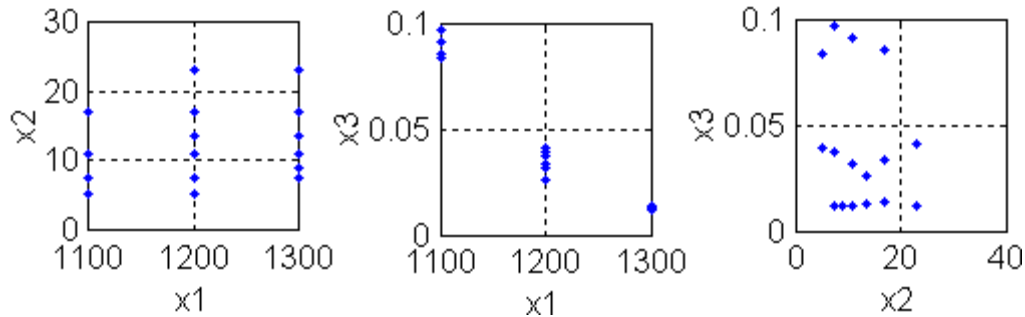

Note the correlation between x1 and the other two predictor variables.

Use ridge and x2fx to 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('x1','x2','x3','x1x2','x1x3','x2x3')
```
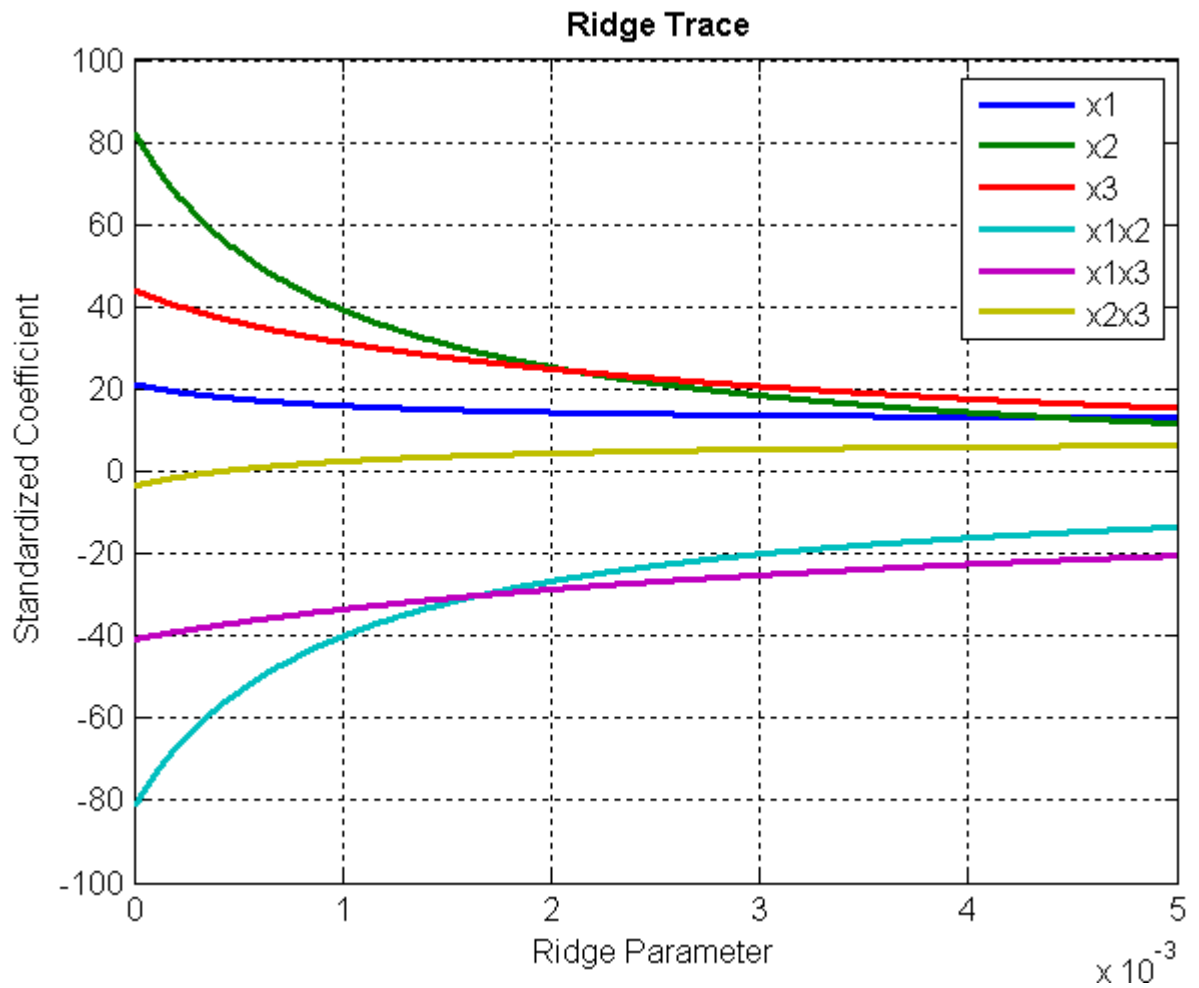
The estimates stabilize to the right of the plot. Note that the coefficient of the x2x3 interaction term changes sign at a value of the ridge parameter $\approx 5 \times 10^{-4}$.

ridge

References   [1] Hoerl, A. E., and R. W. Kennard. "Ridge Regression: Biased Estimation for Nonorthogonal Problems." *Technometrics*. Vol. 12, No. 1, 1970, pp. 55–67.

[2] Hoerl, A. E., and R. W. Kennard. "Ridge Regression: Applications to Nonorthogonal Problems." *Technometrics*. Vol. 12, No. 1, 1970, pp. 69–82.

[3] Marquardt, D.W. "Generalized Inverses, Ridge Regression, Biased Linear Estimation, and Nonlinear Estimation." *Technometrics*. Vol. 12, No. 3, 1970, pp. 591–612.

[4] Marquardt, D. W., and R.D. Snee. "Ridge Regression in Practice." *The American Statistician*. Vol. 29, No. 1, 1975, pp. 3–20.

See Also   regress, stepwise

Purpose   Node risks

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.

Examples   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)
```
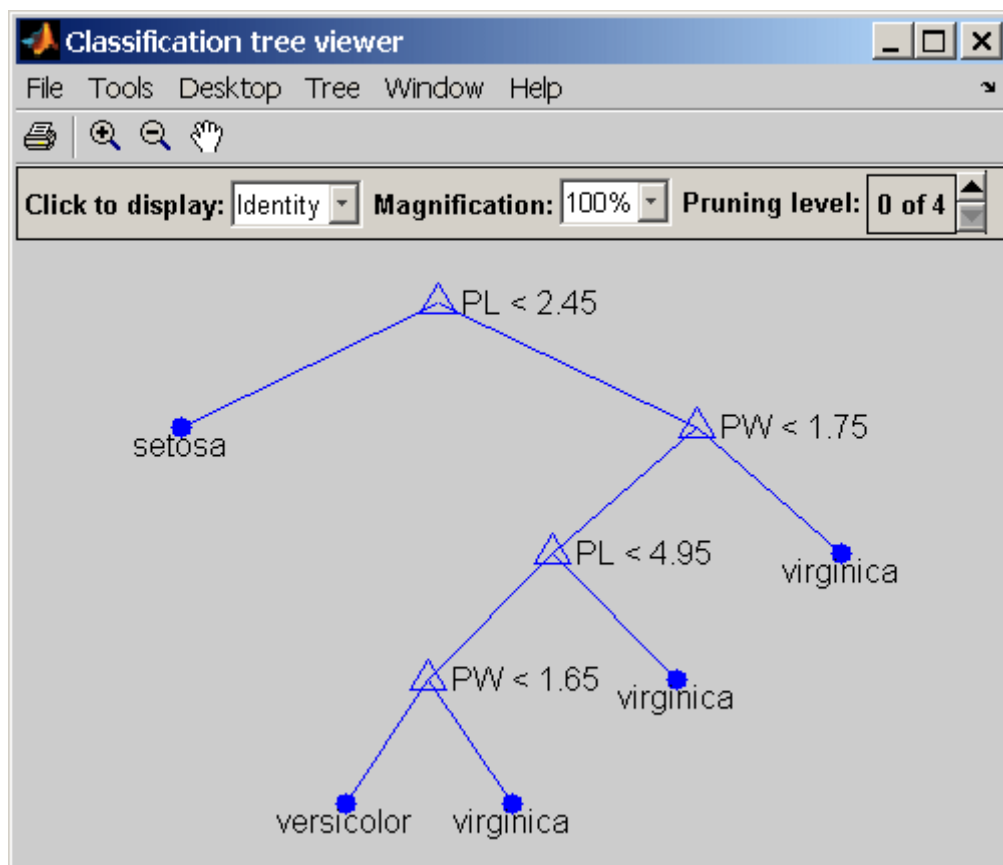
```
e = nodeerr(t);
p = nodeprob(t);
r = risk(t);

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
```

References   [1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees.* Boca Raton, FL: CRC Press, 1984.

See Also   classregtree, nodeerr, nodeprob

Purpose	Interactive robust regression

Syntax

```
robustdemo
robustdemo(x,y)
```

Description robustdemo shows the difference between ordinary least squares and robust regression for data with a single predictor. With no input arguments, robustdemo displays a scatter plot of a sample of roughly linear data with one outlier. The bottom of the figure displays equations of lines fitted to the data using ordinary least squares and robust methods, together with estimates of the root mean squared errors.

Use the right mouse button to click on a point and view its least-squares leverage and robust weight.

Use the left mouse button to click-and-drag a point. The displays will update.

robustdemo(x,y) uses x and y data vectors you supply, in place of the sample data supplied with the function.

Examples The following steps show you how to use robustdemo.

 1 **Start the demo.** To begin using robustdemo with the built-in data, simply type the function name:

```
robustdemo
```

Least squares: Y = -0.188327 + 1.10351*X   RMS error = 2.21375

Robust: Y = -1.77278 + 1.50415*X   RMS error = 1.43663

The resulting figure shows a scatter plot with two fitted lines. The red line is the fit using ordinary least-squares regression. The green line is the fit using robust regression. At the bottom of the figure are the equations for the fitted lines, together with the estimated root mean squared errors for each fit.

2 View leverages and robust weights. Right-click on any data point to see its least-squares leverage and robust weight:

robustdemo

Least squares: Y = -0.188327 + 1.10351*X   RMS error = 2.21375

Robust: Y = -1.77278 + 1.50415*X   RMS error = 1.43663

In the built-in data, the right-most point has a relatively high leverage of 0.35. The point exerts a large influence on the least-squares fit, but its small robust weight shows that it is effectively excluded from the robust fit.

3 See how changes in the data affect the fits. With the left mouse button, click and hold on any data point and drag it to a new location. When you release the mouse button, the displays update:

Least squares: Y = -0.928677 + 1.30648*X   RMS error = 1.28809

Robust: Y = -1.07823 + 1.34094*X   RMS error = 1.34891

Bringing the right-most data point closer to the least-squares line makes the two fitted lines nearly identical. The adjusted right-most data point has significant weight in the robust fit.

See Also   `robustfit`, `leverage`

robustfit

Purpose   Robust 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 1s to X,
corresponding to a constant term in the model. Do not enter a column of
1s directly into X. You can change the default behavior of robustfit
using the input *const*, below.

---

robustfit treats NaNs 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

```
r = 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 1s 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.

Examples   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')
```
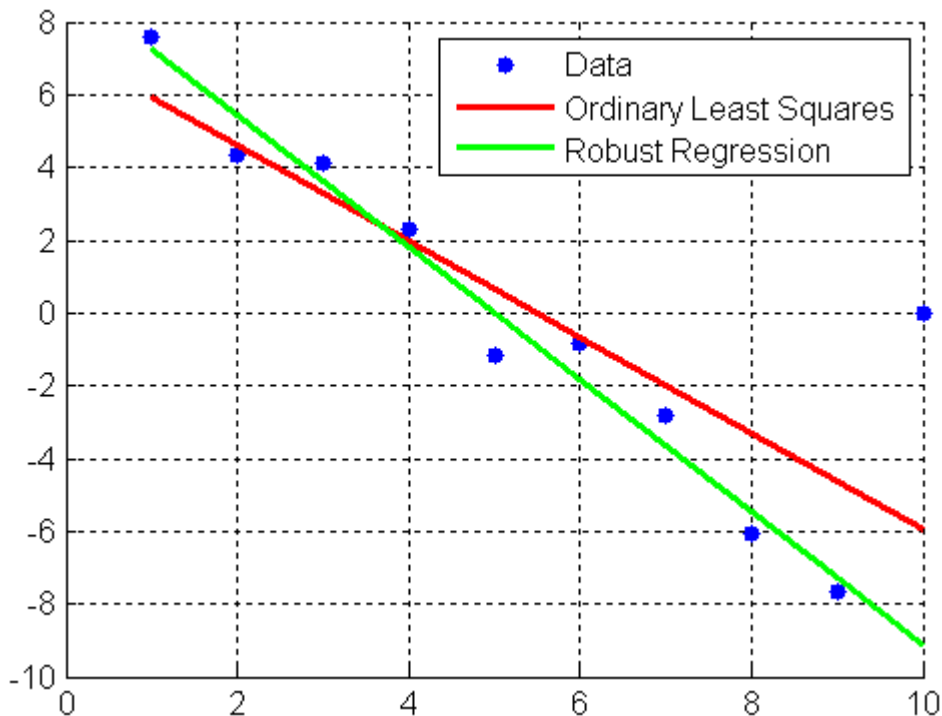
References [1] DuMouchel, W. H., and F. L. O'Brien. "Integrating a Robust Option into a Multiple Regression Computing Environment." *Computer Science and Statistics*: *Proceedings of the 21st Symposium on the Interface.* Alexandria, VA: American Statistical Association, 1989.

[2] Holland, P. W., and 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*. Hoboken, NJ: John Wiley & Sons, Inc., 1981.

[4] Street, J. O., R. J. Carroll, and D. Ruppert. "A Note on Computing Robust Regression Estimates via Iteratively Reweighted Least Squares." *The American Statistician.* Vol. 42, 1988, pp. 152–154.

See Also   regress, robustdemo

categorical.rot90

Purpose	Rotate categorical matrix 90 degrees
Syntax	`B = rot90(A)` `B = rot90(A,k)`
Description	`B = rot90(A)` returns the 90 degree counterclockwise rotation of the 2-D categorical matrix A.
	`B = rot90(A,k)` returns the k*90 degree rotation of A, k = +-1,+-2,....
See Also	`flipdim`, `fliplr`, `flipud`

Purpose   Rotate factor 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 *i* th variable
on the *j* th 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);

% Default (normalized varimax) rotation:
% first three principle components.
LPC = princomp(X);
[L1,T] = rotatefactors(LPC(:,1:3));

% Equamax rotation:
% first three principle components.
[L2,T] = rotatefactors(LPC(:,1:3),...
                        'method','equamax');

% Promax rotation:
% first three factors.
LFA = factoran(X,3,'Rotate','none');
[L3,T] = rotatefactors(LFA(:,1:3),...
                        'method','promax',...
                        'power',2);

% Pattern rotation:
% first three factors.
Tgt = [1 1 1 1 1 0 1 0 1 1; ...
       0 0 0 1 1 1 0 0 0 0; ...
       1 0 0 1 0 1 1 1 1 0]';
[L4,T] = rotatefactors(LFA(:,1:3),...
                        'method','pattern',...
                        'target',Tgt);
inv(T'*T) % Correlation matrix of the rotated factors
```

References

[1] Harman, H. H. *Modern Factor Analysis*. 3rd ed. Chicago: University of Chicago Press, 1976.

[2] Lawley, D. N., and A. E. Maxwell. *Factor Analysis as a Statistical Method.* 2nd ed. New York: American Elsevier Publishing, 1971.

See Also   biplot, factoran, princomp, pcacov, procrustes

Purpose   Row exchange

Syntax
```
dRE = rowexch(nfactors,nruns)
[dRE,X] = rowexch(nfactors,nruns)
[dRE,X] = rowexch(nfactors,nruns,model)
[dRE,X] = rowexch(...,param1,val1,param2,val2,...)
```

Description   `dRE = rowexch(nfactors,nruns)` uses a row-exchange algorithm to
generate a *D*-optimal design `dRE` with `nruns` runs (the rows of `dRE`) for a
linear additive model with `nfactors` factors (the columns of `dRE`). The
model includes a constant term.

`[dRE,X] = rowexch(nfactors,nruns)` also returns the associated
design matrix `X`, whose columns are the model terms evaluated at each
treatment (row) of `dRE`.

`[dRE,X] = rowexch(nfactors,nruns,model)` uses the linear
regression model specified in *model*. *model* is one of the following
strings:

- `'linear'` — Constant and linear terms. This is the default.

- `'interaction'` — Constant, linear, and interaction terms

- `'quadratic'` — Constant, linear, interaction, and squared terms

- `'purequadratic'` — Constant, linear, and squared terms

The order of the columns of `X` for a full quadratic model with *n* terms is:

1 The constant term

2 The linear terms in order 1, 2, ..., *n*

3 The interaction terms 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 factor and one row for each term in the model. The entries in any row of *model* are powers for the factors in the columns. For example, if a model has factors X1, X2, and X3, then a row [0 1 2] in *model* specifies the term (X1.^0).*(X2.^1).*(X3.^2). A row of all zeros in *model* specifies a constant term, which can be omitted.

[dRE,X] = rowexch(...,*param1*,*val1*,*param2*,*val2*,...) specifies additional parameter/value pairs for the design. Valid parameters and their values are listed in the following table.

Parameter	Value
'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 the iteration counter. The default is 'on'.
'excludefun'	Handle to a function that excludes undesirable runs. If the function is *f*, it must support the syntax *b* = *f*(*S*), where *S* is a matrix of treatments with nfactors columns and *b* is a vector of Boolean values with the same number of rows as *S*. *b*(*i*) is true if the *i*th row *S* should be excluded.
'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.

Parameter	Value
`'maxiter'`	Maximum number of iterations. The default is 10.
`'tries'`	Number of times to try to generate a design from a new starting point. The algorithm uses random points for each try, except possibly the first. The default is 1.

Algorithm   Both `cordexch` and `rowexch` use iterative search algorithms. They operate by incrementally changing an initial design matrix X to increase $D = |X^TX|$ at each step. In both algorithms, there is randomness built into the selection of the initial design and into the choice of the incremental changes. As a result, both algorithms may return locally, but not globally, D-optimal designs. Run each algorithm multiple times and select the best result for your final design. Both functions have a `'tries'` parameter that automates this repetition and comparison.

At each step, the row-exchange algorithm exchanges an entire row of X with a row from a design matrix C evaluated at a candidate set of feasible treatments. The `rowexch` function automatically generates a C appropriate for a specified model, operating in two steps by calling the `candgen` and `candexch` functions in sequence. Provide your own C by calling `candexch` directly. In either case, if C is large, its static presence in memory can affect computation.

Examples   Suppose you want a design to estimate the parameters in the following three-factor, seven-term interaction model:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \beta_{12} x_1 x_2 + \beta_{13} x_1 x_3 + \beta_{23} x_2 x_3 + \varepsilon$$

Use `rowexch` to generate a D-optimal design with seven runs:

```
nfactors = 3;
nruns = 7;
[dRE,X] = rowexch(nfactors,nruns,'interaction','tries',10)
dRE =
    -1    -1     1
```

```
              1    -1     1
              1    -1    -1
              1     1     1
             -1    -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    -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
```

Columns of the design matrix X are the model terms evaluated at each row of the design dRE. The terms appear in order from left to right: constant term, linear terms (1, 2, 3), interaction terms (12, 13, 23). Use X to fit the model, as described in "Linear Regression" on page 8-3, to response data measured at the design points in dRE.

See Also   candgen, candexch, cordexch

Purpose   Interactive response surface demonstration

Syntax   rsmdemo

Description   rsmdemo opens a group of three graphical user interfaces for
               interactively investigating response surface methodology (RSM),
               nonlinear fitting, and the design of experiments.

               The interfaces allow you to collect and model data from a simulated
               chemical reaction. Experimental predictors are concentrations of three
               reactants (hydrogen, n-Pentane, and isopentane) and the response is
               the reaction rate. The reaction rate is simulated by a Hougen-Watson
               model (Bates and Watts, [2], pp. 271–272):

               $$rate = \frac{\beta_1 x_2 - x_3 / \beta_5}{1 + \beta_2 x_1 + \beta_3 x_2 + \beta_4 x_3}$$

               where *rate* is the reaction rate, x_1, x_2, and x_3 are the concentrations of
               hydrogen, n-pentane, and isopentane, respectively, and β_1, β_2, ... , β_5 are
               fixed parameters. Random errors are used to perturb the reaction rate
               for each combination of reactants.

               Collect data using one of two methods:

               1 Manually set reactant concentrations in the **Reaction Simulator**
                 interface by editing the text boxes or by adjusting the associated
                 sliders.

rsmdemo
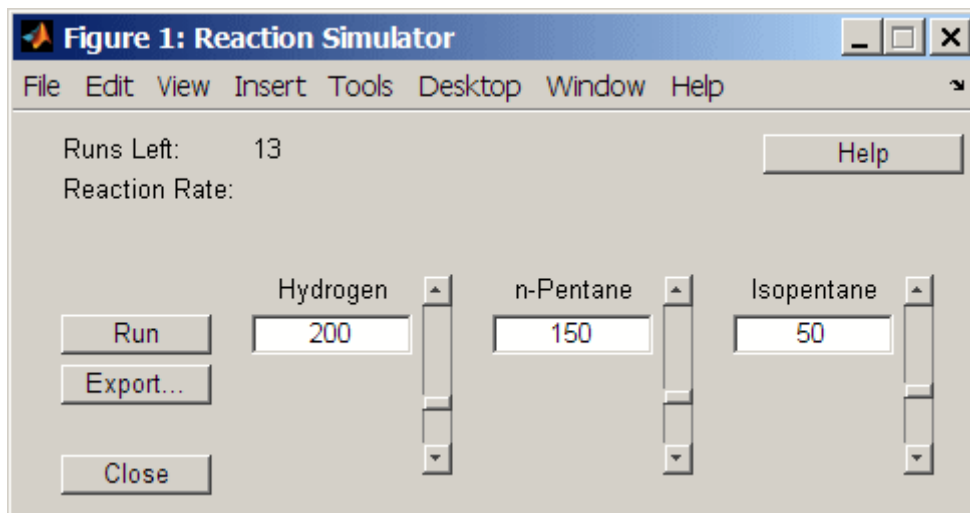
When you click **Run**, the concentrations and simulated reaction rate are recorded on the **Trial and Error Data** interface.
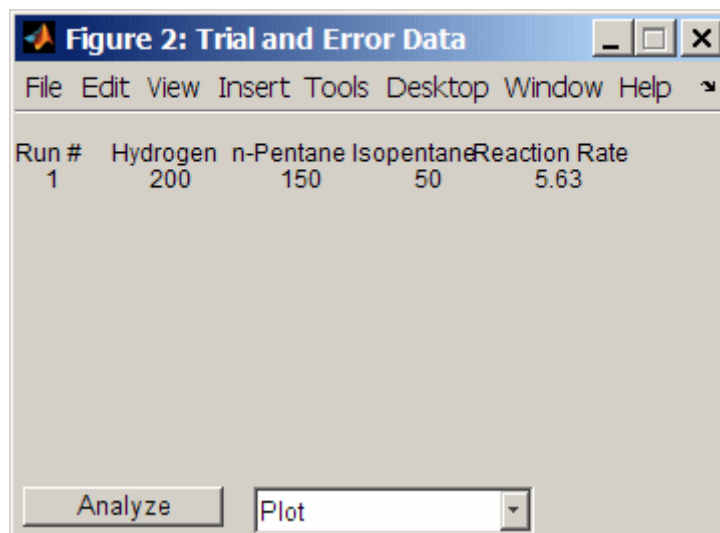
You are allowed up to 13 independent experimental runs for data collection.

2 Use a designed experiment to set reactant concentrations in the **Experimental Data** interface by clicking the **Do Experiment** button.
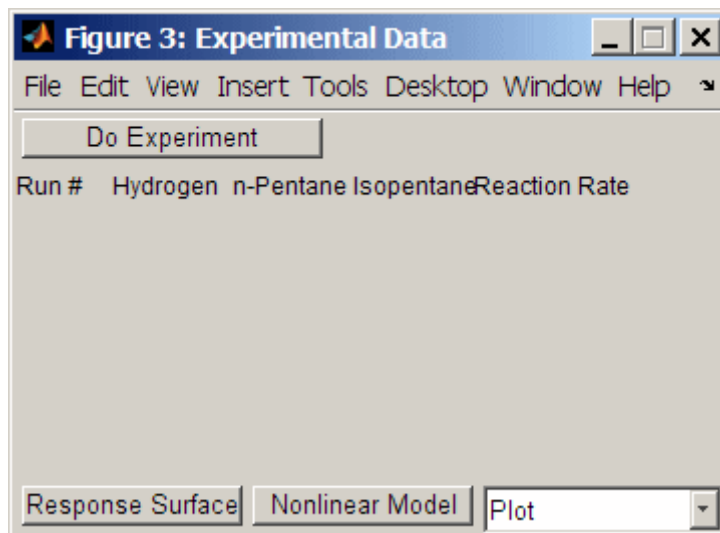
A 13-run *D*-optimal design for a full quadratic model is generated by the cordexch function, and the concentrations and simulated reaction rates are recorded on the same interface.
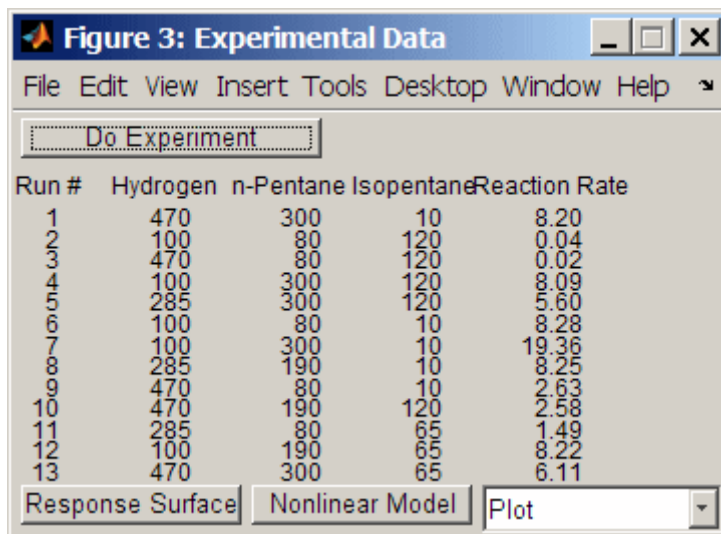
Once data is collected, scatter plots of reaction rates vs. individual predictors are generated by selecting one of the following from the **Plot** pop-up menu below the recorded data:

- **Hydrogen vs. Rate**
- **n-Pentane vs. Rate**
- **Isopentane vs. Rate**

Fit a response surface model to the data by clicking the **Analyze** button below the trial-and-error data or the **Response Surface** button below the experimental data. Both buttons load the data into the Response Surface Tool `rstool`. By default, trial-and-error data is fit with a linear additive model and experimental data is fit with a full quadratic model, but the models can be adjusted in the Response Surface Tool.

For experimental data, you have the additional option of fitting a Hougen-Watson model. Click the **Nonlinear Model** button to load the data and the model in `hougen` into the Nonlinear Fitting Tool `nlintool`.

See Also   hougen, cordexch, rstool, nlintool

Purpose   Interactive response surface modeling

Syntax
```
rstool
rstool(X,Y,model)
rstool(x,y,model,alpha)
rstool(x,y,model,alpha,xname,yname)
```

Description   `rstool` opens a graphical user interface for interactively investigating one-dimensional contours of multidimensional response surface models.
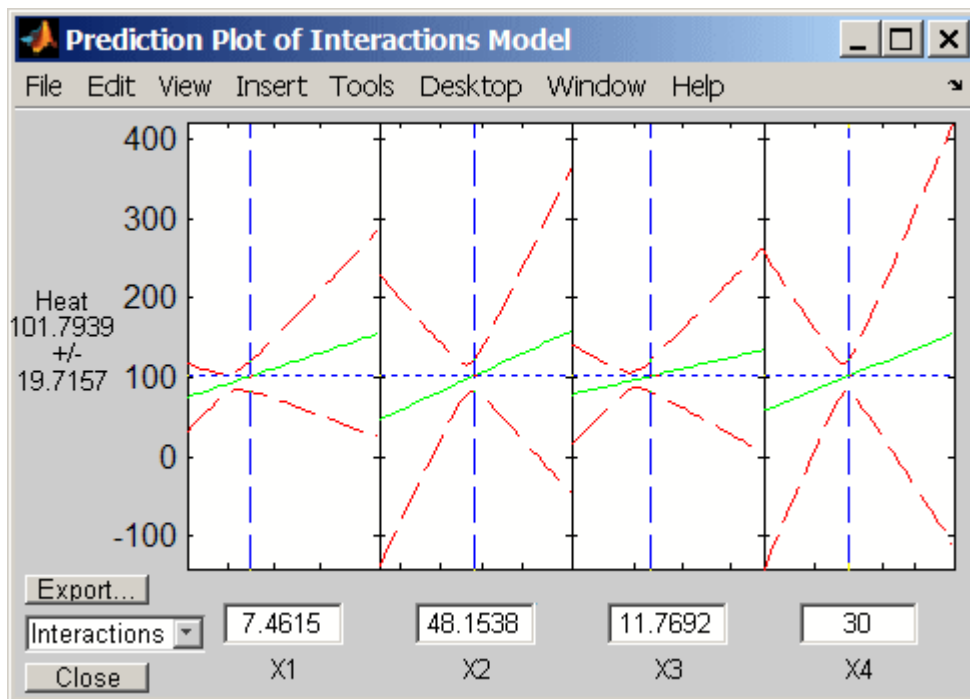
By default, the interface opens with the data from `hald.mat` and a fitted response surface with constant, linear, and interaction terms.

A sequence of plots is displayed, each showing a contour of the response surface against a single predictor, with all other predictors held fixed.

95% global confidence intervals for new observations are shown as dashed red curves above and below the response. Predictor values are displayed in the text boxes on the horizontal axis and are marked by vertical dashed blue lines in the plots. Predictor values are changed by editing the text boxes or by dragging the dashed blue lines. When you change the value of a predictor, all plots update to show the new point in predictor space.

The pop-up menu at the lower left of the interface allows you to choose among the following models:

- `Linear` — Constant and linear terms (the default)

- `Pure Quadratic` — Constant, linear, and squared terms

- `Interactions` — Constant, linear, and interaction terms

- `Full Quadratic` — Constant, linear, interaction, and squared terms

Click **Export** to open the following dialog box:
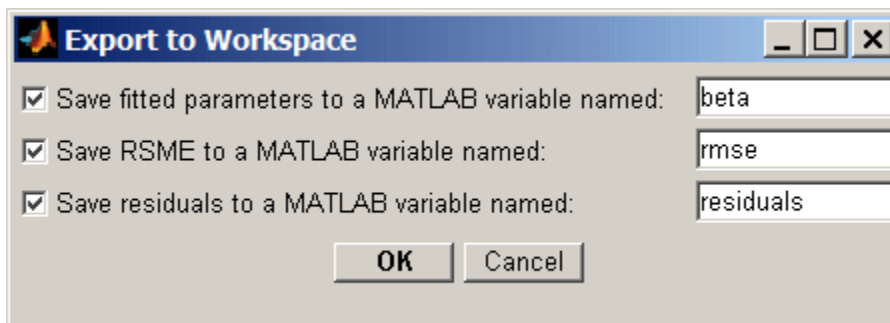
The dialog allows you to save information about the fit to MATLAB workspace variables with valid names.

`rstool(X,Y,model)` opens the interface with the predictor data in X, the response data in Y, and the fitted model *model*. 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.

The optional input *model* can be any one of the following strings:

- `'linear'` — Constant and linear terms (the default)
- `'purequadratic'` — Constant, linear, and squared terms
- `'interaction'` — Constant, linear, and interaction terms
- `'quadratic'` — Constant, linear, interaction, 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.

rstool(x,y,*model*,alpha) uses 100(1-alpha)% global confidence intervals for new observations in the plots.

rstool(x,y,*model*,alpha,xname,yname) labels the axes using the strings in xname and yname. To label each subplot differently, xname and yname can be cell arrays of strings.

Examples The following uses rstool to visualize a quadratic response surface model of the 3-D chemical reaction data in reaction.mat:

```
load reaction
alpha = 0.01; % Significance level
rstool(reactants,rate,'quadratic',alpha,xn,yn)
```
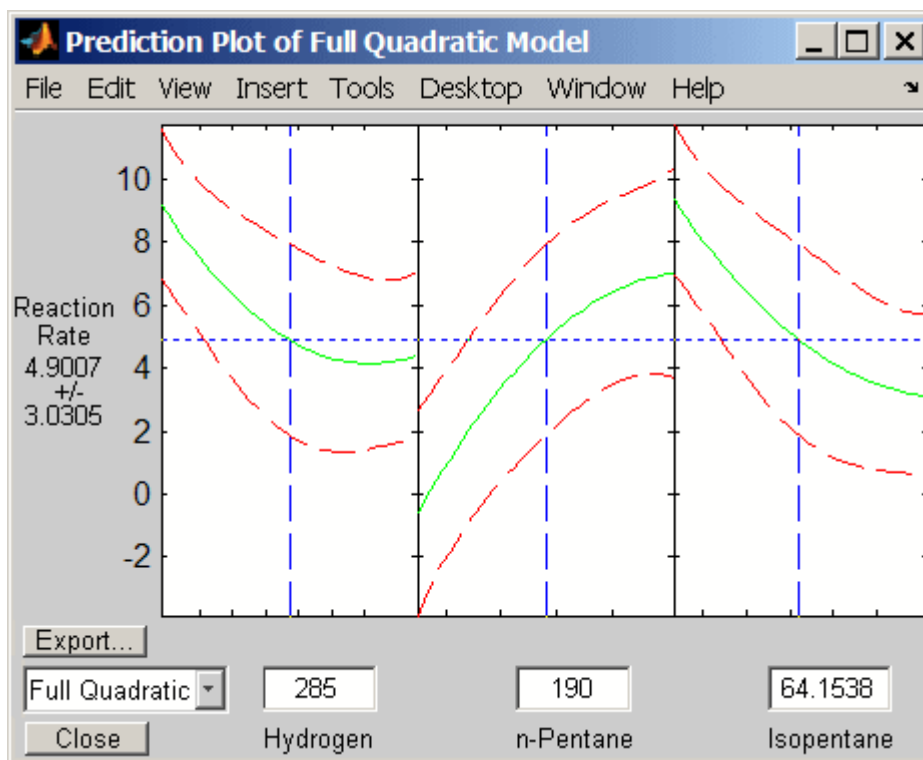
The rstool interface is used by rsmdemo to visualize the results of simulated experiments with data like that in reaction.mat. As described in "Response Surface Designs" on page 13-9, rsmdemo uses a response surface model to generate simulated data at combinations of predictors specified by either the user or by a designed experiment.

See Also   x2fx, rsmdemo, nlintool

runstest

Purpose	Run 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

Examples
```
x = randn(40,1);
[h,p] = runstest(x,median(x))
h =
     0
p =
    0.6286
```

See Also   signrank, signtest

TreeBagger.SampleWithReplacement property

Purpose   Flag to sample with replacement

Description   The `SampleWithReplacement` property is a logical flag specifying if data are sampled for each decision tree with replacement. True if `TreeBagger` samples data with replacement and false otherwise. True by default.

Purpose   Sample size and power of 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

Examples   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],[],0.8,60)
mu1 =
   103.6770
```

Compute the sample size *n* required to distinguish $p = 0.26$ from $p = 0.6$ 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',0.2,0.26,0.6)
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',0.2,0.26,[],nn);
nexact = min(nn(pwr>=0.6))
nexact =
   213

plot(nn,pwr,'b-',[napprox nexact],pwr([napprox nexact]),'ro');
grid on
```
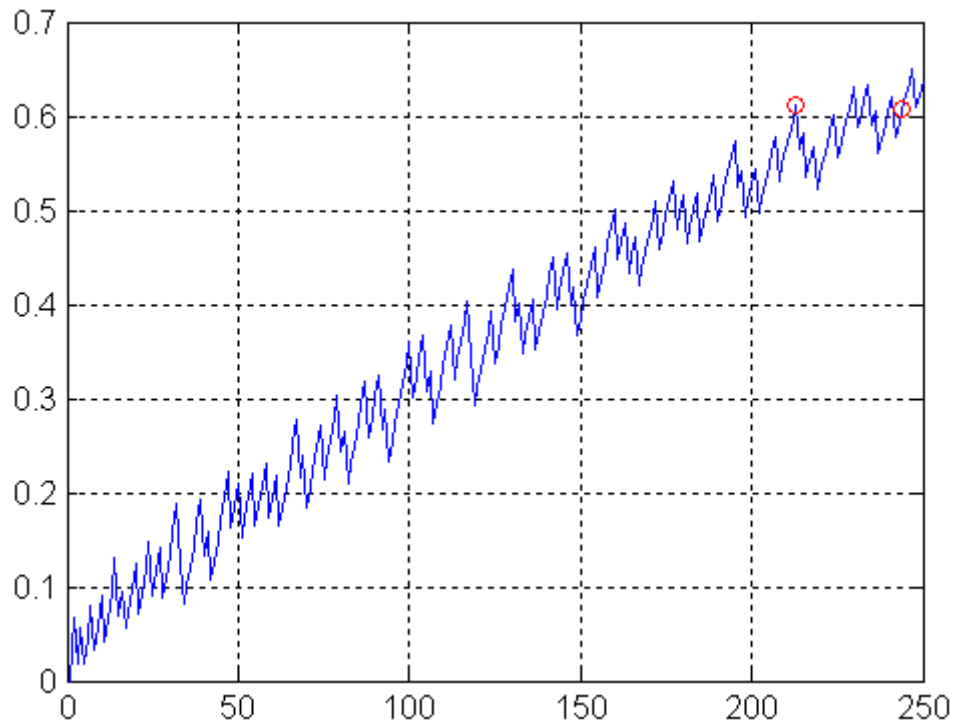
See Also   vartest, ttest, ztest, binocdf

Purpose   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))
```
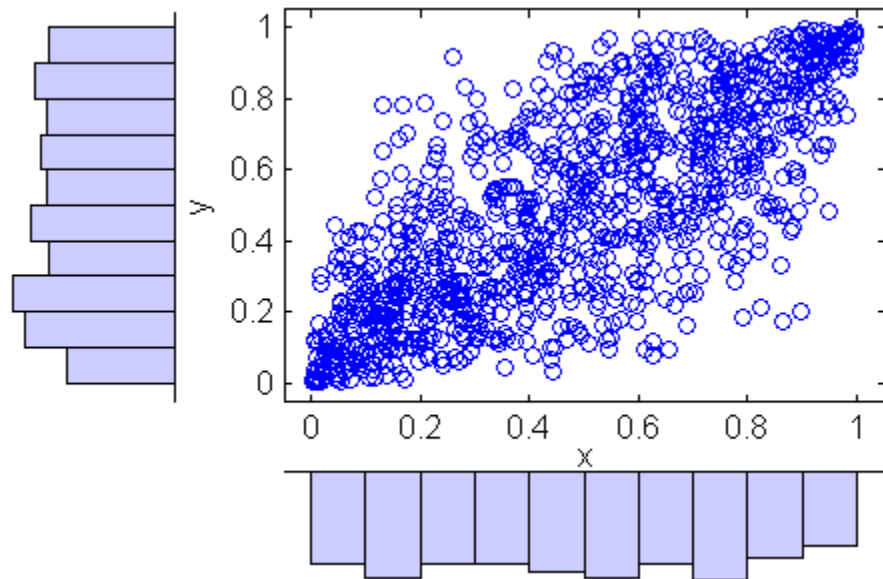
Example 3

Mixed discrete and continuous data:

```
cars = load('carsmall');
scatterhist(cars.Weight,cars.Cylinders,[10 3])
```
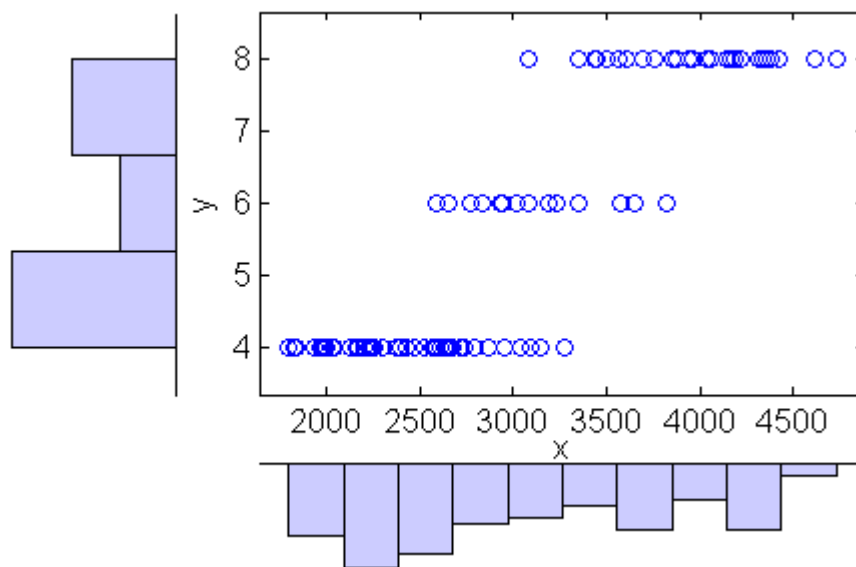
See Also   scatter, hist

Purpose   Scramble quasi-random point set

Syntax   ps = scramble(p,*type*)
                ps = scramble(p,'clear')
                ps = scramble(p)

Description ps = scramble(p,*type*) returns a scrambled copy ps of the point set p
                 of the qrandset class, created using the scramble type specified in the
                 string *type*. Point sets from different subclasses of qrandset support
                 different scramble types, as indicated in the following table.

Subclass	Scramble Types
haltonset class	'RR2' — A permutation of the radical inverse coefficients derived by applying a reverse-radix operation to all of the possible coefficient values. The scramble is described in [1].
sobolset	'MatousekAffineOwen' — A random linear scramble combined with a random digital shift. The scramble is described in [2]

ps = scramble(p,'clear') removes all scramble settings from p and
returns the result in ps.

ps = scramble(p) removes all scramble settings from p and then adds
them back in the order they were originally applied. This typically
results in a different point set because of the randomness of the
scrambling algorithms.

Examples   Use haltonset to generate a 3-D Halton point set, skip the first 1000
                values, and then retain every 101st point:

```
p = haltonset(3,'Skip',1e3,'Leap',1e2)
p =
    Halton point set in 3 dimensions (8.918019e+013 points)
    Properties:
            Skip : 1000
```

```
                         Leap : 100
              ScrambleMethod : none
```

Use `scramble` to apply reverse-radix scrambling:

```
p = scramble(p,'RR2')
p =
    Halton point set in 3 dimensions (8.918019e+013 points)
    Properties:
                Skip : 1000
                Leap : 100
      ScrambleMethod : RR2
```

Use `net` to generate the first four points:

```
X0 = net(p,4)
X0 =
    0.0928    0.6950    0.0029
    0.6958    0.2958    0.8269
    0.3013    0.6497    0.4141
    0.9087    0.7883    0.2166
```

Use parenthesis indexing to generate every third point, up to the 11th point:

```
X = p(1:3:11,:)
X =
    0.0928    0.6950    0.0029
    0.9087    0.7883    0.2166
    0.3843    0.9840    0.9878
    0.6831    0.7357    0.7923
```

References

[1] Kocis, L., and W. J. Whiten. "Computational Investigations of Low-Discrepancy Sequences." *ACM Transactions on Mathematical Software*. Vol. 23, No. 2, 1997, pp. 266–294.

[2] Matousek, J. "On the L2-Discrepancy for Anchored Boxes." *Journal of Complexity*. Vol. 14, No. 4, 1998, pp. 527–556.

See Also   haltonset, sobolset

qrandset.ScrambleMethod property

Purpose   Scramble settings

Description   A structure with fields `Type` and `Options`, giving the scramble settings. Set this property using the `scramble` method. Supported values for `Type` and `Options` depend on the subclass.

piecewisedistribution.segment

Purpose Segments containing 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 nonempty, S is determined by comparing X with the quantile boundary values defined for obj. If P is nonempty, S is determined by comparing P with the probability boundary values.

Examples 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

sequentialfs

Purpose   Sequential feature selection

Syntax
```
inmodel = sequentialfs(fun,X,y)
inmodel = sequentialfs(fun,X,Y,Z,...)
[inmodel,history] = sequentialfs(fun,X,...)
[] = sequentialfs(...,param1,val1,param2,val2,...)
```

Description   `inmodel = sequentialfs(fun,X,y)` selects a subset of features from
the data matrix `X` that best predict the data in `y` by sequentially
selecting features until there is no improvement in prediction. Rows
of `X` correspond to observations; columns correspond to variables or
features. `y` is a column vector of response values or class labels for each
observation in `X`. `X` and `y` must have the same number of rows. `fun` is a
function handle to a function that defines the criterion used to select
features and to determine when to stop. The output `inmodel` is a logical
vector indicating which features are finally chosen.

Starting from an empty feature set, `sequentialfs` creates candidate
feature subsets by sequentially adding each of the features not yet
selected. For each candidate feature subset, `sequentialfs` performs
10-fold cross-validation by repeatedly calling `fun` with different training
subsets of `X` and `y`, `XTRAIN` and `ytrain`, and test subsets of `X` and `y`,
`XTEST` and `ytest`, as follows:

```
criterion = fun(XTRAIN,ytrain,XTEST,ytest)
```

`XTRAIN` and `ytrain` contain the same subset of rows of `X` and `Y`, while
`XTEST` and `ytest` contain the complementary subset of rows. `XTRAIN`
and `XTEST` contain the data taken from the columns of `X` that correspond
to the current candidate feature set.

Each time it is called, `fun` must return a scalar value `criterion`.
Typically, `fun` uses `XTRAIN` and `ytrain` to train or fit a model, then
predicts values for `XTEST` using that model, and finally returns some
measure of distance, or *loss*, of those predicted values from `ytest`.
In the cross-validation calculation for a given candidate feature set,
`sequentialfs` sums the values returned by `fun` and divides that sum

by the total number of test observations. It then uses that mean value to evaluate each candidate feature subset.

Typical loss measures include sum of squared errors for regression models (`sequentialfs` computes the mean-squared error in this case), and the number of misclassified observations for classification models (`sequentialfs` computes the misclassification rate in this case).

Note `sequentialfs` divides the sum of the values returned by `fun` across all test sets by the total number of test observations. Accordingly, `fun` should not divide its output value by the number of test observations.

After computing the mean `criterion` values for each candidate feature subset, `sequentialfs` chooses the candidate feature subset that minimizes the mean criterion value. This process continues until adding more features does not decrease the criterion.

`inmodel = sequentialfs(fun,X,Y,Z,...)` allows any number of input variables X, Y, Z, ... `sequentialfs` chooses features (columns) only from X, but otherwise imposes no interpretation on X, Y, Z, ... All data inputs, whether column vectors or matrices, must have the same number of rows. `sequentialfs` calls `fun` with training and test subsets of X, Y, Z, ... as follows:

```
criterion = fun(XTRAIN,YTRAIN,ZTRAIN,...,
                XTEST,YTEST,ZTEST,...)
```

`sequentialfs` creates XTRAIN, YTRAIN, ZTRAIN, ... , XTEST, YTEST, ZTEST, ... by selecting subsets of the rows of X, Y, Z, ... `fun` must return a scalar value `criterion`, but may compute that value in any way. Elements of the logical vector `inmodel` correspond to columns of X and indicate which features are finally chosen.

`[inmodel,history] = sequentialfs(fun,X,...)` returns information on which feature is chosen at each step. `history` is a scalar structure with the following fields:

- `Crit` — A vector containing the criterion values computed at each step.

- `In` — A logical matrix in which row `i` indicates the features selected at step `i`.

`[] = sequentialfs(...,`*param1*`,`*val1*`,`*param2*`,`*val2*`,...)` specifies optional parameter name/value pairs from the following table.

Parameter	Value
`'cv'`	The validation method used to compute the criterion for each candidate feature subset.
	• When the value is a positive integer `k`, `sequentialfs` uses `k`-fold cross-validation without stratification.
	• When the value is an object of the `cvpartition` class, other forms of cross-validation can be specified.
	• When the value is `'resubstitution'`, the original data are passed to `fun` as both the training and test data to compute the criterion.
	• When the value is `'none'`, `sequentialfs` calls `fun` as `criterion = fun(X,Y,Z,...)`, without separating test and training sets.
	The default value is `10`, that is, 10-fold cross-validation without stratification.
	So-called *wrapper methods* use a function `fun` that implements a learning algorithm. These methods usually apply cross-validation to select features. So-called *filter methods* use a function `fun` that measures characteristics of the data (such as correlation) to select features.

Parameter	Value
`'mcreps'`	A positive integer indicating the number of Monte-Carlo repetitions for cross-validation. The default value is 1. The value must be 1 if the value of `'cv'` is `'resubstitution'` or `'none'`.
`'direction'`	The direction of the sequential search. The default is `'forward'`. A value of `'backward'` specifies an initial candidate set including all features and an algorithm that removes features sequentially until the criterion increases.
`'keepin'`	A logical vector or a vector of column numbers specifying features that must be included. The default is empty.
`'keepout'`	A logical vector or a vector of column numbers specifying features that must be excluded. The default is empty.
`'nfeatures'`	The number of features at which `sequentialfs` should stop. `inmodel` includes exactly this many features. The default value is empty, indicating that `sequentialfs` should stop when a local minimum of the criterion is found. A nonempty value overrides values of `'MaxIter'` and `'TolFun'` in `'options'`.
`'nullmodel'`	A logical value, indicating whether or not the null model (containing no features from X) should be included in feature selection and in the `history` output. The default is `false`.

sequentialfs

Parameter	Value
'options'	Options structure for the iterative sequential search algorithm, as created by statset. sequentialfs uses the following statset parameters: • Display — Amount of information displayed by the algorithm. The default is 'off'. • MaxIter — Maximum number of iterations allowed. The default is Inf. • TolFun — Termination tolerance for the objective function value. The default is 1e-6 if 'direction' is 'forward'; 0 if 'direction' is 'backward'. • TolTypeFun — Use absolute or relative objective function tolerances. The default is 'rel'.

Examples

Perform sequential feature selection for classification of noisy features:

```
load fisheriris;
X = randn(150,10);
X(:,[1 3 5 7 ])= meas;
y = species;

c = cvpartition(y,'k',10);
opts = statset('display','iter');
fun = @(XT,yT,Xt,yt)...
      (sum(~strcmp(yt,classify(Xt,XT,yT,'quadratic')))); 

[fs,history] = sequentialfs(fun,X,y,'cv',c,'options',opts)

Start forward sequential feature selection:
Initial columns included: none
```

```
Columns that can not be included: none
Step 1, added column 7, criterion value 0.04
Step 2, added column 5, criterion value 0.0266667
Final columns included:   5 7

fs =
    0  0  0  0  1  0  1  0  0  0
history =
      In: [2x10 logical]
    Crit: [0.0400 0.0267]

history.In
ans =
    0  0  0  0  0  0  1  0  0  0
    0  0  0  0  1  0  1  0  0  0
```

See Also   crossval, cvpartition, stepwisefit, statset

Purpose   Set and display properties

Syntax
```
set(A)
set(A,PropertyName)
A = set(A,PropertyName,PropertyValue,...)
B = set(A,PropertyName,value)
```

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.

`B = set(A,PropertyName,value)` returns a dataset array B that is a copy of A, but with the property `'PropertyName'` set to the value `value`.

Note Using `set(A,'PropertyName',value)` without assigning to a variable does not modify A's properties. Use `A = set(A,'PropertyName',value)` to modify A.

Examples   Create a dataset array from Fisher's iris data and add a description:

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

CompactTreeBagger.SetDefaultYfit

Purpose   Set default value for `predict`

Syntax   B = SetDefaultYfit(B,Yfit)

Description `B = SetDefaultYfit(B,Yfit)` sets the default prediction for ensemble
                 `B` to `Yfit`. The default prediction must be a character variable for
                 classification or a numeric scalar for regression. This setting controls
                 what predicted value `CompactTreeBagger` returns when no prediction
                 is possible, for example when the `predict` method needs to predict
                 for an observation which has only false values in the matrix supplied
                 through `'useifort'` argument.

See Also   `DefaultYfit`, `predict`, `TreeBagger.DefaultYfit`

Purpose   Set difference for categorical arrays

Syntax   C = setdiff(A,B)
                   [C,I] = setdiff(A,B)

Description   C = setdiff(A,B) when A and B are categorical arrays returns a
                   categorical vector C containing the values in A that are not in B. The
                   result C is sorted. The set of categorical levels for C is the sorted union
                   of the sets of levels of the inputs, as determined by their labels.

                   [C,I] = setdiff(A,B) also returns index vectors I such that C =
                   A(I).

See Also   intersect, ismember, setxor, union, unique

categorical.setlabels

Purpose   Label levels

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 2-D 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 2-D 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`

categorical.setxor

Purpose	Set exclusive-or for categorical arrays
Syntax	C = setxor(A,B) [C,IA,IB] = setxor(A,B)
Description	C = setxor(A,B) when A and B are categorical arrays returns a categorical vector C containing the values not in the intersection of A and B. The result C is sorted. The set of categorical levels for C is the sorted union of the sets of levels of the inputs, as determined by their labels. [C,IA,IB] = setxor(A,B) also returns index vectors IA and IB such that C is a sorted combination of the elements A(IA) and B(IB).
See Also	intersect, ismember, setdiff, union, unique

Purpose   `true` if all covariance matrices are restricted to be the same

Description   Logical `true` if all the covariance matrices are restricted to be the same (pooled estimate); logical `false` otherwise.

categorical.shiftdim

Purpose	Shift dimensions of categorical array
Syntax	`B = shiftdim(A,n)` `[B,nshifts] = shiftdim(A)`
Description	`B = shiftdim(A,n)` shifts the dimensions of the categorical array `A` by N. When `n` is positive, `shiftdim` shifts the dimensions to the left and wraps the `n` leading dimensions to the end. When `n` is negative, `shiftdim` shifts the dimensions to the right and pads with singletons. `[B,nshifts] = shiftdim(A)` returns the array `B` with the same number of elements as `A` but with any leading singleton dimensions removed. `nshifts` returns the number of dimensions that are removed. If `A` is a scalar, `shiftdim` has no effect.
See Also	`circshift`, `reshape`, `squeeze`

Purpose   Input array of covariances

Description Input array of covariances `SIGMA`.

signrank

Purpose   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)

Examples

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.* New York: Marcel Dekker, 1985.

[2] Hollander, M., and D. A. Wolfe. *Nonparametric Statistical Methods.* Hoboken, NJ: John Wiley & Sons, Inc., 1999.

See Also

`ranksum`, `ttest`, `ztest`

signtest

Purpose   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)

Examples   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*. New York: Marcel Dekker, 1985.

[2] Hollander, M., and D. A. Wolfe. *Nonparametric Statistical Methods*. Hoboken, NJ: John Wiley & Sons, Inc., 1999.

See Also   `ranksum`, `signrank`, `ttest`, `ztest`

silhouette

Purpose   Silhouette plot

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 2-33.) silhouette treats NaNs 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 given in the following table.

Metric	**Description**
'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)

Metric	Description
`'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,:)))
```

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.* Hoboken, NJ: John Wiley & Sons, Inc., 1990.

silhouette

See Also   "Grouped Data" on page 2-33

dendrogram, kmeans, linkage, pdist

Purpose   Convert categorical array to single array

Syntax   B = single(A)

Description   B = single(A) converts the categorical array A to a single array. Each element of B contains the internal categorical level code for the corresponding element of A.

See Also   double

dataset.single

Purpose   Convert dataset variables to single array

Syntax   B = single(A)
                     B = single(A,vars)

Description   B = single(A) returns the contents of the dataset A, converted to one single array. The classes of the variables in the dataset must support the conversion.

                     B = single(A,vars) returns the contents of the dataset variables 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.

See Also   dataset, double, replacedata

Purpose   Size of categorical array

Syntax   d = size(A)
                 [m,n] = size(A)
                 [m1,m2,m3,...,mn] = size(A)
                 m = size(A,dim)

Description d = size(A) returns the two-element row vector d = [m,n] containing
                 the number of rows and columns in the matrix for an m-by-n categorical
                 matrix A. For n-D categorical arrays, size(A) returns a 1-by-n vector of
                 dimension lengths. Trailing singleton dimensions are ignored.

                 [m,n] = size(A) for a categorical matrix A, returns the number of
                 rows and columns in A as separate output variables.

                 [m1,m2,m3,...,mn] = size(A), for n>1, returns the sizes of the
                 first n dimensions of the categorical array A. If the number of output
                 arguments n does not equal ndims(A), then for:

n > ndims(A)	size returns ones in the "extra" variables, i.e., outputs ndims(A)+1 through n.
n < ndims(A)	mn contains the product of the sizes of dimensions n through ndims(A).

                 m = size(A,dim) returns the length of the dimension specified by the
                 scalar dim. For example, size(A,1) returns the number of rows. If
                 dim > ndims(A), m will be 1.

See Also   length, ndims, numel

dataset.size

Purpose   Size of dataset array

Syntax   B = single(A)
              B = single(A,vars)

Description   B = single(A) returns the contents of the dataset A, converted to one single array. The classes of the variables in the dataset must support the conversion.

B = single(A,vars) returns the contents of the dataset variables 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.

See Also   length, ndims, numel

Purpose   Number of dimensions in matrix

Syntax
```
d = size(p)
[m,n] = size(p)
m = size(p,dim)
```

Description   `d = size(p)` returns the two-element row vector `d = [m,n]` containing the number of points in the point set and the number of dimensions the points are in, for the point set `p`. These correspond to the number of rows and columns in the matrix that would be produced by the expression `p(:,:)`.

`[m,n] = size(p)` returns the number of points and dimensions for `p` as separate output variables.

`m = size(p,dim)` returns the length of the dimension specified by the scalar `dim`. For example, `size(p,1)` returns the number of rows (points in the point set). If `dim` is greater than 2, `m` will be 1.

Examples   The commands

```
P = sobolset(12);
d = size(P)
```

return

```
d = [9.0072e+015 12]
```

The command

```
[m,n] = size(P)
```

returns

```
m = 9.0072e+015
n = 12
```

The command

qrandset.size

```
m2 = size(P, 2)
```

returns

```
m2 = 12
```

See Also   length, ndims, qrandset

Purpose   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 kth point omitted in
                   the generated sequence. Values beyond the kth 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.

slicesample

Examples   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',[0.8 .8 1]);
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','LineWidth',2)
```
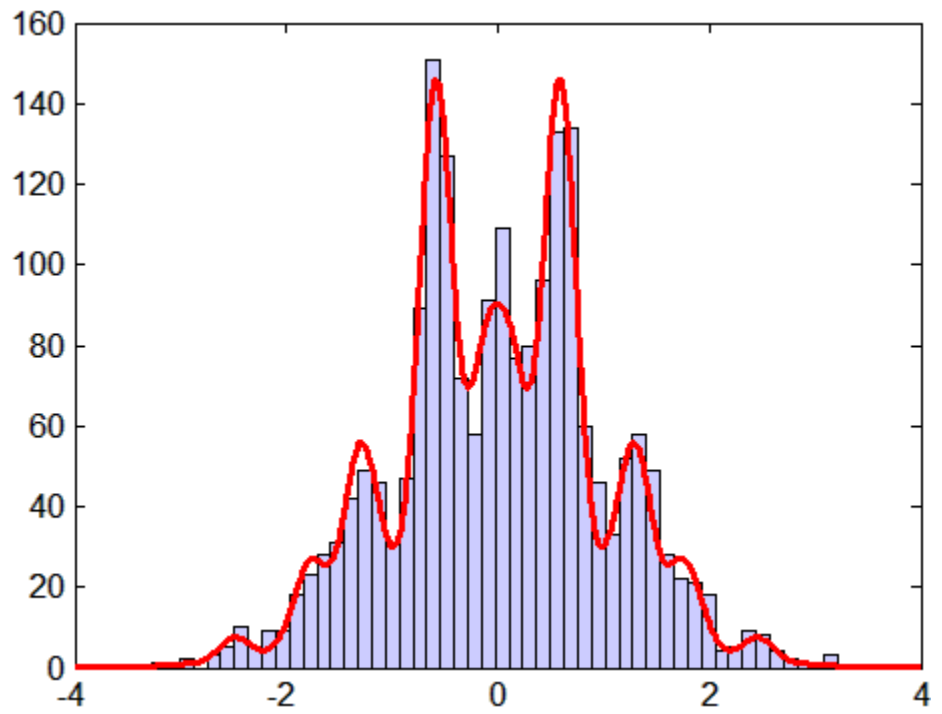
See Also   rand, mhsample, randsample

skewness

Purpose   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 NaNs 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.

Examples   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

qrandset.Skip property

Purpose   Number of initial points omitted

Description   Number of initial points omitted (positive integer).

Superclasses	qrandset	
Purpose	Sobol quasi-random point sets	

Description sobolset is a quasi-random point set class that produces points from the Sobol sequence. The Sobol sequence is a base-2 digital sequence that fills space in a highly uniform manner.

Construction	sobolset	Construct Sobol quasi-random point set

Methods **Inherited Methods**

Methods in the following table are inherited from qrandset.

	disp	Display qrandset object
	end	Last index in indexing expression for point set
	length	Length of point set
	ndims	Number of dimensions in matrix
	net	Generate quasi-random point set
	scramble	Scramble quasi-random point set
	size	Number of dimensions in matrix
	subsref	Subscripted reference for qrandset

Properties

	PointOrder	Point generation method

Inherited Properties

Properties in the following table are inherited from qrandset.

sobolset class

Dimensions	Number of dimensions
Leap	Interval between points
ScrambleMethod	Scramble settings
Skip	Number of initial points omitted
Type	Subclass

Copy Semantics

Value. To learn how this affects your use of the class, see Comparing Handle and Value Classes in the MATLAB Object-Oriented Programming documentation.

References

[1] Bratley, P., and B. L. Fox, "ALGORITHM 659 Implementing Sobol's Quasirandom Sequence Generator," ACM Transactions on Mathematical Software, Vol. 14, No. 1, pp. 88-100, 1988.

[2] Joe, S., and F. Y. Kuo, "Remark on Algorithm 659: Implementing Sobol's Quasirandom Sequence Generator," ACM Transactions on Mathematical Software, Vol. 29, No. 1, pp. 49-57, 2003.

[3] Hong, H. S., and F. J. Hickernell, "ALGORITHM 823: Implementing Scrambled Digital Sequences," ACM Transactions on Mathematical Software, Vol. 29, No. 2, pp. 95-109, 2003.

[4] Matousek, J., "On the L2-discrepancy for anchored boxes," Journal of Complexity, Vol. 14, pp. 527-556, 1998.

See Also

"Quasi-Random Point Sets" on page 5-144

haltonset

Purpose   Construct Sobol quasi-random point set

Syntax   p = sobolset(d)
p = sobolset(d,*prop1*,*val1*,*prop2*,*val2*,...)

Description   p = sobolset(d) constructs a d-dimensional point set p of the
sobolset class, with default property settings.

p = sobolset(d,*prop1*,*val1*,*prop2*,*val2*,...) specifies property
name/value pairs used to construct p.

The object p returned by sobolset encapsulates properties of a
specified quasi-random sequence. The point set is finite, with a length
determined by the Skip and Leap properties and by limits on the size
of point set indices (maximum value of 2^{53}). Values of the point set are
not generated and stored in memory until you access p using net or
parenthesis indexing.

Examples   Generate a 3-D Sobol point set, skip the first 1000 values, and then
retain every 101st point:

```
p = sobolset(3,'Skip',1e3,'Leap',1e2)
p =
    Sobol point set in 3 dimensions (8.918019e+013 points)
    Properties:
                Skip : 1000
                Leap : 100
    ScrambleMethod : none
        PointOrder : standard
```

Use scramble to apply a random linear scramble combined with a
random digital shift:

```
p = scramble(p,'MatousekAffineOwen')
p =
    Sobol point set in 3 dimensions (8.918019e+013 points)
    Properties:
                Skip : 1000
```

```
              Leap : 100
   ScrambleMethod : MatousekAffineOwen
       PointOrder : standard
```

Use net to generate the first four points:

```
X0 = net(p,4)
X0 =
    0.7601    0.5919    0.9529
    0.1795    0.0856    0.0491
    0.5488    0.0785    0.8483
    0.3882    0.8771    0.8755
```

Use parenthesis indexing to generate every third point, up to the 11th point:

```
X = p(1:3:11,:)
X =
    0.7601    0.5919    0.9529
    0.3882    0.8771    0.8755
    0.6905    0.4951    0.8464
    0.1955    0.5679    0.3192
```

References

[1] Bratley, P., and B. L. Fox. "Algorithm 659 Implementing Sobol's Quasirandom Sequence Generator." *ACM Transactions on Mathematical Software*. Vol. 14, No. 1, 1988, pp. 88–100.

[2] Joe, S., and F. Y. Kuo. "Remark on Algorithm 659: Implementing Sobol's Quasirandom Sequence Generator." *ACM Transactions on Mathematical Software*. Vol. 29, No. 1, 2003, pp. 49–57.

[3] Hong, H. S., and F. J. Hickernell. "Algorithm 823: Implementing Scrambled Digital Sequences." *ACM Transactions on Mathematical Software*. Vol. 29, No. 2, 2003, pp. 95–109.

[4] Matousek, J. "On the L2-Discrepancy for Anchored Boxes." *Journal of Complexity*. Vol. 14, No. 4, 1998, pp. 527–556.

See Also   haltonset, net, scramble

ordinal.sort

Purpose	Sort elements of 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.

Examples   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

dataset.sortrows

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,:)`.

Examples   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

Purpose   Sort rows

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 2-D 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.

Examples   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

Purpose   Format 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.

Examples
```
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`.

squareform

See Also   pdist

Purpose   Squeeze singleton dimensions from categorical array

Syntax   B = squeeze(A)

Description   B = squeeze(A) returns an array B with the same elements as the
               categorical array A but with all the singleton dimensions removed. A
               singleton is a dimension such that size(A,dim)==1. 2-D arrays are
               unaffected by squeeze so that row vectors remain rows.

See Also   shiftdim

qrandstream.State property

Purpose   Point set index

Description   Point set index of the next number to be drawn (positive integer).

Purpose   Access values in 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

statset

Purpose   Create statistics options structure

Syntax   statset
                     statset(statfun)
                     options = statset(...)
                     options = statset(*fieldname1*,*val1*,*fieldname2*,*val2*,...)
                     options = statset(oldopts,*fieldname1*,*val1*,*fieldname2*,*val2*,
                         ...)
                     options = statset(oldopts,newopts)

Description   statset with no input arguments and no output arguments displays all
                     fields of a statistics options structure and their possible values.

                     statset(statfun) displays fields and default values used by the
                     Statistics Toolbox function statfun. Specify statfun using a string
                     name or a function handle.

                     options = statset(...) creates a statistics options structure options.
                     With no input arguments, all fields of the options structure are set to
                     []. With a specified statfun, function-specific fields are set to default
                     values and the remaining fields are set to []. Function-specific fields
                     set to [] indicate that the function is to use its default value for that
                     parameter.

                     options = statset(*fieldname1*,*val1*,*fieldname2*,*val2*,...)
                     creates an options structure in which the named fields have the specified
                     values. Any unspecified values are set to []. Use strings for field names.
                     For fields that are string-valued, the complete string is required for the
                     value. If an invalid string is provided for a value, the default is used.

                     options =
                     statset(oldopts,*fieldname1*,*val1*,*fieldname2*,*val2*,...)
                     creates a copy of oldopts with the named parameters changed to
                     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 corresponding
                     parameters in oldopts.

Options   The following table lists the fields of a statistics options structure, the parameters they contain, and their possible values.

Field	Parameter	Values
DerivStep	Relative difference used in finite difference derivative calculations.	A positive scalar, or a vector of positive scalars the same size as the vector of parameters estimated by the Statistics Toolbox function using the options structure.
Display	Amount of information displayed by the algorithm.	• 'off' — Displays no information. • 'final' — Displays the final output. • 'iter' — Displays iterative output to the command window for some functions; otherwise displays the final output.
FunValCheck	Check for invalid values, such as NaN or Inf, from the objective function.	• 'off' • 'on'
GradObj	Flags whether or not the objective function returns a gradient vector as a second output.	• 'off' • 'on'
Jacobian	Flags whether or not the objective function returns a Jacobian as a second output.	• 'off' • 'on'

Field	Parameter	Values
MaxFunEvals	Maximum number of objective function evaluations allowed.	Positive integer.
MaxIter	Maximum number of iterations allowed.	Positive integer.
Robust	Invoke robust fitting option.	• `'off'` • `'on'`
TolBnd	Parameter bound tolerance.	Positive scalar.
TolFun	Termination tolerance for the objective function value.	Positive scalar.
TolTypeFun	Use `TolFun` for absolute or relative objective function tolerances.	• `'abs'` • `'rel'`
TolTypeX	Use `TolX` for absolute or relative parameter tolerances.	• `'abs'` • `'rel'`
TolX	Termination tolerance for the parameters.	Positive scalar.
Tune	The tuning constant used in robust fitting 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.

Field	Parameter	Values
WgtFun	A weight function for robust fitting. Valid only when Robust is `'on'`. Can also be a function handle that accepts a normalized residual as input and returns the robust weights as output.	• `'bisquare'` • `'andrews'` • `'cauchy'` • `'fair'` • `'huber'` • `'logistic'` • `'talwar'` • `'welsch'`

Examples   Suppose you want to change the default parameter values for the function evfit, which fits an extreme value distribution to data. The defaults parameter values are:

```
statset('evfit')
ans =
        Display: 'off'
     MaxFunEvals: []
         MaxIter: []
          TolBnd: []
          TolFun: []
            TolX: 1.0000e-006
         GradObj: []
       DerivStep: []
     FunValCheck: []
          Robust: []
          WgtFun: []
            Tune: []
```

The only parameters that evfit uses are Display and TolX. To create an options structure with the value of TolX set to 1e-8, enter:

```
options = statset('TolX',1e-8)
```

```
options =
        Display: []
    MaxFunEvals: []
        MaxIter: []
         TolBnd: []
         TolFun: []
           TolX: 1.0000e-008
        GradObj: []
      DerivStep: []
    FunValCheck: []
         Robust: []
         WgtFun: []
           Tune: []
```

Pass options to evfit as follows:

```
mu = 1;
sigma = 1;
data = evrnd(mu,sigma,1,100);

paramhat = evfit(data,[],[],[],options)
paramhat =
    0.9051    1.0193
```

This call to evfit uses the default value of Display and the value of 1e-8 for TolX.

See Also   statget

Purpose	Return standard deviation of ProbDistUnivParam object
Syntax	*S* = std(*PD*)
Description	*S* = std(*PD*) returns *S*, the standard deviation of the ProbDistUnivParam object *PD*.

Inputs

PD An object of the class ProbDistUnivParam.

Outputs

S The standard deviation of the ProbDistUnivParam object *PD*.

See Also std

Purpose	Interactive stepwise regression

Syntax

```
stepwise
stepwise(X,y)
stepwise(X,y,inmodel,penter,premove)
```

Description stepwise uses the sample data in hald.mat to display a graphical user interface for performing stepwise regression of the response values in heat on the predictive terms in ingredients.

The upper left of the interface displays estimates of the coefficients for all potential terms, with horizontal bars indicating 90% (colored) and 95% (grey) confidence intervals. The red color indicates that, initially, the terms are not in the model. Values displayed in the table are those that would result if the terms were added to the model.

The middle portion of the interface displays summary statistics for the entire model. These statistics are updated with each step.

The lower portion of the interface, **Model History**, displays the RMSE for the model. The plot tracks the RMSE from step to step, so you can compare the optimality of different models. Hover over the blue dots in the history to see which terms were in the model at a particular step. Click on a blue dot in the history to open a copy of the interface initialized with the terms in the model at that step.

Initial models, as well as entrance/exit tolerances for the *p*-values of *F*-statistics, are specified using additional input arguments to `stepwise`. Defaults are an initial model with no terms, an entrance tolerance of 0.05, and an exit tolerance of 0.10.

To center and scale the input data (compute *z*-scores) to improve conditioning of the underlying least-squares problem, select `Scale Inputs` from the **Stepwise** menu.

You proceed through a stepwise regression in one of two ways:

1 Click **Next Step** to select the recommended next step. The recommended next step either adds the most significant term or removes the least significant term. When the regression reaches a local minimum of RMSE, the recommended next step is "Move no terms." You can perform all of the recommended steps at once by clicking **All Steps**.

2 Click a line in the plot or in the table to toggle the state of the corresponding term. Clicking a red line, corresponding to a term not currently in the model, adds the term to the model and changes the line to blue. Clicking a blue line, corresponding to a term currently in the model, removes the term from the model and changes the line to red.

To call `addedvarplot` and produce an added variable plot from the `stepwise` interface, select **Added Variable Plot** from the **Stepwise** menu. A list of terms is displayed. Select the term you want to add, and then click **OK**.

Click **Export** to display a dialog box that allows you to select information from the interface to save to the MATLAB workspace.

Check the information you want to export and, optionally, change the names of the workspace variables to be created. Click **OK** to export the information.

stepwise(X,y) displays the interface using the *p* predictive terms in the *n*-by-*p* matrix X and the response values in the *n*-by-1 vector y. Distinct predictive terms should appear in different columns of X.

Note stepwise automatically includes a constant term in all models. Do not enter a column of 1s directly into X.

---

stepwise treats NaN values in either X or y as missing values, and ignores them.

stepwise(X,y,inmodel,penter,premove) additionally specifies the initial model (inmodel) and the entrance (penter) and exit (premove) tolerances for the *p*-values of *F*-statistics. inmodel is either a logical vector with length equal to the number of columns of X, or a vector of indices, with values ranging from 1 to the number of columns in X. The value of penter must be less than or equal to the value of premove.

See Also   stepwisefit, addedvarplot, regress

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 response values in the n-by-1 vector `y` on the p predictive terms in the n-by-p matrix `X`. Distinct predictive terms should appear in different columns of `X`. `b` is a p-by-1 vector of estimated coefficients for all of the terms in `X`. The value in `b` for a term not included in the final model is the coefficient estimate that would result from adding the term to the model.

Note `stepwisefit` automatically includes a constant term in all models. Do not enter a column of 1s directly into `X`.

`stepwisefit` treats `NaN` values in either `X` or `y` as missing values, and ignores them.

`[b,se,pval,inmodel,stats,nextstep,history] = stepwisefit(...)` returns the following additional information:

- `se` — A vector of standard errors for `b`

- `pval` — A vector of p-values for testing whether elements of `b` are 0

- `inmodel` — A logical vector, with length equal to the number of columns in `X`, specifying which terms are in the final model

- `stats` — A structure of additional statistics with the following fields. All statistics pertain to the final model except where noted.

  - `source` — The string `'stepwisefit'`

  - `dfe` — Degrees of freedom for error

  - `df0` — Degrees of freedom for the regression

- **SStotal** — Total sum of squares of the response

- **SSresid** — Sum of squares of the residuals

- **fstat** — *F*-statistic for testing the final model vs. no model (mean only)

- **pval** — *p*-value of the *F*-statistic

- **rmse** — Root mean square error

- **xr** — Residuals for predictors not in the final model, after removing the part of them explained by predictors in the model

- **yr** — Residuals for the response using predictors in the final model

- **B** — Coefficients for terms in final model, with values for a term not in the model set to the value that would be obtained by adding that term to the model

- **SE** — Standard errors for coefficient estimates

- **TSTAT** — *t* statistics for coefficient estimates

- **PVAL** — *p*-values for coefficient estimates

- **intercept** — Estimated intercept

- **wasnan** — Indicates which rows in the data contained NaN values

- **nextstep** — The recommended next step—either the index of the next term to move in or out of the model, or 0 if no further steps are recommended

- **history** — A structure containing information on steps taken, with the following fields:

  - **rmse** — Root mean square errors for the model at each step

  - **df0** — Degrees of freedom for the regression at each step

  - **in** — Logical array indicating which predictors are in the model at each step

[...] = stepwisefit(X,y,*param1*,*val1*,*param2*,*val2*,...)
specifies one or more of the name/value pairs described in the following
table.

Parameter	Value
'inmodel'	A logical vector specifying terms to include in the initial fit. The default is to specify no terms.
'penter'	The maximum *p*-value for a term to be added. The default is 0.05.
'premove'	The minimum *p*-value for a term to be removed. The default is the maximum of the value of 'penter' and 0.10.
'display'	'on' displays information about each step in the command window. This is the default. 'off' omits the display.
'maxiter'	The maximum number of steps in the regression. The default is Inf.
'keep'	A logical vector specifying terms to keep in their initial state. The default is to specify no terms.
'scale'	'on' centers and scales each column of X (computes *z*-scores) before fitting. 'off' does not scale the terms. This is the default.

Algorithm   *Stepwise regression* is a systematic method for adding and removing
terms from a multilinear model based on their statistical significance
in a regression. The method begins with an initial model and then
compares the explanatory power of incrementally larger and smaller
models. At each step, the *p*-value of an *F*-statistic is computed to test
models with and without a potential term. If a term is not currently
in the model, the null hypothesis is that the term would have a zero
coefficient if added to the model. If there is sufficient evidence to reject

the null hypothesis, the term is added to the model. Conversely, if a term is currently in the model, the null hypothesis is that the term has a zero coefficient. If there is insufficient evidence to reject the null hypothesis, the term is removed from the model. The method proceeds as follows:

1 Fit the initial model.

2 If any terms not in the model have *p*-values less than an entrance tolerance (that is, if it is unlikely that they would have zero coefficient if added to the model), add the one with the smallest *p*-value and repeat this step; otherwise, go to step 3.

3 If any terms in the model have *p*-values greater than an exit tolerance (that is, if it is unlikely that the hypothesis of a zero coefficient can be rejected), remove the one with the largest *p*-value and go to step 2; otherwise, end.

Depending on the terms included in the initial model and the order in which terms are moved in and out, the method may build different models from the same set of potential terms. The method terminates when no single step improves the model. There is no guarantee, however, that a different initial model or a different sequence of steps will not lead to a better fit. In this sense, stepwise models are locally optimal, but may not be globally optimal.

Examples   Load the data in `hald.mat`, which contains observations of the heat of reaction of various cement mixtures:

```
load hald
whos
  Name          Size    Bytes   Class    Attributes

  Description   22x58   2552    char
  hald          13x5    520     double
  heat          13x1    104     double
  ingredients   13x4    416     double
```

The response (`heat`) depends on the quantities of the four predictors (the columns of `ingredients`).

Use `stepwisefit` to carry out the stepwise regression algorithm, beginning with no terms in the model and using entrance/exit tolerances of 0.05/0.10 on the *p*-values:

```
stepwisefit(ingredients,heat,...
            'penter',0.05,'premove',0.10);
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]
```

`stepwisefit` automatically includes an intercept term in the model, so you do not add it explicitly to `ingredients` as you would for `regress`. For terms not in the model, coefficient estimates and their standard errors are those that result if the term is added.

The `inmodel` parameter is used to specify terms in an initial model:

```
initialModel = ...
            [false true false false]; % Force in 2nd term
stepwisefit(ingredients,heat,...
            'inmodel',initialModel,...
            'penter',.05,'premove',0.10);
Initial columns included:  2
Step 1, added column 1, p=2.69221e-007
Final columns included: 1 2
    '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]
```

The preceding two models, built from different initial models, use different subsets of the predictive terms. Terms 2 and 4, swapped in the two models, are highly correlated:

```
term2 = ingredients(:,2);
term4 = ingredients(:,4);
R = corrcoef(term2,term4)
R =
    1.0000   -0.9730
   -0.9730    1.0000
```

To compare the models, use the stats output of stepwisefit:

```
[betahat1,se1,pval1,inmodel1,stats1] = ...
          stepwisefit(ingredients,heat,...
          'penter',.05,'premove',0.10,...
          'display','off');
[betahat2,se2,pval2,inmodel2,stats2] = ...
          stepwisefit(ingredients,heat,...
                     'inmodel',initialModel,...
                     'penter',.05,'premove',0.10,...
                     'display','off');
RMSE1 = stats1.rmse
RMSE1 =
    2.7343
RMSE2 = stats2.rmse
RMSE2 =
    2.4063
```

The second model has a lower Root Mean Square Error (RMSE).

References   [1] Draper, N. R., and H. Smith. *Applied Regression Analysis*. Hoboken, NJ: Wiley-Interscience, 1998. pp. 307–312.

See Also   stepwise, addedvarplot, regress

categorical.subsasgn

Purpose	Subscripted assignment for categorical array
Syntax	A = subsasgn(A,S,B)

Description   A = subsasgn(A,S,B) is called for the syntax A(i)=B. S is a structure array with the fields:

type	String containing '()' specifying the subscript type. Only parenthesis subscripting is allowed.
subs	Cell array or string containing the actual subscripts.

See Also   categorical, subsref

Purpose   Subscripted reference for classregtree object

Syntax

Description   Subscript assignment is not allowed for a classregtree object.

See Also   classregtree

dataset.subsasgn

Purpose   Subscripted assignment to dataset array

Description   A = subsasgn(A,S,B) is called for the syntax A(i,j)=B, A{i,j}=B, or
A.var=B when A is a dataset array. S is a structure array with the fields:

type	String containing '()', '{}', or '.' specifying the subscript type.
subs	Cell array or string containing the actual subscripts.

A(i,j) = B assigns the contents of the dataset array B to a subset of
the observations and variables in the dataset array A. i and j are one of
the following types:

- positive integers
- vectors of positive integers
- observation/variable names
- cell arrays containing one or more observation/variable names
- logical vectors

The assignment does not use observation names, variable names, or any
other properties of B to modify properties of A; however properties of A
are extended with default values if the assignment expands the number
of observations or variables in A. Elements of B are assigned into A by
position, not by matching names.

A{i,j} = B assigns the value B into an element of the dataset array A.
i and J are positive integers, or logical vectors. Cell indexing cannot
assign into multiple dataset elements, that is, the subscripts i and
j must each refer to only a single observation or variable. B is cast
to the type of the target variable if necessary. If the dataset element
already exists, A{i,j} may also be followed by further subscripting as
supported by the variable.

17-1214

For dataset variables that are cell arrays, assignments such as `A{1,'CellVar'} = B` assign into the contents of the target dataset element in the same way that {}-indexing of an ordinary cell array does.

For dataset variables that are n-D arrays, i.e., each observation is a matrix or array, an assignment such as `A{1,'ArrayVar'} = B` assigns into the second and following dimensions of the target dataset element, i.e., the assignment adds a leading singleton dimension to `B` to account for the observation dimension of the dataset variable.

`A.var = B` or `A.(varname) = B` assigns `B` to a dataset variable. `var` is a variable name literal, or `varname` is a character variable containing a variable name. If the dataset variable already exists, the assignment completely replaces that variable. To assign into an element of the variable, `A.var` or `A.(varname)` may be followed by further subscripting as supported by the variable. In particular, `A.var(obsnames,...) = B` and `A.var{obsnames,...} = B` (when supported by `var`) provide assignment into a dataset variable using observation names.

`A.properties.propertyname = P` assigns to a dataset property. `propertyname` is one of the following:

- `'ObsNames'`
- `'VarNames'`
- `'Description'`
- `'Units'`
- `'DimNames'`
- `'UserData'`

To assign into an element of the property, `A.properties.propertyname` may also be followed by further subscripting as supported by the property.

Limitations

You cannot assign multiple values into dataset variables or properties using assignments such as `[A.CellVar{1:2}] = B`,

[A.StructVar(1:2).field] = B, or [A.Properties.ObsNames{1:2}] = B. Use multiple assignments of the form A.CellVar{1} = B instead.

Similarly, if a dataset variable is a cell array with multiple columns or is an n-D cell array, then the contents of that variable for a single observation consists of multiple cells, and you cannot assign to all of them using the syntax A{1,'CellVar'} = B. Use multiple assignments of the form [A.CellVar{1,1}] = B instead.

See Also   dataset, set, subsref

Purpose   Subscripted reference for Gaussian mixture distribution object

Description   Subscript assignment is not allowed for `gmdistribution` objects.

See Also   `gmdistribution`

NaiveBayes.subsasgn

Purpose   Subscripted reference for `NaiveBayes` object

Description   Subscript assignment is not allowed for a `NaiveBayes` object.

categorical.subsindex

Purpose   Subscript index for categorical array

Syntax   I = subsindex(A)

Description   I = subsindex(A) is called for the syntax 'X(A)' when A is a
                   categorical array and X is one of the built-in types (most commonly
                   'double'). subsindex returns the internal categorical level codes
                   of A converted to zero-based integer indices. subsindex is invoked
                   separately on all the subscripts in an expression such as X(A,B).

Examples
```
load fisheriris
a = ordinal(species,[],unique(species));
colmeans = grpstats(meas,a,@mean);
residuals = meas - colmeans(a,:);
```

See Also   categorical, double

classregtree.subsref

Purpose	Subscripted reference for `classregtree` object

Syntax	`B = subsref(T,S)`

Description   `B = subsref(T,S)` is called for the syntax `T(X)` when `T` is a `classregtree` object. `S` is a structure array with the fields:

type	String containing `'()'`, `'{}'`, or `'.'` specifying the subscript type.
subs	Cell array or string containing the actual subscripts.

`[...]=T(...)` invokes the `eval` method for the tree `T`.

See Also	`classregtree`, `eval`

Purpose   Subscripted reference for categorical array

Syntax   A = subsref(A,S,B)

Description A = subsref(A,S,B) is called for the syntax A(I)=B. S is a structure
                array with the fields:

type	String containing '()' specifying the subscript type. Only parenthesis subscripting is allowed.
subs	Cell array or string containing the actual subscripts.

See Also   categorical, subsasgn

dataset.subsref

Purpose   Subscripted reference for dataset array

Syntax   B = subsref(A,S)

Description   B = subsref(A,S) is called for the syntax A(i,j), A{i,j}, or A.var when A is a dataset array. S is a structure array with the fields:

type	String containing '()', '{}', or '.' specifying the subscript type.
subs	Cell array or string containing the actual subscripts.

B = A(i,j) returns a dataset array that contains a subset of the observations and variables in the dataset array A. i and j are one of the following types:

- positive integers

- vectors of positive integers

- observation/variable names

- cell arrays containing one or more observation/variable names

- logical vectors

B contains the same property values as A, subsetted for observations or variables where appropriate.

B = A{i,j} returns an element of a dataset variable. i and j are positive integers, or logical vectors. Cell indexing cannot return multiple dataset elements, that is, the subscripts i and j must each refer to only a single observation or variable. A{i,j} may also be followed by further subscripting as supported by the variable.

For dataset variables that are cell arrays, expressions such as A{1,'CellVar'} return the contents of the referenced dataset element in the same way that {}-indexing on an ordinary cell array does. If the dataset variable is a single column of cells, the contents of a single cell

is returned. If the dataset variable has multiple columns or is n-D, multiple outputs containing the contents of multiple cells are returned.

For dataset variables that are n-D arrays, i.e., each observation is a matrix or an array, expressions such as `A{1,'ArrayVar'}` return `A.ArrayVar(1,:,...)` with the leading singleton dimension squeezed out.

`B = A.var` or `A.(varname)` returns a dataset variable. `var` is a variable name literal, or `varname` is a character variable containing a variable name. `A.var` or `A.(varname)` may also be followed by further subscripting as supported by the variable. In particular, `A.var(obsnames,...)` and `A.var{obsnames,...}` (when supported by `var`) provide subscripting into a dataset variable using observation names.

`P = A.properties.propertyname` returns a dataset property. `propertyname` is one of the following:

- `'ObsNames'`
- `'VarNames'`
- `'Description'`
- `'Units'`
- `'DimNames'`
- `'UserData'`

`A.properties.propertyname` may also be followed by further subscripting as supported by the property.

Limitations

Subscripting expressions such as `A.CellVar{1:2}`, `A.StructVar(1:2).field`, or `A.Properties.ObsNames{1:2}` are valid, but result in `subsref` returning multiple outputs in the form of a comma-separated list. If you explicitly assign to output arguments on the left-hand side of an assignment, for example, `[cellval1,cellval2] = A.CellVar{1:2}`, those variables will receive

the corresponding values. However, if there are no output arguments, only the first output in the comma-separated list is returned.

Similarly, if a dataset variable is a cell array with multiple columns or is an n-D cell array, then subscripting expressions such as `A{1,'CellVar'}` result in `subsref` returning the contents of multiple cells. You should explicitly assign to output arguments on the left-hand side of an assignment, for example, `[cellval1,cellval2] = A{1,'CellVar'}`.

See Also   `dataset`, `set`, `subsasgn`

Purpose   Subscripted reference for Gaussian mixture distribution object

Syntax   B = subsref(T,S)

Description   B = subsref(T,S) is called for the syntax T(X) when T is a
gmdistribution object. S is a structure array with the following fields:

type	String containing '()', '{}', or '.' specifying the subscript type.
subs	Cell array or string containing the actual subscripts.

See Also   gmdistribution

NaiveBayes.subsref

Purpose	Subscripted reference for `NaiveBayes` object
Syntax	`b = subsref(nb,s)`
Description	`b = subsref(nb,s)` is called for the syntax `nb(s)` when `nb` is a `NaiveBayes` object. `S` is a structure array with the fields:

type	string containing '()', '{}', or '.' specifying the subscript type.
subs	Cell array or string containing the actual subscripts.

Purpose   Subscripted reference for qrandset

Syntax   x = p(i,j)
                x = subsref(p,s)

Description x = p(i,j) returns a matrix that contains a subset of the points from
                the point set p. The indices in i select points from the set and the
                indices in j select columns from those points. i and j are vector of
                positive integers or logical vectors. A colon used as a subscript, as in
                p(i,:), indicates the entire row (or column).

                x = subsref(p,s) is called for the syntax p(i), p{i}, or p.i. s is a
                structure array with the fields:

type	string containing '()', '{}', or '.' specifying the subscript type.
subs	Cell array or string containing the actual subscripts.

Examples

Command	Returns
p = sobolset(5);	The fifth point
x = p(1:10,:)	All columns of the first 10 points
x = p(end,1)	The first column of the last point
x = p([1,4,5], :)	Points 1, 4, and 5

See Also   qrandset

categorical.summary

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`.

Examples   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

dataset.summary

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
     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

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

ProbDist.Support property

Purpose   Read-only structure containing information about support of ProbDist object

Description   Support is a read-only property of the ProbDist class. Support is a structure containing information about the support of a ProbDist object. It includes the following fields:

- range
- closedbound
- iscontinuous

Values   The values for the three fields in the structure are:

- range — A two-element vector [L, U], such that all of the probability is contained from L to U.

- closedbound — A two-element logical vector indicating whether the corresponding range endpoint is included. Possible values for each endpoint are 1 (true) or 0 (false).

- iscontinuous — A logical value indicates if the distribution takes values on the entire interval from L to U (true), or if it takes only integer values within this range (false). Possible values are 1 (true) or 0 (false).

Use this information to view and compare information about the support of distributions.

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.

tabulate

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 nonnegative 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.

Examples

```
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   "Grouped Data" on page 2-33

pareto

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'`.

tblread

Examples
```
[data,varnames,casenames] = tblread('sat.dat')
data =
  470  530
  520  480

varnames =
Male
Female

casenames =
Verbal
Quantitative
```

See Also   tblwrite, tdfread, caseread

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'.

Examples   Continuing the example from tblread:

                   tblwrite(data,varnames,casenames,'sattest.dat')

tblwrite

```
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 *v* degrees of freedom will fall in the interval [–∞, *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, tpdf, tinv, tstat, trnd

Purpose   Read tab-delimited file

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.

tdfread

Examples

The following displays the contents of the file `sat2.dat`:

```
type sat2.dat

Test,Gender,Score
Verbal,Male,470
Verbal,Female,530
Quantitative,Male,520
Quantitative,Female,480
```

The following creates the variables `Gender`, `Score`, and `Test` from the file `sat2.dat` and displays the contents of the MATLAB workspace:

```
tdfread('sat2.dat',',')

Name     Size     Bytes    Class      Attributes

Gender   4x6      48       char
Score    4x1      32       double
Test     4x12     96       char
```

See Also   tblread, caseread

Purpose   Error rate

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 for methods other than `'resubstitution'`, 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.

Examples   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)
```
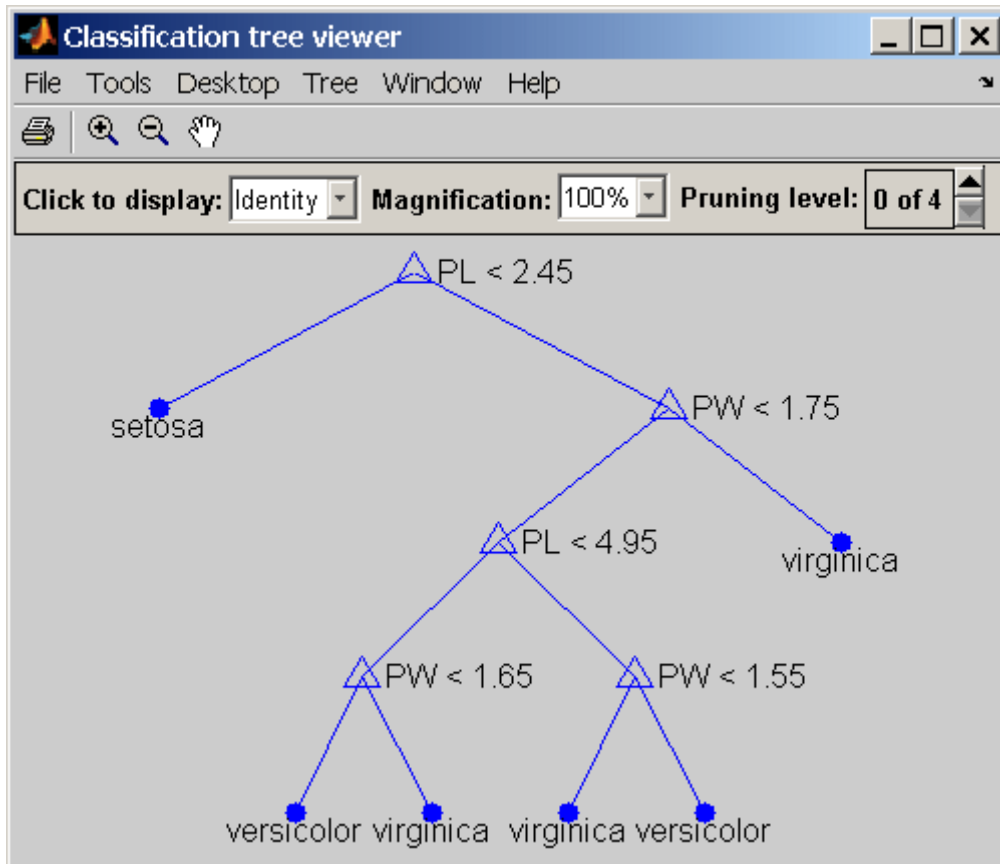

Find the minimum-cost tree:

```
[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)
```
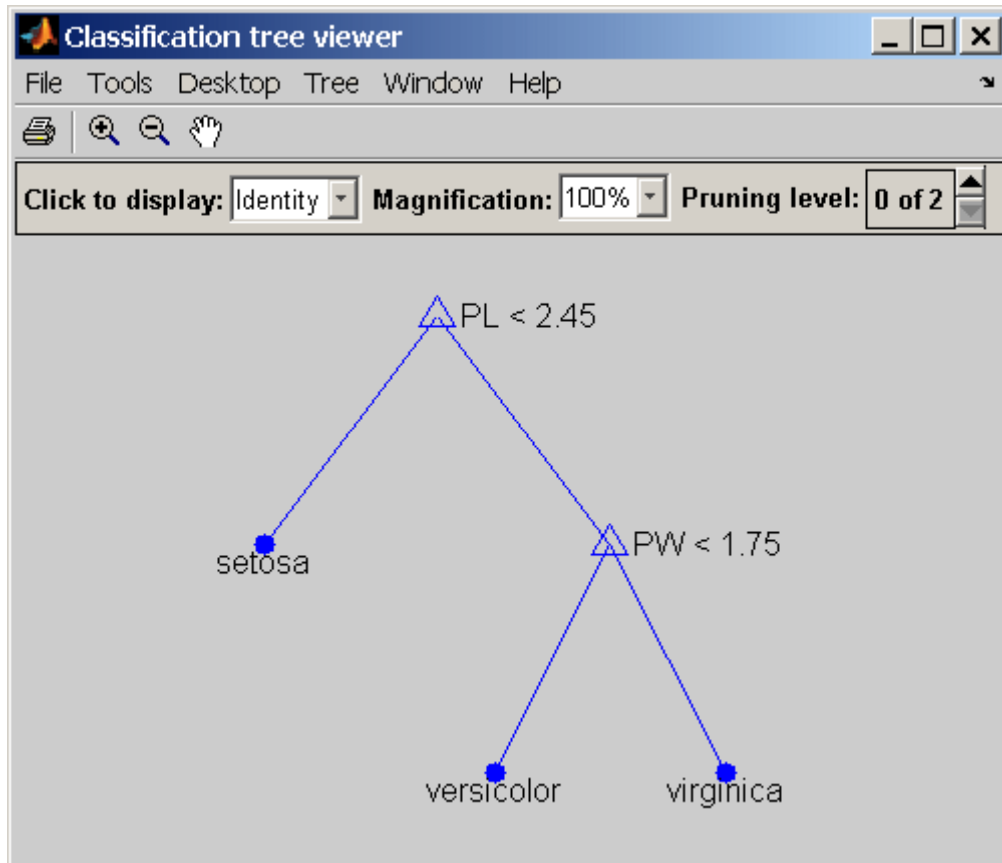

Plot the smallest tree within one standard error of the minimum cost tree:

```
[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')
```
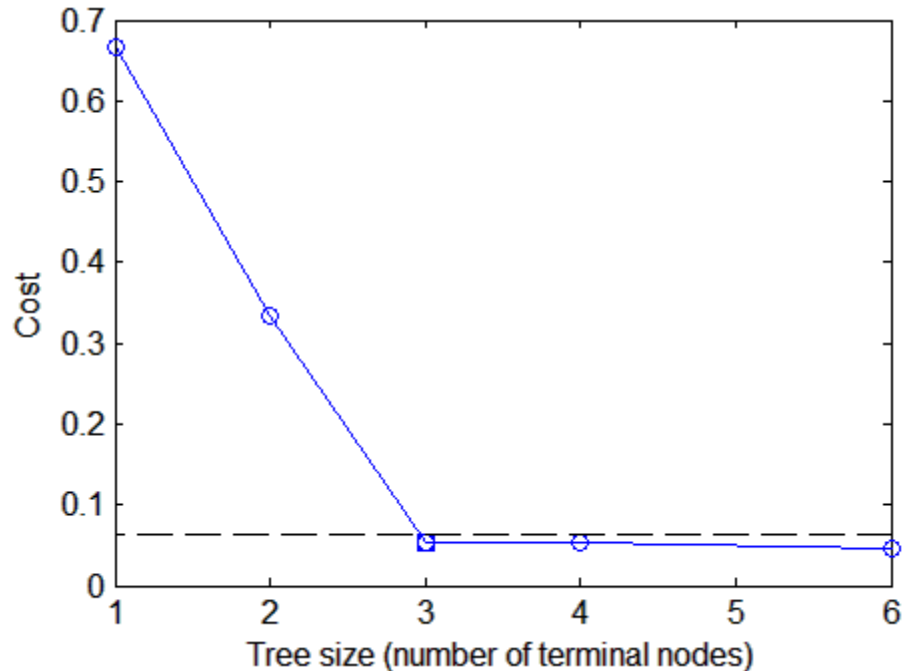

The solid line shows the estimated cost for each tree size, the dashed line marks one standard error above the minimum, and the square marks the smallest tree under the dashed line.

References   [1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

See Also   classregtree, eval, view, prune

Purpose	Test indices for cross-validation

Syntax

```
idx = test(c)
idx = test(c,i)
```

Description

`idx = test(c)` returns the logical vector `idx` of test indices for an object `c` of the `cvpartition` class of type `'holdout'` or `'resubstitution'`.

If `c.Type` is `'holdout'`, `idx` specifies the observations in the test set.

If `c.Type` is `'resubstitution'`, `idx` specifies all observations.

`idx = test(c,i)` returns the logical vector `idx` of test indices for repetition `i` of an object `c` of the `cvpartition` class of type `'kfold'` or `'leaveout'`.

If `c.Type` is `'kfold'`, `idx` specifies the observations in the test set in fold `i`.

If `c.Type` is `'leaveout'`, `idx` specifies the observation left out at repetition `i`.

Examples

Identify the test indices in the first fold of a partition of 10 observations for 3-fold cross-validation:

```
c = cvpartition(10,'kfold',3)
c =
K-fold cross validation partition
             N: 10
   NumTestSets: 3
     TrainSize: 7  6  7
      TestSize: 3  4  3

test(c,1)
ans =
     1
     1
     0
     0
```

```
                    0
                    0
                    0
                    0
                    1
                    0
```

See Also   cvpartition, training

cvpartition.TestSize property

Purpose   Size of each test set

Description   Value is a vector in partitions of type `'kfold'` and `'leaveout'`.
Value is a scalar in partitions of type `'holdout'` and
`'resubstitution'`.

tiedrank

Purpose   Rank adjusted 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

categorical.times

Purpose   Product of categorical arrays

Syntax   C = times(A,B)

Description C = times(A,B) returns a categorical array each of whose elements
has the level formed from the concatenation of the levels of the
corresponding elements of A and B. The set of levels of C is the cartesian
product of the sets of levels of A and of B. The syntax A .* B calls C
= times(A,B).

See Also   categorical

Purpose   Student's *t* inverse 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 are 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

                     $$x = F^{-1}(p|v) = \{x : F(x|v) = p\}$$

                     where

                     $$p = F(x|v) = \int_{-\infty}^{x} \frac{\Gamma\left(\frac{v+1}{2}\right)}{\Gamma\left(\frac{v}{2}\right)} \frac{1}{\sqrt{v\pi}} \frac{1}{\left(1+\frac{t^2}{v}\right)^{\frac{v+1}{2}}} dt$$

                     The result, *x*, is the solution of the cdf integral with parameter *v*, 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

tpdf

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, tstat, trnd

Purpose   Training indices for cross-validation

Syntax   idx = training(c)
idx = training(c,i)

Description   idx = training(c) returns the logical vector idx of training indices
for an object c of the cvpartition class of type 'holdout' or
'resubstitution'.

If c.Type is 'holdout', idx specifies the observations in the training
set.

If c.Type is 'resubstitution', idx specifies all observations.

idx = training(c,i) returns the logical vector idx of training indices
for repetition i of an object c of the cvpartition class of type 'kfold'
or 'leaveout'.

If c.Type is 'kfold', idx specifies the observations in the training
set in fold i.

If c.Type is 'leaveout', idx specifies the observations left in at
repetition i.

Examples   Identify the training indices in the first fold of a partition of 10
observations for 3-fold cross-validation:

```
c = cvpartition(10,'kfold',3)
c =
K-fold cross validation partition
            N: 10
   NumTestSets: 3
     TrainSize: 7  6  7
      TestSize: 3  4  3

training(c,1)
ans =
     0
     0
```

```
                    1
                    1
                    1
                    1
                    1
                    1
                    0
                    1
```

See Also   cvpartition, test

Purpose   Size of each training set

Description   Value is a vector in partitions of type `'kfold'` and `'leaveout'`.
Value is a scalar in partitions of type `'holdout'` and
`'resubstitution'`.

See Also   type

categorical.transpose

Purpose   Transpose categorical matrix

Syntax   B = transpose(A)

Description B = transpose(A) returns the transpose of the 2-D categorical matrix A. ctranspose is identical to transpose for categorical arrays. The syntax A.' calls transpose.

See Also   ctranspose, permute

Purpose   Cell array of arguments for `classregtree`

Description   The `TreeArgs` property is a cell array of arguments for the
                   `classregtree` constructor. `TreeBagger` uses these arguments in
                   growing new trees for the ensemble.

TreeBagger class

Purpose   Bootstrap aggregation for ensemble of decision trees

Description   `TreeBagger` bags an ensemble of decision trees for either classification or regression. Bagging stands for bootstrap aggregation. Every tree in the ensemble is grown on an independently drawn bootstrap replica of input data. Observations not included in this replica are "out of bag" for this tree. To compute prediction of an ensemble of trees for unseen data, `TreeBagger` takes an average of predictions from individual trees. To estimate the prediction error of the bagged ensemble, you can compute predictions for each tree on its out-of-bag observations, average these predictions over the entire ensemble for each observation and then compare the predicted out-of-bag response with the true value at this observation.

`TreeBagger` relies on the `classregtree` functionality for growing individual trees. In particular, `classregtree` accepts the number of features selected at random for each decision split as an optional input argument.

The `Compact` property contains another class, `CompactTreeBagger`, with sufficient information to make predictions using new data. This information includes the tree ensemble, variable names, and class names (for classification). `CompactTreeBagger` requires less memory than `TreeBagger`, but only `TreeBagger` has methods for growing more trees for the ensemble. Once you grow an ensemble of trees using `TreeBagger` and no longer need access to the training data, you can opt to work with the compact version of the trained ensemble from then on.

Construction

TreeBagger	Create ensemble of bagged decision trees

Methods

append	Append new trees to ensemble
compact	Compact ensemble of decision trees

error	Error (misclassification probability or MSE)
fillProximities	Proximity matrix for training data
growTrees	Train additional trees and add to ensemble
margin	Classification margin
mdsProx	Multidimensional scaling of proximity matrix
meanMargin	Mean classification margin
oobError	Out-of-bag error
oobMargin	Out-of-bag margins
oobMeanMargin	Out-of-bag mean margins
oobPredict	Ensemble predictions for out-of-bag observations
predict	Predict response

Properties

ClassNames	Names of classes
ComputeOOBPrediction	Flag to compute out-of-bag predictions
ComputeOOBVarImp	Flag to compute out-of-bag variable importance
Cost	Misclassification costs
DefaultYfit	Default value returned by `predict` and `oobPredict`
DeltaCritDecisionSplit	Split criterion contributions for each predictor

FBoot	Fraction of in-bag observations
MergeLeaves	Flag to merge leaves that do not improve risk
Method	Method used by trees (classification or regression)
MinLeaf	Minimum number of observations per tree leaf
NTrees	Number of decision trees in ensemble
NVarToSample	Number of variables for random feature selection
OOBIndices	Indicator matrix for out-of-bag observations
OOBInstanceWeight	Count of out-of-bag trees for each observation
OOBPermutedVarCountRaiseMargin	Variable importance for raising margin
OOBPermutedVarDeltaError	Variable importance for classification error
OOBPermutedVarDeltaMeanMargin	Variable importance for classification margin
OutlierMeasure	Measure for determining outliers
Prior	Prior class probabilities
Proximity	Proximity matrix for observations
Prune	Flag to prune trees
SampleWithReplacement	Flag to sample with replacement
TreeArgs	Cell array of arguments for `classregtree`
Trees	Decision trees in ensemble

VarNames	Variable names
X	X data used to create ensemble
Y	Y data used to create ensemble

Copy Semantics

Value. To learn how this affects your use of the class, see Comparing Handle and Value Classes in the MATLAB Object-Oriented Programming documentation.

See Also

"Regression and Classification by Bagging Decision Trees" on page 11-14

Classification Trees

Regression Tress

Grouped Data

TreeBagger

Purpose	Create ensemble of bagged decision trees
Syntax	B = TreeBagger(ntrees,X,Y) B = TreeBagger(ntrees,X,Y,'param1',val1,'param2',val2,...)

Description   B = TreeBagger(ntrees,X,Y) creates an ensemble B of ntrees
decision trees for predicting response Y as a function of predictors
X. By default TreeBagger builds an ensemble of classification trees.
The function can build an ensemble of regression trees by setting the
optional input argument 'method' to 'regression'.

X is a numeric matrix of training data. Each row represents an
observation and each column represents a predictor or feature. Y is an
array of true class labels for classification or numeric function values for
regression. True class labels can be a numeric vector, character matrix,
vector cell array of strings or categorical vector. TreeBagger converts
labels to a cell array of strings for classification.

B = TreeBagger(ntrees,X,Y,'param1',val1,'param2',val2,...)
specifies optional parameter name/value pairs:

'FBoot'	Fraction of input data to sample with replacement from the input data for growing each new tree.
'oobpred'	'on' to store info on what observations are out of bag for each tree. This info can be used by oobPredict to compute the predicted class probabilities for each tree in the ensemble. Default is 'off'.
'OOBVarImp'	'on' to store out-of-bag estimates of feature importance in the ensemble. Default is 'off'. Specifying 'on' also sets the 'ooberr' value to 'on'.
'Method'	Either 'classification' or 'regression'. Regression requires a numeric Y.

'NVarToSample'	Number of variables to select at random for each decision split. Default is the square root of the number of variables for classification and one third of the number of variables for regression. Valid values are 'all' or a positive integer.
'NPrint'	Number of training cycles (grown trees) after which TreeBagger displays a diagnostic message showing training progress. Default is no diagnostic messages.
'MinLeaf'	Minimum number of observations per tree leaf. Default is 1 for classification and 5 for regression.

In addition to the optional arguments above, this method accepts all optional classregtree arguments with the exception of 'minparent'. Refer to the documentation for classregtree for more detail.

Examples

```
load fisheriris
b = TreeBagger(50,meas,species,'oobpred','on')
plot(oobError(b))
xlabel('number of grown trees')
ylabel('out-of-bag classification error')
```

returns

```
b =

Ensemble with 50 bagged decision trees:
                   Training X:            [150x4]
                   Training Y:            [150x1]
                       Method:     classification
                        Nvars:                  4
                NVarToSample:                  2
                     MinLeaf:                  1
                        FBoot:                  1
      SampleWithReplacement:                  1
       ComputeOOBPrediction:                  1
          ComputeOOBVarImp:                  0
```

```
                          Proximity:                      []
                             Prune:                        0
                      MergeLeaves:                          0
                          TreeArgs:
                        ClassNames: 'setosa' 'versicolor' 'virginica'
```
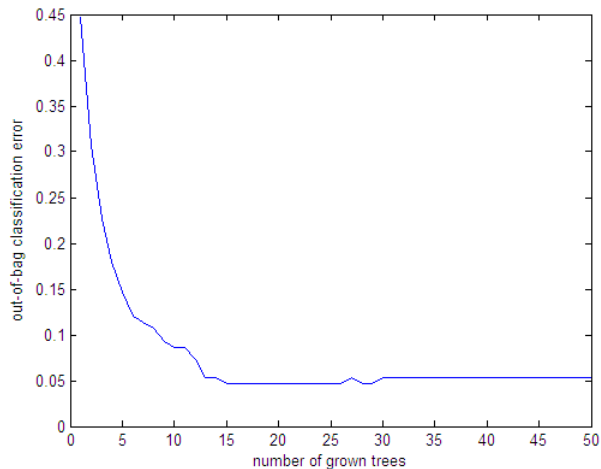

See Also   "Regression and Classification by Bagging Decision Trees" on page
11-14, "Grouped Data" on page 2-33

classregtree, CompactTreeBagger

Purpose   Plot tree

Syntax   treedisp(t)
               treedisp(t,*param1*,*val1*,*param2*,*val2*,...)

Description

Note This function is superseded by the view method of 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, as described in the following table.

Menu Choice	Displays
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, listed in the following table.

Parameter	Value
'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'});
```
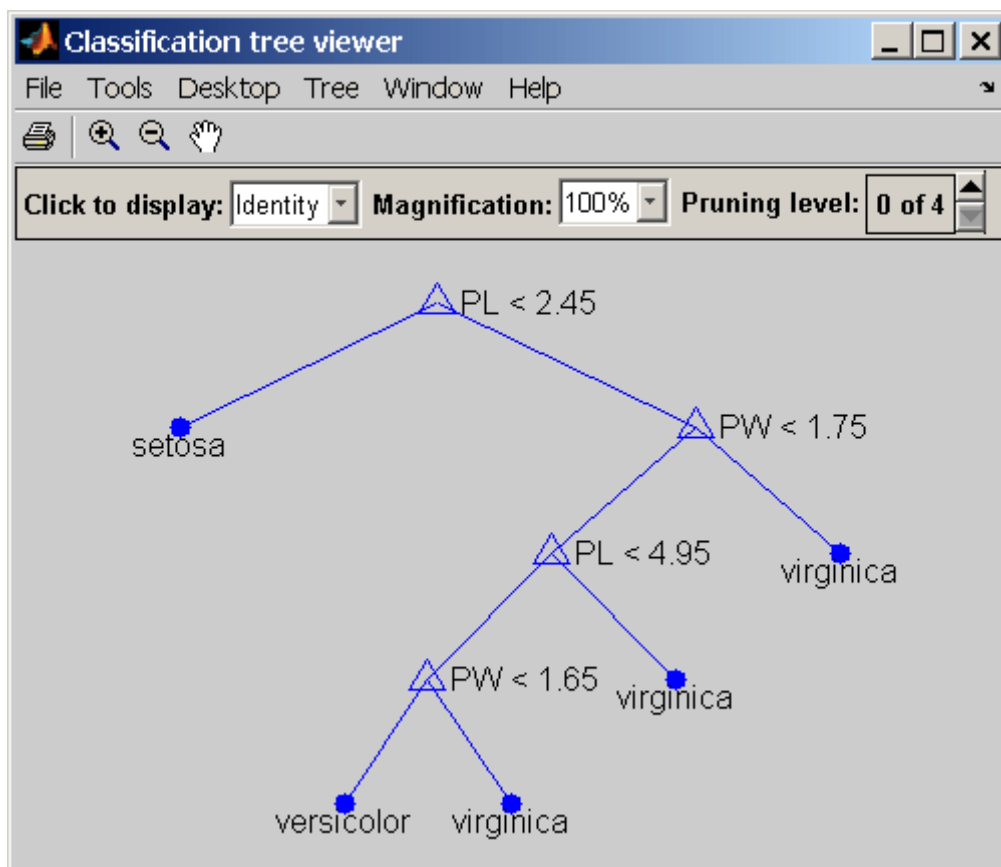
References   [1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

See Also   treefit, treeprune, treetest

treefit

Purpose   Fit tree

Syntax   t = treefit(X,y)
                   t = treefit(X,y,*param1*,*val1*,*param2*,*val2*,...)

Description

> **Note** This function is superseded by the classregtree constructor
> of the classregtree class 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:

The following table lists parameters available for all trees.

Parameter	Value
'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.

The following table lists parameters available for classification trees only.

Parameter	Value
'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 2-33), 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'});
```
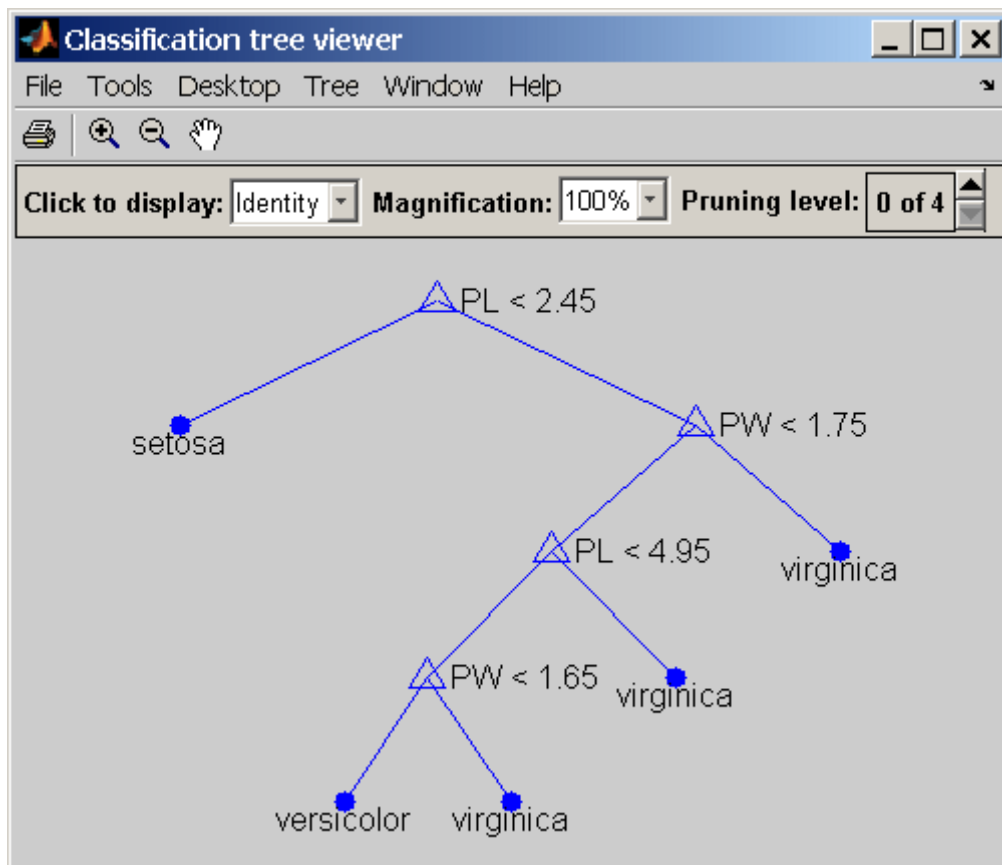
References   [1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

See Also   "Grouped Data" on page 2-33

treedisp, treetest

Purpose   Prune tree

Syntax   t2 = treeprune(t1,'level',level)
              t2 = treeprune(t1,'nodes',nodes)
              t2 = treeprune(t1)

Description

Note This function is superseded by the prune method of 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'});
```
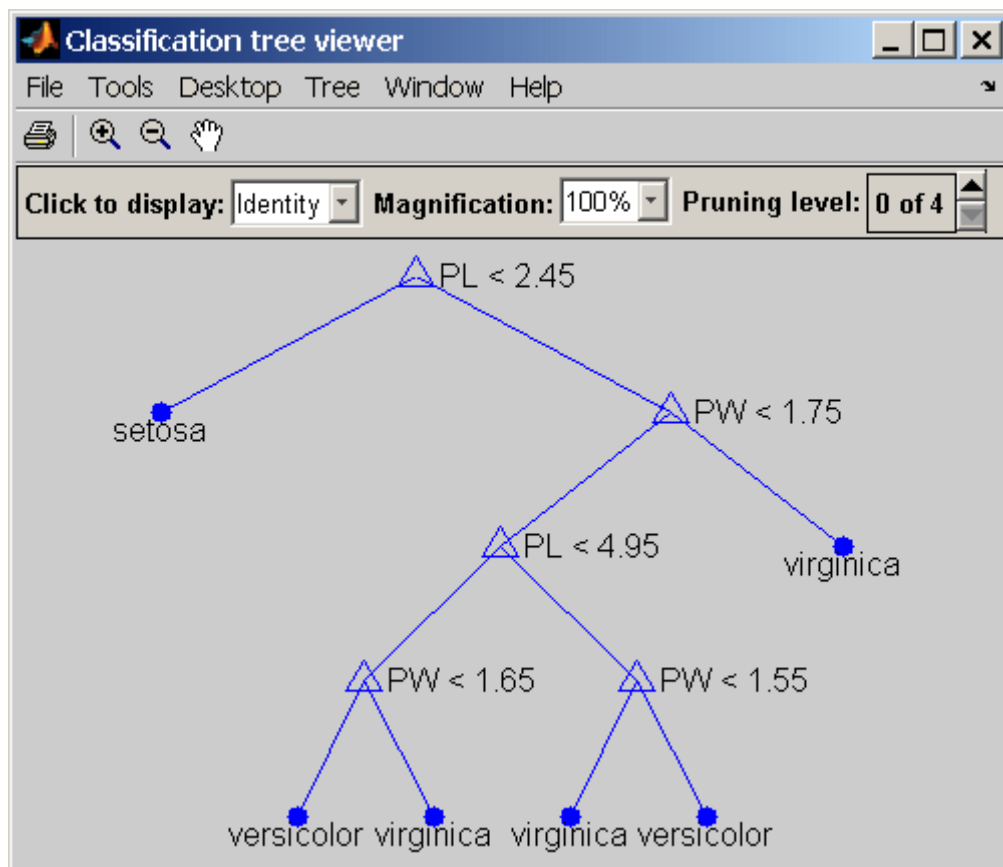


```
t2 = treeprune(t1,'level',1);
treedisp(t2,'names',{'SL' 'SW' 'PL' 'PW'});
```
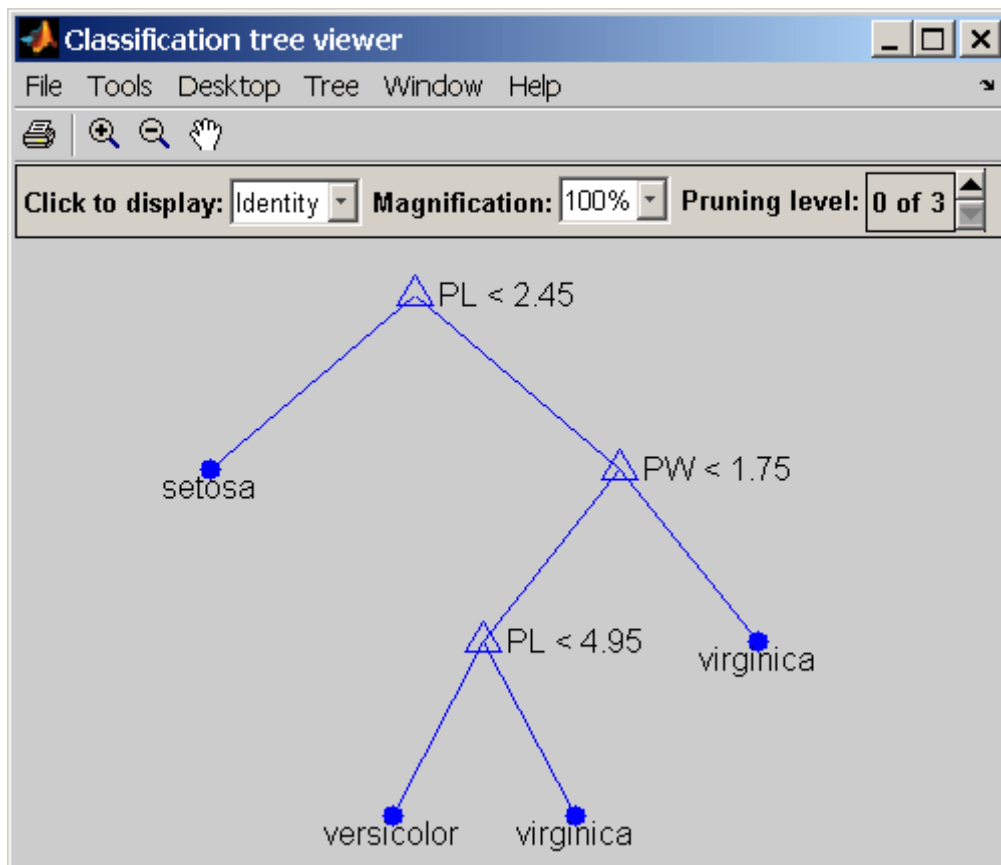
References   [1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

See Also   treefit, treetest, treedisp

CompactTreeBagger.Trees property

Purpose　　　Decision trees in ensemble

Description　　The Trees property is a cell array of size NTrees-by-1 containing the trees in the ensemble.

See Also　　　NTrees

Purpose   Decision trees in ensemble

Description   The Trees property is a cell array of size NTrees-by-1 containing the
                  trees in the ensemble.

See Also   NTrees

treetest

Purpose   Error rate

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 of the `classregtree` class 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 table.

Parameter	Value
'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(best+1),c(best+1),'bs',...
     n,(mincost+s(minloc))*ones(size(n)),'k--');
xlabel('Tree size (number of terminal nodes)')
ylabel('Cost')
```
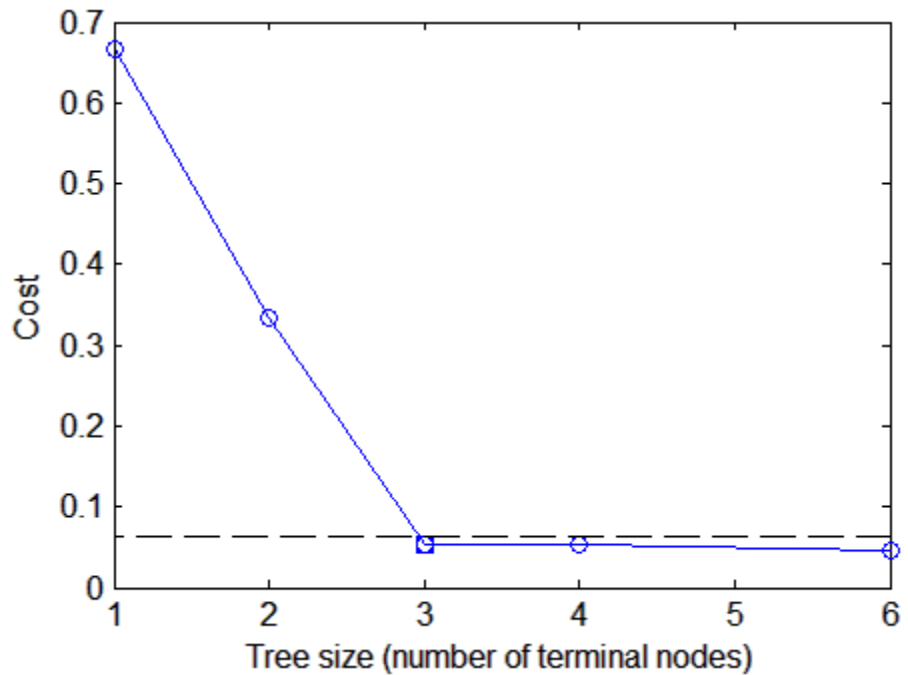

References   [1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

See Also   treefit, treedisp

Purpose   Predicted responses

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 of 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 jth 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.

treeval

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 in correct class
ans =
    0.9800
```

References   [1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

See Also   treefit, treeprune, treetest

Purpose   Mean excluding outliers

Syntax
```
m = trimmean(X,percent)
trimmean(X,percent,dim)
m = trimmean(X,percent,flag)
m = trimmean(x,percent,flag,dim)
```

Description   `m = trimmean(X,percent)` calculates the trimmed mean of the values in X. For a vector input, m is the mean of X, excluding the highest and lowest k data values, where `k=n*(percent/100)/2` and where n is the number of values in X. For a matrix input, m is a row vector containing the trimmed mean of each column of X. For n-D arrays, `trimmean` operates along the first non-singleton dimension. `percent` is a scalar between 0 and 100.

`trimmean(X,percent,dim)` takes the trimmed mean along dimension `dim` of X.

`m = trimmean(X,percent,flag)` controls how to trim when k is not an integer. `flag` can be chosen from the following:

`'round'`	Round k to the nearest integer (round to a smaller integer if k is a half integer). This is the default.
`'floor'`	Round k down to the next smaller integer.
`'weight'`	If `k=i+f` where `i` is the integer part and `f` is the fraction, compute a weighted mean with weight `(1-f)` for the `(i+1)`th and `(n-i)`th values, and full weight for the values between them.

`m = trimmean(x,percent,flag,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   **Example 1**

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
```

Example 2

Generate random data from the t distribution, which tends to have outliers:

```
reset(RandStream.getDefaultStream)
x = trnd(1,40,1);
probplot(x)
```

Probability plot for Normal distribution
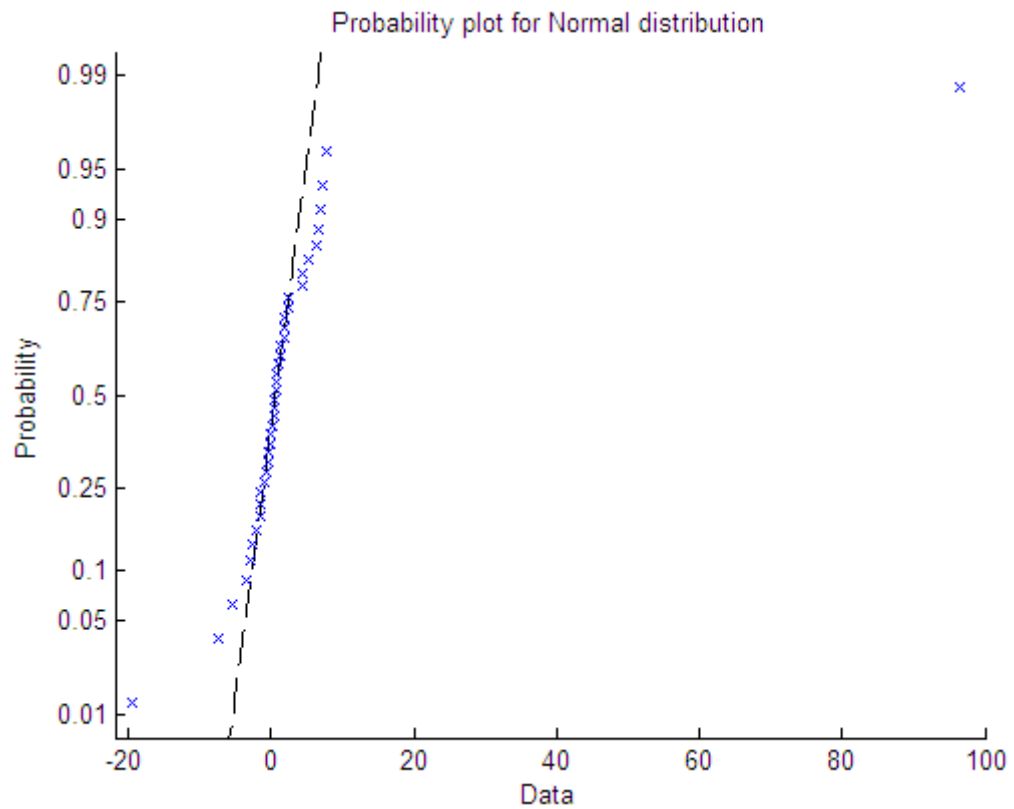
Though the distribution is symmetric around zero, there are several outliers which will affect the mean. The trimmed mean is much closer to zero, which is much more representative of the data:

```
mean(x)

ans =
2.7991

trimmean(x,25)

ans =
```

```
        0.8797
```

See Also   mean, median, geomean, harmmean

Purpose	Student's t random numbers

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.

Examples
```
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 random, tpdf, tcdf, tinv, tstat

tstat

Purpose	Student's t mean and variance

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 v is zero for values of v greater than 1. If v is one, the mean does not exist. The variance for values of v greater than 2 is $v/(v-2)$.

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   tpdf, tcdf, tinv, trnd

Purpose   One-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, $\mu = 0$ (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

Examples   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

$$t = \frac{\overline{x} - \overline{y}}{\sqrt{\frac{s_x^2}{n} + \frac{s_y^2}{m}}}$$

where $\overline{x}$ and $\overline{y}$ 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 $100*(1 - \text{alpha})\%$ 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').

Examples   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 a = 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 a. 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

Purpose	Tree type
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.
Examples	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
```

References   [1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

See Also   classregtree

Purpose   Type of partition

Description   The type of validation partition. It is `'kfold'`, `'holdout'`, `'leaveout'`, or `'resubstitution'`.

See Also   trainsize

qrandset.Type property

Purpose   Subclass

Description   Subclass (string). Read only. Set at construction.

Purpose   Convert categorical array to unsigned 8-bit integers

Syntax   `B = uint8(A)`

Description   `B = uint8(A)` converts the categorical array `A` to unsigned 8-bit integers. Each element of `B` contains the internal categorical level code for the corresponding element of `A`. Undefined elements of `A` are assigned the value 0 in `B`. If `A` contains more than `intmax('uint8')` levels, the internal codes will saturate to `intmax('uint8')` when cast to `int8`.

See Also   `double`, `int8`

categorical.uint16

Purpose

Convert categorical array to unsigned 16-bit integers

Syntax

B = uint16(A)

Description

B = uint16(A) converts the categorical array A to unsigned 16-bit integers. Each element of B contains the internal categorical level code for the corresponding element of A.

Undefined elements of A are assigned the value 0 in B.

See Also

double, int16

Purpose   Convert categorical array to unsigned 32-bit integers

Syntax   B = uint32(A)

Description   B = uint32(A) converts the categorical array A to unsigned 32-bit integers. Each element of B contains the internal categorical level code for the corresponding element of A.

Undefined elements of A are assigned the value 0 in B.

See Also   double, int32

categorical.uint64

Purpose   Convert categorical array to unsigned 64-bit integers

Syntax   B = uint64(A)

Description   B = uint64(A) converts the categorical array A to unsigned 64-bit
                  integers. Each element of B contains the internal categorical level code
                  for the corresponding element of A.

                  Undefined elements of A are assigned the value 0 in B.

See Also   double, int64

Purpose   Text label for undefined levels

Description Text label for undefined levels. Constant property with value
                 `'<undefined>'`.

dataset.unique

Purpose   Unique observations in dataset array

Syntax
```
B = unique(A)
B = unique(A,vars)
[B,i,j] = unique(A)
[...] = unique(A,vars,'first')
```

Description   `B = unique(A)` returns a copy of the dataset A that contains only the sorted unique observations. A must contain only variables whose class has a `unique` method, including:

- numeric
- character
- logical
- categorical
- cell arrays of strings

For a variable with multiple columns, its class's `unique` method must support the `'rows'` flag.

`B = unique(A,vars)` returns a dataset that contains only one observation for each unique combination of values for the variables in A specified in `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 includes all variables from A. The values in B for the variables not specified in `vars` are taken from the last occurrence among observations in A with each unique combination of values for the variables specified in `vars`.

`[B,i,j] = unique(A)` also returns index vectors `i` and `j` such that `B = A(i,:)` and `A = B(j,:)`.

`[...] = unique(A,vars,'first')` returns the vector `i` to index the first occurrence of each unique observation in A. `unique(A,vars,'last')`, the default, returns the vector `i` to index

the last occurrence. Specify `vars` as `[]` to use the default value of all variables.

See Also   `dataset`, `set`, `subsasgn`

dataset.units property

Purpose   Units of variables in data set

Description   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.

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

$$p = F(x|N) = \frac{floor(x)}{N} I_{(1,\,...,\,N)}(x)$$

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`, `unidpdf`, `unidinv`, `unidstat`, `unidrnd`, `mle`

unidinv

Purpose	Discrete uniform inverse 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, unidstat, unidrnd

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

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

                   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, unidstat, unidrnd

unidrnd

Purpose	Discrete uniform random numbers

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`.

Examples   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   random, unidpdf, unidcdf, unidinv, unidstat

Purpose   Discrete uniform mean of and variance

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   unidpdf, unidcdf, unidinv, unidrnd

unifcdf

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, unifpdf, unifinv, unifstat, unifit, unifrnd

Purpose   Continuous uniform inverse 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, unifpdf, unifstat, unifit, unifrnd

unifit

Purpose	Continuous uniform parameter estimates

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.

Examples
```
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   `mle, unifpdf, unifcdf, unifinv, unifstat, unifrnd`

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, unifstat, unifit, unifrnd

unifrnd

Purpose	Continuous uniform random numbers

Syntax	`R = unifrnd(A,B)`
	`R = unifrnd(A,B,m,n,...)`
	`R = unifrnd(A,B,[m,n,...])`

Description `R = unifrnd(A,B)` returns an array R of random numbers generated from the continuous uniform distributions with lower and upper endpoints specified by A and B, respectively. If A and B are arrays, `R(i,j)` is generated from the distribution specified by the corresponding elements of A and B. If either A or B is a scalar, it is expanded to the size of the other input.

`R = unifrnd(A,B,m,n,...)` or `R = unifrnd(A,B,[m,n,...])` returns an m-by-n-by-... array. If A and B are scalars, all elements of R are generated from the same distribution. If either A or B is an array, they must be m-by-n-by-... .

Examples Generate one random number each from the continuous uniform distributions on the intervals (0,1), (0,2), ..., (0,5):

```
a = 0; b = 1:5;
r1 = unifrnd(a,b)
r1 =
    0.8147    1.8116    0.3810    3.6535    3.1618
```

Generate five random numbers each from the same distributions:

```
B = repmat(b,5,1);
R = unifrnd(a,B)
R =
    0.0975    0.3152    0.4257    2.6230    3.7887
    0.2785    1.9412    1.2653    0.1428    3.7157
    0.5469    1.9143    2.7472    3.3965    1.9611
    0.9575    0.9708    2.3766    3.7360    3.2774
    0.9649    1.6006    2.8785    2.7149    0.8559
```

Generate five random numbers from the continuous uniform distribution on (0,2):

```
r2 = unifrnd(a,b(2),1,5)
r2 =
    1.4121    0.0637    0.5538    0.0923    0.1943
```

See Also   rand, random, unifpdf, unifcdf, unifinv, unifstat, unifit

unifstat

Purpose	Continuous uniform mean and variance
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   unifpdf, unifcdf, unifinv, unifit, unifrnd

Purpose Set union for categorical arrays

Syntax `C = union(A,B)`
`[C,IA,IB] = union(A,B)`

Description `C = union(A,B)` when A and B are categorical arrays returns a categorical vector C containing the combined values from A and B but with no repetitions. The result C is sorted. The set of categorical levels for C is the sorted union of the sets of levels of the inputs, as determined by their labels.

`[C,IA,IB] = union(A,B)` also returns index vectors IA and IB such that C is a sorted combination of the elements A(IA) and B(IB).

See Also `intersect`, `ismember`, `setdiff`, `setxor`, `unique`

categorical.unique

Purpose   Unique values in categorical array

Syntax   B = unique(A)
                   [B,I,J] = unique(A)
                   [B,I,J] = unique(A,'first')

Description   B = unique(A) returns a categorical array containing the unique
                   elements of A, sorted by the order of A's levels.

                   [B,I,J] = unique(A) also returns index vectors I and J such that
                   B = A(I) and A = B(J).

                   [B,I,J] = unique(A,'first') returns the vector I to index the first
                   occurrence of each unique value in A. unique(A,'last'), the default,
                   returns the vector I to index the last occurrence.

See Also   intersect, ismember, setdiff, setxor, union

Purpose   Upper Pareto tails parameters

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.

Examples   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

dataset.UserData property

Purpose   Variable containing additional information associated with data set

Description Any variable containing additional information to be associated with the data set. The default is an empty array.

ProbDistUnivParam.var

Purpose	Return variance of ProbDistUnivParam object
Syntax	*V* = var(*PD*)
Description	*V* = var(*PD*) returns *V*, the variance of the ProbDistUnivParam object *PD*.
Inputs	*PD* An object of the class ProbDistUnivParam.
Outputs	*V* The variance of the ProbDistUnivParam object *PD*.
See Also	var

classregtree.varimportance

Purpose   Compute embedded estimates of input feature importance

Syntax   `imp = varimportance(t)`

Description `imp = varimportance(t)` computes estimates of input feature importance for tree `t` by summing changes in the risk due to splits on every feature. The returned vector `imp` has one element for each input variable in the data used to train this tree. At each node, the risk is estimated as node impurity if impurity was used to split nodes and node error otherwise. This risk is weighted by the node probability. Variable importance associated with this split is computed as the difference between the risk for the parent node and the total risk for the two children.

See Also   `risk`

Purpose   Variable names

Description   The VarNames property is a cell array containing the names of the predictor variables (features). These names are taken from the optional 'names' parameter that supplied to TreeBagger. The default names are 'x1', 'x2', etc.

dataset.VarNames property

Purpose   Cell array giving names of variables in data set

Description   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.

Purpose   Variable names

Description   The VarNames property is a cell array containing the names of the
                   predictor variables (features). TreeBagger takes these names from the
                   optional 'names' parameter. The default names are 'x1', 'x2', etc.

vartest

Purpose　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.

Examples   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 [].

Examples   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 2-33.) The function tests for equal variances across groups.

`vartestn` treats NaNs 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.

Examples

Does the variance of mileage measurements differ significantly from one model year to another?

```
load carsmall
p = vartestn(MPG,Model_Year)
p =
    0.8327
```

Group Summary Table

Group	Count	Mean	Std Dev
70	29	17.6897	5.33923
76	34	21.5735	5.8893
82	31	31.7097	5.39255
Pooled	94	23.7181	5.562

Bartlett's statistic	0.36619
Degrees of freedom	2
p-value	0.83269
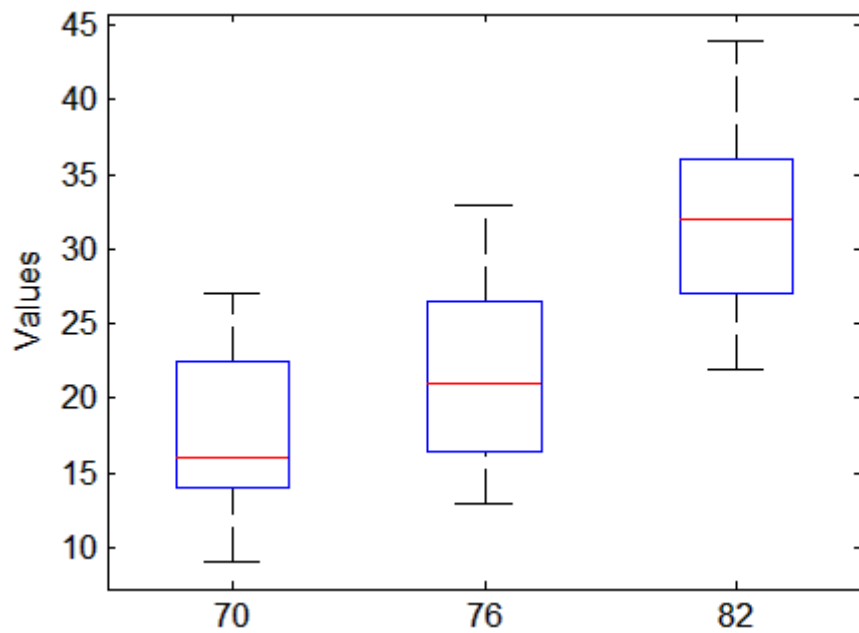
See Also   "Grouped Data" on page 2-33

vartest, vartest2, anova1

Purpose   Vertical concatenation for categorical arrays

Syntax   C = vertcat(dim,A,B,...)
                  C = vertcat(A,B)

Description   C = vertcat(dim,A,B,...) vertically concatenates the categorical
                  arrays A,B,... . For matrices, all inputs must have the same number
                  of rows. For n-D arrays, all inputs must have the same sizes except in
                  the second dimension. The set of categorical levels for C is the sorted
                  union of the sets of levels of the inputs, as determined by their labels.

                  C = vertcat(A,B) is called for the syntax [A B].

See Also   cat, horzcat

dataset.vertcat

Purpose	Vertical concatenation for dataset arrays
Syntax	ds = vertcat(ds1, ds2, ...)
Description	ds = vertcat(ds1, ds2, ...) vertically concatenates the dataset arrays ds1, ds2, ... Observation names, when present, must be unique across datasets. vertcat fills in default observation names for the output when some of the inputs have names and some do not.
	Variable names for all dataset arrays must be identical except for order. vertcat concatenates by matching variable names.
See Also	cat, horzcat

Purpose   Plot 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.

Examples   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)
```

References   [1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

See Also classregtree, eval, prune, test

wblcdf

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, wblpdf, wblinv, wblstat, wblfit, wbllike, wblrnd

wblfit

Purpose   Weibull parameter estimates

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.

Examples
```
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   mle, wbllike, wblpdf, wblcdf, wblinv, wblstat, wblrnd

wblinv

Purpose	Weibull inverse 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

$$\log a + \frac{\log q}{b}$$

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

$$x = F^{-1}(p \mid a,b) = -a\left[\ln\left(1-p\right)\right]^{1/b} I_{[0,1]}(p)$$

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   icdf, wblcdf, wblpdf, wblstat, wblfit, wbllike, wblrnd

wbllike

Purpose	Weibull negative log-likelihood

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

$$(-\log L) = -\log \prod_{i=1}^{n} f(a,b|x_i) = -\sum_{i=1}^{n} \log f(a,b|x_i)$$

where f is the Weibull pdf.

`wbllike` is a utility function for maximum likelihood estimation.

Examples 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
```

References [1] Patel, J. K., C. H. Kapadia, and D. B. Owen. *Handbook of Statistical Distributions*. New York: Marcel Dekker, 1976.

See Also wblfit, wblpdf, wblcdf, wblinv, wblstat, wblrnd

wblpdf

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
```

References   [1] Devroye, L. *Non-Uniform Random Variate Generation.* New York: Springer-Verlag, 1986.

See Also   pdf, wblcdf, wblfit, wblinv, wbllike, wblplot, wblrnd, wblstat

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.

Examples
```
r = wblrnd(1.2,1.5,50,1);
wblplot(r)
```

wblplot

Weibull Probability Plot
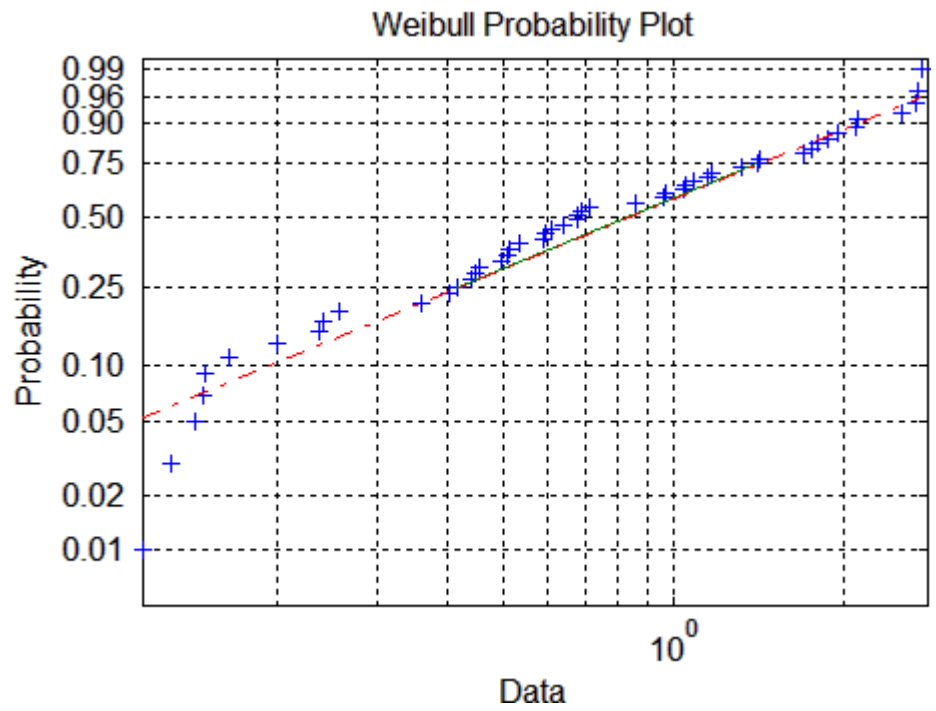
See Also   probplot, normplot, wblcdf, wblfit, wblinv, wbllike, wblpdf, wblrnd, wblstat

Purpose   Weibull random numbers

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.

Examples   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

References   [1] Devroye, L. *Non-Uniform Random Variate Generation*. New York:
                  Springer-Verlag, 1986.

See Also   random, wblpdf, wblcdf, wblinv, wblstat, wblfit, wbllike

wblstat

Purpose   Weibull mean and variance

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   wblpdf, wblcdf, wblinv, wblfit, wbllike, wblrnd

wishrnd

Purpose   Wishart random numbers

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

TreeBagger.X property

Purpose   X data used to create ensemble

Description   The X property is a numeric matrix of size Nobs-by-Nvars, where Nobs is the number of observations (rows) and Nvars is the number of variables (columns) in the training data. This matrix contains the predictor (or feature) values.

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* is one of the following strings:

- 'linear' — Constant and linear terms. This is 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* specifies 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   regstats, rstool, candexch, candgen, cordexch, and rowexch.

TreeBagger.Y property

Purpose   Y data used to create ensemble

Description   The Y property is an array of true class labels for classification, or response values for regression. Y can be a numeric column vector, a character matrix, or a cell array of strings.

Purpose   Standardized *z*-scores

Syntax   Z = zscore(X)
                   [Z,mu,sigma] = zscore(X)
                   [...] = zscore(X,1)
                   [...] = zscore(X,flag,dim)

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.

                   [...] = zscore(X,1) normalizes X using std(X,1), that is, by
                   computing the standard deviation(s) using *n* rather than *n*–1, where *n*
                   is the length of the dimension along which zscore works. zscore(X,0)
                   is the same as zscore(X).

                   [...] = zscore(X,flag,dim) standardizes X by working along the
                   dimension dim of X. Set flag to 0 to use the default normalization by
                   n–1; set flag to 1 to use *n*.

Examples   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))
```
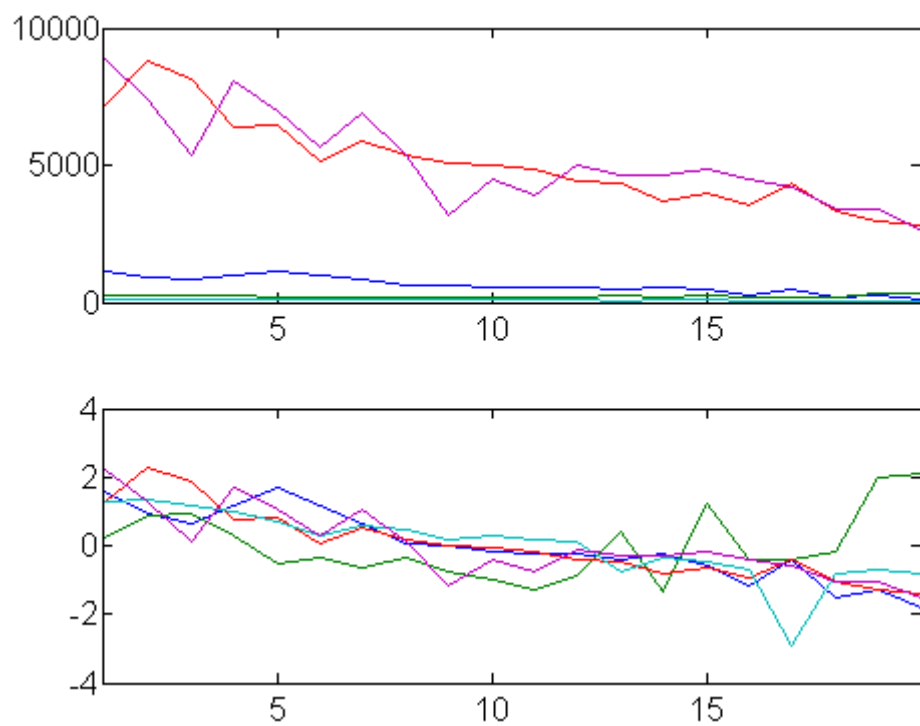
See Also   mean, std

Purpose   *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.

Examples   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 α = 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

ztest

A

Data Sets

Statistics Toolbox software includes the sample data sets in the following table.

To load a data set into the MATLAB workspace, type:

```
load filename
```

where *filename* is one of the files listed in the table.

Data sets contain individual data variables, description variables with references, and dataset arrays encapsulating the data set and its description, as appropriate.

File	Description of Data Set
acetylene.mat	Chemical reaction data with correlated predictors
arrhythmia.mat	Cardiac arrhythmia data from the UCI machine learning repository
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 1936 iris data
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
hospital.mat	Simulated hospital data
imports-85.mat	1985 Auto Imports Database from the UCI repository

File	Description of Data Set
ionosphere.mat	Ionosphere dataset from the UCI machine learning repository
kmeansdata.mat	Four-dimensional clustered data
lawdata.mat	Grade point average and LSAT 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
ovariancancer.mat	Grouped observations on 4000 predictors
parts.mat	Dimensional run-out on 36 circular parts
polydata.mat	Sample data for polynomial fitting
popcorn.mat	Popcorn yield by popper type and brand
reaction.mat	Reaction kinetics 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)
spectra.mat	NIR spectra and octane numbers of 60 gasoline samples
stockreturns.mat	Simulated stock returns

Distribution Reference

Bernoulli Distribution

Definition of the Bernoulli Distribution

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

Beta Distribution

In this section...
"Definition" on page B-4
"Background" on page B-4
"Parameters" on page B-5
"Example" on page B-6

Definition

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

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 v degrees of freedom, then the following transformation generates X, which is beta distributed.

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

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

This relationship is used to compute values of the t cdf and inverse function as well as generating t distributed random numbers.

Parameters

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

The shape of the beta distribution is quite variable depending on the values of the parameters, as illustrated by the plot below.
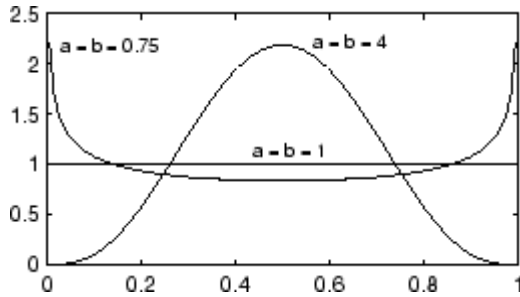
The constant pdf (the flat line) shows that the standard uniform distribution is a special case of the beta distribution.

Binomial Distribution

Definition

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 $k = 0, 1, 2, \ldots n$, where it is nonzero.

Background

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.

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

Parameters

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

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,'+')
```
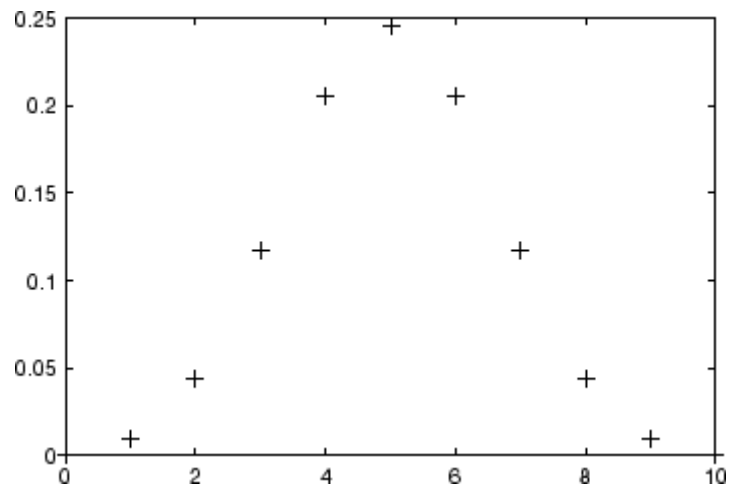
Birnbaum-Saunders Distribution

In this section...
"Definition" on page B-10
"Background" on page B-10
"Parameters" on page B-10

Definition

The Birnbaum-Saunders distribution has the density function

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

with scale parameter β > 0 and shape parameter γ > 0, for x > 0.

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

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

has a standard normal distribution.

Background

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.

Parameters

See `mle`, `dfittool`.

Chi-Square Distribution

Definition

The χ^2 pdf is

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

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

Background

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

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

This relationship is used to calculate confidence intervals for the estimate of the normal parameter σ^2 in the function `normfit`.

Example

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

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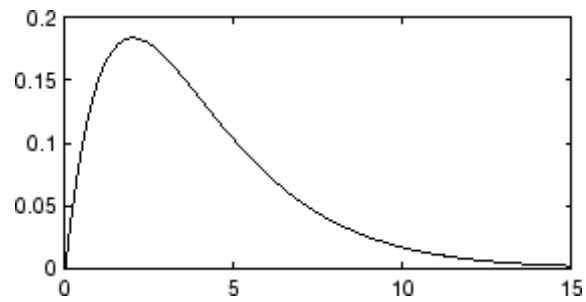
Copulas

See "Copulas" on page 5-99.

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

The exponential pdf is

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

Background

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.

Parameters

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 $\mu = 700$.

```
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

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)
```
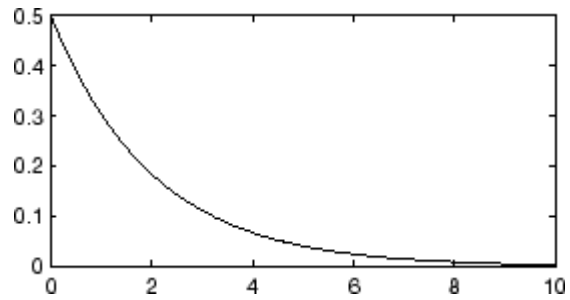
Extreme Value Distribution

In this section...

Definition

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, then log T has an extreme value distribution with parameters $\mu = \log a$ and $\sigma = 1/b$.

Background

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 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.

Parameters

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
```

Example

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

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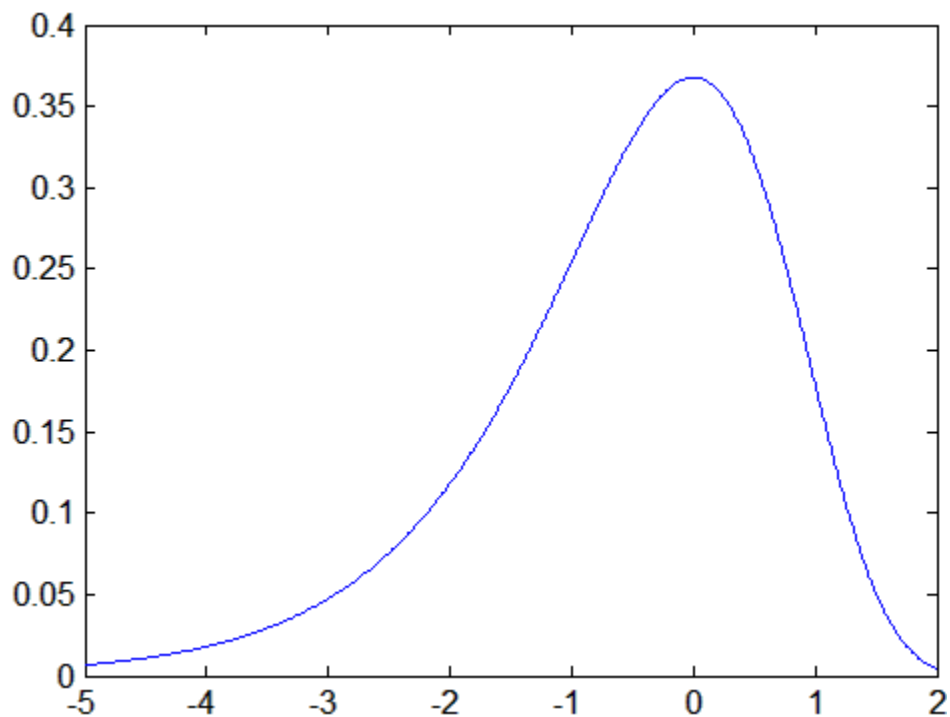

The extreme value distribution is skewed to the left, and its general shape remains the same for all parameter values. The location parameter, mu, shifts the distribution along the real line, and the scale parameter, sigma, expands or contracts the distribution. This example plots the probability function 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'}, ...
       'Location','NW')
xlabel('x')
ylabel('f(x|mu,sigma)')
```
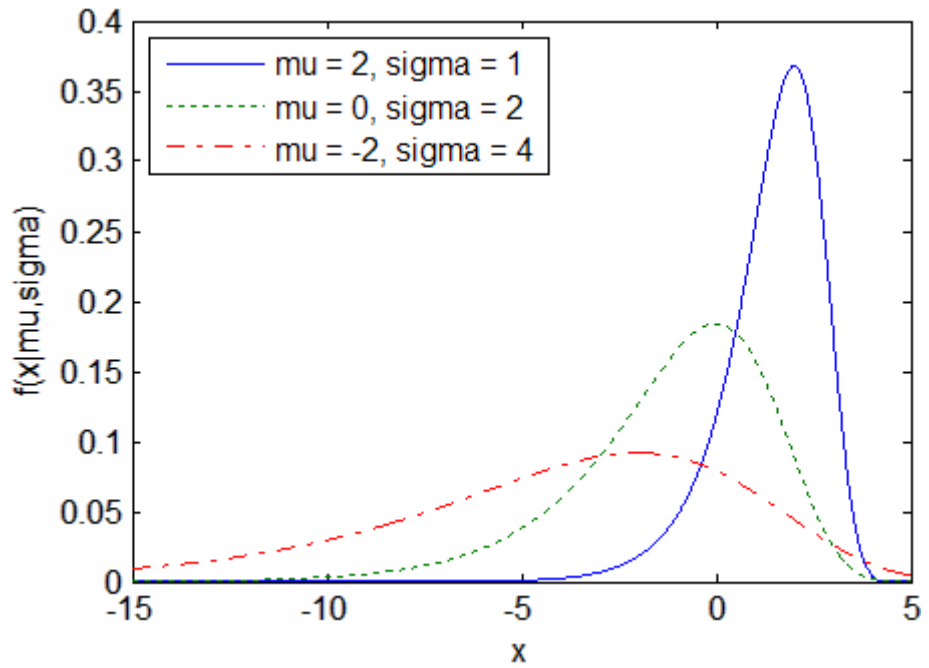
F Distribution

In this section...
"Definition" on page B-22
"Background" on page B-22
"Example" on page B-23

Definition

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

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.

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

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

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)
```
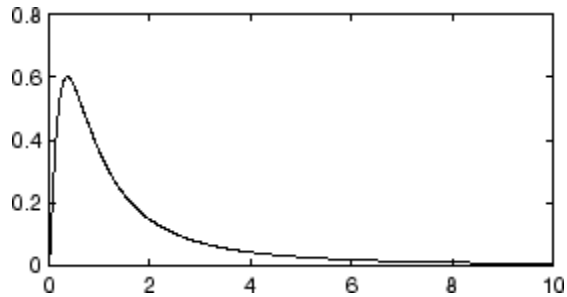
Gamma Distribution

Definition

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

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.

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

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.

Parameters

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

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,'-.')
```
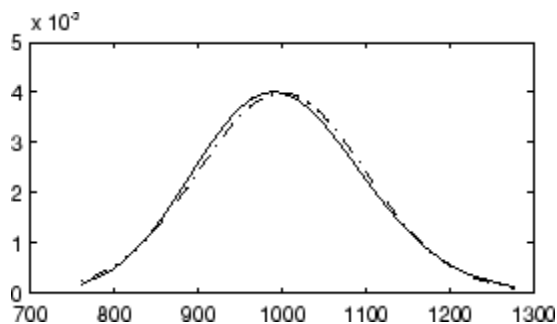
Gaussian Distribution

See "Normal Distribution" on page B-75.

Gaussian Mixture Distributions

See the discussion of the `gmdistribution` class in the "Gaussian Mixture Models" on page 5-91 section of "Multivariate Modeling" on page 5-91 and the "Gaussian Mixture Models" on page 10-28 section of Chapter 10, "Cluster Analysis".

Generalized Extreme Value Distribution

Definition

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

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

B-29

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 or wblfit, respectively. Finally, the Type II (Frechet) case is equivalent to taking the reciprocal of values from a standard Weibull distribution.

Parameters

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 =
    0.2438    1.1760    5.8045
```

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',[.8 .8 1])
xgrid = linspace(2,20,1000);
line(xgrid,nblocks*...
    gevpdf(xgrid,paramEsts(1),paramEsts(2),paramEsts(3)));
```
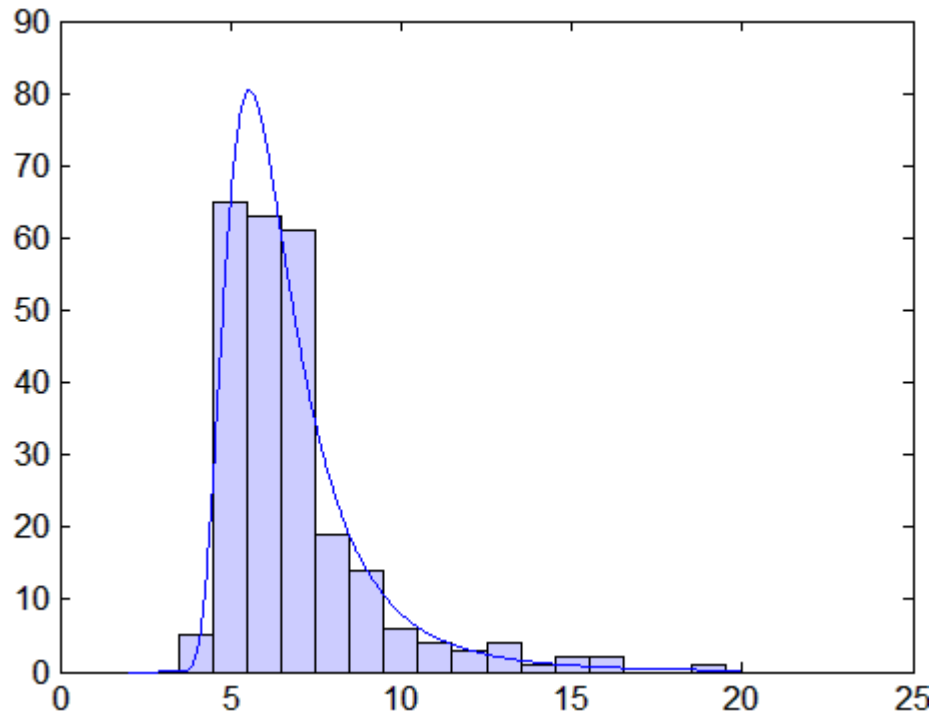

Example

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'});
```
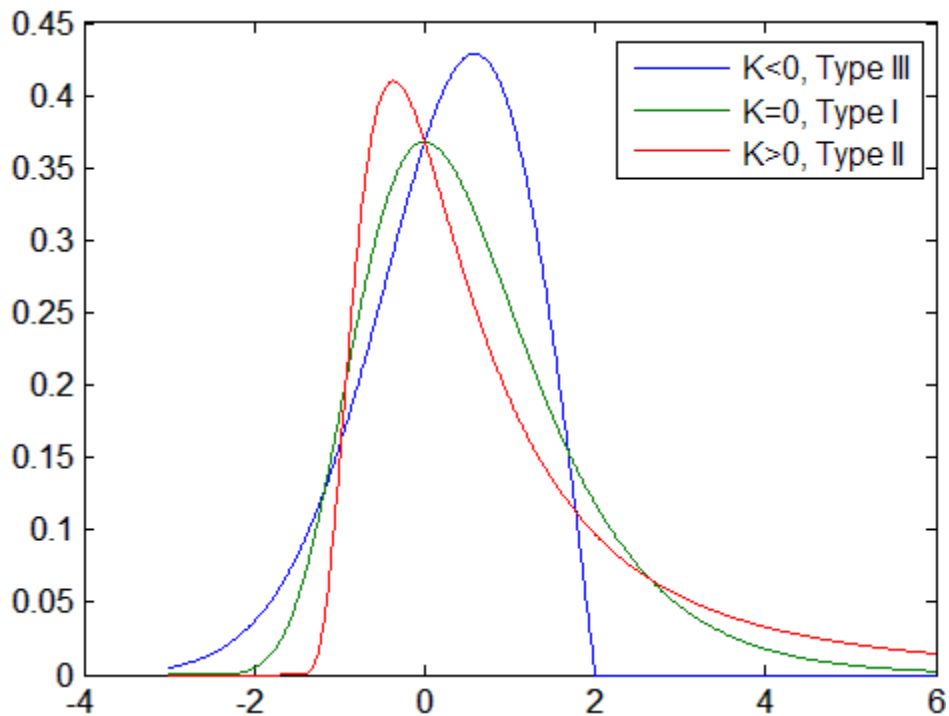

Notice that for k > 0, the distribution has zero probability density for x such that

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

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

In this section...
"Definition" on page B-34
"Background" on page B-34
"Parameters" on page B-35
"Example" on page B-36

Definition

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

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

for $\theta < x$, when $k > 0$, or for $\theta < x < -\sigma/k$ when $k < 0$.

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

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

for $\theta < x$.

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

Background

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.

The generalized Pareto distribution is used in the tails of distribution fit objects of the `paretotails` class.

Parameters

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 =
    0.1267    0.8134
```

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',[.8 .8 1])
xgrid = linspace(2,12,1000);
line(xgrid,.5*length(y)*...
     gppdf(xgrid,paramEsts(1),paramEsts(2),2));
```
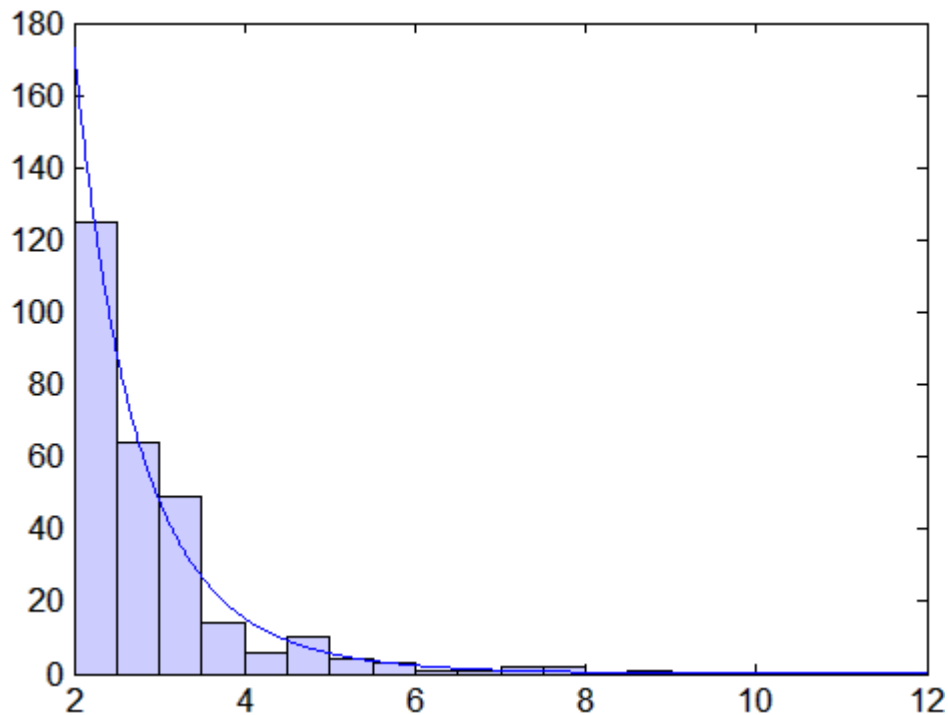

Example

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'});
```
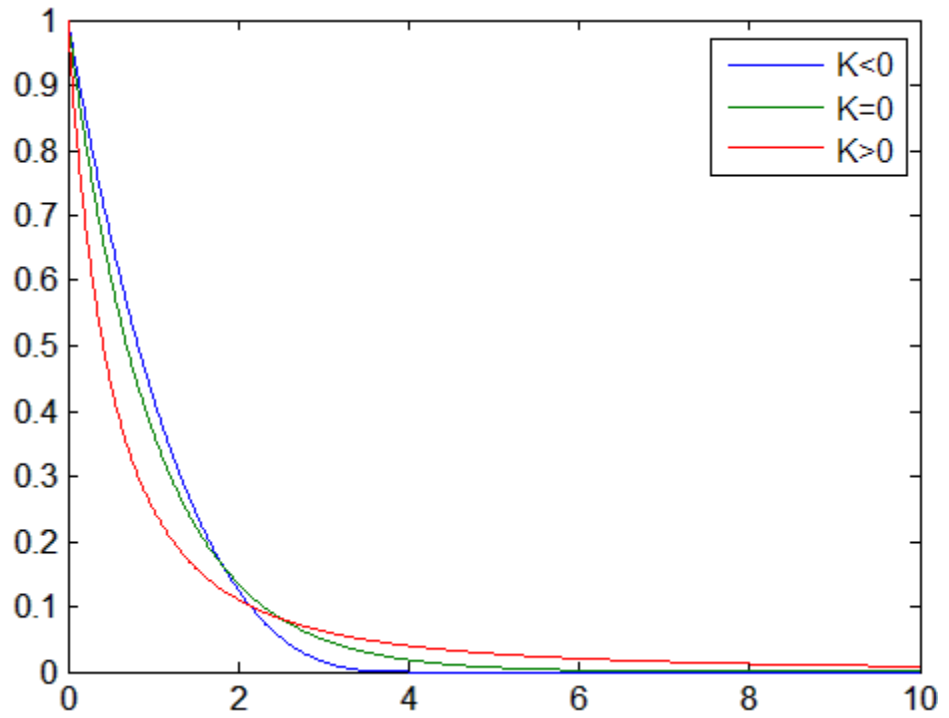

Notice that for k < 0, the distribution has zero probability density for $x > -\frac{\sigma}{k}$, while for k ≥ 0, there is no upper bound.

Geometric Distribution

Definition

The geometric pdf is

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

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

Background

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

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)
```
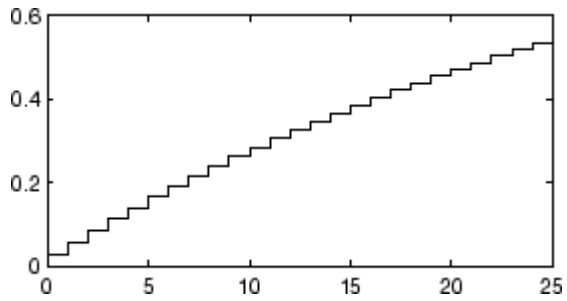
Hypergeometric Distribution

In this section...
"Definition" on page B-40
"Background" on page B-40
"Example" on page B-41

Definition

The hypergeometric pdf is

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

Background

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

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)
```
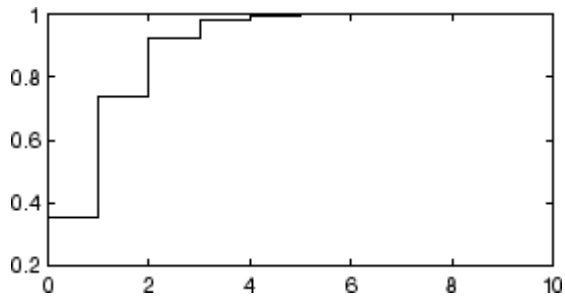
Inverse Gaussian Distribution

In this section...
"Definition" on page B-42
"Background" on page B-42
"Parameters" on page B-42

Definition

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

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.

Parameters

See `mle`, `dfittool`.

Inverse Wishart Distribution

Definition

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

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

Only random matrix generation is supported for the inverse Wishart, and only for nonsingular Σ and v greater than $d - 1$.

Johnson System

See "Pearson and Johnson Systems" on page 5-84.

Logistic Distribution

Definition

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

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.

Parameters

See `mle`, `dfittool`.

Loglogistic Distribution

Definition

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.

Parameters

See `mle`, `dfittool`.

Lognormal Distribution

Definition

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

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

Suppose the income of a family of four in the United States follows a lognormal distribution with $\mu = \log(20{,}000)$ and $\sigma^2 = 1.0$. 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'))
```
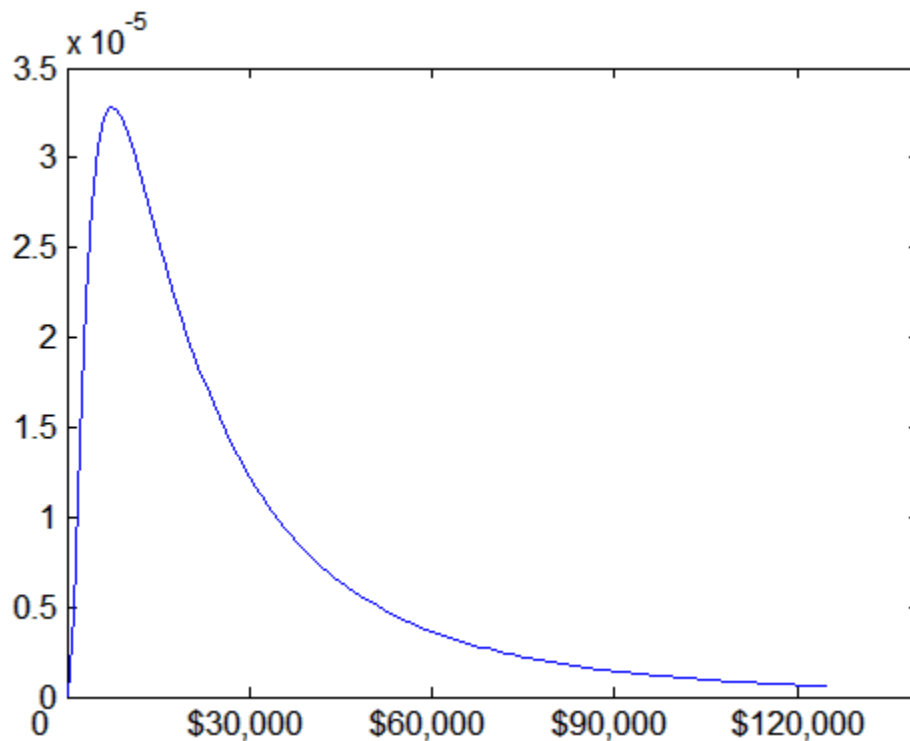
Multinomial Distribution

In this section...
"Definition" on page B-49
"Background" on page B-49
"Example" on page B-49

Definition

The multinomial pdf is

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

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

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 $p_1, \ldots, p_k$.

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

The following uses mnpdf to produce a visualization of a trinomial 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')
```

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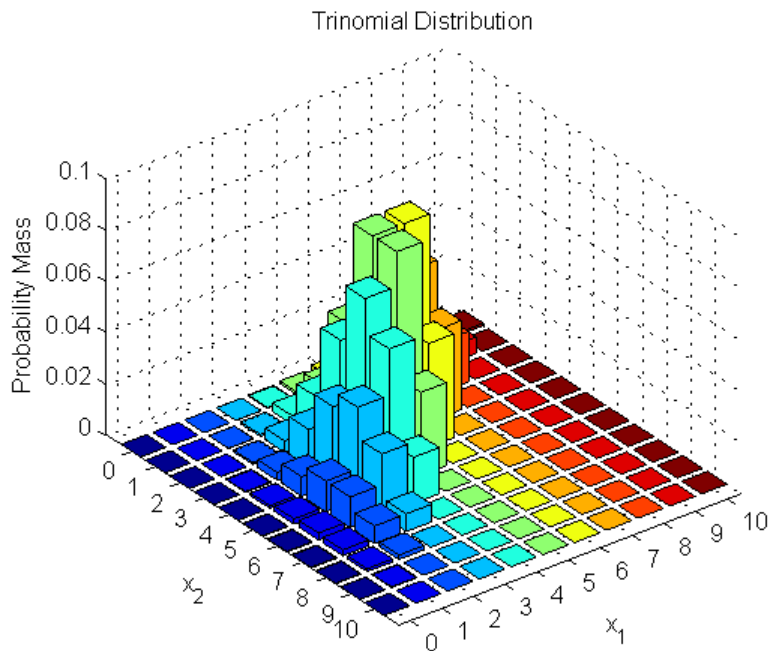
Note that the visualization does not show x_3, which is determined by the constraint $x_1 + x_2 + x_3 = n$.

Multivariate Gaussian Distribution

See "Multivariate Normal Distribution" on page B-53.

Multivariate Normal Distribution

In this section...
"Definition" on page B-53
"Background" on page B-53
"Example" on page B-54

Definition

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

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

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. Only random vector generation is supported 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 software uses that orientation.

Background

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

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');
```
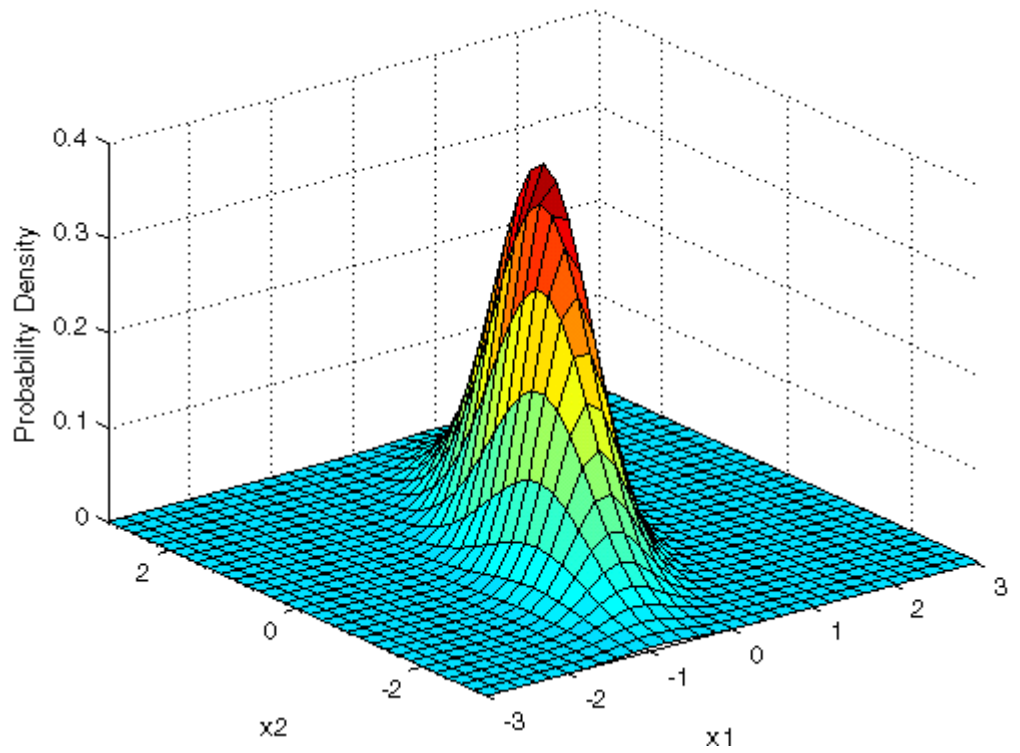
```
F = mvncdf([X1(:) X2(:)],mu,Sigma);
F = reshape(F,length(x2),length(x1));
surf(x1,x2,F);
caxis([min(F(:))-.5*range(F(:)),max(F(:))]);
axis([-3 3 -3 3 0 1])
xlabel('x1'); ylabel('x2'); zlabel('Cumulative Probability');
```
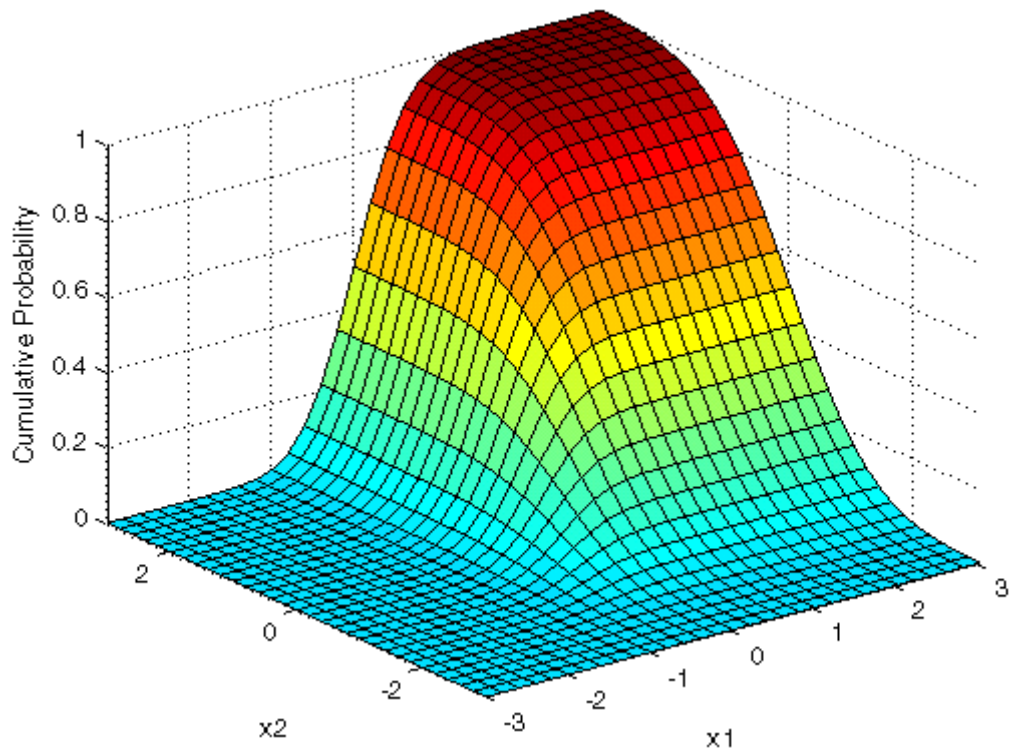
Since the bivariate normal distribution is defined on the plane, you can also compute cumulative probabilities over rectangular regions. For example, this contour plot illustrates the computation that follows, of the probability contained within the unit square.

```
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');
```
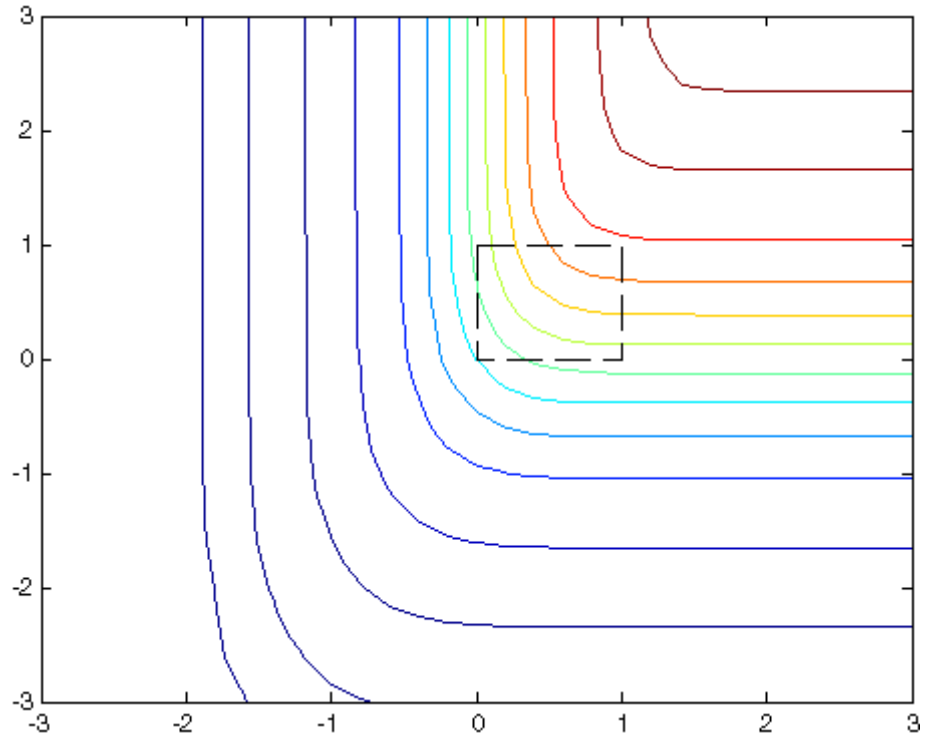
```
mvncdf([0 0],[1 1],mu,Sigma)
ans =
      0.20974
```

Computing a multivariate cumulative probability requires significantly more work than computing a univariate probability. By default, the `mvncdf` function computes values to less than full machine precision, and returns an estimate of the error as an optional second output:

```
[F,err] = mvncdf([0 0],[1 1],mu,Sigma)
F =
      0.20974
err =
       1e-008
```

B-57

Multivariate t Distribution

In this section...
"Definition" on page B-58
"Background" on page B-58
"Example" on page B-59

Definition

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 *v* 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, only random number generation is supported. 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 software uses that orientation.

Background

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, *v*. *v* 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

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');
```
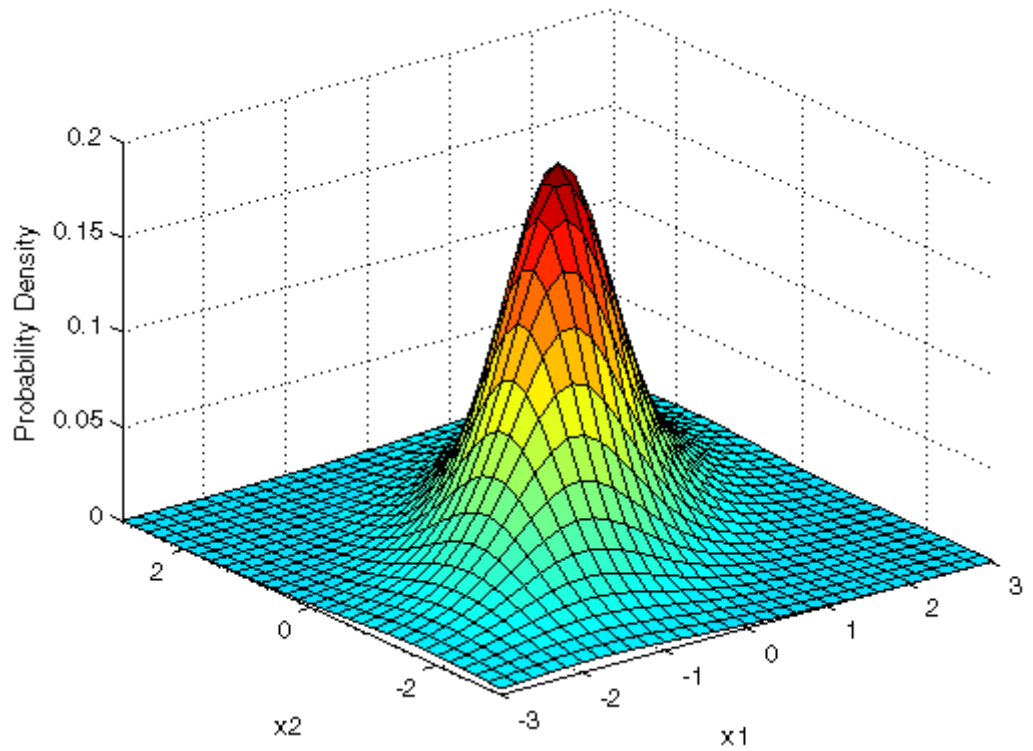
```
F = mvtcdf([X1(:) X2(:)],Rho,nu);
F = reshape(F,length(x2),length(x1));
surf(x1,x2,F);
caxis([min(F(:))-.5*range(F(:)),max(F(:))]);
axis([-3 3 -3 3 0 1])
xlabel('x1'); ylabel('x2'); zlabel('Cumulative Probability');
```
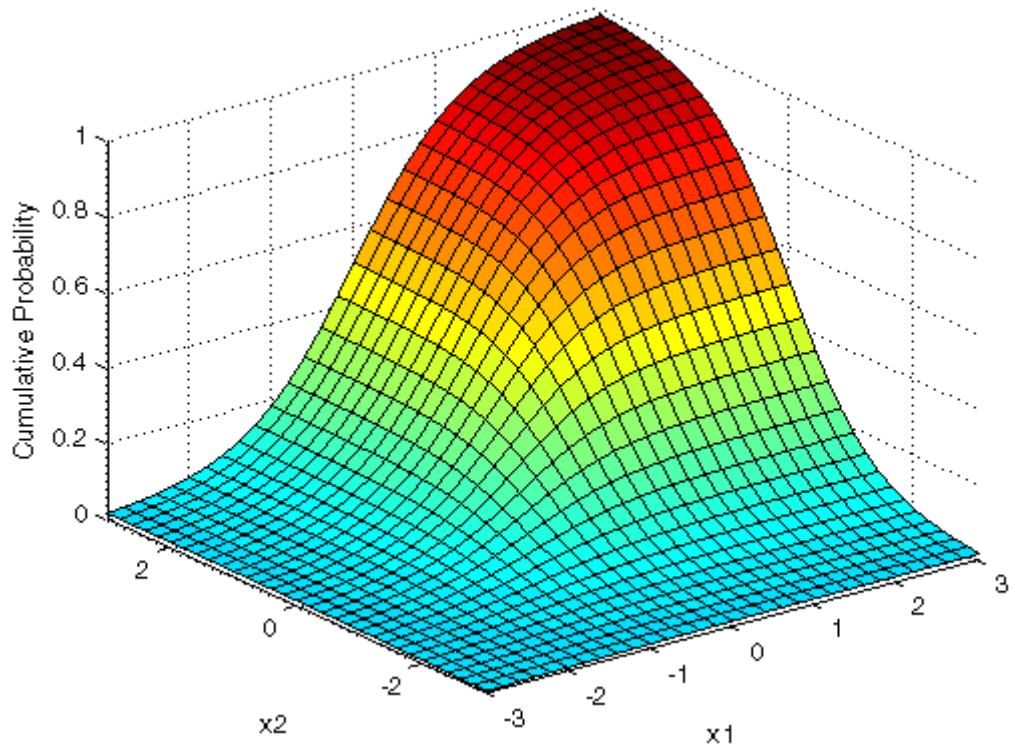
Since the bivariate Student's *t* distribution is defined on the plane, you can also compute cumulative probabilities over rectangular regions. For example, this contour plot illustrates the computation that follows, of the probability contained within the unit square.

```
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');
```
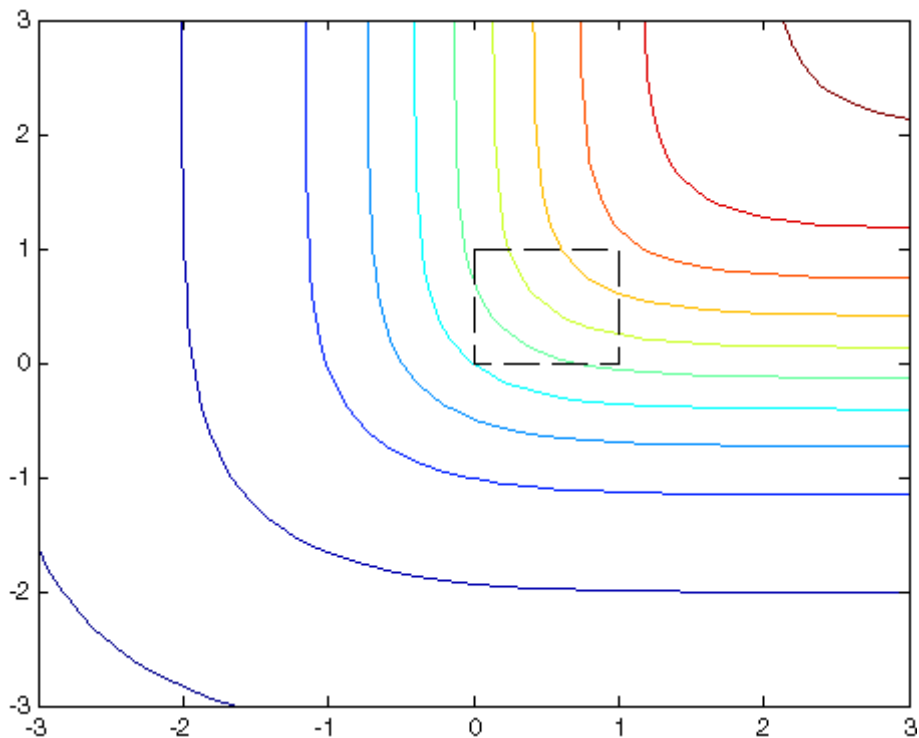
```
mvtcdf([0 0],[1 1],Rho,nu)
ans =
      0.14013
```

Computing a multivariate cumulative probability requires significantly more work than computing a univariate probability. By default, the `mvtcdf` function computes values to less than full machine precision, and returns an estimate of the error as an optional second output:

```
[F,err] = mvtcdf([0 0],[1 1],Rho,nu)
F =
      0.14013
err =
       1e-008
```

Nakagami Distribution

In this section...
"Definition" on page B-63
"Background" on page B-63
"Parameters" on page B-63

Definition

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

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.

Parameters

See `mle`, `dfittool`.

Negative Binomial Distribution

In this section...
"Definition" on page B-64
"Background" on page B-64
"Parameters" on page B-65
"Example" on page B-66

Definition

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

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.

Parameters

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 =
    1.0060    0.1109
pci =
    0.2152    0.0171
    1.7968    0.2046
```

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')
```
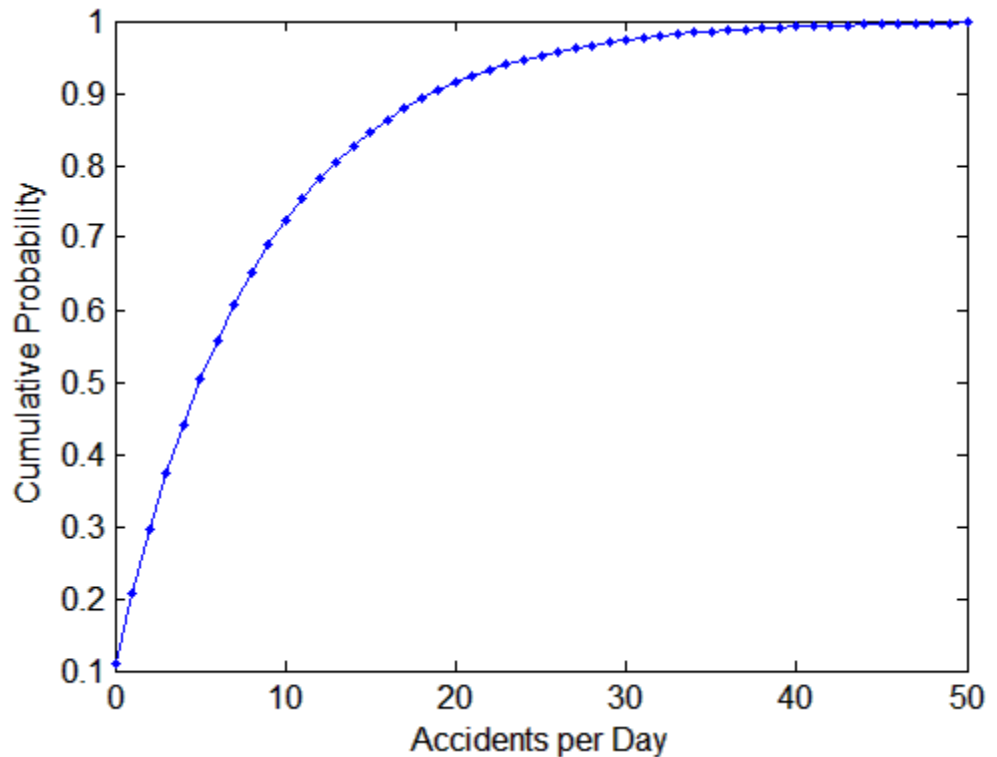

Example

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)')
```
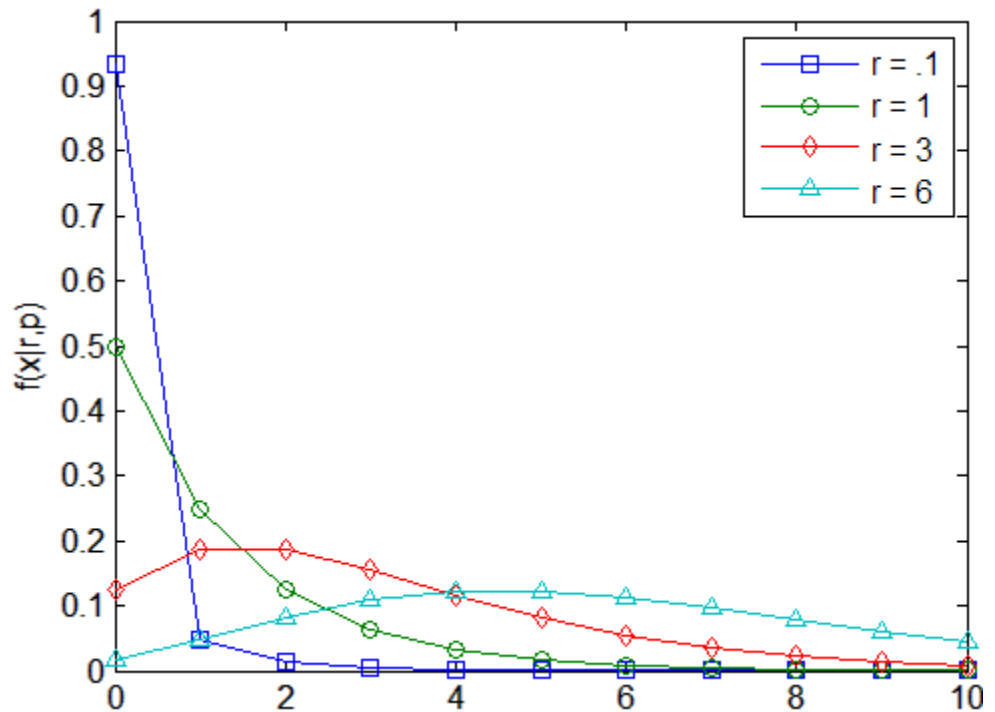
Noncentral Chi-Square Distribution

In this section...
"Definition" on page B-68
"Background" on page B-68
"Example" on page B-69

Definition

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. The cumulative distribution function is computed 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

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

where δ is the noncentrality parameter.

Background

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 Rician or generalized Rayleigh distribution.

Example

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,'-')
```
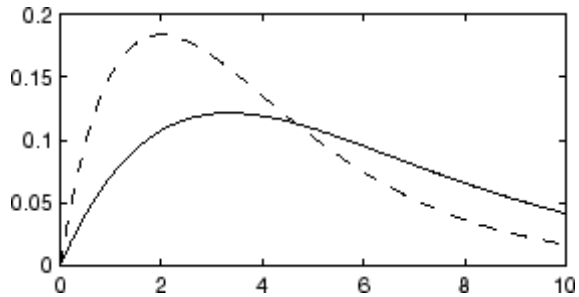
Noncentral F Distribution

In this section...
"Definition" on page B-70
"Background" on page B-70
"Example" on page B-70

Definition

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.

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

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

Background

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

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,'-')
```
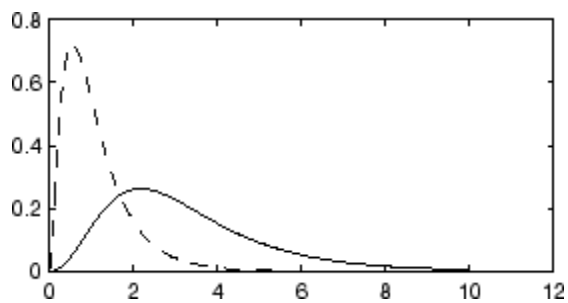
Noncentral t Distribution

Definition

The most general representation of the noncentral t distribution is quite complicated. Johnson and Kotz [58] give a formula for the probability that a noncentral t variate falls in the range [$-t$, t].

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

$I(x | a, b)$ 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

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

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,'-')
```
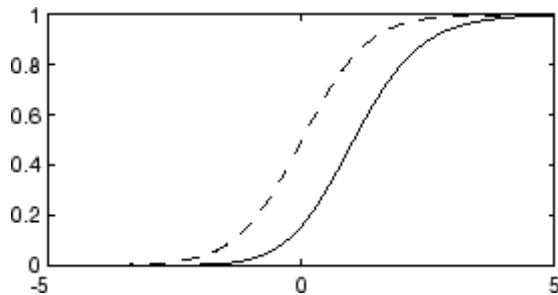
Nonparametric Distributions

See the discussion of `ksdensity` in "Nonparametric Estimation" on page 5-13.

Normal Distribution

In this section...

Definition

The normal pdf is

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

Background

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.

Parameters

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

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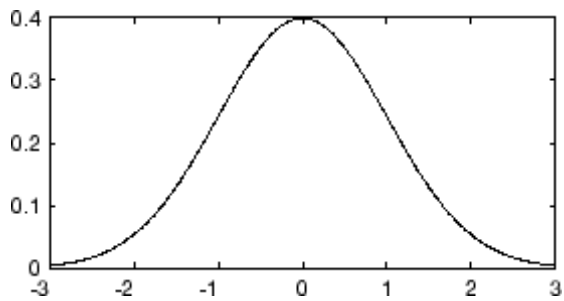
Pareto Distribution

See "Generalized Pareto Distribution" on page B-34.

Pearson System

See "Pearson and Johnson Systems" on page 5-84.

Piecewise Distributions

See the discussion of the @piecewisedistribution class in "Fitting Piecewise Distributions" on page 5-29.

Poisson Distribution

Definition

The Poisson pdf is

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

Background

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.

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.

Parameters

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. This is used to calculate confidence intervals λ. As λ gets large the Poisson distribution can be approximated by a normal distribution with $\mu = \lambda$ and $\sigma^2 = \lambda$. This approximation is used to calculate confidence intervals for values of λ greater than 100.

Example

The plot shows the probability for each nonnegative integer when $\lambda = 5$.

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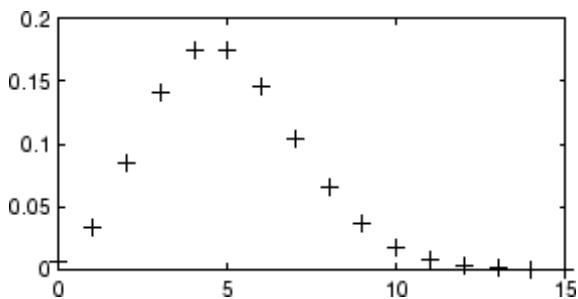
Rayleigh Distribution

Definition

The Rayleigh pdf is

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

Background

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.

Parameters

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

The following commands generate a plot of the Rayleigh pdf.

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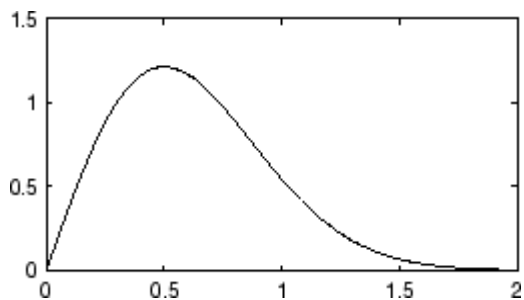
Rician Distribution

In this section...
"Definition" on page B-85
"Background" on page B-85
"Parameters" on page B-85

Definition

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

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.

Parameters

See `mle`, `dfittool`.

Student's t Distribution

In this section...
"Definition" on page B-86
"Background" on page B-86
"Example" on page B-87

Definition

Student's t pdf is

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

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

Background

The t distribution is a family of curves depending on a single parameter v (the degrees of freedom). As v 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

The plot compares the *t* distribution with $v = 5$ (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,'-.')
```
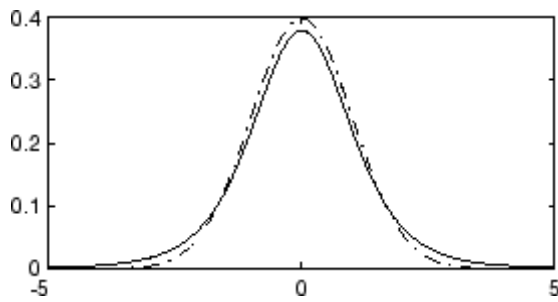
t Location-Scale Distribution

Definition

The *t* location-scale distribution has the density function

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

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

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.

Parameters

See mle, dfittool.

Uniform Distribution (Continuous)

Definition

The uniform cdf is

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

Background

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.

Parameters

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

Example

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)
```
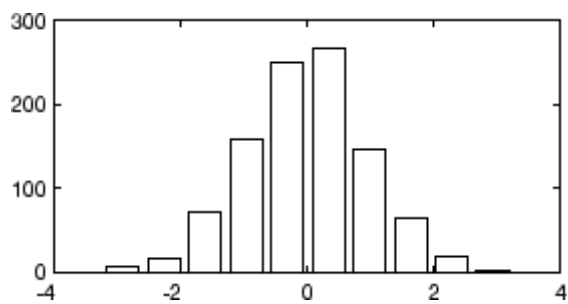
Uniform Distribution (Discrete)

In this section...

Definition

The discrete uniform pdf is

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

Background

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

Example

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])
```
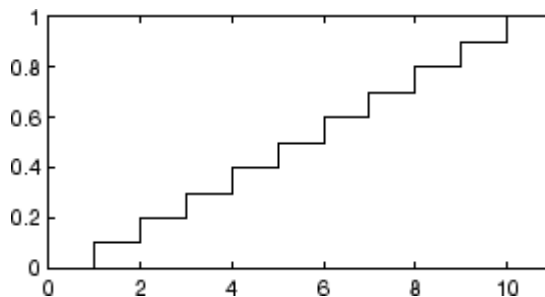
Pick a random sample of 10 from a list of 553 items:

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

Weibull Distribution

Definition

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

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.

Parameters

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

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,'-')
```
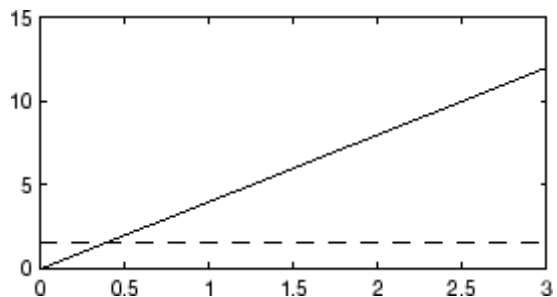
Wishart Distribution

In this section...

Definition

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

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

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. Only random matrix generation is supported for the Wishart distribution, including the singular cases.

Background

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.

v is analogous to the degrees of freedom parameter of a univariate chi-square distribution, and Σv 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

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
```

Bibliography

[1] Atkinson, A. C., and A. N. Donev. *Optimum Experimental Designs*. New York: Oxford University Press, 1992.

[2] Bates, D. M., and D. G. Watts. *Nonlinear Regression Analysis and Its Applications*. Hoboken, NJ: John Wiley & Sons, Inc., 1988.

[3] Belsley, D. A., E. Kuh, and R. E. Welsch. *Regression Diagnostics*. Hoboken, NJ: John Wiley & Sons, Inc., 1980.

[4] Berry, M. W., et al. "Algorithms and Applications for Approximate Nonnegative Matrix Factorization." *Computational Statistics and Data Analysis*. Vol. 52, No. 1, 2007, pp. 155–173.

[5] Bookstein, Fred L. *Morphometric Tools for Landmark Data*. Cambridge, UK: Cambridge University Press, 1991.

[6] Bouye, E., V. Durrleman, A. Nikeghbali, G. Riboulet, and T. Roncalli. "Copulas for Finance: A Reading Guide and Some Applications." Working Paper. Groupe de Recherche Operationnelle, Credit Lyonnais, 2000.

[7] Bowman, A. W., and A. Azzalini. *Applied Smoothing Techniques for Data Analysis*. New York: Oxford University Press, 1997.

[8] Box, G. E. P., and N. R. Draper. *Empirical Model-Building and Response Surfaces*. Hoboken, NJ: John Wiley & Sons, Inc., 1987.

[9] Box, G. E. P., W. G. Hunter, and J. S. Hunter. *Statistics for Experimenters*. Hoboken, NJ: Wiley-Interscience, 1978.

[10] Bratley, P., and B. L. Fox. "ALGORITHM 659 Implementing Sobol's Quasirandom Sequence Generator." *ACM Transactions on Mathematical Software*. Vol. 14, No. 1, 1988, pp. 88–100.

[11] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

[12] Bulmer, M. G. *Principles of Statistics*. Mineola, NY: Dover Publications, Inc., 1979.

[13] Bury, K.. *Statistical Distributions in Engineering*. Cambridge, UK: Cambridge University Press, 1999.

[14] Chatterjee, S., and A. S. Hadi. "Influential Observations, High Leverage Points, and Outliers in Linear Regression." *Statistical Science*. Vol. 1, 1986, pp. 379–416.

[15] Collett, D. *Modeling Binary Data*. New York: Chapman & Hall, 2002.

[16] Conover, W. J. *Practical Nonparametric Statistics*. Hoboken, NJ: John Wiley & Sons, Inc., 1980.

[17] Cook, R. D., and S. Weisberg. *Residuals and Influence in Regression*. New York: Chapman & Hall/CRC Press, 1983.

[18] Cox, D. R., and D. Oakes. *Analysis of Survival Data*. London: Chapman & Hall, 1984.

[19] Davidian, M., and D. M. Giltinan. *Nonlinear Models for Repeated Measurements Data*. New York: Chapman & Hall, 1995.

[20] Deb, P., and M. Sefton. "The Distribution of a Lagrange Multiplier Test of Normality." *Economics Letters*. Vol. 51, 1996, pp. 123–130.

[21] de Jong, S. "SIMPLS: An Alternative Approach to Partial Least Squares Regression." *Chemometrics and Intelligent Laboratory Systems*. Vol. 18, 1993, pp. 251–263.

[22] Demidenko, E. *Mixed Models: Theory and Applications*. Hoboken, NJ: John Wiley & Sons, Inc., 2004.

[23] 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.

[24] Devroye, L. *Non-Uniform Random Variate Generation*. New York: Springer-Verlag, 1986.

[25] Dobson, A. J. *An Introduction to Generalized Linear Models*. New York: Chapman & Hall, 1990.

[26] Draper, N. R., and H. Smith. *Applied Regression Analysis*. Hoboken, NJ: Wiley-Interscience, 1998.

[27] Drezner, Z. "Computation of the Trivariate Normal Integral." *Mathematics of Computation*. Vol. 63, 1994, pp. 289–294.

[28] Drezner, Z., and G. O. Wesolowsky. "On the Computation of the Bivariate Normal Integral." *Journal of Statistical Computation and Simulation*. Vol. 35, 1989, pp. 101–107.

[29] DuMouchel, W. H., and 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.

[30] Durbin, R., S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis*. Cambridge, UK: Cambridge University Press, 1998.

[31] Efron, B., and R. J. Tibshirani. *An Introduction to the Bootstrap*. New York: Chapman & Hall, 1993.

[32] Embrechts, P., C. Klüppelberg, and T. Mikosch. *Modelling Extremal Events for Insurance and Finance*. New York: Springer, 1997.

[33] Evans, M., N. Hastings, and B. Peacock. *Statistical Distributions*. 2nd ed., Hoboken, NJ: John Wiley & Sons, Inc., 1993, pp. 50–52, 73–74, 102–105, 147, 148.

[34] Genz, A. "Numerical Computation of Rectangular Bivariate and Trivariate Normal and t Probabilities." *Statistics and Computing*. Vol. 14, No. 3, 2004, pp. 251–260.

[35] Genz, A., and F. Bretz. "Comparison of Methods for the Computation of Multivariate t Probabilities." *Journal of Computational and Graphical Statistics*. Vol. 11, No. 4, 2002, pp. 950–971.

[36] Genz, A., and F. Bretz. "Numerical Computation of Multivariate t Probabilities with Application to Power Calculation of Multiple Contrasts." *Journal of Statistical Computation and Simulation*. Vol. 63, 1999, pp. 361–378.

[37] Gibbons, J. D. *Nonparametric Statistical Inference*. New York: Marcel Dekker, 1985.

[38] Goodall, C. R. "Computation Using the QR Decomposition." *Handbook in Statistics*. Vol. 9, Amsterdam: Elsevier/North-Holland, 1993.

[39] Hahn, Gerald J., and S. S. Shapiro. *Statistical Models in Engineering*. Hoboken, NJ: John Wiley & Sons, Inc., 1994, p. 95.

[40] Hald, A. *Statistical Theory with Engineering Applications*. Hoboken, NJ: John Wiley & Sons, Inc., 1960.

[41] Harman, H. H. *Modern Factor Analysis*. 3rd Ed. Chicago: University of Chicago Press, 1976.

[42] Hastie, T., R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. New York: Springer, 2001.

[43] Hochberg, Y., and A. C. Tamhane. *Multiple Comparison Procedures*. Hoboken, NJ: John Wiley & Sons, 1987.

[44] Hoerl, A. E., and R. W. Kennard. "Ridge Regression: Applications to Nonorthogonal Problems." *Technometrics*. Vol. 12, No. 1, 1970, pp. 69–82.

[45] Hoerl, A. E., and R. W. Kennard. "Ridge Regression: Biased Estimation for Nonorthogonal Problems." *Technometrics*. Vol. 12, No. 1, 1970, pp. 55–67.

[46] Hogg, R. V., and J. Ledolter. *Engineering Statistics*. New York: MacMillan, 1987.

[47] Holland, P. W., and R. E. Welsch. "Robust Regression Using Iteratively Reweighted Least-Squares." *Communications in Statistics: Theory and Methods*, *A6*, 1977, pp. 813–827.

[48] Hollander, M., and D. A. Wolfe. *Nonparametric Statistical Methods*. Hoboken, NJ: John Wiley & Sons, Inc., 1999.

[49] Hong, H. S., and F. J. Hickernell. "ALGORITHM 823: Implementing Scrambled Digital Sequences." *ACM Transactions on Mathematical Software*. Vol. 29, No. 2, 2003, pp. 95–109.

[50] Huber, P. J. *Robust Statistics*. Hoboken, NJ: John Wiley & Sons, Inc., 1981.

[51] Jackson, J. E. *A User's Guide to Principal Components*. Hoboken, NJ: John Wiley and Sons, 1991.

[52] Jain, A., and R. Dubes. *Algorithms for Clustering Data*. Upper Saddle River, NJ: Prentice-Hall, 1988.

[53] 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. 163–172.

[54] Joe, S., and F. Y. Kuo. "Remark on Algorithm 659: Implementing Sobol's Quasirandom Sequence Generator." *ACM Transactions on Mathematical Software*. Vol. 29, No. 1, 2003, pp. 49–57.

[55] Johnson, N., and S. Kotz. *Distributions in Statistics: Continuous Univariate Distributions-2*. Hoboken, NJ: John Wiley & Sons, Inc., 1970, pp. 130–148, 189–200, 201–219.

[56] Johnson, N. L., N. Balakrishnan, and S. Kotz. *Continuous Multivariate Distributions*. Vol. 1. Hoboken, NJ: Wiley-Interscience, 2000.

[57] Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*. Vol. 1, Hoboken, NJ: Wiley-Interscience, 1993.

[58] Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*. Vol. 2, Hoboken, NJ: Wiley-Interscience, 1994.

[59] Johnson, N. L., S. Kotz, and N. Balakrishnan. *Discrete Multivariate Distributions*. Hoboken, NJ: Wiley-Interscience, 1997.

[60] Johnson, N. L., S. Kotz, and A. W. Kemp. *Univariate Discrete Distributions*. Hoboken, NJ: Wiley-Interscience, 1993.

[61] Jolliffe, I. T. *Principal Component Analysis*. 2nd ed., New York: Springer-Verlag, 2002.

[62] Jöreskog, K. G. "Some Contributions to Maximum Likelihood Factor Analysis." *Psychometrika*. Vol. 32, 1967, pp. 443–482.

[63] Kaufman L., and P. J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. Hoboken, NJ: John Wiley & Sons, Inc., 1990.

[64] Kendall, David G. "A Survey of the Statistical Theory of Shape." *Statistical Science*. Vol. 4, No. 2, 1989, pp. 87–99.

[65] Kocis, L., and W. J. Whiten. "Computational Investigations of Low-Discrepancy Sequences." *ACM Transactions on Mathematical Software*. Vol. 23, No. 2, 1997, pp. 266–294.

[66] Kotz, S., and S. Nadarajah. *Extreme Value Distributions: Theory and Applications*. London: Imperial College Press, 2000.

[67] Krzanowski, W. J. *Principles of Multivariate Analysis: A User's Perspective*. New York: Oxford University Press, 1988.

[68] Lawless, J. F. *Statistical Models and Methods for Lifetime Data*. Hoboken, NJ: Wiley-Interscience, 2002.

[69] Lawley, D. N., and A. E. Maxwell. *Factor Analysis as a Statistical Method*. 2nd ed. New York: American Elsevier Publishing, 1971.

[70] Lilliefors, H. W. "On the Kolmogorov-Smirnov test for normality with mean and variance unknown." *Journal of the American Statistical Association*. Vol. 62, 1967, pp. 399–402.

[71] 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.

[72] Lindstrom, M. J., and D. M. Bates. "Nonlinear mixed-effects models for repeated measures data." *Biometrics*. Vol. 46, 1990, pp. 673–687.

[73] Little, Roderick J. A., and Donald B. Rubin. *Statistical Analysis with Missing Data*. 2nd ed., Hoboken, NJ: John Wiley & Sons, Inc., 2002.

[74] Mardia, K. V., J. T. Kent, and J. M. Bibby. *Multivariate Analysis*. Burlington, MA: Academic Press, 1980.

[75] Marquardt, D.W. "Generalized Inverses, Ridge Regression, Biased Linear Estimation, and Nonlinear Estimation." *Technometrics*. Vol. 12, No. 3, 1970, pp. 591–612.

[76] Marquardt, D. W., and R.D. Snee. "Ridge Regression in Practice." *The American Statistician*. Vol. 29, No. 1, 1975, pp. 3–20.

[77] Marsaglia, G., and W. W. Tsang. "A Simple Method for Generating Gamma Variables." *ACM Transactions on Mathematical Software*. Vol. 26, 2000, pp. 363–372.

[78] Marsaglia, G., W. Tsang, and J. Wang. "Evaluating Kolmogorov's Distribution." *Journal of Statistical Software*. Vol. 8, Issue 18, 2003.

[79] Martinez, W. L., and A. R. Martinez. *Computational Statistics with MATLAB*. New York: Chapman & Hall/CRC Press, 2002.

[80] Massey, F. J. "The Kolmogorov-Smirnov Test for Goodness of Fit." *Journal of the American Statistical Association*. Vol. 46, No. 253, 1951, pp. 68–78.

[81] Matousek, J. "On the L2-Discrepancy for Anchored Boxes." *Journal of Complexity*. Vol. 14, No. 4, 1998, pp. 527–556.

[82] McLachlan, G., and D. Peel. *Finite Mixture Models*. Hoboken, NJ: John Wiley & Sons, Inc., 2000.

[83] McCullagh, P., and J. A. Nelder. *Generalized Linear Models*. New York: Chapman & Hall, 1990.

[84] McGill, R., J. W. Tukey, and W. A. Larsen. "Variations of Boxplots." *The American Statistician*. Vol. 32, No. 1, 1978, pp. 12–16.

[85] Meeker, W. Q., and L. A. Escobar. *Statistical Methods for Reliability Data*. Hoboken, NJ: John Wiley & Sons, Inc., 1998.

[86] Meng, Xiao-Li, and Donald B. Rubin. "Maximum Likelihood Estimation via the ECM Algorithm." *Biometrika*. Vol. 80, No. 2, 1993, pp. 267–278.

[87] Meyers, R. H., and D.C. Montgomery. *Response Surface Methodology: Process and Product Optimization Using Designed Experiments*. Hoboken, NJ: John Wiley & Sons, Inc., 1995.

[88] Miller, L. H. "Table of Percentage Points of Kolmogorov Statistics." *Journal of the American Statistical Association*. Vol. 51, No. 273, 1956, pp. 111–121.

[89] Milliken, G. A., and D. E. Johnson. *Analysis of Messy Data, Volume 1: Designed Experiments*. Boca Raton, FL: Chapman & Hall/CRC Press, 1992.

[90] Montgomery, D. *Introduction to Statistical Quality Control*. Hoboken, NJ: John Wiley & Sons, 1991, pp. 369–374.

[91] Montgomery, D. C. *Design and Analysis of Experiments*. Hoboken, NJ: John Wiley & Sons, Inc., 2001.

[92] Mood, A. M., F. A. Graybill, and D. C. Boes. *Introduction to the Theory of Statistics*. 3rd ed., New York: McGraw-Hill, 1974. pp. 540–541.

[93] Moore, J. *Total Biochemical Oxygen Demand of Dairy Manures*. Ph.D. thesis. University of Minnesota, Department of Agricultural Engineering, 1975.

[94] Mosteller, F., and J. Tukey. *Data Analysis and Regression*. Upper Saddle River, NJ: Addison-Wesley, 1977.

[95] Nelson, L. S. "Evaluating Overlapping Confidence Intervals." *Journal of Quality Technology*. Vol. 21, 1989, pp. 140–141.

[96] Patel, J. K., C. H. Kapadia, and D. B. Owen. *Handbook of Statistical Distributions*. New York: Marcel Dekker, 1976.

[97] Pinheiro, J. C., and D. M. Bates. "Approximations to the log-likelihood function in the nonlinear mixed-effects model." *Journal of Computational and Graphical Statistics*. Vol. 4, 1995, pp. 12–35.

[98] Rice, J. A. *Mathematical Statistics and Data Analysis*. Pacific Grove, CA: Duxbury Press, 1994.

[99] Rosipal, R., and N. Kramer. "Overview and Recent Advances in Partial Least Squares." *Subspace, Latent Structure and Feature Selection: Statistical and Optimization Perspectives Workshop (SLSFS 2005), Revised Selected Papers (Lecture Notes in Computer Science 3940)*. Berlin, Germany: Springer-Verlag, 2006, pp. 34–51.

[100] Sachs, L. *Applied Statistics: A Handbook of Techniques*. New York: Springer-Verlag, 1984, p. 253.

[101] 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.

[102] Seber, G. A. F. *Linear Regression Analysis*. Hoboken, NJ: Wiley-Interscience, 2003.

[103] Seber, G. A. F. *Multivariate Observations*. Hoboken, NJ: John Wiley & Sons, Inc., 1984.

[104] Seber, G. A. F., and C. J. Wild. *Nonlinear Regression*. Hoboken, NJ: Wiley-Interscience, 2003.

[105] Sexton, Joe, and A. R. Swensen. "ECM Algorithms that Converge at the Rate of EM." *Biometrika*. Vol. 87, No. 3, 2000, pp. 651–662.

[106] Snedecor, G. W., and W. G. Cochran. *Statistical Methods*. Ames, IA: Iowa State Press, 1989.

[107] Spath, H. *Cluster Dissection and Analysis: Theory, FORTRAN Programs, Examples*. Translated by J. Goldschmidt. New York: Halsted Press, 1985.

[108] Stein, M. "Large sample properties of simulations using latin hypercube sampling." *Technometrics*. Vol. 29, No. 2, 1987, pp. 143–151. Correction, Vol. 32, p. 367.

[109] Stephens, M. A. "Use of the Kolmogorov-Smirnov, Cramer-Von Mises and Related Statistics Without Extensive Tables." *Journal of the Royal Statistical Society*. Series B, Vol. 32, No. 1, 1970, pp. 115–122.

[110] Street, J. O., R. J. Carroll, and D. Ruppert. "A Note on Computing Robust Regression Estimates via Iteratively Reweighted Least Squares." *The American Statistician*. Vol. 42, 1988, pp. 152–154.

[111] Student. "On the Probable Error of the Mean." *Biometrika*. Vol. 6, No. 1, 1908, pp. 1–25.

[112] Vellemen, P. F., and D. C. Hoaglin. *Application, Basics, and Computing of Exploratory Data Analysis*. Pacific Grove, CA: Duxbury Press, 1981.

[113] Weibull, W. "A Statistical Theory of the Strength of Materials." *Ingeniors Vetenskaps Akademiens Handlingar*. Stockholm: Royal Swedish Institute for Engineering Research, No. 151, 1939.

[114] Zahn, C. T. "Graph-theoretical methods for detecting and describing Gestalt clusters." *IEEE Transactions on Computers*. Vol. C-20, Issue 1, 1971, pp. 68–86.

Index