

# Statistics Toolbox

For Use with MATLAB®

- Computation
- Visualization
- Programming

User's Guide  
*Version 5*



## How to Contact The MathWorks:



www.mathworks.com      Web  
comp.soft-sys.matlab      Newsgroup



support@mathworks.com      Technical support  
suggest@mathworks.com      Product enhancement suggestions  
bugs@mathworks.com      Bug reports  
doc@mathworks.com      Documentation error reports  
service@mathworks.com      Order status, license renewals, passcodes  
info@mathworks.com      Sales, pricing, and general information



508-647-7000      Phone



508-647-7001      Fax



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

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

### *Statistics Toolbox User's Guide*

© COPYRIGHT 1993 - 2004 by The MathWorks, Inc.

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

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

MATLAB, Simulink, Stateflow, Handle Graphics, and Real-Time Workshop are registered trademarks, and TargetBox is a trademark of The MathWorks, Inc.

Other product or brand names are trademarks or registered trademarks of their respective holders.

## History:

September 1993	First printing	Version 1.0
March 1996	Second printing	Version 2.0
January 1997	Third printing	Version 2.11
November 2000	Fourth printing	Revised for Version 3.0 (Release 12)
May 2001	Fifth printing	Minor revisions
July 2002	Sixth printing	Revised for Version 4.0 (Release 13)
February 2003	Online only	Revised for Version 4.1 (Release 13.0.1)
June 2004	Seventh printing	Revised for Version 5.0 (Release 14)
October 2004	Online only	Revised for Version 5.0.1 (Release 14SP1)



## Introduction

1

<b>What Is the Statistics Toolbox?</b> .....	1-2
<b>Primary Topic Areas</b> .....	1-3
<b>Random Number Generators in the Statistics Toolbox</b> ...	1-5
<b>Mathematical Notation</b> .....	1-6

## Probability Distributions

2

<b>Introduction</b> .....	2-2
<b>Displaying Probability Distributions and Random Samples</b> .....	2-3
Probability Distributions Demo .....	2-3
Random Sample Generation Demo .....	2-4
<b>Overview of the Functions</b> .....	2-6
Probability Density Function (pdf) .....	2-6
Cumulative Distribution Function (cdf) .....	2-7
Inverse Cumulative Distribution Function .....	2-7
Random Number Generator .....	2-9
Mean and Variance as a Function of Parameters .....	2-11
<b>Distribution Fitting Tool</b> .....	2-13
Main Window of the Distribution Fitting Tool .....	2-13
Example: Fitting a Distribution .....	2-16
Creating and Managing Data Sets .....	2-20
Creating a New Fit .....	2-24
Displaying Results .....	2-28

Managing Fits .....	2-30
Evaluating Fits .....	2-31
Excluding Data .....	2-34
Saving and Loading Sessions .....	2-39
Generating an M-File to Fit and Plot Distributions .....	2-39
Using Custom Distributions .....	2-41
Additional Distributions Available in the Distribution Fitting Tool .....	2-41
<b>Overview of the Distributions .....</b>	<b>2-45</b>
Reproducing the Output of Random Number Functions .....	2-46
Beta Distribution .....	2-48
Binomial Distribution .....	2-50
Chi-Square Distribution .....	2-52
Noncentral Chi-Square Distribution .....	2-53
Discrete Uniform Distribution .....	2-55
Exponential Distribution .....	2-56
Extreme Value Distribution .....	2-58
F Distribution .....	2-61
Noncentral F Distribution .....	2-63
Gamma Distribution .....	2-64
Geometric Distribution .....	2-66
Hypergeometric Distribution .....	2-67
Lognormal Distribution .....	2-69
Negative Binomial Distribution .....	2-70
Normal Distribution .....	2-73
Poisson Distribution .....	2-76
Rayleigh Distribution .....	2-78
Student's t Distribution .....	2-79
Noncentral t Distribution .....	2-80
Uniform (Continuous) Distribution .....	2-82
Weibull Distribution .....	2-83

## Descriptive Statistics

### 3

<b>Measures of Central Tendency (Location) .....</b>	<b>3-2</b>
--	------------

<b>Measures of Dispersion</b> .....	<b>3-4</b>
<b>Functions for Data with Missing Values (NaNs)</b> .....	<b>3-6</b>
<b>Function for Grouped Data</b> .....	<b>3-8</b>
<b>Percentiles and Graphical Descriptions</b> .....	<b>3-10</b>
Percentiles .....	<b>3-10</b>
Probability Density Estimation .....	<b>3-12</b>
Empirical Cumulative Distribution Function .....	<b>3-15</b>
<b>The Bootstrap</b> .....	<b>3-17</b>

## Linear Models

# 4

<b>Introduction</b> .....	<b>4-2</b>
<b>One-Way Analysis of Variance (ANOVA)</b> .....	<b>4-3</b>
Example: One-Way ANOVA .....	<b>4-3</b>
Multiple Comparisons .....	<b>4-5</b>
Example: Multiple Comparisons .....	<b>4-6</b>
<b>Two-Way Analysis of Variance (ANOVA)</b> .....	<b>4-8</b>
Example: Two-Way ANOVA .....	<b>4-9</b>
<b>N-Way Analysis of Variance</b> .....	<b>4-11</b>
Example: N-Way ANOVA with Small Data Set .....	<b>4-11</b>
Example: N-Way ANOVA with Large Data Set .....	<b>4-13</b>
<b>ANOVA with Random Effects</b> .....	<b>4-18</b>
Setting Up the Model .....	<b>4-18</b>
Fitting a Random Effects Model .....	<b>4-19</b>
F Statistics for Models with Random Effects .....	<b>4-20</b>
Variance Components .....	<b>4-22</b>

<b>Analysis of Covariance</b> .....	<b>4-25</b>
The aocool Demo .....	<b>4-25</b>
<b>Multiple Linear Regression</b> .....	<b>4-33</b>
Mathematical Foundations of Multiple Linear Regression ...	<b>4-33</b>
Example: Multiple Linear Regression .....	<b>4-35</b>
Polynomial Curve Fitting Demo .....	<b>4-36</b>
<b>Quadratic Response Surface Models</b> .....	<b>4-42</b>
Exploring Graphs of Multidimensional Polynomials .....	<b>4-42</b>
<b>Stepwise Regression</b> .....	<b>4-45</b>
Stepwise Regression Demo .....	<b>4-45</b>
<b>Generalized Linear Models</b> .....	<b>4-50</b>
Example: Generalized Linear Models .....	<b>4-50</b>
Generalized Linear Model Demo .....	<b>4-54</b>
<b>Robust and Nonparametric Methods</b> .....	<b>4-55</b>
Robust Regression .....	<b>4-55</b>
Kruskal-Wallis Test .....	<b>4-59</b>
Friedman's Test .....	<b>4-60</b>

## Nonlinear Regression Models

# 5

<b>Nonlinear Least Squares</b> .....	<b>5-2</b>
Example: Nonlinear Modeling .....	<b>5-2</b>
An Interactive GUI for Nonlinear Fitting and Prediction .....	<b>5-6</b>
<b>Regression and Classification Trees</b> .....	<b>5-8</b>



## Hypothesis Tests

# 6

<b>Introduction</b> .....	<b>6-2</b>
<b>Hypothesis Test Terminology</b> .....	<b>6-3</b>
<b>Hypothesis Test Assumptions</b> .....	<b>6-4</b>
<b>Example: Hypothesis Testing</b> .....	<b>6-5</b>
<b>Available Hypothesis Tests</b> .....	<b>6-9</b>

## Multivariate Statistics

# 7

<b>Principal Components Analysis</b> .....	<b>7-2</b>
Example: Principal Components Analysis .....	<b>7-3</b>
The Principal Component Coefficients (First Output) .....	<b>7-6</b>
The Component Scores (Second Output) .....	<b>7-6</b>
The Component Variances (Third Output) .....	<b>7-10</b>
Hotelling's $T^2$ (Fourth Output) .....	<b>7-11</b>
Visualizing the results of a Principal Components Analysis — the Biplot .....	<b>7-12</b>
<b>Factor Analysis</b> .....	<b>7-15</b>
Example: Finding Common Factors Affecting Stock Prices ..	<b>7-16</b>
Factor Rotation .....	<b>7-18</b>
Predicting Factor Scores .....	<b>7-19</b>
Visualizing the Results of a Factor Analysis — the Biplot ...	<b>7-21</b>
Comparison of Factor Analysis and Principal Components Analysis .....	<b>7-23</b>
<b>Multivariate Analysis of Variance (MANOVA)</b> .....	<b>7-24</b>
Example: Multivariate Analysis of Variance .....	<b>7-24</b>
<b>Cluster Analysis</b> .....	<b>7-30</b>

Hierarchical Clustering .....	7-30
K-Means Clustering .....	7-46
<b>Multidimensional Scaling .....</b>	<b>7-53</b>
Overview .....	7-53
Classical Multidimensional Scaling .....	7-53
Nonclassical Metric Multidimensional Scaling .....	7-56
Nonmetric Multidimensional Scaling .....	7-58
Example — Reconstructing a Map from Intercity Distances .	7-60

## Statistical Plots

# 8

<b>Introduction .....</b>	<b>8-2</b>
<b>Box Plots .....</b>	<b>8-3</b>
<b>Distribution Plots .....</b>	<b>8-4</b>
Normal Probability Plots .....	8-4
Quantile-Quantile Plots .....	8-6
Weibull Probability Plots .....	8-7
Empirical Cumulative Distribution Function (CDF) .....	8-8
<b>Scatter Plots .....</b>	<b>8-10</b>

## Statistical Process Control

# 9

<b>Control Charts .....</b>	<b>9-2</b>
Xbar Charts .....	9-2
S Charts .....	9-3
EWMA Charts .....	9-4
<b>Capability Studies .....</b>	<b>9-5</b>

<b>Introduction</b> .....	10-2
<b>Full Factorial Designs</b> .....	10-4
<b>Fractional Factorial Designs</b> .....	10-6
<b>Response Surface Designs</b> .....	10-8
Central Composite Designs .....	10-8
Box-Behnken Designs .....	10-9
Design of Experiments Demo .....	10-10
<b>D-Optimal Designs</b> .....	10-18
Generating D-Optimal Designs .....	10-18
Augmenting D-Optimal Designs .....	10-21
Designing Experiments with Uncontrolled Inputs .....	10-23
Controlling Candidate Points .....	10-24
Including Categorical Factors .....	10-24

<b>Introduction</b> .....	11-2
<b>Example of a Hidden Markov Model</b> .....	11-4
<b>Markov Chains</b> .....	11-6
How the Toolbox Generates Random Sequences .....	11-7
<b>Analyzing a Hidden Markov Model</b> .....	11-8
Setting Up the Model and Generating Data .....	11-8
Computing the Most Likely Sequence of States .....	11-9
Estimating the Transition and Emission Matrices .....	11-9
Calculating Posterior State Probabilities .....	11-12
Changing the Probabilities of the Initial States .....	11-13

Example: Changing the Initial Probabilities . . . . .	11-14
References . . . . .	11-16

## Function Reference

# 12

<b>Functions — Categorical List . . . . .</b>	<b>12-2</b>
Probability Distributions . . . . .	12-2
Descriptive Statistics . . . . .	12-9
Statistical Plotting . . . . .	12-10
Statistical Process Control . . . . .	12-11
Linear Models . . . . .	12-12
Nonlinear Regression . . . . .	12-13
Design of Experiments . . . . .	12-13
Multivariate Statistics . . . . .	12-14
Decision Tree Techniques . . . . .	12-15
Hypothesis Tests . . . . .	12-16
Distribution Testing . . . . .	12-16
Nonparametric Testing . . . . .	12-16
Hidden Markov Models . . . . .	12-16
File I/O . . . . .	12-17
Demonstrations . . . . .	12-17
Data . . . . .	12-18
Utility . . . . .	12-18
 <b>Functions — Alphabetical List . . . . .</b>	 <b>12-19</b>

## Selected Bibliography

# A

## Index

# Introduction

---

What Is the Statistics Toolbox? (p. 1-2)	Lists statistical tasks supported by the toolbox, and explains the role of functions and graphical tools.
Primary Topic Areas (p. 1-3)	Lists the statistical topics addressed in the toolbox and covered in this book.
Random Number Generators in the Statistics Toolbox (p. 1-5)	Tells you how to duplicate the results shown in examples that generate data using random numbers.
Mathematical Notation (p. 1-6)	Describes mathematical notation used in this guide.

## What Is the Statistics Toolbox?

The Statistics Toolbox, for use with MATLAB<sup>®</sup>, is a collection of statistical tools built on the MATLAB numeric computing environment. The toolbox supports a wide range of common statistical tasks, from random number generation, to curve fitting, to design of experiments and statistical process control. The toolbox provides two categories of tools:

- Building-block probability and statistics functions
- Graphical, interactive tools

The first category of tools is made up of functions that you can call from the command line or from your own applications. Many of these functions are MATLAB M-files, series of MATLAB statements that implement specialized statistics algorithms. You can view the MATLAB code for these functions using the statement

```
type function_name
```

You can change the way any toolbox function works by copying and renaming the M-file, then modifying your copy. You can also extend the toolbox by adding your own M-files.

Secondly, the toolbox provides a number of interactive tools that let you access many of the functions through a graphical user interface (GUI). Together, the GUI-based tools provide an environment for polynomial fitting and prediction, as well as probability function exploration.

## Primary Topic Areas

The Statistics Toolbox has more than 200 M-files, supporting work in these topical areas:

### Probability Distributions

The Statistics Toolbox supports 20 probability distributions. For each distribution there are five associated functions. They are

- Probability density function (pdf)
- Cumulative distribution function (cdf)
- Inverse of the cumulative distribution function
- Random number generator
- Mean and variance as a function of the parameters

For most distributions, the Statistics Toolbox also provides functions for computing parameter estimates and confidence intervals.

### Descriptive Statistics

The Statistics Toolbox provides functions for describing the features of a data sample. These descriptive statistics include measures of location and spread, percentile estimates and functions for dealing with data having missing values.

### Linear Models

In the area of linear models, the Statistics Toolbox supports one-way, two-way, and higher-way analysis of variance (ANOVA), analysis of covariance (ANOCOVA), multiple linear regression, stepwise regression, response surface prediction, ridge regression, and one-way multivariate analysis of variance (MANOVA). It supports nonparametric versions of one- and two-way ANOVA. It also supports multiple comparisons of the estimates produced by ANOVA and ANOCOVA functions.

### Nonlinear Models

For nonlinear models, the Statistics Toolbox provides functions for parameter estimation, interactive prediction and visualization of multidimensional nonlinear fits, and confidence intervals for parameters and predicted values. It

provides functions for using classification and regression trees to approximate regression relationships.

### **Hypothesis Tests**

The Statistics Toolbox also provides functions that do the most common tests of hypothesis — t-tests, Z-tests, nonparametric tests, and distribution tests.

### **Multivariate Statistics**

The Statistics Toolbox supports methods in multivariate statistics, including principal components analysis, factor analysis, one-way multivariate analysis of variance, cluster analysis, and classical multidimensional scaling.

### **Statistical Plots**

The Statistics Toolbox adds box plots, normal probability plots, Weibull probability plots, control charts, and quantile-quantile plots to the arsenal of graphs in MATLAB. There is also extended support for polynomial curve fitting and prediction. There are functions to create scatter plots or matrices of scatter plots for grouped data, and to identify points interactively on such plots. There is a function to interactively explore a fitted regression model.

### **Statistical Process Control (SPC)**

For SPC, the Statistics Toolbox provides functions for plotting common control charts and performing process capability studies.

### **Design of Experiments (DOE)**

The Statistics Toolbox supports full and fractional factorial designs, response surface designs, and D-optimal designs. There are functions for generating designs, augmenting designs, and optimally assigning units with fixed covariates.

### **Hidden Markov Models**

The Statistics Toolbox provides functions for analyzing hidden Markov models — models in which you do not know all the state information. These include functions for generating random data, calculating the most probable state sequence for an observed sequence, estimating model parameters, calculating posterior state probabilities, and calculating maximum likelihood estimates for parameters.



## Random Number Generators in the Statistics Toolbox

The Statistics Toolbox contains a number of functions, such as random number generators, that return random output. These functions use the MATLAB functions `rand` and `randn` to generate their output. If you want to make the output for any of these functions reproducible, you must set the states for `rand` and `randn` each time you call the function. See “Reproducing the Output of Random Number Functions” on page 2-46 for more information.

---

**Note** Prior to Version 5, MATLAB employed a different random number generator, which used the syntax `'seed'` instead of `'state'`. Although use of the `'seed'` syntax is backward compatible in MATLAB Version 7, you should use the `'state'` syntax instead.

---

## Mathematical Notation

This manual and the Statistics Toolbox functions use the following mathematical notation conventions.

$\beta$	Parameters in a linear model.
$E(x)$	Expected value of $x$ . $E(x) = \int tf(t)dt$
$f(x   a, b)$	Probability density function. $x$ is the independent variable; $a$ and $b$ are fixed parameters.
$F(x   a, b)$	Cumulative distribution function.
$I([a, b])$ or $I_{[a, b]}$	Indicator function. In this example the function takes the value 1 on the closed interval from $a$ to $b$ and is 0 elsewhere.
$p$ and $q$	$p$ is the probability of some event. $q$ is the probability of $\sim p$ , so $q = 1-p$ .

# Probability Distributions

---

Introduction (p. 2-2)	Introduces the concept of a probability distribution and discusses the difference between continuous and discrete distributions.
Displaying Probability Distributions and Random Samples (p. 2-3)	Describes tools for displaying probability distributions and creating histograms of random samples.
Overview of the Functions (p. 2-6)	Discusses the five functions that the Statistics Toolbox provides for each distribution: probability density function (pdf), cumulative distribution function (cdf), inverse cumulative distribution function, random number generator, and mean and variance as a function of the distribution parameters.
Distribution Fitting Tool (p. 2-13)	Describes the Distribution Fitting Tool, a GUI for fitting distributions to data.
Overview of the Distributions (p. 2-45)	Lists and discusses the probability distributions that the Statistics Toolbox supports.

## Introduction

Probability distributions arise from experiments where the outcome is subject to chance. The nature of the experiment dictates which probability distributions may be appropriate for modeling the resulting random outcomes. There are two types of probability distributions – *continuous* and *discrete*.

<b>Continuous (data)</b>	<b>Continuous (statistics)</b>	<b>Discrete</b>
Beta	Chi-square	Binomial
Exponential	Noncentral Chi-square	Discrete Uniform
Extreme Value		
Gamma	F	Geometric
Lognormal	Noncentral F	Hypergeometric
Normal	t	Negative Binomial
Rayleigh	Noncentral t	Poisson
Uniform		
Weibull		

Suppose you are studying a machine that produces videotape. One measure of the quality of the tape is the number of visual defects per hundred feet of tape. The result of this experiment is an integer, since you cannot observe 1.5 defects. To model this experiment you should use a discrete probability distribution.

A measure affecting the cost and quality of videotape is its thickness. Thick tape is more expensive to produce, while variation in the thickness of the tape on the reel increases the likelihood of breakage. Suppose you measure the thickness of the tape every 1000 feet. The resulting numbers can take a continuum of possible values, which suggests using a continuous probability distribution to model the results.

Using a probability model does not allow you to predict the result of any individual experiment but you can determine the probability that a given outcome will fall inside a specific range of values.

## Displaying Probability Distributions and Random Samples

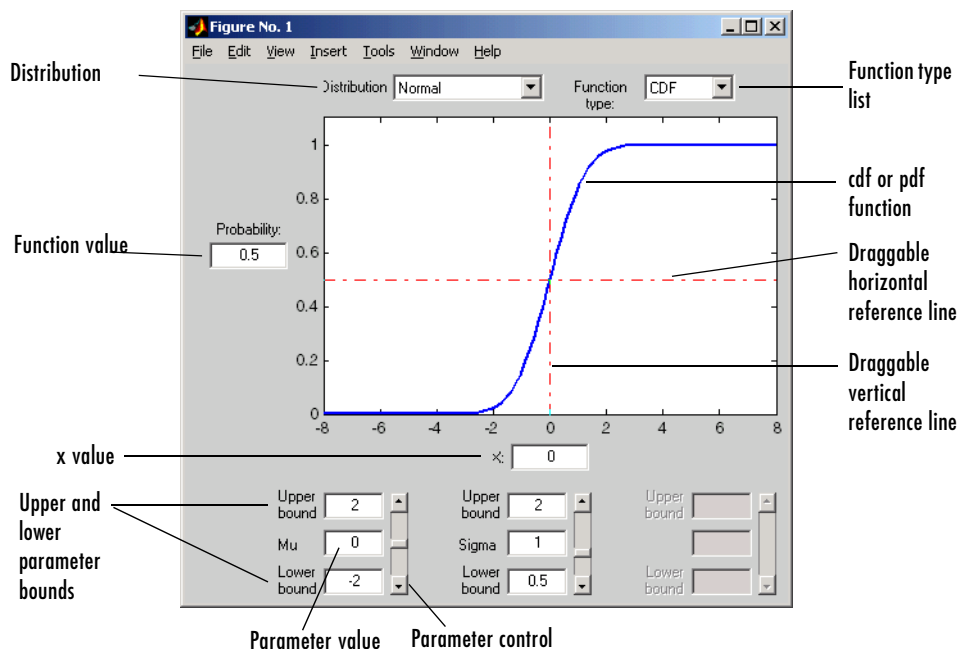
The Statistics Toolbox provides two demos for visualizing probability distributions and graphing random data:

- The probability distribution demo enables you to create interactive plots of probability distributions.
- The random sample generation demo enables you to generate random samples from specified distributions and create histograms of the data.

### Probability Distributions Demo

The probability distributions demo creates interactive plots of probability distributions. It provides a graphic environment for developing an intuitive understanding of probability distributions.

You can run this tool by typing `disttool` at the command line. You can also run it from the Demos tab in the Help browser.



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

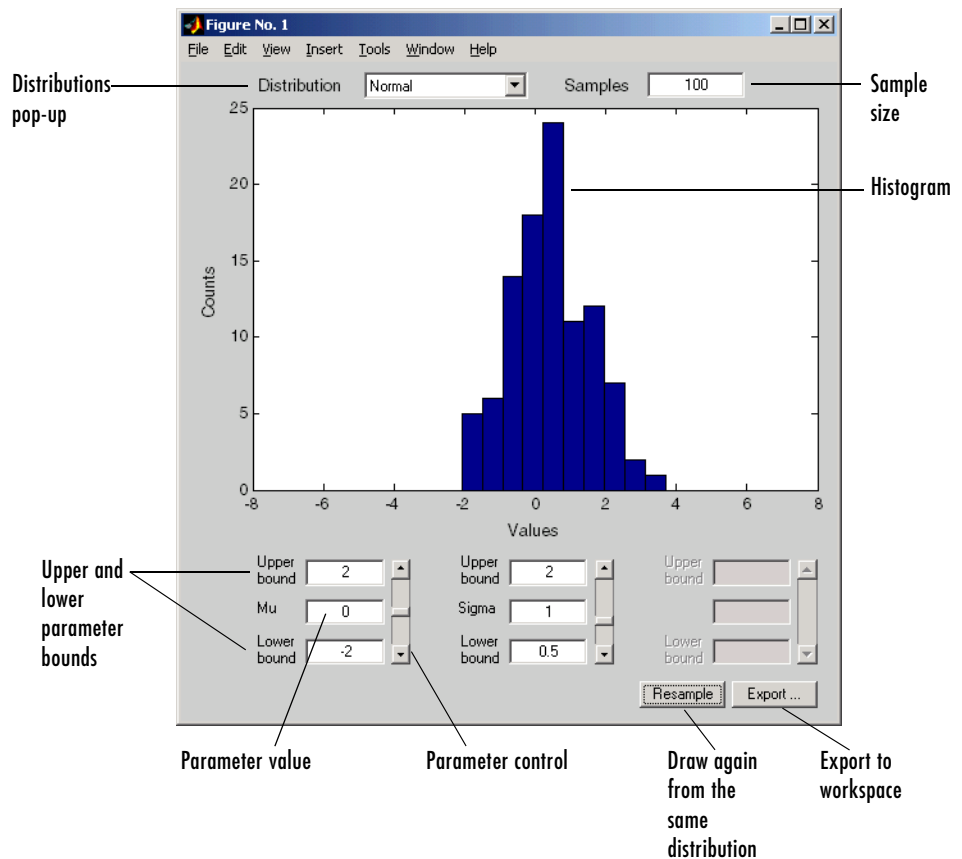
Once the plot displays, you can

- Calculate a new function value by typing a new  $x$  value in the text box on the  $x$ -axis, dragging the vertical reference line, or clicking in the figure where you want the line to be. The new function value displays in the text box to the left of the plot.
- For cdf plots, find critical values corresponding to a specific probability by typing the desired probability in the text box on the  $y$ -axis or by dragging the horizontal reference line.
- Use the controls at the bottom of the window to set parameter values for the distribution and to change their upper and lower bounds.

### **Random Sample Generation Demo**

The random sample generation demo is a graphical environment that generates random samples from specified probability distributions and displays the samples as histograms. You can use `randtool` to explore the effects of changing parameters and sample size on the samples.

You can run this 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.

## Overview of the Functions

The Statistics Toolbox provides five functions for each distribution. They are discussed in the following sections:

- “Probability Density Function (pdf)” on page 2-6
- “Cumulative Distribution Function (cdf)” on page 2-7
- “Inverse Cumulative Distribution Function” on page 2-7
- “Random Number Generator” on page 2-9
- “Mean and Variance as a Function of Parameters” on page 2-11

### Probability Density Function (pdf)

The probability density function (pdf) has a different meaning depending on whether the distribution is discrete or continuous.

For discrete distributions, the pdf is the probability of observing a particular outcome. In the videotape example, the probability that there is exactly one defect in a given hundred feet of tape is the value of the pdf at 1.

Unlike discrete distributions, the pdf of a continuous distribution at a value is not the probability of observing that value. For continuous distributions the probability of observing any particular value is zero. To get probabilities you must integrate the pdf over an interval of interest. For example the probability of the thickness of a videotape being between one and two millimeters is the integral of the appropriate pdf from one to two.

A pdf has two theoretical properties:

- The pdf is zero or positive for every possible outcome.
- The integral of a pdf over its entire range of values is one.

A pdf is not a single function. Rather a pdf is a family of functions characterized by one or more parameters. Once you choose (or estimate) the parameters of a pdf, you have uniquely specified the function.

The pdf function call has the same general format for every distribution in the Statistics Toolbox. The following commands illustrate how to call the pdf for the normal distribution.

```
x = [-3:0.1:3];  
f = normpdf(x,0,1);
```



The variable `f` contains the density of the normal pdf with parameters  $\mu=0$  and  $\sigma=1$  at the values in `x`. The first input argument of every pdf is the set of values for which you want to evaluate the density. Other arguments contain as many parameters as are necessary to define the distribution uniquely. The normal distribution requires two parameters; a location parameter (the mean,  $\mu$ ) and a scale parameter (the standard deviation,  $\sigma$ ).

## Cumulative Distribution Function (cdf)

If  $f$  is a probability density function for random variable  $X$ , the associated cumulative distribution function (cdf)  $F$  is

$$F(x) = P(X \leq x) = \int_{-\infty}^x f(t) dt$$

The cdf of a value  $x$ ,  $F(x)$ , is the probability of observing any outcome less than or equal to  $x$ .

A cdf has two theoretical properties:

- The cdf ranges from 0 to 1.
- If  $y > x$ , then the cdf of  $y$  is greater than or equal to the cdf of  $x$ .

The cdf function call has the same general format for every distribution in the Statistics Toolbox. The following commands illustrate how to call the cdf for the normal distribution.

```
x = [-3:0.1:3];  
p = normcdf(x,0,1);
```

The variable `p` contains the probabilities associated with the normal cdf with parameters  $\mu=0$  and  $\sigma=1$  at the values in `x`. The first input argument of every cdf is the set of values for which you want to evaluate the probability. Other arguments contain as many parameters as are necessary to define the distribution uniquely.

## Inverse Cumulative Distribution Function

The inverse cumulative distribution function returns critical values for hypothesis testing given significance probabilities. To understand the

relationship between a continuous cdf and its inverse function, try the following.

```
x = [-3:0.1:3];  
xnew = norminv(normcdf(x,0,1),0,1);
```

How does `xnew` compare with `x`? Conversely, try this.

```
p = [0.1:0.1:0.9];  
pnew = normcdf(norminv(p,0,1),0,1);
```

How does `pnew` compare with `p`?

Calculating the cdf of values in the domain of a continuous distribution returns probabilities between zero and one. Applying the inverse cdf to these probabilities yields the original values.

For discrete distributions, the relationship between a cdf and its inverse function is more complicated. It is likely that there is no  $x$  value such that the cdf of  $x$  yields  $p$ . In these cases the inverse function returns the first value  $x$  such that the cdf of  $x$  equals or exceeds  $p$ . Try this.

```
x = [0:10];  
y = binoinv(binocdf(x,10,0.5),10,0.5);
```

How does `x` compare with `y`?

The commands below illustrate the problem with reconstructing the probability  $p$  from the value  $x$  for discrete distributions.

```
p = [0.1:0.2:0.9];  
pnew = binocdf(binoinv(p,10,0.5),10,0.5)  
  
pnew =  
  
0.1719    0.3770    0.6230    0.8281    0.9453
```

The inverse function is useful in hypothesis testing and production of confidence intervals. Here is the way to get a 99% confidence interval for a normally distributed sample.

```
p = [0.005 0.995];  
x = norminv(p,0,1)
```

This returns

```
x =  
    -2.5758    2.5758
```

The variable  $x$  contains the values associated with the normal inverse function with parameters  $\mu=0$  and  $\sigma=1$  at the probabilities in  $p$ . The difference  $p(2) - p(1)$  is 0.99. Thus, the values in  $x$  define an interval that contains 99% of the standard normal probability.

The inverse function call has the same general format for every distribution in the Statistics Toolbox. The first input argument of every inverse function is the set of probabilities for which you want to evaluate the critical values. Other arguments contain as many parameters as are necessary to define the distribution uniquely.

## Random Number Generator

The methods for generating random numbers from any distribution all start with uniform random numbers. Once you have a uniform random number generator, you can produce random numbers from other distributions either directly or by using inversion or rejection methods, described below. See “Syntax for Random Number Functions” on page 2-10 for details on using generator functions.

### Direct

Direct methods flow from the definition of the distribution.

As an example, consider generating binomial random numbers. You can think of binomial random numbers as the number of heads in  $n$  tosses of a coin with probability  $p$  of a heads on any toss. If you generate  $n$  uniform random numbers and count the number that are less than  $p$ , the result is binomial with parameters  $n$  and  $p$ .

### Inversion

The inversion method works due to a fundamental theorem that relates the uniform distribution to other continuous distributions.

If  $F$  is a continuous distribution with inverse  $F^{-1}$ , and  $U$  is a uniform random number, then  $F^{-1}(U)$  has distribution  $F$ .

So, you can generate a random number from a distribution by applying the inverse function for that distribution to a uniform random number. Unfortunately, this approach is usually not the most efficient.

### Rejection

The functional form of some distributions makes it difficult or time consuming to generate random numbers using direct or inversion methods. Rejection methods can sometimes provide an elegant solution in these cases.

Suppose you want to generate random numbers from a distribution with pdf  $f$ . To use rejection methods you must first find another density,  $g$ , and a constant,  $c$ , so that the inequality below holds

$$f(x) \leq cg(x)$$

for all  $x$ .

You then generate the random numbers you want using the following steps:

- 1 Generate a random number  $x$  from distribution  $G$  with density  $g$ .
- 2 Form the ratio  $r = \frac{cg(x)}{f(x)}$ .
- 3 Generate a uniform random number  $u$ .
- 4 If the product of  $u$  and  $r$  is less than one, return  $x$ .
- 5 Otherwise repeat steps one to three.

For efficiency you need a cheap method for generating random numbers from  $G$ , and the scalar  $c$  should be small. The expected number of iterations is  $c$ .

### Syntax for Random Number Functions

You can generate random numbers from each distribution. This function provides a single random number or a matrix of random numbers, depending on the arguments you specify in the function call.

For example, here is the way to generate random numbers from the beta distribution. Four statements obtain random numbers: the first returns a

single number, the second returns a 2-by-2 matrix of random numbers, and the third and fourth return 2-by-3 matrices of random numbers.

```
a = 1;
b = 2;
c = [.1 .5; 1 2];
d = [.25 .75; 5 10];
m = [2 3];
nrow = 2;
ncol = 3;

r1 = betarnd(a,b)
r1 =

    0.4469

r2 = betarnd(c,d)
r2 =

    0.8931    0.4832
    0.1316    0.2403

r3 = betarnd(a,b,m)
r3 =

    0.4196    0.6078    0.1392
    0.0410    0.0723    0.0782

r4 = betarnd(a,b,nrow,ncol)
r4 =

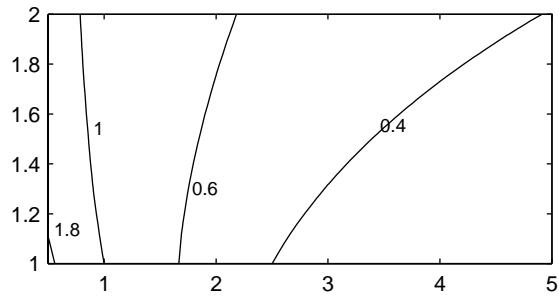
    0.0520    0.3975    0.1284
    0.3891    0.1848    0.5186
```

## Mean and Variance as a Function of Parameters

The mean and variance of a probability distribution are generally simple functions of the parameters of the distribution. The Statistics Toolbox functions ending in "stat" all produce the mean and variance of the desired distribution for the given parameters.

The example below shows a contour plot of the mean of the Weibull distribution as a function of the parameters.

```
x = (0.5:0.1:5);  
y = (1:0.04:2);  
[X,Y] = meshgrid(x,y);  
Z = wblstat(X,Y);  
[c,h] = contour(x,y,Z,[0.4 0.6 1.0 1.8]);  
clabel(c);
```



## Distribution Fitting Tool

The Distribution Fitting Tool is a graphical user interface (GUI) for fitting univariate distributions to data. This section describes how to use the Distribution Fitting Tool and covers the following topics:

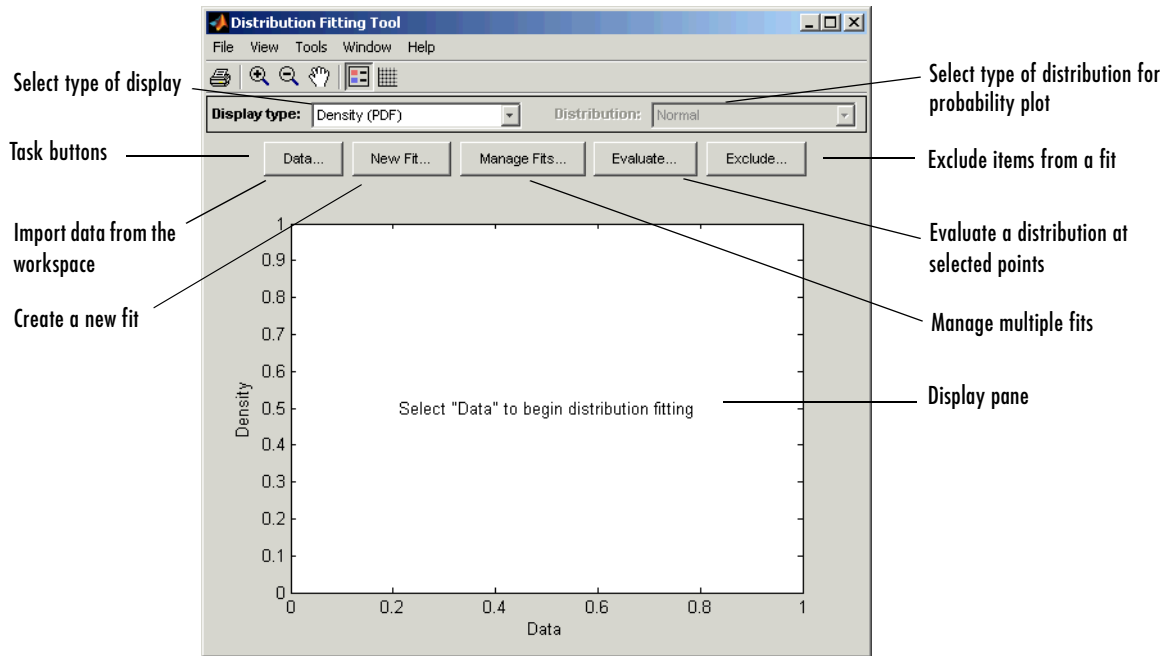
- “Main Window of the Distribution Fitting Tool” on page 2-13
- “Example: Fitting a Distribution” on page 2-16
- “Creating and Managing Data Sets” on page 2-20
- “Creating a New Fit” on page 2-24
- “Displaying Results” on page 2-28
- “Managing Fits” on page 2-30
- “Evaluating Fits” on page 2-31
- “Excluding Data” on page 2-34
- “Saving and Loading Sessions” on page 2-39
- “Generating an M-File to Fit and Plot Distributions” on page 2-39
- “Using Custom Distributions” on page 2-41
- “Additional Distributions Available in the Distribution Fitting Tool” on page 2-41

### 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.



### 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.



“Display Type” on page 2-28 provides more information about the available display types.

### 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 2-20.
- **New Fit** — Create new fits. See “Creating a New Fit” on page 2-24.
- **Manage Fits** — Manage existing fits. See “Managing Fits” on page 2-30.
- **Evaluate** — Evaluate fits at any points you choose. See “Evaluating Fits” on page 2-31.
- **Exclude** — Create rules specifying which values to exclude when fitting a distribution. See “Excluding Data” on page 2-34.

### 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 2-39.
- 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 2-39.
- Define and import custom distributions — see “Using Custom Distributions” on page 2-41.

### 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 2-16
- “Import Data into the Distribution Fitting Tool” on page 2-16
- “Create a New Fit” on page 2-18

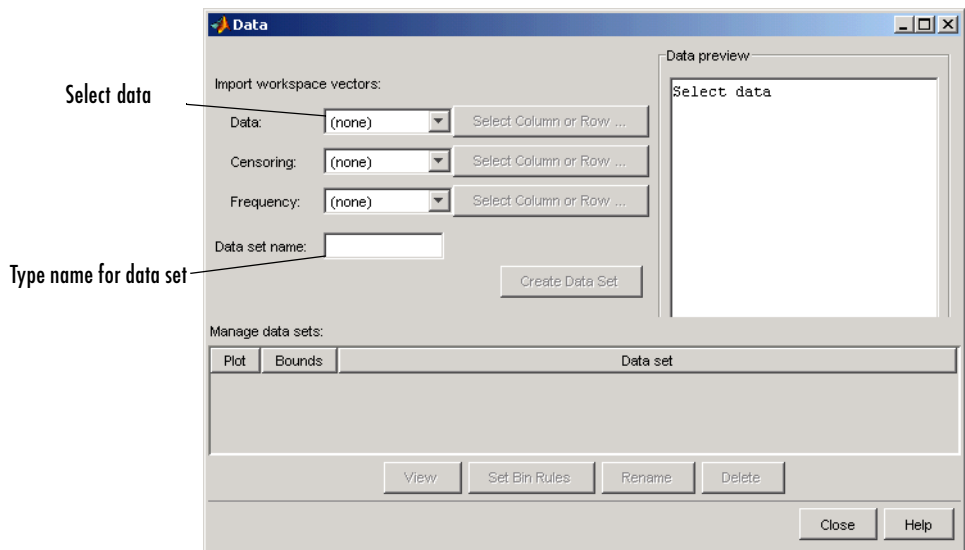
#### Create Random Data for the Example

To try the example, first generate some random data to which you will fit a distribution. The following command generates a vector data, of length 100, whose entries are random numbers from a normal distribution with mean .36 and standard deviation 1.4.

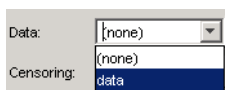
```
data = normrnd(.36, 1.4, 100, 1);
```

#### Import Data into the Distribution Fitting Tool

To import the vector data into the Distribution Fitting Tool, click the **Data** button in main window. This opens the window shown in the following figure.

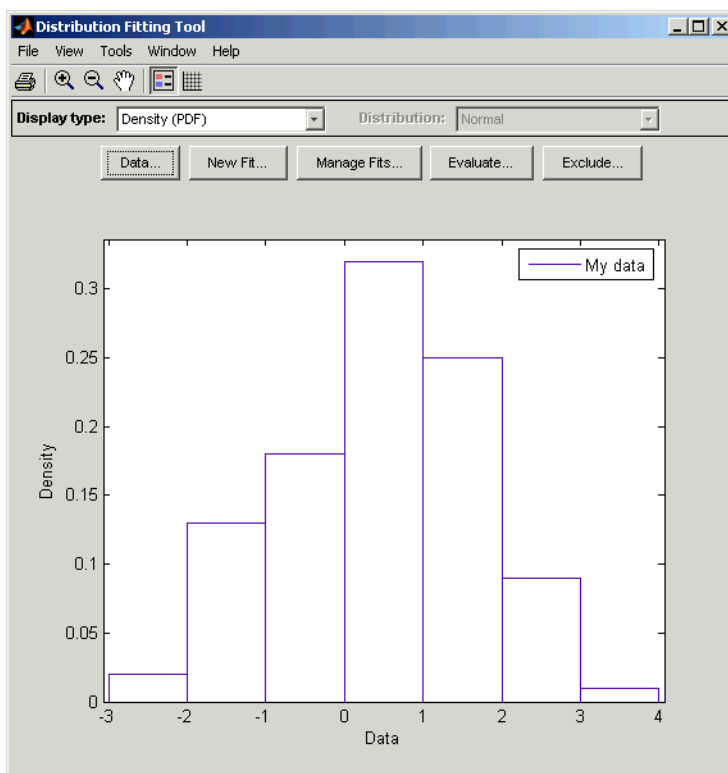


The **Data** field displays all numeric arrays in the MATLAB workspace. Select data from the drop-down list, as shown in the following figure.



This displays a histogram of the data in the **Data preview** pane.

In the **Data set name** field, type a name for the data set, such as My data, and click **Create Data Set** to create the data set. The main window of the Distribution Fitting Tool now displays a larger version of the histogram in the **Data preview** pane, as shown in the following figure.



**Histogram of the Data**

---

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

---

### Create a New Fit

To fit a distribution to the data, click **New Fit** in the main window of the Distribution Fitting Tool. This opens the window shown in the following figure.

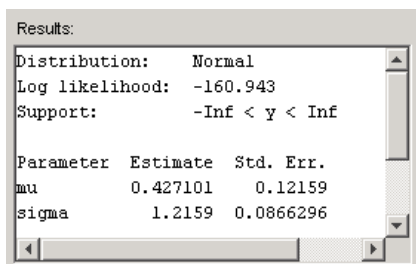


To fit a normal distribution, the default entry of the **Distribution** field, to My data,

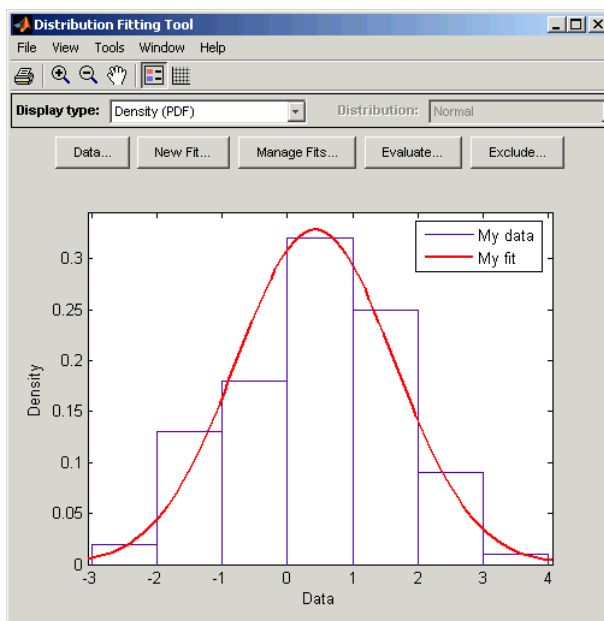
- Enter a name for the fit, such as My fit, in the **Fit name** field.

- Select My data from the drop-down list in the **Data** field.
- 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.



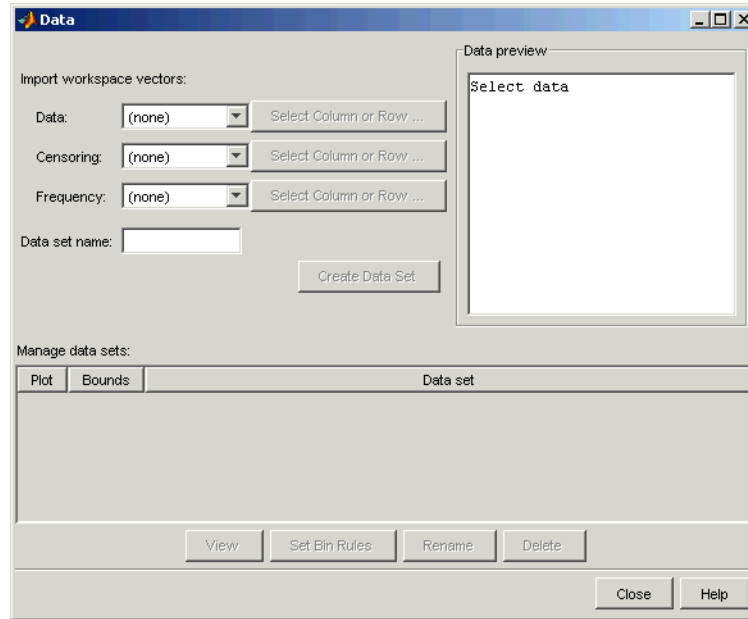
The main window of the Distribution Fitting Tool displays a plot of the normal distribution with this mean and standard deviation, as shown in the following figure.



**Plot of the Distribution and Data**

## Creating and Managing Data Sets

This section describes how create and manage data sets. To begin, click the **Data** button in the main window of the Distribution Fitting Tool to open the **Data** window shown in the following figure.



### Importing Data

The **Import workspace vectors** pane enables you to create a data set by importing a vector from the MATLAB workspace. The following sections describe the fields of the **Import workspace vectors** pane.

**Data.** The drop-down list in the **Data** field contains the names of all matrices and vectors, other than 1-by-1 matrices (scalars) in the MATLAB workspace. Select the array containing the data you want to fit. The actual data you import must be a vector. If you select a matrix in the **Data** field, the first column of the matrix is imported by default. To select a different column or row of the matrix, click **Select Column or Row**. This displays the matrix in the Array Editor, where you can select a row or column by highlighting it with the mouse.

Alternatively, you can enter any valid MATLAB expression in the **Data** field.

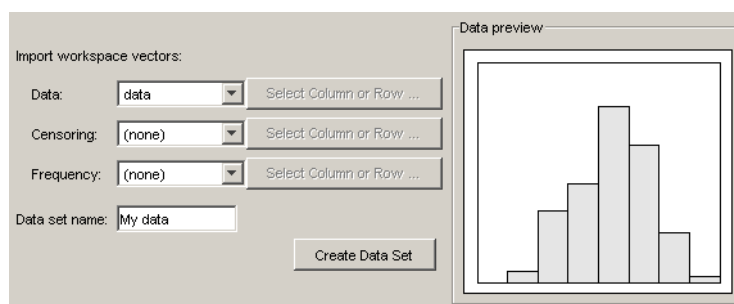
When you select a vector in the **Data** field, a histogram of the data is displayed in the **Data preview** pane.

**Censoring.** If some of the points in the data set are censored, enter a Boolean vector, of the same size as the data vector, specifying the censored entries of the data. A 1 in the censoring vector specifies that the corresponding entry of the data vector is censored, while a 0 specifies that the entry is not censored. If you enter a matrix, you can select a column or row by clicking **Select Column or Row**. If you do not want to censor any data, leave the **Censoring** field blank.

**Frequency.** Enter a vector of positive integers of the same size as the data vector to specify the frequency of the corresponding entries of the data vector. For example, a value of 7 in the 15th entry of frequency vector specifies that there are 7 data points corresponding to the value in the 15th entry of the data vector. If all entries of the data vector have frequency 1, leave the **Frequency** field blank.

**Data name.** Enter a name for the data set you import from the workspace, such as My\_data.

As an example, if you create the vector data described in “Example: Fitting a Distribution” on page 2-16, and select it in the **Data** field, the upper half of the **Data** window appears as in the following figure.

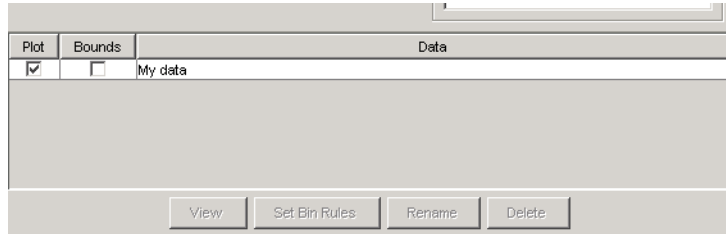


After you have entered the information in the preceding fields, click **Create Data Set** to create the data set My\_data.

## 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. See “Histogram of the Data” on page 2-17 for an example. 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. See “Display Type” on page 2-28.
- 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:

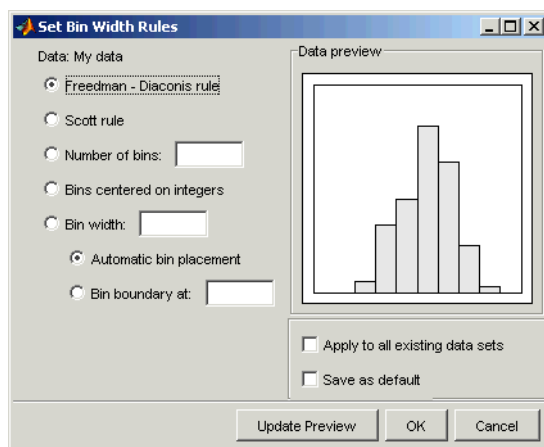
- **View** — Displays the data in a table in a new window.
- **Set Bin Rules** — Defines the histogram bins used in a density (PDF) plot. See “Setting Bin Rules” on page 2-23.
- **Rename** — Renames the data set.



- **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:

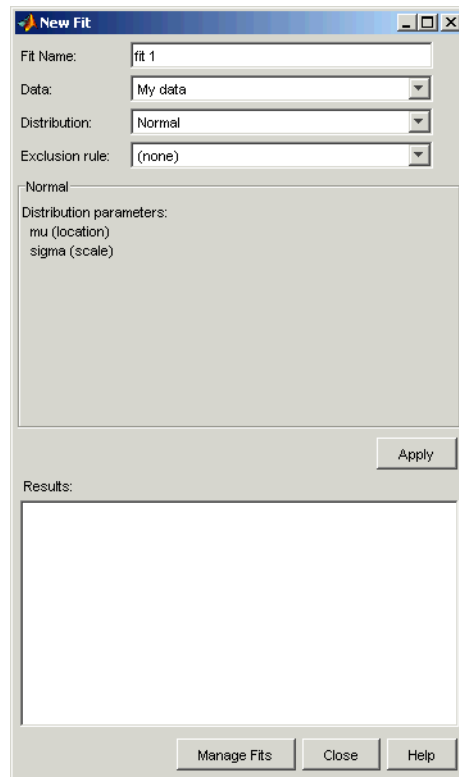
- **Freedom-Diaconis rule** — Algorithm that chooses bin widths and locations automatically, based on the sample size and the spread of the data. This rule, which is the default, is suitable for many kinds of data
- **Scott rule** — Algorithm intended for data that are approximately normal. The algorithm chooses bin widths and locations 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 2-16, 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 in the **Distribution** drop-down list. “Available Distributions” on page 2-26 lists the available distributions.

---

**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 **Exclusion Rule**. 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 2-27.

### 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. “Excluding Data” on page 2-34.

### 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. “Plot of the Distribution and Data” on page 2-19 shows the main window when you fit a normal distribution to My 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 of these distributions are supported by Statistics Toolbox functions that you can use to fit distributions at the command line. For these distributions, the corresponding command-line functions are also listed. Distributions that do not have corresponding command-line functions are described in “Additional Distributions Available in the Distribution Fitting Tool” on page 2-41.

**Non-parametric.** Fits a nonparametric model using kernel smoothing with the function `ksdensity`. “Options for Nonparametric Fits” on page 2-27 describes the available options.

**Beta.** Fits a beta distribution using the function `betafit`. See “Beta Distribution” on page 2-48.

**Birnbaum-Saunders.** Fits a Birnbaum-Saunders distribution. See “Birnbaum-Saunders Distribution” on page 2-42.

**Exponential.** Fits an exponential distribution using the function `expfit`. See “Exponential Distribution” on page 2-56.

**Extreme Value.** Fits an extreme value distribution using the function `evfit`. See “Extreme Value Distribution” on page 2-58.

**Gamma.** Fits a gamma distribution using the function `gamfit`. See “Gamma Distribution” on page 2-64.

**Inverse Gaussian.** Fits an inverse Gaussian distribution. See “Inverse Gaussian Distribution” on page 2-42.

**Log-Logistic.** Fits a log-logistic distribution. See “Log-Logistic Distribution” on page 2-42.

**Logistic.** Fits a logistic distribution. See “Logistic Distribution” on page 2-42.

**Lognormal.** Fits a lognormal distribution using the function `lognfit`. See “Lognormal Distribution” on page 2-69.

**Nakagami.** Fits a Nakagami distribution. See “Nakagami Distribution” on page 2-42.

**Normal.** Fits a normal distribution using the function `normfit`. See “Normal Distribution” on page 2-73. See

**Rayleigh.** Fits a Rayleigh distribution using the function `raylfit`. “Rayleigh Distribution” on page 2-78.

**Rician.** Fits a Rician distribution. “Rician Distribution” on page 2-43.

**t Location-scale.** Fits a t location-scale distribution. “t Location-Scale Distribution” on page 2-43.

**Weibull.** Fits a Weibull 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 picture.

The options for nonparametric distributions are

- **Kernel** — The 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

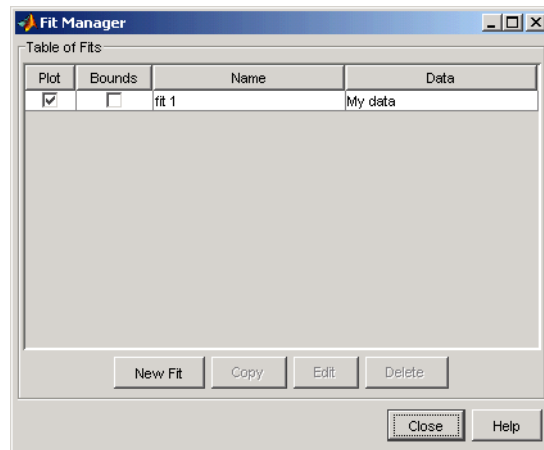
You can display confidence bounds for data sets and fits, provided that you set **Display Type** to Cumulative probability (CDF), Survivor function, Cumulative hazard, or Quantile for fits only.

- 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.

---

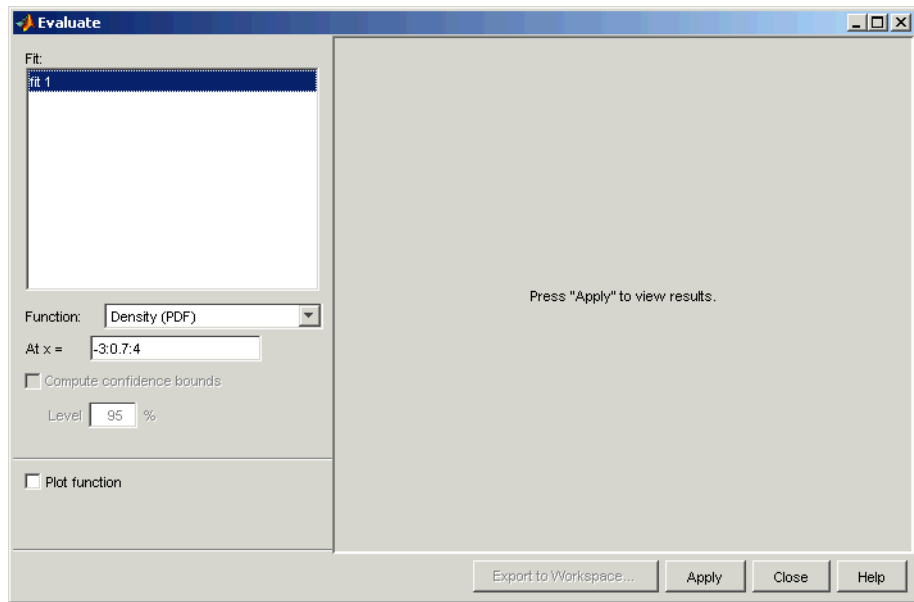
**Note** You can only edit the currently selected fit in the **Edit Fit** window. To edit a different fit, select it in the **Table of fits** and click **Edit** to open another **Edit Fit** window.

---

- **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. You can select multiple fits by pressing the **Ctrl** button and clicking the names of the fits with the mouse.
- **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.
  - Quartile (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 **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 2-16, at the points in the vector -3: 0.5: 3.

X	My fit		
	Y	LB	UB
-3	0.00241	0.00056	0.00868
-2.5	0.00803	0.00258	0.02178
-2	0.02296	0.00972	0.04892
-1.5	0.05649	0.03002	0.09865
-1	0.12026	0.07648	0.17926
-0.5	0.22289	0.16218	0.2948
0	0.3627	0.29003	0.44069
0.5	0.5239	0.44582	0.60107
1	0.68124	0.60427	0.75109
1.5	0.81122	0.74244	0.86734
2	0.9021	0.8478	0.94064
2.5	0.95589	0.91871	0.97795
3	0.98283	0.96092	0.99326

The window displays the following values in the columns of the table to the right of the **Fit** pane:

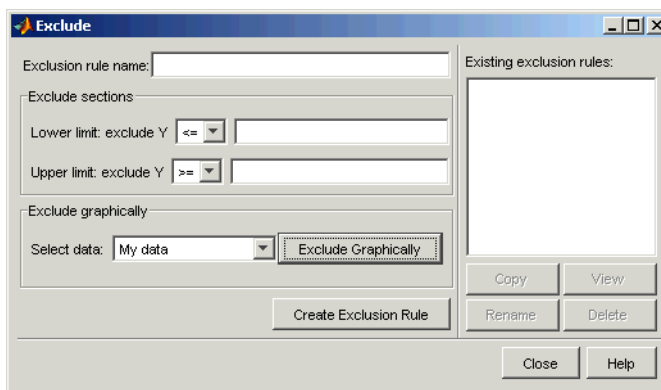
- **X** — The entries of the vector you enter in **At x =** field
- **Y** — The corresponding values of the CDF at the entries of X
- **LB** — The lower bounds for the confidence interval, if you select **Compute confidence bounds**
- **UB** — The upper bounds for the confidence interval, if you select **Compute confidence bounds**

To save the data displayed in the **Evaluate** window, click **Export to Workspace**. This saves the values in the table to a matrix in the MATLAB workspace.

### Excluding Data

To exclude values from fit, click the **Exclude** button in the main window of the Distribution Fitting Tool. This opens the **Exclude** window, in which you can create rules for excluding specified values. You can use these rules to exclude

data when you create a new fit in the **New Fit** window. The following figure shows the **Exclude** window.



The following sections describe how to create an exclusion rule.

### Exclusion Rule Name

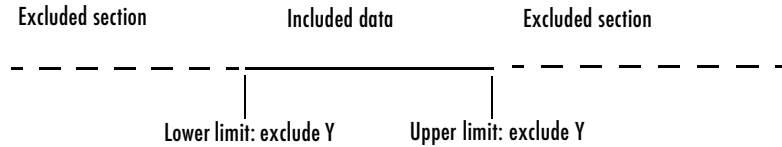
Enter a name for the exclusion rule in the **Exclusion rule name** field.

### Exclude Sections

In the **Exclude sections** pane, you can specify bounds for the excluded data:

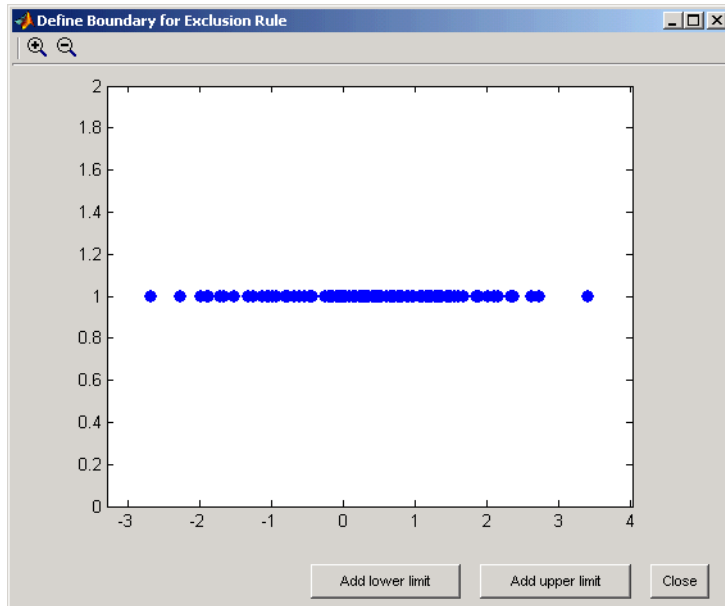
- In the **Lower limit: exclude Y** drop-down list, select  $\leq$  or  $<$  from the drop-down list and enter a scalar in the field to the right. This excludes values that are either less than or equal to or less than that scalar, respectively.
- In the **Upper limit: exclude Y** drop-down list, select  $\geq$  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.

The following diagram illustrates the values that are excluded by lower and upper limits.

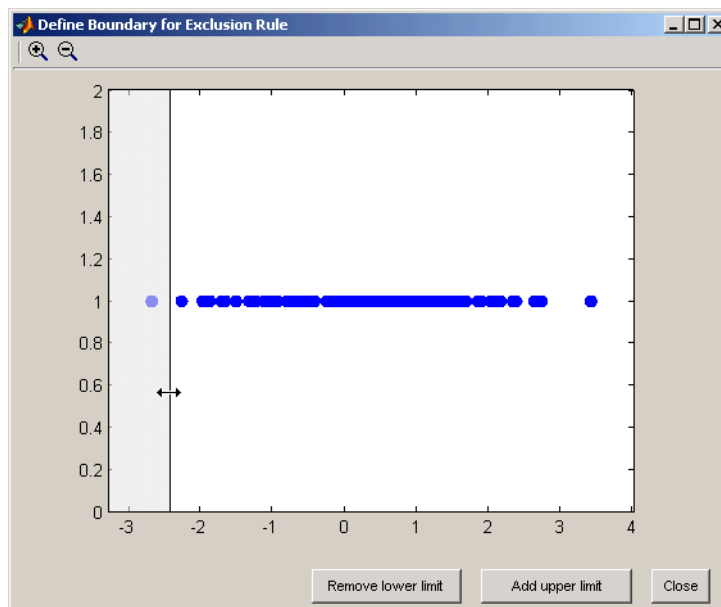


### Exclude Graphically

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 2-20, 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 Y** field in the **Exclude** window, as shown in the following figure.

The value displayed corresponds to the x-coordinate of the vertical line.

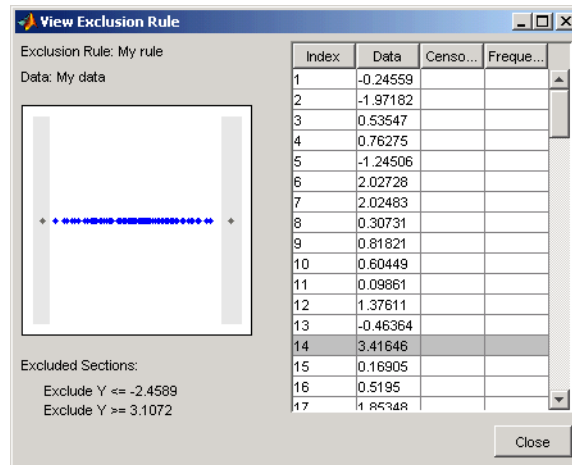
Similarly, you can set the upper limit for the boundary of the excluded region by clicking **Add Upper Limit** and moving the vertical line that appears at the right side of the plot window. After setting the lower and upper limits, click **Close** and return to the Exclude window.

### 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.

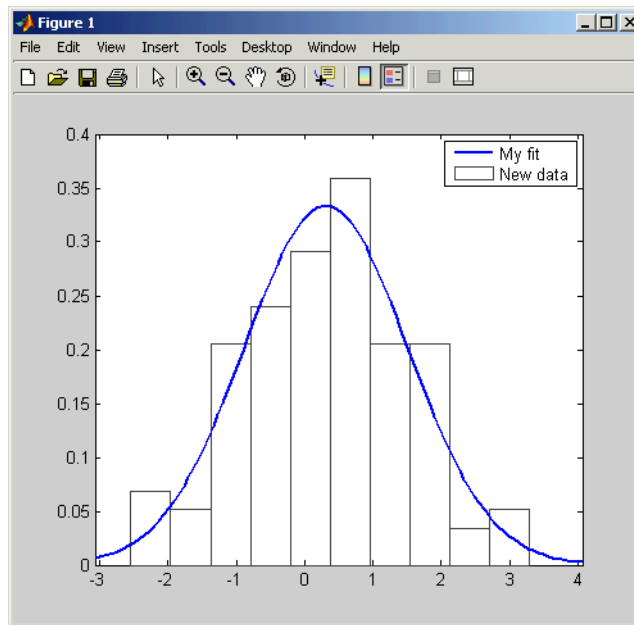
As an example, assuming you created the fit described in “Creating a New Fit” on page 2-24, do the following steps:

- 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

This section describes the distributions that are available in the Distribution Fitting Tool, but which have no corresponding command-line functions. For a complete list of the distributions you can use with the Distribution Fitting Tool, see “Available Distributions” on page 2-26.

**Birnbaum-Saunders Distribution**

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

If  $x$  has a Birnbaum-Saunders distribution with parameters  $\beta$  and  $\gamma$ , then

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

has a standard normal distribution.

**Inverse Gaussian Distribution**

The inverse Gaussian distribution has the density function

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

**Log-Logistic Distribution**

The variable  $x$  has a log logistic distribution with location parameter  $\mu$  and scale parameter  $\sigma > 0$  if  $\ln x$  has a logistic distribution with parameters  $\mu$  and  $\sigma$ .

**Logistic Distribution**

The logistic distribution has the density function

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

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

**Nakagami Distribution**

The Nakagami distribution has the density function

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

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

### Rician Distribution

The Rician distribution has the density function

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

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

### t Location-Scale Distribution

The t location-scale distribution has the density function

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

with location parameter  $\mu$ , scale parameter  $\sigma > 0$ , and shape parameter  $\nu > 0$ . If  $x$  has a t location-scale distribution, with parameters  $\mu$ ,  $\sigma$ , and  $\nu$ , then  $\frac{x-\mu}{\sigma}$  has a Student's t distribution with  $\nu$  degrees of freedom.

### Using the Distributions with Command-Line Functions

You can specify the distributions described in this section when using the functions `mle`, `cdf`, `icdf`, and `pdf`. To do so, set the first argument of the function to one of the following distribution names:

- 'birnbaumsaunders'
- 'inversegaussian'

- 'loglogistic'
- 'nakagami'
- 'rician'
- 'tlocationscale'

## Overview of the Distributions

The following sections describe the probability distributions that are available using functions at the command line. For additional distributions that are only available through the Distribution Fitting Tool, see “Additional Distributions Available in the Distribution Fitting Tool” on page 2-41:

- “Reproducing the Output of Random Number Functions” on page 2-46
- “Beta Distribution” on page 2-48
- “Binomial Distribution” on page 2-50
- “Chi-Square Distribution” on page 2-52
- “Noncentral Chi-Square Distribution” on page 2-53
- “Discrete Uniform Distribution” on page 2-55
- “Exponential Distribution” on page 2-56
- “Extreme Value Distribution” on page 2-58
- “F Distribution” on page 2-61
- “Noncentral F Distribution” on page 2-63
- “Gamma Distribution” on page 2-64
- “Geometric Distribution” on page 2-66
- “Hypergeometric Distribution” on page 2-67
- “Lognormal Distribution” on page 2-69
- “Negative Binomial Distribution” on page 2-70
- “Normal Distribution” on page 2-73
- “Poisson Distribution” on page 2-76
- “Rayleigh Distribution” on page 2-78
- “Student’s t Distribution” on page 2-79
- “Noncentral t Distribution” on page 2-80
- “Uniform (Continuous) Distribution” on page 2-82
- “Weibull Distribution” on page 2-83

## Reproducing the Output of Random Number Functions

The Statistics Toolbox contains functions that generate random samples from the distributions described in this section. These functions use the MATLAB functions `rand` and/or `randn` to generate their output. If you want to reproduce the exact output for any of these functions, you must set the state for `rand` and/or `randn` each time you call the function. For example, the following code sets the states for `rand` and `randn`:

```
state = 137;
rand('state', state);
randn('state', state);
```

If you execute this code with any fixed value for `state`, before calling one of the random number functions, the function always returns the same output. You might want to save these commands in an M-file script called `initstate.m`. Then, instead of three separate commands, you need only enter `initstate`.

The following table lists the random number functions and indicates their dependencies on `rand` and `randn`. To reproduce the output of a function in the left-hand column, set the states of the functions listed in the right-hand column

Random Number Functions	Dependencies on <code>rand</code> , <code>randn</code>
<code>betarnd</code>	<code>rand</code> , <code>randn</code>
<code>binornd</code>	<code>rand</code>
<code>chi2rnd</code>	<code>rand</code> , <code>randn</code>
<code>exprnd</code>	<code>rand</code>
<code>evrnd</code>	<code>rand</code>
<code>frnd</code>	<code>rand</code> , <code>randn</code>
<code>gamrnd</code>	<code>rand</code>
<code>geornd</code>	<code>rand</code>
<code>hygernd</code>	<code>rand</code>



<b>Random Number Functions</b>	<b>Dependencies on rand, randn</b>
iwishrnd	rand, randn
lognrnd	randn
mvnrnd	randn
mvtrnd	rand, randn
nbinrnd	rand, randn
ncfrnd	rand, randn
nctrnd	rand, randn
ncx2rnd	randn
normrnd	randn
poissrnd	rand, randn
raylrnd	randn
trnd	rand, randn
unidrnd	rand
unifrnd	rand
wblrnd	rand
wblrnd	rand
wishrnd	rand, randn

## Beta Distribution

The following sections provide an overview of the beta distribution.

### Background on the Beta Distribution

The beta distribution describes a family of curves that are unique in that they are nonzero only on the interval (0 1). A more general version of the function assigns parameters to the end-points 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  $X \sim \beta\left(\frac{v}{2}, \frac{v}{2}\right)$ .

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

### Definition of the Beta Distribution

The beta pdf is

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

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

### Parameter Estimation for the Beta Distribution

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

One popular criterion of goodness is to maximize the likelihood function. The likelihood has the same form as the beta pdf. But for the pdf, the parameters are known constants and the variable is  $x$ . The likelihood function reverses the roles of the variables. Here, the sample values (the  $x$ 's) are already observed. So they are the fixed constants. The variables are the unknown parameters.

Maximum likelihood estimation (MLE) involves calculating the values of the parameters that give the highest likelihood given the particular set of data.

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

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

```
phat =
    4.5330    0.2301
```

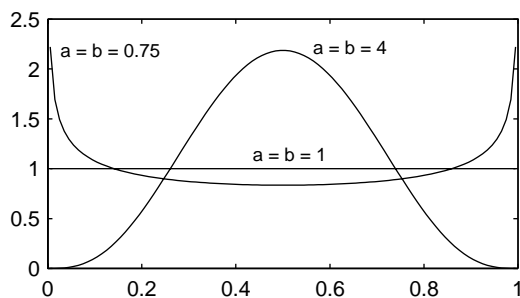
```
pci =
    2.8051    0.1771
    6.2610    0.2832
```

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

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

### Example and Plot of the Beta Distribution

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



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

## Binomial Distribution

The following sections provide an overview of the binomial distribution.

### Background of the Binomial Distribution

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

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

James Bernoulli derived the binomial distribution in 1713 (*Ars Conjectandi*). Earlier, Blaise Pascal had considered the special case where  $p = 1/2$ .

### Definition of the Binomial Distribution

The binomial pdf is

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

$$\text{where } \binom{n}{x} = \frac{n!}{x!(n-x)!} \text{ and } q = 1 - p.$$

The binomial distribution is discrete. For zero and for positive integers less than  $n$ , the pdf is nonzero.

### Parameter Estimation for the Binomial Distribution

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

One popular criterion of goodness is to maximize the likelihood function. The likelihood has the same form as the binomial pdf above. But for the pdf, the parameters ( $n$  and  $p$ ) are known constants and the variable is  $x$ . The likelihood function reverses the roles of the variables. Here, the sample values (the  $x$ 's)

are already observed. So they are the fixed constants. The variables are the unknown parameters. MLE involves calculating the value of  $p$  that give the highest likelihood given the particular set of data.

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

```
r = binornd(100,0.9)

r =
    88

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

phat =
    0.8800

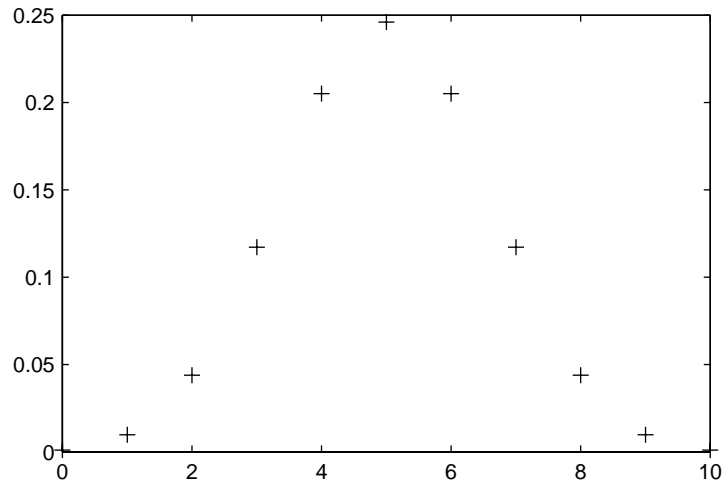
pci =
    0.7998
    0.9364
```

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

### Example and Plot of the Binomial Distribution

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

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



### Chi-Square Distribution

The following sections provide an overview of the  $\chi^2$  distribution.

#### Background of the Chi-Square Distribution

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

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

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

## Definition of the Chi-Square Distribution

The  $\chi^2$  pdf is

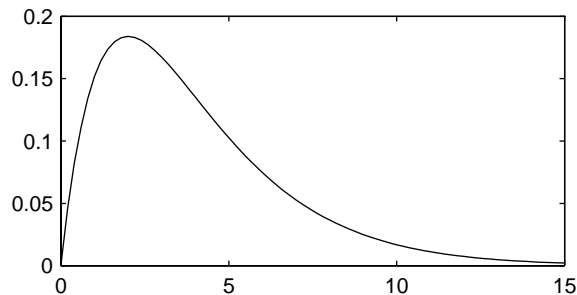
$$y = f(x|v) = \frac{x^{(v-2)/2} e^{-x/2}}{2^{v/2} \Gamma(v/2)}$$

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

## Example and Plot of the Chi-Square Distribution

The  $\chi^2$  distribution is skewed to the right especially for few degrees of freedom ( $v$ ). The plot shows the  $\chi^2$  distribution with four degrees of freedom.

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



## Noncentral Chi-Square Distribution

The following sections provide an overview of the noncentral  $\chi^2$  distribution.

### Background of the Noncentral Chi-Square Distribution

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

What if the normally distributed quantities have a mean other than zero? The sum of squares of these numbers yields the noncentral chi-square distribution.

The noncentral chi-square distribution requires two parameters; the degrees of freedom and the noncentrality parameter. The noncentrality parameter is the sum of the squared means of the normally distributed quantities.

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

### Definition of the Noncentral Chi-Square Distribution

There are many equivalent formulas for the noncentral chi-square distribution function. One formulation uses a modified Bessel function of the first kind. Another uses the generalized Laguerre polynomials. The Statistics Toolbox computes the cumulative distribution function values using a weighted sum of  $\chi^2$  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.

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

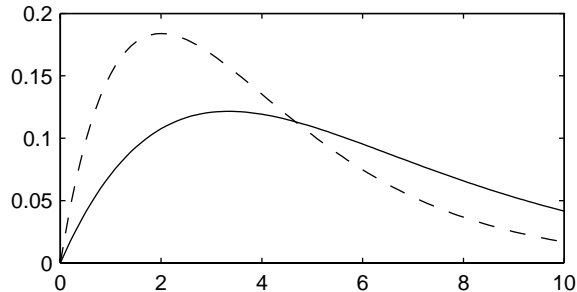
where  $\delta$  is the noncentrality parameter.

### Example of the Noncentral Chi-Square Distribution

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

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





## Discrete Uniform Distribution

The following sections provide an overview of the discrete uniform distribution.

### Background of the Discrete Uniform Distribution

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

### Definition of the Discrete Uniform Distribution

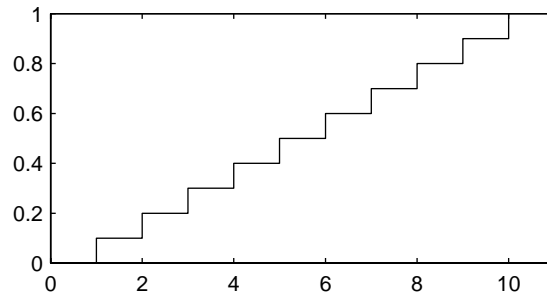
The discrete uniform pdf is

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

### Example and Plot of the Discrete Uniform Distribution

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

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



To 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
```

## Exponential Distribution

The following sections provide an overview of the exponential distribution.

### Background of the Exponential Distribution

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

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

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.

### Definition of the Exponential Distribution

The exponential pdf is

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

## Parameter Estimation for the Exponential Distribution

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

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

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

```
lifetimes = exprnd(700,100,1);
[muhat, muc1] = expfit(lifetimes)

muhat =

    672.8207

muc1 =

    547.4338
    810.9437
```

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

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

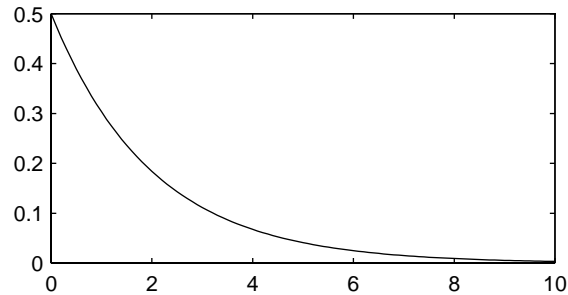
## Example and Plot of the Exponential Distribution

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

```
l = 10:10:60;  
lpd = l+0.1;  
deltap = (expcdf(lpd,50)-expcdf(l,50))./(1-expcdf(l,50))  
  
deltap =  
    0.0020    0.0020    0.0020    0.0020    0.0020    0.0020
```

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

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



## Extreme Value Distribution

The following sections provide an overview of the extreme value distribution.

### Background of the Extreme Value Distribution

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

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

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

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

### Definition of the Extreme Value Distribution

The probability density function for the extreme value distribution with location parameter  $\mu$  and scale parameter  $\sigma$  is

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

If  $T$  has a Weibull distribution with parameters  $a$  and  $b$ , as described in “Weibull Distribution” on page 2-83, then  $\log T$  has an extreme value distribution with parameters  $\mu = \log a$  and  $\sigma = 1/b$ .

### Parameter Estimation for the Extreme Value Distribution

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

Suppose you want to model the size of the smallest washer in each batch of 1000 from a manufacturing process. If you believe that the sizes are independent within and between each batch, you can fit an extreme value distribution to measurements of the minimum diameter from a series of eight experimental batches. The following code returns the MLEs of the distribution parameters as `parmhat` and the confidence intervals as the columns of `parmc1`.

```
x = [19.774 20.141 19.44 20.511 21.377 19.003 19.66 18.83];
[parmhat, parmc1] = evfit(x)

parmhat =
    20.2506    0.8223
```

```
parmc1 =  
    19.644 0.49861  
    20.857 1.3562
```

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

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

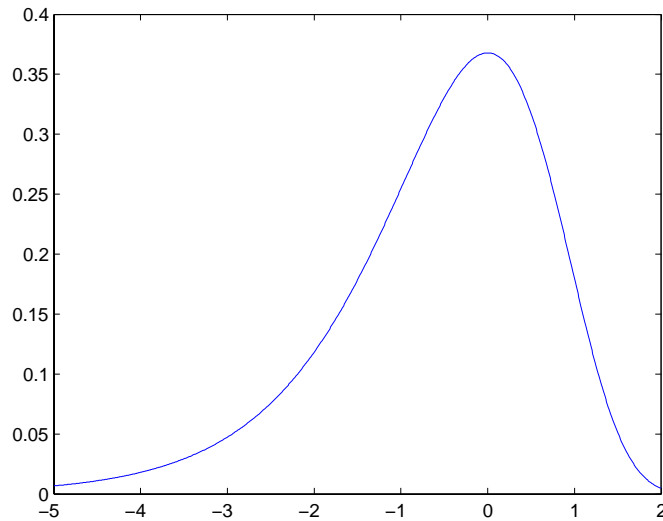
```
meanfit =  
    19.776
```

```
varfit =  
    1.1123
```

### Plot of the Extreme Value Distribution

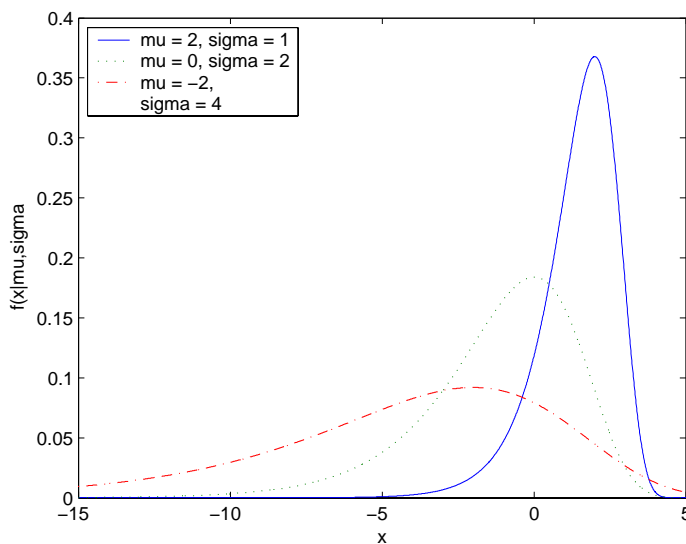
The following code generates a plot of the pdf for the extreme value distribution.

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



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

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



## F Distribution

The following sections provide an overview of the F distribution.

### Background of the F distribution

The F distribution has a natural relationship with the chi-square distribution. If  $\chi_1$  and  $\chi_2$  are both chi-square with  $v_1$  and  $v_2$  degrees of freedom respectively, then the statistic  $F$  below is F distributed.

$$F(v_1, v_2) = \frac{\frac{\chi_1}{v_1}}{\frac{\chi_2}{v_2}}$$

The two parameters,  $v_1$  and  $v_2$ , are the numerator and denominator degrees of freedom. That is,  $v_1$  and  $v_2$  are the number of independent pieces of information used to calculate  $\chi_1$  and  $\chi_2$ , respectively.

### Definition of the F distribution

The pdf for the F distribution is

$$y = f(x|v_1, v_2) = \frac{\Gamma\left[\frac{(v_1 + v_2)}{2}\right] \left(\frac{v_1}{v_2}\right)^{\frac{v_1}{2}} x^{\frac{v_1 - 2}{2}}}{\Gamma\left(\frac{v_1}{2}\right)\Gamma\left(\frac{v_2}{2}\right) \left[1 + \left(\frac{v_1}{v_2}\right)x\right]^{\frac{v_1 + v_2}{2}}}$$

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

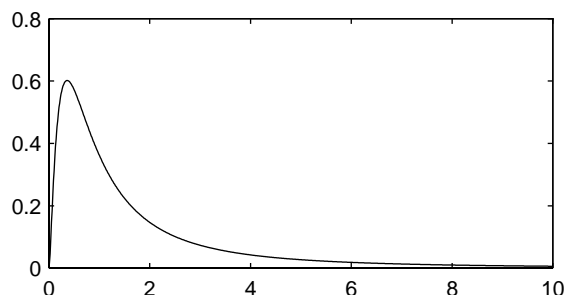
### Example and Plot of the F distribution

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

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

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





## Noncentral F Distribution

The following sections provide an overview of the noncentral F distribution.

### Background of the Noncentral F Distribution

As with the  $\chi^2$  distribution, the F distribution is a special case of the noncentral F distribution. The F distribution is the result of taking the ratio of two  $\chi^2$  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.

### Definition of the Noncentral F Distribution

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

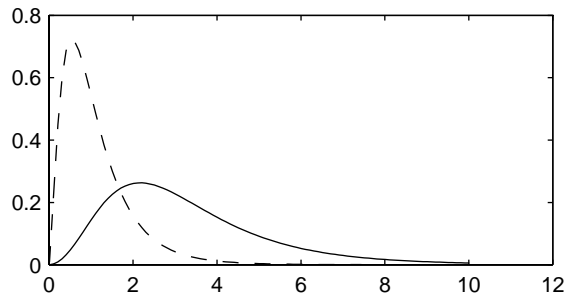
$$F(x|v_1, v_2, \delta) = \sum_{j=0}^{\infty} \left( \frac{\left(\frac{1}{2}\delta\right)^j}{j!} e^{-\frac{\delta}{2}} \right) I\left(\frac{v_1 \cdot x}{v_2 + v_1 \cdot x} \middle| \frac{v_1}{2} + j, \frac{v_2}{2}\right)$$

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

### Example and Plot of the Noncentral F Distribution

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

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



## Gamma Distribution

The following sections provide an overview of the gamma distribution.

### Background of the Gamma Distribution

The gamma distribution is a family of curves 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.

$$\Gamma(x|a, b) = \text{gammainc}\left(\frac{x}{b}, a\right)$$

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.

## Definition of the Gamma Distribution

The gamma pdf is

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

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

## Parameter Estimation for the Gamma Distribution

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

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

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

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

phat =

    10.9821    4.7258

pci =

    7.4001    3.1543
   14.5640    6.2974
```

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

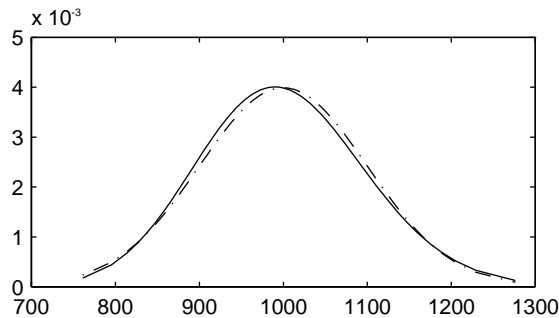
Similarly the MLE for parameter  $b$  is 4.7, compared to the true value of 5. The 95% confidence interval for  $b$  goes from 3.2 to 6.3, which also includes the true value.

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

### Example and Plot of the Gamma Distribution

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

```
x = gaminv((0.005:0.01:0.995),100,10);  
y = gampdf(x,100,10);  
y1 = normpdf(x,1000,100);  
plot(x,y,'-',x,y1,'-.-')
```



### Geometric Distribution

The following sections provide an overview of the geometric distribution.

#### Background of the Geometric Distribution

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

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

## Definition of the Geometric Distribution

The geometric pdf is

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

where  $q = 1 - p$ .

## Example and Plot of the Geometric Distribution

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

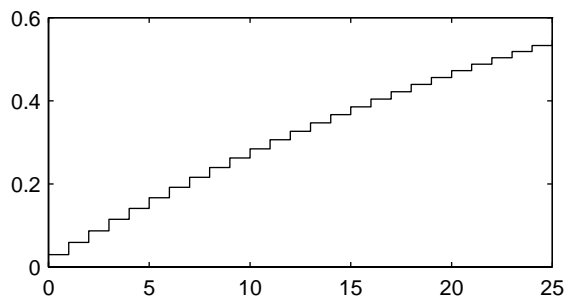
```
1 - geocdf(25,0.03)
```

```
ans =
```

```
0.4530
```

The plot shows the cdf for this scenario.

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



## Hypergeometric Distribution

The following sections provide an overview of the hypergeometric distribution.

### Background of the Hypergeometric Distribution

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

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

The hypergeometric distribution has three parameters that have direct physical interpretations.  $M$  is the size of the population.  $K$  is the number of items with the desired characteristic in the population.  $n$  is the number of samples drawn. Sampling “without replacement” means that once a particular sample is chosen, it is removed from the relevant population for all subsequent selections.

### Definition of the Hypergeometric Distribution

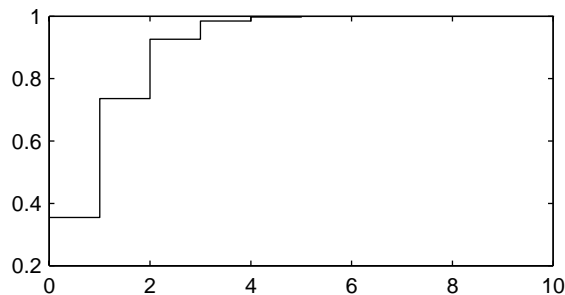
The hypergeometric pdf is

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

### Example and Plot of the Hypergeometric Distribution

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

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



## Lognormal Distribution

The following sections provide an overview of the lognormal distribution.

### Background of the Lognormal Distribution

The normal and lognormal distributions are closely related. If  $X$  is distributed lognormal with parameters  $\mu$  and  $\sigma^2$ , then  $\ln X$  is distributed normal with parameters  $\mu$  and  $\sigma^2$ .

The lognormal distribution is applicable when the quantity of interest must be positive, since  $\ln X$  exists only when the random variable  $X$  is positive. Economists often model the distribution of income using a lognormal distribution.

### Definition of the Lognormal Distribution

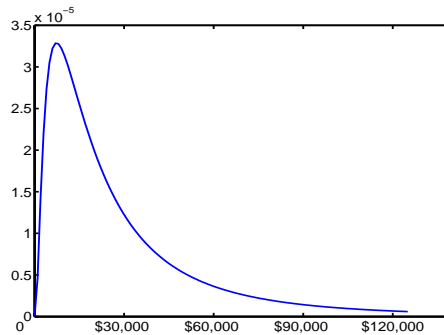
The lognormal pdf is

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

### Example and Plot of the Lognormal Distribution

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'))
```



### Negative Binomial Distribution

The following sections provide an overview of the negative binomial distribution.

- “Background of the Negative Binomial Distribution” on page 2-70
- “Definition of the Negative Binomial Distribution” on page 2-71
- “Parameter Estimation for the Negative Binomial Distribution” on page 2-71
- “Example and Plot of the Negative Binomial Distribution” on page 2-73

#### Background of the Negative Binomial Distribution

In its simplest form, the negative binomial distribution models the number of successes before a specified number of failures is reached in an independent series of repeated identical trials. It can also be thought of as modeling the total number of trials required before a specified number of successes, thus motivating its name as the inverse of the binomial distribution. Its parameters are the probability of success in a single trial,  $p$ , and the number of failures,  $r$ . A special case of the negative binomial distribution, when  $r = 1$ , is the geometric distribution (also known as the Pascal distribution), which models the number of successes before the first failure.

More generally, the  $r$  parameter can take on noninteger values. This form of the negative binomial has no interpretation in terms of repeated trials, but, like the Poisson distribution, it is useful in modeling count data. It is, however, more general than the Poisson, because the negative binomial has a variance that is greater than its mean, often making it suitable for count data that do not meet the assumptions of the Poisson distribution. In the limit, as the



parameter  $r$  increases to infinity, the negative binomial distribution approaches the Poisson distribution.

### Definition of the Negative Binomial Distribution

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

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

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

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

### Parameter Estimation for the Negative Binomial Distribution

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

```
accident = [2 3 4 2 3 1 12 8 14 31 23 1 10 7 0];
mean(accident)
ans =
    8.0667

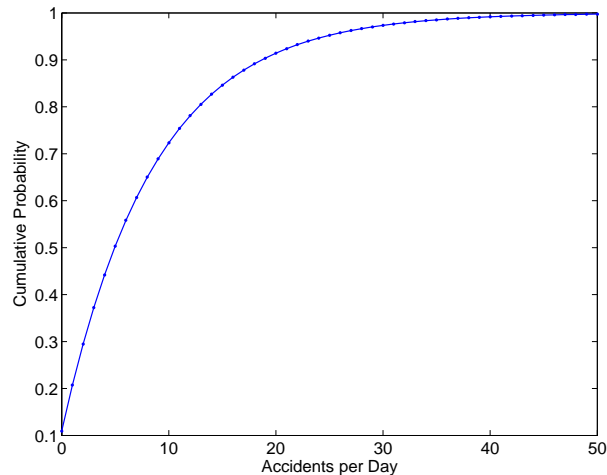
var(accident)
ans =
   79.352
```

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

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

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

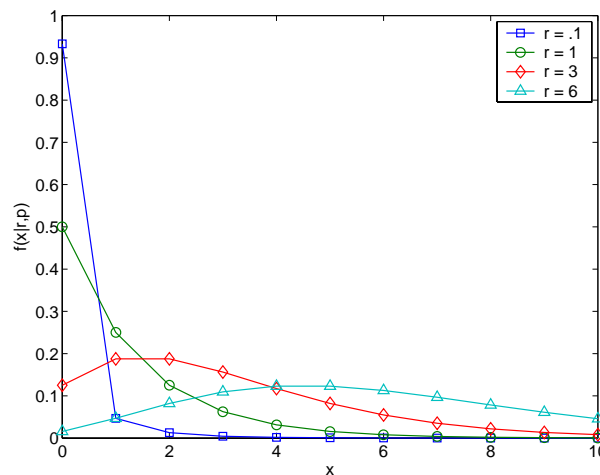
```
plot(0:50,nbincdf(0:50,phat(1),phat(2)),'.-');
xlabel('Accidents per Day')
ylabel('Cumulative Probability')
```



## Example and Plot of the Negative Binomial Distribution

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

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



## Normal Distribution

The following sections provide an overview of the normal distribution.

### Background of the Normal Distribution

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

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

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

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.

### Definition of the Normal Distribution

The normal pdf is

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

### Parameter Estimation for the Normal Distribution

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

The MVUEs of parameters  $\mu$  and  $\sigma^2$  for the normal distribution are the sample average and variance. The sample average is also the MLE for  $\mu$ . 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  $\sigma^2$ , and equation 2 is the MVUE.

As an example, suppose you want to estimate the mean,  $\mu$ , and the variance,  $\sigma^2$ , of the heights of all 4th grade children in the United States. The function `normfit` returns the MVUE for  $\mu$ , the square root of the MVUE for  $\sigma^2$ , and confidence intervals for  $\mu$  and  $\sigma^2$ . Here is a playful example modeling the heights in inches of a randomly chosen 4th grade class.

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

mu =
    50.2025

s =
    1.7946

muci =
    49.5210
    50.8841

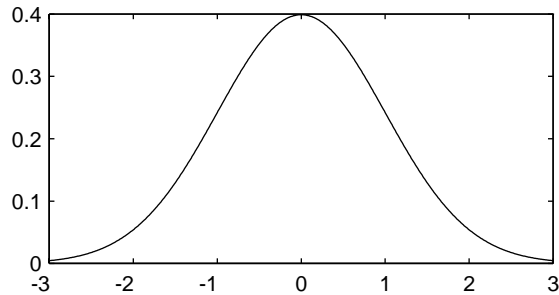
sci =
    1.4292
    2.4125
```

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

```
s^2
ans =
    3.2206
```

### Example and Plot of the Normal Distribution

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



### Poisson Distribution

The following sections provide an overview of the Poisson distribution.

#### Background of the Poisson Distribution

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

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

As Poisson (1837) showed, the Poisson distribution is the limiting case of a binomial distribution where  $N$  approaches infinity and  $p$  goes to zero while  $Np = \lambda$ .

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

## Definition of the Poisson Distribution

The Poisson pdf is

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

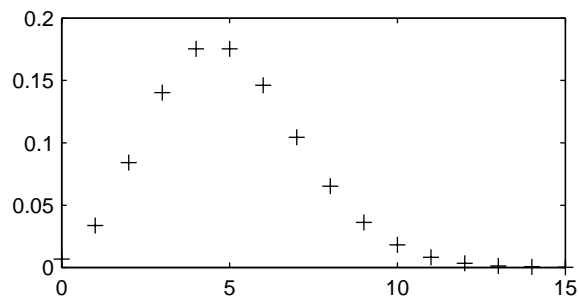
## Parameter Estimation for the Poisson Distribution

The MLE and the MVUE of the Poisson parameter,  $\lambda$ , 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. The Statistics Toolbox makes use of this fact to calculate confidence intervals on  $\lambda$ . As  $\lambda$  gets large the Poisson distribution can be approximated by a normal distribution with  $\mu = \lambda$  and  $\sigma^2 = \lambda$ . The Statistics Toolbox uses this approximation for calculating confidence intervals for values of  $\lambda$  greater than 100.

## Example and Plot of the Poisson Distribution

The plot shows the probability for each nonnegative integer when  $\lambda = 5$ .

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



## Rayleigh Distribution

The following sections provide an overview of the Rayleigh distribution.

### Background of the Rayleigh Distribution

The Rayleigh distribution is a special case of the Weibull distribution. If  $A$  and  $B$  are the parameters of the Weibull distribution, then the Rayleigh distribution with parameter  $b$  is equivalent to the Weibull distribution with parameters  $A = 1/(2b^2)$  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.

### Definition of the Rayleigh Distribution

The Rayleigh pdf is

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

### Parameter Estimation for the Rayleigh Distribution

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

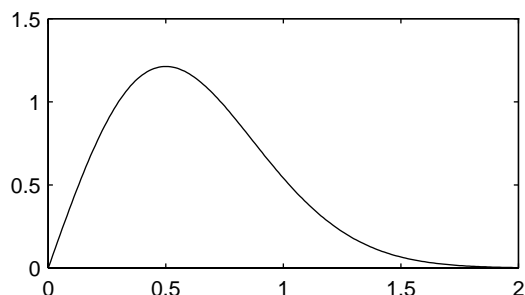
$$b = \sqrt{\frac{1}{2n} \sum_{i=1}^n x_i^2}$$

### Example and Plot of the Rayleigh Distribution

The following commands generate a plot of the Rayleigh pdf.

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





## Student's t Distribution

The following sections provide an overview of Student's t distribution.

### Background of Student's t Distribution

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

W. S. Gossett (1908) discovered the distribution through his work at the Guinness brewery. At that time, Guinness did not allow its staff to publish, so Gossett used the pseudonym Student.

If  $\bar{x}$  and  $s$  are the mean and standard deviation of an independent random sample of size  $n$  from a normal distribution with mean  $\mu$  and  $\sigma^2 = n$ , then

$$t(\nu) = \frac{\bar{x} - \mu}{s}$$

$$\nu = n - 1$$

### Definition of Student's t Distribution

Student's t pdf is

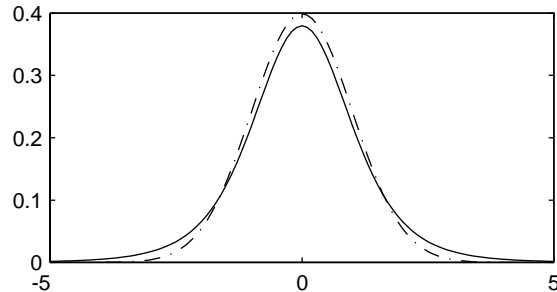
$$y = f(x|\nu) = \frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\Gamma\left(\frac{\nu}{2}\right)} \frac{1}{\sqrt{\nu\pi}} \frac{1}{\left(1 + \frac{x^2}{\nu}\right)^{\frac{\nu+1}{2}}}$$

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

### Example and Plot of Student's t Distribution

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

```
x = -5:0.1:5;  
y = tpdf(x,5);  
z = normpdf(x,0,1);  
plot(x,y,'-',x,z,'-.')
```



### Noncentral t Distribution

The following sections provide an overview of the noncentral t distribution.

#### Background of the Noncentral t Distribution

The noncentral t distribution is a generalization of the familiar Student's t distribution. Recall that the ordinary Student's t distribution is defined as follows. If  $\bar{x}$  and  $s$  are the mean and standard deviation, respectively, of an independent random sample of size  $n$  from a normal distribution with mean  $\mu_0$  and standard deviation  $\sigma$ , then

$$t(\nu) = \frac{\bar{x} - \mu_0}{s / (\sqrt{n})}$$

where  $\nu = n - 1$ , has the ordinary Student's t distribution with  $\nu$  degrees of freedom.

Now, suppose that the mean of  $x$  is  $\mu$ , rather than the hypothesized value  $\mu_0$ . Then the ratio on the right-hand side of the preceding equation has a noncentral  $t$  distribution with a noncentrality parameter  $\delta$  equal to

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

$\delta$  is the normalized difference between the true mean and the hypothesized mean.

The noncentral  $t$  distribution enables you to determine the probability of detecting a difference between  $\mu$  and  $\mu_0$  in a  $t$  test. This probability is the *power* of the test. The power increases as the difference  $\mu - \mu_0$  increases, and also as the sample size increases.

### Definition of the Noncentral $t$ Distribution

The most general representation of the noncentral  $t$  distribution is quite complicated. Johnson and Kotz (1970) give a formula for the probability that a noncentral  $t$  variate falls in the range  $[-t, t]$ .

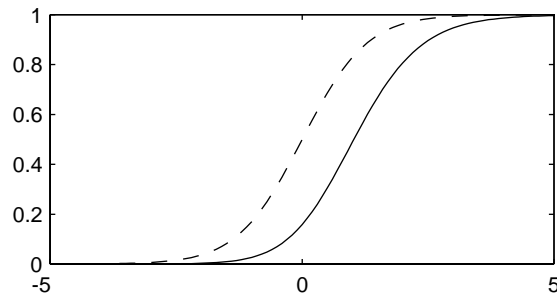
$$Pr((-t) < x < t | (v, \delta)) = \sum_{j=0}^{\infty} \left( \frac{\left(\frac{1}{2}\delta^2\right)^j}{j!} e^{-\frac{\delta^2}{2}} \right) I\left(\frac{x^2}{v+x^2} \middle| \frac{1}{2} + j, \frac{v}{2}\right)$$

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

### Example and Plot of the Noncentral $t$ Distribution

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

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



### Uniform (Continuous) Distribution

The following sections provide an overview of the uniform distribution.

#### Background of the Uniform Distribution

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

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

#### Definition of the Uniform Distribution

The uniform cdf is

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

#### Parameter Estimation for the Uniform Distribution

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

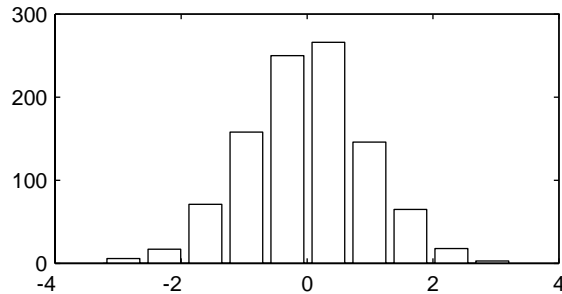
#### Example and Plot of the Uniform Distribution

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

```

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

```



## Weibull Distribution

The following sections provide an overview of the Weibull distribution.

### Background of the Weibull Distribution

Waloddi Weibull (1939) 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.

### Definition of the Weibull Distribution

The Weibull pdf is

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

### Parameter Estimation for the Weibull Distribution

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

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

p =
0.4715    1.9811

ci =

    0.4248    1.7067
    0.5233    2.2996
```

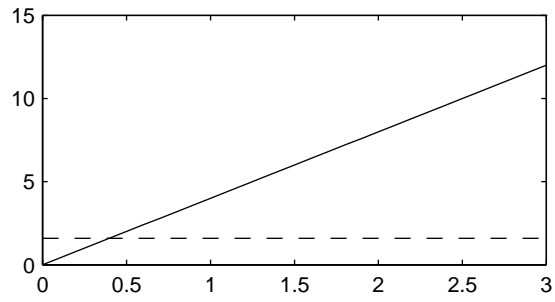
The default 95% confidence interval for each parameter contains the true value.

### Example and Plot of the Weibull Distribution

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

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

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







# Descriptive Statistics

---

Measures of Central Tendency (Location) (p. 3-2)	Describes how to calculate measures of central tendency, such as mean and median.
Measures of Dispersion (p. 3-4)	Describes how to calculate measures of dispersion, such as variance and standard deviation.
Functions for Data with Missing Values (NaNs) (p. 3-6)	Describes tools for analyzing data that has missing values
Function for Grouped Data (p. 3-8)	Explains how to use the <code>grpstats</code> function to group subsets of data in order to analyze them.
Percentiles and Graphical Descriptions (p. 3-10)	Describes how to analyze and estimate data distributions empirically.
The Bootstrap (p. 3-17)	Describes how to implement the bootstrap procedure.

## Measures of Central Tendency (Location)

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

The table gives the function names and descriptions.

<b>Measures of Location</b>	
geomean	Geometric mean
harmmean	Harmonic mean
mean	Arithmetic average (in MATLAB)
median	50th percentile (in MATLAB)
trimmean	Trimmed mean

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

Unfortunately, outliers, data entry errors, or glitches exist in almost all real data. The sample average 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.

<b>Measures of Dispersion</b>	
iqr	Interquartile Range
mad	Mean Absolute Deviation
range	Range
std	Standard deviation (in MATLAB)
var	Variance (in MATLAB)

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

The standard deviation and the variance are popular measures of spread that are optimal for normally distributed samples. The sample variance is the MVUE of the normal parameter  $\sigma^2$ . 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<sup>2</sup>, 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
```

## Functions for Data with Missing Values (NaNs)

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

Here is a simple example.

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

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

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

```
sum(m)  
  
ans =  
    NaN    NaN    15
```

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

```
nansum(m)  
  
ans =  
     7    10    13
```

---

### NaN Functions

---

nanmax	Maximum ignoring NaNs
nanmean	Mean ignoring NaNs
nanmedian	Median ignoring NaNs
nanmin	Minimum ignoring NaNs

---

<b>NaN Functions</b>	
nanstd	Standard deviation ignoring NaNs
nansum	Sum ignoring NaNs

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

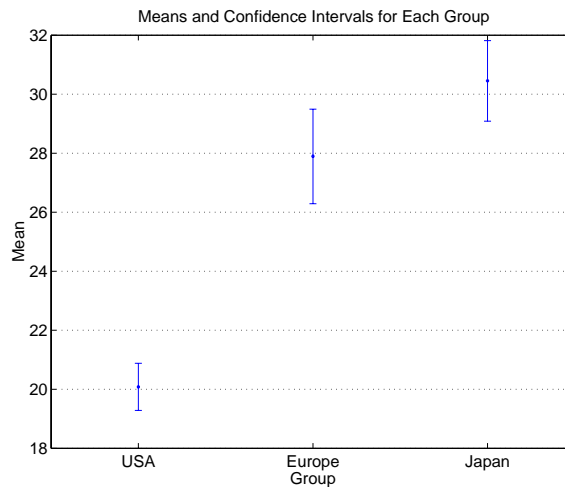
## Function for Grouped Data

As you saw in the previous section, the descriptive statistics functions can compute statistics on each column in a matrix. Sometimes, however, you may have your data arranged differently so that measurements appear in one column or variable, and a grouping code appears in a second column or variable. Although the MATLAB syntax makes it simple to apply functions to a subset of an array, in this case it is simpler to use the `grpstats` function.

The `grpstats` function can compute the mean, standard error of the mean, and count (number of observations) for each group defined by one or more grouping variables. If you supply a significance level, it also creates a graph of the group means with confidence intervals.

As an example, load the larger car data set. You can look at the average value of MPG (miles per gallon) for cars grouped by `org` (location of the origin of the car).

```
load carbig
grpstats(MPG,org,0.05)
ans =
    20.084
    27.891
    30.451
```





You can also get the complete set of statistics for MPG grouped by three variables: org, cyl4 (the engine has four cylinders or not), and when (when the car was made).

```
[m,s,c,n] = grpstats(MPG,{org cyl4 when});
[n num2cell([m s c])]

ans =
```

'USA'	'Other'	'Early'	[14.896]	[0.33306]	[77]
'USA'	'Other'	'Mid'	[17.479]	[0.30225]	[75]
'USA'	'Other'	'Late'	[21.536]	[0.97961]	[25]
'USA'	'Four'	'Early'	[23.333]	[0.87328]	[12]
'USA'	'Four'	'Mid'	[27.027]	[0.75456]	[22]
'USA'	'Four'	'Late'	[29.734]	[0.71126]	[38]
'Europe'	'Other'	'Mid'	[ 17.5]	[ 0.9478]	[ 4]
'Europe'	'Other'	'Late'	[30.833]	[ 3.1761]	[ 3]
'Europe'	'Four'	'Early'	[24.714]	[0.73076]	[21]
'Europe'	'Four'	'Mid'	[26.912]	[ 1.0116]	[26]
'Europe'	'Four'	'Late'	[ 35.7]	[ 1.4265]	[16]
'Japan'	'Other'	'Early'	[ 19]	[0.57735]	[ 3]
'Japan'	'Other'	'Mid'	[20.833]	[0.92796]	[ 3]
'Japan'	'Other'	'Late'	[ 26.5]	[ 2.0972]	[ 4]
'Japan'	'Four'	'Early'	[26.083]	[ 1.1772]	[12]
'Japan'	'Four'	'Mid'	[ 29.5]	[0.86547]	[25]
'Japan'	'Four'	'Late'	[ 35.3]	[0.68346]	[32]

## Percentiles and Graphical Descriptions

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

- “Percentiles” on page 3-10
- “Probability Density Estimation” on page 3-12
- “Empirical Cumulative Distribution Function” on page 3-15

### Percentiles

Another option is to compute a reasonable number of the sample percentiles. This provides information about the shape of the data as well as its location and spread.

The example shows the result of looking at every quartile of a sample containing a mixture of two distributions.

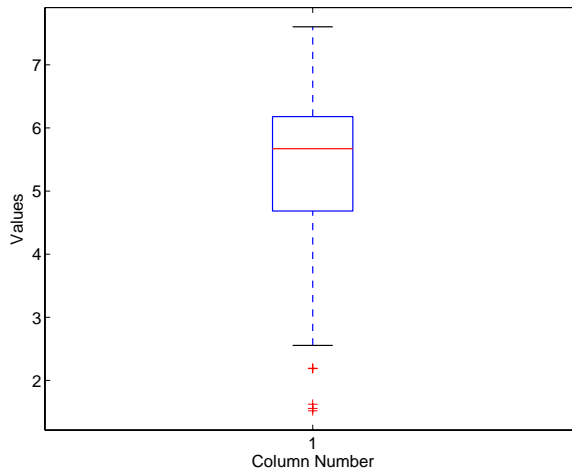
```
x = [normrnd(4,1,1,100) normrnd(6,0.5,1,200)];
p = 100*(0:0.25:1);
y = prctile(x,p);
z = [p;y]

z =
           0   25.0000   50.0000   75.0000  100.0000
    1.5172   4.6842   5.6706   6.1804   7.6035
```

Compare the first two quantiles to the rest.

The box plot is a graph for descriptive statistics. The following graph is a box plot of the preceding data.

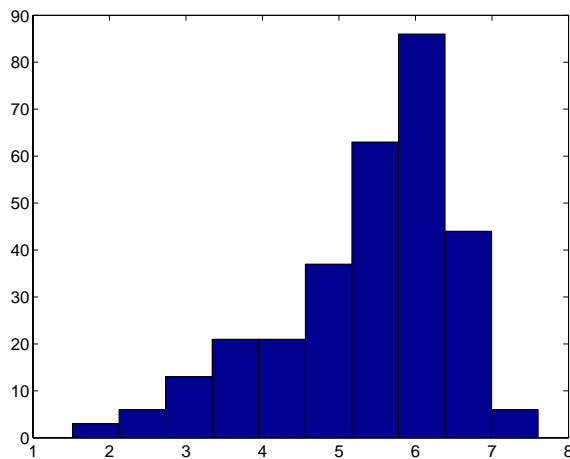
```
boxplot(x)
```



The long lower tail and plus signs show the lack of symmetry in the sample values. For more information on box plots, see “Statistical Plots” on page 8-1.

The histogram is a complementary graph.

`hist(x)`

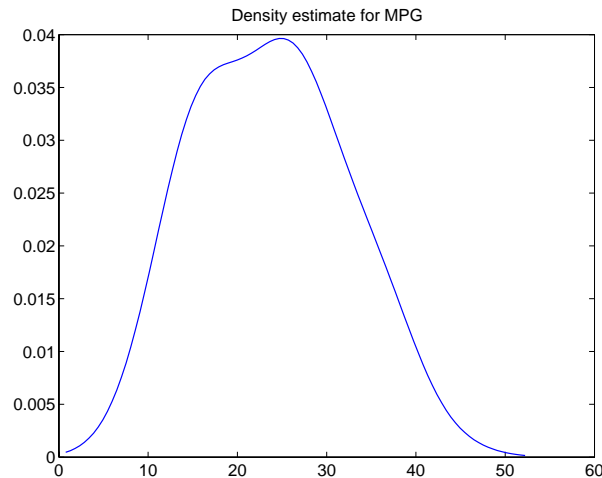


## Probability Density Estimation

You can also describe a data sample by estimating its density in a nonparametric way. The `ksdensity` function does this by using a kernel smoothing function and an associated bandwidth to estimate the density.

This example uses the `carsmall` data set to estimate the probability density of the MPG (miles per gallon) measurements for 94 cars. It uses the default kernel function, a normal distribution, and its default bandwidth.

```
cars = load('carsmall','MPG','Origin');
MPG = cars.MPG;
Origin = cars.Origin;
[f,x] = ksdensity(MPG);
plot(x,f);
title('Density estimate for MPG')
```

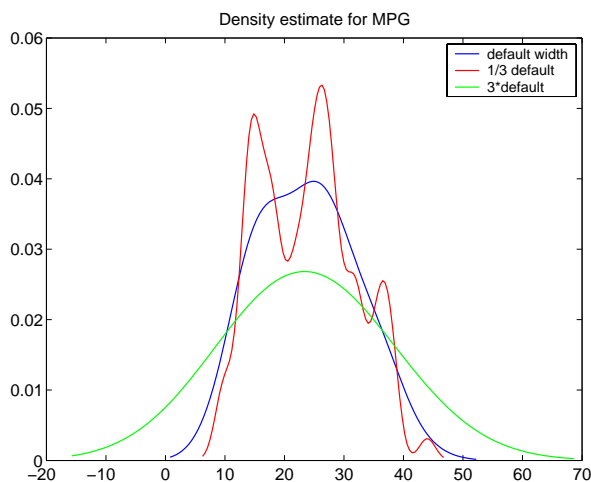


### Kernel Bandwidth

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

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

```
[f,x,u] = ksdensity(MPG);
plot(x,f)
title('Density estimate for MPG')
hold on
[f,x] = ksdensity(MPG,'width',u/3);
plot(x,f,'r');
[f,x] = ksdensity(MPG,'width',u*3);
plot(x,f,'g');
legend('default width','1/3 default','3*default')
hold off
```



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

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

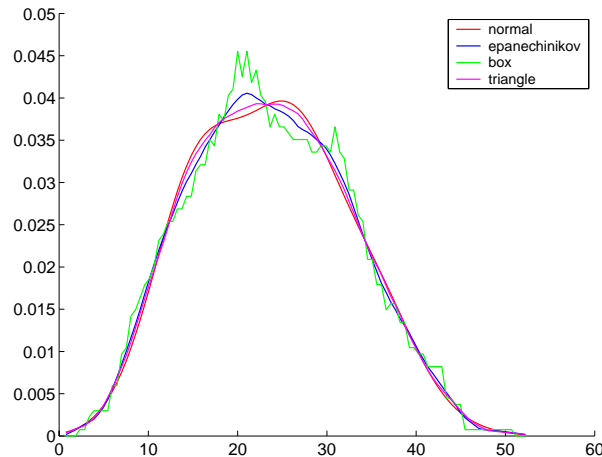
A reasonable choice of width might lead to a curve that is intermediate between the red and blue curves.

### Kernel Smoothing Function

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

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

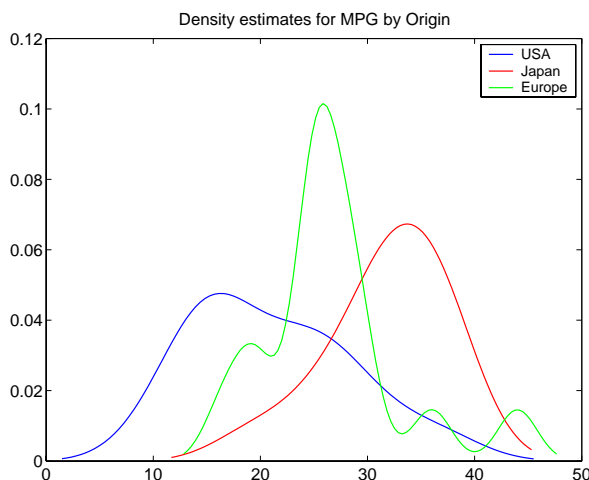
```
hname = {'normal' 'epanechnikov' 'box' 'triangle'};
hold on;
colors = {'r' 'b' 'g' 'm'};
for j=1:4
    [f,x] = ksdensity(MPG,'kernel',hname{j});
    plot(x,f,colors{j});
end
legend(hname{:});
hold off
```



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

## Usefulness of Smooth Density Estimates

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



## Empirical Cumulative Distribution Function

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

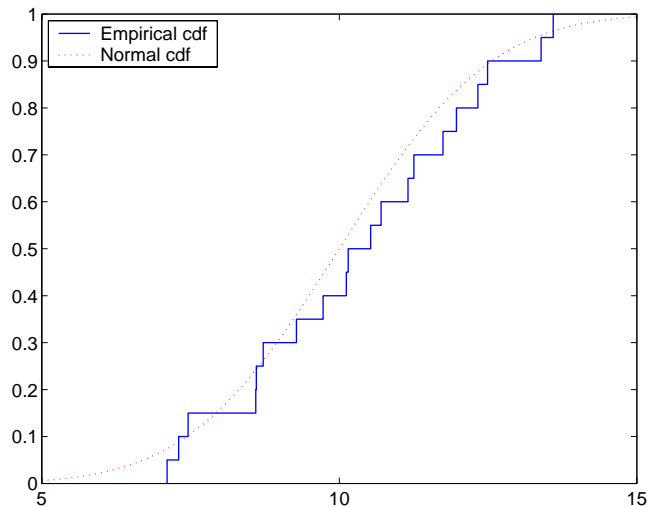
Similarly, it is possible to produce an empirical version of the cumulative distribution function (cdf). The `ecdf` function computes this empirical cdf. It returns the values of a function  $F$  such that  $F(x)$  represents the proportion of observations in a sample less than or equal to  $x$ .

The idea behind the empirical cdf is simple. It is a function that assigns probability  $1/n$  to each of  $n$  observations in a sample. Its graph has a stair-step appearance. If a sample comes from a distribution in a parametric family (such as a normal distribution), its empirical cdf is likely to resemble the

parametric distribution. If not, its empirical distribution still gives an estimate of the cdf for the distribution that generated the data.

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

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



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



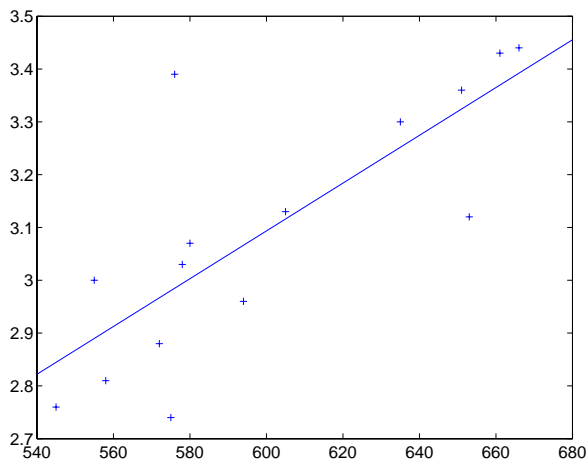
## The Bootstrap

In recent years the statistical literature has examined the properties of resampling as a means to acquire information about the uncertainty of statistical estimators.

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 (1993) 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 `corrcoef` function.

```
rhohat = corrcoef(lsat,gpa)

rhohat =

    1.0000    0.7764
    0.7764    1.0000
```

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

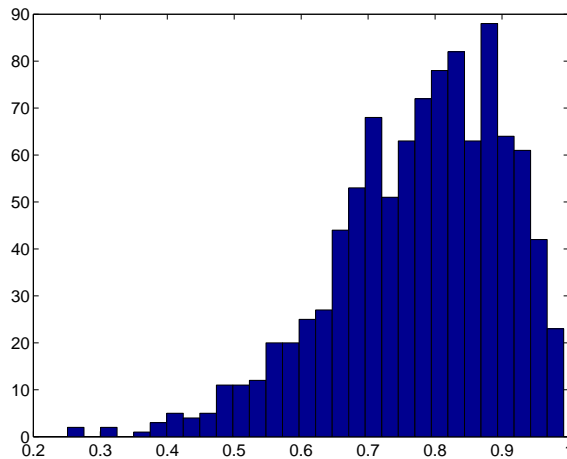
Using the `bootstrp` function you can resample the `lsat` and `gpa` vectors as many times as you like and consider the variation in the resulting correlation coefficients.

Here is an example.

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

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

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



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

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.



# Linear Models

---

Introduction (p. 4-2)	Introduces the concept of a linear model.
One-Way Analysis of Variance (ANOVA) (p. 4-3)	Describes how to perform one-way analysis of variance.
Two-Way Analysis of Variance (ANOVA) (p. 4-8)	Describes how to perform two-way analysis of variance.
N-Way Analysis of Variance (p. 4-11)	Describes how to perform analysis of variance with more than two factors.
ANOVA with Random Effects (p. 4-18)	Describes how to perform analysis of variance with random effects.
Analysis of Covariance (p. 4-25)	Describes how to perform analysis of covariance.
Multiple Linear Regression (p. 4-33)	Describes how to perform multiple linear regression.
Quadratic Response Surface Models (p. 4-42)	Describes how to analyze quadratic response surface models.
Stepwise Regression (p. 4-45)	Describes an interactive tool for performing stepwise regression.
Generalized Linear Models (p. 4-50)	Describes how to analyze generalized linear models.
Robust and Nonparametric Methods (p. 4-55)	Describes robust and nonparametric methods.

## Introduction

Linear models represent the relationship between a continuous response variable and one or more predictor variables (either continuous or categorical) in the form

$$y = X\beta + \varepsilon$$

where:

- $y$  is an  $n$ -by-1 vector of observations of the response variable.
- $X$  is the  $n$ -by- $p$  design matrix determined by the predictors.
- $\beta$  is a  $p$ -by-1 vector of parameters.
- $\varepsilon$  is an  $n$ -by-1 vector of random disturbances, independent of each other and usually having a normal distribution.

MATLAB uses this general form of the linear model to solve a variety of specific regression and analysis of variance (ANOVA) problems. For example, for polynomial and multiple regression problems, the columns of  $X$  are predictor variable values or powers of such values. For one-way, two-way, and higher-way ANOVA models, the columns of  $X$  are dummy (or indicator) variables that encode the predictor categories. For analysis of covariance (ANOCOVA) models,  $X$  contains values of a continuous predictor and codes for a categorical predictor.

---

**Note** See Chapter 5, “Nonlinear Regression Models” for information on fitting nonlinear models.

---

## One-Way Analysis of Variance (ANOVA)

The purpose of one-way ANOVA is to find out whether data from several groups have a common mean. That is, to determine whether the groups are actually different in the measured characteristic.

One-way ANOVA is a simple special case of the linear model. The one-way ANOVA form of the model is

$$y_{ij} = \alpha_{.j} + \varepsilon_{ij}$$

where:

- $y_{ij}$  is a matrix of observations in which each column represents a different group.
- $\alpha_{.j}$  is a matrix whose columns are the group means. (The “dot j” notation means that  $\alpha$  applies to all rows of the  $j$ th column. That is, the value  $\alpha_{ij}$  is the same for all  $i$ .)
- $\varepsilon_{ij}$  is a matrix of random disturbances.

The model assumes that the columns of  $y$  are a constant plus a random disturbance. You want to know if the constants are all the same.

The following sections explore one-way ANOVA in greater detail:

- “Example: One-Way ANOVA” on page 4-3
- “Multiple Comparisons” on page 4-5

### Example: One-Way ANOVA

The data below comes from a study by Hogg and Ledolter (1987) 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

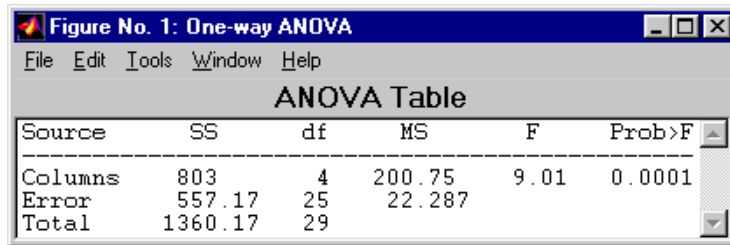
```

```

[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.



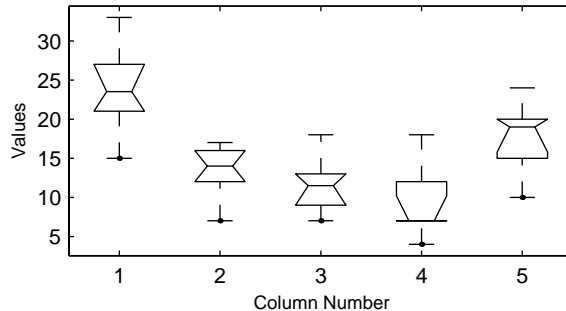
You can use the F statistic to do a hypothesis test to find out if the bacteria counts are the same. `anova1` returns the p-value from this hypothesis test.

In this case the p-value is about 0.0001, a very small value. This is a strong indication that the bacteria counts from the different tankers are not the same. An F statistic as extreme as the observed F would occur by chance only once in 10,000 times if the counts were truly equal.

The p-value returned by `anova1` depends on assumptions about the random disturbances  $\epsilon_{ij}$  in the model equation. For the p-value to be correct, these disturbances need to be independent, normally distributed, and have constant variance. See “Robust and Nonparametric Methods” on page 4-55 for a nonparametric function that does not require a normal assumption.



You can get some graphical assurance that the means are different by looking at the box plots in the second figure window displayed by anova1.



## Multiple Comparisons

Sometimes you need to determine not just whether there are any differences among the means, but specifically which pairs of means are significantly different. It is tempting to perform a series of  $t$  tests, one for each pair of means, but this procedure has a pitfall.

In a  $t$  test, you compute a  $t$  statistic and compare it to a critical value. The critical value is chosen so that when the means are really the same (any apparent difference is due to random chance), the probability that the  $t$  statistic will exceed the critical value is small, say 5%. When the means are different, the probability that the statistic will exceed the critical value is larger.

In this example there are five means, so there are 10 pairs of means to compare. It stands to reason that if all the means are the same, and if there is a 5% chance of incorrectly concluding that there is a difference in one pair, then the probability of making at least one incorrect conclusion among all 10 pairs is much larger than 5%.

Fortunately, there are procedures known as *multiple comparison procedures* that are designed to compensate for multiple tests.

## Example: Multiple Comparisons

You can perform a multiple comparison test using the `multcompare` function and supplying it with the stats output from `anova1`.

```
[c,m] = multcompare(stats)
```

```
c =
```

```

1.0000    2.0000    2.4953    10.5000    18.5047
1.0000    3.0000    4.1619    12.1667    20.1714
1.0000    4.0000    6.6619    14.6667    22.6714
1.0000    5.0000   -2.0047    6.0000    14.0047
2.0000    3.0000   -6.3381    1.6667    9.6714
2.0000    4.0000   -3.8381    4.1667    12.1714
2.0000    5.0000  -12.5047   -4.5000    3.5047
3.0000    4.0000   -5.5047    2.5000    10.5047
3.0000    5.0000  -14.1714   -6.1667    1.8381
4.0000    5.0000  -16.6714   -8.6667   -0.6619

```

```
m =
```

```

23.8333    1.9273
13.3333    1.9273
11.6667    1.9273
9.1667     1.9273
17.8333    1.9273

```

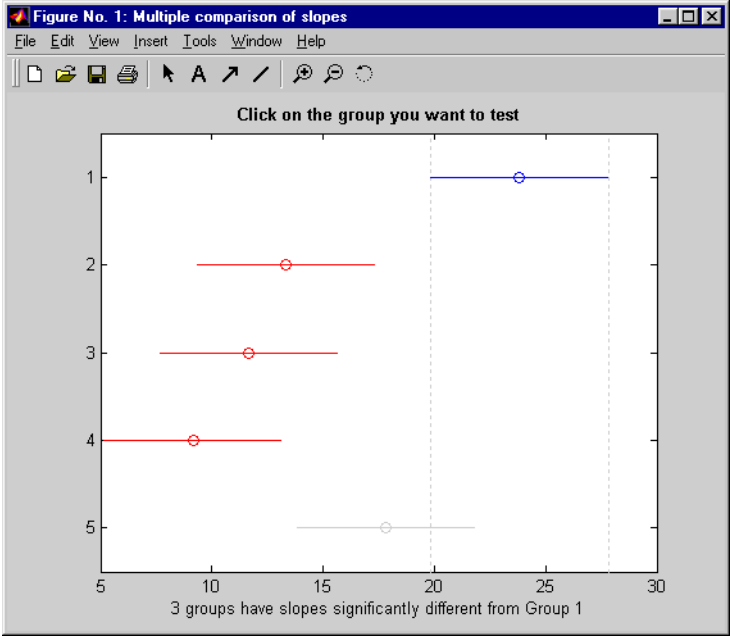
The first output from `multcompare` has one row for each pair of groups, with an estimate of the difference in group means and a confidence interval for that group. For example, the second row has the values

```
1.0000    3.0000    4.1619    12.1667    20.1714
```

indicating that the mean of group 1 minus the mean of group 3 is estimated to be 12.1667, and a 95% confidence interval for this difference is [4.1619, 20.1714]. This interval does not contain 0, so you can conclude that the means of groups 1 and 3 are different.

The second output contains the mean and its standard error for each group.

It is easier to visualize the difference between group means by looking at the graph that `multcompare` produces.



The graph shows that group 1 is significantly different from groups 2, 3, and 4. By using the mouse to select group 4, you can determine that it is also significantly different from group 5. Other pairs are not significantly different.

## Two-Way Analysis of Variance (ANOVA)

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  $k$ ).
- $\mu$  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  $\mu$ ) that are attributable to the car's *model*. All values in a given column of  $\alpha_{.j}$  are identical, and the values in each row of  $\alpha_{.j}$  sum to 0.
- $\beta_{i.}$  is a matrix whose rows are the deviations of each car's gas mileage (from the mean gas mileage  $\mu$ ) 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.

- $\gamma_{ij}$  is a matrix of interactions. The values in each row of  $\gamma_{ij}$  sum to 0, and the values in each column of  $\gamma_{ij}$  sum to 0.
- $\varepsilon_{ijk}$  is a matrix of random disturbances.

The next section provides an example of a two-way analysis.

### 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.

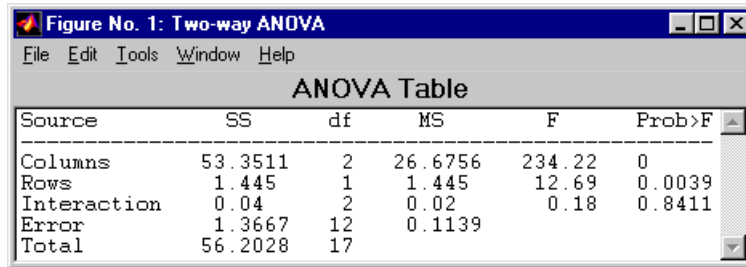
```
load mileage
mileage

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.



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 of 100 times) given that there is no interaction.

The p-values returned by `anova2` depend on assumptions about the random disturbances  $\varepsilon_{ijk}$  in the model equation. For the p-values to be correct these disturbances need to be independent, normally distributed, and have constant variance. See “Robust and Nonparametric Methods” on page 4-55 for nonparametric methods that do not require a normal distribution.

In addition, `anova2` requires that data be *balanced*, which in this case means there must be the same number of cars for each combination of model and factory. The next section discusses a function that supports unbalanced data with any number of predictors.

## N-Way Analysis of Variance

You can use N-way ANOVA to determine if the means in a set of data differ when grouped by multiple factors. If they do differ, you can determine which factors or combinations of factors are associated with the difference.

N-way ANOVA is a generalization of two-way ANOVA. For three factors, the model can be written

$$y_{ijkl} = \mu + \alpha_{.j} + \beta_{i..} + \gamma_{..k} + (\alpha\beta)_{ij.} + (\alpha\gamma)_{i.k} + (\beta\gamma)_{.jk} + (\alpha\beta\gamma)_{ijk} + \varepsilon_{ijkl}$$

In this notation parameters with two subscripts, such as  $(\alpha\beta)_{ij.}$ , represent the interaction effect of two factors. The parameter  $(\alpha\beta\gamma)_{ijk}$  represents the three-way interaction. An ANOVA model can have the full set of parameters or any subset, but conventionally it does not include complex interaction terms unless it also includes all simpler terms for those factors. For example, one would generally not include the three-way interaction without also including all two-way interactions.

The `anovan` function performs N-way ANOVA. Unlike the `anova1` and `anova2` functions, `anovan` does not expect data in a tabular form. Instead, it expects a vector of response measurements and a separate vector (or text array) containing the values corresponding to each factor. This input data format is more convenient than matrices when there are more than two factors or when the number of measurements per factor combination is not constant.

The following examples explore `anovan` in greater detail:

- “Example: N-Way ANOVA with Small Data Set” on page 4-11
- “Example: N-Way ANOVA with Large Data Set” on page 4-13

### Example: N-Way ANOVA with Small Data Set

Consider the following two-way example using `anova2`.

```
m = [23 15 20;27 17 63;43 3 55;41 9 90]
```

```
m =
```

```
    23    15    20
    27    17    63
    43     3    55
    41     9    90
```

```
anova2(m,2)

ans =
    0.0197    0.2234    0.2663
```

The factor information is implied by the shape of the matrix  $m$  and the number of measurements at each factor combination (2). Although `anova2` does not actually require arrays of factor values, for illustrative purposes you could create them as follows.

```
cfactor = repmat(1:3,4,1)

cfactor =
     1     2     3
     1     2     3
     1     2     3
     1     2     3

rfactor = [ones(2,3); 2*ones(2,3)]

rfactor =
     1     1     1
     1     1     1
     2     2     2
     2     2     2
```

The `cfactor` matrix shows that each column of  $m$  represents a different level of the column factor. The `rfactor` matrix shows that the top two rows of  $m$  represent one level of the row factor, and bottom two rows of  $m$  represent a second level of the row factor. In other words, each value  $m(i, j)$  represents an observation at column factor level `cfactor(i, j)` and row factor level `rfactor(i, j)`.

To solve the above problem with `anovan`, you need to reshape the matrices  $m$ , `cfactor`, and `rfactor` to be vectors.

```
m = m(:);
cfactor = cfactor(:);
rfactor = rfactor(:);

[m cfactor rfactor]
```



```

ans =

    23     1     1
    27     1     1
    43     1     2
    41     1     2
    15     2     1
    17     2     1
     3     2     2
     9     2     2
    20     3     1
    63     3     1
    55     3     2
    90     3     2

anovan(m,{cfactor rfactor},2)

ans =

    0.0197
    0.2234
    0.2663

```

## Example: N-Way ANOVA with 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
    
```

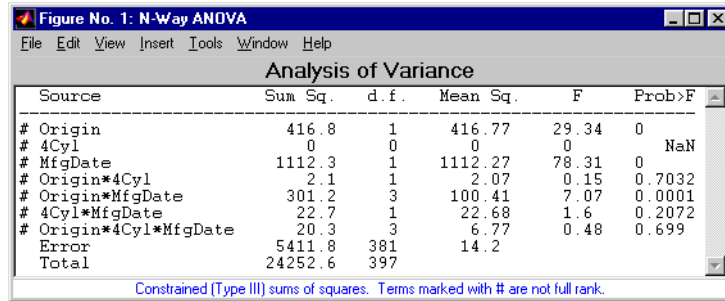
The example focusses on four variables. MPG is the number of miles per gallon for each of 406 cars (though some have missing values coded as NaN). The other three variables are factors: cyl4 (four-cylinder car or not), org (car originated in Europe, Japan, or the USA), and when (car was built early in 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,termvec] = anovan(MPG,{org cyl4 when},2,3,varnames);
termvec
```

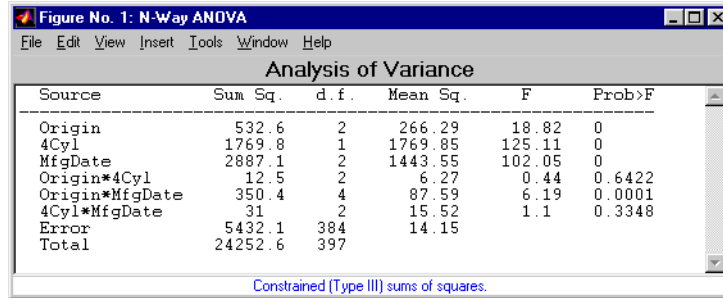
```
termvec =
```

1	0	0
0	1	0
0	0	1

```

1    1    0
1    0    1
0    1    1

```



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 `termvec` variable returns a vector of codes, each of which is a bit pattern representing a term. You can omit terms from the model by deleting their entries from `termvec` and running `anovan` again, this time supplying the resulting vector as the model argument.

```
termvec([4 6],:) = []
```

```
termvec =
```

```

1    0    0
0    1    0
0    0    1
1    0    1

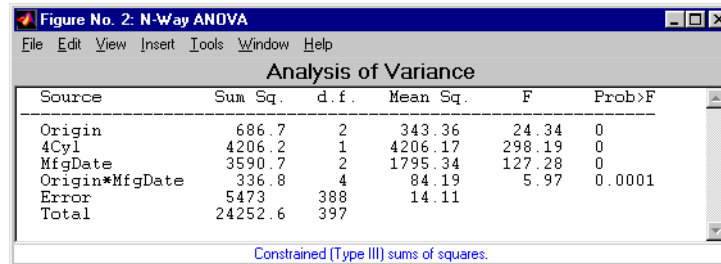
```

```
anovan(MPG,{org cyl4 when},termvec,3,varnames)
```

```
ans =
```

```
1.0e-003 *
```

0.0000  
0  
0  
0.1140



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

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 `anova` to fit models where a factor's levels represent a random selection from a larger (infinite) set of possible levels.

This section covers the following topics:

- “Setting Up the Model” on page 4-18
- “Fitting a Random Effects Model” on page 4-19
- “F Statistics for Models with Random Effects” on page 4-20
- “Variance Components” on page 4-22

### 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 `anova` 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 4-18, 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 4-18, consider the mileage for a particular car of a particular model made at a random factory. The variance of that car is the sum of components, or contributions, one from each of the random terms.

```
stats.rtnames

ans =

    'Factory'
    'Factory*Car Model'
    'Error'
```

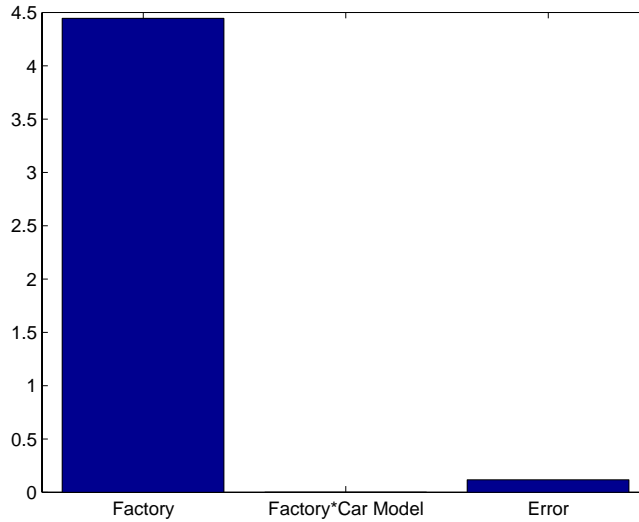
You do not know those variances, but you can estimate them from the data. Recall that the `ems` field of the `stats` structure expresses the expected value of each term's mean square as a linear combination of unknown variances for random terms, and unknown quadratic forms for fixed terms. If you take the

expected mean square expressions for the random terms, and equate those expected values to the computed mean squares, you get a system of equations that you can solve for the unknown variances. These solutions are the variance component estimates. The `varest` field contains a variance component estimate for each term. The `rtnames` field contains the names of the random terms.

```
stats.varest  
  
ans =  
  
    4.4426  
   -0.0313  
    0.1139
```

Under some conditions, the variability attributed to a term is unusually low, and that term's variance component estimate is negative. In those cases it is common to set the estimate to zero, which you might do, for example, to create a bar graph of the components.

```
bar(max(0,stats.varest))  
set(gca,'xtick',1:3,'xticklabel',stats.rtnames)  
bar(max(0,stats.varest))
```



You can also compute confidence bounds for the variance estimate. The `anovan` function does this by computing confidence bounds for the variance expected mean squares, and finding lower and upper limits on each variance component containing all of these bounds. This procedure leads to a set of bounds that is conservative for balanced data. (That is, 95% confidence bounds will have a probability of at least 95% of containing the true variances if the number of observations for each combination of grouping variables is the same.) For unbalanced data, these are approximations that are not guaranteed to be conservative.

```
[{'Term' 'Estimate' 'Lower' 'Upper'};
 stats.rtnames, num2cell([stats.varest stats.varci])]

ans =

    'Term'                'Estimate'    'Lower'      'Upper'
    'Factory'             [ 4.4426]    [1.0736]     [175.6038]
    'Factory*Car Model'   [-0.0313]    [ NaN]       [ NaN]
    'Error'               [ 0.1139]    [0.0586]     [ 0.3103]
```

## Analysis of Covariance

Analysis of covariance is a technique for analyzing grouped data having a response ( $y$ , the variable to be predicted) and a predictor ( $x$ , the variable used to do the prediction). Using analysis of covariance, you can model  $y$  as a linear function of  $x$ , with the coefficients of the line possibly varying from group to group.

### The aocool Demo

The aocool demo is an interactive graphical environment for fitting and prediction with analysis of covariance (anocova) models. It is similar to the polytool demo. The aocool function 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	$y = (\alpha + \alpha_i) + (\beta + \beta_i)x + \varepsilon$

In the parallel lines model, for example, the intercept varies from one group to the next, but the slope is the same for each group. In the same mean model, there is a common intercept and no slope. In order to make the group coefficients well determined, the demo imposes the constraints

$$\sum \alpha_j = \sum \beta_j = 0.$$

The following sections provide an illustrative example.

- “Exploring the aocool Interface” on page 4-26
- “Confidence Bounds” on page 4-29
- “Multiple Comparisons” on page 4-31

## Exploring the aocool Interface

- 1 Load the data.** The Statistics Toolbox has a small data set with information about cars from the years 1970, 1976, and 1982. This example studies the relationship between the weight of a car and its mileage, and whether this relationship has changed over the years. To start the demonstration, load the data set.

```
load carsmall
```

The Workspace browser shows the variables in the data set.

The screenshot shows the MATLAB Workspace browser window. The title bar reads 'Workspace'. The menu bar includes 'File', 'Edit', 'View', 'Web', 'Window', and 'Help'. Below the menu bar are icons for file operations and a 'Stack' dropdown menu set to 'Base'. The main area is a table with the following data:

Name	Size	Bytes	Class
MPG	100x1	800	double array
Model_Year	100x1	800	double array
Weight	100x1	800	double array
Horsepower	100x1	800	double array
Cylinders	100x1	800	double array
Acceleration	100x1	800	double array
Displacement	100x1	800	double array
Model	100x36	7200	char array
Origin	100x7	1400	char array

You can also use aocool with your own data.

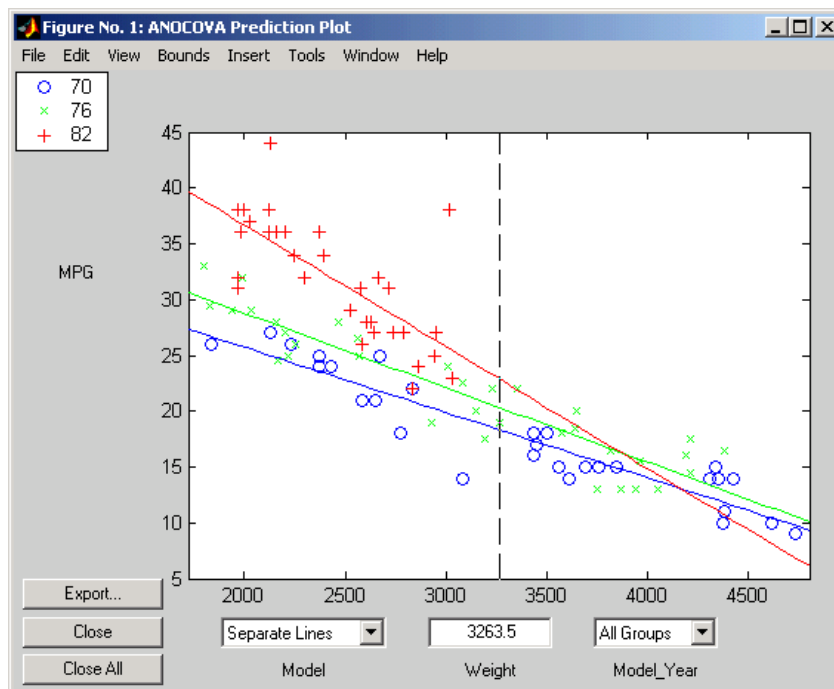
- 2 Start the tool.** The following command calls aocool to fit a separate line to the column vectors Weight and MPG for each of the three model group defined in Model\_Year. The initial fit models the  $y$  variable, MPG, as a linear function of the  $x$  variable, Weight.

```
[h,atab,ctab,stats] = aocool(Weight,MPG,Model_Year);
```

Note: 6 observations with missing values have been removed.

See the aocool function reference page for detailed information about calling aocool.

**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.

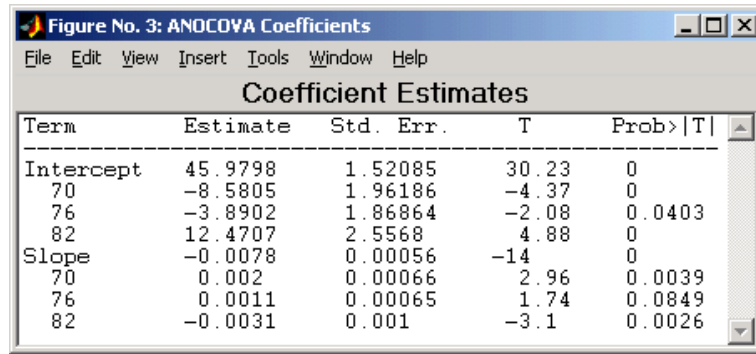


The coefficients of the three lines appear in the figure titled **ANOCOVA Coefficients**. You can see that the slopes are roughly  $-0.0078$ , with a small deviation for each group:

$$\text{Model year 1970: } y = (45.9798 - 8.5805) + (-0.0078 + 0.002)x + \varepsilon$$

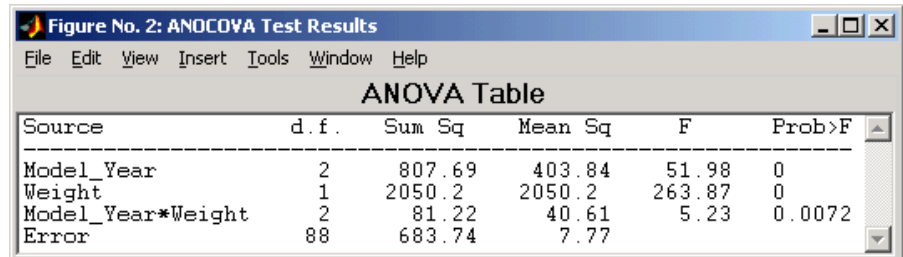
$$\text{Model year 1976: } y = (45.9798 - 3.8902) + (-0.0078 + 0.0011)x + \varepsilon$$

$$\text{Model year 1982: } y = (45.9798 + 12.4707) + (-0.0078 - 0.0031)x + \varepsilon$$



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

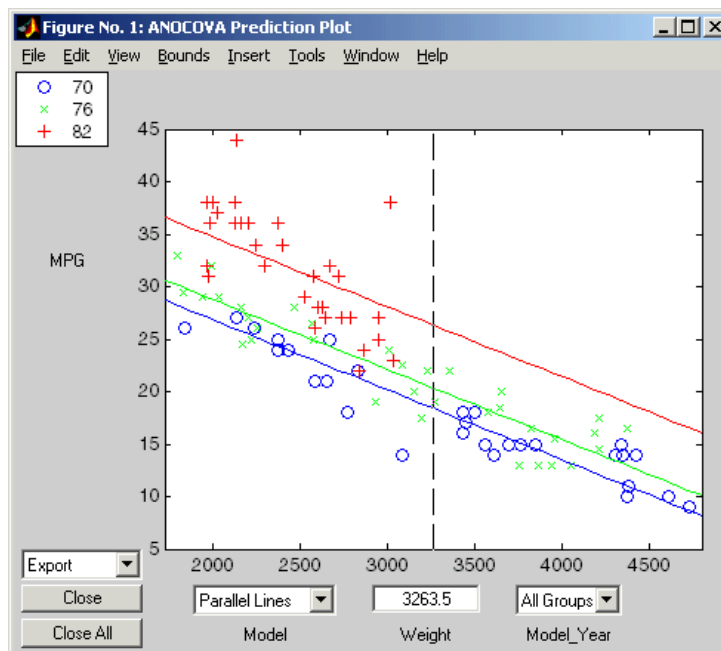
Because the three fitted lines have slopes that are roughly similar, you may wonder if they really are the same. The `Model_Year*Weight` interaction expresses the difference in slopes, and the ANOVA table shows a test for the significance of this term. With an F statistic of 5.23 and a p-value of 0.0072, the slopes are significantly different.



Source	d.f.	Sum Sq	Mean Sq	F	Prob>F
Model_Year	2	807.69	403.84	51.98	0
Weight	1	2050.2	2050.2	263.87	0
Model_Year*Weight	2	81.22	40.61	5.23	0.0072
Error	88	683.74	7.77		

- 4 Constrain the slopes to be the same.** To examine the fits when the slopes are constrained to be the same, return to the **ANCOVA Prediction Plot** window and use the **Model** pop-up menu to select a **Parallel Lines** model. The window updates to show the following graph.



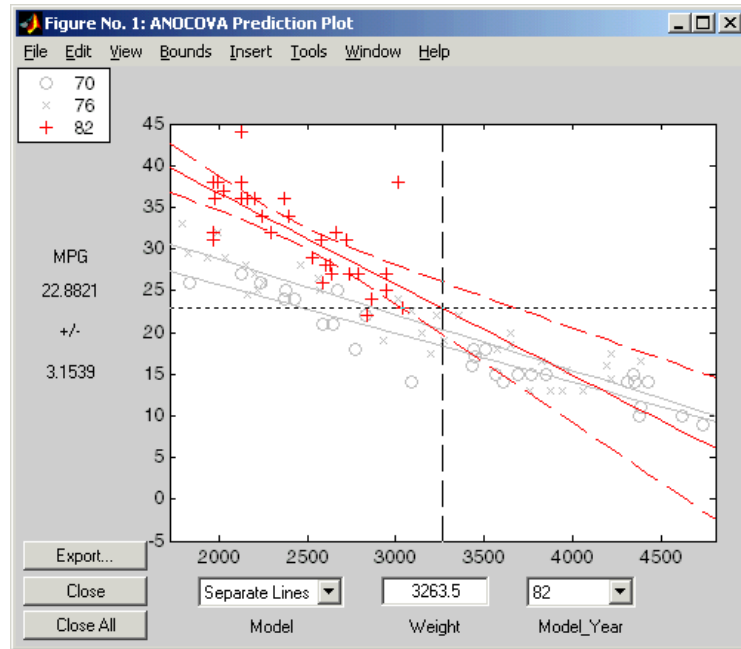


Though this fit looks reasonable, it is significantly worse than the **Separate Lines** model. Use the **Model** pop-up menu again to return to the original model.

### Confidence Bounds

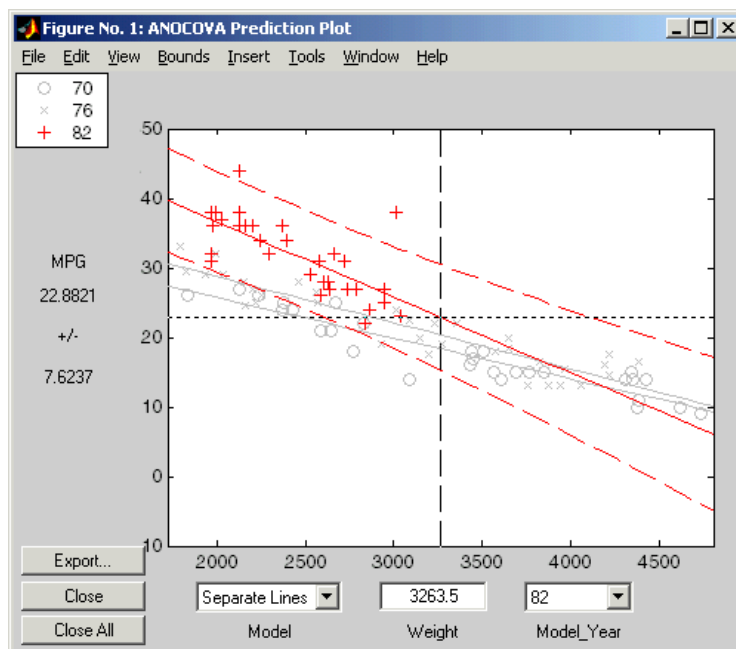
The example in “Exploring the aocool Interface” on page 4-26 provides estimates of the relationship between MPG and Weight for each Model\_Year, but how accurate are these estimates? To find out, you can superimpose confidence bounds on the fits by examining them one group at a time.

- 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 aocool 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 `aocool` function has crosshairs 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.

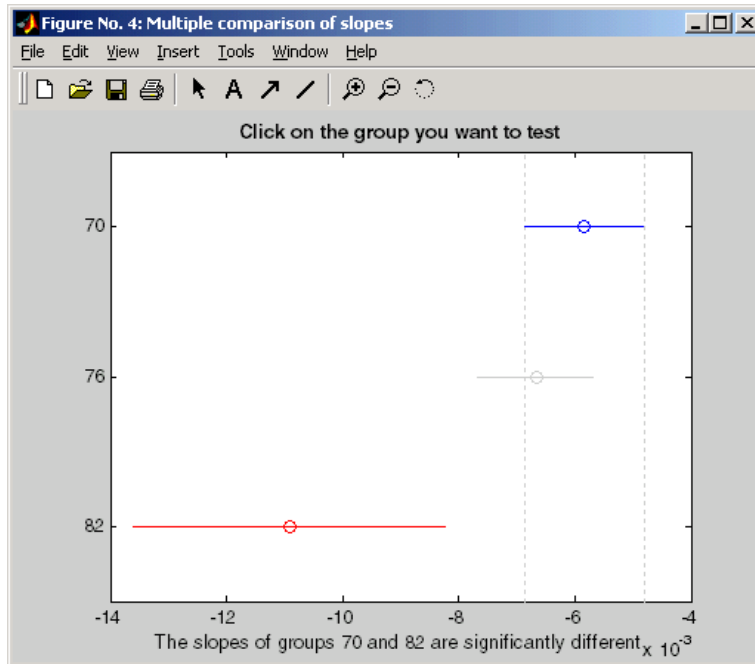
## Multiple Comparisons

You can perform a multiple comparison test by using the stats output structure from `aocool` 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 “Exploring the `aocool` Interface” on page 4-26, 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.

## Multiple Linear Regression

The purpose of multiple linear regression is to establish a quantitative relationship between a group of predictor variables (the columns of  $X$ ) and a response,  $y$ . This relationship is useful for:

- Understanding which predictors have the greatest effect.
- Knowing the direction of the effect (i.e., increasing  $x$  increases/decreases  $y$ ).
- Using the model to predict future values of the response when only the predictors are currently known.

The following sections explain multiple linear regression in greater detail:

- “Mathematical Foundations of Multiple Linear Regression” on page 4-33
- “Example: Multiple Linear Regression” on page 4-35
- “Polynomial Curve Fitting Demo” on page 4-36

### Mathematical Foundations of Multiple Linear Regression

The linear model takes its common form

$$y = X\beta + \varepsilon$$

where:

- $y$  is an  $n$ -by-1 vector of observations.
- $X$  is an  $n$ -by- $p$  matrix of regressors.
- $\beta$  is a  $p$ -by-1 vector of parameters.
- $\varepsilon$  is an  $n$ -by-1 vector of random disturbances.

The solution to the problem is a vector,  $b$ , which estimates the unknown vector of parameters,  $\beta$ . The least squares solution is

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

This equation is useful for developing later statistical formulas, but has poor numeric properties. `regress` uses QR decomposition of  $X$  followed by the backslash operator to compute  $b$ . The QR decomposition is not necessary for computing  $b$ , but the matrix  $R$  is useful for computing confidence intervals.

You can plug  $b$  back into the model formula to get the predicted  $y$  values at the data points.

$$\hat{y} = Xb = Hy$$

$$H = X(X^T X)^{-1} X^T$$

---

**Note** Statisticians use a hat (circumflex) over a letter to denote an estimate of a parameter or a prediction from a model. The projection matrix  $H$  is called the *hat matrix*, because it puts the “hat” on  $y$ .

---

The residuals are the difference between the observed and predicted  $y$  values.

$$r = y - \hat{y} = (I - H)y$$

The residuals are useful for detecting failures in the model assumptions, since they correspond to the errors,  $\epsilon$ , in the model equation. By assumption, these errors each have independent normal distributions with mean zero and a constant variance.

The residuals, however, are correlated and have variances that depend on the locations of the data points. It is a common practice to scale (“Studentize”) the residuals so they all have the same variance.

In the equation below, the scaled residual,  $t_i$ , has a Student’s  $t$  distribution with  $(n-p-1)$  degrees of freedom

$$t_i = \frac{r_i}{\hat{\sigma}_{(i)} \sqrt{1 - h_i}}$$

where

$$\hat{\sigma}_{(i)}^2 = \frac{\|r\|^2}{n - p - 1} - \frac{r_i^2}{(n - p - 1)(1 - h_i)}$$

and:

- $t_i$  is the scaled residual for the  $i$ th data point.
- $r_i$  is the raw residual for the  $i$ th data point.

- $n$  is the sample size.
- $p$  is the number of parameters in the model.
- $h_i$  is the  $i$ th diagonal element of  $H$ .

The left-hand side of the second equation is the estimate of the variance of the errors excluding the  $i$ th data point from the calculation.

A hypothesis test for outliers involves comparing  $t_i$  with the critical values of the  $t$  distribution. If  $t_i$  is large, this casts doubt on the assumption that this residual has the same variance as the others.

A confidence interval for the mean of each error is

$$c_i = r_i \pm t_{\left(1 - \frac{\alpha}{2}, v\right)} \hat{\sigma}_{(i)} \sqrt{1 - h_i}$$

Confidence intervals that do not include zero are equivalent to rejecting the hypothesis (at a significance probability of  $\alpha$ ) that the residual mean is zero. Such confidence intervals are good evidence that the observation is an outlier for the given model.

## Example: Multiple Linear Regression

The example comes from Chatterjee and Hadi (1986) in a paper on regression diagnostics. The data set (originally from Moore (1975)) has five predictor variables and one response.

```
load moore
X = [ones(size(moore,1),1) moore(:,1:5)];
```

Matrix  $X$  has a column of ones, and then one column of values for each of the five predictor variables. The column of ones is necessary for estimating the  $y$ -intercept of the linear model.

```
y = moore(:,6);
[b,bint,r,rint,stats] = regress(y,X);
```

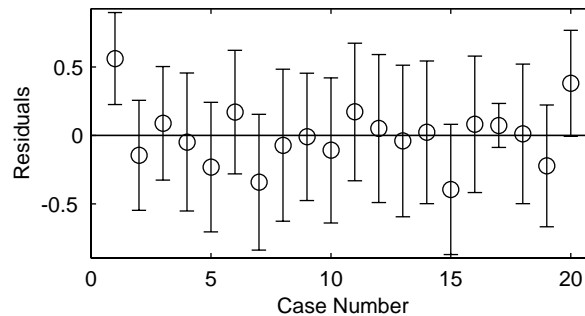
The  $y$ -intercept is  $b(1)$ , which corresponds to the column index of the column of ones.

```
stats
stats =
    0.8107    11.9886    0.0001
```

The elements of the vector `stats` are the regression  $R^2$  statistic, the  $F$  statistic (for the hypothesis test that all the regression coefficients are zero), and the  $p$ -value associated with this  $F$  statistic.

$R^2$  is 0.8107 indicating the model accounts for over 80% of the variability in the observations. The  $F$  statistic of about 12 and its  $p$ -value of 0.0001 indicate that it is highly unlikely that all of the regression coefficients are zero.

```
rcoplot(r,rint)
```



The plot shows the residuals plotted in case order (by row). The 95% confidence intervals about these residuals are plotted as error bars. The first observation is an outlier since its error bar does not cross the zero reference line.

In problems with just a single predictor, it is simpler to use the `polytool` function (see “Polynomial Curve Fitting Demo” on page 4-36). This function can form an  $X$  matrix with predictor values, their squares, their cubes, and so on.

## Polynomial Curve Fitting Demo

The `polytool` demo is an interactive graphic environment for polynomial curve fitting and prediction. You can use `polytool` to do curve fitting and prediction for any set of  $x$ - $y$  data, but, for the sake of demonstration, the Statistics Toolbox provides a data set (`polydata.mat`) to illustrate some basic concepts.



With the `polytool` demo you can

- Plot the data, the fitted polynomial, and global confidence bounds on a new predicted value.
- Change the degree of the polynomial fit.
- Evaluate the polynomial at a specific  $x$ -value, or drag the vertical reference line to evaluate the polynomial at varying  $x$ -values.
- Display the predicted  $y$ -value and its uncertainty at the current  $x$ -value.
- Control the confidence bounds and choose between least squares or robust fitting.
- Export fit results to the workspace.

---

**Note** From the command line, you can call `polytool` and specify the data set, the order of the polynomial, and the confidence intervals, as well as labels to replace **X Values** and **Y Values**. See the `polytool` function reference page for details.

---

The following sections explore the use of `polytool`:

- “Fitting a Polynomial” on page 4-37
- “Confidence Bounds” on page 4-40

### Fitting a Polynomial

**1 Load the data.** Before you start the demonstration, you must first load a data set. This example uses `polydata.mat`. For this data set, the variables `x` and `y` are observations made with error from a cubic polynomial. The variables `x1` and `y1` are data points from the “true” function without error.

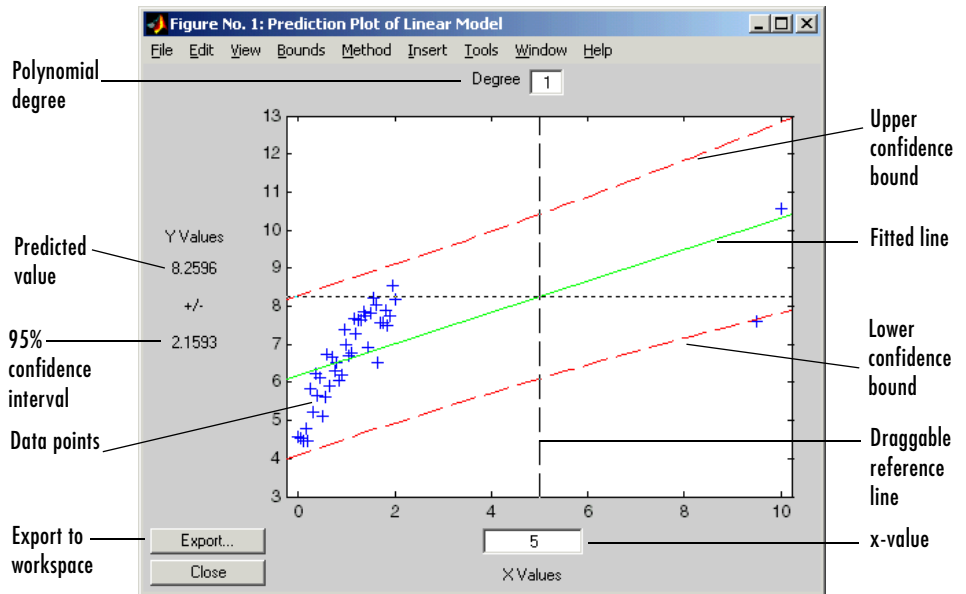
```
load polydata
```

Your variables appear in the Workspace browser.

Name	Size	Bytes	Class
y1	1x101	808	double array
x1	1x101	808	double array
x	1x43	344	double array
y	1x43	344	double array

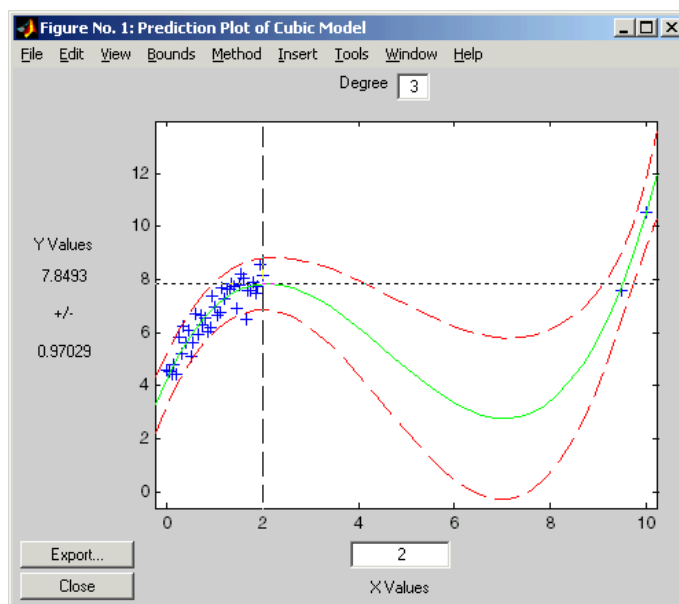
**2 Try a linear fit.** Run `polytool` and provide it with the data to which the polynomial is fit. Because this code does not specify the degree of the polynomial, `polytool` does a linear fit to the data.

```
polytool(x,y)
```



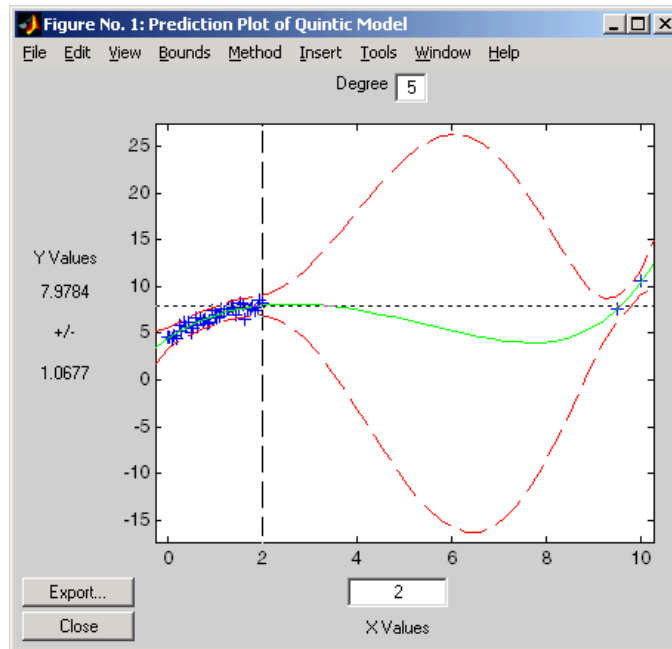
The linear fit is not very good. The bulk of the data with  $x$ -values between 0 and 2 has a steeper slope than the fitted line. The two points to the right are dragging down the estimate of the slope.

- 3 Try a cubic fit.** In the **Degree** text box at the top, type 3 for a cubic model. Then, drag the vertical reference line to the  $x$ -value of 2 (or type 2 in the **X Values** text box).



This graph shows a much better fit to the data. The confidence bounds are closer together indicating that there is less uncertainty in prediction. The data at both ends of the plot track the fitted curve.

- 4 Finally, overfit the data.** If the cubic polynomial is a good fit, it is tempting to try a higher order polynomial to see if even more precise predictions are possible. Since the true function is cubic, this amounts to overfitting the data. Use the data entry box for degree and type 5 for a quintic model.



As measured by the confidence bounds, the fit is precise near the data points. But, in the region between the data groups, the uncertainty of prediction rises dramatically.

This bulge in the confidence bounds happens because the data really does not contain enough information to estimate the higher order polynomial terms precisely, so even interpolation using polynomials can be risky in some cases.

### Confidence Bounds

By default, the confidence bounds are nonsimultaneous bounds for a new observation. What does this mean? Let  $p(x)$  be the true but unknown function you want to estimate. The graph contains the following three curves:

- $f(x)$ , the fitted function
- $l(x)$ , the lower confidence bounds
- $u(x)$ , the upper confidence bounds

Suppose you plan to take a new observation at the value  $x_{n+1}$ . Call it  $y_{n+1}(x_{n+1})$ . This new observation has its own error  $\varepsilon_{n+1}$ , so it satisfies the equation

$$y_{n+1}(x_{n+1}) = p(x_{n+1}) + \varepsilon_{n+1}$$

What are the likely values for this new observation? The confidence bounds provide the answer. The interval  $[l_{n+1}, u_{n+1}]$  is a 95% confidence bound for  $y_{n+1}(x_{n+1})$ .

These are the default bounds, but the **Bounds** menu on the polytool figure window provides options for changing the meaning of these bounds. This menu has options that enable you to specify whether the bounds should be simultaneous or not, and whether the bounds are to apply to the estimated function, i.e., curve, or to a new observation. Using these options you can produce any of the following types of confidence bounds.

<b>Simultaneous?</b>	<b>For Quantity</b>	<b>Yields Confidence Bounds for</b>
Nonsimultaneous	Observation	$y_{n+1}(x_{n+1})$ (default)
Nonsimultaneous	Curve	$p(x_{n+1})$
Simultaneous	Observation	$y_{n+1}(x)$ , globally for any $x$
Simultaneous	Curve	$p(x)$ , simultaneously for all $x$

## Quadratic Response Surface Models

Response Surface Methodology (RSM) is a tool for understanding the quantitative relationship between multiple input variables and one output variable.

Consider one output,  $z$ , as a polynomial function of two inputs,  $x$  and  $y$ . The function  $z = f(x,y)$  describes a two-dimensional surface in the space  $(x,y,z)$ . In general, you can have as many input variables as you want and the resulting surface becomes a hypersurface. Also, you can have multiple output variables with a separate hypersurface for each one.

For three inputs  $(x_1, x_2, x_3)$ , the equation of a quadratic response surface is

$$\begin{aligned}
 y &= b_0 + b_1x_1 + b_2x_2 + b_3x_3 + \dots && \text{(linear terms)} \\
 &+ b_{12}x_1x_2 + b_{13}x_1x_3 + b_{23}x_2x_3 + \dots && \text{(interaction terms)} \\
 &+ b_{11}x_1^2 + b_{22}x_2^2 + b_{33}x_3^2 && \text{(quadratic terms)}
 \end{aligned}$$

It is difficult to visualize a  $k$ -dimensional surface in  $k+1$  dimensional space for  $k>2$ . The function `rstool` is a graphical user interface (GUI) designed to make this visualization more intuitive, as is discussed in the next section.

### Exploring Graphs of Multidimensional Polynomials

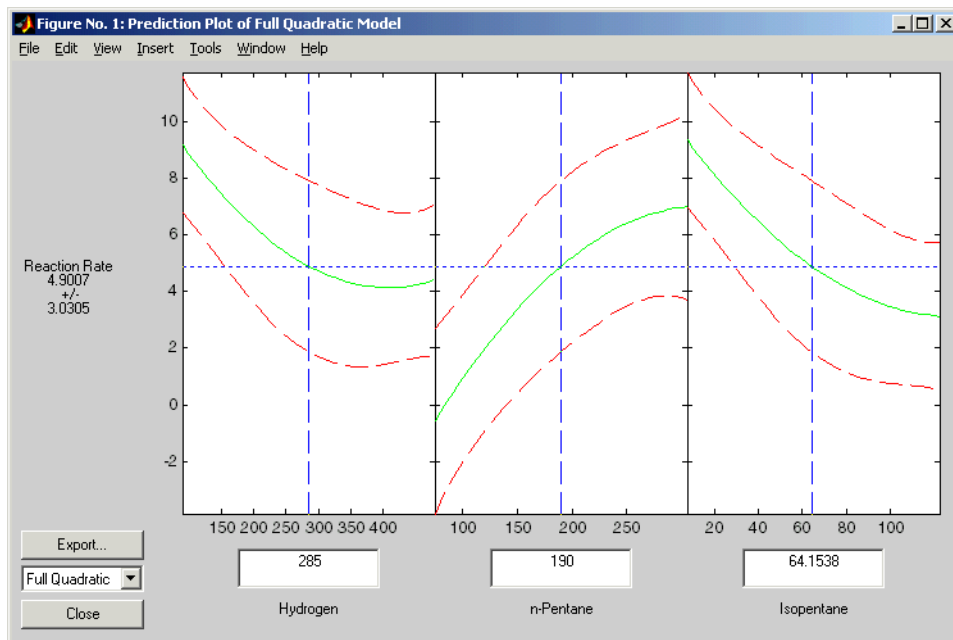
The function `rstool` performs an interactive fit and plot of a multidimensional response surface (RSM). Note that, in general, this GUI provides an environment for exploration of the graph of a multidimensional polynomial.

You can learn about `rstool` by trying the commands below. The chemistry behind the data in `reaction.mat` deals with reaction kinetics as a function of the partial pressure of three chemical reactants: hydrogen, n-pentane, and isopentane.

```
load reaction
rstool(reactants,rate,'quadratic',0.01,xn,yn)
```

`rstool` displays a “vector” of three plots. The dependent variable of all three plots is the reaction rate. The first plot has hydrogen as the independent

variable. The second and third plots have n-pentane and isopentane respectively.



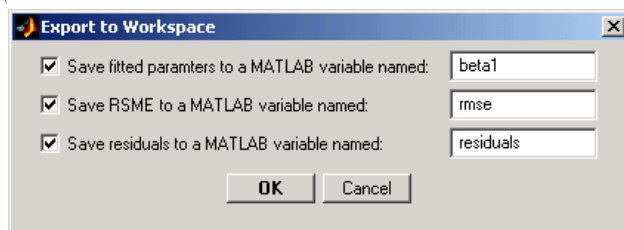
Each plot shows the fitted relationship of the reaction rate to the independent variable at a fixed value of the other two independent variables. The fixed value of each independent variable is in an editable text box below each axis, and is marked by a vertical dashed blue line. You can change the fixed value of any independent variable by either typing a new value in the box or by dragging any of the three vertical lines to a new position.

When you change the value of an independent variable, all the plots update to show the current picture at the new point in the space of the independent variables.

Note that while this example only uses three inputs (reactants) and one output (rate), *rstool* can accommodate an arbitrary number of inputs and outputs. Interpretability may be limited by the size of your monitor for large numbers of inputs or outputs.

## Exporting Variables to the Workspace

Click **Export** to save variables in the GUI to the base workspace.



Fitted parameters, i.e., coefficients, appear in the following order. Some polynomial models use a subset of these terms but keep them in this order.

- 1 Constant term
- 2 Linear terms
- 3 Interaction terms formed by taking pairwise products of the columns of the input matrix
- 4 Squared terms

## Changing the Order of the Polynomial

Below the **Export** button, there is a pop-up menu that enables you to change the polynomial model. If you use the commands above, this menu has the string **Full Quadratic** already selected. The choices are:

- **Linear** – includes constant and linear terms.
- **Pure Quadratic** – includes constant, linear and squared terms.
- **Interactions** – includes constant, linear, and cross product terms.
- **Full Quadratic** – includes interactions and squared terms.
- **User Specified** – available only if you provide a matrix of model terms as the third argument to `rstool`. See the `rstool` and `x2fx` function reference pages for details.)

The `rstool` GUI is used by the `rsmdemo` function to visualize the results of a designed experiment for studying a chemical reaction. See “Design of Experiments Demo” on page 10-10.



## Stepwise Regression

Stepwise regression is a technique for choosing the variables, i.e., terms, to include in a multiple regression model. Forward stepwise regression starts with no model terms. At each step it adds the most statistically significant term (the one with the highest F statistic or lowest p-value) until there are none left. Backward stepwise regression starts with all the terms in the model and removes the least significant terms until all the remaining terms are statistically significant. It is also possible to start with a subset of all the terms and then add significant terms or remove insignificant terms.

An important assumption behind the method is that some input variables in a multiple regression do not have an important explanatory effect on the response. If this assumption is true, then it is a convenient simplification to keep only the statistically significant terms in the model.

One common problem in multiple regression analysis is multicollinearity of the input variables. The input variables may be as correlated with each other as they are with the response. If this is the case, the presence of one input variable in the model may mask the effect of another input. Stepwise regression might include different variables depending on the choice of starting model and inclusion strategy.

The Statistics includes two functions for performing stepwise regression:

- `stepwise` — an interactive graphical tool that enables you to explore stepwise regression. See “Stepwise Regression Demo” on page 4-45 for an example of how to use this tool.
- `stepwisefit` — a command-line tool for performing stepwise regression. You can use `stepwisefit` to return the results of a stepwise regression to the MATLAB workspace.

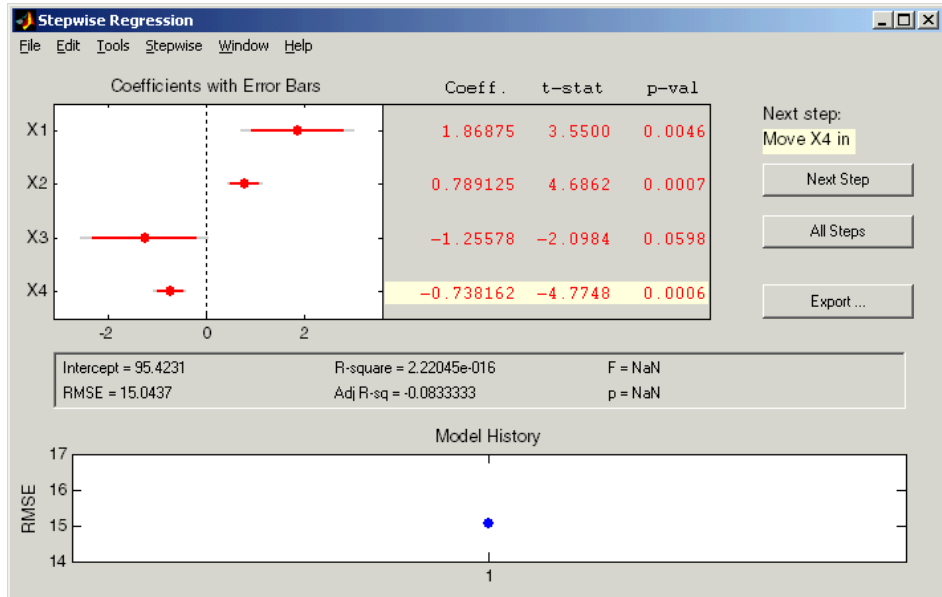
### Stepwise Regression Demo

The `stepwise` function provides an interactive graphical interface that you can use to compare competing models.

This example uses the Hald (1960) data set. The Hald data come from a study of the heat of reaction of various cement mixtures. There are four components in each mixture, and the amount of heat produced depends on the amount of each ingredient in the mixture.

Here are the commands to get started.

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



For each term on the  $y$ -axis, the plot shows the regression (least squares) coefficient as a dot with horizontal bars indicating confidence intervals. Blue dots represent terms that are in the model, while red dots indicate terms that are not currently in the model. The horizontal bars indicate 90% (colored) and 95% (grey) confidence intervals.

To the right of each bar, a table lists the value of the regression coefficient for that term, along with its  $t$ -statistic and  $p$ -value. The coefficient for a term that is not in the model is the coefficient that would result from adding that term to the current model.

From the **Stepwise** menu, select **Scale Inputs** to center and normalize the columns of the input matrix to have a standard deviation of 1.

---

**Note** When you call the stepwise function, you can also specify the initial state of the model and the confidence levels to use. See the stepwise function reference page for details.

---

### Additional Diagnostic Statistics

Several diagnostic statistics appear below the plot.

- Intercept – the estimated value of the constant term
- RMSE – the root mean squared error of the current model
- R-square – the amount of response variability explained by the model
- Adjusted R-square – the R-square statistic adjusted for the residual degrees of freedom
- F – the overall F statistic for the regression
- P – the associated significance probability

### Moving Terms In and Out of the Model

There are two ways you can move terms in and out of the model:

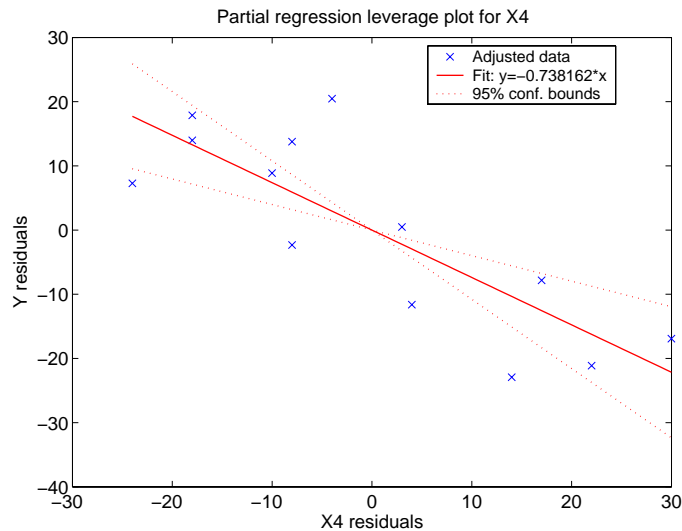
- Click on a line in the plot or in the table to toggle the state of the corresponding term. The resulting change to the model depends on the color of the line:
  - Clicking a blue line, corresponding to a term currently in the model, removes the term from the model and changes the line to red.
  - Clicking a red line, corresponding to a term currently not in the model, adds the term to the model and changes the line to blue.
- Select the recommended step shown under **Next Step** to the right of the table. The recommended step is either to add the most statistically significant term, or to remove the least significant term. Click **Next Step** to perform the recommended step. After you do so, the stepwise GUI displays the next term to add or remove. When there are no more recommended steps, the GUI displays “Move no terms.”

Alternatively, you can perform all the recommended steps at once by clicking **All Steps**.

### Assessing the Effect of Adding a Term

The demo can produce a partial regression leverage plot for the term you choose. If the term is not in the model, the plot shows the effect of adding it by plotting the residuals of the terms that are in the model against the residuals of the chosen term. If the term is in the model, the plot shows the effect of adding it if it were not already in the model. That is, the demo plots the residuals of all *other* terms in the model against the residuals of the chosen term.

From the **Stepwise** menu, select **Added Variable Plot** to display a list of terms. Select the term for which you want a plot, and click **OK**. This example selects X4, the recommended term in the figure above.



### Model History

The Model History plot shows the RMSE for every model generated during the current session. Click one of the dots to return to the model at that point in the analysis.

### **Exporting Variables**

The **Export** pop-up menu enables you to export variables from the stepwise function to the base workspace. Check the variables you want to export and, optionally, change the variable name in the corresponding edit box. Click **OK**.

## Generalized Linear Models

So far, the functions in this section have dealt with models that have a linear relationship between the response and one or more predictors. Sometimes you may have a nonlinear relationship instead. To fit nonlinear models you can use the functions described in “Nonlinear Regression Models” on page 5-1. However, there are some nonlinear models, known as generalized linear models, that you can fit using simpler linear methods. To understand generalized linear models, first review the linear models you have seen so far. Each of these models has the following three characteristics:

- The response has a normal distribution with mean  $\mu$ .
- A coefficient vector  $b$  defines a linear combination  $X*b$  of the predictors  $X$ .
- The model equates the two as  $\mu = X*b$ .

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

- The response has a distribution that may be normal, binomial, Poisson, gamma, or inverse Gaussian, with parameters including a mean  $\mu$ .
- A coefficient vector  $b$  defines a linear combination  $X*b$  of the predictors  $X$ .
- A link function  $f(\cdot)$  defines the link between the two as  $f(\mu) = X*b$ .

The following sections explore these models in greater detail.

- “Example: Generalized Linear Models” on page 4-50
- “Generalized Linear Model Demo” on page 4-54

### Example: Generalized Linear Models

For example, consider the following data derived from the carbig data set, in which the cars have various weights. You record the total number of cars of each weight and the number qualifying as poor-mileage cars because their miles per gallon value is below some target. Assume that you don’t know the miles per gallon for each car, only the number passing the test. It might be reasonable to assume that the value of the variable poor follows a binomial distribution with parameter  $N=\text{total}$  and with a  $p$  parameter that depends on the car weight. A plot shows that the proportion of poor-mileage cars follows a nonlinear S-shape.

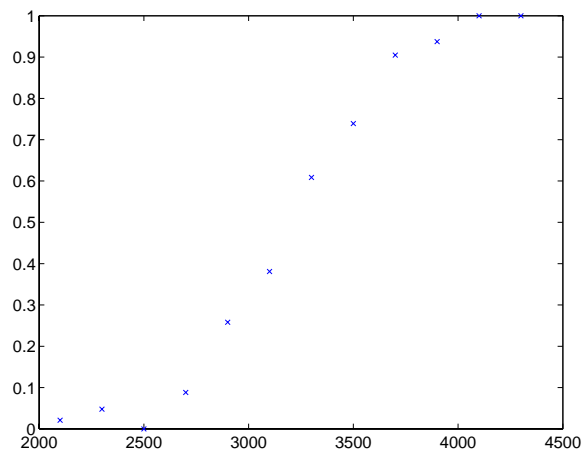
```
w = [2100 2300 2500 2700 2900 3100 3300 3500 3700 3900 4100 4300]';  
poor = [1 2 0 3 8 8 14 17 19 15 17 21]';  
total = [48 42 31 34 31 21 23 23 21 16 17 21]';
```

```
[w poor total]
```

```
ans =
```

2100	1	48
2300	2	42
2500	0	31
2700	3	34
2900	8	31
3100	8	21
3300	14	23
3500	17	23
3700	19	21
3900	15	16
4100	17	17
4300	21	21

```
plot(w,poor./total,'x')
```



This shape is typical of graphs of proportions, as they have natural boundaries at 0.0 and 1.0.

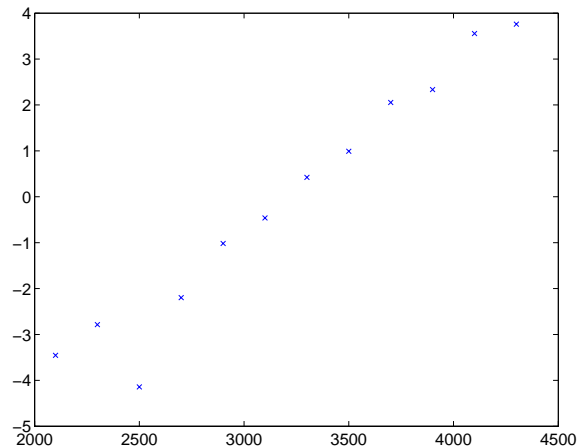
A linear regression model would not produce a satisfactory fit to this graph. Not only would the fitted line not follow the data points, it would produce invalid proportions less than 0 for light cars, and higher than 1 for heavy cars.

There is a class of regression models for dealing with proportion data. The logistic model is one such model. It defines the relationship between proportion  $p$  and weight  $w$  to be

$$\log\left(\frac{p}{1-p}\right) = b_1 + b_2 w$$

Is this a good model for the data? It would be helpful to graph the data on this scale, to see if the relationship appears linear. However, some of the proportions are 0 and 1, so you cannot explicitly evaluate the left-hand-side of the equation. A useful trick is to compute adjusted proportions by adding small increments to the poor and total values — say a half observation to poor and a full observation to total. This keeps the proportions within range. A graph now shows a more nearly linear relationship.

```
padj = (poor+.5) ./ (total+1);
plot(w, log(padj ./ (1-padj)), 'x')
```





You can use the `glmfit` function to fit this logistic model.

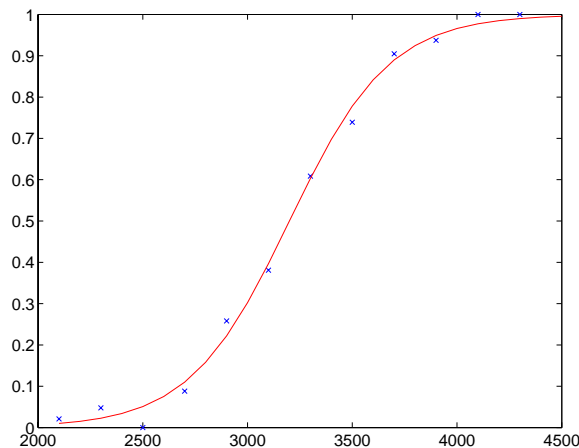
```
b = glmfit(w,[poor total],'binomial')
b =
  -13.3801
   0.0042
```

To use these coefficients to compute a fitted proportion, you have to invert the logistic relationship. Some simple algebra shows that the logistic equation can also be written as

$$p = \frac{1}{1 + \exp(-b_1 - b_2 w)}$$

Fortunately, the function `glmval` can decode this link function to compute the fitted values. Using this function, you can graph fitted proportions for a range of car weights, and superimpose this curve on the original scatter plot.

```
x = 2100:100:4500;
y = glmval(b,x,'logit');
plot(w,poor./total,'x',x,y,'r-')
```



Generalized linear models can fit a variety of distributions with a variety of relationships between the distribution parameters and the predictors. A full description is beyond the scope of this document. For more information see

Dobson (1990), or McCullagh and Nelder (1990). Also see the reference material for `glmfit`.

### **Generalized Linear Model Demo**

The `glimdemo` function begins a slide show describing generalized linear models. It presents examples of what functions and distributions are available with generalized linear models. It then presents an example where traditional linear least squares fitting is not appropriate, and shows how to use the `glmfit` function to fit a logistic regression model and the `glmval` function to compute predictions from that model. (See the `glmfit` and `glmval` function reference pages for details.)

To run `glimdemo` from the command line, type `playshow glimdemo`.

## Robust and Nonparametric Methods

As mentioned in the previous sections, regression and analysis of variance procedures depend on certain assumptions, such as a normal distribution for the error term. Sometimes such an assumption is not warranted. For example, if the distribution of the errors is asymmetric or prone to extreme outliers, that is a violation of the assumption of normal errors.

The Statistics Toolbox has a robust regression function that is useful when there may be outliers. Robust methods are designed to be relatively insensitive to large changes in a small part of the data.

The Statistics Toolbox also has nonparametric versions of the one-way and two-way analysis of variance functions. Unlike classical tests, nonparametric tests make only mild assumptions about the data, and are appropriate when the distribution of the data is not normal. On the other hand, they are less powerful than classical methods for normally distributed data.

The following sections describe the robust regression and nonparametric functions in greater detail:

- “Robust Regression” on page 4-55
- “Kruskal-Wallis Test” on page 4-59
- “Friedman’s Test” on page 4-60

Both of the nonparametric functions described here can return a `stats` structure that you can use as input to the `multcompare` function to perform multiple comparisons.

---

**Note** See “Regression and Classification Trees” on page 5-8 for information on another type of nonparametric regression.

---

### Robust Regression

“Example: Multiple Linear Regression” on page 4-35 shows that there is an outlier when you use ordinary least squares regression to model a response as a function of five predictors. How does that outlier affect the results?

To answer this question, estimate the coefficients using the `robustfit` function.

```
load moore
x = moore(:,1:5);
y = moore(:,6);
[br,statsr] = robustfit(x,y);
br
br =
    -1.7742
     0.0000
     0.0009
     0.0002
     0.0062
     0.0001
```

Compare these estimates to those you obtain from the `regress` function.

```
b
b =
    -2.1561
    -0.0000
     0.0013
     0.0001
     0.0079
     0.0001
```

To understand why the two differ, it is helpful to look at the weight variable from the robust fit. It measures how much weight was given to each point during the fit. In this case, the first point had a very low weight so it was effectively ignored.

```
statsr.w'
ans =
Columns 1 through 7
    0.0577    0.9977    0.9776    0.9455    0.9687    0.8734    0.9177
Columns 8 through 14
    0.9990    0.9653    0.9679    0.9768    0.9882    0.9998    0.9979
Columns 15 through 20
    0.8185    0.9757    0.9875    0.9991    0.9021    0.6953
```

## Robust Fitting Demo

The `robustdemo` function presents a simple comparison of least squares and robust fits for a response and a single predictor. You can use data provided by the demo or supply your own. See the `robustdemo` function reference page for information about using your own data:

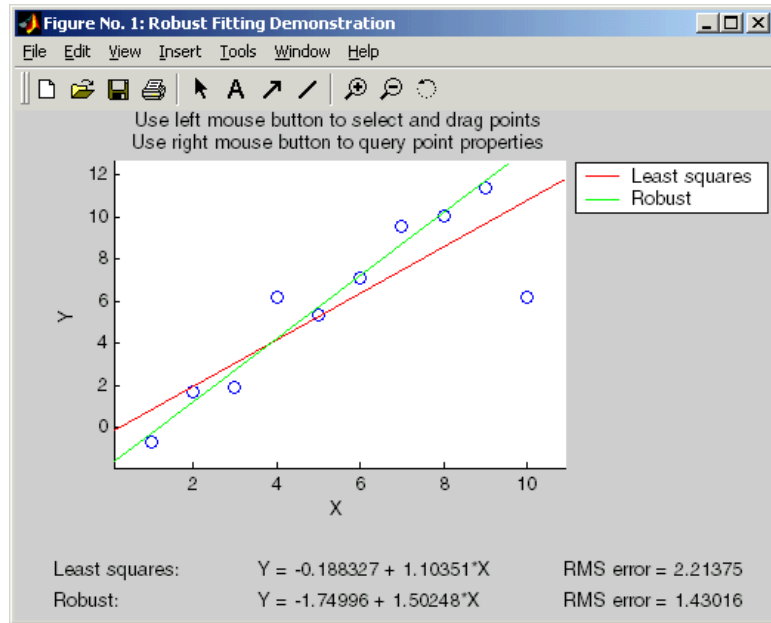
- 1 **Start the demo.** To begin using `robustdemo` with the built-in sample data, simply type the function name.

```
robustdemo
```

The resulting figure presents a scatter plot with two fitted lines. One line is the fit from an ordinary least squares regression. The other is from a robust regression. Along the bottom of the figure are the equations for the fitted line and the estimated error standard deviation for each fit.

The effect of any point on the least squares fit depends on the residual and leverage for that point. The residual is the vertical distance from the point to the line. The leverage is a measure of how far the point is from the center of the  $x$  data.

The effect of any point on the robust fit also depends on the weight assigned to the point. Points far from the line get lower weight.

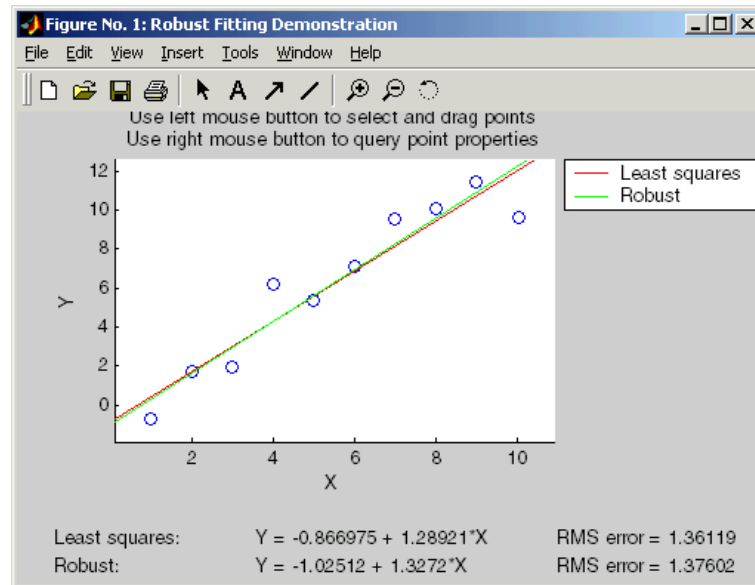


**2 Compare effects of leverage and weight.** Use the right mouse button to click on any point and see its least squares leverage and robust weight.

In this example, the rightmost point has a leverage value of 0.35. It is also far from the line, so it exerts a large influence on the least squares fit. It has a small weight, though, so it is effectively excluded from the robust fit.

**3 See how changes in data affect the two fits.** Using the left mouse button, select any point, and drag it to a new location while holding the left button down. When you release the point, both fits update.

Bringing the rightmost point closer to the line makes the two fitted lines nearly identical. Now, the point has nearly full weight in the robust fit.



## Kruskal-Wallis Test

The example “One-Way Analysis of Variance (ANOVA)” on page 4-3 uses one-way analysis of variance to determine if the bacteria counts of milk varied from shipment to shipment. The one-way analysis rests on the assumption that the measurements are independent, and that each has a normal distribution with a common variance and with a mean that was constant in each column. You can conclude that the column means were not all the same. The following example repeats that analysis using a nonparametric procedure.

The Kruskal-Wallis test is a nonparametric version of one-way analysis of variance. The assumption behind this test is that the measurements come from a continuous distribution, but not necessarily a normal distribution. The test is based on an analysis of variance using the ranks of the data values, not the data values themselves. Output includes a table similar to an anova table, and a box plot.

You can run this test as follows.

```
p = kruskalwallis(hogg)
p =
    0.0020
```

The low p-value means the Kruskal-Wallis test results agree with the one-way analysis of variance results.

## **Friedman's Test**

The example “Two-Way Analysis of Variance (ANOVA)” on page 4-8 uses two-way analysis of variance to study the effect of car model and factory on car mileage. The example tests whether either of these factors has a significant effect on mileage, and whether there is an interaction between these factors. The conclusion of the example is there is no interaction, but that each individual factor has a significant effect. The next example examines whether a nonparametric analysis leads to the same conclusion.

Friedman's test is a nonparametric test for data having a two-way layout (data grouped by two categorical factors). Unlike two-way analysis of variance, Friedman's test does not treat the two factors symmetrically and it does not test for an interaction between them. Instead, it is a test for whether the columns are different after adjusting for possible row differences. The test is based on an analysis of variance using the ranks of the data across categories of the row factor. Output includes a table similar to an anova table.

You can run Friedman's test as follows.

```
p = friedman(mileage, 3)
ans =
    7.4659e-004
```

Recall the classical analysis of variance gave a p-value to test column effects, row effects, and interaction effects. This p-value is for column effects. Using either this p-value or the p-value from ANOVA ( $p < 0.0001$ ), you conclude that there are significant column effects.



In order to test for row effects, you need to rearrange the data to swap the roles of the rows in columns. For a data matrix  $x$  with no replications, you could simply transpose the data and type

```
p = friedman(x')
```

With replicated data it is slightly more complicated. A simple way is to transform the matrix into a three-dimensional array with the first dimension representing the replicates, swapping the other two dimensions, and restoring the two-dimensional shape.

```
x = reshape(mileage, [3 2 3]);
x = permute(x, [1 3 2]);
x = reshape(x, [9 2])
x =
  33.3000    32.6000
  33.4000    32.5000
  32.9000    33.0000
  34.5000    33.4000
  34.8000    33.7000
  33.8000    33.9000
  37.4000    36.6000
  36.8000    37.0000
  37.6000    36.7000
```

```
friedman(x, 3)
```

```
ans =
```

```
0.0082
```

Again, the conclusion is similar to that of the classical analysis of variance. Both this p-value and the one from ANOVA ( $p = 0.0039$ ) lead you to conclude that there are significant row effects.

You cannot use Friedman's test to test for interactions between the row and column factors.



# Nonlinear Regression Models

---

Nonlinear Least Squares (p. 5-2)

Fits a model that has a known parametric form but unknown parameter values.

Regression and Classification Trees (p. 5-8)

Approximates a regression relationship using a decision tree. Such a tree partitions the data set into regions, using values of the predictor variables, so that the response variables are roughly constant in each region.

## Nonlinear Least Squares

Response Surface Methodology (RSM) is an empirical modeling approach using polynomials as local approximations to the true input/output relationship. This empirical approach is often adequate for process improvement in an industrial setting.

In scientific applications there is usually relevant theory for constructing a mechanistic model. Often such models are nonlinear in the unknown parameters. Nonlinear models are more difficult to fit, requiring iterative methods that start with an initial guess of the unknown parameters. Each iteration alters the current guess until the algorithm converges.

The Statistics Toolbox has functions for fitting nonlinear models of the form

$$y = f(X, \beta) + \varepsilon$$

where:

- $y$  is an  $n$ -by-1 vector of observations.
- $f$  is any function of  $X$  and  $\beta$ .
- $X$  is an  $n$ -by- $p$  matrix of input variables.
- $\beta$  is a  $p$ -by-1 vector of unknown parameters to be estimated.
- $\varepsilon$  is an  $n$ -by-1 vector of random disturbances.

This is explored further in the following sections:

- “Example: Nonlinear Modeling” on page 5-2
- “An Interactive GUI for Nonlinear Fitting and Prediction” on page 5-6

### Example: Nonlinear Modeling

The Hougen-Watson model (Bates and Watts, [2]) for reaction kinetics is one specific example of this type. The form of the model is

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

where  $\beta_1, \beta_2, \dots, \beta_5$  are the unknown parameters, and  $x_1, x_2,$  and  $x_3$  are the three input variables. The three inputs are hydrogen, n-pentane, and

isopentane. It is easy to see that the parameters do not enter the model linearly.

The file `reaction.mat` contains simulated data from this reaction.

```
load reaction
who
Your variables are:

beta          rate          xn
model         reactants    yn
```

The variables are as follows:

- `rate` is a 13-by-1 vector of observed reaction rates.
- `reactants` is a 13-by-3 matrix of reactants.
- `beta` is 5-by-1 vector of initial parameter estimates.
- `model` is a string containing the nonlinear function name.
- `xn` is a string matrix of the names of the reactants.
- `yn` is a string containing the name of the response.

The data and model are explored further in the following sections:

- “Fitting the Hougen-Watson Model” on page 5-3
- “Confidence Intervals on the Parameter Estimates” on page 5-5
- “Confidence Intervals on the Predicted Responses” on page 5-5
- “An Interactive GUI for Nonlinear Fitting and Prediction” on page 5-6

### Fitting the Hougen-Watson Model

The Statistics Toolbox provides the function `nlinfit` for finding parameter estimates in nonlinear modeling. `nlinfit` returns the least squares parameter estimates. That is, it finds the parameters that minimize the sum of the squared differences between the observed responses and their fitted values. It uses the Gauss-Newton algorithm with Levenberg-Marquardt modifications for global convergence.

`nlinfit` requires the input data, the responses, and an initial guess of the unknown parameters. You must also supply the name of a function that takes the input data and the current parameter estimate and returns the predicted responses. In MATLAB terminology, `nlinfit` is called a “function” function.

Here is the hougen function.

```
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);
```

To fit the reaction data, call the function `nlinfit`.

```
load reaction
betahat = nlinfit(reactants,rate,'hougen',beta)

betahat =

    1.2526
    0.0628
    0.0400
    0.1124
    1.1914
```

`nlinfit` has two optional outputs. They are the residuals and Jacobian matrix at the solution. The residuals are the differences between the observed and fitted responses. The Jacobian matrix is the direct analog of the matrix  $X$  in the standard linear regression model.

These outputs are useful for obtaining confidence intervals on the parameter estimates and predicted responses.

### Confidence Intervals on the Parameter Estimates

Using `nlparci`, form 95% confidence intervals on the parameter estimates, `betahat`, from the reaction kinetics example.

```
[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
```

### Confidence Intervals on the Predicted Responses

Using `nlpredci`, form 95% confidence intervals on the predicted responses from the reaction kinetics example.

```
[yhat,delta] = nlpredci('hougen',reactants,betahat,resid,J);  
opd = [rate yhat delta]
```

```
opd =
```

```
8.5500    8.2937    0.9178  
3.7900    3.8584    0.7244  
4.8200    4.7950    0.8267  
0.0200   -0.0725    0.4775  
2.7500    2.5687    0.4987  
14.3900   14.2227    0.9666  
2.5400    2.4393    0.9247  
4.3500    3.9360    0.7327  
13.0000   12.9440    0.7210  
8.5000    8.2670    0.9459  
0.0500   -0.1437    0.9537  
11.3200   11.3484    0.9228  
3.1300    3.3145    0.8418
```

Matrix `opd` has the observed rates in column 1 and the predictions in column 2. The 95% confidence interval is column 2±column 3. These are simultaneous confidence intervals for the estimated function at each input value. They are not intervals for new response observations at those inputs, even though most of the confidence intervals do contain the original observations.

### **An Interactive GUI for Nonlinear Fitting and Prediction**

The function `nlintool` for nonlinear models is a direct analog of `rstool` for polynomial models. `nlintool` calls `nlinfit` and requires the same inputs.

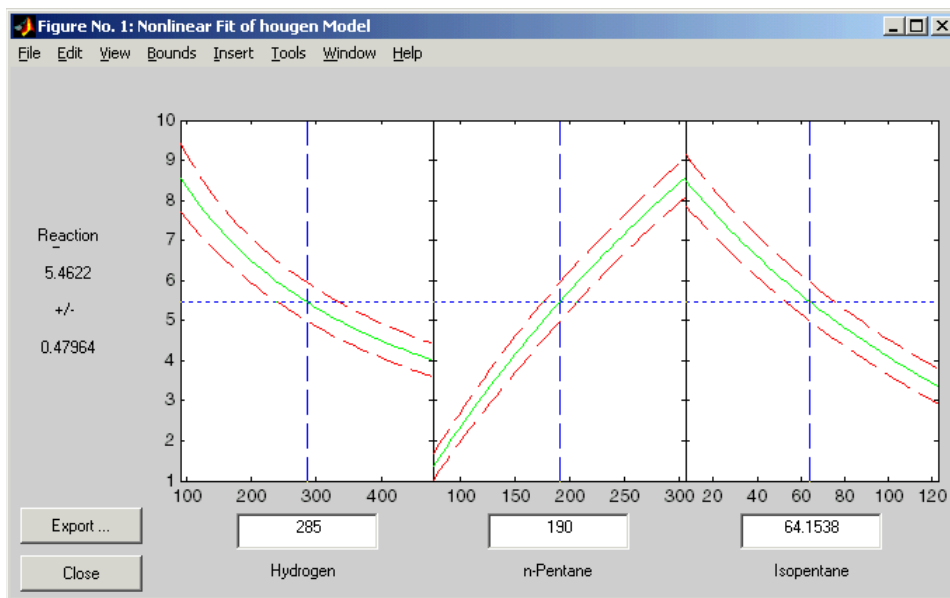
The purpose of `nlintool` is larger than just fitting and prediction for nonlinear models. This GUI provides an environment for exploring the graph of a multidimensional nonlinear function.

Start by loading the data set `reaction.mat` and starting `nlintool`. The chemistry behind the data in `reaction.mat` deals with reaction kinetics as a function of the partial pressure of three chemical reactants: hydrogen, n-pentane, and isopentane.

```
load reaction
nlintool(reactants,rate,'hougen',beta,0.01,xn,yn)
```

You will see a “vector” of three plots. The dependent variable of all three plots is the reaction rate. The first plot has hydrogen as the independent variable. The second and third plots have n-pentane and isopentane respectively.





Each plot shows the fitted relationship of the reaction rate to the independent variable at a fixed value of the other two independent variables. The fixed value of each independent variable is in an editable text box below each axis. You can change the fixed value of any independent variable by either typing a new value in the box or by dragging any of the three vertical lines to a new position.

When you change the value of an independent variable, all the plots update to show the current picture at the new point in the space of the independent variables.

Note that while this example only uses three reactants, `nlintool` can accommodate an arbitrary number of independent variables. Interpretability may be limited by the size of your monitor for large numbers of inputs.

## Regression and Classification Trees

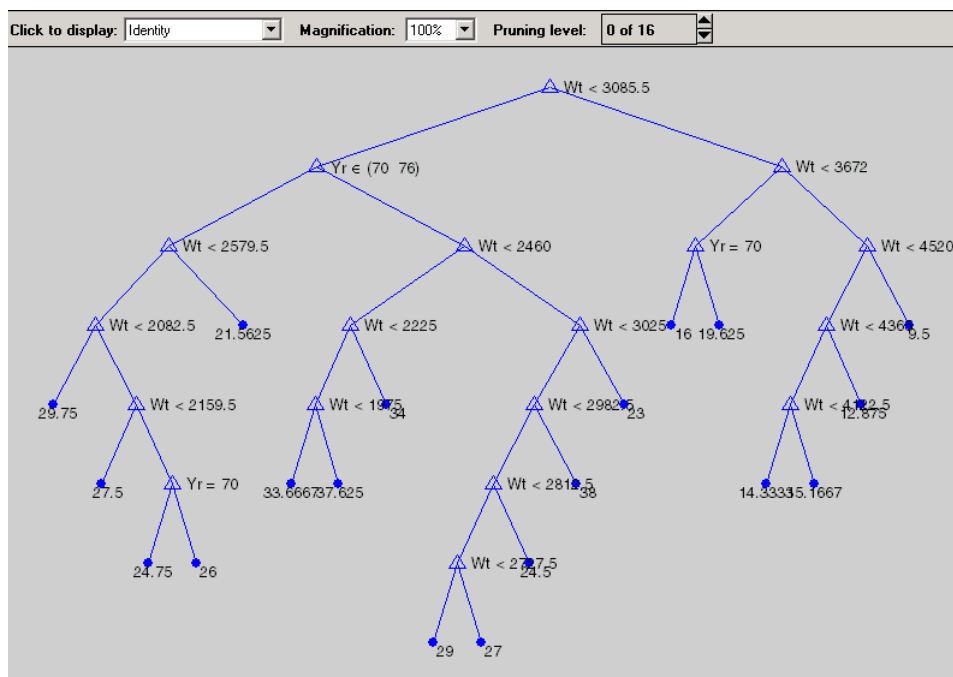
In nonlinear least squares you suppose that you know the form of the relationship between the response and predictor. Suppose instead that you do not know that relationship, and also that you are unwilling to assume that the relationship can be well approximated by a linear model. You need a more nonparametric type of regression fitting approach. One such approach is based on “trees.”

A regression tree is a sequence of questions that can be answered as yes or no, plus a set of fitted response values. Each question asks whether a predictor satisfies a given condition. Predictors can be continuous or discrete. Depending on the answers to one question, you either proceed to another question or arrive at a fitted response value.

This example fits a regression tree to variables from the `carsmall` data set. The example uses the same variables as in the Analysis of Covariance example (see “The `aoctool` Demo” on page 4-25), so there is one continuous predictor (car weight) and one discrete predictor (model year).

The object of the example is to model mileage (MPG) as a function of car weight and model year. First load the data and create a matrix `x` of predictor values and a vector `y` of response variables. Then fit a regression tree, specifying the model year column as a categorical variable. In this data set there are cars from the three different model years 1970, 1976, and 1982.

```
load carsmall
x = [Weight,Model_Year];
y = MPG;
t = treefit(x,y,'catid',2);
treedisp(t,'name',{'Wt' 'Yr'});
```



Now you want to use this model to determine the predicted mileage for a car weighing 3000 pounds from model year 1982. Start at the top node. The weight is less than the cutoff value of 3085.5, so you take the left branch. The model year is not 1970 or 1976, so you take the right branch. Continue moving down the tree until you arrive at a terminal node that gives the predicted value. In this case, the predicted value is 38 miles per gallon. You can use the `treeval` function to find the fitted value for any set of predictor values.

```
treeval(t,[3000 82])
```

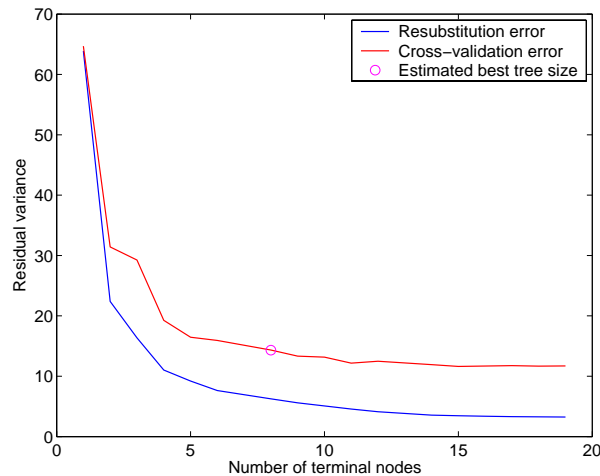
```
ans =  
    38
```

With a tree like this one, having many branches, there is a danger that it fits the current data set well but would not do a good job at predicting new values. Some of its lower branches might be strongly affected by outliers and other artifacts of the current data set. If possible you would prefer to find a simpler tree that avoids this problem of overfitting.

You can estimate the best tree size by cross validation. First, compute a *resubstitution* estimate of the error variance for this tree and a sequence of simpler trees and plot it as the lower (blue) line in the figure. This estimate probably under-estimates the true error variance. Then compute a *cross-validation* estimate of the same quantity and plot it as the upper (red) line. The cross-validation procedure also provides an estimate of the pruning level, *best*, needed to achieve the best tree size.

```
[c,s,ntn] = treetest(t,'resub');
[c2,s2,n2,best] = treetest(t,'cross',x,y);
plot(ntn,c,'b-', n2,c2,'r-', n2(best+1),c2(best+1),'mo');
xlabel('Number of terminal nodes')
ylabel('Residual variance')
legend('Resubstitution error','Cross-validation
error','Estimated best tree size')
best
```

```
best =
    10
```



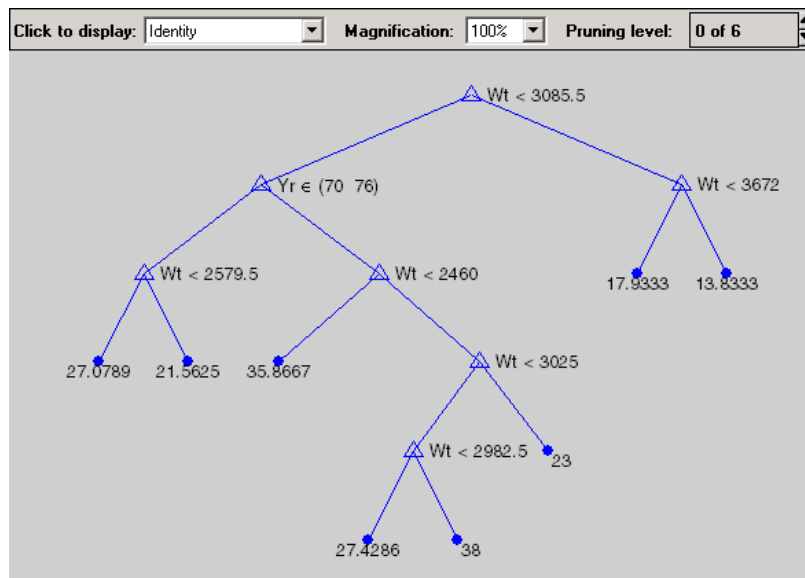
The best tree is the one that has a residual variance that is no more than one standard error above the minimum value along the cross-validation line. In this case the variance is just over 14. The output *best* takes on values starting

with 0 (representing no pruning), so you need to add 1 to use it as an index into the other output arguments.

```
c2(best+1)
ans =
    14.3440
```

Use the output `best` to create a smaller tree that is pruned to the estimated best size.

```
t0 = treeprune(t, 'level', best);
treedisp(t0, 'name', {'Wt' 'Yr'})
```

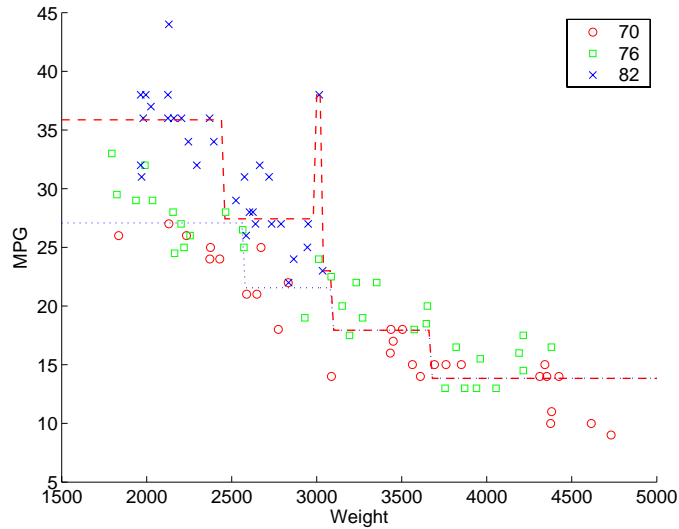


Now plot the original data and overlay the fitted values that you get using this tree. Notice that this tree does not distinguish between cars from 1970 or 1976, so create a vector `yo1d` containing fitted values for 1976 and another `ynew` for year 1982. Cars from 1970 have the same fitted values as those from 1976.

```

xx = (1500:20:5000)';
ynew = treeval(t0,[xx 82*ones(size(xx))]);
yold = treeval(t0,[xx 76*ones(size(xx))]);
gscatter(Weight,MPG,Model_Year,'rgb','osx');
hold on; plot(xx,yold,'b:',xx,ynew,'r--'); hold off

```



The tree functions (`treedisp`, `treefit`, `treeprune`, `treetest`, and `treeval`) can also accept a categorical response variable. In that case, the fitted value from the tree is the category with the highest predicted probability for the range of predictor values falling in a given node. The demo Classification, in the Multivariate Analysis section of the Statistics Toolbox demos, shows how to use decision trees for classification.

# Hypothesis Tests

---

Introduction (p. 6-2)	Introduces the concept of hypothesis tests.
Hypothesis Test Terminology (p. 6-3)	Explains some basic terminology of hypothesis tests.
Hypothesis Test Assumptions (p. 6-4)	Explains some assumptions underlying hypothesis tests.
Example: Hypothesis Testing (p. 6-5)	Provides an example of hypothesis testing.
Available Hypothesis Tests (p. 6-9)	Describes the available hypothesis tests.

## Introduction

A hypothesis test is a procedure for determining if an assertion about a characteristic of a population is reasonable.

For example, suppose that someone says that the average price of a gallon of regular unleaded gas in Massachusetts is \$1.15. How would you decide whether this statement is true? You could try to find out what every gas station in the state was charging and how many gallons they were selling at that price. That approach might be definitive, but it could end up costing more than the information is worth.

A simpler approach is to find out the price of gas at a small number of randomly chosen stations around the state and compare the average price to \$1.15.

Of course, the average price you get will probably not be exactly \$1.15 due to variability in price from one station to the next. Suppose your average price was \$1.18. Is this three cent difference a result of chance variability, or is the original assertion incorrect? A hypothesis test can provide an answer.

The following sections provide an overview of hypothesis testing with the Statistics Toolbox:

- “Hypothesis Test Terminology” on page 6-3
- “Hypothesis Test Assumptions” on page 6-4
- “Example: Hypothesis Testing” on page 6-5
- “Available Hypothesis Tests” on page 6-9



## Hypothesis Test Terminology

To get started, there are some terms to define and assumptions to make:

- The *null hypothesis* is the original assertion. In this case the null hypothesis is that the average price of a gallon of gas is \$1.15. The notation is  $H_0: \mu = 1.15$ .
- There are three possibilities for the *alternative hypothesis*. You might only be interested in the result if gas prices were actually higher. In this case, the alternative hypothesis is  $H_1: \mu > 1.15$ . The other possibilities are  $H_1: \mu < 1.15$  and  $H_1: \mu \neq 1.15$ .
- The *significance level* is related to the degree of certainty you require in order to reject the null hypothesis in favor of the alternative. By taking a small sample you cannot be certain about your conclusion. So you decide in advance to reject the null hypothesis if the probability of observing your sampled result is less than the significance level. For a typical significance level of 5%, the notation is  $\alpha = 0.05$ . For this significance level, the probability of incorrectly rejecting the null hypothesis when it is actually true is 5%. If you need more protection from this error, then choose a lower value of  $\alpha$ .
- The *p-value* is the probability of observing the given sample result under the assumption that the null hypothesis is true. If the p-value is less than  $\alpha$ , then you reject the null hypothesis. For example, if  $\alpha = 0.05$  and the p-value is 0.03, then you reject the null hypothesis.

The converse is not true. If the p-value is greater than  $\alpha$ , you have insufficient evidence to reject the null hypothesis.

- The outputs for many hypothesis test functions also include *confidence intervals*. Loosely speaking, a confidence interval is a range of values that have a chosen probability of containing the true hypothesized quantity. Suppose, in the example, 1.15 is inside a 95% confidence interval for the mean,  $\mu$ . That is equivalent to being unable to reject the null hypothesis at a significance level of 0.05. Conversely if the 100(1- $\alpha$ ) confidence interval does not contain 1.15, then you reject the null hypothesis at the  $\alpha$  level of significance.

## Hypothesis Test Assumptions

The difference between hypothesis test procedures often arises from differences in the assumptions that the researcher is willing to make about the data sample. For example, the Z-test assumes that the data represents independent samples from the same normal distribution and that you know the standard deviation,  $\sigma$ . The t-test has the same assumptions except that you estimate the standard deviation using the data instead of specifying it as a known quantity.

Both tests have an associated signal-to-noise ratio

$$Z = \frac{\bar{x} - \mu}{\sigma} \quad \text{or} \quad T = \frac{\bar{x} - \mu}{s}$$

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

The signal is the difference between the average and the hypothesized mean. The noise is the standard deviation posited or estimated.

If the null hypothesis is true, then  $Z$  has a standard normal distribution,  $N(0,1)$ .  $T$  has a Student's t distribution with the degrees of freedom,  $\nu$ , equal to one less than the number of data values.

Given the observed result for  $Z$  or  $T$ , and knowing the distribution of  $Z$  and  $T$  assuming the null hypothesis is true, it is possible to compute the probability (p-value) of observing this result. A very small p-value casts doubt on the truth of the null hypothesis. For example, suppose that the p-value was 0.001, meaning that the probability of observing the given  $Z$  or  $T$  was one in a thousand. That should make you skeptical enough about the null hypothesis that you reject it rather than believe that your result was just a lucky 999 to 1 shot.

There are also nonparametric tests that do not even require the assumption that the data come from a normal distribution. In addition, there are functions for testing whether the normal assumption is reasonable.

## Example: Hypothesis Testing

This example uses the gasoline price data in `gas.mat`. There are two samples of 20 observed gas prices for the months of January and February, 1993.

```
load gas
prices = [price1 price2];
```

As a first step, you may want to test whether the samples from each month follow a normal distribution. As each sample is relatively small, you might choose to perform a Lilliefors test (rather than a Jarque-Bera test).

```
lillietest(price1)
```

```
ans =
```

```
0
```

```
lillietest(price2)
```

```
ans =
```

```
0
```

The result of the hypothesis test is a Boolean value that is 0 when you do not reject the null hypothesis, and 1 when you do reject that hypothesis. In each case, there is no need to reject the null hypothesis that the samples have a normal distribution.

Suppose it is historically true that the standard deviation of gas prices at gas stations around Massachusetts is four cents a gallon. The Z-test is a procedure for testing the null hypothesis that the average price of a gallon of gas in January (`price1`) is \$1.15.

```
[h,pvalue,ci] = ztest(price1/100,1.15,0.04)
```

```
h =
```

```
0
```

```
pvalue =
```

```
0.8668
```

```
ci =
```

```
1.1340    1.1690
```

The Boolean output is  $h = 0$ , so you do not reject the null hypothesis.

The result suggests that \$1.15 is reasonable. The 95% confidence interval [1.1340 1.1690] neatly brackets \$1.15.

What about February? Try a t-test with price2. Now you are not assuming that you know the standard deviation in price.

```
[h,pvalue,ci] = ttest(price2/100,1.15)

h =
     1

pvalue =
    4.9517e-04

ci =
    1.1675    1.2025
```

With the Boolean result  $h = 1$ , you can reject the null hypothesis at the default significance level, 0.05.

It looks like \$1.15 is not a reasonable estimate of the gasoline price in February. The low end of the 95% confidence interval is greater than 1.15.

The function `ttest2` allows you to compare the means of the two data samples.

```
[h,sig,ci] = ttest2(price1,price2)

h =
     1

sig =
    0.0083

ci =
   -5.7845   -0.9155
```

The confidence interval (ci above) indicates that gasoline prices were between one and six cents lower in January than February.

If the two samples were not normally distributed but had similar shape, it would have been more appropriate to use the nonparametric rank sum test in

place of the t-test. You can still use the rank sum test with normally distributed data, but it is less powerful than the t-test.

```
[p,h,stats] = ranksum(price1, price2)

p =
    0.0092

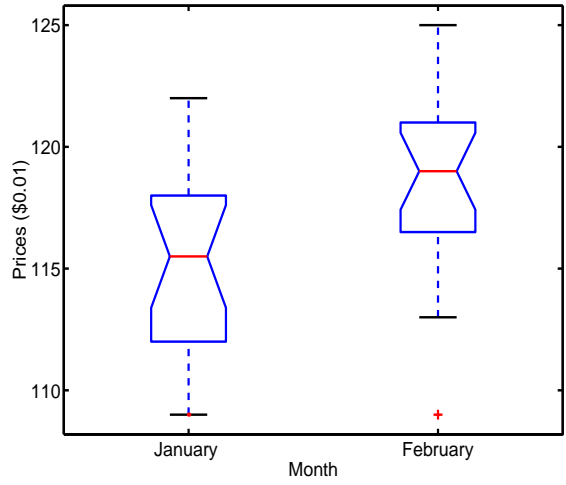
h =
     1

stats =
    zval: -2.6064
    ranksum: 314
```

As might be expected, the rank sum test leads to the same conclusion but it is less sensitive to the difference between samples (higher p-value).

The box plot below gives the same conclusion graphically. Note that the notches have little, if any, overlap. Refer to “Statistical Plots” on page 8-1 for more information about box plots.

```
boxplot(prices,1)
set(gca,'XtickLabel',str2mat('January','February'))
xlabel('Month')
ylabel('Prices ($0.01)')
```



## Available Hypothesis Tests

The Statistics Toolbox has functions for performing the following tests.

<b>Function</b>	<b>What it Tests</b>
<code>jbtest</code>	Normal distribution for one sample
<code>kstest</code>	Any specified distribution for one sample
<code>kstest2</code>	Equal distributions for two samples
<code>lillietest</code>	Normal distribution for one sample
<code>ranksum</code>	Median of two unpaired samples
<code>signrank</code>	Median of two paired samples
<code>signtest</code>	Median of two paired samples
<code>ttest</code>	Mean of one normal sample
<code>ttest2</code>	Mean of two normal samples
<code>ztest</code>	Mean of normal sample with known standard deviation





# Multivariate Statistics

---

Principal Components Analysis (p. 7-2)	Describes how to perform principal components analysis.
Factor Analysis (p. 7-15)	Describes how to perform factor analysis.
Multivariate Analysis of Variance (MANOVA) (p. 7-24)	Describes how to perform multivariate analysis of variance.
Cluster Analysis (p. 7-30)	Describes how to perform cluster analysis.
Multidimensional Scaling (p. 7-53)	Describes how to perform multidimensional scaling.

## Principal Components Analysis

One of the difficulties inherent in multivariate statistics is the problem of visualizing data that has many variables. In MATLAB, the `plot` command displays a graph of the relationship between two variables. The `plot3` and `surf` commands display different three-dimensional views. But when there are more than three variables, it is more difficult to visualize their relationships.

Fortunately, in data sets with many variables, groups of variables often move together. One reason for this is that more than one variable might be measuring the same driving principle governing the behavior of the system. In many systems there are only a few such driving forces. But an abundance of instrumentation enables you to measure dozens of system variables. When this happens, you can take advantage of this redundancy of information. You can simplify the problem by replacing a group of variables with a single new variable.

Principal components analysis is a quantitatively rigorous method for achieving this simplification. The method generates a new set of variables, called *principal components*. Each principal component is a linear combination of the original variables. All the principal components are orthogonal to each other, so there is no redundant information. The principal components as a whole form an orthogonal basis for the space of the data.

There are an infinite number of ways to construct an orthogonal basis for several columns of data. What is so special about the principal component basis?

The first principal component is a single axis in space. When you project each observation on that axis, the resulting values form a new variable. And the variance of this variable is the maximum among all possible choices of the first axis.

The second principal component is another axis in space, perpendicular to the first. Projecting the observations on this axis generates another new variable. The variance of this variable is the maximum among all possible choices of this second axis.

The full set of principal components is as large as the original set of variables. But it is commonplace for the sum of the variances of the first few principal components to exceed 80% of the total variance of the original data. By

examining plots of these few new variables, researchers often develop a deeper understanding of the driving forces that generated the original data.

You can use the function `princomp` to find the principal components. The following sections provide an example and explain the four outputs of `princomp`:

- “Example: Principal Components Analysis” on page 7-3
- “The Principal Component Coefficients (First Output)” on page 7-6
- “The Component Scores (Second Output)” on page 7-6
- “The Component Variances (Third Output)” on page 7-10
- “Hotelling’s T2 (Fourth Output)” on page 7-11
- “Visualizing the results of a Principal Components Analysis — the Biplot” on page 7-12

To use `princomp`, you need to have the actual measured data you want to analyze. However, if you lack the actual data, but have the sample covariance or correlation matrix for the data, you can still use the function `pcacov` to perform a principal components analysis. See the reference page for `pcacov` for a description of its inputs and outputs.

## Example: Principal Components Analysis

Consider a sample application that uses nine different indices of the quality of life in 329 U.S. cities. These are climate, housing, health, crime, transportation, education, arts, recreation, and economics. For each index, higher is better. For example, a higher index for crime means a lower crime rate.

Start by loading the data in `cities.mat`.

```
load cities
whos
```

Name	Size	Bytes	Class
categories	9x14	252	char array
names	329x43	28294	char array
ratings	329x9	23688	double array

The `whos` command generates a table of information about all the variables in the workspace.

The `cities` data set contains three variables:

- `categories`, a string matrix containing the names of the indices
- `names`, a string matrix containing the 329 city names
- `ratings`, the data matrix with 329 rows and 9 columns

The `categories` variable has the following values:

```
categories

categories =
  climate
  housing
  health
  crime
  transportation
  education
  arts
  recreation
  economics
```

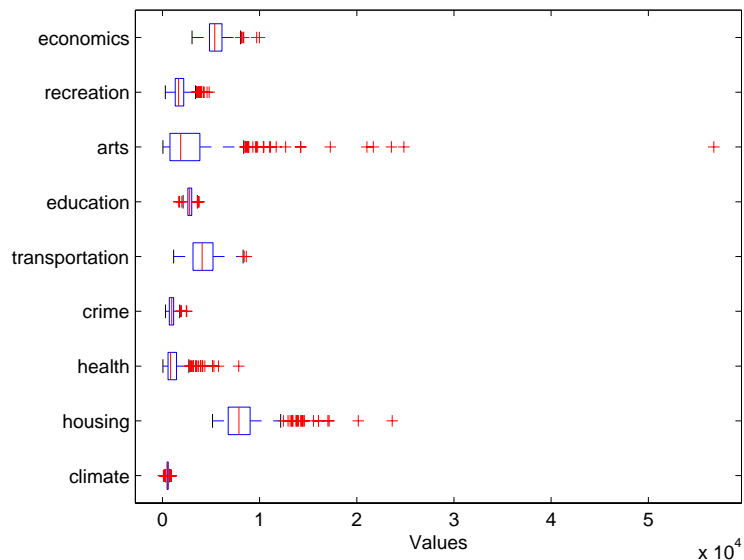
The first five rows of `names` are

```
first5 = names(1:5,:)
first5 =
  Abilene, TX
  Akron, OH
  Albany, GA
  Albany-Troy, NY
  Albuquerque, NM
```

To get a quick impression of the `ratings` data, make a box plot.

```
boxplot(ratings,'orientation','horizontal','labels',categories)
```

This command generates the plot below. Note that there is substantially more variability in the ratings of the arts and housing than in the ratings of crime and climate.



Ordinarily you might also graph pairs of the original variables, but there are 36 two-variable plots. Perhaps principal components analysis can reduce the number of variables you need to consider.

Sometimes it makes sense to compute principal components for raw data. This is appropriate when all the variables are in the same units. Standardizing the data is often preferable when the variables are in different units or when the variance of the different columns is substantial (as in this case).

You can standardize the data by dividing each column by its standard deviation.

```
stdr = std(ratings);
sr = ratings./repmat(stdr,329,1);
```

Now you are ready to find the principal components.

```
[coefs,scores,variances,t2] = princomp(sr);
```

The following sections explain the four outputs from `princomp`.

## The Principal Component Coefficients (First Output)

The first output of the `princomp` function, `coefs`, contains the coefficients for nine principal components. These are the linear combinations of the original variables that generate the new variables.

The first three principal component coefficient vectors are

```
c3 = coefs(:,1:3)
c3 =
    0.2064    0.2178   -0.6900
    0.3565    0.2506   -0.2082
    0.4602   -0.2995   -0.0073
    0.2813    0.3553    0.1851
    0.3512   -0.1796    0.1464
    0.2753   -0.4834    0.2297
    0.4631   -0.1948   -0.0265
    0.3279    0.3845   -0.0509
    0.1354    0.4713    0.6073
```

The largest coefficients in the first column (first principal component) are the third and seventh elements, corresponding to the variables `health` and `arts`. All the coefficients of the first principal component have the same sign, making it a weighted average of all the original variables.

Because the principal components are unit length and orthogonal, premultiplying the matrix `c3` by its transpose yields the identity matrix.

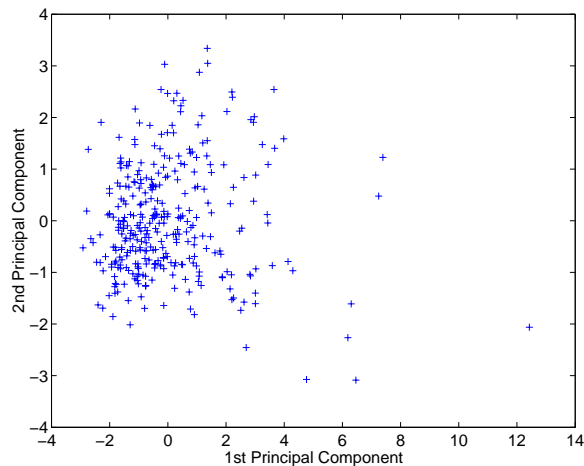
```
I = c3'*c3
I =
    1.0000   -0.0000   -0.0000
   -0.0000    1.0000   -0.0000
   -0.0000   -0.0000    1.0000
```

## The Component Scores (Second Output)

The second output, `scores`, is the original data mapped into the new coordinate system defined by the principal components. This output is the same size as the input data matrix.

A plot of the first two columns of scores shows the ratings data projected onto the first two principal components. `princomp` computes the scores to have mean zero.

```
plot(scores(:,1),scores(:,2),'+')
xlabel('1st Principal Component');
ylabel('2nd Principal Component');
```



Note the outlying points in the right half of the plot.

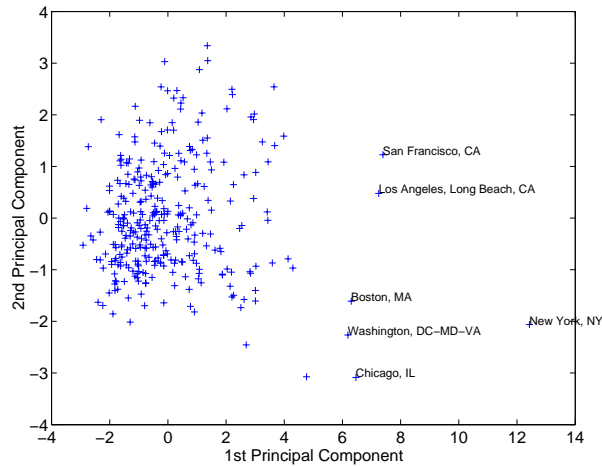
While it is possible to create a three-dimensional plot using three columns of scores, the examples in this section create two-dimensional plots, which are easier to describe.

The function `gname` is useful for graphically identifying a few points in a plot like this. You can call `gname` with a string matrix containing as many case labels as points in the plot. The string matrix `names` works for labeling points with the city names.

```
gname(names)
```

Move your cursor over the plot and click once near each point in the right half. As you click each point, MATLAB labels it with the proper row from the `names` string matrix. When you are finished labeling points, press the **Return** key.

Here is the resulting plot.

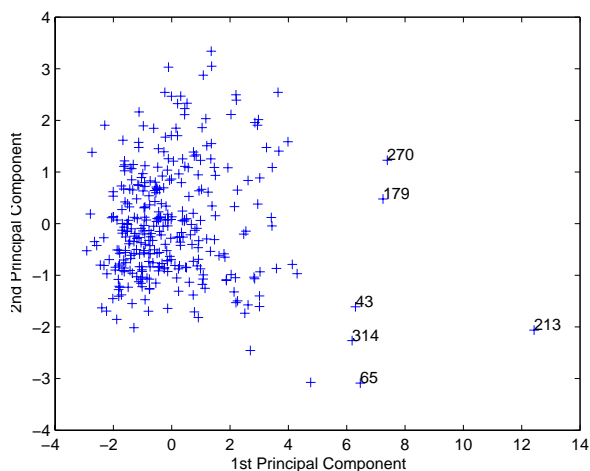


The labeled cities are some of the biggest population centers in the United States. They are definitely different from the remainder of the data, so perhaps they should be considered separately. To remove the labeled cities from the data, first identify their corresponding row numbers as follows:

- 1 Close the plot window.
- 2 Redraw the plot by entering

```
plot(scores(:,1),scores(:,2),'+')
xlabel('1st Principal Component');
ylabel('2nd Principal Component');
```
- 3 Enter gname without any arguments.
- 4 Click near the points you labeled in the preceding figure. This labels the points by their row numbers, as shown in the following figure.





Then you can create an index variable containing the row numbers of all the metropolitan areas you choose.

```
metro = [43 65 179 213 234 270 314];
names(metro,:)
```

```
ans =
Boston, MA
Chicago, IL
Los Angeles, Long Beach, CA
New York, NY
Philadelphia, PA-NJ
San Francisco, CA
Washington, DC-MD-VA
```

To remove these rows from the ratings matrix, enter the following.

```
rsubset = ratings;
nsubset = names;
nsubset(metro,:) = [];
rsubset(metro,:) = [];
```

```
size(rsubset)
ans =
    322     9
```

## The Component Variances (Third Output)

The third output, `variances`, is a vector containing the variance explained by the corresponding principal component. Each column of scores has a sample variance equal to the corresponding element of `variances`.

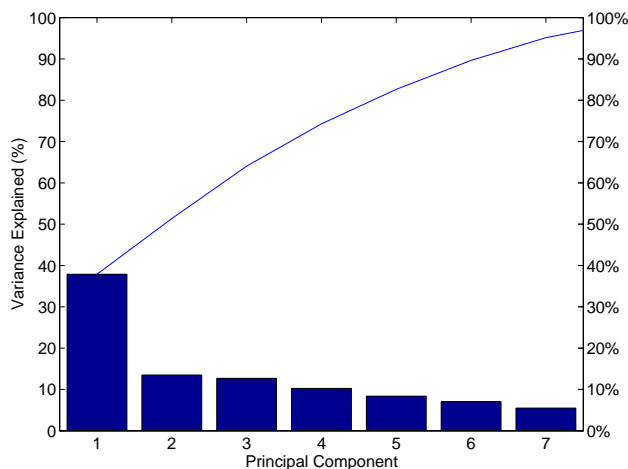
```
variances
variances =
    3.4083
    1.2140
    1.1415
    0.9209
    0.7533
    0.6306
    0.4930
    0.3180
    0.1204
```

You can easily calculate the percent of the total variability explained by each principal component.

```
percent_explained = 100*variances/sum(variances)
percent_explained =
    37.8699
    13.4886
    12.6831
    10.2324
    8.3698
    7.0062
    5.4783
    3.5338
    1.3378
```

Use the `pareto` function to make a scree plot of the percent variability explained by each principal component.

```
pareto(percent_explained)
xlabel('Principal Component')
ylabel('Variance Explained (%)')
```



The preceding figure shows that the only clear break in the amount of variance accounted for by each component is between the first and second components. However, that component by itself explains less than 40% of the variance, so more components are probably needed. You can see that the first three principal components explain roughly two thirds of the total variability in the standardized ratings, so that might be a reasonable way to reduce the dimensions in order to visualize the data.

## Hotelling's $T^2$ (Fourth Output)

The last output of the `princomp` function, `t2`, is Hotelling's  $T^2$ , a statistical measure of the multivariate distance of each observation from the center of the data set. This is an analytical way to find the most extreme points in the data.

```
[st2, index] = sort(t2, 'descend'); % Sort in descending order.
extreme = index(1)
extreme =
```

```
213
```

```
names(extreme, :)
```

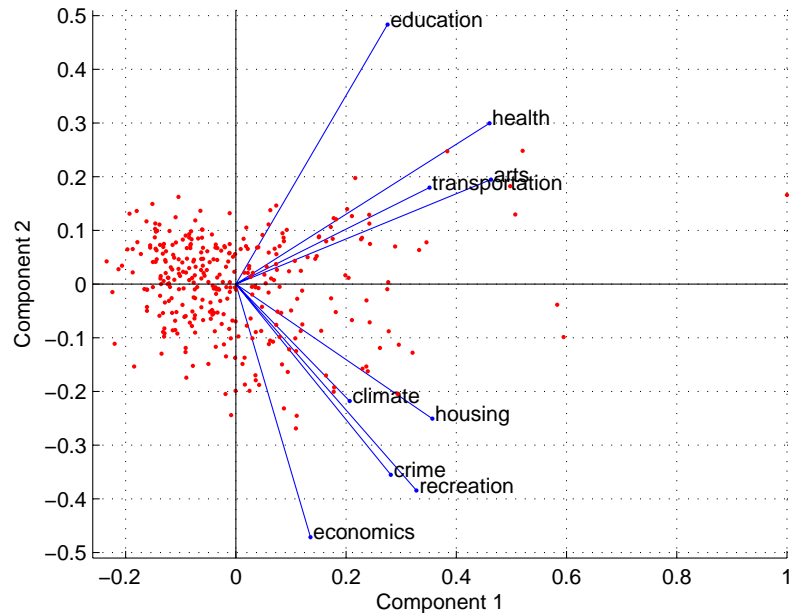
```
ans =
New York, NY
```

It is not surprising that the ratings for New York are the furthest from the average U.S. town.

## Visualizing the results of a Principal Components Analysis – the Biplot

You can use the biplot function to help visualize both the principal component coefficients for each variable and the principal component scores for each observation in a single plot. For example, the following command plots the results from the principal components analysis on the cities and labels each of the variables.

```
biplot(coefs(:,1:2), 'scores', scores(:,1:2), ...
'varlabels', categories);
axis([-0.26 1 -0.51 0.51]);
```



Each of the nine variables is represented in this plot by a vector, and the direction and length of the vector indicates how each variable contributes to the two principal components in the plot. For example, you have seen that the first principal component, represented in this biplot by the horizontal axis, has positive coefficients for all nine variables. That corresponds to the nine vectors directed into the right half of the plot. You have also seen that the second principal component, represented by the vertical axis, has positive coefficients for the variables education, health, arts, and education, and negative coefficients for the remaining five variables. That corresponds to vectors directed into the top and bottom halves of the plot, respectively. This indicates that this component distinguishes between cities that have high values for the first set of variables and low for the second, and cities that have the opposite.

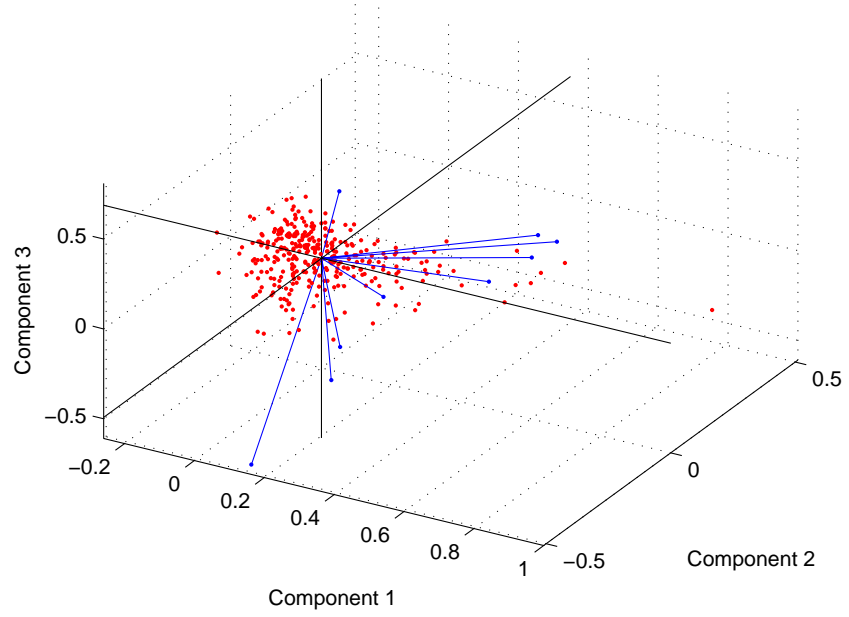
The variable labels in this figure are somewhat crowded. You could either leave out the 'VarLabels' parameter when making the plot, or simply select and drag some of the labels to better positions using the Edit Plot tool from the figure window toolbar.

Each of the 329 observations is represented in this plot by a point, and their locations indicate the score of each observation for the two principal components in the plot. For example, points near the left edge of this plot have the lowest scores for the first principal component. The points are scaled to fit within the unit square, so only their relative locations may be determined from the plot.

You can use the **Data Cursor**, in the **Tools** menu in the figure window, to identify the items in this plot. By clicking on a variable (vector), you can read off that variable's coefficients for each principal component. By clicking on an observation (point), you can read off that observation's scores for each principal component.

You can also make a biplot in three dimensions. This can be useful if the first two principal coordinates do not explain enough of the variance in your data. Selecting **Rotate 3D** in the **Tools** menu enables you to rotate the figure to see it from different angles.

```
biplot(coefs(:,1:3), 'scores', scores(:,1:3), ...  
      'obslabels', names);  
axis([-0.26 1 -0.51 0.51 -0.61 0.81]);  
view([30 40]);
```



## Factor Analysis

Multivariate data often includes a large number of measured variables, and sometimes those variables overlap, in the sense that groups of them might be dependent. For example, in a decathlon, each athlete competes in 10 events, but several of them can be thought of as speed events, while others can be thought of as strength events, etc. Thus, you can think of a competitor's 10 event scores as largely dependent on a smaller set of three or four types of athletic ability.

Factor analysis is a way to fit a model to multivariate data to estimate just this sort of interdependence. In a factor analysis model, the measured variables depend on a smaller number of unobserved (latent) factors. Because each factor might affect several variables in common, they are known as *common factors*. Each variable is assumed to be dependent on a linear combination of the common factors, and the coefficients are known as loadings. Each measured variable also includes a component due to independent random variability, known as “specific variance” because it is specific to one variable.

Specifically, factor analysis assumes that the covariance matrix of your data is of the form

$$\sigma_x = \Lambda\Lambda^T + \Psi$$

where  $\Lambda$  is the matrix of loadings, and the elements of the diagonal matrix  $\Psi$  are the specific variances. The function `factoran` fits the Factor Analysis model using maximum likelihood.

This section includes these topics:

- “Example: Finding Common Factors Affecting Stock Prices” on page 7-16
- “Factor Rotation” on page 7-18
- “Predicting Factor Scores” on page 7-19
- “Visualizing the Results of a Factor Analysis — the Biplot” on page 7-21
- “Comparison of Factor Analysis and Principal Components Analysis” on page 7-23

## Example: Finding Common Factors Affecting Stock Prices

Over the course of 100 weeks, the percent change in stock prices for ten companies has been recorded. Of the ten companies, the first four can be classified as primarily technology, the next three as financial, and the last three as retail. It seems reasonable that the stock prices for companies that are in the same sector might vary together as economic conditions change. Factor Analysis can provide quantitative evidence that companies within each sector do experience similar week-to-week changes in stock price.

In this example, you first load the data, and then call `factoran`, specifying a model fit with three common factors. By default, `factoran` computes rotated estimates of the loadings to try and make their interpretation simpler. But in this example, you specify an unrotated solution.

```
load stockreturns
[Loadings,specificVar,T,stats] = factoran(stocks,3,...
                                         'rotate','none');
```

The first two `factoran` return arguments are the estimated loadings and the estimated specific variances. Each row of the loadings matrix represents one of the ten stocks, and each column corresponds to a common factor. With unrotated estimates, interpretation of the factors in this fit is difficult because most of the stocks contain fairly large coefficients for two or more factors.

```
Loadings
Loadings =
    0.8885    0.2367   -0.2354
    0.7126    0.3862    0.0034
    0.3351    0.2784   -0.0211
    0.3088    0.1113   -0.1905
    0.6277   -0.6643    0.1478
    0.4726   -0.6383    0.0133
    0.1133   -0.5416    0.0322
    0.6403    0.1669    0.4960
    0.2363    0.5293    0.5770
    0.1105    0.1680    0.5524
```



---

**Note** “Factor Rotation” on page 7-18 helps to simplify the structure in the Loadings matrix, to make it easier to assign meaningful interpretations to the factors.

---

From the estimated specific variances, you can see that the model indicates that a particular stock price varies quite a lot beyond the variation due to the common factors.

```
specificVar
specificVar =
  0.0991
  0.3431
  0.8097
  0.8559
  0.1429
  0.3691
  0.6928
  0.3162
  0.3311
  0.6544
```

A specific variance of 1 would indicate that there is *no* common factor component in that variable, while a specific variance of 0 would indicate that the variable is *entirely* determined by common factors. These data seem to fall somewhere in between.

The p-value returned in the stats structure fails to reject the null hypothesis of three common factors, suggesting that this model provides a satisfactory explanation of the covariation in these data.

```
stats.p
ans =
  0.8144
```

To determine whether fewer than three factors can provide an acceptable fit, you can try a model with two common factors. The p-value for this second fit is highly significant, and rejects the hypothesis of two factors, indicating that the simpler model is not sufficient to explain the pattern in these data.

```
[Loadings2,specificVar2,T2,stats2] = factoran(stocks, 2,...
```

```

                                'rotate', 'none');
stats2.p
ans =
    3.5610e-006

```

## Factor Rotation

As the results in “Example: Finding Common Factors Affecting Stock Prices” on page 7-16 illustrate, the estimated loadings from an unrotated factor analysis fit can have a complicated structure. The goal of factor rotation is to find a parameterization in which each variable has only a small number of large loadings. That is, each variable is affected by a small number of factors, preferably only one. This can often make it easier to interpret what the factors represent.

If you think of each row of the loadings matrix as coordinates of a point in M-dimensional space, then each factor corresponds to a coordinate axis. Factor rotation is equivalent to rotating those axes and computing new loadings in the rotated coordinate system. There are various ways to do this. Some methods leave the axes orthogonal, while others are oblique methods that change the angles between them. For this example, you can rotate the estimated loadings by using the promax criterion, a common oblique method.

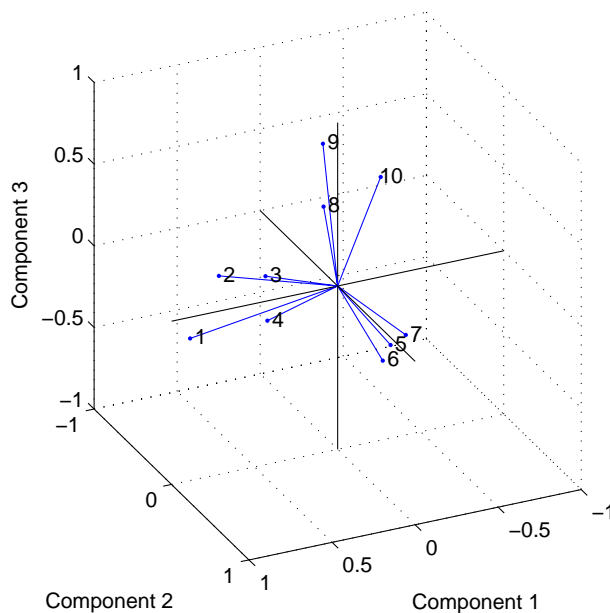
```

[LoadingsPM,specVarPM] = factoran(stocks,3,'rotate','promax');
LoadingsPM
LoadingsPM =
    0.9452    0.1214   -0.0617
    0.7064   -0.0178    0.2058
    0.3885   -0.0994    0.0975
    0.4162   -0.0148   -0.1298
    0.1021    0.9019    0.0768
    0.0873    0.7709   -0.0821
   -0.1616    0.5320   -0.0888
    0.2169    0.2844    0.6635
    0.0016   -0.1881    0.7849
   -0.2289    0.0636    0.6475

```

Promax rotation creates a simpler structure in the loadings, one in which most of the stocks have a large loading on only one factor. To see this structure more clearly, you can use the biplot function to plot each stock using its factor loadings as coordinates.

```
biplot(LoadingsPM, 'varlabels', num2str((1:10)'));
axis square
view(155,27);
```



This plot shows that promax has rotated the factor loadings to a simpler structure. Each stock depends primarily on only one factor, and it is possible to describe each factor in terms of the stocks that it affects. Based on which companies are near which axes, you could reasonably conclude that the first factor axis represents the financial sector, the second retail, and the third technology. The original conjecture, that stocks vary primarily within sector, is apparently supported by the data.

## Predicting Factor Scores

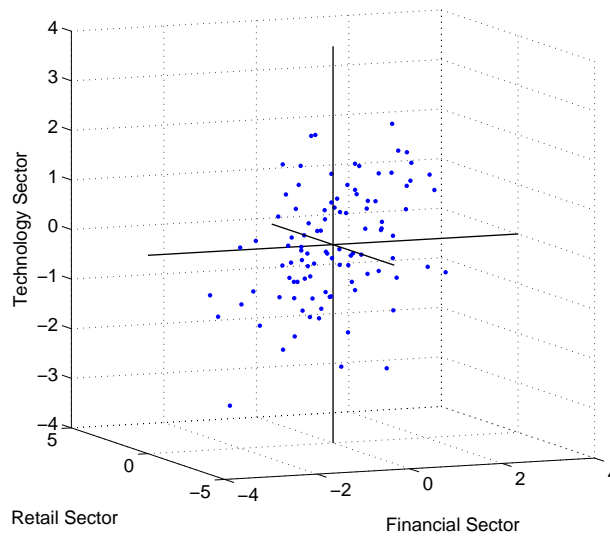
Sometimes, it is useful to be able to classify an observation based on its factor scores. For example, if you accepted the three-factor model and the interpretation of the rotated factors, you might want to categorize each week

in terms of how favorable it was for each of the three stock sectors, based on the data from the ten observed stocks.

Because the data in this example are the raw stock price changes, and not just their correlation matrix, you can have `factoran` return estimates of the value of each of the three rotated common factors for each week. You can then plot the estimated scores to see how the different stock sectors were affected during each week.

```
[LoadingsPM,specVarPM,TPM,stats,F] = factoran(stocks, 3,...
                                                'rotate','promax');

subplot(1,1,1);
plot3(F(:,1),F(:,2),F(:,3),'b. ');
line([-4 4 NaN 0 0 NaN 0 0], [0 0 NaN -4 4 NaN 0 0],...
      [0 0 NaN 0 0 NaN -4 4], 'Color','black');
xlabel('Financial Sector');
ylabel('Retail Sector');
zlabel('Technology Sector');
grid on;
axis square;
view(-22.5, 8);
```



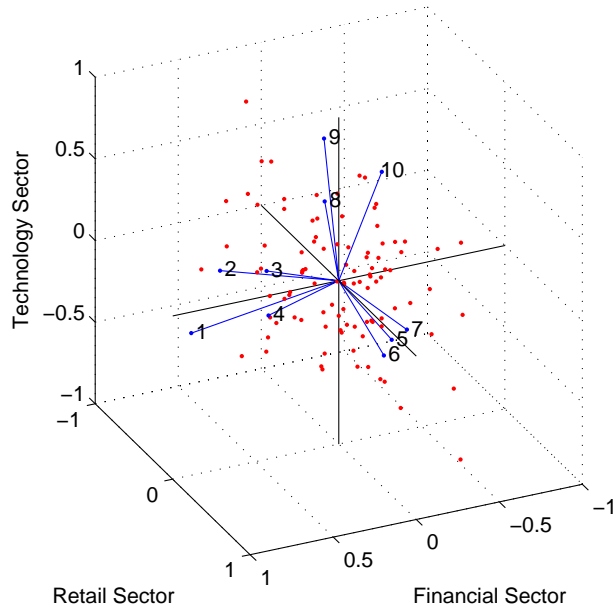
Oblique rotation often creates factors that are correlated. This plot shows some evidence of correlation between the first and third factors, and you can investigate further by computing the estimated factor correlation matrix.

```
inv(TPM'*TPM)
ans =
    1.0000    0.1559    0.4082
    0.1559    1.0000   -0.0559
    0.4082   -0.0559    1.0000
```

## Visualizing the Results of a Factor Analysis – the Biplot

You can use the `biplot` function to help visualize both the factor loadings for each variable and the factor scores for each observation in a single plot. For example, the following command plots the results from the factor analysis on the stock data and labels each of the 10 stocks.

```
biplot(LoadingsPM, 'scores',F, 'varlabels',num2str((1:10)'));
xlabel('Financial Sector'); ylabel('Retail Sector');
zlabel('Technology Sector');
axis square
view(155,27);
```



In this case, the factor analysis includes three factors, and so the biplot is three-dimensional. Each of the 10 stocks is represented in this plot by a vector, and the direction and length of the vector indicates how each stock depends on the underlying factors. For example, you have seen that after promax rotation, the first four stocks have positive loadings on the first factor, and unimportant loadings on the other two factors. That first factor, interpreted as a financial sector effect, is represented in this biplot as one of the horizontal axes. The dependence of those four stocks on that factor corresponds to the four vectors directed approximately along that axis. Similarly, the dependence of stocks 5, 6, and 7 primarily on the second factor, interpreted as a retail sector effect, is represented by vectors directed approximately along that axis.

Each of the 100 observations is represented in this plot by a point, and their locations indicate the score of each observation for the three factors. For example, points near the top of this plot have the highest scores for the technology sector factor. The points are scaled to fit within the unit square, so only their relative locations can be determined from the plot.

You can use the **Data Cursor** tool from the **Tools** menu in the figure window to identify the items in this plot. By clicking a stock (vector), you can read off that stock's loadings for each factor. By clicking an observation (point), you can read off that observation's scores for each factor.

## **Comparison of Factor Analysis and Principal Components Analysis**

There is a good deal of overlap in terminology and goals between principal components analysis (PCA) and factor analysis (FA). Much of the literature on the two methods does not distinguish between them, and some algorithms for fitting the FA model involve PCA. Both are dimension-reduction techniques, in the sense that they can be used to replace a large set of observed variables with a smaller set of new variables. However, the two methods are different in their goals and in their underlying models. Roughly speaking, you should use PCA when you simply need to summarize or approximate your data using fewer dimensions (to visualize it, for example), and you should use FA when you need an explanatory model for the correlations among your data.

## Multivariate Analysis of Variance (MANOVA)

The analysis of variance technique in “One-Way Analysis of Variance (ANOVA)” on page 4-3 takes a set of grouped data and determine whether the mean of a variable differs significantly between groups. Often there are multiple variables, and you are interested in determining whether the entire set of means is different from one group to the next. There is a multivariate version of analysis of variance that can address that problem, as illustrated in the following example.

### Example: Multivariate Analysis of Variance

The `carsmall` data set has measurements on a variety of car models from the years 1970, 1976, and 1982. Suppose you are interested in whether the characteristics of the cars have changed over time.

First, load the data.

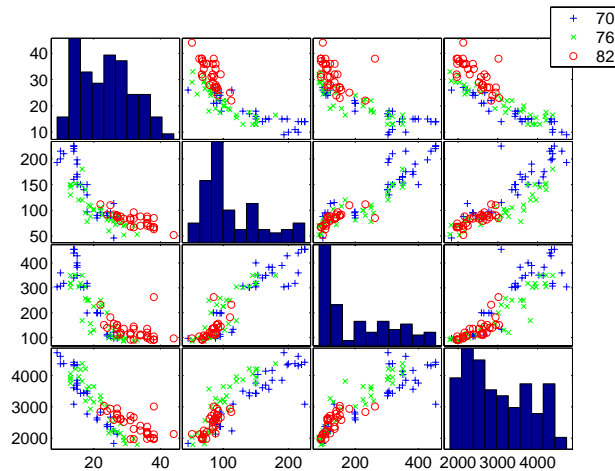
```
load carsmall
whos
```

Name	Size	Bytes	Class
Acceleration	100x1	800	double array
Cylinders	100x1	800	double array
Displacement	100x1	800	double array
Horsepower	100x1	800	double array
MPG	100x1	800	double array
Model	100x36	7200	char array
Model_Year	100x1	800	double array
Origin	100x7	1400	char array
Weight	100x1	800	double array

Four of these variables (Acceleration, Displacement, Horsepower, and MPG) are continuous measurements on individual car models. The variable `Model_Year` indicates the year in which the car was made. You can create a grouped plot matrix of these variables using the `gplotmatrix` function.

```
x = [MPG Horsepower Displacement Weight];
gplotmatrix(x,[],Model_Year,[],'+xo')
```





(When the second argument of `gplotmatrix` is empty, the function graphs the columns of the `x` argument against each other, and places histograms along the diagonals. The empty fourth argument produces a graph with the default colors. The fifth argument controls the symbols used to distinguish between groups.)

It appears the cars do differ from year to year. The upper right plot, for example, is a graph of MPG versus Weight. The 1982 cars appear to have higher mileage than the older cars, and they appear to weigh less on average. But as a group, are the three years significantly different from one another? The `manova1` function can answer that question.

```
[d,p,stats] = manova1(x,Model_Year)

d =
    2

p =
    1.0e-006 *
         0
    0.1141

stats =
         W: [4x4 double]
```

```
      B: [4x4 double]
      T: [4x4 double]
    dfW: 90
    dfB:  2
    dfT: 92
  lambda: [2x1 double]
  chisq: [2x1 double]
  chisqdf: [2x1 double]
  eigenval: [4x1 double]
  eigenvec: [4x4 double]
  canon: [100x4 double]
  mdist: [100x1 double]
  gmdist: [3x3 double]
```

The `manova1` function produces three outputs:

- The first output, `d`, is an estimate of the dimension of the group means. If the means were all the same, the dimension would be 0, indicating that the means are at the same point. If the means differed but fell along a line, the dimension would be 1. In the example the dimension is 2, indicating that the group means fall in a plane but not along a line. This is the largest possible dimension for the means of three groups.
- The second output, `p`, is a vector of p-values for a sequence of tests. The first p-value tests whether the dimension is 0, the next whether the dimension is 1, and so on. In this case both p-values are small. That's why the estimated dimension is 2.
- The third output, `stats`, is a structure containing several fields, described in the following section.

### The Fields of the `stats` Structure

The `W`, `B`, and `T` fields are matrix analogs to the within, between, and total sums of squares in ordinary one-way analysis of variance. The next three fields are the degrees of freedom for these matrices. Fields `lambda`, `chisq`, and `chisqdf` are the ingredients of the test for the dimensionality of the group means. (The p-values for these tests are the first output argument of `manova1`.)

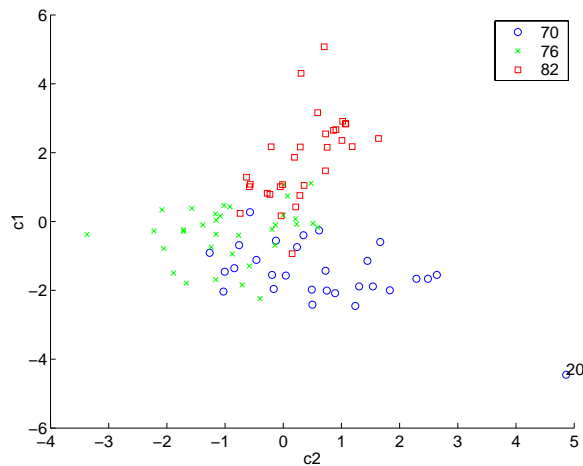
The next three fields are used to do a canonical analysis. Recall that in principal components analysis (“Principal Components Analysis” on page 7-2) you look for the combination of the original variables that has the largest possible variation. In multivariate analysis of variance, you instead look for the

linear combination of the original variables that has the largest separation between groups. It is the single variable that would give the most significant result in a univariate one-way analysis of variance. Having found that combination, you next look for the combination with the second highest separation, and so on.

The `eigenvec` field is a matrix that defines the coefficients of the linear combinations of the original variables. The `eigenval` field is a vector measuring the ratio of the between-group variance to the within-group variance for the corresponding linear combination. The `canon` field is a matrix of the canonical variable values. Each column is a linear combination of the mean-centered original variables, using coefficients from the `eigenvec` matrix.

A grouped scatter plot of the first two canonical variables shows more separation between groups than a grouped scatter plot of any pair of original variables. In this example it shows three clouds of points, overlapping but with distinct centers. One point in the bottom right sits apart from the others. By using the `gname` function, you can see that this is the 20th point.

```
c1 = stats.canon(:,1);
c2 = stats.canon(:,2);
gscatter(c2,c1,Model_Year,[],'oxs')
gname
```



Roughly speaking, the first canonical variable, `c1`, separates the 1982 cars (which have high values of `c1`) from the older cars. The second canonical variable, `c2`, reveals some separation between the 1970 and 1976 cars.

The final two fields of the `stats` structure are Mahalanobis distances. The `mdist` field measures the distance from each point to its group mean. Points with large values may be outliers. In this data set, the largest outlier is the one in the scatter plot, the Buick Estate station wagon. (Note that you could have supplied the model name to the `gname` function above if you wanted to label the point with its model name rather than its row number.)

```
max(stats.mdist)
ans =

    31.5273

find(stats.mdist == ans)
ans =

    20

Model(20,:)
ans =

    buick_estate_wagon_(sw)
```

The `gmdist` field measures the distances between each pair of group means. The following commands examine the group means and their distances:

```
grpstats(x, Model_Year)

ans =

    1.0e+003 *
    0.0177    0.1489    0.2869    3.4413
    0.0216    0.1011    0.1978    3.0787
    0.0317    0.0815    0.1289    2.4535

stats.gmdist

ans =

     0    3.8277    11.1106
    3.8277     0    6.1374
    11.1106    6.1374     0
```

As might be expected, the multivariate distance between the extreme years 1970 and 1982 (11.1) is larger than the difference between more closely spaced years (3.8 and 6.1). This is consistent with the scatter plots, where the points seem to follow a progression as the year changes from 1970 through 1976 to 1982. If you had more groups, you might find it instructive to use the `manovacluster` function to draw a diagram that presents clusters of the groups, formed using the distances between their means.

## Cluster Analysis

Cluster analysis, also called segmentation analysis or taxonomy analysis, is a way to create groups of objects, or *clusters*, in such a way that the profiles of objects in the same cluster are very similar and the profiles of objects in different clusters are quite distinct.

Cluster analysis can be performed on many different types of data sets. For example, a data set might contain a number of observations of subjects in a study where each observation contains a set of variables.

Many different fields of study, such as engineering, zoology, medicine, linguistics, anthropology, psychology, and marketing, have contributed to the development of clustering techniques and the application of such techniques. For example, cluster analysis can help in creating “balanced” treatment and control groups for a designed study. If you find that each cluster contains roughly equal numbers of treatment and control subjects, then statistical differences found between the groups can be attributed to the experiment and not to any initial difference between the groups.

This section explores two kinds of clustering:

- “Hierarchical Clustering” on page 7-30
- “K-Means Clustering” on page 7-46

### Hierarchical Clustering

Hierarchical clustering is a way to investigate grouping in your data, simultaneously over a variety of scales, by creating a cluster tree. The tree is not a single set of clusters, but rather a multilevel hierarchy, where clusters at one level are joined as clusters at the next higher level. This allows you to decide what level or scale of clustering is most appropriate in your application.

The following sections explore the hierarchical clustering features in the Statistics Toolbox:

- “Terminology and Basic Procedure” on page 7-31
- “Finding the Similarities Between Objects” on page 7-31
- “Defining the Links Between Objects” on page 7-34
- “Evaluating Cluster Formation” on page 7-36
- “Creating Clusters” on page 7-42

## Terminology and Basic Procedure

To perform hierarchical cluster analysis on a data set using the Statistics Toolbox functions, follow this procedure:

- 1 Find the similarity or dissimilarity between every pair of objects in the data set.** In this step, you calculate the *distance* between objects using the `pdist` function. The `pdist` function supports many different ways to compute this measurement. See “Finding the Similarities Between Objects” on page 7-31 for more information.
- 2 Group the objects into a binary, hierarchical cluster tree.** In this step, you link pairs of objects that are in close proximity using the `linkage` function. The `linkage` function uses the distance information generated in step 1 to determine the proximity of objects to each other. As objects are paired into binary clusters, the newly formed clusters are grouped into larger clusters until a hierarchical tree is formed. See “Defining the Links Between Objects” on page 7-34 for more information.
- 3 Determine where to cut the hierarchical tree into clusters.** In this step, you use the `cluster` function to prune branches off the bottom of the hierarchical tree, and assign all the objects below each cut to a single cluster. This creates a partition of the data. The `cluster` function can create these clusters by detecting natural groupings in the hierarchical tree or by cutting off the hierarchical tree at an arbitrary point. See “Creating Clusters” on page 7-42 for more information.

The following sections provide more information about each of these steps.

---

**Note** The Statistics Toolbox includes a convenience function, `clusterdata`, which performs all these steps for you. You do not need to execute the `pdist`, `linkage`, or `cluster` functions separately.

---

## Finding the Similarities Between Objects

You use the `pdist` function to calculate the distance between every pair of objects in a data set. For a data set made up of  $m$  objects, there are  $m \cdot (m - 1)/2$  pairs in the data set. The result of this computation is commonly known as a distance or dissimilarity matrix.

There are many ways to calculate this distance information. By default, the `pdist` function calculates the Euclidean distance between objects; however, you can specify one of several other options. See `pdist` for more information.

---

**Note** You can optionally normalize the values in the data set before calculating the distance information. In a real world data set, variables can be measured against different scales. For example, one variable can measure Intelligence Quotient (IQ) test scores and another variable can measure head circumference. These discrepancies can distort the proximity calculations. Using the `zscore` function, you can convert all the values in the data set to use the same proportional scale. See `zscore` for more information.

---

For example, consider a data set,  $X$ , made up of five objects where each object is a set of  $x,y$  coordinates.

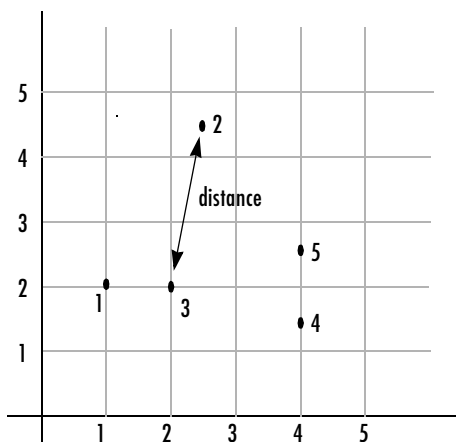
- **Object 1:** 1, 2
- **Object 2:** 2.5, 4.5
- **Object 3:** 2, 2
- **Object 4:** 4, 1.5
- **Object 5:** 4, 2.5

You can define this data set as a matrix

$$X = [1 \ 2; 2.5 \ 4.5; 2 \ 2; 4 \ 1.5; 4 \ 2.5]$$

and pass it to `pdist`. The `pdist` function calculates the distance between object 1 and object 2, object 1 and object 3, and so on until the distances between all the pairs have been calculated. The following figure plots these objects in a graph. The Euclidean distance between object 2 and object 3 is shown to illustrate one interpretation of distance.





**Returning Distance Information.** The `pdist` function returns this distance information in a vector, `Y`, where each element contains the distance between a pair of objects.

```
Y = pdist(X)
```

```
Y =
```

```
Columns 1 through 7
    2.9155    1.0000    3.0414    3.0414    2.5495    3.3541    2.5000
Columns 8 through 10
    2.0616    2.0616    1.0000
```

To make it easier to see the relationship between the distance information generated by `pdist` and the objects in the original data set, you can reformat the distance vector into a matrix using the `squareform` function. In this matrix, element  $i,j$  corresponds to the distance between object  $i$  and object  $j$  in the original data set. In the following example, element 1,1 represents the distance between object 1 and itself (which is zero). Element 1,2 represents the distance between object 1 and object 2, and so on.

```
squareform(Y)
```

```
ans =
```

```
    0    2.9155    1.0000    3.0414    3.0414
    2.9155    0    2.5495    3.3541    2.5000
    1.0000    2.5495    0    2.0616    2.0616
```

```

3.0414    3.3541    2.0616         0    1.0000
3.0414    2.5000    2.0616    1.0000         0

```

### Defining the Links Between Objects

Once the proximity between objects in the data set has been computed, you can determine how objects in the data set should be grouped into clusters, using the linkage function. The linkage function takes the distance information generated by `pdist` and links pairs of objects that are close together into binary clusters (clusters made up of two objects). The linkage function then links these newly formed clusters to each other and to other objects to create bigger clusters until all the objects in the original data set are linked together in a hierarchical tree.

For example, given the distance vector `Y` generated by `pdist` from the sample data set of  $x$ - and  $y$ -coordinates, the linkage function generates a hierarchical cluster tree, returning the linkage information in a matrix, `Z`.

```

Z = linkage(Y)
Z =

    4.0000    5.0000    1.0000
    1.0000    3.0000    1.0000
    6.0000    7.0000    2.0616
    2.0000    8.0000    2.5000

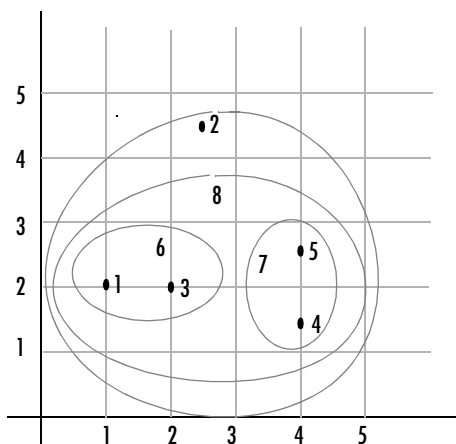
```

In this output, each row identifies a link between objects or clusters. The first two columns identify the objects that have been linked, that is, object 1, object 2, and so on. The third column contains the distance between these objects. For the sample data set of  $x$ - and  $y$ -coordinates, the linkage function begins by grouping objects 1 and 3, which have the closest proximity (distance value = 1.0000). The linkage function continues by grouping objects 4 and 5, which also have a distance value of 1.0000.

The third row indicates that the linkage function grouped objects 6 and 7. If the original sample data set contained only five objects, what are objects 6 and 7? Object 6 is the newly formed binary cluster created by the grouping of objects 1 and 3. When the linkage function groups two objects into a new cluster, it must assign the cluster a unique index value, starting with the value  $m+1$ , where  $m$  is the number of objects in the original data set. (Values 1 through  $m$  are already used by the original data set.) Similarly, object 7 is the cluster formed by grouping objects 4 and 5.

linkage uses distances to determine the order in which it clusters objects. The distance vector  $Y$  contains the distances between the original objects 1 through 5. But linkage must also be able to determine distances involving clusters that it creates, such as objects 6 and 7. By default, linkage uses a method known as single linkage. However, there are a number of different methods available. See the linkage reference page for more information.

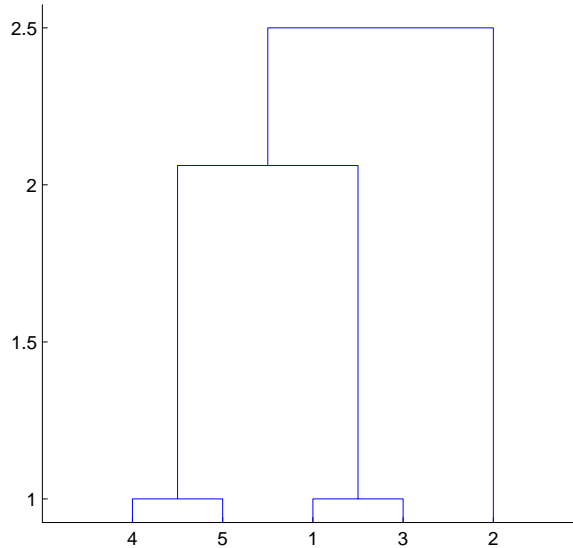
As the final cluster, the linkage function grouped object 8, the newly formed cluster made up of objects 6 and 7, with object 2 from the original data set. The following figure graphically illustrates the way linkage groups the objects into a hierarchy of clusters.



### Plotting the Cluster Tree

The hierarchical, binary cluster tree created by the linkage function is most easily understood when viewed graphically. The Statistics Toolbox includes the dendrogram function that plots this hierarchical tree information as a graph, as in the following example.

```
dendrogram(Z)
```



In the figure, the numbers along the horizontal axis represent the indices of the objects in the original data set. The links between objects are represented as upside-down U-shaped lines. The height of the U indicates the distance between the objects. For example, the link representing the cluster containing objects 1 and 3 has a height of 1. The link representing the cluster that groups object 2 together with objects 1, 3, 4, and 5, (which are already clustered as object 8) has a height of 2.5. The height represents the distance linkage computes between objects 2 and 8. For more information about creating a dendrogram diagram, see the dendrogram reference page.

### Evaluating Cluster Formation

After linking the objects in a data set into a hierarchical cluster tree, you might want to verify that the distances (that is, heights) in the tree reflect the original distances accurately. In addition, you might want to investigate natural divisions that exist among links between objects. The Statistics Toolbox provides functions to perform both these tasks, as described in the following sections:

- “Verifying the Cluster Tree” on page 7-37
- “Getting More Information About Cluster Links” on page 7-38

**Verifying the Cluster Tree.** In a hierarchical cluster tree, any two objects in the original data set are eventually linked together at some level. The height of the link represents the distance between the two clusters that contain those two objects. This height is known as the *cophenetic distance* between the two objects. One way to measure how well the cluster tree generated by the linkage function reflects your data is to compare the cophenetic distances with the original distance data generated by the `pdist` function. If the clustering is valid, the linking of objects in the cluster tree should have a strong correlation with the distances between objects in the distance vector. The `cophenet` function compares these two sets of values and computes their correlation, returning a value called the *cophenetic correlation coefficient*. The closer the value of the cophenetic correlation coefficient is to 1, the more accurately the clustering solution reflects your data.

You can use the cophenetic correlation coefficient to compare the results of clustering the same data set using different distance calculation methods or clustering algorithms. For example, you can use the `cophenet` function to evaluate the clusters created for the sample data set

```
c = cophenet(Z,Y)
```

```
c =
```

```
0.8615
```

where `Z` is the matrix output by the linkage function and `Y` is the distance vector output by the `pdist` function.

Execute `pdist` again on the same data set, this time specifying the city block metric. After running the linkage function on this new `pdist` output using the average linkage method, call `cophenet` to evaluate the clustering solution.

```
Y = pdist(X,'cityblock');
```

```
Z = linkage(Y,'average');
```

```
c = cophenet(Z,Y)
```

```
c =
```

```
0.9044
```

The cophenetic correlation coefficient shows that using a different distance and linkage method creates a tree that represents the original distances slightly better.

**Getting More Information About Cluster Links.** One way to determine the natural cluster divisions in a data set is to compare the height of each link in a cluster tree with the heights of neighboring links below it in the tree.

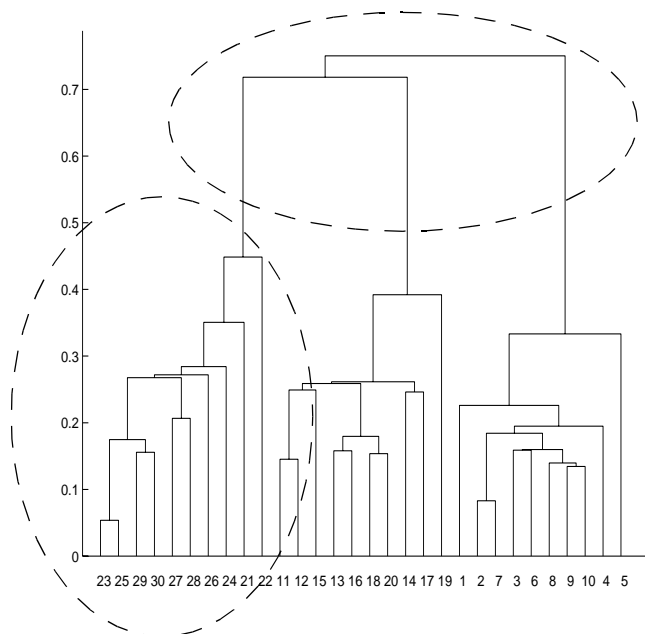
A link that is approximately the same height as the links below it indicates that there are no distinct divisions between the objects joined at this level of the hierarchy. These links are said to exhibit a high level of consistency, because the distance between the objects being joined is approximately the same as the distances between the objects they contain.

On the other hand, a link whose height differs noticeably from the height of the links below it indicates that the objects joined at this level in the cluster tree are much farther apart from each other than their components were when they were joined. This link is said to be inconsistent with the links below it.

In cluster analysis, inconsistent links can indicate the border of a natural division in a data set. The `cluster` function uses a quantitative measure of inconsistency to determine where to partition your data set into clusters. (See “Creating Clusters” on page 7-42 for more information.)

The following dendrogram, created using a data set of random numbers, illustrates inconsistent links. Note how the objects in the dendrogram fall into three 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 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 low inconsistency coefficient; links that join indistinct clusters have a high inconsistency coefficient.

To generate a listing of the inconsistency coefficient for each link in the cluster tree, use the `inconsistent` function. By default, the `inconsistent` function compares each link in the cluster hierarchy with adjacent links that are less than two levels below it in the cluster hierarchy. This is called the *depth* of the comparison. You can also specify other depths. The objects at the bottom of the cluster tree, called leaf nodes, that have no further objects below them, have an inconsistency coefficient of zero. Clusters that join two leaves also have a zero inconsistency coefficient.

For example, you can use the `inconsistent` function to calculate the inconsistency values for the links created by the `linkage` function in “Defining the Links Between Objects” on page 7-34.

```
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.

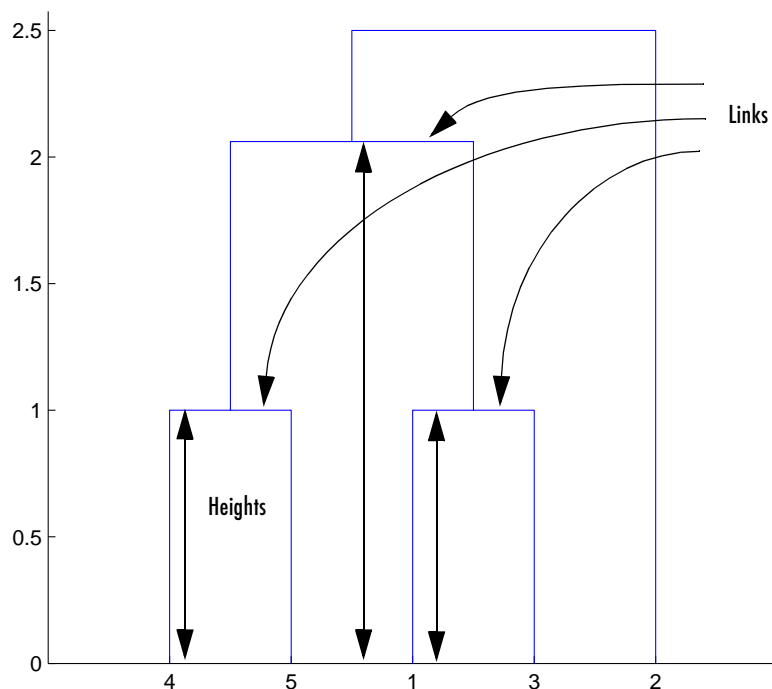
Column	Description
1	Mean of the heights of all the links included in the calculation
2	Standard deviation of all the links included in the calculation
3	Number of links included in the calculation
4	Inconsistency coefficient

In the sample output, the first row represents the link between objects 1 and 3. (This cluster is assigned the index 6 by the `linkage` function.) Because they are both leaf nodes, the inconsistency coefficient for the cluster is zero. The second row represents the link between objects 4 and 5, 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.

The following figure illustrates the links and heights included in this calculation.





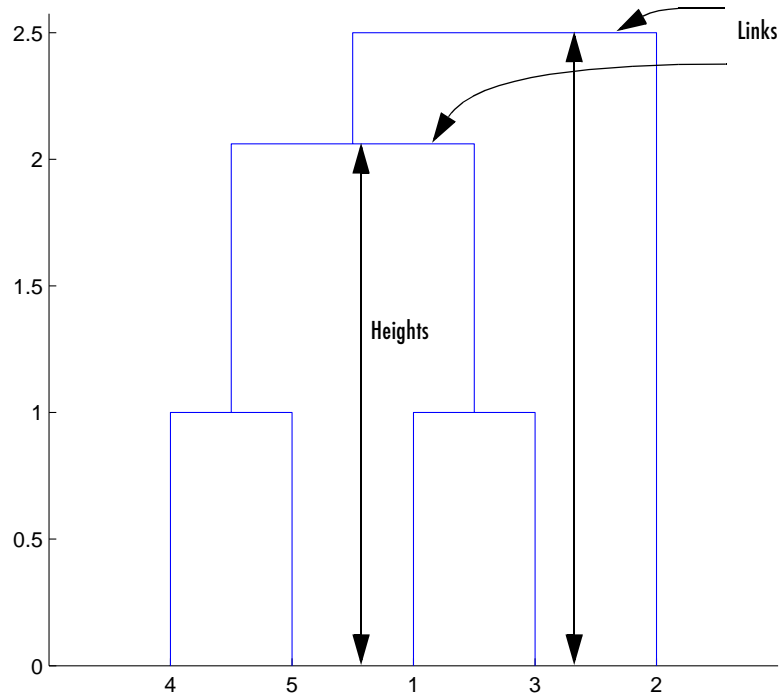

---

**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 7-43
- “Specifying Arbitrary Clusters” on page 7-44

**Finding Natural Divisions in Data.** The hierarchical cluster tree may naturally divide the data into distinct, well-separated clusters. This can be particularly evident in a dendrogram diagram created from data where groups of objects are densely packed in certain areas and not in others. The inconsistency coefficient of the links in the cluster tree can identify these divisions where the similarities between objects change abruptly. (See “Evaluating Cluster Formation” on page 7-36 for more information about the inconsistency coefficient.) You can use this value to determine where the cluster function creates cluster boundaries.

For example, if you use the `cluster` function to group the sample data set into clusters, specifying an inconsistency coefficient threshold of 1.2 as the value of the `cutoff` argument, the `cluster` function groups all the objects in the sample data set into one cluster. In this case, none of the links in the cluster hierarchy had an inconsistency coefficient greater than 1.2.

```
T = cluster(Z, 'cutoff', 1.2)
T =
     1
     1
     1
     1
     1
```

The `cluster` function outputs a vector, `T`, that is the same size as the original data set. Each element in this vector contains the number of the cluster into which the corresponding object from the original data set was placed.

If you lower the inconsistency coefficient threshold to 0.8, the `cluster` function divides the sample data set into three separate clusters.

```
T = cluster(Z, 'cutoff', 0.8)
T =
     1
     3
     1
     2
     2
```

This output indicates that objects 1 and 3 were placed in cluster 1, objects 4 and 5 were placed in cluster 2, and object 2 was placed in cluster 3.

**Specifying Arbitrary Clusters.** Instead of letting the `cluster` function create clusters determined by the natural divisions in the data set, you can specify the number of clusters you want created.

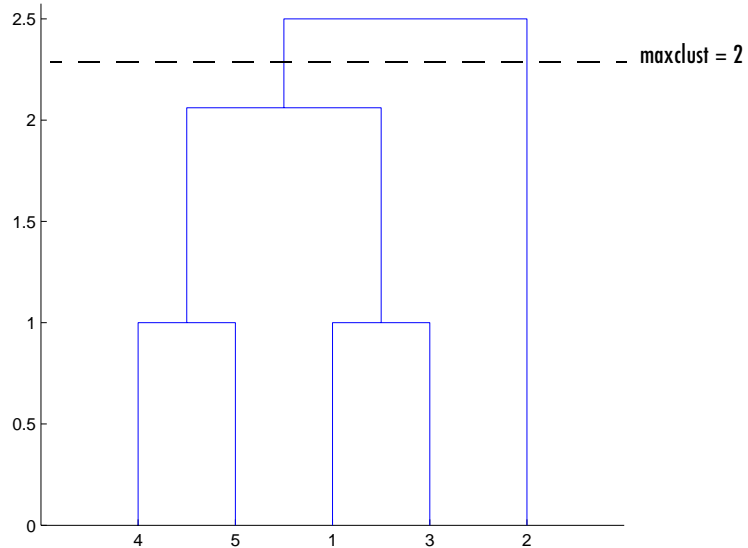
For example, you can specify that you want the `cluster` function to partition the sample data set into two clusters. In this case, the `cluster` function creates one cluster containing objects 1, 3, 4, and 5 and another cluster containing object 2.

```
T = cluster(Z, 'maxclust', 2)
```

```
T =
```

```
    2  
    1  
    2  
    2  
    2
```

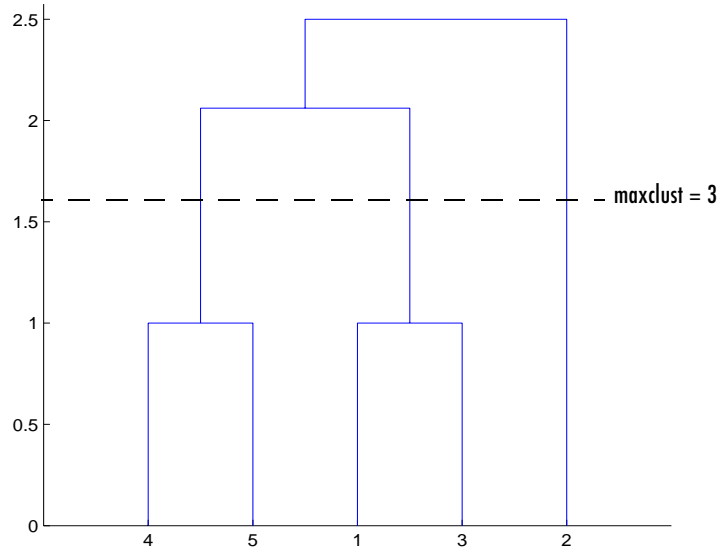
To help you visualize how the `cluster` function determines these clusters, the following figure shows the dendrogram of the hierarchical cluster tree. The horizontal dashed line intersects two lines of the dendrogram, corresponding to setting `'maxclust'` to 2. These two lines partition the objects into two clusters: the objects below the left-hand line, namely 1, 3, 4, and 5, belong to one cluster, while the object below the right-hand line, namely 2, belongs to the other cluster.



On the other hand, if you set 'maxclust' to 3, the cluster function groups objects 4 and 5 in one cluster, objects 1 and 3 in a second cluster, and object 2 in a third cluster. The following command illustrates this.

```
T = cluster(Z, 'maxclust', 3)
T =
     1
     3
     1
     2
     2
```

This time, the `cluster` function cuts off the hierarchy at a lower point, corresponding to the horizontal line that intersects three lines of the dendrogram in the following figure.



## K-Means Clustering

This section gives a description and an example of using the MATLAB function for K-means clustering, `kmeans`.

- “Overview of K-Means Clustering” on page 7-46
- “Example: Clustering Data in Four Dimensions” on page 7-47

### Overview of K-Means Clustering

K-means clustering can best be described as a partitioning method. That is, the function `kmeans` partitions the observations in your data into  $K$  mutually exclusive clusters, and returns a vector of indices indicating to which of the  $k$  clusters it has assigned each observation. Unlike the hierarchical clustering methods used in `linkage` (see “Hierarchical Clustering” on page 7-30), `kmeans` does not create a tree structure to describe the groupings in your data, but rather creates a single level of clusters. Another difference is that K-means

clustering uses the actual observations of objects or individuals in your data, and not just their proximities. These differences often mean that `kmeans` is more suitable for clustering large amounts of data.

`kmeans` treats each observation in your data as an object having a location in space. It finds a partition in which objects within each cluster are as close to each other as possible, and as far from objects in other clusters as possible. You can choose from five different distance measures, depending on the kind of data you are clustering.

Each cluster in the partition is defined by its member objects and by its centroid, or center. The centroid for each cluster is the point to which the sum of distances from all objects in that cluster is minimized. `kmeans` computes cluster centroids differently for each distance measure, to minimize the sum with respect to the measure that you specify.

`kmeans` uses an iterative algorithm that minimizes the sum of distances from each object to its cluster centroid, over all clusters. This algorithm moves objects between clusters until the sum cannot be decreased further. The result is a set of clusters that are as compact and well-separated as possible. You can control the details of the minimization using several optional input parameters to `kmeans`, including ones for the initial values of the cluster centroids, and for the maximum number of iterations.

### **Example: Clustering Data in Four Dimensions**

This example explores possible clustering in four-dimensional data by analyzing the results of partitioning the points into three, four, and five clusters.

---

**Note** Because each part of this example generates random numbers sequentially, i.e., without setting a new state, you must perform all steps in sequence to duplicate the results shown. If you perform the steps out of sequence, the answers will be essentially the same, but the intermediate results, number of iterations, or ordering of the silhouette plots may differ. See “Random Number Generators in the Statistics Toolbox” on page 1-5 to set the correct seed.

---

**Creating Clusters and Determining Separation.** First, load some data.

```
load kmeansdata;  
size(X)  
  
ans =  
    560     4
```

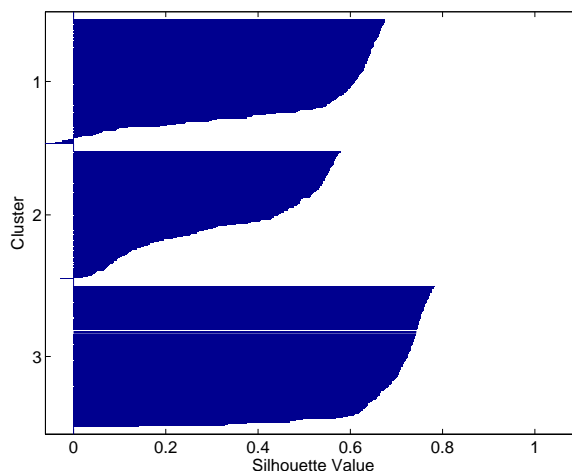
Even though these data are four-dimensional, and cannot be easily visualized, `kmeans` enables you to investigate whether a group structure exists in them. Call `kmeans` with `k`, the desired number of clusters, equal to 3. For this example, specify the city block distance measure, and use the default starting method of initializing centroids from randomly selected data points.

```
idx3 = kmeans(X,3,'distance','city');
```

To get an idea of how well-separated the resulting clusters are, you can make a silhouette plot using the cluster indices output from `kmeans`. The silhouette plot displays a measure of how close each point in one cluster is to points in the neighboring clusters. This measure ranges from +1, indicating points that are very distant from neighboring clusters, through 0, indicating points that are not distinctly in one cluster or another, to -1, indicating points that are probably assigned to the wrong cluster. `silhouette` returns these values in its first output.

```
[silh3,h] = silhouette(X,idx3,'city');  
xlabel('Silhouette Value')  
ylabel('Cluster')
```





From the silhouette plot, you can see that most points in the third cluster have a large silhouette value, greater than 0.6, indicating that the cluster is somewhat separated from neighboring clusters. However, the second cluster contains many points with low silhouette values, and the first contains a few points with negative values, indicating that those two clusters are not well separated.

**Determining the Correct Number of Clusters.** Increase the number of clusters to see if kmeans can find a better grouping of the data. This time, use the optional 'display' parameter to print information about each iteration.

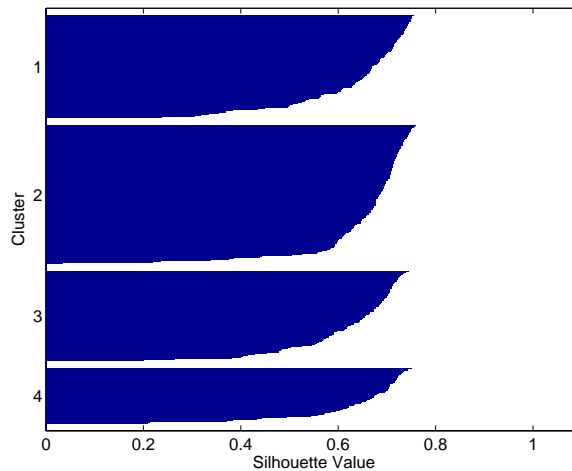
```
idx4 = kmeans(X,4, 'dist','city', 'display','iter');
iter phase    num      sum
   1     1     560    2897.56
   2     1     53    2736.67
   3     1     50    2476.78
   4     1    102    1779.68
   5     1     5    1771.1
   6     2     0    1771.1
6 iterations, total sum of distances = 1771.1
```

Notice that the total sum of distances decreases at each iteration as kmeans reassigns points between clusters and recomputes cluster centroids. In this case, the second phase of the algorithm did not make any reassignments,

indicating that the first phase reached a minimum after five iterations. In some problems, the first phase might not reach a minimum, but the second phase always will.

A silhouette plot for this solution indicates that these four clusters are better separated than the three in the previous solution.

```
[silh4,h] = silhouette(X,idx4,'city');
xlabel('Silhouette Value')
ylabel('Cluster')
```



A more quantitative way to compare the two solutions is to look at the average silhouette values for the two cases.

```
mean(silh3)
ans =
    0.52594
```

```
mean(silh4)
ans =
    0.63997
```

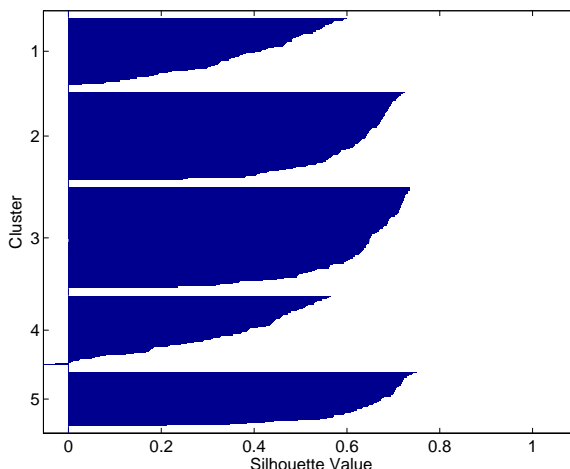
Finally, try clustering the data using five clusters.

```
idx5 = kmeans(X,5,'dist','city','replicates',5);
[silh5,h] = silhouette(X,idx5,'city');
```

```

xlabel('Silhouette Value')
ylabel('Cluster')
mean(silh5)
ans =
    0.52657

```



This silhouette plot indicates that this is probably not the right number of clusters, since two of the clusters contain points with mostly low silhouette values. Without some knowledge of how many clusters are really in the data, it is a good idea to experiment with a range of values for  $k$ .

**Avoiding Local Minima.** Like many other types of numerical minimizations, the solution that `kmeans` reaches often depends on the starting points. It is possible for `kmeans` to reach a local minimum, where reassigning any one point to a new cluster would increase the total sum of point-to-centroid distances, but where a better solution does exist. However, you can use the optional `'replicates'` parameter to overcome that problem.

For four clusters, specify five replicates, and use the `'display'` parameter to print out the final sum of distances for each of the solutions.

```

[idx4,cent4,sumdist] = kmeans(X,4,'dist','city',...
                             'display','final','replicates',5);
17 iterations, total sum of distances = 2303.36
5 iterations, total sum of distances = 1771.1

```

```
6 iterations, total sum of distances = 1771.1
5 iterations, total sum of distances = 1771.1
8 iterations, total sum of distances = 2303.36
```

The output shows that, even for this relatively simple problem, nonglobal minima do exist. Each of these five replicates began from a different randomly selected set of initial centroids, and `kmeans` found two different local minima. However, the final solution that `kmeans` returns is the one with the lowest total sum of distances, over all replicates.

```
sum(sumdist)
ans =
    1771.1
```

## Multidimensional Scaling

The following sections explain how to perform multidimensional scaling, using the functions `cmdscale` and `mdscale`:

- “Overview” on page 7-53
- “Classical Multidimensional Scaling” on page 7-53
- “Nonclassical Metric Multidimensional Scaling” on page 7-56
- “Nonmetric Multidimensional Scaling” on page 7-58
- “Example — Reconstructing a Map from Intercity Distances” on page 7-60

### Overview

One of the most important goals in visualizing data is to get a sense of how near or far points are from each other. Often, you can do this with a scatter plot. However, for some analyses, the data that you have might not be in the form of points at all, but rather in the form of pairwise similarities or dissimilarities between cases, observations, or subjects. Without any points, you cannot make a scatter plot.

Even if your data are in the form of points rather than pairwise distances, a scatter plot of those data might not be useful. For some kinds of data, the relevant way to measure how “near” two points are might not be their Euclidean distance. While scatter plots of the raw data make it easy to compare Euclidean distances, they are not always useful when comparing other kinds of interpoint distances, city block distance for example, or even more general dissimilarities. Also, with a large number of variables, it is very difficult to visualize distances unless the data can be represented in a small number of dimensions. Some sort of dimension reduction is usually necessary.

Multidimensional scaling (MDS) is a set of methods that address all these problems. MDS allows you to visualize how near points are to each other for many kinds of distance or dissimilarity measures and can produce a representation of your data in a small number of dimensions. MDS does not require raw data, but only a matrix of pairwise distances or dissimilarities.

### Classical Multidimensional Scaling

The function `cmdscale` performs classical (metric) multidimensional scaling, also known as principal coordinates analysis. `cmdscale` takes as an input a

matrix of interpoint distances and creates a configuration of points. Ideally, those points are in two or three dimensions, and the Euclidean distances between them reproduce the original distance matrix. Thus, a scatter plot of the points created by `cmdscale` provides a visual representation of the original distances.

### A Simple Example

As a very simple example, you can reconstruct a set of points from only their interpoint 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 interpoint 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 roundoff error) eigenvalues, the columns of `Y` corresponding to the positive eigenvalues provide an exact reconstruction of `D`, in the sense that their interpoint Euclidean distances, computed using `pdist`, for example, are identical (within roundoff) 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 interpoint Euclidean distances are given by `D`. If the largest negative eigenvalue is small in magnitude relative to the largest positive eigenvalues, then the configuration returned by `cmdscale` might still reproduce `D` well. “Example — Reconstructing a Map from Intercity Distances” on page 7-60 demonstrates this.

## Nonclassical Metric Multidimensional Scaling

The function `cmdscale` performs classical multidimensional scaling (MDS). The Statistics Toolbox also includes the function `mdscale` to perform non-classical MDS. As with `cmdscale`, you can use `mdscale` either to visualize dissimilarity data for which no "locations" exist, or to visualize high-dimensional data by reducing its dimensionality. Both functions take a matrix of dissimilarities as an input and produce a configuration of points. However, `mdscale` offers a choice of different criteria to construct the configuration, and allows missing data and weights.

For example, the cereal data include measurements on 10 variables describing breakfast cereals. You can use `mdscale` to visualize these data in two dimensions. First, load the data. For clarity, this example code selects a subset of 22 of the observations.

```
load cereal.mat
X = [Calories Protein Fat Sodium Fiber Carbo Sugars Shelf...
     Potass Vitamins];
X = X(strmatch('G',Mfg),:); % take a subset from a single
                             % manufacturer

size(X)
ans =
     22 10
```

Then use `pdist` to transform the 10-dimensional data into dissimilarities. The output from `pdist` is a symmetric dissimilarity matrix, stored as a vector containing only the  $(23 \cdot 22/2)$  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



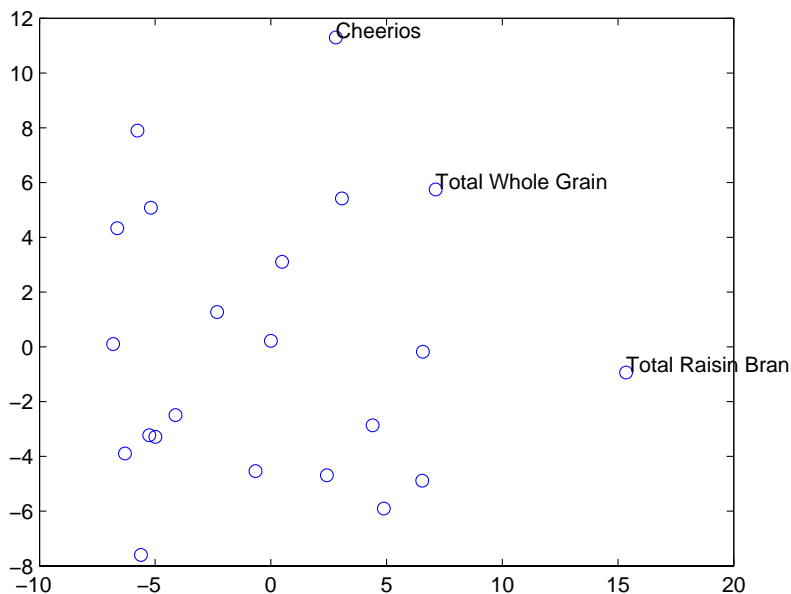
choices, see the `mdscale` reference page in the online documentation. The second output from `mdscale` is the value of that criterion evaluated for the output configuration. It measures the how well the interpoint distances of the output configuration approximate the original input dissimilarities.

```
[Y, stress] =...
mdscale(dissimilarities,2,'criterion','metricstress');
stress
stress =

    0.1856
```

A scatterplot of the output from `mdscale` represents the original 10-dimensional data in two dimensions, and you can use the `gname` function to label selected points.

```
plot(Y(:,1),Y(:,2),'o');
gname(Name(strmatch('G',Mfg)))
```



## Nonmetric Multidimensional Scaling

Metric multidimensional scaling creates a configuration of points whose interpoint 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.

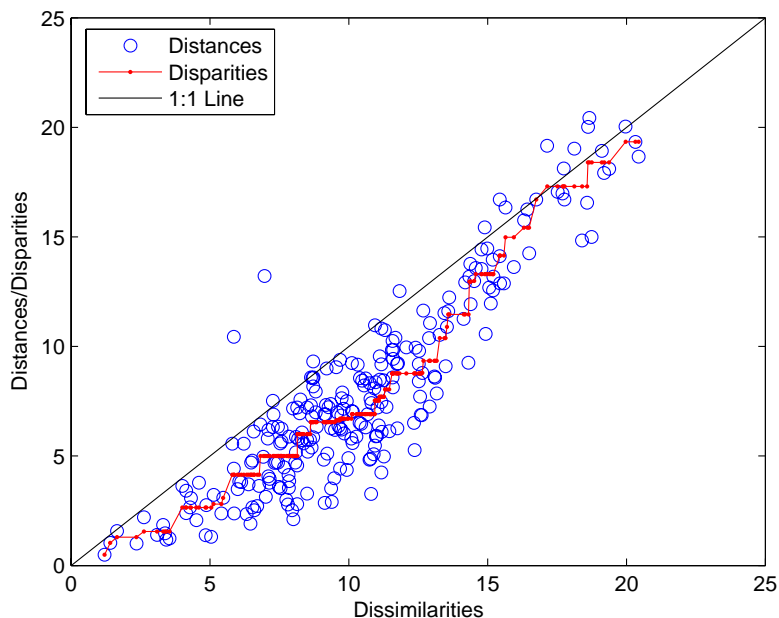
You use `mdscale` to perform nonmetric MDS in much the same way as for metric scaling. The nonmetric STRESS criterion is a common method for computing the output; for more choices, see the `mdscale` reference page in the online documentation. As with metric scaling, the second output from `mdscale` is the value of that criterion evaluated for the output configuration. For nonmetric scaling, however, it measures the how well the interpoint 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 interpoint 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 5 replicates of MDS, each starting at a different randomly-chosen initial configuration. The criterion value is printed out for each replication; `mdscale` returns the configuration with the best fit.

```

opts = statset('Display','final');
[Y, stress] =...
mdscale(dissimilarities,2,'criterion','stress',...
'start','random','replicates',5,'Options',opts);
90 iterations, Final stress criterion = 0.156209
100 iterations, Final stress criterion = 0.195546
116 iterations, Final stress criterion = 0.156209
85 iterations, Final stress criterion = 0.156209
106 iterations, Final stress criterion = 0.17121

```

Notice that `mdscale` finds several different local solutions, some of which do not have as low a stress value as the solution found with the `cmdscale` starting point.

## Example – Reconstructing a Map from Intercity Distances

Given only the distances between 10 US cities, `cmdscale` can construct a map of those cities. First, create the distance matrix and pass it to `cmdscale`. In this example, `D` is a full distance matrix: it is square and symmetric, has positive entries off the diagonal, and has zeros on the diagonal.

```

cities =
{'Atl','Chi','Den','Hou','LA','Mia','NYC','SF','Sea','WDC'};
D = [
    0  587 1212  701 1936  604  748 2139 2182  543;
    587  0  920  940 1745 1188  713 1858 1737  597;
    1212 920  0  879  831 1726 1631  949 1021 1494;
    701  940  879  0 1374  968 1420 1645 1891 1220;
    1936 1745  831 1374  0 2339 2451  347  959 2300;
    604 1188 1726  968 2339  0 1092 2594 2734  923;
    748  713 1631 1420 2451 1092  0 2571 2408  205;
    2139 1858  949 1645  347 2594 2571  0  678 2442;
    2182 1737 1021 1891  959 2734 2408  678  0 2329;
    543  597 1494 1220 2300  923  205 2442 2329  0];
[Y,eigvals] = cmdscale(D);

```

Next, look at the eigenvalues returned by `cmdscale`. Some of these are negative, indicating that the original distances are not Euclidean. This is because of the curvature of the earth.

```
format short g
```

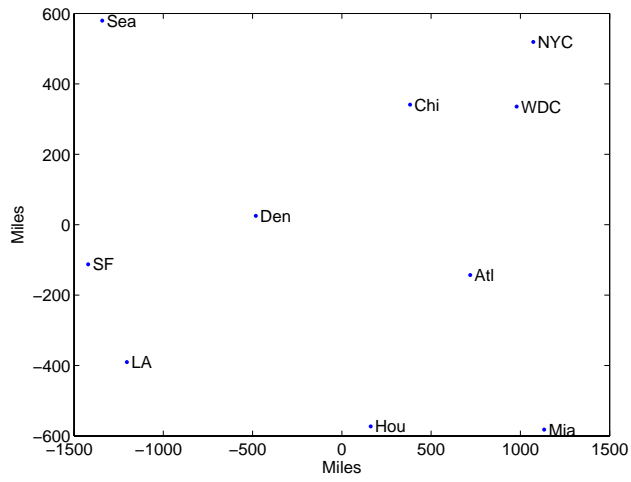
```
[eigvals eigvals/max(abs(eigvals))]
ans =
    9.5821e+006         1
    1.6868e+006     0.17604
         8157.3     0.0008513
         1432.9     0.00014954
         508.67    5.3085e-005
         25.143    2.624e-006
    5.3394e-010    5.5722e-017
        -897.7   -9.3685e-005
       -5467.6   -0.0005706
       -35479   -0.0037026
```

However, in this case, the two largest positive eigenvalues are much larger in magnitude than the remaining eigenvalues. So, despite the negative eigenvalues, the first two coordinates of  $Y$  are sufficient for a reasonable reproduction of  $D$ .

```
Dtriu = D(find(tril(ones(10),-1)))';
maxrelerr = max(abs(Dtriu - pdist(Y(:,1:2)))) ./ max(Dtriu)
maxrelerr =
    0.0075371
```

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

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



# Statistical Plots

---

Introduction (p. 8-2)

Introduces the tools for creating statistical plots.

Box Plots (p. 8-3)

Explains how to create box plots.

Distribution Plots (p. 8-4)

Explains how to create distribution plots.

Scatter Plots (p. 8-10)

Explains how to create scatter plots.

## Introduction

The Statistics Toolbox adds specialized plots to the extensive graphics capabilities of MATLAB:

- *Box plots* are graphs for describing data samples. They are also useful for graphic comparisons of the means of many samples (see “One-Way Analysis of Variance (ANOVA)” on page 4-3).
- *Distribution plots* are graphs for visualizing the distribution of one or more samples. They include *normal and Weibull probability plots*, *quantile-quantile plots*, and *empirical cumulative distribution plots*.
- *Scatter plots* are graphs for visualizing the relationship between a pair of variables or several such pairs. Grouped versions of these plots use different plotting symbols to indicate group membership. The `gname` function can label points on these plots with a text label or an observation number.

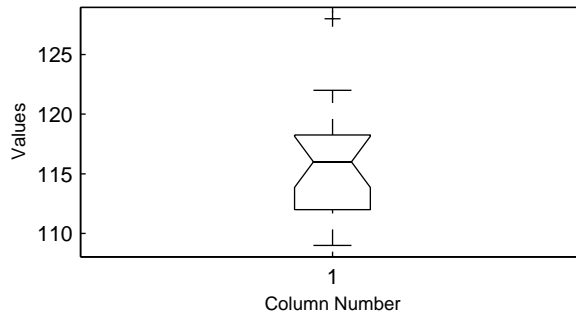
The plot types are described further in the following sections:

- “Box Plots” on page 8-3
- “Distribution Plots” on page 8-4
- “Scatter Plots” on page 8-10



## Box Plots

The graph shows an example of a notched box plot.



This plot has several graphic elements:

- The lower and upper lines of the “box” are the 25th and 75th percentiles of the sample. The distance between the top and bottom of the box is the interquartile range.
- The line in the middle of the box is the sample median. If the median is not centered in the box, that is an indication of skewness.
- The “whiskers” are lines extending above and below the box. They show the extent of the rest of the sample (unless there are outliers). Assuming no outliers, the maximum of the sample is the top of the upper whisker. The minimum of the sample is the bottom of the lower whisker. 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.
- The plus sign at the top of the plot is an indication of an outlier in the data. This point may be the result of a data entry error, a poor measurement or a change in the system that generated the data.
- The notches in the box are a graphic confidence interval about the median of a sample. Box plots do not have notches by default.

A side-by-side comparison of two notched box plots is the graphical equivalent of a t-test. See Chapter 6, “Hypothesis Tests.”

## Distribution Plots

There are several types of plots for examining the distribution of one or more samples, as described in the following sections:

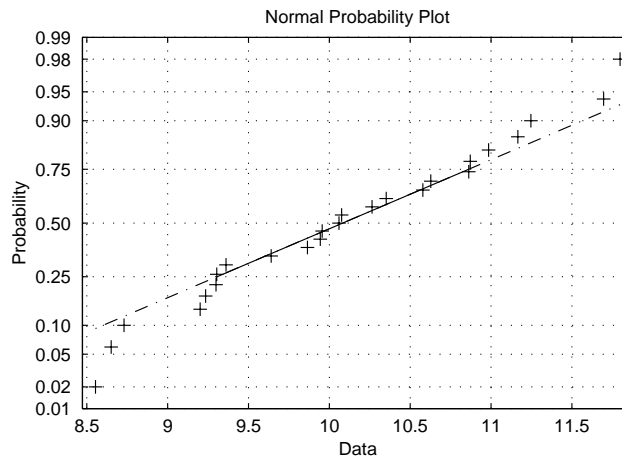
- “Normal Probability Plots” on page 8-4
- “Quantile-Quantile Plots” on page 8-6
- “Weibull Probability Plots” on page 8-7
- “Empirical Cumulative Distribution Function (CDF)” on page 8-8

### Normal Probability Plots

A normal probability plot is a useful graph for assessing whether data comes from a normal distribution. Many statistical procedures make the assumption that the underlying distribution of the data is normal, so this plot can provide some assurance that the assumption of normality is not being violated, or provide an early warning of a problem with your assumptions.

This example shows a typical normal probability plot.

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

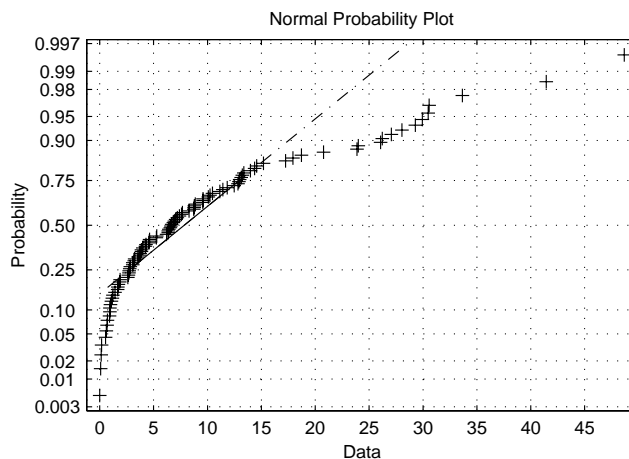


The plot has three graphical elements. The plus signs show the empirical probability versus the data value for each point in the sample. The solid line connects the 25th and 75th percentiles of the data and represents a robust linear fit (i.e., insensitive to the extremes of the sample). The dashed line extends the solid line to the ends of the sample.

The scale of the y-axis is not uniform. The y-axis values are probabilities and, as such, go from zero to one. The distance between the 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 moving away from the median. Compare the vertical distance from the bottom of the plot to the probability 0.25 with the distance from 0.25 to 0.50. Similarly, compare the distance from the top of the plot to the probability 0.75 with the distance from 0.75 to 0.50.

If all the data points fall near the line, the assumption of normality is reasonable. But, if the data is nonnormal, the plus signs may follow a curve, as in the example using exponential data below.

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



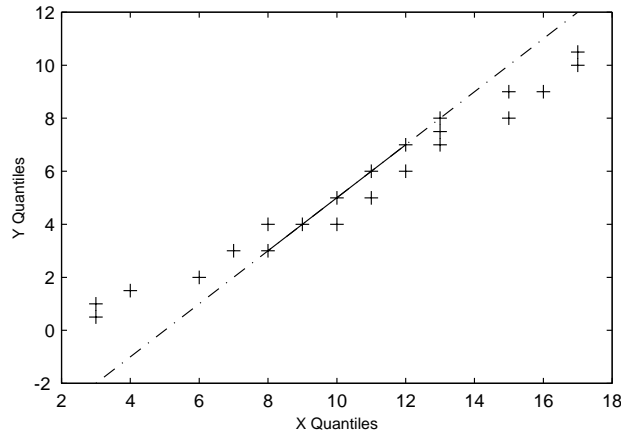
This plot is clear evidence that the underlying distribution is not normal.

## Quantile-Quantile Plots

A quantile-quantile plot is useful for determining whether two samples come from the same distribution (whether normally distributed or not).

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

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

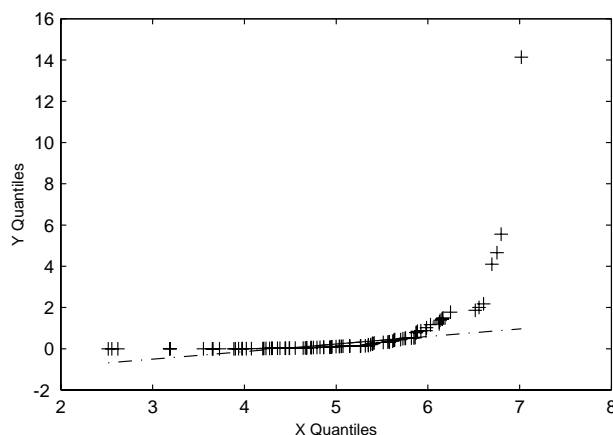


Even though the parameters and sample sizes are different, the straight line relationship shows that the two samples come from the same distribution.

Like the normal probability plot, the quantile-quantile plot has three graphical elements. The pluses are the quantiles of each sample. By default the number of pluses is the number of data values in the smaller sample. The solid line joins the 25th and 75th percentiles of the samples. The dashed line extends the solid line to the extent of the sample.

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

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



These samples clearly are not from the same distribution.

It is incorrect to interpret a linear plot as a *guarantee* that the two samples come from the same distribution. But, for assessing the validity of a statistical procedure that depends on the two samples coming from the same distribution (e.g., ANOVA), a linear quantile-quantile plot should be sufficient.

## Weibull Probability Plots

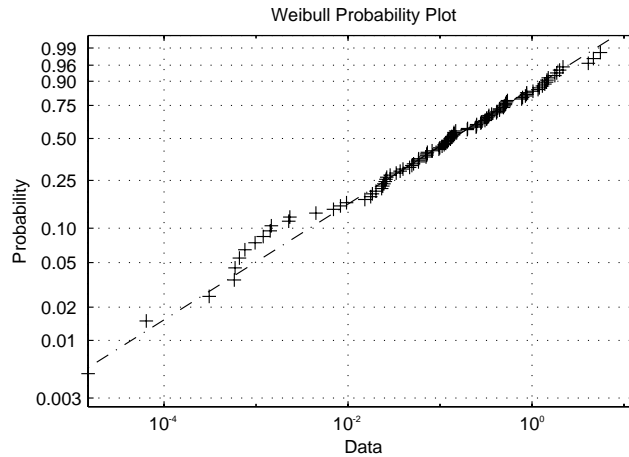
A Weibull probability plot is a useful graph for assessing whether data comes from a Weibull distribution. Many reliability analyses make the assumption that the underlying distribution of the lifetimes is Weibull, so this plot can provide some assurance that this assumption is not being violated, or provide an early warning of a problem with your assumptions.

The scale of the  $y$ -axis is not uniform. The  $y$ -axis values are probabilities and, as such, go from zero to one. The distance between the tick marks on the  $y$ -axis matches the distance between the quantiles of a Weibull distribution.

If the data points (pluses) fall near the line, the assumption that the data comes from a Weibull distribution is reasonable.

This example shows a typical Weibull probability plot.

```
y = weibrnd(2,0.5,100,1);  
weibplot(y)
```

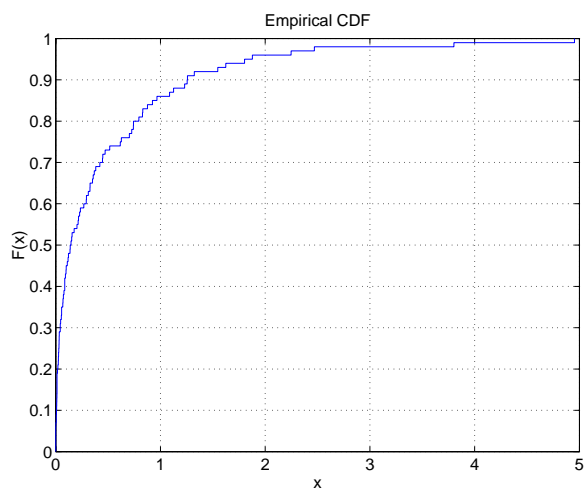


## Empirical Cumulative Distribution Function (CDF)

If you are not willing to assume that your data follows a specific probability distribution, you can use the `cdfplot` function to graph an empirical estimate of the cumulative distribution function (cdf). This function computes the proportion of data points less than each  $x$  value, and plots the proportion as a function of  $x$ . The  $y$ -axis scale is linear, not a probability scale for a specific distribution.

This example shows the empirical cumulative distribution function for a Weibull sample.

```
y = weibrnd(2,0.5,100,1);  
cdfplot(y)
```



The plot shows a probability function that rises steeply near  $x=0$  and levels off for larger values. Over 80% of the observations are less than 1, with the remaining values spread over the range [1 5].

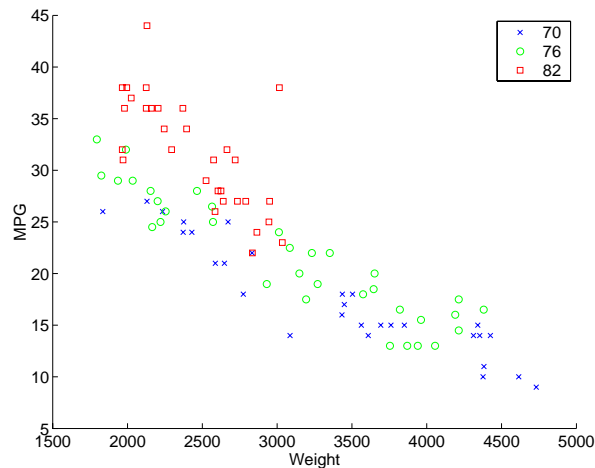
## Scatter Plots

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

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

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

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



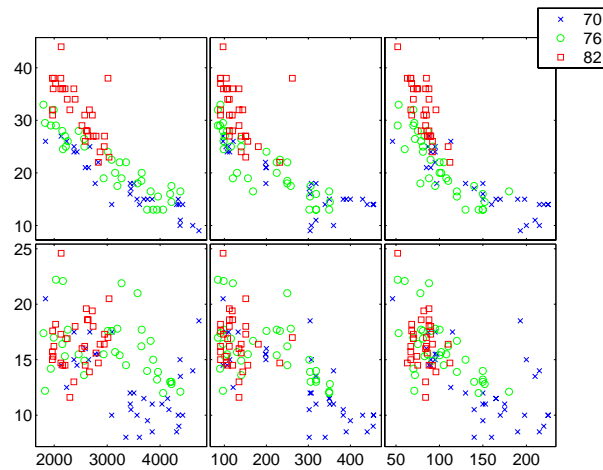
This shows that not only is there a strong relationship between the weight of a car and its mileage, but also that newer cars tend to be lighter and have better gas mileage than older cars.

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



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

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



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

The `gplotmatrix` function can also graph all pairs from a single list of variables, along with histograms for each variable. See “Multivariate Analysis of Variance (MANOVA)” on page 7-24.



# Statistical Process Control

---

Statistical process control (SPC) refers to a number of methods for assessing and monitoring the quality of manufactured goods. The following sections describe the SPC features of the Statistics Toolbox:

Control Charts (p. 9-2)

Explains how to create control charts.

Capability Studies (p. 9-5)

Describes how to perform capability studies.

## Control Charts

Control charts were popularized by Walter Shewhart in his work in the 1920s at Western Electric. A control chart is a plot of a measurements over time with statistical limits applied. Actually, *control* chart is a slight misnomer. The chart itself is really a monitoring tool. The control activity might occur if the chart indicates that the process is changing in an undesirable systematic direction.

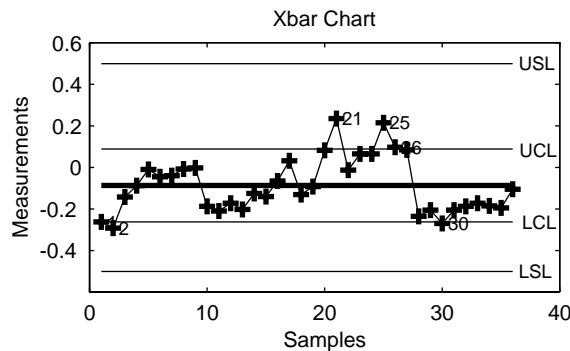
The Statistics Toolbox supports three common control charts, described in the following sections:

- “Xbar Charts” on page 9-2
- “S Charts” on page 9-3
- “EWMA Charts” on page 9-4

### Xbar Charts

Xbar charts are a plot of the average of a sample of a process taken at regular intervals. Suppose you are manufacturing pistons to a tolerance of 0.5 thousandths of an inch. You measure the runout (deviation from circularity in thousandths of an inch) at four points on each piston.

```
load parts
conf = 0.99;
spec = [-0.5 0.5];
xbarplot(runout,conf,spec)
```

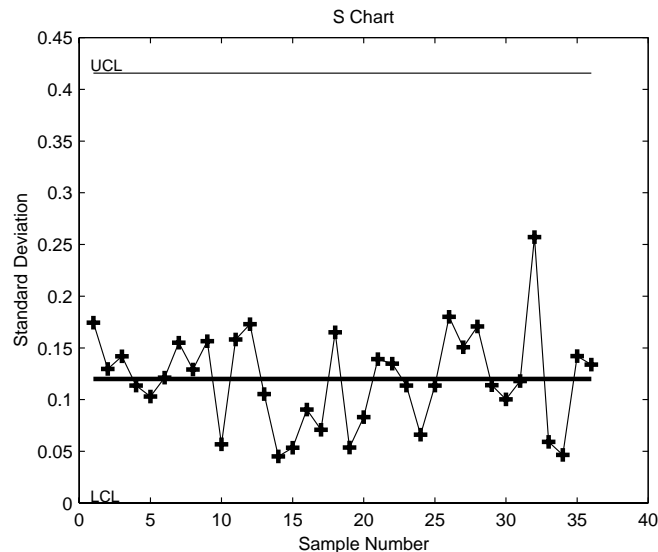


The lines at the bottom and the top of the plot show the process specifications. The central line is the average runout over all the pistons. The two lines flanking the center line are the 99% statistical control limits. By chance only one measurement in 100 should fall outside these lines. You can see that even in this small run of 36 parts, there are several points outside the boundaries (labeled by their observation numbers). This is an indication that the process mean is not in statistical control. This might not be of much concern in practice, since all the parts are well within specification.

## S Charts

The S chart is a plot of the standard deviation of a process taken at regular intervals. The standard deviation is a measure of the variability of a process. So, the plot indicates whether there is any systematic change in the process variability. Continuing with the piston manufacturing example, you can look at the standard deviation of each set of four measurements of runout.

```
schart(runout)
```



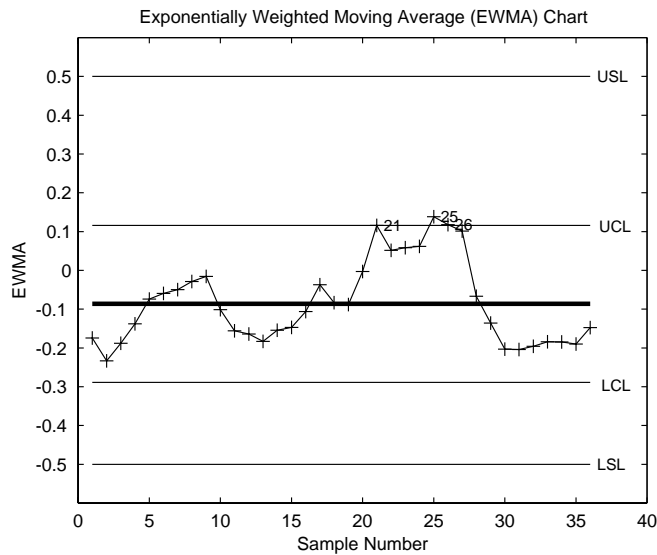
The average runout is about 0.1 thousandths of an inch. There is no indication of nonrandom variability.

## EWMA Charts

The exponentially-weighted moving average (EWMA) chart is another chart for monitoring the process average. It operates on slightly different assumptions than the Xbar chart. The mathematical model behind the Xbar chart posits that the process mean is actually constant over time and any variation in individual measurements is due entirely to chance.

The EWMA model is a little looser. Here you assume that the mean may be varying in time. Here is an EWMA chart of the runout example. Compare this with the plot in “Xbar Charts” on page 9-2.

```
ewmplot(runout,0.5,0.01,spec)
```



## Capability Studies

Before going into full-scale production, many manufacturers run a pilot study to determine whether their process can actually build parts to the specifications demanded by the engineering drawing.

Using the data from these capability studies with a statistical model enables you to get a preliminary estimate of the percentage of parts that will fall outside the specifications.

```
[p, Cp, Cpk] = capable(mean(runout), spec)
```

```
p =  
1.3940e-09
```

```
Cp =  
2.3950
```

```
Cpk =  
1.9812
```

The result above shows that the probability ( $p = 1.3940e-09$ ) of observing an unacceptable runout is extremely low. Cp and Cpk are two popular capability indices.

$C_p$  is the ratio of the range of the specifications to six times the estimate of the process standard deviation.

$$C_p = \frac{USL - LSL}{6\sigma}$$

For a process that has its average value on target, a  $C_p$  of 1 translates to a little more than one defect per thousand. Recently many industries have set a quality goal of one part per million. This would correspond to a  $C_p = 1.6$ . The higher the value of  $C_p$ , the more capable the process.

$C_{pk}$  is the ratio of difference between the process mean and the closer specification limit to three times the estimate of the process standard deviation.

$$C_{pk} = \min\left(\frac{USL - \mu}{3\sigma}, \frac{\mu - LSL}{3\sigma}\right)$$

where the process mean is  $\mu$ . For processes that do not maintain their average on target,  $C_{pk}$ , is a more descriptive index of process capability.



# Design of Experiments

---

Introduction (p. 10-2)	Introduces the topic of design of experiments.
Full Factorial Designs (p. 10-4)	Explains how to create full factorial designs.
Fractional Factorial Designs (p. 10-6)	Explains how to create fractional factorial designs.
Response Surface Designs (p. 10-8)	Explains how to create response surface designs.
D-Optimal Designs (p. 10-18)	Explains how to create D-optimal designs.

## Introduction

There is a world of difference between data and information. To extract information from data you have to make assumptions about the system that generated the data. Using these assumptions and physical theory you may be able to develop a mathematical model of the system.

Generally, even rigorously formulated models have some unknown constants. The goal of experimentation is to acquire data that enable you to estimate these constants.

But why do you need to experiment at all? You could instrument the system you want to study and just let it run. Sooner or later you would have all the data you could use.

In fact, this is a fairly common approach. There are three characteristics of historical data that pose problems for statistical modeling:

- Suppose you observe a change in the operating variables of a system followed by a change in the outputs of the system. That does *not* necessarily mean that the change in the system *caused* the change in the outputs.
- A common assumption in statistical modeling is that the observations are independent of each other. This is not the way a system in normal operation works.
- Controlling a system in operation often means changing system variables in tandem. But if two variables change together, it is impossible to separate their effects mathematically.

Designed experiments directly address these problems. The overwhelming advantage of a designed experiment is that you actively manipulate the system you are studying. With Design of Experiments (DOE) you may generate fewer data points than by using passive instrumentation, but the quality of the information you get will be higher.

The Statistics Toolbox provides several functions for generating experimental designs appropriate to various situations. These are discussed in the following sections:

- “Full Factorial Designs” on page 10-4
- “Fractional Factorial Designs” on page 10-6
- “Response Surface Designs” on page 10-8
- “D-Optimal Designs” on page 10-18

## Full Factorial Designs

Suppose you want to determine whether the variability of a machining process is due to the difference in the lathes that cut the parts or the operators who run the lathes.

If the same operator always runs a given lathe then you cannot tell whether the machine or the operator is the cause of the variation in the output. By allowing every operator to run every lathe you can separate their effects.

This is a factorial approach. `fullfact` is the function that generates the design. Suppose you have four operators and three machines. What is the factorial design?

```
d = fullfact([4 3])
```

```
d =  
  1  1  
  2  1  
  3  1  
  4  1  
  1  2  
  2  2  
  3  2  
  4  2  
  1  3  
  2  3  
  3  3  
  4  3
```

Each row of `d` represents one operator/machine combination. Note that there are  $4 \times 3 = 12$  rows.

One special subclass of factorial designs is when all the variables take only two values. Suppose you want to quickly determine the sensitivity of a process to high and low values of three variables.

```
d2 = ff2n(3)
```

```
d2 =  
  0  0  0  
  0  0  1
```

0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

There are  $2^3 = 8$  combinations to check.

## Fractional Factorial Designs

One difficulty with factorial designs is that the number of combinations increases exponentially with the number of variables you want to manipulate.

For example, the sensitivity study discussed above might be impractical if there were seven variables to study instead of just three. A full factorial design would require  $2^7 = 128$  runs!

If you assume that the variables do not act synergistically in the system, you can assess the sensitivity with far fewer runs. The theoretical minimum number is eight. A design known as the Plackett-Burman design uses a Hadamard matrix to define this minimal number of runs. To see the design (X) matrix for the Plackett-Burman design, you use the hadamard function.

$X = \text{hadamard}(8)$

X =

1	1	1	1	1	1	1	1
1	-1	1	-1	1	-1	1	-1
1	1	-1	-1	1	1	-1	-1
1	-1	-1	1	1	-1	-1	1
1	1	1	1	-1	-1	-1	-1
1	-1	1	-1	-1	1	-1	1
1	1	-1	-1	-1	-1	1	1
1	-1	-1	1	-1	1	1	-1

The last seven columns are the actual variable settings (-1 for low, 1 for high.) The first column (all ones) enables you to measure the mean effect in the linear equation,  $y = X\beta + \varepsilon$ .

The Plackett-Burman design enables you to study the main (linear) effects of each variable with a small number of runs. It does this by using a fraction, in this case 8/128, of the runs required for a full factorial design. A drawback of this design is that if the effect of one variable does vary with the value of another variable, then the estimated effects will be biased (that is, they will tend to be off by a systematic amount).

At a cost of a somewhat larger design, you can find a fractional factorial that is much smaller than a full factorial, but that does allow estimation of main effects independent of interactions between pairs of variables. You can do this by specifying generators that control the confounding between variables.

As an example, suppose you create a design with the first four variables varying independently as in a full factorial, but with the other three variables formed by multiplying different triplets of the first four. With this design the effects of the last three variables are confounded with three-way interactions among the first four variables. The estimated effect of any single variable, however, is not confounded with (is independent of) interaction effects between any pair of variables. Interaction effects are confounded with each other. Box, Hunter, and Hunter [5] present the properties of these designs and provide the generators needed to produce them.

The `fracfact` function can produce this fractional factorial design using the generator strings that Box, Hunter, and Hunter provide.

```
X = fracfact('a b c d abc bcd acd')
```

```
X =
```

```

-1  -1  -1  -1  -1  -1  -1
-1  -1  -1   1  -1   1   1
-1  -1   1  -1   1   1   1
-1  -1   1   1   1  -1  -1
-1   1  -1  -1   1   1  -1
-1   1  -1   1   1  -1   1
-1   1   1  -1  -1  -1   1
-1   1   1   1  -1   1  -1
  1  -1  -1  -1   1  -1   1
  1  -1  -1   1   1   1  -1
  1  -1   1  -1  -1   1  -1
  1  -1   1   1  -1  -1   1
  1   1  -1  -1  -1   1   1
  1   1  -1   1  -1  -1  -1
  1   1   1  -1   1  -1  -1
  1   1   1   1   1   1   1
```

## Response Surface Designs

Sometimes simple linear and interaction models are not adequate. For example, suppose that the outputs are defects or yield, and the goal is to minimize defects and maximize yield. If these optimal points are in the interior of the region in which the experiment is to be conducted, you need a mathematical model that can represent curvature so that it has a local optimum. The simplest such model has the quadratic form

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_{12} X_1 X_2 + \beta_{11} X_1^2 + \beta_{22} X_2^2$$

containing linear terms for all factors, squared terms for all factors, and products of all pairs of factors.

Designs for fitting these types of models are known as response surface designs. One such design is the full factorial design having three values for each input. Although the Statistics Toolbox is capable of generating this design, it is not really a satisfactory design in most cases because it has many more runs than are necessary to fit the model.

The two most common designs generally used in response surface modeling are central composite designs and Box-Behnken designs. In these designs the inputs take on three or five distinct values (levels), but not all combinations of these values appear in the design.

The functions described here produce specific response surface designs:

- “Central Composite Designs” on page 10-8
- “Box-Behnken Designs” on page 10-9

If these do not serve your purposes, consider creating a D-optimal design. “Design of Experiments Demo” on page 10-10 uses a D-optimal design to fit data that conforms to a response surface model. For more information see “D-Optimal Designs” on page 10-18.

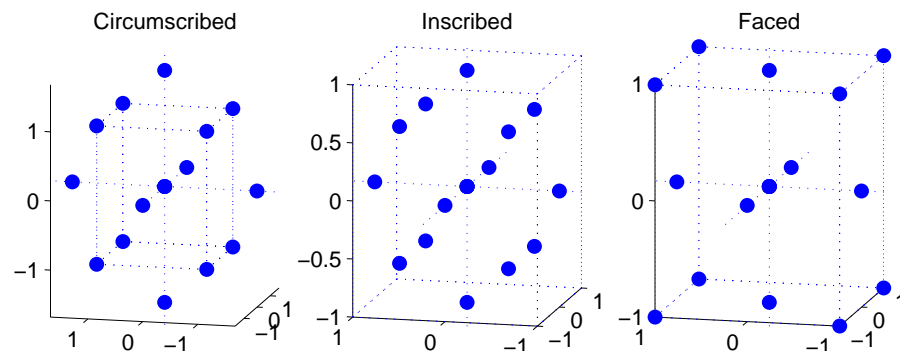
### Central Composite Designs

Central composite designs are response surface designs that can fit a full quadratic model. To picture a central composite design, imagine you have several factors that can vary between low and high values. For convenience, suppose each factor varies from -1 to +1.



One central composite design consists of cube points at the corners of a unit cube that is the product of the intervals  $[-1,1]$ , star points along the axes at or outside the cube, and center points at the origin.

Central composite designs are of three types. Circumscribed (CCC) designs are as described above. Inscribed (CCI) designs are as described above, but scaled so the star points take the values  $-1$  and  $+1$ , and the cube points lie in the interior of the cube. Faced (CCF) designs have the star points on the faces of the cube. Faced designs have three levels per factor, in contrast with the other types, which have five levels per factor. The following figure shows these three types of designs for three factors.



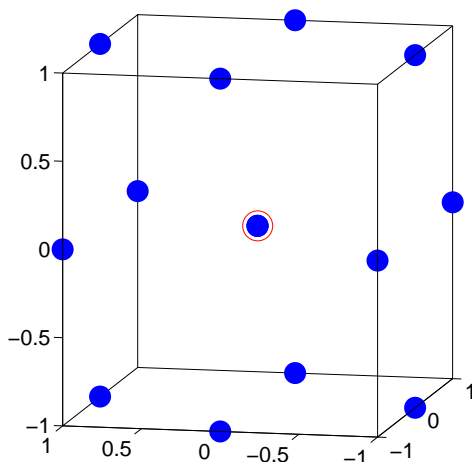
## Box-Behnken Designs

Like central composite designs, Box-Behnken designs are response surface designs that can fit a full quadratic model. Unlike most central composite designs, Box-Behnken designs use just three levels of each factor. This makes them appealing when the factors are quantitative but the set of achievable values is small.

Central composite faced (CCF) designs also use just three factor levels. However, they are not rotatable as Box-Behnken designs are. On the other hand, Box-Behnken designs can be expected to have poorer prediction ability in the corners of the cube that encloses the design, because unlike CCF designs they do not include points at the corners of that cube.

The following figure shows a Box-Behnken design for three factors, with the circled point appearing at the origin and possibly repeated for several runs. A

repeated center point makes it possible to compute an estimate of the error term that does not depend on the fitted model. For this design all points except the center point appear at a distance  $\sqrt{2}$  from the origin. That does not hold true for Box-Behnken designs with different numbers of factors.



## Design of Experiments Demo

The `rsmdemo` utility is an interactive graphic environment that demonstrates the design of experiments and surface fitting through the simulation of a chemical reaction. The goal of the demo is to find the levels of the reactants needed to maximize the reaction rate.

Suitable designs for this experiment include the central composite designs and Box-Behnken designs, described in the previous two sections, and the D-optimal designs, described in “D-Optimal Designs” on page 10-18. This demo uses D-optimal designs.

There are two parts to the demo:

- “Comparing Results from Trial-and-Error Data and a Designed Experiment” on page 10-11
- “Comparing Results Using a Polynomial Model and a Nonlinear Model” on page 10-15

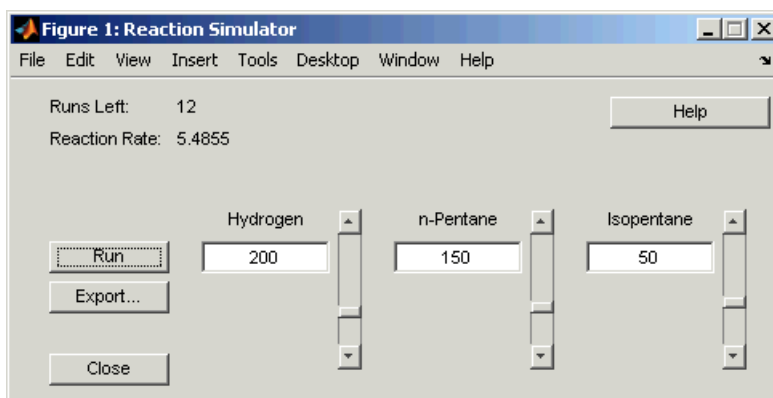
## Comparing Results from Trial-and-Error Data and a Designed Experiment

This part of the experiment compares the results obtained using data gathered through trial and error and using data from a designed experiment:

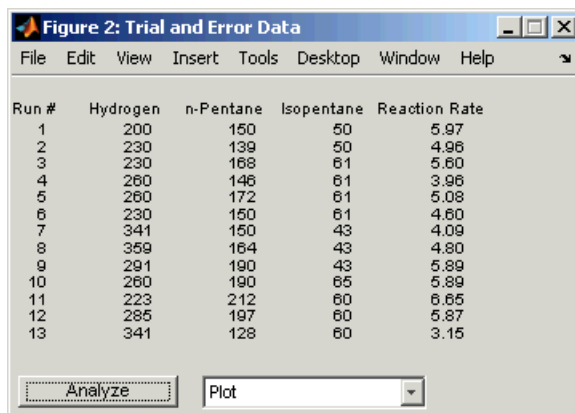
- 1 To begin, run the `rsmdemo` function.

```
rsmdemo
```

- 2 Click **Run** in the Reaction Simulator window to generate a test reaction for the trial and error phase of the demo.



To perform the experiment, you can click **Run** as many as 13 times. For each run, based on the results of previous runs, you can move the sliders in the Reaction Simulator window to change the levels of the reactants to increase or decrease the reaction rate. Each time you click the **Run** button, the levels for the reactants and results of the run are displayed in the Trial and Error Data window, as shown in the following figure after 13 trials.



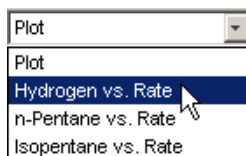
Run #	Hydrogen	n-Pentane	Isopentane	Reaction Rate
1	200	150	50	5.97
2	230	139	50	4.96
3	230	168	61	5.60
4	260	146	61	3.96
5	260	172	61	5.08
6	230	150	61	4.60
7	341	150	43	4.09
8	359	164	43	4.80
9	291	190	43	5.89
10	260	190	65	5.89
11	223	212	60	6.65
12	285	197	60	5.87
13	341	128	60	3.15

---

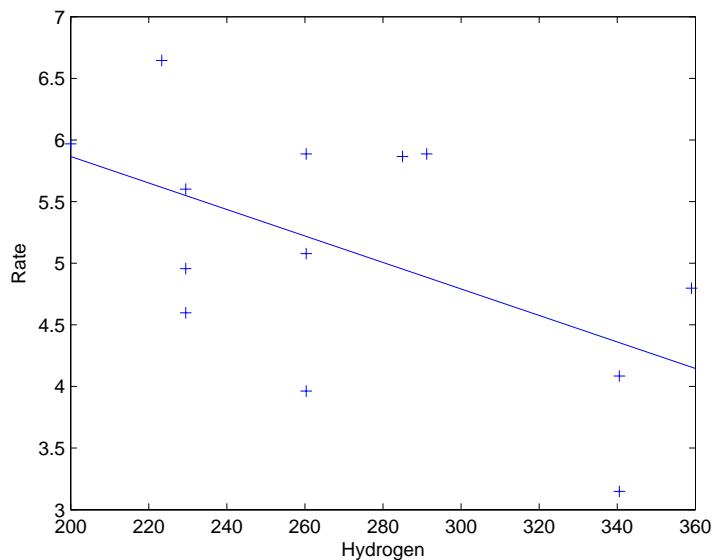
**Note** The results are determined using an underlying model that takes into account the noise in the process, so even if you keep all of the levels the same, the results will vary from run to run. In this case however, the **Analyze** function will not be able to generate a fit for the results.

---

- 3** When you have completed 13 runs, select **Hydrogen vs. Rate**, in the field next to **Analyze**, to plot the relationships between the reactants and the reaction rate.



For this set of 13 runs, rsmdemo produces the following plot.



- 4** Click the **Analyze** button to call the `rstool` function, which you can then use to try to optimize the results. See “Exploring Graphs of Multidimensional Polynomials” on page 4-42 for more information about using the `rstool` demo.
- 5** Next, perform another set of 13 runs, this time from a designed experiment. In the Experimental Data window, click the **Do Experiment** button. `rsmdemo` calls the `cordexch` function to generate a D-optimal design, and then, for each run, computes the reaction rate.

Figure 3: Experimental Data

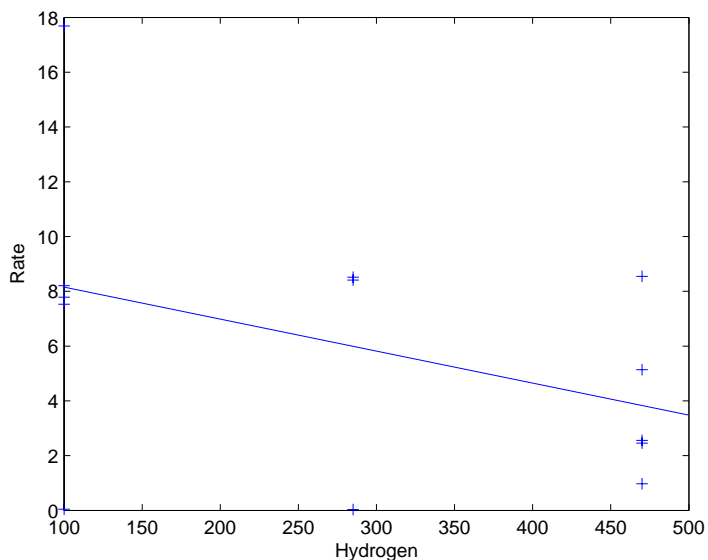
File Edit View Insert Tools Desktop Window Help

Do Experiment

Run #	Hydrogen	n-Pentane	Isopentane	Reaction Rate
1	100	80	10	7.29
2	285	300	120	6.15
3	100	300	10	19.76
4	470	300	65	6.08
5	470	300	10	8.74
6	470	190	120	2.72
7	100	300	120	7.34
8	100	80	120	0.04
9	100	190	65	8.16
10	285	190	10	8.10
11	470	80	120	0.02
12	470	80	10	2.70
13	285	80	65	1.56

Response Surface Nonlinear Model Plot

- 6 Select **Hydrogen vs. Rate** in the field next to Nonlinear Model in the Experimental Data window. This displays the following plot.



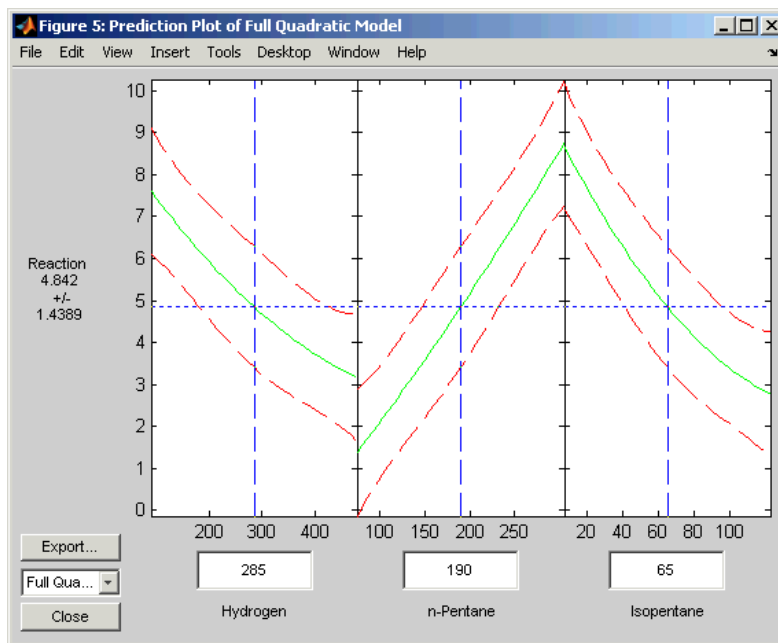
- 7 You can also click the **Response Surface** button to call `rstool` to find the optimal levels of the reactants.

- 8 Compare the analysis results for the two sets of data. It is likely (though not certain) that you'll find some or all of these differences:
  - You can fit a full quadratic model with the data from the designed experiment, but the trial and error data may be insufficient for fitting a quadratic model or interactions model.
  - Using the data from the designed experiment, you are more likely to be able to find levels for the reactants that result in the maximum reaction rate. Even if you find the best settings using the trial and error data, the confidence bounds are likely to be wider than those from the designed experiment.

### Comparing Results Using a Polynomial Model and a Nonlinear Model

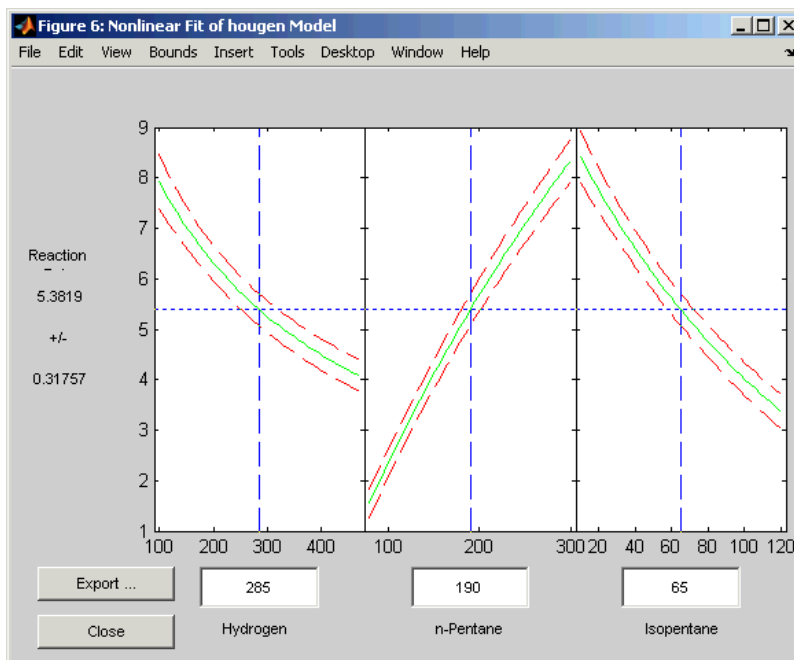
This part of the experiment analyzes the experimental design data with a polynomial (response surface) model and a nonlinear model, and compare the results. The true model for the process, which is used to generate the data, is actually a nonlinear model. However, within the range of the data, a quadratic model approximates the true model quite well:

- 1 Using the results generated in the designed experiment part of “Comparing Results from Trial-and-Error Data and a Designed Experiment” on page 10-11, click the **Response Surface** button on the Experimental Data window. `rsmdemo` calls `rsTool`, which fits a full quadratic model to the data. Drag the reference lines to change the levels of the reactants, and find the optimal reaction rate. Observe the width of the confidence intervals.



- 2 Now click the **Nonlinear Model** button on the Experimental Data window. `rsmdemo` calls `nlintool`, which fits a Hougen-Watson model to the data. As with the quadratic model, you can drag the reference lines to change the reactant levels. Observe the reaction rate and the confidence intervals.





- 3** Compare the analysis results for the two models. Even though the true model is nonlinear, you may find that the polynomial model provides a good fit. Because polynomial models are much easier to fit and work with than nonlinear models, a polynomial model is often preferable even when modeling a nonlinear process. Keep in mind, however, that such models are unlikely to be reliable for extrapolating outside the range of the data.

## D-Optimal Designs

The designs above pre-date the computer age, and some were in use by early in the 20th century. In the 1970s statisticians started to use the computer in experimental design by recasting the design of experiments (DOE) in terms of optimization. A D-optimal design is one that maximizes the determinant of Fisher's information matrix,  $X^T X$ . This matrix is proportional to the inverse of the covariance matrix of the parameters. So maximizing  $\det(X^T X)$  is equivalent to minimizing the determinant of the covariance of the parameters.

A D-optimal design minimizes the volume of the confidence ellipsoid of the regression estimates of the linear model parameters,  $\beta$ .

There are several functions in the Statistics Toolbox that generate D-optimal designs. These are `cordexch`, `daugment`, `dcovary`, and `rowexch`. The following sections explore D-optimal design in greater detail:

- “Generating D-Optimal Designs” on page 10-18
- “Augmenting D-Optimal Designs” on page 10-21
- “Designing Experiments with Uncontrolled Inputs” on page 10-23
- “Controlling Candidate Points” on page 10-24
- “Including Categorical Factors” on page 10-24

### Generating D-Optimal Designs

The `cordexch` and `rowexch` functions provide two competing optimization algorithms for computing a D-optimal design given a model specification.

Both `cordexch` and `rowexch` are iterative algorithms. They operate by improving a starting design by making incremental changes to its elements. In the coordinate exchange algorithm, the increments are the individual elements of the design matrix. In row exchange, the elements are the rows of the design matrix. Atkinson and Donev [1] is a reference.

To generate a D-optimal design you must specify the number of inputs, the number of runs, and the order of the model you want to fit.

Both `cordexch` and `rowexch` take the following strings to specify the model:

- 'linear' or 'l' – the default model with constant and first order terms
- 'interaction' or 'i' – includes constant, linear, and cross product terms

- 'quadratic' or 'q' – interactions plus squared terms
- 'purequadratic' or 'p' – includes constant, linear and squared terms

Alternatively, you can use a matrix of integers to specify the terms. Details are in the help for the utility function `x2fx`.

For a simple example using the coordinate-exchange algorithm, consider the problem of quadratic modeling with two inputs. The model form is

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_{12} x_1 x_2 + \beta_{11} x_1^2 + \beta_{22} x_2^2 + \varepsilon$$

Suppose you want the D-optimal design for fitting this model with nine runs.

```
settings = cordexch(2,9,'q')
settings =
    -1     1
     1     1
     0     1
     1    -1
    -1    -1
     0    -1
     1     0
     0     0
    -1     0
```

You can plot the columns of `settings` against each other to get a better picture of the design.

```
h = plot(settings(:,1),settings(:,2),'.');
set(gca,'Xtick',[-1 0 1])
set(gca,'Ytick',[-1 0 1])
set(h,'Markersize',20)
```



For a simple example using the row-exchange algorithm, consider the interaction model with two inputs. The model form is

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_{12} x_1 x_2 + \varepsilon$$

Suppose you want the D-optimal design for fitting this model with four runs.

```
[settings, X] = rowexch(2,4,'i')
```

```
settings =
```

```

-1    1
-1   -1
 1   -1
 1    1
```

```
X =
```

```

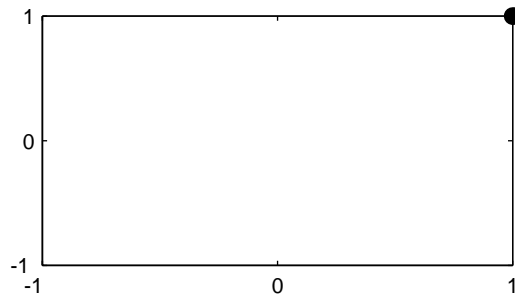
 1   -1    1   -1
 1   -1   -1    1
 1    1   -1   -1
 1    1    1    1
```

The settings matrix shows how to vary the inputs from run to run. The X matrix is the design matrix for fitting the above regression model. The first column of X is for fitting the constant term. The last column is the element-wise product of the second and third columns.

The associated plot is simple but elegant.

```
h = plot(settings(:,1),settings(:,2),'.');
set(gca,'Xtick',[-1 0 1])
```

```
set(gca,'Ytick',[-1 0 1])
set(h,'Markersize',20)
```



## Augmenting D-Optimal Designs

In practice, experimentation is an iterative process. You often want to add runs to a completed experiment to learn more about the system. The function `daugment` allows you choose these extra runs optimally.

Suppose you execute the eight-run design below for fitting a linear model to four input variables.

```
settings = cordexch(4,8)
settings =
    1    -1     1     1
   -1    -1     1    -1
   -1     1     1     1
    1     1     1    -1
   -1     1    -1     1
    1    -1    -1     1
   -1    -1    -1    -1
    1     1    -1    -1
```

This design is adequate to fit the linear model for four inputs, but cannot fit the six cross-product (interaction) terms. Suppose you are willing to do eight more runs to fit these extra terms. The following code show how to do so.

```
[augmented, X] = daugment(settings,8,'i');
```

```
augmented
augmented =
```

```

1    -1    1    1
-1   -1    1   -1
-1    1    1    1
1     1    1   -1
-1    1   -1    1
1    -1   -1    1
-1   -1   -1   -1
1     1   -1   -1
-1   -1   -1    1
1     1    1    1
-1   -1    1    1
-1    1    1   -1
1    -1    1   -1
1    -1   -1   -1
-1    1   -1   -1
1     1   -1    1
```

```
info = X'*X
info =
```

```

16    0    0    0    0    0    0    0    0    0    0
 0   16    0    0    0    0    0    0    0    0    0
 0    0   16    0    0    0    0    0    0    0    0
 0    0    0   16    0    0    0    0    0    0    0
 0    0    0    0   16    0    0    0    0    0    0
 0    0    0    0    0   16    0    0    0    0    0
 0    0    0    0    0    0   16    0    0    0    0
 0    0    0    0    0    0    0   16    0    0    0
 0    0    0    0    0    0    0    0   16    0    0
 0    0    0    0    0    0    0    0    0   16    0
 0    0    0    0    0    0    0    0    0    0   16
```

The augmented design is orthogonal, since  $X' * X$  is a multiple of the identity matrix. In fact, this design is the same as a  $2^4$  factorial design.

## Designing Experiments with Uncontrolled Inputs

Sometimes it is impossible to control every experimental input. But you might know the values of some inputs in advance. An example is the time each run takes place. If a process is experiencing linear drift, you might want to include the time of each test run as a variable in the model.

The function `dcovary` enables you to choose the settings for each run in order to maximize your information despite a linear drift in the process.

Suppose you want to execute an eight-run experiment with three factors that is optimal with respect to a linear drift in the response over time. First you create the drift input variable. Note that drift is normalized to have mean zero. Its minimum is -1 and its maximum is 1.

```
drift = (linspace(-1,1,8))'
drift =

    -1.0000
    -0.7143
    -0.4286
    -0.1429
     0.1429
     0.4286
     0.7143
     1.0000

settings = dcovary(3,drift,'linear')
settings =

     1.0000     1.0000    -1.0000    -1.0000
    -1.0000    -1.0000    -1.0000    -0.7143
    -1.0000     1.0000     1.0000    -0.4286
     1.0000    -1.0000     1.0000    -0.1429
    -1.0000     1.0000    -1.0000     0.1429
     1.0000     1.0000     1.0000     0.4286
    -1.0000    -1.0000     1.0000     0.7143
     1.0000    -1.0000    -1.0000     1.0000
```

## Controlling Candidate Points

The `rowexch` function generates a candidate set of possible design points, and then uses a D-optimal algorithm to select a design from those points. It does this by invoking the `candgen` and `candexch` functions. If you need to supply your own candidate set, or if you need to modify the one that the `candgen` function provides, you might prefer to call these functions separately.

This example creates a design that represents proportions of a mixture, so the sum of the proportions cannot exceed 1.

```
% Generate a matrix of (x,y) values with x+y<=1
[x,y]=meshgrid(0:.1:1);
xy = [x(:) y(:)];
xy = xy(sum(xy,2)<=1,:);

% Compute quadratic model terms for these points.
f = x2fx(xy,'q');

% Generate a 10-point design and display it
r=candexch(f,10);
xy(r,:)
ans =
    0         0
    0    1.0000
    1.0000    0
    0    0.5000
    0.5000    0
    0    1.0000
    1.0000    0
    0.5000    0.5000
    0.5000    0
    0.5000    0.5000
```

## Including Categorical Factors

Another example where it is useful to call `candexch` directly is to generate a design that includes categorical factors. For these designs you create a candidate set containing dummy variables for the categorical factors. The `dummyvar` function is useful to create such a candidate set.



This example contains three categorical factors, each taking three levels. You create a candidate set  $F$  containing all 27 combinations of these factor levels. Then you create a matrix  $C$  containing dummy variables for the factors, and remove enough columns to make the resulting matrix full rank. (You remove one column for each factor except the first factor.) Finally, you use the `candexch` function to generate a nine-run design. The resulting design has the property that for each pair of factors, each of the 9 possible combinations of levels appears exactly once.

```
F = fullfact([3 3 3]);
C = dummyvar(F);
C(:,[4 7]) = [];
rows = candexch(C,9);
D = F(rows,:)
D =
    3  1  3
    1  3  2
    3  3  1
    1  2  3
    2  2  1
    2  1  2
    3  2  2
    2  3  3
    1  1  1
```



# Hidden Markov Models

---

Introduction (p. 11-2)

Introduces the concept of a hidden Markov model.

Example of a Hidden Markov Model  
(p. 11-4)

Gives an example of a hidden Markov model.

Markov Chains (p. 11-6)

Provides a definition of Markov chains.

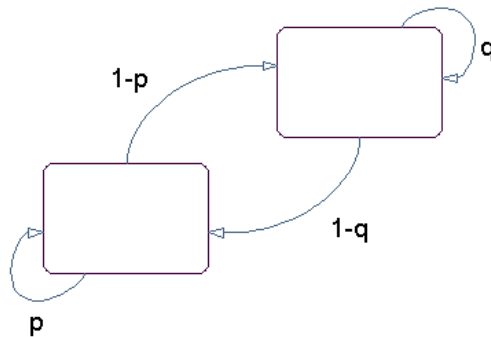
Analyzing a Hidden Markov Model  
(p. 11-8)

Shows how to use Statistics Toolbox functions to analyze a hidden Markov model.

## Introduction

Markov models are mathematical models of stochastic processes — processes that generate random sequences of outcomes according to certain probabilities. A simple example of a stochastic process is a sequence of coin tosses, the outcomes being heads or tails. People use Markov models to analyze a wide variety of stochastic processes, from daily stock prices to the positions of genes in a chromosome.

You can construct Markov models very easily using *state diagrams*, such as the one shown in this figure.



### A State Diagram for a Markov Model

The rectangles in the diagram represent the possible states of the process you are trying to model, and the arrows represent transitions between states. The label on each arrow represents the probability of that transition, which depends on the process you are modeling. At each step of the process, the model generates an output, or *emission*, depending on which state it is in, and then makes a transition to another state.

For example, if you are modeling a sequence of coin tosses, the two states are heads and tails. The most recent coin toss determines the current state of the model and each subsequent toss determines the transition to the next state. If the coin is fair, the transition probabilities are all  $1/2$ . In this simple example, the emission at any moment in time is simply the current state. However, in more complicated models, the states themselves can contain random processes that affect their emissions. For example, after each flip of the coin, you could roll a die to determine the emission at that step.

A *hidden Markov model* is one in which you observe a sequence of emissions, but you do not know the sequence of states the model went through to generate the emissions. In this case, your goal is to recover the state information from the observed data. The next section, “Example of a Hidden Markov Model” on page 11-4, provides an example.

The Statistics Toolbox includes five functions for analyzing hidden Markov models:

- `hmmdecode` — Calculates the posterior state probabilities of a sequence
- `hmmgenerate` — Generates a sequence for a hidden Markov model
- `hmmestimate` — Estimates the parameters for a Markov model
- `hmmtrain` — Calculates the maximum likelihood estimate of hidden Markov model parameters
- `hmmviterbi` — Calculates the most likely state path for a hidden Markov model sequence

“Analyzing a Hidden Markov Model” on page 11-8 explains how to use these functions in detail.

## Example of a Hidden Markov Model

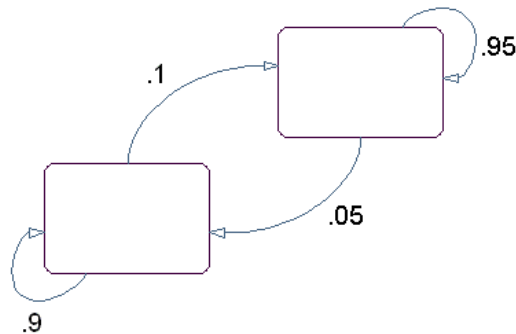
This section describes a simple example of a Markov model in which there are two states and six possible emissions. The example uses the following objects:

- A red die, having six sides, labeled 1 through 6.
- A green die, having twelve sides, five of which are labeled 2 through 6, while the remaining seven sides are labeled 1.
- A weighted red coin, for which the probability of heads is  $.9$  and the probability of tails is  $.1$ .
- A weighted green coin, for which the probability of heads is  $.95$  and the probability of tails is  $.05$ .

You create a sequence of numbers from the set  $\{1, 2, 3, 4, 5, 6\}$  using the following rules:

- Begin by rolling the red die and writing down the number that comes up, which is the emission.
- Toss the red coin and do one of the following:
  - If the result is heads, roll the red die and write down the result.
  - If the result is tails, roll the green die and write down the result.
- At each subsequent step, you flip the coin that has the same color as the die you rolled in the previous step. If the coin comes up heads, roll the same die as in the previous step. If the coin comes up tails, switch to the other die.

You can model this example with a state diagram that has two states, red and green, as shown in the following figure.



You determine the emission from a state by rolling the die with the same color as the state, and the transition to the next state by flipping the coin with the same color as the state.

So far, the model is not hidden, because you know the sequence of states from the colors of the coins and dice. But, suppose that someone else is generating the emissions without showing you the dice or coins. All you can see is the sequence of numbers. If you start seeing more 1s than other numbers, you might suspect that the model is in the green state, but you cannot be sure because you cannot see the color of the die being rolled. This is an example of a hidden Markov model: you can observe the sequence of emissions, but you do not know what state the model is in — that is, what color die is being rolled — when the emission occurs.

Not knowing the state the model is in raises the following problems:

- Given a sequence, what is the most likely state path?
- How can you estimate the parameters of the model given the state path?
- How can you estimate the parameters of the model without knowing the state path?
- What is the probability that the model generates a given sequence? This is known as the *forward probability*.
- What is the probability that the model is in a particular state at any point in the sequence? This is the *posterior probability*.

## Markov Chains

This section defines *Markov chains*, which are the mathematical descriptions of Markov models. A Markov chain contains the following elements:

- A set of states  $\{1, 2, \dots, M\}$
- An  $M$ -by- $M$  *transition matrix*  $T$  whose  $i, j$  entry is the probability of a transition from state  $i$  to state  $j$ . The matrix corresponds to a state diagram like the one shown in the Figure , A State Diagram for a Markov Model, on page 11-2. 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*,  $\{s_1, s_2, \dots, s_N\}$ . By default, the set of emissions is  $\{1, 2, \dots, N\}$ , where  $N$  is the number of possible emissions, but you can choose a different set of numbers or symbols.
- An  $M$ -by- $N$  *emission matrix*  $E$  whose  $i, k$  entry gives the probability of emitting symbol  $s_k$  given that the model is in state  $i$ .

When the model is in state  $i_1$ , it emits an output  $s_{k_1}$  with probability  $E_{i_1 k_1}$ . The model then makes a transition to state  $i_2$  with probability  $T_{i_1 i_2}$ , and emits another symbol.

You can represent the example in “Example of a Hidden Markov Model” on page 11-4 by a Markov chain with two states, red and green. You determine transitions between states by flipping the coins. The transition matrix is

$$T = \begin{bmatrix} 0.9 & 0.1 \\ 0.05 & 0.95 \end{bmatrix}$$

You determine emissions by rolling the dice. The emissions matrix is

$$E = \begin{bmatrix} \frac{1}{6} & \frac{1}{6} & \frac{1}{6} & \frac{1}{6} & \frac{1}{6} & \frac{1}{6} \\ \frac{7}{12} & \frac{1}{12} & \frac{1}{12} & \frac{1}{12} & \frac{1}{12} & \frac{1}{12} \end{bmatrix}$$

“Analyzing a Hidden Markov Model” on page 11-8 shows how to analyze this model using functions in the Statistics Toolbox.



## How the Toolbox Generates Random Sequences

The hidden Markov model functions in the Statistics Toolbox generate random sequences using the transition matrix,  $T$ , and the emission matrix,  $E$ , as described in the preceding section. The functions always begin with the model in the *initial state*,  $i_0 = 1$ , at step 0. The model then makes a transition to state  $i_1$  with probability  $T_{1i_1}$ , and emits an output  $s_{k_1}$  with probability  $E_{i_1k_1}$ . Consequently, the probability of observing the sequence of states  $i_1i_2\dots i_r$  and the sequence of emissions  $s_{k_1}s_{k_2}\dots s_{k_r}$  in the first  $r$  steps, is

$$T_{1i_1}E_{i_1k_1}T_{i_1i_2}E_{i_2k_2}\dots T_{i_{r-1}i_r}E_{i_rk_r}$$

Note that if the function returns a generated sequence of states, the first state in the sequence is  $i_1$ : the initial state,  $i_0$ , is not included.

In this implementation, the initial state is 1 with probability 1, and all other states have probability 0 of being the initial state. At times, you might want to change the probabilities of the initial states. You can do so by adding a new artificial state 1 that has transitions to the other states with any probabilities you want, but that never occurs after step 0. See “Changing the Probabilities of the Initial States” on page 11-13 to learn how to do this.

## Analyzing a Hidden Markov Model

This section explains how to use functions in the Statistics Toolbox to analyze hidden Markov models. For illustration, the section uses the example described in “Example of a Hidden Markov Model” on page 11-4. The section shows how to recover information about the model, assuming that you do not know some of the model’s parameters. The section covers the following topics:

- “Setting Up the Model and Generating Data” on page 11-8
- “Computing the Most Likely Sequence of States” on page 11-9
- “Estimating the Transition and Emission Matrices” on page 11-9
- “Changing the Probabilities of the Initial States” on page 11-13
- “Example: Changing the Initial Probabilities” on page 11-14

### Setting Up the Model and Generating Data

This section shows how to set up a hidden Markov model and use it to generate data. First, create the transition and emission matrices by entering the following commands.

```
TRANS = [.9 .1; .05 .95];  
  
EMIS = [1/6, 1/6, 1/6, 1/6, 1/6, 1/6; ...  
7/12, 1/12, 1/12, 1/12, 1/12, 1/12];
```

Next, generate a random sequence of emissions from the model, `seq`, of length 1000, using the function `hmmgenerate`. You can also return the corresponding random sequence of states in the model as the second output, `states`.

```
[seq, states] = hmmgenerate(1000, TRANS, EMIS);
```

---

**Note** In generating the sequences `seq` and `states`, `hmmgenerate` begins with the model in state  $i_0 = 1$  at step 0. The model then makes a transition to state  $i_1$  at step 1, and returns  $i_1$  as the first entry in `states`.

---

## Computing the Most Likely Sequence of States

Suppose you know the transition and emission matrices, TRANS and EMIS. If you observe a sequence, seq, of emissions, how can you compute the most likely sequence of states that generated the sequence? The function `hmmviterbi` uses the Viterbi algorithm to compute the most likely sequence of states that the model would go through to generate the given sequence of emissions.

```
likelystates = hmmviterbi(seq, TRANS, EMIS);
```

`likelystates` is a sequence of the same length as `seq`.

To test the accuracy of `hmmviterbi`, you can compute the percentage of the time that the actual sequence states agrees with the sequence `likelystates`.

```
sum(states==likelystates)/1000  
  
ans =  
  
0.8200
```

This shows that the most likely sequence of states agrees with the actual sequence 82% of the time. Note that your results might differ if you run the same commands, because the sequence `seq` is random.

---

**Note** The states at the beginning of the sequence returned by `hmmviterbi` are less reliable because of the computational delay in the Viterbi algorithm.

---

## Estimating the Transition and Emission Matrices

Suppose you do not know the transition and emission matrices in the model, and you observe a sequence of emissions, `seq`. There are two functions that you can use to estimate the matrices:

- `hmmestimate`
- `hmmtrain`

### Using `hmmestimate`

To use `hmmestimate`, you also need to know the corresponding sequence of states that the model went through to generate `seq`. The following command

takes the emission and state sequences, `seq` and `states`, and returns estimates of the transition and emission matrices, `TRANS_EST` and `EMIS_EST`.

```
[TRANS_EST, EMIS_EST] = hmmestimate(seq, states)

TRANS_EST =

0.8989    0.1011
0.0585    0.9415

EMIS_EST =

0.1721    0.1721    0.1749    0.1612    0.1803    0.1393
0.5836    0.0741    0.0804    0.0789    0.0726    0.1104
```

You can compare these outputs with the original transition and emission matrices, `TRANS` and `EMIS`, to see how well `hmmestimate` estimates them.

```
TRANS

TRANS =

0.9000    0.1000
0.0500    0.9500

EMIS

EMIS =

0.1667    0.1667    0.1667    0.1667    0.1667    0.1667
0.5833    0.0833    0.0833    0.0833    0.0833    0.0833
```

### Using `hmmtrain`

If you do not know the sequence of states, but you have an initial guess as to the values of `TRANS` and `EMIS`, you can estimate the transition and emission matrices using the function `hmmtrain`. For example, suppose you have the following initial guesses for `TRANS` and `EMIS`.

```
TRANS_GUESS = [.85 .15; .1 .9];
EMIS_GUESS = [.17 .16 .17 .16 .17 .17;.6 .08 .08 .08 .08 .08];
```

You can estimate TRANS and EMIS with the following command.

```
[TRANS_EST2, EMIS_EST2] = hmmtrain(seq, TRANS_GUESS, EMIS_GUESS)
```

```
TRANS_EST2 =
```

```
0.2286    0.7714
```

```
0.0032    0.9968
```

```
EMIS_EST2 =
```

```
0.1436    0.2348    0.1837    0.1963    0.2350    0.0066
```

```
0.4355    0.1089    0.1144    0.1082    0.1109    0.1220
```

`hmmtrain` uses an iterative algorithm that alters the matrices `TRANS_GUESS` and `EMIS_GUESS` so that at each step the adjusted matrices are more likely to generate the observed sequence, `seq`. The algorithm halts when the matrices in two successive iterations are within a small tolerance of each other. See the reference page for `hmmtrain` for more information about the tolerance.

If the algorithm fails to reach this tolerance within a maximum number of iterations, whose default value is 100, the algorithm halts. In this case, `hmmtrain` returns the last values of `TRANS_EST` and `EMIS_EST` and issues a warning that the tolerance was not reached.

If the algorithm fails to reach the desired tolerance, you can increase the default value of the maximum number of iterations with the command

```
hmmtrain(seq, TRANS_GUESS, EMIS_GUESS, 'maxiterations', maxiter)
```

where `maxiter` is the maximum number of steps the algorithm executes.

You can also change default value of the tolerance with the command

```
hmmtrain(seq, TRANS_GUESS, EMIS_GUESS, 'tolerance', tol)
```

where `tol` is the desired value of the tolerance. Increasing the value of `tol` makes the algorithm halt sooner, but the results are less accurate.

---

**Note** If the sequence `seq` is long, the `hmmtrain` algorithm might take a long time to run. If so, you might want to lower the maximum number of iterations temporarily at first to find out how much time the algorithm requires.

---

There are two factors that can make the output matrices of `hmmtrain` less reliable:

- The algorithm might converge to a local maximum that does not represent the true transition and emission matrices. If you suspect that this is the case, try different initial guesses for the matrices `TRANS_EST` and `EMIS_EST`.
- The sequence `seq` might be too short to properly train the matrices. If you suspect this is the case, try using a longer sequence for `seq`.

## Calculating Posterior State Probabilities

The posterior state probabilities of an emission sequence `seq` are the conditional probabilities that the model is in a particular state when it generates a symbol in `seq`, given that `seq` is emitted. You can compute the posterior state probabilities with the following command:

```
PSTATES = hmmdecode(seq, TRANS, EMIS)
```

The output `PSTATES` is an  $M$ -by- $L$  matrix, where  $M$  is the number of states and  $L$  is the length of `seq`. `PSTATES(i, j)` is the conditional probability that the model is in state  $i$  when it generates the  $j$ th symbol of `seq`, given that `seq` is emitted.

---

**Note** The function `hmmdecode` begins with the model in state 1 at step 0, prior to the first emission. `PSTATES(i, 1)` is the probability that the model is in state  $i$  at the following step 1.

---

You can also return the logarithm of the probability of the sequence `seq` as the second output argument.

```
[PSTATES, logpseq] = hmmdecode(seq, TRANS, EMIS)
```

The actual probability of a sequence tends to 0 rapidly as the length of the sequence increases, so the probability of a sufficiently long sequence is less than the smallest positive number your computer can represent. Consequently, `hmmdecode` returns the logarithm of the probability instead.

For example, the following code returns the logarithm probability of the one-element sequence [3].

```
[PSTATES, logpseq] = hmmdecode([3], TRANS, EMIS);
exp(logpseq)

ans =

0.1583
```

Note that you can compute this answer directly as

$$\sum_{j=1}^6 T_{1j} E_{j3}$$

by the commands

```
TRANS(1, :)*EMIS(:, 3)

ans =

0.1583
```

## Changing the Probabilities of the Initial States

By default, the hidden Markov model functions begin with the model in state 1 at step 0. In other words, with probability 1, the initial state is 1, and all other states have probability 0 of being the initial state. See “How the Toolbox Generates Random Sequences” on page 11-7.

For some models, you might want to assign different probabilities to the initial states. For example, you might want to choose initial state probabilities from a probability vector  $p$  satisfying  $pT = p$ . This assignment makes the Markov chain time independent: the probability of observing a given output at a specified step of the sequence is independent of the step number. This section explains how to assign any vector of probabilities for the initial states in your model.

To assign a vector of probabilities,  $p = [p_1, p_2, \dots, p_M]$ , to the initial states, do the following:

- 1 Create an  $M+1$ -by- $M+1$  augmented transition matrix,  $T$ , that has the following form:

$$\hat{T} = \begin{bmatrix} 0 & p \\ 0 & T \end{bmatrix}$$

where  $T$  is the true transition matrix. The first column of  $T$  contains  $M+1$  zeros.

- 2 Create an  $M+1$ -by- $N$  augmented emission matrix,  $E$ , that has the following form:

$$\hat{T} = \begin{bmatrix} 0 \\ E \end{bmatrix}$$

If the transition and emission matrices are `TRANS` and `EMIS`, respectively, you can create the augmented matrices with the following commands:

```
TRANS_HAT = [0 p; zeros(size(TRANS,1),1) TRANS];
EMIS_HAT = [zeros(1,size(EMIS,2)); EMIS];
```

### Example: Changing the Initial Probabilities

For example, suppose that you have the following transition and emission matrices.

```
TRANS = [.9 .1; .05 .95];
EMIS = [1/6, 1/6, 1/6, 1/6, 1/6, 1/6; ...
        7/12, 1/12, 1/12, 1/12, 1/12, 1/12];
```

You want to assign the states initial probabilities that are given by a left eigenvector,  $p$ , for `TRANS`, corresponding to the maximum eigenvalue 1. These initial probabilities make the Markov model time independent: the probability of observing a given emission is the same at each step of the output sequence.



To find the vector  $p$ , enter the following commands.

```
[V,D] = eigs(TRANS')
```

```
V =
```

```
-0.4472   -0.7071
-0.8944    0.7071
```

```
D =
```

```
1.0000         0
         0    0.8500
```

The first column of  $V$  is the right eigenvector for  $TRANS'$  corresponding to eigenvalue 1. So the transpose of this vector is a left eigenvector for  $TRANS$ . You can create this vector as follows.

```
p = V(:, 1)'
```

```
p =
```

```
-0.4472   -0.8944
```

```
p*TRANS
```

```
ans =
```

```
-0.4472   -0.8944
```

This is not yet a probability vector, so divide  $p$  by its sum.

```
p = p/sum(p)
```

```
p =
```

```
0.3333    0.6667
```

Next, create the augmented matrices  $TRANS\_HAT$  and  $EMIS\_HAT$ .

```
TRANS_HAT = [0 p; zeros(size(TRANS,1),1) TRANS]
```

```
TRANS_HAT =
```

```
0    0.3333    0.6667
0    0.9000    0.1000
0    0.0500    0.9500
```

```
EMIS_HAT = [zeros(1,size(EMIS,2)); EMIS]

EMIS_HAT =

    0         0         0         0         0         0
0.1667    0.1667    0.1667    0.1667    0.1667    0.1667
0.5833    0.0833    0.0833    0.0833    0.0833    0.0833
```

This assignment of probabilities makes the Markov model time independent. For example, you can calculate the probability of seeing symbol 3 at step 1 of an emission sequence using the function `hmmdecode` as follows.

```
[pStates, logp]=hmmdecode([3],TRANS_HAT,EMIS_HAT);

exp(logp)

ans =
0.1111
```

Note that the second output argument, `logp`, is the logarithm of the probability of the sequence [3].

On the other hand, the probability of seeing symbol 3 at step 2 is the sum of the probabilities of the sequences [1 3], [2 3], [3 3], [4 3], [5 3], and [6 3].

```
sum = 0;
for n = 1:6
    [pStates, logp] = hmmdecode([n 3],TRANS_HAT,EMIS_HAT);
    sum = sum + exp(logp);
end;
sum

sum =
0.1111
```

## References

To learn more about hidden Markov models and their applications, see the following reference.

Durbin, R., S. Eddy, A. Krogh, and G. Mitchison, *Biological Sequence Analysis*, Cambridge Univ. Press, 1998.

# Function Reference

---

Functions — Categorical List  
(p. 12-2)

Brief descriptions of the Statistics Toolbox functions, arranged by category

Functions — Alphabetical List  
(p. 12-1)

Reference pages for the Statistics Toolbox functions in alphabetical order

## Functions – Categorical List

The Statistics Toolbox provides several categories of functions.

Probability Distributions	Specific functions for each supported distribution
Descriptive Statistics	Descriptive statistics for data samples
Statistical Plotting	Statistical plots
Statistical Process Control	Statistical Process Control
Linear Models	Fitting linear models to data
Nonlinear Regression	Fitting nonlinear regression models
Design of Experiments	Design of Experiments
Multivariate Statistics	Multivariate statistics
Decision Tree Techniques	Decision trees
Hypothesis Tests	Statistical tests of hypotheses
Distribution Testing	Tests for fitting distributions to data
Nonparametric Testing	Nonparametric testing
Hidden Markov Models	Finding hidden Markov models
File I/O	Reading data from and writing data to operating-system files
Demonstrations	Demonstrations
Data	Data for examples
Utility	Utility functions

### Probability Distributions

For each supported distribution, the Statistics Toolbox provides associated functions in each of these categories

Parameter Estimation

Cumulative Distribution Functions (cdf)

Probability Density Functions (pdf)

Inverse Cumulative Distribution Functions

Random Number Generators

Moments of Distribution Functions

### Parameter Estimation

betafit	Parameter estimation for the beta distribution
betalike	Negative beta log-likelihood function
binofit	Parameter estimation for the binomial distribution
evfit	Parameter estimation for the extreme value distribution
evlike	Negative log-likelihood for the extreme value distribution
expfit	Parameter estimation for the exponential distribution
explike	Negative log-likelihood for the exponential distribution
gamfit	Parameter estimation for the gamma distribution
gamlike	Negative gamma log-likelihood function
lognfit	Parameter estimation for the lognormal distribution
lognlike	Negative log-likelihood for the lognormal distribution
mle	Maximum likelihood estimation
nbinfit	Parameter estimates and confidence intervals for negative binomial data
normfit	Parameter estimation for the normal distribution
normlike	Negative normal log-likelihood function
poissfit	Parameter estimation for the Poisson distribution
raylfit	Rayleigh parameter estimation
unifit	Parameter estimation for the uniform distribution
wblfit	Weibull parameter estimation
wbllike	Weibull negative log-likelihood function

**Cumulative Distribution Functions (cdf)**

betacdf	Beta cdf
binocdf	Binomial cdf
cdf	Parameterized cdf routine
chi2cdf	Chi-square cdf
ecdf	Empirical (Kaplan-Meier) cdf
evcdf	Extreme value cdf
expcdf	Exponential cdf
fcdf	F cdf
gamcdf	Gamma cdf
geocdf	Geometric cdf
hygecdf	Hypergeometric cdf
logncdf	Lognormal cdf
nbincdf	Negative binomial cdf
ncfcdf	Noncentral F cdf
nctcdf	Noncentral t cdf
ncx2cdf	Noncentral Chi-square cdf
normcdf	Normal (Gaussian) cdf
poisscdf	Poisson cdf
raylcdf	Rayleigh cdf
tcdf	Student's t cdf
unidcdf	Discrete uniform cdf
unifcdf	Continuous uniform cdf
wblcdf	Weibull cdf

**Probability Density Functions (pdf)**

betapdf	Beta pdf
binopdf	Binomial pdf
chi2pdf	Chi-square pdf
evpdf	Extreme value pdf
exppdf	Exponential pdf
fpdf	F pdf
gampdf	Gamma pdf
geopdf	Geometric pdf
hygepdf	Hypergeometric pdf
lognpdf	Lognormal pdf
mvnpdf	Multivariate normal pdf
nbinpdf	Negative binomial pdf
ncfpdf	Noncentral F pdf
nctpdf	Noncentral t pdf
ncx2pdf	Noncentral Chi-square pdf
normpdf	Normal (Gaussian) pdf
pdf	Parameterized pdf routine
poisspdf	Poisson pdf
raylpdf	Rayleigh pdf
tpdf	Student's t pdf
unidpdf	Discrete uniform pdf
unifpdf	Continuous uniform pdf
wblpdf	Weibull pdf

**Inverse Cumulative Distribution Functions**

betainv	Beta critical values
binoinv	Binomial critical values
chi2inv	Chi-square critical values
evinv	Extreme value critical values
expinv	Exponential critical values
finv	F critical values
gaminv	Gamma critical values
geoinv	Geometric critical values
hygeinv	Hypergeometric critical values
icdf	Parameterized inverse distribution routine
logninv	Lognormal critical values
nbinv	Negative binomial critical values
ncfinv	Noncentral F critical values
nctinv	Noncentral t critical values
ncx2inv	Noncentral Chi-square critical values
norminv	Normal (Gaussian) critical values
poissinv	Poisson critical values
raylinv	Rayleigh critical values
tinvs	Student's t critical values
unidinv	Discrete uniform critical values
unifinv	Continuous uniform critical values
wblinv	Weibull critical values

**Random Number Generators**

betarnd	Beta random numbers
binornd	Binomial random numbers



---

chi2rnd	Chi-square random numbers
evrnd	Extreme value random numbers
exprnd	Exponential random numbers
frnd	F random numbers
gamrnd	Gamma random numbers
geornd	Geometric random numbers
hygernd	Hypergeometric random numbers
iwishrnd	Inverse Wishart random matrix
lhsdesign	Latin hypercube sample
lhsnorm	Latin hypercube sample with normal distribution
lognrnd	Lognormal 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 (Gaussian) random numbers
poissrnd	Poisson random numbers
random	Parameterized random number routine
randsample	Random sample, with or without replacement
raylrnd	Rayleigh random numbers
trnd	Student's t random numbers
unidrnd	Discrete uniform random numbers
unifrnd	Continuous uniform random numbers

wblrnd            Weibull random numbers  
wishrnd           Wishart random matrix

### **Moments of Distribution Functions**

betastat           Beta mean and variance  
binostat           Binomial mean and variance  
chi2stat           Chi-square mean and variance  
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  
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 (Gaussian) mean and variance  
poisstat           Poisson mean and variance  
raylstat           Rayleigh mean and variance  
tstat              Student's t mean and variance  
unidstat           Discrete uniform mean and variance  
unifstat           Continuous uniform mean and variance  
wblstat            Weibull mean and variance

## Descriptive Statistics

<code>bootstrp</code>	Bootstrap statistics for any function
<code>corr</code>	Linear or rank correlation, and p-values
<code>corrcoef</code>	Correlation coefficients (in MATLAB)
<code>cov</code>	Covariance matrix (in MATLAB)
<code>crosstab</code>	Cross tabulation
<code>geomean</code>	Geometric mean
<code>grpstats</code>	Summary statistics by group
<code>harmmean</code>	Harmonic mean
<code>iqr</code>	Interquartile range
<code>kurtosis</code>	Sample kurtosis
<code>mad</code>	Mean absolute deviation
<code>mean</code>	Arithmetic average (in MATLAB)
<code>median</code>	50th percentile (in MATLAB)
<code>mlecov</code>	Asymptotic covariance matrix of maximum likelihood estimators
<code>moment</code>	Central moments of all orders
<code>nanmax</code>	Maximum ignoring missing data
<code>nanmean</code>	Average ignoring missing data
<code>nanmedian</code>	Median ignoring missing data
<code>nanmin</code>	Minimum ignoring missing data
<code>nanstd</code>	Standard deviation ignoring missing data
<code>nansum</code>	Sum ignoring missing data
<code>nanvar</code>	Variance ignoring NaNs
<code>prctile</code>	Empirical percentiles of a sample
<code>quantile</code>	Quantiles of a sample
<code>range</code>	Sample range

skewness	Sample skewness
std	Standard deviation (in MATLAB)
tabulate	Frequency table
trimmean	Trimmed mean
var	Variance

## **Statistical Plotting**

addedvarplot	Added-variable plot for stepwise regression
andrewsplot	Andrews plot for multivariate data
biplot	Biplot of variable-factor coefficients from a factor analysis or a principal components analysis
boxplot	Box plots
cdfplot	Plot of empirical cumulative distribution function
ecdfhist	Histogram from ecdf output
errorbar	Error bar plot
fsurfht	Interactive contour plot of a function
gline	Interactive line drawing
glyphplot	Plot stars or Chernoff faces for multivariate data
gname	Interactive point labeling
gplotmatrix	Matrix of scatter plots grouped by a common variable
gscatter	Scatter plot of two variables grouped by a third
hist	Histogram
hist3	Three-dimensional histogram of bivariate data
lsline	Add least-squares fit line to plotted data
normplot	Normal probability plots
parallelcoords	Parallel coordinates plot for multivariate data
pareto	Pareto charts

probplot	Probability plot
qqplot	Quantile-Quantile plots
rcoplot	Regression case order plot
refcurve	Reference polynomial
refline	Reference line
surfht	Interactive interpolating contour plot
wblplot	Weibull plotting

### **Statistical Process Control**

capable	Quality capability indices
capaplot	Plot of process capability
ewmaplot	Exponentially weighted moving average plot
histfit	Histogram and normal density curve
normspec	Plot normal density between limits
schart	Time plot of standard deviation
xbarplot	Time plot of means

## Linear Models

<code>anova1</code>	One-way Analysis of Variance (ANOVA)
<code>anova2</code>	Two-way Analysis of Variance
<code>anovan</code>	N-way analysis of variance
<code>aocool</code>	Interactive tool for analysis of covariance
<code>dummyvar</code>	Dummy-variable coding
<code>friedman</code>	Friedman's test (nonparametric two-way ANOVA)
<code>glmfit</code>	Generalized linear model fitting
<code>glmval</code>	Compute predictions for generalized linear model
<code>kruskalwallis</code>	Kruskal-Wallis test (nonparametric one-way ANOVA)
<code>leverage</code>	Regression diagnostic
<code>lscov</code>	Regression given a covariance matrix (in MATLAB)
<code>manova1</code>	One-way multivariate analysis of variance
<code>manovacluster</code>	Draw clusters of group means for <code>manova1</code>
<code>multcompare</code>	Multiple comparisons of means and other estimates
<code>polyconf</code>	Polynomial prediction with confidence intervals
<code>polyfit</code>	Polynomial fitting (in MATLAB)
<code>polyval</code>	Polynomial prediction (in MATLAB)
<code>rcoplot</code>	Residuals case order plot
<code>regress</code>	Multiple linear regression
<code>regstats</code>	Regression diagnostics for linear models
<code>ridge</code>	Ridge regression
<code>rstool</code>	Response surface tool
<code>robustfit</code>	Robust regression model fitting
<code>rstool</code>	Multidimensional response surface visualization (RSM)
<code>stepwise</code>	Stepwise regression GUI

---

<code>stepwisefit</code>	Fit regression model using stepwise regression
<code>x2fx</code>	Factor settings matrix ( $X$ ) to design matrix ( $D$ )

## Nonlinear Regression

<code>nlinfit</code>	Nonlinear least-squares fitting
<code>nlintool</code>	Prediction graph for nonlinear fits
<code>nlparci</code>	Confidence intervals on parameters
<code>nlpredci</code>	Confidence intervals for prediction
<code>lsqnonneg</code>	Nonnegative least squares (in MATLAB)
<code>treefit</code>	Fit a tree-based model for classification or regression.
<code>treeprune</code>	Produce a sequence of subtrees by pruning.
<code>treedisp</code>	Show classification or regression tree graphically.
<code>treetest</code>	Compute error rate for tree.
<code>treeval</code>	Compute fitted value for decision tree applied to data.

## Design of Experiments

<code>bbdesign</code>	Box-Behnken design
<code>candgen</code>	Candidate set for D-optimal design
<code>candexch</code>	D-optimal design from candidate set using row exchanges
<code>ccdesign</code>	Central composite design
<code>cordexch</code>	D-optimal design using coordinate exchange
<code>daugment</code>	D-optimal augmentation of designs
<code>dcovary</code>	D-optimal design with fixed covariates
<code>ff2n</code>	Two-level full factorial designs
<code>fracfact</code>	Two-level fractional factorial design
<code>fullfact</code>	Mixed level full factorial designs

hadamard	Hadamard designs (in MATLAB)
rowexch	D-optimal design using row exchange

## **Multivariate Statistics**

### **Cluster Analysis**

cluster	Create clusters from linkage output
clusterdata	Create clusters from a data set
cophenet	Calculate the cophenetic correlation coefficient
dendrogram	Plot a hierarchical tree in a dendrogram graph
inconsistent	Calculate the inconsistency values of objects in a cluster hierarchy tree
kmeans	K-means clustering
linkage	Link objects in a dataset into a hierarchical tree of binary clusters
pdist	Calculate the pairwise distance between objects in a dataset
silhouette	Silhouette plot for clustered data
squareform	Reformat output of pdist function from vector to square matrix



### Dimension Reduction Techniques

factoran	Maximum Likelihood Common Factor Analysis
pcacov	PCA from covariance matrix
pcares	Residuals from PCA
princomp	PCA from raw data matrix
rotatefactors	Rotation of factor analysis or principal components analysis loadings

### Other Multivariate Methods

barttest	Bartlett's test
canoncorr	Canonical correlation analysis
classify	Discriminant Analysis
cmdscale	Classical multidimensional scaling
mahal	Mahalanobis distance
manova1	One-way multivariate analysis of variance
manovacluster	Draw clusters of group means for manova1
mdscale	Nonmetric and metric multidimensional scaling
procrustes	Procrustes Analysis
zscore	Normalize a dataset before calculating the distance

### Decision Tree Techniques

treefit	Fit a tree-based model for classification or regression.
treeprune	Produce a sequence of subtrees by pruning.
treedisp	Show classification or regression tree graphically.
treetest	Compute error rate for tree.
treeval	Compute fitted value for decision tree applied to data.

## Hypothesis Tests

<code>ranksum</code>	Wilcoxon rank sum test
<code>signrank</code>	Wilcoxon signed rank test
<code>signtest</code>	Sign test for paired samples
<code>ttest</code>	One sample t-test
<code>ttest2</code>	Two sample t-test
<code>ztest</code>	Z-test

## Distribution Testing

<code>jbtest</code>	Jarque-Bera test of normality
<code>kstest</code>	Kolmogorov-Smirnov test for one sample
<code>kstest2</code>	Kolmogorov-Smirnov test for two samples
<code>lillietest</code>	Lilliefors test of normality

## Nonparametric Testing

<code>friedman</code>	Friedman's test (nonparametric two-way anova)
<code>kruskalwallis</code>	Kruskal-Wallis test (nonparametric one-way anova)
<code>ksdensity</code>	Probability density estimate using a kernel smoothing method
<code>ranksum</code>	Wilcoxon rank sum test (independent samples)
<code>signrank</code>	Wilcoxon sign rank test (paired samples)
<code>signtest</code>	Sign test (paired samples)

## Hidden Markov Models

<code>hmmdecode</code>	Calculate the posterior state probabilities of a sequence
<code>hmmestimate</code>	Estimate the parameters for a hidden Markov model
<code>hmmgenerate</code>	Generate a sequence for a hidden Markov model

<code>hmmtrain</code>	Calculate the maximum likelihood estimate of hidden Markov model parameters
<code>hmmviterbi</code>	Calculate the most probable state path for a hidden Markov model sequence

## File I/O

<code>caseread</code>	Read case names from a file
<code>casewrite</code>	Write casenames from a string matrix to a file
<code>tblread</code>	Retrieve tabular data from the file system
<code>tblwrite</code>	Write data in tabular form to the file system
<code>tdfread</code>	Read in text and numeric data from tab-delimited file

## Demonstrations

To run `glimdemo` from the command line, type `playshow glimdemo`. You can run all other demos by typing the demo name as a command.

<code>aoctool</code>	Interactive tool for analysis of covariance
<code>disttool</code>	Interactive exploration of distribution functions
<code>glimdemo</code>	Generalized linear model slide show.
<code>randtool</code>	Interactive random number generation
<code>polytool</code>	Interactive fitting of polynomial models
<code>rsmdemo</code>	Interactive process experimentation and analysis
<code>robustdemo</code>	Interactive tool to compare robust and least squares fits

**Data**

<code>carbig.mat</code>	Measurements on 406 car models
<code>carsmall.mat</code>	Measurements on 100 car models from 1970, 1976, and 1982
<code>census.mat</code>	U. S. Population 1790 to 1980
<code>cities.mat</code>	Names of U.S. metropolitan areas
<code>discrim.mat</code>	Classification data
<code>gas.mat</code>	Gasoline prices
<code>hald.mat</code>	Hald data
<code>hogg.mat</code>	Bacteria counts from milk shipments
<code>lawdata.mat</code>	GPA versus LSAT for 15 law schools
<code>mileage.mat</code>	Mileage data for three car models from two factories
<code>moore.mat</code>	Five factor – one response regression data
<code>parts.mat</code>	Dimensional run out on 36 circular parts
<code>popcorn.mat</code>	Data for popcorn example ( <code>anova2</code> , <code>friedman</code> )
<code>polydata.mat</code>	Data for <code>polytool</code> demo
<code>reaction.mat</code>	Reaction kinetics data
<code>sat.dat</code>	ASCII data for <code>tblread</code> example

**Utility**

<code>statget</code>	Get parameter values from a statistics options structure
<code>statset</code>	Create or edit a statistics options structure

## **Functions — Alphabetical List**

This section contains function reference pages listed alphabetically.

# addedvarplot

---

**Purpose** Create added-variable plot for stepwise regression

**Syntax** `addedvarplot(X, y, vnum, inmodel)`

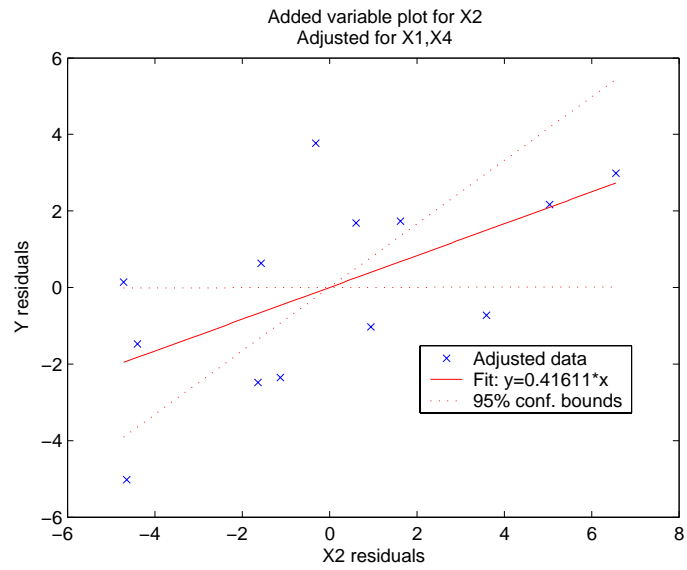
**Description** `addedvarplot(X, y, vnum, inmodel)` produces an added variable plot for the response `y` and the predictor in column `vnum` of `X`. This plot illustrates the incremental affect of this predictor in a regression model in which the columns listed in the vector `inmodel` are used as predictors. `X` is an `n`-by-`p` matrix of predictor values. `y` is vector of `n` response values. `vnum` is a scalar index specifying the column of `X` to use in the plot. `inmodel` is a logical vector of `p` elements specifying the columns of `X` to use in the base model. By default, all elements of `inmodel` are false, which means that the model has no predictors. You can call the function `stepwisefit`, which produces a regression model using stepwise regression, to create the vector `inmodel`.

`addedvarplot(X, y, vnum, inmodel, stats)` uses the structure `stats`, which contains fitted model results created by the `stepwisefit` function. If you create the structure `stats` by calling `stepwisefit`, prior to calling `addedvarplot`, you can save computational time by including the argument `stats` in `addedvarplot`.

An added variable plot contains data and fitted lines. If `X1` is column `vnum` of `X`, the data curve plots `y` versus `X1` after removing the effects of the other predictors specified by the vector `inmodel`. The solid line is a least squares fit to the data curve, and its slope is the coefficient that `X1` would have if it were included in the model. The dotted lines are 95% confidence bounds for the fitted line, which you can use to judge the significance of `X1`.

**Example** The following code performs a stepwise regression on the Hald data, and creates an added variable plot for the predictor in column 2.

```
load hald
[b,se,p,inmodel,stats] = stepwisefit(ingredients,heat);
addedvarplot(ingredients,heat,2,inmodel,stats)
```



## See Also

`stepwisefit`

# andrewsplot

---

**Purpose** Andrews plot for multivariate data.

**Syntax**

```
andrewsplot(X)
andrewsplot(X, ..., 'Standardize', 'on')
andrewsplot(X, ..., 'Standardize', 'PCA')
andrewsplot(X, ..., 'Standardize', 'PCAStd')
andrewsplot(X, ..., 'Quantile', alpha)
andrewsplot(X, ..., 'Group', group)
andrewsplot(X, ..., 'PropertyName', PropertyValue, ...)
```

**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. An Andrews plot is a tool for visualizing high-dimensional data, where each observation is represented 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

$$f(t) = X(i, 1)/\sqrt{2} + X(i, 2)\sin 2\pi t + X(i, 3)\cos 2\pi t + \dots$$

`andrewsplot` treats NaNs 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.

`andrewsplot(X, ..., 'Standardize', 'PCAStd')` creates an Andrews plot using the standardized principal component scores.

`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 character matrix or a cell array of strings, containing a group name for each observation.



`andrewsplot(X, ..., 'PropertyName', PropertyValue, ...)` sets properties to the specified property values for all line graphics objects created by `andrewsplot`.

`h = andrewsplot(X, ...)` returns a column vector of handles to the line 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 lines 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 raw data
load fisheriris
andrewsplot(meas, 'group',species);

% plot only the median and quartiles of each group
andrewsplot(meas, 'group',species, 'quantile',.25);
```

## See Also

`parallelcoords`, `glyphplot`

# anova1

---

**Purpose** One-way analysis of variance (ANOVA)

**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 a balanced one-way ANOVA for comparing the means of two or more columns of data in the  $m$ -by- $n$  matrix  $X$ , where each column represents an independent sample containing  $m$  mutually independent observations. The function returns the p-value for the null hypothesis that all samples in  $X$  are drawn from the same population (or from different populations with the same mean).

If the p-value is near zero, this casts doubt on the null hypothesis and suggests that at least one sample mean is significantly different than the other sample means. 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 `anova1` function displays two figures. The first figure is the standard ANOVA table, which divides the variability of the data in  $X$  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 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) for each source, which is the ratio  $SS/df$ .
- The fifth shows the  $F$  statistic, which is the ratio of the MS's.

- The sixth shows the p-value, which is derived from the cdf of  $F$ . As  $F$  increases, the p-value decreases.

The second figure displays box plots of each column of  $X$ . Large differences in the center lines of the box plots correspond to large values of  $F$  and correspondingly small p-values.

`p = anova1(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  $X$ .

When  $X$  is a vector, `anova1` performs a one-way ANOVA on the samples contained in  $X$ , as indexed by input `group` (a 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. The vector-input form of `anova1` does not require equal numbers of observations in each sample, so it is appropriate for unbalanced data.

It is not necessary to label samples sequentially (1, 2, 3, ...). For example, if  $X$  contains measurements taken at three different temperatures,  $-27^\circ$ ,  $65^\circ$ , and  $110^\circ$ , 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 = 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'`.

`[p,table] = anova1(...)` 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] = anova1(...)` returns a `stats` structure that you can use to perform a follow-up multiple comparison test. The `anova1` test 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 to modest violations of the first two assumptions.

## Examples

### Example 1

The five columns of `X` are the constants one through five plus a random normal disturbance 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 =
```

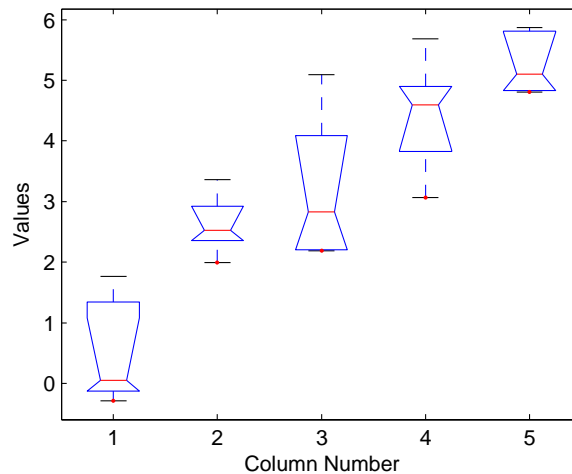
```
-0.0741    2.7782    2.2129    4.0802    5.7902
 1.2018    1.9937    3.7520    3.0627    5.1053
 1.7629    2.5245    2.8331    4.6357    4.8414
-0.2882    3.3643    2.1838    5.6820    5.8709
 0.0470    2.4820    5.0941    4.5936    4.8052
```

```
p = anova1(X)
```

```
p =
```

```
4.0889e-007
```

Source	SS	df	MS	F	Prob>F
Columns	66.294	4	16.5735	22.16	4.08898e-007
Error	14.9553	20	0.7478		
Total	81.2493	24			



The very small p-value of  $6e-5$  indicates that differences between the column means are highly significant. The probability of this outcome under the null hypothesis (i.e., the probability that samples actually drawn from the same population would have means differing by the amounts seen in X) is less than 6 in 100,000. The test therefore strongly supports the alternate hypothesis, that one or more of the samples are drawn from populations with different means.

### Example 2

The following example comes from a study of the material strength of structural beams in Hogg (1987). The vector strength measures the deflection of a beam in thousandths of an inch under 3,000 pounds of force. Stronger beams deflect less. The civil engineer performing the study wanted to determine whether the strength of steel beams was equal to the strength of two

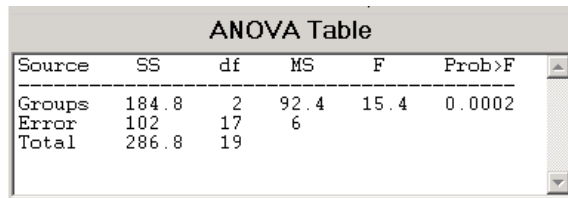
# anova1

more expensive alloys. Steel is coded 'st' in the vector alloy. The other materials are coded 'a11' and 'a12'.

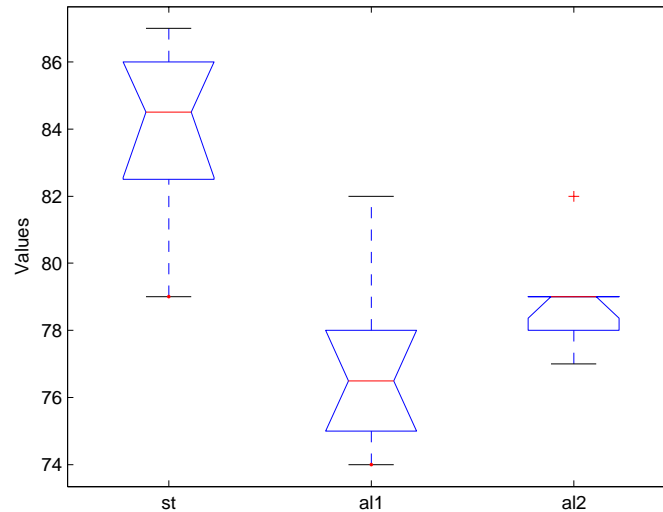
```
strength = [82 86 79 83 84 85 86 87 74 82 78 75 76 77 79 ...  
           79 77 78 82 79];  
  
alloy = {'st','st','st','st','st','st','st','st',...  
        'a11','a11','a11','a11','a11','a11',...  
        'a12','a12','a12','a12','a12','a12'};
```

Though alloy is sorted in this example, you do not need to sort the grouping variable.

```
p = anova1(strength,alloy)  
  
p =  
  
1.5264e-004
```



Source	SS	df	MS	F	Prob>F
Groups	184.8	2	92.4	15.4	0.0002
Error	102	17	6		
Total	286.8	19			



The p-value indicates that the three alloys are significantly different. The box plot confirms this graphically and shows that the steel beams deflect more than the more expensive alloys.

## References

[1] Hogg, R. V., and J. Ledolter. *Engineering Statistics*. MacMillan Publishing Company, 1987.

## See Also

anova2, anovan, boxplot, ttest

# anova2

**Purpose** Two-way Analysis of Variance (ANOVA)

**Syntax**

```
p = anova2(X, reps)
p = anova2(X, reps, 'displayopt')
[p, table] = anova2(...)
[p, table, stats] = anova2(...)
```

**Description** `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 “cell,” 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.

$$\begin{array}{cc} \begin{array}{c} \text{1} \\ \text{=} \\ \text{A} \end{array} & \begin{array}{c} \text{2} \\ \text{=} \\ \text{A} \end{array} \\ \left[ \begin{array}{cc} x_{111} & x_{121} \\ x_{112} & x_{122} \\ x_{211} & x_{221} \\ x_{212} & x_{222} \\ x_{311} & x_{321} \\ x_{312} & x_{322} \end{array} \right] & \left. \begin{array}{l} \left. \begin{array}{l} \left. \begin{array}{l} \\ \\ \end{array} \right\} \text{B} = 1 \\ \left. \begin{array}{l} \\ \\ \end{array} \right\} \text{B} = 2 \\ \left. \begin{array}{l} \\ \\ \end{array} \right\} \text{B} = 3 \end{array} \right\} \end{array}$$

When `reps` is 1 (default), `anova2` returns two p-values in vector `p`:

- 1 The p-value for the null hypothesis,  $H_{0A}$ , that all samples from factor A (i.e., all column-samples in  $X$ ) are drawn from the same population
- 2 The p-value for the null hypothesis,  $H_{0B}$ , that all samples from factor B (i.e., all row-samples in  $X$ ) are drawn from the same population

When `reps` is greater than 1, `anova2` returns a third p-value in vector `p`:



- 3 The p-value for the null hypothesis,  $H_{0AB}$ , that the effects due to factors A and B are *additive* (i.e., that there is no interaction between factors A and B)

If any p-value is near zero, this casts doubt on the associated null hypothesis. A sufficiently small p-value for  $H_{0A}$  suggests that at least one column-sample mean is significantly different than 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'.

## anova2

---

[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
```

Source	SS	df	MS	F	Prob>F
Columns	15.75	2	7.875	56.7	0
Rows	4.5	1	4.5	32.4	0.0001
Interaction	0.0833	2	0.04167	0.3	0.7462
Error	1.6667	12	0.13889		
Total	22	17			

The vector `p` shows the p-values for the three brands of popcorn, 0.0000, the two popper types, 0.0001, and the interaction between brand and popper type, 0.7462. These values indicate that both popcorn brand and popper type affect the yield of popcorn, but there is no evidence of a synergistic (interaction) effect of the two.

The conclusion is that you can get the greatest yield using the Gourmet brand and an Air popper (the three values `popcorn(4:6, 1)`).

## Reference

[1] Hogg, R. V. and J. Ledolter. *Engineering Statistics*. MacMillan Publishing Company, 1987.

## See Also

`anova1`, `anovan`

# anovan

---

**Purpose** N-way analysis of variance (ANOVA)

**Syntax**

```
p = anovan(x,group)
p = anovan(x,group,'Param1',val1,'Param2',val2,...)
[p,table] = anovan(...)
[p,table,stats] = anovan(...)
[p,table,stats,terms] = anovan(...)
```

**Description** `p = anovan(X,group)` performs a balanced or unbalanced multiway analysis of variance (ANOVA) for comparing the means of the observations in the vector `X` with respect to `N` different factors. The factors and factor levels of the observations in `X` are assigned by the cell array `group`. Each of the `N` cells in `group` contains a list of factor levels identifying the observations in `X` with respect to one of the `N` factors. The list within each cell can be a vector, character array, or cell array of strings, and must have the same number of elements as `X`. For an example, see “Example of Three-Way ANOVA” on page 12-38.

`P = anovan(x,group,'Param1',val1,'Param2',val2,...)` specifies one or more of the name/value pairs described in the following table.

Parameter Name	Parameter Value
'sstype'	1, 2, or 3, to specify the type of sum of squares (default = 3)
'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 4-18 for an example of how to use 'varnames'.
'display'	'on' displays an ANOVA table (the default) 'off' omits the display

Parameter Name	Parameter Value
'random'	A vector of indices indicating which grouping variables are random effects (all are fixed by default). See “ANOVA with Random Effects” on page 4-18 for an example of how to use 'random'.
'alpha'	A number between 0 and 1 requesting 100(1 - alpha)% confidence bounds (default 0.05 for 95% confidence)
'model'	The type of model used. See “Model Types” on page 12-35 for a description of the options for this argument.

[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 12-40 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.

The rest of this section explains the following topics:

- “Model Types” on page 12-35
- “Example of Three-Way ANOVA” on page 12-38
- “The stats Structure” on page 12-40

## Model Types

This section explains how to use the argument 'model' with the syntax

```
p = anovan(x,group,'model', modeltype)
```

The argument `modeltype`, which specifies the type of model the function uses, can be any one of the following:

- 'linear' — The default 'linear' model computes only the p-values for the null hypotheses on the  $N$  main effects.
- 'interaction' — The 'interaction' model computes the p-values for null hypotheses on the  $N$  main effects and the  $\binom{N}{2}$  two-factor interactions.
- 'full' — The 'full' model computes the p-values for null hypotheses on the  $N$  main effects and interactions at all levels.
- An integer — For an integer value of `modeltype`,  $k$  ( $k \leq N$ ), `anovan` computes all interaction levels through the  $k$ th level. For example, the value 3 means main effects plus two- and three-factor interactions. The values  $k=1$  and  $k=2$  are equivalent to the 'linear' and 'interaction' specifications, respectively, while the value  $k=N$  is equivalent to the 'full' specification.
- A matrix of term definitions having the same form as the input to the `x2fx` function. All entries must be 0 or 1 (no higher powers).

For more precise control over the main and interaction terms that `anovan` computes, `modeltype` can specify a matrix containing one row for each main or interaction term to include in the ANOVA model. Each row defines one term using a vector of  $N$  zeros and ones. The table below illustrates the coding for a 3-factor ANOVA.

Row of Matrix	Corresponding ANOVA Term
[ 1 0 0 ]	Main term A
[ 0 1 0 ]	Main term B
[ 0 0 1 ]	Main term C
[ 1 1 0 ]	Interaction term AB
[ 0 1 1 ]	Interaction term BC
[ 1 0 1 ]	Interaction term AC
[ 1 1 1 ]	Interaction term ABC

For example, if `modeltype` is the matrix  $[0 \ 1 \ 0; 0 \ 0 \ 1; 0 \ 1 \ 1]$ , the output vector `p` contains the p-values for the null hypotheses on the main effects B and C and the interaction effect BC, in that order. A simple way to generate the `modeltype` matrix is to modify the `terms` output, which codes the terms in the current model using the format described above. If `anovan` returns  $[0 \ 1 \ 0; 0 \ 0 \ 1; 0 \ 1 \ 1]$  for `terms`, for example, and there is no significant result for interaction BC, you can recompute the ANOVA on just the main effects B and C by specifying  $[0 \ 1 \ 0; 0 \ 0 \ 1]$  for `modeltype`.

`p = anovan(x,group,'sstype', integer)` computes the ANOVA using the type of sum-of-squares specified by `integer`, which can be 1, 2, or 3 to designate Type 1, Type 2, or Type 3 sum-of-squares, respectively. The default is 3. The value of `integer` 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:

<b>Term</b>	<b>Type 1 SS</b>	<b>Type 2 SS</b>	<b>Type 3 SS</b>
A	$R(1)-R(A)$	$R(B)-R(A,B)$	$R(B,AB)-R(A,B,AB)$
B	$R(A)-R(A,B)$	$R(A)-R(A,B)$	$R(A,AB)-R(A,B,AB)$
AB	$R(A,B)-R(A,B,AB)$	$R(A,B)-R(A,B,AB)$	$R(A,B)-R(A,B,AB)$

The models for Type 3 sum of squares have sigma restrictions imposed. This means, for example, that in fitting  $R(B,AB)$ , the array of AB effects is constrained to sum to 0 over A for each value of B, and over B for each value of A.

## Example of Three-Way ANOVA

As an example of three-way ANOVA, consider the vector  $y$  and group inputs below.

```
y = [52.7 57.5 45.9 44.5 53.0 57.0 45.9 44.0]';  
g1 = [1 2 1 2 1 2 1 2];  
g2 = {'hi'; 'hi'; 'lo'; 'lo'; 'hi'; 'hi'; 'lo'; 'lo'};  
g3 = {'may'; 'may'; 'may'; 'may'; 'june'; 'june'; 'june'; 'june'};
```

This defines a three-way ANOVA with two levels of each factor. Every observation in  $y$  is identified by a combination of factor levels. If the factors are A, B, and C, then observation  $y(1)$  is associated with

- Level 1 of factor A
- Level 'hi' of factor B
- Level 'may' of factor C

Similarly, observation  $y(6)$  is associated with

- Level 2 of factor A
- Level 'hi' of factor B
- Level 'june' of factor C

To compute the ANOVA, enter

```
p = anovan(y, {g1 g2 g3})  
  
p =  
    0.4174  
    0.0028  
    0.9140
```

Output vector  $p$  contains p-values for the null hypotheses on the  $N$  main effects. Element  $p(1)$  contains the p-value for the null hypotheses,  $H_{0A}$ , that samples at all levels of factor A are drawn from the same population;



element  $p(2)$  contains the p-value for the null hypotheses,  $H_{0B}$ , that samples at all levels of factor B are drawn from the same population; and so on.

If any p-value is near zero, this casts doubt on the associated null hypothesis. For example, a sufficiently small p-value for  $H_{0A}$  suggests that at least one A-sample mean is significantly different from the other A-sample means; that is, there is a main effect due to factor A. You need to choose a bound for the p-value to determine whether a result is statistically significant. It is common to declare a result significant if the p-value is less than 0.05 or 0.01.

anovan also displays a figure showing the standard ANOVA table, which by default divides the variability of the data in  $x$  into

- The variability due to differences between the levels of each factor accounted for in the model (one row for each factor)
- The remaining variability not explained by any systematic source

The ANOVA table has six columns:

- The first shows the source of the variability.
- The second shows the sum of squares (SS) due to each source.
- The third shows the degrees of freedom (df) associated with each source.
- The fourth shows the mean squares (MS), which is the ratio  $SS/df$ .
- The fifth shows the F statistics, which are the ratios of the mean squares.
- The sixth shows the p-values for the F statistics.

The table is shown in the following figure:

Source	Sum Sq.	d.f.	Mean Sq.	F	Prob>F
X1	3.781	1	3.781	0.82	0.4174
X2	199.001	1	199.001	42.95	0.0028
X3	0.061	1	0.061	0.01	0.914
Error	18.535	4	4.634		
Total	221.379	7			

Constrained (Type III) sums of squares.

**Two-Factor Interactions.** By default, anovan computes p-values just for the three main effects. To also compute p-values for the two-factor interactions,  $X1 * X2$ ,

## anovan

X1\*X3, and X2\*X3, add the name/value pair 'model', 'interaction' as input arguments.

```
p = anovan(y, {g1 g2 g3}, 'model', 'interaction')
```

```
p =
```

```
0.0347
```

```
0.0048
```

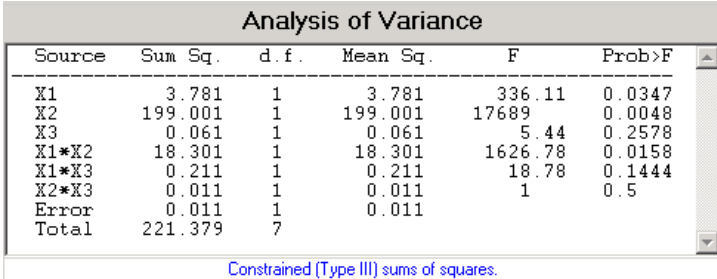
```
0.2578
```

```
0.0158
```

```
0.1444
```

```
0.5000
```

The first three entries of `p` are the p-values for the main effects. The last three entries are the p-values for the two-factor interactions. You can determine the order in which the two-factor interactions occur from the ANOVAN table shown in the following figure.



Source	Sum Sq.	d. f.	Mean Sq.	F	Prob>F
X1	3.781	1	3.781	336.11	0.0347
X2	199.001	1	199.001	17689	0.0048
X3	0.061	1	0.061	5.44	0.2578
X1*X2	18.301	1	18.301	1626.78	0.0158
X1*X3	0.211	1	0.211	18.78	0.1444
X2*X3	0.011	1	0.011	1	0.5
Error	0.011	1	0.011		
Total	221.379	7			

Constrained (Type III) sums of squares.

### The stats Structure

The `anovan` test evaluates the hypothesis that the different levels of a factor (or more generally, a term) have the same effect, against the alternative that they do not all have the same effect. Sometimes it is preferable to perform a test to determine which pairs of levels are significantly different, and which are not. Use the `multcompare` function to perform such tests by supplying the `stats` structure as input.

The `stats` structure contains the fields listed below, in addition to a number of other fields required for doing multiple comparisons using the `multcompare` function:

<b>Stats Field</b>	<b>Meaning</b>
<code>coeffs</code>	Estimated coefficients
<code>coeffnames</code>	Name of term for each coefficient
<code>vars</code>	Matrix of grouping variable values for each term
<code>resid</code>	Residuals from the fitted model

The `stats` structure also contains the following fields if there are random effects:

<b>Stats Field</b>	<b>Meaning</b>
<code>ems</code>	Expected mean squares
<code>denom</code>	Denominator definition
<code>rtnames</code>	Names of random terms
<code>varest</code>	Variance component estimates (one per random term)
<code>varci</code>	Confidence intervals for variance components

## Examples

“Two-Way Analysis of Variance (ANOVA)” on page 4-8 shows how to use `anova2` to analyze the effects of two factors on a response in a balanced design. For a design that is not balanced, use `anovan` instead.

In this example, the data set `carbig` contains a number of measurements on 406 cars. You can use `anonvan` to study how the mileage depends on where and when the cars were made.

```
load carbig
```

## anovan

```
anovan(MPG,{org when},2,3,{'Origin';'Mfg date'})  
  
ans =  
      0  
      0  
      0.3059
```

The p-value for the interaction term is not small, indicating little evidence that the effect of the car's year or manufacture (when) depends on where the car was made (org). The linear effects of those two factors, though, are significant.

Analysis of Variance					
Source	Sum Sq.	d.f.	Mean Sq.	F	Prob>F
Origin	5727.2	2	2863.58	115.09	0
Mfg date	4710.3	2	2355.15	94.65	0
Origin*Mfg date	120.5	4	30.12	1.21	0.3059
Error	9679.1	389	24.88		
Total	24252.6	397			

Constrained (Type III) sums of squares.

### Reference

[1] Hogg, R. V., and J. Ledolter. *Engineering Statistics*, MacMillan Publishing Company, 1987.

### See Also

anova1, anova2, multcompare

**Purpose** Interactive plot for fitting and predicting analysis of covariance models

**Syntax**

```
aoctool(x,y,g)
aoctool(x,y,g,alpha)
aoctool(x,y,g,alpha,xname,yname,gname)
aoctool(x,y,g,alpha,xname,yname,gname,'displayopt')
aoctool(x,y,g,alpha,xname,yname,gname,'displayopt','model')
h = aoctool(...)
[h,atab,ctab] = aoctool(...)
[h,atab,ctab,stats] = aoctool(...)
```

**Description** `aoctool(x,y,g)` fits a separate line to the column vectors, `x` and `y`, for each group defined by the values in the array `g`. These types of models are known as one-way analysis of covariance (ANOCOVA) models. The output consists of three figures:

- An interactive graph of the data and prediction curves
- An ANOVA table
- A table of parameter estimates

You can use the figures to change models and to test different parts of the model. More information about interactive use of the `aoctool` function appears in “The `aoctool` Demo” on page 4-25.

`aoctool(x,y,g,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,g,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,g,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,g,alpha,xname,yname,gname,'displayopt','model')` specifies the initial model to fit. The value of `'model'` can be any of the following:

- `'same mean'` – fit a single mean, ignoring grouping
- `'separate means'` – fit a separate mean to each group
- `'same line'` – fit a single line, ignoring grouping
- `'parallel lines'` – fit a separate line to each group, but constrain the lines to be parallel
- `'separate lines'` – fit a separate line to each group, with no constraints

`h = aoctool(...)` returns a vector of handles to the line objects in the plot.

`[h,atab,ctab] = aoctool(...)` returns cell arrays containing the entries in ANOVA table (`atab`) and the table of coefficient estimates (`ctab`). (You can copy a text version of either table to the clipboard by using the **Copy Text** item on the **Edit** menu.)

`[h,atab,ctab,stats] = aoctool(...)` returns a `stats` structure that you can use to perform a follow-up multiple comparison test. The ANOVA table output includes tests of the hypotheses that the slopes or intercepts are all the same, against a general alternative that they are not all the same. Sometimes it is preferable to perform a test to determine which pairs of values are significantly different, and which are not. You can use the `multcompare` function to perform such tests by supplying the `stats` structure as input. You can test either the slopes, the intercepts, or population marginal means (the heights of the curves at the mean `x` value).

## Example

This example illustrates how to fit different models non-interactively. After loading the smaller car data set and fitting a separate-slopes model, you can examine the coefficient estimates.

```
load carsmall
[h,a,c,s] = aoctool(Weight,MPG,Model_Year,0.05,...
    '',' ',' ','off','separate lines');
c(:,1:2)
```

```
ans =
  'Term'      'Estimate'
  'Intercept' [45.97983716833132]
  ' 70'      [-8.58050531454973]
  ' 76'      [-3.89017396094922]
  ' 82'      [12.47067927549897]
  'Slope'     [-0.00780212907455]
  ' 70'      [ 0.00195840368824]
  ' 76'      [ 0.00113831038418]
  ' 82'      [-0.00309671407243]
```

Roughly speaking, the lines relating MPG to Weight have an intercept close to 45.98 and a slope close to -0.0078. Each group's coefficients are offset from these values somewhat. For instance, the intercept for the cars made in 1970 is  $45.98 - 8.58 = 37.40$ .

Next, try a fit using parallel lines. (The ANOVA table shows that the parallel-lines fit is significantly worse than the separate-lines fit.)

```
[h,a,c,s] = aoctool(Weight,MPG,Model_Year,0.05,...
                    ' ',' ',' ','off','parallel lines');
c(:,1:2)

ans =

  'Term'      'Estimate'
  'Intercept' [43.38984085130596]
  ' 70'      [-3.27948192983761]
  ' 76'      [-1.35036234809006]
  ' 82'      [ 4.62984427792768]
  'Slope'     [-0.00664751826198]
```

Again, there are different intercepts for each group, but this time the slopes are constrained to be the same.

**See Also**

anova1, multcompare, polytool

# barttest

---

**Purpose** Bartlett's test for dimensionality

**Syntax**  
`ndim = barttest(x,alpha)`  
`[ndim,prob,chisquare] = barttest(x,alpha)`

**Description** `ndim = barttest(x,alpha)` returns the number of dimensions necessary to explain the nonrandom variation in the data matrix `x`, using the significance probability `alpha`. The dimension is determined by a series of hypothesis tests. The test for `ndim=1` tests the hypothesis that the variances of the data values along each principal component are equal, the test for `ndim=2` tests the hypothesis that the variances along the second through last components are equal, and so on.

`[ndim,prob,chisquare] = barttest(x,alpha)` returns the number of dimensions, the significance values for the hypothesis tests, and the  $\chi^2$  values associated with the tests.

**Example**

```
x = mvnrnd([0 0],[1 0.99; 0.99 1],20);
x(:,3:4) = mvnrnd([0 0],[1 0.99; 0.99 1],20);
x(:,5:6) = mvnrnd([0 0],[1 0.99; 0.99 1],20);
[ndim, prob] = barttest(x,0.05)

ndim =
     3

prob =
     0
     0
     0
    0.5081
    0.6618
```

**See Also** `princomp`, `pcacov`, `pcares`



<b>Purpose</b>	Generate Box-Behnken design
<b>Syntax</b>	<pre>D = bbdesign(nfactors) D = bbdesign(nfactors, 'pname1', pvalue1, 'pname2', pvalue2, ...) [D, blk] = bbdesign(...)</pre>
<b>Description</b>	<p><code>D = bbdesign(nfactors)</code> generates a Box-Behnken design for <code>nfactors</code> factors. The output matrix <code>D</code> is <code>n</code>-by-<code>nfactors</code>, where <code>n</code> is the number of points in the design. Each row lists the settings for all factors, scaled between -1 and 1.</p> <p><code>[D, blk] = bbdesign(nfactors)</code> requests a blocked design. The output vector <code>blk</code> is a vector of block numbers. Blocks are groups of runs that are to be measured under similar conditions (for example, on the same day). Blocked designs minimize the effect of between-block differences on the parameter estimates.</p> <p><code>[...] = bbdesign(nfactors, 'pname1', pvalue1, 'pname2', pvalue2, ...)</code> allows you to specify additional parameters and their values. Valid parameters are:</p> <ul style="list-style-type: none"><li>'center'      Number of center points to include.</li><li>'blocksize'    Maximum number of points allowed in a block.</li></ul>
<b>Remarks</b>	Box and Behnken proposed designs when the number of factors was equal to 3-7, 9-12, or 16. This function produces those designs. For other values of <code>nfactors</code> , this function produces designs that are constructed in a similar way, even though they were not tabulated by Box and Behnken, and they may be too large to be practical.
<b>See Also</b>	<code>ccdesign</code> , <code>cordexch</code> , <code>rowexch</code>

# betacdf

---

**Purpose** Beta cumulative distribution function (cdf)

**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. The result, `p`, is the probability that a single observation from a beta distribution with parameters `a` and `b` will fall in the interval `[0, x]`.

## Examples

```
x = 0.1:0.2:0.9;
a = 2;
b = 2;
p = betacdf(x,a,b)

p =

    0.0280    0.2160    0.5000    0.7840    0.9720

a = [1 2 3];
p = betacdf(0.5,a,a)

p =

    0.5000    0.5000    0.5000
```

**See Also** `betafit`, `betainv`, `betalike`, `betapdf`, `betarnd`, `betastat`, `cdf`

**Purpose** Parameter estimates and confidence intervals for beta distributed data

**Syntax**  
`phat = betafit(data)`  
`[phat,pci] = betafit(data,alpha)`

**Description** `phat = betafit(data)` computes the maximum likelihood estimates of the beta distribution parameters  $a$  and  $b$  from the data in the vector `data`, where the beta cdf is given by

$$F(x|a, b) = \frac{1}{B(a, b)} \int_0^x t^{a-1}(1-t)^{b-1} dt$$

and  $B(\cdot)$  is the Beta function. The elements of `data` must lie in the interval  $(0, 1)$ .

`[phat,pci] = betafit(data,alpha)` returns confidence intervals on the  $a$  and  $b$  parameters in the 2-by-2 matrix `pci`. The first column of the matrix contains the lower and upper confidence bounds for parameter  $a$ , and the second column contains the confidence bounds for parameter  $b$ . The optional input argument `alpha` is a value in the range  $[0, 1]$  specifying the width of the confidence intervals. By default, `alpha` is 0.05, which corresponds to 95% confidence intervals.

**Example** This example generates 100 beta distributed observations. The true  $a$  and  $b$  parameters are 4 and 3, respectively. Compare these to the values returned in `p`. Note that the columns of `ci` both bracket the true parameters.

```
data = betarnd(4,3,100,1);
[p,ci] = betafit(data,0.01)
```

```
p =
    3.9010    2.6193
```

```
ci =
    2.5244    1.7488
    5.2776    3.4898
```

**Reference** [1] Hahn, Gerald J., & Shapiro, Samuel, S. *Statistical Models in Engineering*. John Wiley & Sons, New York. 1994. p. 95.

# betafit

---

## See Also

betalike, mle

**Purpose** Inverse of the beta cumulative distribution function

**Syntax** `X = betainv(P,A,B)`

**Description** `X = betainv(P,A,B)` computes the inverse of the beta cdf with parameters specified by A and B for the corresponding probabilities in P. P, A, and B can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs. The parameters in A and B must all be positive, and the values in P must lie on the interval [0 1].

The inverse beta cdf for a given probability  $p$  and a given pair of parameters  $a$  and  $b$  is

$$x = F^{-1}(p|a, b) = \{x:F(x|a, b) = p\}$$

where

$$p = F(x|a, b) = \frac{1}{B(a, b)} \int_0^x t^{a-1}(1-t)^{b-1} dt$$

and  $B(\cdot)$  is the Beta function. Each element of output X is the value whose cumulative probability under the beta cdf defined by the corresponding parameters in A and B is specified by the corresponding value in P.

**Algorithm** The `betainv` function uses Newton's method with modifications to constrain steps to the allowable range for  $x$ , i.e., [0 1].

**Examples**

```
p = [0.01 0.5 0.99];
x = betainv(p,10,5)
```

```
x =
    0.3726    0.6742    0.8981
```

According to this result, for a beta cdf with  $a=10$  and  $b=5$ , a value less than or equal to 0.3726 occurs with probability 0.01. Similarly, values less than or equal to 0.6742 and 0.8981 occur with respective probabilities 0.5 and 0.99.

**See Also** `betafit`, `icdf`

# betalike

---

**Purpose** Negative beta log-likelihood function

**Syntax** `logL = betalike(params,data)`  
`[logL,AVAR] = betalike(params,data)`

**Description** `logL = 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 `logL` is the length of `data`.

`[logL,AVAR] = betalike(params,data)` also returns `AVAR`, which is the asymptotic variance-covariance matrix of the parameter estimates if the values in `params` are the maximum likelihood estimates. `AVAR` is the inverse of Fisher's information matrix. The diagonal elements of `AVAR` are the asymptotic variances of their respective parameters.

`betalike` is a utility function for maximum likelihood estimation of the beta distribution. The likelihood assumes that all the elements in the data sample are mutually independent. Since `betalike` returns the negative beta log-likelihood function, minimizing `betalike` using `fminsearch` is the same as maximizing the likelihood.

**Example** This example continues the `betafit` example, which calculates estimates of the beta parameters for some randomly generated beta distributed data.

```
r = betarnd(4,3,100,1);
[logl,AVAR] = betalike(betafit(r),r)

logl =

    -39.1615

AVAR =

    0.3717    0.2644
    0.2644    0.2414
```

**See Also** `betafit`, `fminsearch`, `gamlike`, `mle`, `normlike`, `wbllike`

**Purpose** Beta probability density function (pdf)

**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 result,  $y$ , is the probability that a single observation from a beta distribution with parameters  $a$  and  $b$  will have value  $x$ . The indicator function  $I_{(0,1)}(x)$  ensures that only values of  $x$  in the range  $(0, 1)$  have nonzero probability. The uniform distribution on  $(0, 1)$  is a degenerate case of the beta pdf where  $a = 1$  and  $b = 1$ .

A *likelihood function* is the pdf viewed as a function of the parameters. Maximum likelihood estimators (MLEs) are the values of the parameters that maximize the likelihood function for a fixed value of  $x$ .

## Examples

```
a = [0.5 1; 2 4]
a =
    0.5000    1.0000
    2.0000    4.0000

y = betapdf(0.5,a,a)
y =
    0.6366    1.0000
    1.5000    2.1875
```

## See Also

`betacdf`, `betafit`, `betainv`, `betalike`, `betarnd`, `betastat`, `pdf`

# betarnd

---

**Purpose** Random numbers from the beta distribution

**Syntax**

```
R = betarnd(A,B)
R = betarnd(A,B,v)
R = betarnd(A,B,m,n)
R = betarnd(A,B,m,n,o,...)
```

**Description** `R = betarnd(A,B)` generates random numbers from the beta distribution with parameters specified by `A` and `B`. `A` and `B` can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of `R`. A scalar input for `A` or `B` is expanded to a constant array with the same dimensions as the other input.

`R = betarnd(A,B,v)` generates an array `R` of size `v` containing random numbers from the beta distribution with parameters `A` and `B`, where `v` is a row vector. If `v` is a 1-by-2 vector, `R` is a matrix with `v(1)` rows and `v(2)` columns. If `v` is 1-by-`n`, `R` is an `n`-dimensional array.

`R = betarnd(A,B,m,n)` generates an `m`-by-`n` matrix containing random numbers from the beta distribution with parameters `A` and `B`.

`R = betarnd(A,B,m,n,o,...)` generates an `m`-by-`n`-by-`o`-by-... multidimensional array containing random numbers from the beta distribution with parameters `A` and `B`.

## Reproducing the Output of betarnd

`betarnd` uses the MATLAB functions `rand` and `randn` to generate random numbers. When you call `betarnd`, you change the current states of `rand` and `randn`, and thereby alter the output of subsequent calls to `betarnd` or any other functions that depend on `rand` or `randn`. If you want to reproduce the output of `betarnd`, reset the states of `rand` and `randn` to the same fixed values each time you call `betarnd`. For an example of how to do this, and a list of the Statistics Toolbox functions that depend on `rand` or `randn`, see “Reproducing the Output of Random Number Functions” on page 2-46.



---

**Note** The results in the following examples depend on the current states of `rand` and `randn`. If you run the code in these examples, your results may differ from the answers shown here.

---

## 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

`betacdf`, `betafit`, `betainv`, `betalike`, `betapdf`, `betastat`, `rand`, `randn`, `randtool`

# betastat

---

**Purpose** Mean and variance for the beta distribution

**Syntax** [M,V] = betastat(A,B)

**Description** [M,V] = betastat(A,B) returns the mean and variance for the beta distribution with parameters specified by A and B. A and B can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of M and V. A scalar input for A or B is expanded to a constant array with the same dimensions as the other input.

The mean of the beta distribution with parameters  $a$  and  $b$  is  $a/(a+b)$  and the variance is

$$\frac{ab}{(a+b+1)(a+b)^2}$$

**Examples** If parameters  $a$  and  $b$  are equal, the mean is 1/2.

```
a = 1:6;  
[m,v] = betastat(a,a)
```

```
m =  
    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000  
v =  
    0.0833    0.0500    0.0357    0.0278    0.0227    0.0192
```

**See Also** betacdf, betafit, betainv, betalike, betapdf, betarnd

**Purpose** Binomial cumulative distribution function (cdf)

**Syntax** `Y = binocdf(X,N,P)`

**Description** `binocdf(X,N,P)` computes a binomial cdf at each of the values in  $X$  using the corresponding parameters in  $N$  and  $P$ .  $X$ ,  $N$ , and  $P$  can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions of the other inputs. The values in  $N$  must all be positive integers, the values in  $X$  must lie on the interval  $[0, N]$ , and the values in  $P$  must lie on the interval  $[0, 1]$ .

The binomial cdf for a given value  $x$  and given pair of parameters  $n$  and  $p$  is

$$y = F(x|n, p) = \sum_{i=0}^x \binom{n}{i} p^i q^{(n-i)} I_{(0, 1, \dots, n)}(i)$$

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  $I_{(0, 1, \dots, n)}(i)$  ensures that  $x$  only adopts values of  $0, 1, \dots, n$ .

**Examples** If a baseball team plays 162 games in a season and has a 50-50 chance of winning any game, then the probability of that team winning more than 100 games in a season is:

```
1 - binocdf(100,162,0.5)
```

The result is 0.001 (i.e., 1-0.999). If a team wins 100 or more games in a season, this result suggests that it is likely that the team's true probability of winning any game is greater than 0.5.

**See Also** `binofit`, `binoinv`, `binopdf`, `binornd`, `binostat`, `cdf`

# binofit

---

**Purpose** Parameter estimates and confidence intervals for binomial data

**Syntax**

```
phat = binofit(x,n)
[phat,pci] = binofit(x,n)
[phat,pci] = binofit(x,n,alpha)
```

**Description**

`phat = binofit(x,n)` returns a maximum likelihood estimate of the probability of success in a given binomial trial based on the number of successes,  $x$ , observed in  $n$  independent trials. If  $x = (x(1), x(2), \dots, x(k))$  is a vector, `binofit` returns a vector of the same size as  $x$  whose  $i$ th entry is the parameter estimate for  $x(i)$ . All  $k$  estimates are independent of each other. If  $n = (n(1), n(2), \dots, n(k))$  is a vector of the same size as  $x$ , `binofit` returns a vector whose  $i$ th 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  $100(1 - \text{alpha})\%$  confidence intervals. For example, `alpha = 0.01` yields 99% confidence intervals.

---

**Note** `binofit` behaves differently than other functions in the Statistics Toolbox that compute parameter estimates, in that it returns independent estimates for each entry of  $x$ . By comparison, `expfit` returns a single parameter estimate based on all the entries of  $x$ .

---

**Example**

This example generates a binomial sample of 100 elements, where the probability of success in a given trial is 0.6, and then estimates this probability from the outcomes in the sample.

```
r = binornd(100,0.6);
[phat,pci] = binofit(r,100)

phat =

    0.5800
```

```
pci =  
    0.4771    0.6780
```

The 95% confidence interval, `pci`, contains the true value, 0.6.

**Reference**

[1] Johnson, N. L., S. Kotz, and A.W. Kemp, “*Univariate Discrete Distributions, Second Edition*,” Wiley 1992. pp. 124–130.

**See Also**

`binocdf`, `binoinv`, `binopdf`, `binornd`, `binostat`, `mle`

# binoinv

---

**Purpose** Inverse of the binomial cumulative distribution function (cdf)

**Syntax** `X = binoinv(Y,N,P)`

**Description** `X = binoinv(Y,N,P)` returns the smallest integer  $X$  such that the binomial cdf evaluated at  $X$  is equal to or exceeds  $Y$ . You can think of  $Y$  as the probability of observing  $X$  successes in  $N$  independent trials where  $P$  is the probability of success in each trial. Each  $X$  is a positive integer less than or equal to  $N$ .

$Y$ ,  $N$ , and  $P$  can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs. The parameters in  $N$  must be positive integers, and the values in both  $P$  and  $Y$  must lie on the interval  $[0\ 1]$ .

**Examples** If a baseball team has a 50-50 chance of winning any game, what is a reasonable range of games this team might win over a season of 162 games?

```
binoinv([0.05 0.95],162,0.5)
```

```
ans =
```

```
71    91
```

This result means that in 90% of baseball seasons, a .500 team should win between 71 and 91 games.

**See Also** `binocdf`, `binofit`, `binopdf`, `binornd`, `binostat`, `icdf`

**Purpose** Binomial probability density function (pdf)

**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

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

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  $I_{(0,1,\dots,n)}(x)$  ensures that  $x$  only adopts values of  $0, 1, \dots, n$ .

**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?

```
y = binopdf([0:200],200,0.02);
[x,i] = max(y);
```

```
i
i =
    5
```

**See Also** `binocdf`, `binofit`, `binoinv`, `binornd`, `binostat`, `pdf`

# binornd

---

**Purpose** Random numbers from the binomial distribution

**Syntax**

```
R = binornd(N,P)
R = binornd(N,P,v)
R = binornd(N,P,m,n)
```

**Description** `R = binornd(N,P)` generates random numbers from the beta 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 beta 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.

## Reproducing the Output of binornd

`binornd` uses the MATLAB function `rand` to generate random numbers. When you call `binornd`, you change the current state of `rand`, and thereby alter the output of subsequent calls to `binornd` or any other functions that depend on `rand`. If you want to reproduce the output of `binornd`, reset the state of `rand` to the same fixed value each time you call `binornd`. For an example of how to do this, and a list of the Statistics Toolbox functions that depend on `rand`, see “Reproducing the Output of Random Number Functions” on page 2-46.

---

**Note** The results in the following examples depend on the current state of `rand`. If you run the code in these examples, your results may differ from the answers shown here.

---



**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**

`binocdf`, `binofit`, `binoinv`, `binopdf`, `binostat`, `rand`, `randtool`

# binostat

---

**Purpose** Mean and variance for the binomial distribution

**Syntax** `[M,V] = binostat(N,P)`

**Description** `[M,V] = binostat(N,P)` returns the mean and variance for the binomial distribution with parameters specified by `N` and `P`. `N` and `P` can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of `M` and `V`. A scalar input for `N` or `P` is expanded to a constant array with the same dimensions as the other input.

The mean of the binomial distribution with parameters  $n$  and  $p$  is  $np$ . The variance is  $npq$ , where  $q = 1-p$ .

## Examples

```
n = logspace(1,5,5)
n =
    10    100   1000  10000 100000

[m,v] = binostat(n,1./n)

m =
    1    1    1    1    1

v =
    0.9000    0.9900    0.9990    0.9999    1.0000

[m,v] = binostat(n,1/2)

m =
    5    50   500   5000 50000

v =
    1.0e+04 *
    0.0003    0.0025    0.0250    0.2500    2.5000
```

**See Also** `binocdf`, `binofit`, `binoinv`, `binopdf`, `binornd`

**Purpose** Biplot of variable/factor coefficients and scores

**Syntax**

```
biplot(coefs)
biplot(coefs, ..., 'Scores', scores)
biplot(coefs, ..., 'VarLabels', varlabs)
biplot(coefs, ..., 'Scores', scores, 'ObsLabels', obslabs)
biplot(coeffs, ..., 'PropertyName', PropertyValue, ...)
h = biplot(coefs, ...)
```

**Description** `biplot(coefs)` creates a biplot of the coefficients in the matrix `coefs`. The biplot is two dimensional if `coefs` has two columns or three dimensional if it has three columns. `coefs` usually contains principal component coefficients created with `princomp`, `pcacov`, or factor loadings estimated with `factoran`. The axes in the biplot represent the principal components or latent factors (columns of `coefs`), and the observed variables (rows of `coefs`) are represented as vectors.

`biplot(coefs, ..., 'Scores', scores)` plots both `coefs` and the scores in the matrix `scores` in the biplot. `scores` usually contains principal component scores created with `princomp` or factor scores estimated with `factoran`. Each observation (row of `scores`) is represented as a point in the biplot.

A biplot allows you to visualize the magnitude and sign of each variable's contribution to the first two or three principal components, and how each observation is represented in terms of those components.

`biplot` imposes a sign convention, forcing the element with largest magnitude in each column of `coefs` is positive.

`biplot(coefs, ..., 'VarLabels', varlabs)` labels each vector (variable) with the text in the character array or cell array `varlabs`.

`biplot(coefs, ..., 'Scores', scores, 'ObsLabels', obslabs)` labels each point (observation) with the text in the character array or cell array `obslabs`.

`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`. `h` contains, in order, handles corresponding to

# biplot

---

variables (line handles, followed by marker handles, followed by text handles), to observations (if present, marker handles followed by text handles), and to the axis lines.

## Example

```
load carsmall
x = [Acceleration Displacement Horsepower MPG Weight];
x = x(all(~isnan(x),2),:);
[coefs,score] = princomp(zscore(x));
vlabs = {'Accel','Disp','HP','MPG','Wgt'};
biplot(coefs(:,1:3), 'scores',score(:,1:3), 'varlabels',vlabs);
```

## See Also

factoran, princomp, pcacov, rotatefactors

**Purpose** Bootstrap statistics through resampling of data

**Syntax** `bootstat = bootstrp(nboot,bootfun,d1,d2,...)`  
`[bootstat,bootsam] = bootstrp(...)`

**Description** `bootstat = bootstrp(nboot,bootfun,d1,d2,...)` draws `nboot` bootstrap samples from each of the input data sets, `d1`, `d2`, etc., and passes the bootstrap samples to function `bootfun` for analysis. `bootfun` is a function handle specified using the `@` sign. `nboot` must be a positive integer, and each input data set must contain the same number of rows, `n`. Each bootstrap sample contains `n` rows chosen randomly (with replacement) from the corresponding input data set (`d1`, `d2`, etc.).

Each row of the output, `bootstat`, contains the results of applying `bootfun` to one bootstrap sample. If `bootfun` returns multiple output arguments, only the first is stored in `bootstat`. If the first output from `bootfun` is a matrix, the matrix is reshaped to a row vector for storage in `bootstat`.

`[bootstat,bootsam] = bootstrp(...)` returns an `n`-by-`n` boot 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`, `d2`, etc., 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`, `d2`, etc., 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.)

**Example** Correlate 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,'corrcoef',lsat,gpa);

bootstat(1:5,:)

ans =

    1.0000    0.3021    0.3021    1.0000
```

# bootstrp

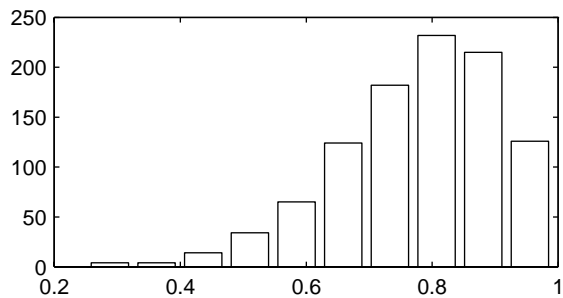
```
1.0000    0.6869    0.6869    1.0000
1.0000    0.8346    0.8346    1.0000
1.0000    0.8711    0.8711    1.0000
1.0000    0.8043    0.8043    1.0000
```

```
bootsam(:,1:5)
```

```
ans =
```

```
 4     7     5    12     8
 1    11    10     8     4
11     9    12     4     2
11    14    15     5    15
15    13     6     6     2
 6     8     4     3     8
 8     2    15     8     6
13    10    11    14     5
 1     7    12    14    14
 1    11    10     1     8
 8    14     2    14     7
11    12    10     8    15
 1     4    14     8     1
 6     1     5     5    12
 2    12     7    15    12
```

```
hist(bootstat(:,2))
```



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.

**Purpose** Box plots of a data sample

**Syntax**

```
boxplot(X)
boxplot(x,G)
boxplot(...,'Param1', val1, 'Param2', val2,...)
h = boxplot(...)
```

**Description** `boxplot(X)` produces a box and whisker plot for each column of the matrix `X`. The box has lines at the lower quartile, median, and upper quartile values. The whiskers are lines extending from each end of the box to show the extent of the rest of the data. Outliers are data with values beyond the ends of the whiskers. If there is no data outside the whisker, a dot is placed at the bottom whisker.

`boxplot(x,G)` produces a box and whisker plot for the vector `x` grouped by `G`. `G` is a grouping variable defined as a vector, string matrix, or cell array of strings. `G` can also be a cell array of several grouping variables (such as `{G1 G2 G3}`) to group the values in `x` by each unique combination of grouping variable values.

`boxplot(...,'Param1', val1, 'Param2', val2,...)` specifies optional parameter name/value pairs, as described in the following table.

Parameter Name	Parameter Values
'notch'	'on' to include notches (default is 'off')
'symbol'	Symbol to use for outliers (default is 'r+'). See <code>LineStyle</code> for a description of symbols.
'orientation'	Box orientation, 'vertical' (default) or 'horizontal'
'whisker'	Maximum whisker length in units of interquartile range (default 1.5)
'labels'	Character array or cell array of strings containing column labels (used only if <code>X</code> is a matrix, and the default label is the column number)

# boxplot

Parameter Name	Parameter Values
'colors'	A string, such as 'bgry', or a three-column matrix of box colors. Letters in the string specify colors, as described in LineSpec. Each box (outline, median line, and whiskers) is drawn in the corresponding color. The default is to draw all boxes with blue outline, red median, and black whiskers. Colors are reused if necessary.
'widths'	A numeric vector of box widths. The default is 0.5, or slightly smaller for fewer than three boxes. Widths are reused if necessary.
'positions'	A numeric vector of box positions. The default is 1:n.
'grouporder'	When the grouping variable G is given, a character array or cell array of group names, specifying the ordering of the groups in G. Ignored when G is not given.

In a notched box plot the notches represent a robust estimate of the uncertainty about the medians for box-to-box comparison. Boxes whose notches do not overlap indicate that the medians of the two groups differ at the 5% significance level.

Whiskers extend from the box out to the most extreme data value within  $whis \cdot iqr$ , where *whis* is the value of the 'whisker' parameter and *iqr* is the interquartile range of the sample.

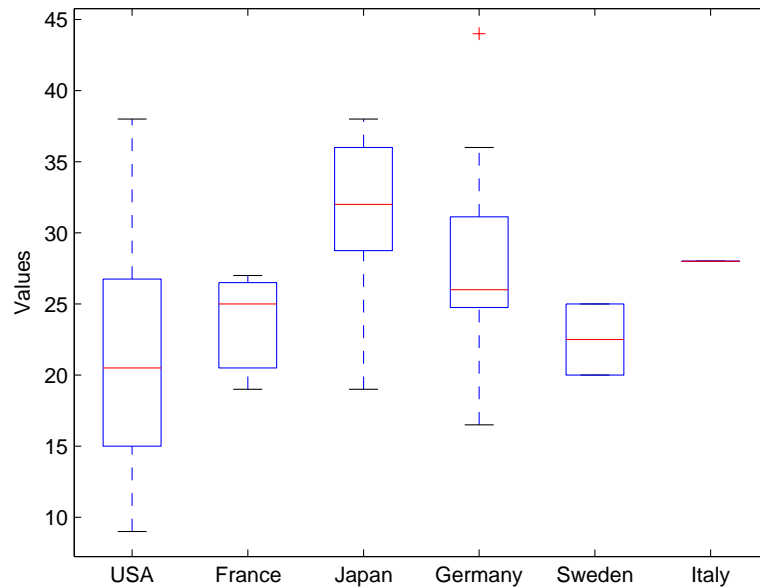
`h = boxplot(...)` returns a matrix of handles to the lines in the box plot. *h* contains one column for each box. Each column contains seven handles corresponding to the upper whisker, lower whisker, upper adjacent value, lower adjacent value, box, median, and outliers.

## Examples

The following commands create a box plot of car mileage grouped by country.

```
load carsmall
boxplot(MPG, Origin)
```

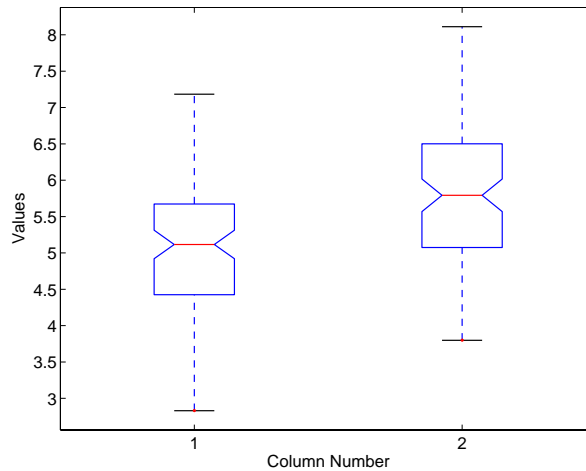




The following example produces notched box plots for two groups of sample data.

```
x1 = normrnd(5,1,100,1);  
x2 = normrnd(6,1,100,1);  
boxplot([x1,x2], 'notch', 'on')
```

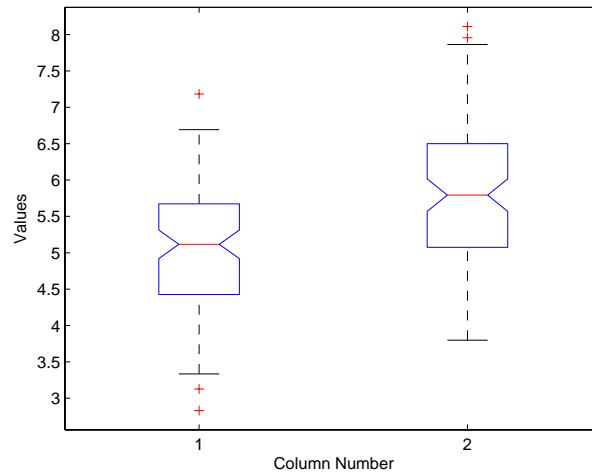
# boxplot



The difference between the medians of the two groups is approximately 1. Since the notches in the boxplot do not overlap, you can conclude, with 95% confidence, that the true medians do differ.

The following figure shows the boxplot for same data with the length of the whiskers specified as 1.0 times the interquartile range. Points beyond the whiskers are displayed using '+'.

```
boxplot([x1,x2], 'notch', 'on', 'whisker', 1)
```



## References

- [1] McGill, R., J.W. Tukey, and W.A. Larsen, "Variations of Boxplots," *The American Statistician*, Vol. 32, pp.12-16, 1978.
- [2] Velleman, P.F. and D.C. Hoaglin, *Applications, Basics, and Computing of Exploratory Data Analysis*, Duxbury Press, 1981.
- [3] Nelson, L.S., "Evaluating Overlapping Confidence Intervals," *Journal of Quality Technology*, Vol. 21, pp. 140-141, 1989.

## See Also

`anova1`, `kruskalwallis`, `multcompare`

# candexch

---

**Purpose** D-optimal design from candidate set using row exchanges

**Syntax**  
`rlist = candexch(C,nrows)`  
`rlist = candexch(C,nrows,'param1',value1,'param2',value2,...)`

**Description** `rlist = candexch(C,nrows)` uses a row-exchange algorithm to select a D-optimal design from the candidate set `C`. `C` is an  $n$ -by- $p$  matrix containing the values of  $p$  model terms at each of  $n$  points. `nrows` is the desired number of rows in the design. `rlist` is a vector of length `nrows` listing the selected rows.

The `candexch` function selects a starting design  $X$  at random, and uses a row-exchange algorithm to iteratively replace rows of  $X$  by rows of  $C$  in an attempt to improve the determinant of  $X' * X$ .

`rlist = candexch(C,nrows,'param1',value1,'param2',value2,...)` provides more control over the design generation through a set of parameter/value pairs. Valid parameters are the following:

Parameter	Value
'display'	Either 'on' or 'off' to control display of iteration number The default is 'on'.
'init'	Initial design as an <code>nrows</code> -by- <code>p</code> matrix. The default is a random subset of the rows of <code>C</code> .
'maxiter'	Maximum number of iterations. The default is 10.

---

**Note** The `rowexch` function also generates D-optimal designs using a row-exchange algorithm, but it accepts a model type and automatically selects a candidate set that is appropriate for such a model.

---

**Examples** Generate a D-optimal design when there is a restriction on the candidate set. In this case, the `rowexch` function isn't appropriate.

```
F = (fullfact([5 5 5])-1)/4;    % Factor settings in unit cube.  
T = sum(F,2)<=1.51;          % Find rows matching a restriction.
```

```
F = F(T,:); % Take only those rows.  
C = [ones(size(F,1),1) F F.^2]; % Compute model terms including  
% a constant and all squared terms.  
R = candexch(C,12); % Find a D-optimal 12-point subset.  
X = F(R,:); % Get factor settings.
```

## See Also

candgen, cordexch, rowexch, x2fx

# candgen

---

**Purpose** Generate candidate set for D-optimal design

**Syntax**  
`xcand = candgen(nfactors, 'model')`  
`[xcand, fxcand] = candgen(nfactors, 'model')`

**Description** `xcand = candgen(nfactors, 'model')` generates a candidate set appropriate for a D-optimal design with `nfactors` factors and the model `model`. The output matrix `xcand` has `nfactors` columns, with each row representing the coordinates of a candidate point. `model` is one of:

'linear'	Constant and linear terms (the default)
'interaction'	Constant, linear, and cross product terms
'quadratic'	Interactions plus squared terms
'purequadratic'	Constant, linear, and squared terms

Alternatively, `model` can be a matrix of term definitions as accepted by the `x2fx` function.

`[xcand, fxcand] = candgen(nfactors, model)` returns both the matrix of factor values `xcand` and the matrix of term values `fxcan`. You can input the latter to `candexch` to generate the D-optimal design.

---

**Note** The `rowexch` function automatically generates a candidate set using `candgen`, and creates a D-optimal design from that candidate set using `candexch`. Call these functions separately if you want to modify the default candidate set.

---

**See Also** `candexch`, `rowexch`, `x2fx`

**Purpose** Canonical correlation analysis

**Syntax**

```
[A,B] = canoncorr(X,Y)
[A,B,r] = canoncorr(X,Y)
[A,B,r,U,V] = canoncorr(X,Y)
[A,B,r,U,V,stats] = canoncorr(X,Y)
```

**Description** `[A,B] = canoncorr(X,Y)` computes the sample canonical coefficients for the n-by-d1 and n-by-d2 data matrices X and Y. X and Y must have the same number of observations (rows) but can have different numbers of variables (columns). A and B are d1-by-d and d2-by-d matrices, where  $d = \min(\text{rank}(X), \text{rank}(Y))$ . The jth columns of A and B contain the canonical coefficients, i.e., the linear combination of variables making up the jth 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 jth element of r is the correlation between the jth 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

$$U = (X - \text{repmat}(\text{mean}(X), N, 1)) * A$$

$$V = (Y - \text{repmat}(\text{mean}(Y), N, 1)) * B$$

`[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 dth correlations are all zero, for  $k = 0:(d-1)$ . stats contains seven fields, each a 1-by-d vector with elements corresponding to the values of k, as described in the following table:

Wilks	Wilks' lambda (likelihood ratio) statistic
chisq	Bartlett's approximate chi-squared statistic for $H_0^{(k)}$ with Lawley's modification
pChisq	Right-tail significance level for chisq

## canoncorr

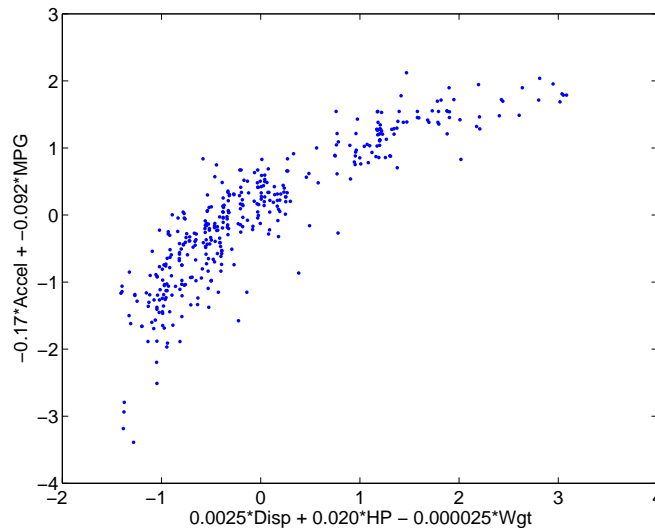
---

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')
```



### See Also

manova1, princomp

### References

[1] Krzanowski, W.J., *Principles of Multivariate Analysis*, Oxford University Press, Oxford, 1988.



[2] Seber, G.A.F., *Multivariate Observations*, Wiley, New York, 1984.

# capable

---

**Purpose** Process capability indices

**Syntax**  $p = \text{capable}(\text{data}, \text{specs})$   
 $[p, C_p, C_{pk}] = \text{capable}(\text{data}, \text{specs})$

**Description**  $p = \text{capable}(\text{data}, \text{specs})$  computes the probability that a sample, `data`, from some process falls outside the bounds specified in `specs`, a 2-element vector of the form `[lower upper]`.

The assumptions are that the measured values in the vector `data` are normally distributed with constant mean and variance and that the measurements are statistically independent.

$[p, C_p, C_{pk}] = \text{capable}(\text{data}, \text{specs})$  additionally returns the capability indices  $C_p$  and  $C_{pk}$ .

$C_p$  is the ratio of the range of the specifications to six times the estimate of the process standard deviation:

$$C_p = \frac{USL - LSL}{6\sigma}$$

For a process that has its average value on target, a  $C_p$  of 1 translates to a little more than one defect per thousand. Recently, many industries have set a quality goal of one part per million. This would correspond to  $C_p = 1.6$ . The higher the value of  $C_p$ , the more capable the process.

$C_{pk}$  is the ratio of difference between the process mean and the closer specification limit to three times the estimate of the process standard deviation:

$$C_{pk} = \min\left(\frac{USL - \mu}{3\sigma}, \frac{\mu - LSL}{3\sigma}\right)$$

where the process mean is  $\mu$ . For processes that do not maintain their average on target,  $C_{pk}$  is a more descriptive index of process capability.

**Example** Imagine a machined part with specifications requiring a dimension to be within three thousandths of an inch of nominal. Suppose that the machining process cuts too thick by one thousandth of an inch on average and also has a

standard deviation of one thousandth of an inch. What are the capability indices of this process?

```
data = normrnd(1,1,30,1);
[p,Cp,Cpk] = capable(data,[-3 3]);

indices = [p Cp Cpk]
indices =

    0.0172    1.1144    0.7053
```

You expect 17 parts out of a thousand to be out-of-specification. Cpk is less than Cp because the process is not centered.

**Reference**

[1] Montgomery, D., *Introduction to Statistical Quality Control*, 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 and variance of the observations in input vector `data`, and plots the pdf of the resulting T distribution. The observations in `data` are assumed to be normally distributed. The output, `p`, is the probability that a new observation from the estimated distribution will fall within the range specified by the two-element vector `specs`. The portion of the distribution between the lower and upper bounds specified in `specs` is shaded in the plot.

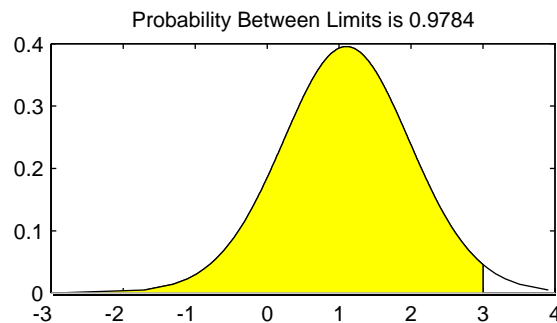
`[p,h] = capaplot(data, specs)` additionally returns handles to the plot elements in `h`.

**Example** Imagine a machined part with specifications requiring a dimension to be within 3 thousandths of an inch of nominal. Suppose that the machining process cuts too thick by one thousandth of an inch on average and also has a standard deviation of one thousandth of an inch.

```
data = normrnd(1,1,30,1);  
p = capaplot(data, [-3 3])  
p =
```

0.9784

The probability of a new observation being within specs is 97.84%.



**See Also**

capable, histfit

# caseread

---

**Purpose** Read casenames from a 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 pathname of any file elsewhere. `caseread` treats each line as a separate case.

`names = caseread` displays the **Select File to Open** dialog box for interactive selection of the input file.

**Example** Read the file `months.dat` created using the function `casewrite` on the next page.

```
type months.dat

January
February
March
April
May

names = caseread('months.dat')
names =

January
February
March
April
May
```

**See Also** `tblread`, `gname`, `casewrite`, `tdfread`

**Purpose** Write casenames from a string matrix to a 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 casename. `filename` is the name of a file in the current directory, or the complete pathname of any file elsewhere. `casewrite` writes each name to a separate line in `filename`.

`casewrite(strmat)` displays the **Select File to Write** dialog box for interactive specification of the output file.

**Example**

```
strmat = str2mat('January', 'February', 'March', 'April', 'May')
strmat =
January
February
March
April
May

casewrite(strmat, 'months.dat')
type months.dat

January
February
March
April
May
```

**See Also** `gname`, `caseread`, `tblwrite`, `tdfread`

# ccdesign

---

**Purpose** Generate central composite design

**Syntax**

```
D = ccdesign(nfactors)
D = ccdesign(nfactors, 'pname1', pvalue1, 'pname2', pvalue2, ...)
[D, blk] = ccdesign(...)
```

**Description** `D = ccdesign(nfactors)` generates a central composite design for `nfactors` factors. The output matrix `D` is `n`-by-`nfactors`, where `n` is the number of points in the design. Each row represents one run of the design, and it has the settings of all factors for that run. Factor values are normalized so that the cube points take values between -1 and 1.

`[D, blk] = ccdesign(nfactors)` requests a blocked design. The output vector `blk` is a vector of block numbers. Blocks are groups of runs that are to be measured under similar conditions (for example, on the same day). Blocked designs minimize the effect of between-block differences on the parameter estimates.

`[...] = ccdesign(nfactors, 'pname1', pvalue1, 'pname2', pvalue2, ...)` enables you to specify additional parameters and their values. Valid parameters are:

'center'	Number of center points:
	Integer      Specific number of center points to include
	'uniform'      Number of center points is selected to give uniform precision
	'orthogonal'      Number of center points is selected to give an orthogonal design (default)
'fraction'	Fraction of full factorial for cube portion expressed as an exponent of 1/2. For example:
	0      Whole design
	1      1/2 fraction
	2      1/4 fraction
'type'	Either 'inscribed', 'circumscribed', or 'faced'
'blocksize'	Maximum number of points allowed in a block.



**See Also**

bbdesign, cordexch, rowexch

# cdf

---

**Purpose** Computes a chosen cumulative distribution function (cdf)

**Syntax** `P = cdf('name',X,A1,A2,A3)`

**Description** `P = cdf('name',X,A1,A2,A3)` returns a matrix of probabilities, where `name` is a string containing the name of the distribution, `X` is a matrix of values, and `A`, `A2`, and `A3` are matrices of distribution parameters. Depending on the distribution, some of these parameters may not be necessary.

Vector or matrix inputs for `X`, `A1`, `A2`, and `A3` must have the same size, which is also the size of `P`. A scalar input for `X`, `A1`, `A2`, or `A3` is expanded to a constant matrix with the same dimensions as the other inputs.

`cdf` is a utility routine allowing you to access all the cdfs in the Statistics Toolbox by using the name of the distribution as a parameter. See “Overview of the Distributions” on page 2-45 for the list of available distributions.

## Examples

```
p = cdf('Normal',-2:2,0,1)
p =
    0.0228    0.1587    0.5000    0.8413    0.9772
```

```
p = cdf('Poisson',0:5,1:6)
p =
    0.3679    0.4060    0.4232    0.4335    0.4405    0.4457
```

## See Also

`betacdf`, `binocdf`, `chi2cdf`, `expcdf`, `fcdf`, `gamcdf`, `geocdf`, `hygecdf`, `icdf`, `logncdf`, `mle`, `nbinocdf`, `ncfcdf`, `nctcdf`, `ncx2cdf`, `normcdf`, `pdf`, `poisscdf`, `random`, `raylcdf`, `tcdf`, `unidcdf`, `unifcdf`, `wblcdf`

**Purpose** Plot of empirical cumulative distribution function

**Syntax**

```
cdfplot(X)
h = cdfplot(X)
[h,stats] = cdfplot(X)
```

**Description** `cdfplot(X)` displays a plot of the empirical cumulative distribution function (cdf) for the data in the vector `X`. The empirical cdf  $F(x)$  is defined as the proportion of `X` values less than or equal to  $x$ .

This plot, like those produced by `hist` and `normplot`, is useful for examining the distribution of a sample of data. You can overlay a theoretical cdf on the same plot to compare the empirical distribution of the sample to the theoretical distribution.

The `kstest`, `kstest2`, and `lillietest` functions compute test statistics that are derived from the empirical cdf. You may find the empirical cdf plot produced by `cdfplot` useful in helping you to understand the output from those functions.

`H = cdfplot(X)` returns a handle to the cdf curve.

`[h,stats] = cdfplot(X)` also returns a `stats` structure with the following fields.

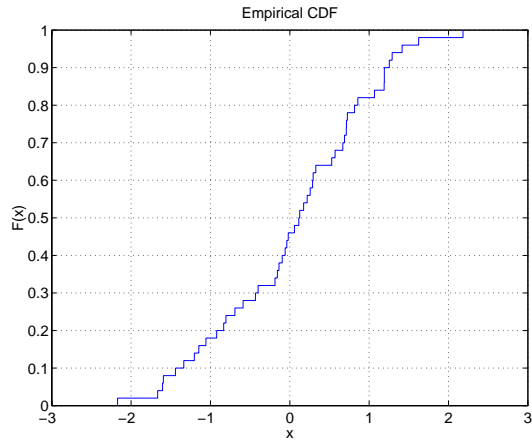
Field	Contents
<code>stats.min</code>	Minimum value
<code>stats.max</code>	Maximum value
<code>stats.mean</code>	Sample mean
<code>stats.median</code>	Sample median (50th percentile)
<code>stats.std</code>	Sample standard deviation

**Examples** Generate a normal sample and an empirical cdf plot of the data.

```
x = normrnd(0,1,50,1);
cdfplot(x)
```

# cdfplot

---



## See Also

`ecdf`, `hist`, `kstest`, `kstest2`, `lillietest`, `normplot`

**Purpose** Chi-square ( $\chi^2$ ) cumulative distribution function (cdf)

**Syntax** `P = chi2cdf(X,V)`

**Description** `P = chi2cdf(X,V)` computes the  $\chi^2$  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 1]`.

The  $\chi^2$  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 result,  $p$ , is the probability that a single observation from a  $\chi^2$  distribution with  $v$  degrees of freedom will fall in the interval `[0 x]`.

The  $\chi^2$  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`, `chi2inv`, `chi2pdf`, `chi2rnd`, `chi2stat`

# chi2inv

---

**Purpose** Inverse of the chi-square ( $\chi^2$ ) cumulative distribution function (cdf)

**Syntax** `X = chi2inv(P,V)`

**Description** `X = chi2inv(P,V)` computes the inverse of the  $\chi^2$  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  $\chi^2$  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  $\chi^2$  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  $\chi^2$  distribution with 10 degrees of freedom.

```
x = chi2inv(0.95,10)
```

```
x =
```

```
18.3070
```

You would observe values greater than 18.3 only 5% of the time by chance.

**See Also** `chi2cdf`, `chi2pdf`, `chi2rnd`, `chi2stat`, `icdf`

**Purpose** Chi-square ( $\chi^2$ ) probability density function (pdf)

**Syntax** `Y = chi2pdf(X,V)`

**Description** `Y = chi2pdf(X,V)` computes the  $\chi^2$  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 1]`.

The  $\chi^2$  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. The result,  $y$ , is the probability that a single observation from a  $\chi^2$  distribution with  $v$  degrees of freedom will have value  $x$ .

If  $x$  is standard normal, then  $x^2$  is distributed  $\chi^2$  with one degree of freedom. If  $x_1, x_2, \dots, x_n$  are  $n$  independent standard normal observations, then the sum of the squares of the  $x$ 's is distributed  $\chi^2$  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  $\chi^2$  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

`chi2cdf`, `chi2inv`, `chi2rnd`, `chi2stat`, `pdf`

# chi2rnd

---

**Purpose** Random numbers from the chi-square ( $\chi^2$ ) distribution

**Syntax**

```
R = chi2rnd(V)
R = chi2rnd(V,u)
R = chi2rnd(V,m,n)
```

**Description** `R = chi2rnd(V)` generates random numbers from the  $\chi^2$  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  $\chi^2$  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  $\chi^2$  distribution with degrees of freedom parameter `V`.

## Reproducing the Output of chi2rnd

`chi2rnd` uses the MATLAB functions `rand` and `randn` to generate random numbers. When you call `chi2rnd`, you change the current states of `rand` and `randn`, and thereby alter the output of subsequent calls to `chi2rnd` or any other functions that depend on `rand` or `randn`. If you want to reproduce the output of `chi2rnd`, reset the states of `rand` and `randn` to the same fixed values each time you call `chi2rnd`. For an example of how to do this, and a list of the Statistics Toolbox functions that depend on `rand` or `randn`, see “Reproducing the Output of Random Number Functions” on page 2-46.

---

**Note** The results in the following examples depend on the current states of `rand` and `randn`. If you run the code in these examples, your results may differ from the answers shown here.

---

**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

[chi2cdf](#), [chi2inv](#), [chi2pdf](#), [chi2stat](#)

# chi2stat

---

**Purpose** Mean and variance for the chi-square ( $\chi^2$ ) distribution

**Syntax** [M,V] = chi2stat(NU)

**Description** [M,V] = chi2stat(NU) returns the mean and variance for the  $\chi^2$  distribution with degrees of freedom parameters specified by NU.

The mean of the  $\chi^2$  distribution is  $\nu$ , the degrees of freedom parameter, and the variance is  $2\nu$ .

## Example

```
nu = 1:10;  
nu = nu'*nu;  
[m,v] = chi2stat(nu)
```

m =

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

v =

2	4	6	8	10	12	14	16	18	20
4	8	12	16	20	24	28	32	36	40
6	12	18	24	30	36	42	48	54	60
8	16	24	32	40	48	56	64	72	80
10	20	30	40	50	60	70	80	90	100
12	24	36	48	60	72	84	96	108	120
14	28	42	56	70	84	98	112	126	140
16	32	48	64	80	96	112	128	144	160
18	36	54	72	90	108	126	144	162	180
20	40	60	80	100	120	140	160	180	200

**See Also** chi2cdf, chi2inv, chi2pdf, chi2rnd

**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(...)
```

**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, and each element defines which group the corresponding row of `training` belongs to. `group` can be a numeric vector, a string array, or a cell array of strings. `training` and `group` must have the same number of rows. `classify` treats NaNs or empty strings in `group` as missing values, and ignores the corresponding rows of `training`. `class` indicates which group each row of `sample` has been assigned to, and is of the same type as `group`.

`class = classify(sample,training,group,type)` enables you to specify the type of discriminant function type as one of:

- |                 |   |
|-----------------|---|
| 'linear'        | Fits a multivariate normal density to each group, with a pooled estimate of covariance (default).                                 |
| 'diaglinear'    | Same as 'linear', except that the covariance matrices are assumed to be diagonal and are estimated as <code>diag(var)</code> .    |
| 'quadratic'     | Fits multivariate normal densities with covariance estimates stratified by group.   |
| 'diagquadratic' | Same as 'quadratic', except that the covariance matrices are assumed to be diagonal and are estimated as <code>diag(var)</code> . |
| 'mahalanobis'   | Uses Mahalanobis distances with stratified covariance estimates.  |

# classify

---

`class = classify(sample, training, group, type, prior)` enables you to specify prior probabilities for the groups in one of three ways. `prior` can be

- A numeric vector of the same length as the number of unique values in `group`. If `group` is numeric, the order of `prior` must correspond to the sorted 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`, and containing unique values indicating which groups the elements of `prob` correspond to.

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.

- The string value 'empirical', indicating that `classify` should estimate the group prior probabilities 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 of the misclassification error rate. `classify` returns the apparent error rate, i.e., the percentage of observations in the training that are misclassified.

`[class, err, posterior] = classify(...)` returns `posterior`, a matrix containing estimates of the posterior probabilities that the  $j$ 'th training group was the source of the  $i$ 'th sample observation, that is,  $\Pr\{\text{group } j \mid \text{obs } i\}$ . `posterior` is not computed for Mahalanobis discrimination.

`[class, err, posterior, logp] = classify(...)` returns `logp`, a vector containing estimates of the logarithms of the unconditional predictive probability density of the sample observations,  $p(\text{obs } i)$ .  $p(\text{obs } i)$  is the sum of  $p(\text{obs } i \mid \text{group } j) * \Pr\{\text{group } j\}$  taken over all groups. `logp` is not computed for Mahalanobis discrimination.

## Examples

```
load discrim
sample = ratings(idx,:);
```

```
training = ratings(1:200,:);  
g = group(1:200);  
class = classify(sample,training,g);  
first5 = class(1:5)
```

```
first5 =
```

```
2  
2  
2  
2  
2
```

## See Also

mahal

## References

- [1] Krzanowski, W.J., *Principles of Multivariate Analysis*, Oxford University Press, Oxford, 1988.
- [2] Seber, G.A.F., *Multivariate Observations*, Wiley, New York, 1984

# cluster

---

**Purpose** Construct clusters from linkage output

**Syntax**

```
T = cluster(Z,'cutoff',c)
T = cluster(Z,'maxclust',n)
T = cluster(...,'criterion','crit')
T = cluster(...,'depth',d)
```

**Description** `T = cluster(Z,'cutoff',c)` constructs clusters from the hierarchical cluster tree, `Z`, generated by the linkage function. `Z` is a matrix of size  $(m-1)$ -by-3, where  $m$  is the number of observations in the original data. `c` is a threshold for cutting `Z` into clusters. Clusters are formed when inconsistent values are less than `c`. See the `inconsistent` function for more information. The output `T` is a vector of size  $m$  that contains the cluster number for each observation in the original data.

`T = cluster(Z,'maxclust',n)` specifies `n` as the maximum number of clusters to form from the hierarchical tree in `Z`.

`T = cluster(...,'criterion','crit')` uses the specified criterion for forming clusters, where `crit` is either `'inconsistent'` or `'distance'`.

`T = cluster(...,'depth',d)` evaluates inconsistent values to a depth of `d` in the tree. The default is `d = 2`. An inconsistency coefficient computation compares a link between two objects in the cluster tree with neighboring links up to the specified depth. See the `inconsistent` function for more information.

**Example** The example uses the `pdist` function to calculate the distance between items in a matrix of random numbers and then uses the `linkage` function to compute the hierarchical cluster tree based on the matrix. The example passes the output of the linkage function to the `cluster` function. The `'maxclust'` value 3 indicates that you want to group the items into three clusters. The `find` function lists all the items grouped into cluster 1.

```
rand('state', 7)
X = [rand(10,3); rand(10,3)+1; rand(10,3)+2];
Y = pdist(X);
Z = linkage(Y);
T = cluster(Z,'maxclust',3);
find(T==1)
```

```
ans =  
  11  
  12  
  13  
  14  
  15  
  16  
  17  
  18  
  19  
  20
```

**See Also**

`clusterdata`, `cophenet`, `inconsistent`, `linkage`, `pdist`

# clusterdata

---

**Purpose** Construct clusters from data

**Syntax** `T = clusterdata(X, cutoff)`  
`T = clusterdata(X, 'param1', val1, 'param2', val2, ...)`

**Description** `T = clusterdata(X, cutoff)` uses the `pdist`, `linkage`, and `cluster` functions to construct clusters from data `X`. `X` is an  $m$ -by- $n$  matrix, treated as  $m$  observations of  $n$  variables. `cutoff` is a threshold for cutting the hierarchical tree generated by `linkage` into clusters. When  $0 < \text{cutoff} < 2$ , `clusterdata` forms clusters when inconsistent values are greater than `cutoff` (see the `inconsistent` function). When `cutoff` is an integer and `cutoff`  $\geq 2$ , then `clusterdata` interprets `cutoff` as the maximum number of clusters to keep in the hierarchical tree generated by `linkage`. The output `T` is a vector of size  $m$  containing a cluster number for each observation.

`T = clusterdata(X, cutoff)` is the same as

```
Y = pdist(X, 'euclid');
Z = linkage(Y, 'single');
T = cluster(Z, 'cutoff', cutoff);
```

`T = clusterdata(X, 'param1', val1, 'param2', val2, ...)` provides more control over the clustering through a set of parameter/value pairs. Valid parameters are:

'distance'	Any of the distance metric names allowed by <code>pdist</code> (follow the 'minkowski' option by the value of the exponent $p$ ).
'linkage'	Any of the linkage methods allowed by the <code>linkage</code> function
'cutoff'	Cutoff for inconsistent or distance measure
'maxclust'	Maximum number of clusters to form
'criterion'	Either 'inconsistent' or 'distance'
'depth'	Depth for computing inconsistent values

**Example** The example first creates a sample data set of random numbers. It then uses `clusterdata` to compute the distances between items in the data set and create a hierarchical cluster tree from the data set. Finally, the `clusterdata` function



groups the items in the data set into three clusters. The example uses the `find` function to list all the items in cluster 2.

```
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  
   115  
    16  
    17  
    18  
    19  
    20
```

## See Also

`cluster`, `inconsistent`, `kmeans`, `linkage`, `pdist`

# cmdscale

---

**Purpose** Classical multidimensional scaling

**Syntax**  $Y = \text{cmdscale}(D)$   
 $[Y, e] = \text{cmdscale}(D)$

**Description**  $Y = \text{cmdscale}(D)$  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 interpoint distances are given by  $D$  can be embedded.

$[Y, e] = \text{cmdscale}(D)$  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 interpoint 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  $\text{cmdscale}$  chooses  $p$  as the number of positive eigenvalues. In this case, the reduction to  $p$  or fewer dimensions provides a reasonable approximation to  $D$  only if the negative elements of  $e$  are small in magnitude.

You can specify  $D$  as either a full dissimilarity matrix, or in upper triangle vector form such as is output by `pdist`. A full dissimilarity matrix must be real and symmetric, and have zeros along the diagonal and positive elements everywhere else. A dissimilarity matrix in upper triangle form must have real, positive entries. You can also specify  $D$  as a full similarity matrix, with ones along the diagonal and all other elements less than one. `cmdscale` transforms a similarity matrix to a dissimilarity matrix in such a way that distances between the points returned in  $Y$  equal or approximate  $\sqrt{1 - D}$ . To use a different transformation, you must transform the similarities prior to calling `cmdscale`.

**Examples** Generate some points in 4-dimensional space, but close to 3-dimensional space, then reduce them to distances only.

```
X = [normrnd(0,1,10,3) normrnd(0,.1,10,1)];  
D = pdist(X,'euclidean');
```

Find a configuration with those inter-point distances.

```
[Y,e] = cmdscale(D);  
% Four, but fourth one small  
dim = sum(e > eps^(3/4))  
% Poor reconstruction  
maxerr2 = max(abs(pdist(X) - pdist(Y(:,1:2))))  
% Good reconstruction  
maxerr3 = max(abs(pdist(X) - pdist(Y(:,1:3))))  
% Exact reconstruction  
maxerr4 = max(abs(pdist(X) - pdist(Y)))  
% D is now non-Euclidean  
D = pdist(X,'cityblock');  
[Y,e] = cmdscale(D);  
% One is large negative  
min(e)  
% Poor reconstruction  
maxerr = max(abs(pdist(X) - pdist(Y)))
```

**See Also** `mdscale`, `pdist`, `procrustes`

**References** [1] Seber, G.A.F., *Multivariate Observations*, Wiley, 1984

# combnk

---

**Purpose** Enumeration of all combinations of  $n$  objects  $k$  at a time

**Syntax** `C = combnk(v,k)`

**Description** `C = combnk(v,k)` returns all combinations of the  $n$  elements in  $v$  taken  $k$  at a time.

`C = combnk(v,k)` produces a matrix  $C$  with  $k$  columns and  $n! / k!(n-k)!$  rows, where each row contains  $k$  of the elements in the vector  $v$ .

It is not practical to use this function if  $v$  has more than about 15 elements.

**Example** Combinations of characters from a string.

```
C = combnk('tendrill',4);
last5 = C(31:35,:);

last5 =

tedr
tenl
teni
tenr
tend
```

Combinations of elements from a numeric vector.

```
c = combnk(1:4,2)

c =

     3     4
     2     4
     2     3
     1     4
     1     3
     1     2
```

**Purpose** Cophenetic correlation coefficient

**Syntax** `c = cophenet(Z,Y)`

**Description** `c = cophenet(Z,Y)` computes the cophenetic correlation coefficient which compares the distance information in `Z`, generated by linkage, and the distance information in `Y`, generated by `pdist`. `Z` is a matrix of size  $(m-1)$ -by-3, with distance information in the third column. `Y` is a vector of size  $m \cdot (m - 1)/2$ .

For example, given a group of objects  $\{1, 2, \dots, m\}$  with distances `Y`, the function `linkage` produces a hierarchical cluster tree. The `cophenet` function measures the distortion of this classification, indicating how readily the data fits into the structure suggested by the classification.

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

$$c = \frac{\sum_{i < j} (Y_{ij} - y)(Z_{ij} - z)}{\sqrt{\sum_{i < j} (Y_{ij} - y)^2 \sum_{i < j} (Z_{ij} - z)^2}}$$

where:

- $Y_{ij}$  is the distance between objects  $i$  and  $j$  in `Y`.
- $Z_{ij}$  is the distance between objects  $i$  and  $j$  in `Z(:,3)`.
- $y$  and  $z$  are the average of `Y` and `Z(:,3)`, respectively.

### Example

```
rand('state',12);
X = [rand(10,3);rand(10,3)+1;rand(10,3)+2];
Y = pdist(X);
Z = linkage(Y,'average');
c = cophenet(Z,Y)
```

```
c =
    0.7951
```

# cophenet

---

## See Also

cluster, dendrogram, inconsistent, linkage, pdist, squareform

**Purpose** D-optimal design of experiments – coordinate exchange algorithm

**Syntax**

```
settings = cordexch(nfactors,nruns)
[settings,X] = cordexch(nfactors,nruns)
[settings,X] = cordexch(nfactors,nruns,'model')
[settings,X] = cordexch(...,'param1',value1,'param2',value2,...)
```

**Description** `settings = cordexch(nfactors,nruns)` generates the factor settings matrix, `settings`, for a D-optimal design using a linear additive model with a constant term. `settings` has `nruns` rows and `nfactors` columns.

`[settings,X] = cordexch(nfactors,nruns)` also generates the associated design matrix `X`.

`[settings,X] = cordexch(nfactors,nruns,'model')` produces a design for fitting a specified regression model. The input, `'model'`, can be one of these strings:

<code>'linear'</code>	Includes constant and linear terms (the default)
<code>'interaction'</code>	Includes constant, linear, and cross-product terms.
<code>'quadratic'</code>	Includes interactions and squared terms.
<code>'purequadratic'</code>	Includes constant, linear and squared terms.

Alternatively `model` can be a matrix of term definitions as accepted by the `x2fx` function.

`[settings,X] = cordexch(...,'param1',value1,'param2',value2,...)` provides more control over the design generation through a set of parameter/value pairs. Valid parameters are:

<code>'display'</code>	Either <code>'on'</code> or <code>'off'</code> to control display of iteration counter. The default is <code>'on'</code> .
<code>'init'</code>	Initial design as an <code>nruns</code> -by- <code>nfactors</code> matrix. The default is a randomly selected set of points.
<code>'maxiter'</code>	Maximum number of iterations. The default is 10.

# cordexch

---

## Example

The D-optimal design for two factors in nine runs using a quadratic model is the  $3^2$  factorial as shown below:

```
settings = cordexch(2,9,'quadratic')
settings =
    -1     1
     1     1
     0     1
     1    -1
    -1    -1
     0    -1
     1     0
     0     0
    -1     0
```

## Algorithm

The `cordexch` function searches for a D-optimal design using a coordinate exchange algorithm. It creates a starting design, and then iterates by changing each coordinate of each design point in an attempt to reduce the variance of the coefficients that would be estimated using this design.

## See Also

`bbdesign`, `candexch`, `candgen`, `ccdesign`, `daugment`, `dcovary`, `rowexch`, `x2fx`



**Purpose** Linear or rank correlation

**Syntax**

```
RHO = corr(X)
RHO = corr(X,Y,...)
[RHO, PVAL] = corr(...)
[...] = corr(..., 'param1', val1, 'param2', val2,...)
```

**Description**

`RHO = corr(X)` returns a  $p$ -by- $p$  matrix containing the pairwise linear correlation coefficient between each pair of columns in the  $n$ -by- $p$  matrix  $X$ .

`RHO = corr(X,Y,...)` returns a  $p1$ -by- $p2$  matrix containing the pairwise correlation coefficient between each pair of columns in the  $n$ -by- $p1$  and  $n$ -by- $p2$  matrices  $X$  and  $Y$ .

`[RHO, PVAL] = corr(...)` also returns `PVAL`, a matrix of  $p$ -values for testing the hypothesis of no correlation against the alternative that there is a non-zero 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'	<ul style="list-style-type: none"> <li>'Pearson' (the default) computes Pearson's linear correlation coefficient</li> <li>'Kendall' computes Kendall's tau</li> <li>'Spearman' computes Spearman's rho</li> </ul>

<b>Parameter</b>	<b>Values</b>
'rows'	<ul style="list-style-type: none"><li>• 'all' (the default) uses all rows regardless of missing values (NaNs)</li><li>• 'complete' uses only rows with no missing values</li><li>• 'pairwise' computes <math>RHO(i, j)</math> using rows with no missing values in column <math>i</math> or <math>j</math></li></ul>
'tail' — The alternative hypothesis against which to compute p-values for testing the hypothesis of no correlation	<ul style="list-style-type: none"><li>• 'ne' — Correlation is not zero (the default)</li><li>• 'gt' — Correlation is greater than zero</li><li>• 'lt' — Correlation is less than zero</li></ul>

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

**Purpose** Correlation coefficients

**Syntax**

```
R = corrcoef(X)
R = corrcoef(x,y)
[R,P]=corrcoef(...)
[R,P,RLO,RUP]=corrcoef(...)
[...]=corrcoef(...,'param1',val1,'param2',val2,...)
```

**Description** `R = corrcoef(X)` returns a matrix `R` of correlation coefficients calculated from an input matrix `X` whose rows are observations and whose columns are variables. The  $(i, j)$ th element of the matrix `R` is related to the covariance matrix `C = cov(X)` by

$$R(i,j) = \frac{C(i,j)}{\sqrt{C(i,i)C(j,j)}}$$

`corrcoef(X)` is the zeroth lag of the covariance function, that is, the zeroth lag of `xcov(x, 'coeff')` packed into a square array.

`R = corrcoef(x,y)` where `x` and `y` are column vectors is the same as `corrcoef([x y])`.

`[R,P]=corrcoef(...)` also returns `P`, a matrix of p-values for testing the hypothesis of no correlation. Each p-value is the probability of getting a correlation as large as the observed value by random chance, when the true correlation is zero. If  $P(i, j)$  is small, say less than 0.05, then the correlation  $R(i, j)$  is significant.

`[R,P,RLO,RUP]=corrcoef(...)` also returns matrices `RLO` and `RUP`, of the same size as `R`, containing lower and upper bounds for a 95% confidence interval for each coefficient.

`[...]=corrcoef(...,'param1',val1,'param2',val2,...)` specifies additional parameters and their values. Valid parameters are the following.

## corrcoef

---

'alpha' A number between 0 and 1 to specify a confidence level of  $100*(1 - \alpha)\%$ . Default is 0.05 for 95% confidence intervals.

'rows' Either 'all' (default) to use all rows, 'complete' to use rows with no NaN values, or 'pairwise' to compute  $R(i, j)$  using rows with no NaN values in either column  $i$  or  $j$ .

The p-value is computed by transforming the correlation to create a t statistic having  $n-2$  degrees of freedom, where  $n$  is the number of rows of  $X$ . The confidence bounds are based on an asymptotic normal distribution of  $0.5*\log((1+R)/(1-R))$ , with an approximate variance equal to  $1/(n-3)$ . These bounds are accurate for large samples when  $X$  has a multivariate normal distribution. The 'pairwise' option can produce an  $R$  matrix that is not positive definite.

The `corrcoef` function is part of the standard MATLAB language.

### Examples

Generate random data having correlation between column 4 and the other columns.

```
x = randn(30,4); % Uncorrelated data
x(:,4) = sum(x,2); % Introduce correlation.
[r,p] = corrcoef(x) % Compute sample correlation and p-values.
[i,j] = find(p<0.05); % Find significant correlations.
[i,j] % Display their (row,col) indices.
```

```
r =
    1.0000    -0.3566     0.1929     0.3457
   -0.3566     1.0000    -0.1429     0.4461
    0.1929    -0.1429     1.0000     0.5183
    0.3457     0.4461     0.5183     1.0000
```

```
p =
    1.0000     0.0531     0.3072     0.0613
    0.0531     1.0000     0.4511     0.0135
    0.3072     0.4511     1.0000     0.0033
    0.0613     0.0135     0.0033     1.0000
```

```
ans =
     4     2
```

4	3
2	4
3	4

**See Also**

cov, mean, std, var

xcorr, xcov in the Signal Processing Toolbox

## COV

---

**Purpose** Covariance matrix.

**Syntax**  $C = \text{cov}(X)$   
 $C = \text{cov}(x,y)$

**Description**  $C = \text{cov}(X)$  computes the covariance matrix. For a single vector,  $\text{cov}(x)$  returns a scalar containing the variance. For matrices, where each row is an observation, and each column a variable,  $\text{cov}(X)$  is the covariance matrix.

The variance function,  $\text{var}(X)$  is the same as  $\text{diag}(\text{cov}(X))$ .

The standard deviation function,  $\text{std}(X)$  is equivalent to  $\text{sqrt}(\text{diag}(\text{cov}(X)))$ .

$\text{cov}(x,y)$ , where  $x$  and  $y$  are column vectors of equal length, gives the same result as  $\text{cov}([x \ y])$ .

The `cov` function is part of the standard MATLAB language.

**Algorithm** The algorithm for `cov` is

```
[n,p] = size(X);  
X = X - ones(n,1) * mean(X);  
Y = X'*X/(n-1);
```

**See Also** `corrcoef`, `mean`, `std`, `var`  
`xcov`, `xcorr` (Signal Processing Toolbox)

**Purpose** Cross-tabulation of several vectors

**Syntax**

```
table = crosstab(col1,col2)
table = crosstab(col1,col2,col3,...)
[table,chi2,p] = crosstab(col1,col2)
[table,chi2,p,label] = crosstab(col1,col2)
```

**Description** `table = crosstab(col1,col2)` takes two vectors of positive integers and returns a matrix, `table`, of cross-tabulations. The  $ij$ th element of `table` contains the count of all instances where  $col1 = i$  and  $col2 = j$ .

Alternatively, `col1` and `col2` can be vectors containing noninteger values, character arrays, or cell arrays of strings. `crosstab` implicitly assigns a positive integer group number to each distinct value in `col1` and `col2`, and creates a cross-tabulation using those numbers.

`table = crosstab(col1,col2,col3,...)` returns `table` as an  $n$ -dimensional array, where  $n$  is the number of arguments you supply. The value of `table(i,j,k,...)` is the count of all instances where  $col1 = i$ ,  $col2 = j$ ,  $col3 = k$ , and so on.

`[table,chi2,p] = crosstab(col1,col2)` also returns the chi-square statistic, `chi2`, for testing the independence of the rows and columns of `table`. The scalar `p` is the significance level of the test. Values of `p` near zero cast doubt on the assumption of independence of the rows and columns of `table`.

`[table,chi2,p,label] = crosstab(col1,col2)` also returns a cell array `label` that has one column for each input argument. The value in `label(i,j)` is the value of `colj` that defines group  $i$  in the  $j$ th dimension.

## Example

### Example 1

This example generates 2 columns of 50 discrete uniform random numbers. The first column has numbers from 1 to 3. The second has only the numbers 1 and 2. The two columns are independent so it would be surprising if `p` were near zero.

```
r1 = unidrnd(3,50,1);
r2 = unidrnd(2,50,1);
[table,chi2,p] = crosstab(r1,r2)
```

# crosstab

---

```
table =  
      10    5  
      8     8  
      6    13  
  
chi2 =  
      4.1723  
  
p =  
      0.1242
```

The result, 0.1242, is not a surprise. A very small value of  $p$  would make you suspect the “randomness” of the random number generator.

## Example 2

you have data collected on several cars over a period of time. How many four-cylinder cars were made in the USA during the late part of this period?

```
[t,c,p,l] = crosstab(cyl4,when,org);  
  
l  
l =  
      'Other'    'Early'    'USA'  
      'Four'    'Mid'     'Europe'  
      []       'Late'    'Japan'  
  
t(2,3,1)  
  
ans =  
      38
```

## See Also

tabulate



**Purpose** D-optimal augmentation of an experimental design

**Syntax**

```
settings = daugment(startdes, nruns)
[settings, X] = daugment(startdes, nruns)
[settings, X] = daugment(startdes, nruns, 'model')
[settings, X] = daugment(..., 'param1', value1, 'param2', value2, ...)
```

**Description** `settings = daugment(startdes, nruns)` adds `nruns` runs to an experimental design using the coordinate exchange D-optimal algorithm. `startdes` is a matrix of factor settings in the original design. The output matrix `settings` is the matrix of factor settings for the design.

`[settings, X] = daugment(startdes, nruns)` also generates the associated design matrix, `X`.

`[settings, X] = daugment(startdes, nruns, 'model')` also controls the order of the regression model. The input, `'model'`, can be one of these:

<code>'linear'</code>	Includes constant and linear terms (the default)
<code>'interaction'</code>	Includes constant, linear, and cross-product terms.
<code>'quadratic'</code>	Includes interactions and squared terms.
<code>'purequadratic'</code>	Includes constant, linear and squared terms.

Alternatively `model` can be a matrix of term definitions as accepted by the `x2fx` function.

`[settings, X] = daugment(..., 'param1', value1, 'param2', value2, ...)` provides more control over the design generation through a set of parameter/value pairs. Valid parameters are the following:

Parameter	Value
<code>'display'</code>	Either <code>'on'</code> or <code>'off'</code> to control display of iteration counter. The default is <code>'on'</code> .

# daugment

---

Parameter	Value
'init'	Initial design as an nruns-by-nfactors matrix. The default is a randomly selected set of points.
'maxiter'	Maximum number of iterations. The default is 10.

## Example

This example adds 5 runs to a  $2^2$  factorial design to fit a quadratic model.

```
startdes = [-1 -1; 1 -1; -1 1; 1 1];
settings = daugment(startdes,5,'quadratic')

settings =

    -1    -1
     1    -1
    -1     1
     1     1
     1     0
    -1     0
     0     1
     0     0
     0    -1
```

The result is a  $3^2$  factorial design.

## See Also

cordexch, x2fx

**Purpose** D-optimal design with specified fixed covariates

**Syntax**

```
settings = dcovary(nfactors, covariates)
[settings, X] = dcovary(nfactors, covariates)
[settings, X] = dcovary(nfactors, covariates, 'model')
[settings, X] = dcovary(..., 'param1', value1, 'param2', value2, ...)
```

**Description** `settings = dcovary(nfactors, covariates)` uses a coordinate exchange algorithm to generate a D-optimal design for `nfactors` factors, subject to the constraint that it also include the fixed covariate values in the input matrix `covariates`. The number of runs in the design is taken to be the number of rows in the `covariates` matrix. The output matrix `settings` is the matrix of factor settings for the design, including the fixed covariates.

`[settings, X] = dcovary(nfactors, covariates)` also generates the associated design matrix, `X`.

`[settings, X] = dcovary(nfactors, covariates, 'model')` also controls the order of the regression model. The input, `'model'`, can be one of these:

<code>'linear'</code>	Includes constant and linear terms (the default)
<code>'interaction'</code>	Includes constant, linear, and cross-product terms.
<code>'quadratic'</code>	Includes interactions and squared terms.
<code>'purequadratic'</code>	Includes constant, linear and squared terms.

Alternatively `'model'` can be a matrix of term definitions as accepted by the `x2fx` function. The model is applied to the fixed covariates as well as the regular factors. If you want to treat the fixed covariates specially, for example by including linear terms for them but quadratic terms for the regular factors, you can do this by creating the proper `'model'` matrix.

`[settings, X] = dcovary(..., 'param1', value1, 'param2', value2, ...)` provides more control over the design generation through a set of parameter/value pairs. Valid parameters are:

'display' Either 'on' or 'off' to control display of iteration counter. The default is 'on'.

'init' Initial design as an nruns-by-nfactors matrix. The default is a randomly selected set of points.

'maxiter' Maximum number of iterations. The default is 10.

## Example

**Example 1.** Generate a design for three factors in 2 blocks of 4 runs.

```
blk = [-1 -1 -1 -1 1 1 1 1]';  
dsgn = dcovary(3,blk)
```

```
dsgn =  
-1 1 1 -1  
1 -1 -1 -1  
-1 1 -1 -1  
1 -1 1 -1  
1 1 -1 1  
1 1 1 1  
-1 -1 1 1  
-1 -1 -1 1
```

**Example 2.** Suppose you want to block an eight run experiment into 4 blocks of size 2 to fit a linear model on two factors.

```
covariates = dummyvar([1 1 2 2 3 3 4 4]);  
settings = dcovary(2,covariates(:,1:3),'linear')  
settings =  
1 1 1 0 0  
-1 -1 1 0 0  
-1 1 0 1 0  
1 -1 0 1 0  
1 1 0 0 1  
-1 -1 0 0 1  
-1 1 0 0 0  
1 -1 0 0 0
```

The first two columns of the output matrix contain the settings for the two factors. The last three columns are *dummy variable* codings for the four blocks.

**Algorithm**

The `dcovary` function creates a starting design that includes the fixed covariate values, and then iterates by changing the non-fixed coordinates of each design point in an attempt to reduce the variance of the coefficients that would be estimated using this design.

**See Also**

`cordexch`, `daugment`, `rowexch`, `x2fx`

# dendrogram

---

**Purpose** Plot dendrogram graphs

**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  $m$  is 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.

`H = dendrogram(Z,p)` generates a dendrogram with only the top  $p$  nodes. By default, `dendrogram` uses 30 as the value of  $p$ . When there are more than 30 initial nodes, a dendrogram may look crowded. To display every node, set  $p = 0$ .

`[H,T] = dendrogram(...)` generates a dendrogram and returns `T`, a vector of length  $m$  that contains the leaf node number for each object in the original data set. `T` is useful when  $p$  is less than the total number of objects, so some leaf nodes in the display correspond to multiple objects. For example, to find out which objects are contained in leaf node  $k$  of the dendrogram, use `find(T==k)`. When there are fewer than  $p$  objects in the original data, all objects are displayed in the dendrogram. In this case, `T` is the identity map, i.e., `T = (1:m)'`, where each node contains only a single object.

`[H,T,perm] = dendrogram(...)` generates a dendrogram and returns the permutation vector of the node labels of the leaves of the dendrogram. `perm` is ordered from left to right on a horizontal dendrogram and bottom to top for a vertical dendrogram.

`[...] = dendrogram(..., 'colorthreshold', t)` assigns a unique color to each group of nodes in the dendrogram where the linkage is less than the threshold `t`. `t` is a value in the interval `[0, max(Z(:,3))]`. Setting `t` to the string `'default'` is the same as `t = .7(max(Z(:,3)))`. `0` is the same as not specifying `'colorthreshold'`. The value `max(Z(:,3))` treats the entire tree as one group and colors it all one color.

`[...] = dendrogram(..., 'orientation', 'orient')` orients the dendrogram within the figure window. The options for `'orient'` are

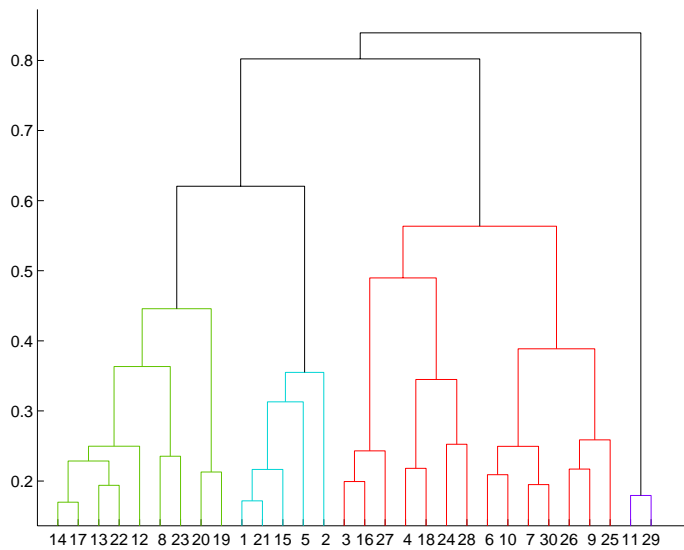
<code>'top'</code>	Top to bottom (default)
<code>'bottom'</code>	Bottom to top
<code>'left'</code>	Left to right
<code>'right'</code>	Right to left

`[...] = dendrogram(..., 'labels', S)` accepts a character array or cell array of strings `S` with one label for each observation. Any leaves in the tree containing a single observation are labeled with that observation's label.

## Example

```
X= rand(100,2);
Y= pdist(X, 'cityblock');
Z= linkage(Y, 'average');
[H,T] = dendrogram(Z, 'colorthreshold', 'default');
```

# 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`



<b>Purpose</b>	Interactive plot of cdf (or pdf) for many probability distributions
<b>Syntax</b>	<code>disttool</code>
<b>Description</b>	The <code>disttool</code> command displays a graphical interface for exploring the effects of changing parameters on the plot of a cdf or pdf. See “Probability Distributions Demo” on page 2-3 for detailed information about the demo.
<b>See Also</b>	<code>randtool</code>

# dummyvar

---

**Purpose** Matrix of 0-1 “dummy” variables

**Syntax** `D = dummyvar(group)`

**Description** `D = dummyvar(group)` generates a matrix, `D`, of 0-1 columns. `D` has one column for each unique value in each column of the matrix `group`. Each column of `group` contains positive integers that indicate the group membership of an individual row.

**Example** Suppose you are studying the effects of two machines and three operators on a process. The first column of `group` would have the values 1 or 2 depending on which machine was used. The second column of `group` would have the values 1, 2, or 3 depending on which operator ran the machine.

```
group = [1 1;1 2;1 3;2 1;2 2;2 3];  
D = dummyvar(group)
```

```
D =  
    1    0    1    0    0  
    1    0    0    1    0  
    1    0    0    0    1  
    0    1    1    0    0  
    0    1    0    1    0  
    0    1    0    0    1
```

**See Also** `pinv`, `regress`

**Purpose** Empirical (Kaplan-Meier) cumulative distribution function

**Syntax**

```
[f,x] = ecdf(y)
[f,x,flo,fup] = ecdf(y)
[...] = ecdf(y,'param1',value1,'param2',value2,...)
```

**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(y,'param1',value1,'param2',value2,...)` specifies additional parameter name-value pairs chosen from the following:

'censoring'	Boolean vector of the same size as <code>x</code> . 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 <code>x</code> containing non-negative integer counts. The <code>j</code> th element of this vector gives the number of times the <code>j</code> th element of <code>x</code> was observed. Default is 1 observation per element of <code>x</code> .
'alpha'	Value between 0 and 1 for a confidence level of $100 \times (1 - \text{alpha})\%$ . Default is <code>alpha=0.05</code> for 95% confidence.
'function'	Type of function returned as the <code>f</code> output argument, chosen from 'cdf' (default), 'survivor', or 'cumulative hazard'.

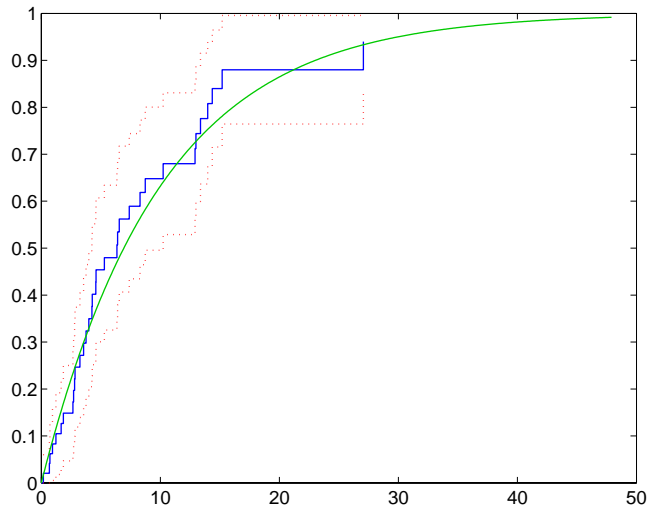
**Examples** Generate random failure times and random censoring times, and compare the empirical cdf with the known true cdf.

```
y = exprnd(10,50,1); % Random failure times are exponential(10)
d = exprnd(20,50,1); % Drop-out times are exponential(20)
t = min(y,d);        % Observe the minimum of these times
censored = (y>d);    % Observe also whether the subject failed
```

# ecdf

```
% Calculate and plot the empirical cdf and confidence bounds
[f,x,flo,fup] = ecdf(t,'censoring',censored);
stairs(x,f);
hold on;
stairs(x,flo,'r:'); stairs(x,fup,'r:');

% Superimpose a plot of the known true cdf
xx = 0:.1:max(t); yy = 1-exp(-xx/10); plot(xx,yy,'g-')
hold off;
```



## See Also

cdfplot

## References

[1] Cox, D.R. and D. Oakes, *Analysis of Survival Data*, Chapman & Hall, London, 1984.

**Purpose** Create histogram from output of ecdf

**Syntax**

```
n = ecdfhist(f, x)
n = ecdfhist(f, x, m)
n = ecdfhist(f, x, c)
[n, c] = ecdfhist(...)
ecdfhist(...)
```

**Description** `n = ecdfhist(f, x)` takes a vector `f` of empirical cumulative distribution function (cdf) values and a vector `x` of evaluation points, and returns a vector `n` containing the heights of histogram bars for 10 equally spaced bins. The function computes the bar heights from the increases in the empirical cdf, and normalizes them so that the area of the histogram is equal to 1. In contrast, `hist` produces bars whose heights represent bin counts.

`n = ecdfhist(f, x, m)`, where `m` is a scalar, uses `m` bins.

`n = ecdfhist(f, x, c)`, where `c` is a vector, uses bins with centers specified by `c`.

`[n, c] = ecdfhist(...)` also returns the position of the bin centers in `c`.

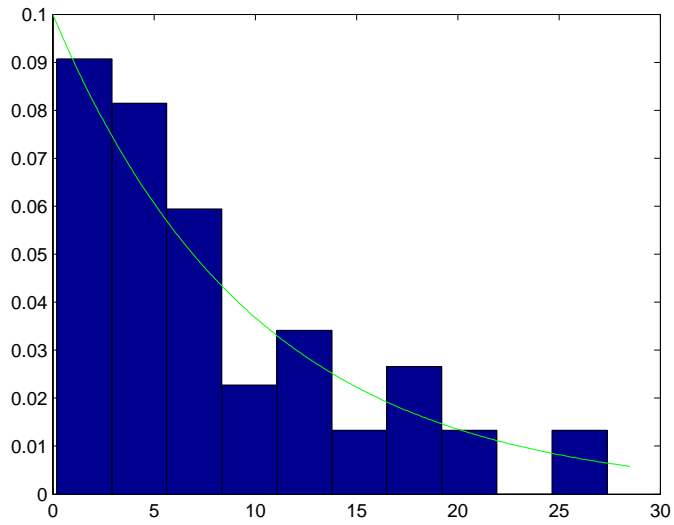
`ecdfhist(...)` without output arguments produces a histogram bar plot of the results.

**Example** The following code generates random failure times and random censoring times, and compares the empirical pdf with the known true pdf.

```
y = exprnd(10,50,1); % random failure times
d = exprnd(20,50,1); % drop-out times
t = min(y,d); % observe the minimum of these times
censored = (y>d); % observe whether the subject failed
% Calculate the empirical cdf and plot a histogram from it
[f,x] = ecdf(t,'censoring',censored);
ecdfhist(f,x);
% Superimpose a plot of the known true pdf
hold on;
xx = 0:.1:max(t); yy = exp(-xx/10)/10; plot(xx,yy,'g-');
hold off;
```

# ecdfhist

---



**See also** `ecdf`, `hist`, `histc`

**Purpose** Plot error bars along a curve

**Syntax**  
`errorbar(X,Y,L,U,symbol)`  
`errorbar(X,Y,L)`  
`errorbar(Y,L)`

**Description** `errorbar(X,Y,L,U,symbol)` plots X versus Y with error bars specified by L and U. X, Y, L, and U must be the same length. If X, Y, L, and U are matrices, then each column produces a separate line. The error bars are each drawn a distance of U(i) above and L(i) below the points in (X, Y). `symbol` is a string that controls the line type, plotting symbol, and color of the error bars.

`errorbar(X,Y,L)` plots X versus Y with symmetric error bars about Y.

`errorbar(Y,L)` plots Y with error bars [Y-L Y+L].

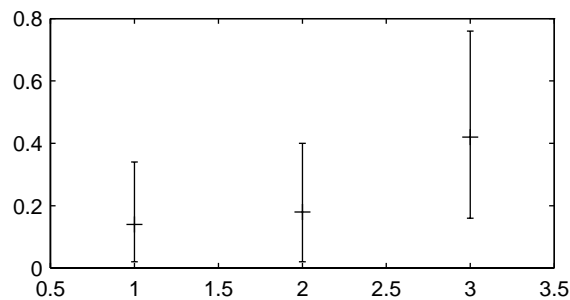
The `errorbar` function is a part of the standard MATLAB language.

## Example

```
lambda = (0.1:0.2:0.5);
r = poissrnd(lambda(ones(50,1),:));
[p,pci] = poissfit(r,0.001);
L = p - pci(1,:);
U = pci(2,:) - p;
errorbar(1:3,p,L,U,'+')
```

```
L =
    0.1200    0.1600    0.2600
```

```
U =
    0.2000    0.2200    0.3400
```



# evcdf

---

**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

$$\frac{\bar{X} - \mu}{\hat{\sigma}}$$

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 `Y` has a Weibull distribution, then `X = log(Y)` has the type 1 extreme value distribution.

**See Also** `cdf`, `evfit`, `evinv`, `evlike`, `evpdf`, `evrnd`, `evstat`



**Purpose** Parameter estimates and confidence intervals for extreme value data

**Syntax**

```
parmhat = evfit(data)
[parmhat, parmci] = evfit(data)
[parmhat, parmci] = evfit(data, alpha)
[...] = evfit(data, alpha, censoring)
[...] = evfit(data, alpha, censoring, freq)
[...] = evfit(data, alpha, censoring, freq, options)
```

**Description** `parmhat = evfit(data)` returns maximum likelihood estimates of the parameters of the type 1 extreme value distribution given the data in the vector `data`. `parmhat(1)` is the location parameter,  $\mu$ , and `parmhat(2)` is the scale parameter,  $\sigma$ .

`[parmhat, parmci] = evfit(data)` returns 95% confidence intervals for the parameter estimates on the  $\mu$  and  $\sigma$  parameters in the 2-by-2 matrix `parmci`. The first column of the matrix contains the lower and upper confidence bounds for the parameter  $\mu$ , and the second column contains the confidence bounds for the parameter  $\sigma$ .

`[parmhat, parmci] = evfit(data, alpha)` returns  $100(1 - \text{alpha})\%$  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  $Y$  has a Weibull distribution, then  $X = \log(Y)$  has the type 1 extreme value distribution.

## See Also

`evcdf`, `evinv`, `evlike`, `evpdf`, `evrnd`, `evstat`, `mle`, `statset`

**Purpose** Inverse of the extreme value cumulative distribution function

**Syntax**  
`X = evinv(P, MU, SIGMA)`  
`[X, XLO, XUP] = evinv(P, MU, SIGMA, PCOV, alpha)`

**Description** `X = evinv(P, MU, SIGMA)` returns the inverse cumulative distribution function (cdf) for a type 1 extreme value distribution with location parameter `MU` and scale parameter `SIGMA`, evaluated at the values in `P`. `P`, `MU`, and `SIGMA` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array of the same size as the other inputs. The default values for `MU` and `SIGMA` are 0 and 1, respectively.

`[X, XLO, XUP] = evinv(P, MU, SIGMA, PCOV, alpha)` produces confidence bounds for `X` when the input parameters `MU` and `SIGMA` are estimates. `PCOV` is the covariance matrix of the estimated parameters. `alpha` is a scalar that specifies 100(1 - `alpha`)% confidence bounds for the estimated parameters, and has a default value of 0.05. `XLO` and `XUP` are arrays of the same size as `X` containing the lower and upper confidence bounds.

The function `evinv` computes confidence bounds for `P` using a normal approximation to the distribution of the estimate

$$\hat{\mu} + \hat{\sigma}q$$

where  $q$  is the  $P$ th quantile from an extreme value distribution with parameters  $\mu = 0$  and  $\sigma = 1$ . The computed bounds give approximately the desired confidence level when you estimate `MU`, `SIGMA`, and `PCOV` from large samples, but in smaller samples other methods of computing the confidence bounds might be more accurate.

The type 1 extreme value distribution is also known as the Gumbel distribution. If  $Y$  has a Weibull distribution, then  $X = \log(Y)$  has the type 1 extreme value distribution.

**See Also** `evcdf`, `evfit`, `evlike`, `evpdf`, `evrnd`, `evstat`, `icdf`

# evlike

---

**Purpose** Negative log-likelihood for the extreme value distribution

**Syntax**

```
nlogL = evlike(params, data)
[nlogL, AVAR] = evlike(params, data)
[...] = evlike(params, data, censoring)
[...] = evlike(params, data, censoring, freq)
```

**Description** `nlogL = evlike(params, data)` returns the negative of the log-likelihood for the type 1 extreme value distribution, evaluated at parameters `params(1) = MU` and `params(2) = SIGMA`, given `data`. `nlogL` is a scalar.

`[nlogL, AVAR] = evlike(params, data)` returns the inverse of Fisher's information matrix, `AVAR`. If the input parameter values in `params` are the maximum likelihood estimates, the diagonal elements of `AVAR` are their asymptotic variances. `AVAR` is based on the observed Fisher's information, not the expected information.

`[...] = evlike(params, data, censoring)` accepts a Boolean vector of the same size as `data`, which is 1 for observations that are right-censored and 0 for observations that are observed exactly.

`[...] = evlike(params, data, censoring, freq)` accepts a frequency vector of the same size as `data`. `freq` typically contains integer frequencies for the corresponding elements in `data`, but can contain any nonnegative values. Pass in `[]` for `censoring` to use its default value.

The type 1 extreme value distribution is also known as the Gumbel distribution. If `Y` has a Weibull distribution, then `X = log(Y)` has the type 1 extreme value distribution.

**See Also** `evcdf`, `evfit`, `evinv`, `evpdf`, `evrnd`, `evstat`

---

<b>Purpose</b>	Extreme value probability density function
<b>Syntax</b>	$Y = \text{evpdf}(X, \text{MU}, \text{SIGMA})$
<b>Description</b>	<p><math>Y = \text{evpdf}(X, \text{MU}, \text{SIGMA})</math> returns the pdf of the type 1 extreme value distribution with location parameter <math>\text{MU}</math> and scale parameter <math>\text{SIGMA}</math>, evaluated at the values in <math>X</math>. <math>X</math>, <math>\text{MU}</math>, and <math>\text{SIGMA}</math> 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 <math>\text{MU}</math> and <math>\text{SIGMA}</math> are 0 and 1, respectively.</p> <p>The type 1 extreme value distribution is also known as the Gumbel distribution. If <math>Y</math> has a Weibull distribution, then <math>X = \log(Y)</math> has the type 1 extreme value distribution.</p>
<b>See Also</b>	<code>evcdf</code> , <code>evfit</code> , <code>evinv</code> , <code>evlike</code> , <code>evrnd</code> , <code>evstat</code> , <code>pdf</code>

# evrnd

---

**Purpose** Random matrices from the extreme value distribution

**Syntax**

```
R = evrnd(MU, SIGMA)
R = evrnd(MU, SIGMA, v)
R = evrnd(MU, SIGMA, m, n)
```

**Description** R = evrnd(MU, SIGMA) generates random numbers from the extreme value distribution with parameters specified by MU and SIGMA. MU and SIGMA can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of R. A scalar input for MU or SIGMA is expanded to a constant array with the same dimensions as the other input.

R = evrnd(MU, SIGMA, v) generates an array R of size v containing random numbers from the extreme value distribution with parameters MU and SIGMA, where v is a row vector. If v is a 1-by-2 vector, R is a matrix with v(1) rows and v(2) columns. If v is 1-by-n, R is an n-dimensional array.

If MU and SIGMA are both scalars, R = evrnd(MU, SIGMA, m, n) returns an m-by-n matrix.

The type 1 extreme value distribution is also known as the Gumbel distribution. If Y has a Weibull distribution, then  $X = \log(Y)$  has the type 1 extreme value distribution.

## Reproducing the Output of evrnd

evrnd uses the MATLAB function rand to generate random numbers. When you call evrnd, you change the current state of rand, and thereby alter the output of subsequent calls to evrnd or any other functions that depend on rand. If you want to reproduce the output of evrnd, reset the state of rand to the same fixed value each time you call evrnd. For an example of how to do this, and a list of the Statistics Toolbox functions that depend on rand, see “Reproducing the Output of Random Number Functions” on page 2-46.

**See Also** evcdf, evfit, evinv, evlike, evpdf, evstat

**Purpose** Mean and variance of the extreme value distribution

**Syntax** `[M, V] = evstat(MU, SIGMA)`

**Description** `[M, V] = evstat(MU, SIGMA)` returns the mean and variance of 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 Y has a Weibull distribution, then  $X = \log(Y)$  has the type 1 extreme value distribution.

**See Also** `evcdf`, `evfit`, `evinv`, `evlike`, `evpdf`, `evrnd`

# ewmplot

---

**Purpose** Exponentially Weighted Moving Average (EWMA) chart for Statistical Process Control (SPC)

**Syntax**

```
ewmplot(data)
ewmplot(data,lambda)
ewmplot(data,lambda,alpha)
ewmplot(data,lambda,alpha,specs)
h = ewmplot(...)
```

**Description** `ewmplot(data)` produces an EWMA chart of the grouped responses in `data`. The rows of `data` contain replicate observations taken at a given time. The rows should be in time order.

`ewmplot(data,lambda)` produces an EWMA chart of the grouped responses in `data`, and specifies how much the current prediction is influenced by past observations. Higher values of `lambda` give more weight to current observations. By default, `lambda = 0.4`; `lambda` must be between 0 and 1.

`ewmplot(data,lambda,alpha)` produces an EWMA chart of the grouped responses in `data`, and specifies the significance level of the upper and lower plotted confidence limits. `alpha` is 0.0027 by default. This value produces three-sigma limits:

```
norminv(1-0.0027/2)
```

```
ans =
      3
```

To get  $k$ -sigma limits, use the expression  $2*(1 - \text{normcdf}(k))$ . For example, the correct `alpha` value for 2-sigma limits is 0.0455, as shown below.

```
k = 2;
2*(1-normcdf(k))
```

```
ans =
      0.0455
```

`ewmplot(data,lambda,alpha,specs)` produces an EWMA chart of the grouped responses in `data`, and specifies a two-element vector, `specs`, for the lower and upper specification limits of the response.

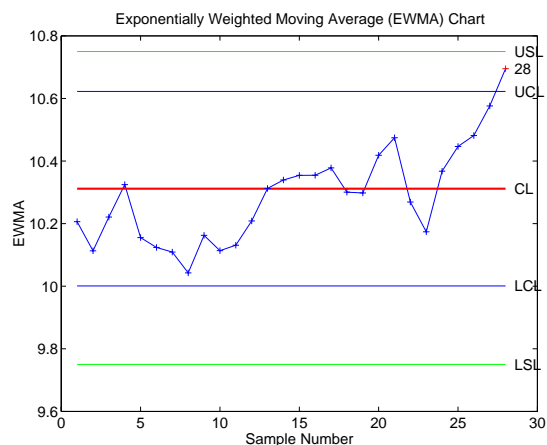


`h = ewmaplot(...)` returns a vector of handles to the plotted lines.

## Example

Consider a process with a slowly drifting mean. An EWMA chart is preferable to an x-bar chart for monitoring this kind of process. The simulation below demonstrates an EWMA chart for a slow linear drift.

```
t = (1:28)';
r = normrnd(10+0.02*t(:,ones(4,1))),0.5);
ewmaplot(r,0.4,0.01,[9.75 10.75])
```



The EWMA value for group 28 is higher than would be expected purely by chance. If you had been monitoring this process continuously, you would have detected the drift when group 28 was collected, and would have had an opportunity to investigate its cause.

## Reference

[1] Montgomery, D., *Introduction to Statistical Quality Control*, John Wiley & Sons 1991. p. 299.

## See Also

`xbarplot`, `schart`

# expcdf

**Purpose** Exponential cumulative distribution function (cdf)

**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  $\mu \cdot \log(2)$ .

```
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,  $\mu$ ?

```
mu = 1:6;  
x = mu;  
p = expcdf(x,mu)  
  
p =  
    0.6321    0.6321    0.6321    0.6321    0.6321    0.6321
```

## See Also

[cdf](#), [expfit](#), [expinv](#), [exppdf](#), [exprnd](#), [expstat](#)

# expfit

---

**Purpose** Parameter estimates and confidence intervals for exponential data

**Syntax**

```
parmhat = expfit(DATA)
[parmhat,parmci] = expfit(DATA)
[parmhat,parmci] = expfit(DATA,alpha)
[...] = expfit(x, alpha, censoring)
[...] = expfit(x, alpha, censoring, freq)
```

**Description** `parmhat = expfit(DATA)` returns estimates of the parameter,  $\mu$ , of the exponential distribution, given the data in DATA. Each entry of `parmhat` corresponds to the data in a column of DATA.

`[parmhat, parmci] = expfit(DATA)` returns 95% confidence intervals for the parameter estimates in matrix `parmci`. The first row of `parmci` contains the lower bounds of the confidence intervals, and the second row contains the upper bounds.

`[parmhat, parmci] = expfit(DATA, alpha)` returns  $100(1 - \text{alpha})\%$  confidence intervals for the parameter estimates, where `alpha` is a value in the range `[0 1]` specifying the width of the confidence intervals. By default, `alpha` is 0.05, which corresponds to 95% confidence intervals.

`[...] = expfit(data, alpha, censoring)` accepts a Boolean vector, `censoring`, of the same size as `data`, which is 1 for observations that are right-censored and 0 for observations that are observed exactly. `data` must be a vector in order to pass in the argument `censoring`.

`[...] = expfit(data, alpha, censoring, freq)` accepts a frequency vector, `freq` of the same size as `data`. Typically, `freq` contains integer frequencies for the corresponding elements in `data`, but can contain any nonnegative values. Pass in `[]` for `alpha`, `censoring`, or `freq` to use their default values.

**Example** This example generates 100 independent samples of exponential data with  $\mu = 3$ . `muhat` is an estimate of  $\mu$  and `muci` is a 99% confidence interval around `muhat`. Notice that `muci` contains the true value of  $\mu$ .

```
data = exprnd(3, 100, 1);
[parmhat, parmci] = expfit(data, 0.01)

parmhat =
```

2.7292

parmci =

2.1384

3.5854

**See Also**

expcdf, expinv, explike, exppdf, exprnd, expstat, mle, statset

# expinv

---

**Purpose** Inverse of the exponential cumulative distribution function (cdf)

**Syntax** `X = expinv(P, MU)`  
`[P, PLO, PUP] = 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  $\mu$  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**`expcdf, expfit, exppdf, exprnd, expstat, icdf`

# explike

---

**Purpose** Negative log-likelihood for the exponential distribution

**Syntax**

```
nlogL = explike(param, data)
[nlogL, avar] = explike(param, data)
[...] = explike(param, data, censoring)
[...] = explike(param, data, censoring, freq)
```

**Description** `nlogL = explike(param, data)` returns the negative of the log-likelihood for the exponential distribution, evaluated at the parameter `param = MU`, given `data`. `nlogL` is a scalar.

`[nlogL, avar] = explike(param, data)` returns the inverse of Fisher's information, `avar`, a scalar. If the input parameter value in `param` is the maximum likelihood estimate, `avar` is its asymptotic variance. `avar` is based on the observed Fisher's information, not the expected information.

`[...] = explike(param, data, censoring)` accepts a Boolean vector, `censoring`, of the same size as `data`, which is 1 for observations that are right-censored and 0 for observations that are observed exactly.

`[...] = explike(param, data, censoring, freq)` accepts a frequency vector, `freq`, of the same size as `data`. The vector `freq` typically contains integer frequencies for the corresponding elements in `data`, but can contain any nonnegative values. Pass in `[]` for `censoring` to use its default value.

**See Also** `expcdf`, `expfit`, `expinv`, `exppdf`, `exprnd`



**Purpose** Exponential probability density function (pdf)

**Syntax** `Y = exp pdf(X, MU)`

**Description** `Y = exp pdf(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 = exp pdf(5,1:5)
```

```
y =
    0.0067    0.0410    0.0630    0.0716    0.0736
```

```
y = exp pdf(1:5,1:5)
```

```
y =
    0.3679    0.1839    0.1226    0.0920    0.0736
```

**See Also** `expcdf`, `expfit`, `expinv`, `exprnd`, `expstat`, `pdf`

# exprnd

---

**Purpose** Generate random numbers from the exponential distribution

**Syntax**

```
R = exprnd(MU)
R = exprnd(MU,v)
R = exprnd(MU,m,n)
```

**Description** `R = exprnd(MU)` generates exponential random numbers with mean `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 exponential random numbers 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 exponential random numbers with mean `MU`, where scalars `m` and `n` are the row and column dimensions of `R`.

## Reproducing the Output of exprnd

`exprnd` uses the MATLAB function `rand` to generate random numbers. When you call `exprnd`, you change the current state of `rand`, and thereby alter the output of subsequent calls to `exprnd` or any other functions that depend on `rand`. If you want to reproduce the output of `exprnd`, reset the state of `rand` to the same fixed value each time you call `exprnd`. For an example of how to do this, and a list of the Statistics Toolbox functions that depend on `rand`, see “Reproducing the Output of Random Number Functions” on page 2-46.

---

**Note** The results in the following examples depend on the current state of `rand`. If you run the code in these examples, your results might differ from the answers shown here.

---

## Examples

```
n1 = exprnd(5:10)

n1 =
    7.5943    18.3400    2.7113    3.0936    0.6078    9.5841

n2 = exprnd(5:10,[1 6])
```

```
n2 =  
    3.2752    1.1110   23.5530   23.4303    5.7190    3.9876  
  
n3 = exprnd(5,2,3)  
  
n3 =  
    24.3339    13.5271    1.8788  
     4.7932     4.3675     2.6468
```

**See Also**

expcdf, expfit, expinv, exppdf, expstat

# expstat

---

**Purpose** Mean and variance for the exponential distribution

**Syntax** `[m,v] = expstat(mu)`

**Description** `[m,v] = expstat(mu)` returns the mean 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  $\mu$ , and the variance is  $\mu^2$ .

**Examples**

```
[m,v] = expstat([1 10 100 1000])  
  
m =  
      1      10      100      1000  
  
v =  
      1      100     10000    1000000
```

**See Also** `expcdf`, `expfit`, `expinv`, `exppdf`, `exprnd`

**Purpose** Maximum likelihood common factor analysis

**Syntax**

```
lambda = factoran(X,m)
[lambda,psi] = factoran(X,m)
[lambda,psi,T] = factoran(X,m)
[lambda,psi,T,stats] = factoran(X,m)
[lambda,psi,T,stats,F] = factoran(X,m)
[...] = factoran(...,'param1',value1,'param2',value2,...)
```

**Definition** factoran computes the maximum likelihood estimate (MLE) of the factor loadings matrix  $\Lambda$  in the factor analysis model

$$x = \mu + \Lambda f + e$$

where  $x$  is a vector of observed variables,  $\mu$  is a constant vector of means,  $\Lambda$  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.  $x$ ,  $\mu$ , 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  $j$ th factor for the  $i$ th variable. By default, factoran calls the function rotatefactors to rotate the estimated factor loadings using the 'varimax' option.

[lambda,psi] = factoran(X,m) also returns maximum likelihood estimates of the specific variances as a column vector psi of length  $d$ .

[lambda,psi,T] = factoran(X,m) also returns the  $m$ -by- $m$  factor loadings rotation matrix  $T$ .

# 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 fields:

`loglike` Maximized log-likelihood value  
`dfe` Error degrees of freedom =  $((d-m)^2 - (d+m))/2$   
`chisq` Approximate chi-squared statistic for the null hypothesis  
`p` Right-tail significance level for the null hypothesis

`factoran` does not compute the `chisq` and `p` fields unless `dfe` is positive, and all the specific variance estimates in `psi` are positive (see “Heywood Case” 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',value1,'param2',value2,...)` enables you to specify optional parameter name/value pairs to control the model fit and the outputs. These are the valid parameters. The most commonly used parameters are listed first.

Parameter	Value
'xtype'	The type of input in the matrix $X$ . 'xtype' can be one of:
	'data' Raw data (default)
	'covariance' Positive definite covariance or correlation matrix
'scores'	Method for predicting factor scores. 'scores' is ignored if $X$ is not raw data.

Parameter	Value	
	'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. May be specified as:	
	'random'	Chooses $d$ uniformly distributed values on the interval [0,1].
	'Rsquared'	Chooses the starting vector as a scale factor times $\text{diag}(\text{inv}(\text{corrcoef}(X)))$ (default). E.g., see Jöreskog [2].
	Positive integer	Performs the given number of maximum likelihood fits, each initialized as with 'random'. factoran returns the fit with the highest likelihood.
	Matrix	Performs one maximum likelihood fit for each column of the specified matrix. The $i$ th optimization is initialized with the values from the $i$ th column. The matrix must have $d$ rows.
'rotate'	Method used to rotate factor loadings and scores. 'rotate' can have the same values as the 'Method' parameter of rotatefactors. See the reference page for rotatefactors for a full description of the available methods.	
	'none'	Performs no rotation

# factoran

Parameter	Value
	'equamax' Special case of the orthomax rotation. Use the 'normalize', 'reltol', and 'maxit' parameters to control the details of the rotation.
	'orthomax' Orthogonal rotation that maximizes a criterion based on the variance of the loadings. Use the 'coeff', 'normalize', 'reltol', and 'maxit' parameters to control the details of the rotation.
	'parsimax' Special case of the orthomax rotation (default). Use the 'normalize', 'reltol', and 'maxit' parameters to control the details of the rotation.
	'pattern' Performs either an oblique rotation (the default) or an orthogonal rotation to best match a specified pattern matrix. Use the 'type' parameter to choose the type of rotation. Use the 'target' parameter to specify the pattern matrix.
	'procrustes' Performs either an oblique (the default) or an orthogonal rotation to best match a specified target matrix in the least squares sense. Use the 'type' parameter to choose the type of rotation. Use 'target' to specify the target matrix.



Parameter	Value
	<p>'promax'</p> <p>Performs an oblique procrustes rotation to a target matrix determined by factoran as a function of an orthomax solution.</p> <p>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'.</p>
	<p>'quartimax'</p> <p>Special case of the orthomax rotation (default). Use the 'normalize', 'reltol', and 'maxit' parameters to control the details of the rotation.</p>
	<p>'varimax'</p> <p>Special case of the orthomax rotation (default). Use the 'normalize', 'reltol', and 'maxit' parameters to control the details of the rotation.</p>
	<p>Function</p> <p>Function handle to rotation function of the form</p> <pre>[B,T] = myrotation(A,...)</pre> <p>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.</p> <p>Use the factoran parameter 'userargs' to pass additional arguments to this rotation function. See Example 4.</p>

Parameter	Value
'coeff'	Coefficient, often denoted as $\gamma$ , 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 $\sqrt{\text{eps}}$ .
'maxit'	Iteration limit for 'orthomax' or 'varimax' rotation. Default is 250.
'target'	Target factor loading matrix for 'procrustes' rotation. Required for 'procrustes' rotation. No default value.
'type'	Type of 'procrustes' rotation. Can be 'oblique' (default) or 'orthogonal'.
'power'	Exponent for creating the target matrix in the 'promax' rotation. Must be $\geq 1$ . Default is 4.
'userargs'	Denotes the beginning of additional input values for a user-defined rotation function. factoran appends all subsequent values, in order and without processing, to the rotation function argument list, following the unrotated factor loadings matrix A. See Example 4.
'nobs'	If X is a covariance or correlation matrix, indicates the number of observations that were used in its estimation. This allows calculation of significance for the null hypothesis even when the original data are not available. There is no default. 'nobs' is ignored if X is raw data.

Parameter	Value
'delta'	A 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 <code>statset</code> . Enter <code>statset('factoran')</code> to see the names and default values of the parameters that <code>factoran</code> accepts in the options structure. See the reference page for <code>statset</code> for more information about these options.

## Remarks

**Observed Data Variables.** The variables in the observed data matrix  $X$  must be linearly independent, i.e.,  $\text{cov}(X)$  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.,  $\lambda \lambda' + \text{diag}(\psi)$  is an estimate of the correlation matrix of the original data  $X$  (although not after an oblique rotation). See Examples 1 and 3.

**Heywood Case.** If elements of  $\psi$  are equal to the value of the 'delta' parameter (i.e, they are essentially zero), the fit is known as a Heywood case, and interpretation of the resulting estimates is problematic. In particular, there may 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.,  $\lambda = \lambda_0 T$ , where  $\lambda_0$  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  $\text{inv}(T')$ , i.e.,  $F = F_0 * \text{inv}(T')$ , where  $F_0$  contains the unrotated predictions. The estimated covariance of  $F$  is  $\text{inv}(T' * T)$ , 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 of X
Lambda*inv(T)      % Unrotate the loadings
F*T'               % Unrotate the factor scores
```

**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 correlation matrix of F, no longer eye(2)
Lambda*inv(T'*T)*Lambda' + diag(Psi) % Est'd correlation
                                % matrix of X
```

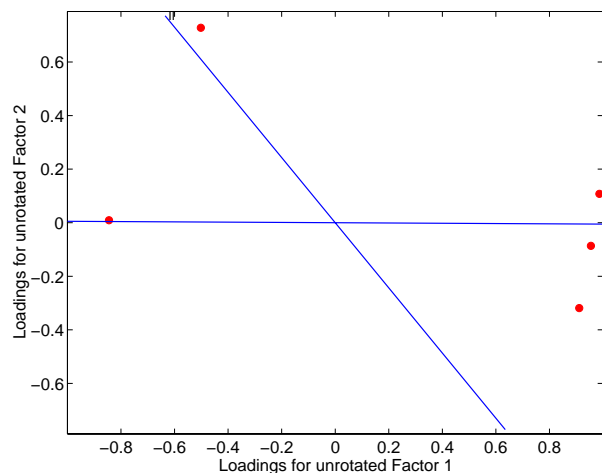
Plot the rotated variables against the oblique axes.

```
invT = inv(T)
Lambda0 = Lambda*invT
plot(Lambda0(:,1),Lambda0(:,2), 'ro');
```

```

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)]);
text(invT(:,1), invT(:,2),[' I '; ' II']);
xlabel('Loadings for unrotated Factor 1')
ylabel('Loadings for unrotated Factor 2')

```



**Example 4.** Syntax for passing additional arguments to a user-defined rotation function.

```

[Lambda,Psi,T] = ...
    factoran(X,2,'rotate',@myrotation,'userargs',1,'two')

```

## See Also

princomp, procrustes, pcacov, rotatefactors, statset

## References

- [1] Harman, H.H., *Modern Factor Analysis*, 3rd Ed., University of Chicago Press, Chicago, 1976.
- [2] Jöreskog, K.G., “Some Contributions to Maximum Likelihood Factor Analysis”, *Psychometrika*, Vol.32, pp.443-482, 1967.
- [3] Lawley, D.N. and A.E. Maxwell, *Factor Analysis as a Statistical Method*, 2nd Ed., American Elsevier Pub. Co., New York, 1971.

# fcdf

---

**Purpose** F cumulative distribution function (cdf)

**Syntax**  $P = \text{fcdf}(X, V1, V2)$

**Description**  $P = \text{fcdf}(X, V1, V2)$  computes the F cdf at each of the values in  $X$  using the corresponding parameters in  $V1$  and  $V2$ .  $X$ ,  $V1$ , and  $V2$  can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant matrix with the same dimensions as the other inputs. The parameters in  $V1$  and  $V2$  must be positive integers.

The F cdf is

$$p = F(x|v_1, v_2) = \int_0^x \frac{\Gamma\left[\frac{(v_1 + v_2)}{2}\right]}{\Gamma\left(\frac{v_1}{2}\right)\Gamma\left(\frac{v_2}{2}\right)} \left(\frac{v_1}{v_2}\right)^{\frac{v_1}{2}} \frac{t^{\frac{v_1-2}{2}}}{\left[1 + \left(\frac{v_1}{v_2}\right)t\right]^{\frac{v_1 + v_2}{2}}} dt$$

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** This example illustrates an important and useful mathematical identity for the F distribution.

```
nu1 = 1:5;
nu2 = 6:10;
x = 2:6;
F1 = fcdf(x, nu1, nu2)

F1 =
    0.7930    0.8854    0.9481    0.9788    0.9919

F2 = 1 - fcdf(1./x, nu2, nu1)

F2 =
    0.7930    0.8854    0.9481    0.9788    0.9919
```

**See Also** `cdf`, `finv`, `fpdf`, `frnd`, `fstat`

**Purpose** Two-level full-factorial designs

**Syntax** `X = ff2n(n)`

**Description** `X = ff2n(n)` creates a two-level full-factorial design, `X`, where `n` is the desired number of columns of `X`. The number of rows in `X` is  $2^n$ .

**Example** `X = ff2n(3)`

```
X =  
    0    0    0  
    0    0    1  
    0    1    0  
    0    1    1  
    1    0    0  
    1    0    1  
    1    1    0  
    1    1    1
```

`X` is the binary representation of the numbers from 0 to  $2^n-1$ .

**See Also** `fracfact`, `fullfact`

# finv

---

**Purpose** Inverse of the F cumulative distribution function (cdf)

**Syntax**  $X = \text{finv}(P, V1, V2)$

**Description**  $X = \text{finv}(P, V1, V2)$  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

$$x = F^{-1}(p|v_1, v_2) = \{x: F(x|v_1, v_2) = p\}$$

where

$$p = F(x|v_1, v_2) = \int_0^x \frac{\Gamma\left(\frac{v_1 + v_2}{2}\right)}{\Gamma\left(\frac{v_1}{2}\right)\Gamma\left(\frac{v_2}{2}\right)} \left(\frac{v_1}{v_2}\right)^{\frac{v_1}{2}} \frac{t^{\frac{v_1-2}{2}}}{\left[1 + \left(\frac{v_1}{v_2}\right)t\right]^{\frac{v_1 + v_2}{2}}} dt$$

**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.

$$x = \text{finv}(0.95, 5, 10)$$

$$x = 3.3258$$

You would observe values greater than 3.3258 only 5% of the time by chance.

**See Also** `fcd`, `fpdf`, `frnd`, `fstat`, `icdf`



**Purpose** F probability density function (pdf)

**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

$$y = f(x|v_1,v_2) = \frac{\Gamma\left[\frac{(v_1+v_2)}{2}\right]}{\Gamma\left(\frac{v_1}{2}\right)\Gamma\left(\frac{v_2}{2}\right)} \left(\frac{v_1}{v_2}\right)^{\frac{v_1}{2}} \frac{x^{\frac{v_1-2}{2}}}{\left[1 + \left(\frac{v_1}{v_2}\right)x\right]^{\frac{v_1+v_2}{2}}}$$

**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**

`fcdf`, `finv`, `frnd`, `fstat`, `pdf`

# fracfact

---

**Purpose** Generate fractional factorial design from generators

**Syntax**

```
x = fracfact('gen')  
[x,conf] = fracfact('gen')
```

**Description** `x = fracfact('gen')` generates a fractional factorial design as specified by the generator string `gen`, and returns a matrix `x` of design points. The input string `gen` is a generator string consisting of “words” separated by spaces. Each word describes how a column of the output design should be formed from columns of a full factorial. Typically `gen` will include single-letter words for the first few factors, plus additional multiple-letter words describing how the remaining factors are confounded with the first few.

The output matrix `x` is a fraction of a two-level full-factorial design. Suppose there are  $m$  words in `gen`, and that each word is formed from a subset of the first  $n$  letters of the alphabet. The output matrix `x` has  $2^n$  rows and  $m$  columns. Let `F` represent the two-level full-factorial design as produced by `ff2n(n)`. The values in column  $j$  of `x` are computed by multiplying together the columns of `F` corresponding to letters that appear in the  $j$ th word of the generator string.

`[x,conf] = fracfact('gen')` also returns a cell array, `conf`, that describes the confounding pattern among the main effects and all two-factor interactions.

## Examples

### Example 1

You want to run an experiment to study the effects of four factors on a response, but you can only afford eight runs. (A run is a single repetition of the experiment at a specified combination of factor values.) Your goal is to determine which factors affect the response. There may be interactions between some pairs of factors.

A total of sixteen runs would be required to test all factor combinations. However, if you are willing to assume there are no three-factor interactions, you can estimate the main factor effects in just eight runs.

```
[x,conf] = fracfact('a b c abc')
```

```
x =  
    -1    -1    -1    -1  
    -1    -1     1     1
```

```

-1    1    -1    1
-1    1     1   -1
 1   -1   -1    1
 1   -1    1   -1
 1    1   -1   -1
 1    1    1    1

```

conf =

'Term'	'Generator'	'Confounding'
'X1'	'a'	'X1'
'X2'	'b'	'X2'
'X3'	'c'	'X3'
'X4'	'abc'	'X4'
'X1*X2'	'ab'	'X1*X2 + X3*X4'
'X1*X3'	'ac'	'X1*X3 + X2*X4'
'X1*X4'	'bc'	'X1*X4 + X2*X3'
'X2*X3'	'bc'	'X1*X4 + X2*X3'
'X2*X4'	'ac'	'X1*X3 + X2*X4'
'X3*X4'	'ab'	'X1*X2 + X3*X4'

The first three columns of the x matrix form a full-factorial design. The final column is formed by multiplying the other three. The confounding pattern shows that the main effects for all four factors are estimable, but the two-factor interactions are not. For example, the X1\*X2 and X3\*X4 interactions are confounded, so it is not possible to estimate their effects separately.

After conducting the experiment, you may find out that the 'ab' effect is significant. In order to determine whether this effect comes from X1\*X2 or X3\*X4 you would have to run the remaining eight runs. You can obtain those runs by reversing the sign of the final generator.

```
fracfact('a b c -abc')
```

ans =

```

-1   -1   -1    1
-1   -1    1   -1
-1    1   -1   -1
-1    1    1    1
 1   -1   -1   -1
 1   -1    1    1
 1    1   -1    1

```

1 1 1 -1

## Example 2

Suppose now you need to study the effects of eight factors. A full factorial would require 256 runs. By clever choice of generators, you can find a sixteen-run design that can estimate those eight effects with no confounding from two-factor interactions.

```
[x,c] = fracfact('a b c d abc acd abd bcd');
c(1:10,:)
```

```
ans =
```

'Term'	'Generator'	'Confounding'
'X1'	'a'	'X1'
'X2'	'b'	'X2'
'X3'	'c'	'X3'
'X4'	'd'	'X4'
'X5'	'abc'	'X5'
'X6'	'acd'	'X6'
'X7'	'abd'	'X7'
'X8'	'bcd'	'X8'
'X1*X2'	'ab'	'X1*X2 + X3*X5 + X4*X7 + X6*X8'

This confounding pattern shows that the main effects are not confounded with two-factor interactions. The final row shown reveals that a group of four two-factor interactions is confounded. Other choices of generators would not have the same desirable property.

```
[x,c] = fracfact('a b c d ab cd ad bc');
c(1:10,:)
```

```
ans =
```

'Term'	'Generator'	'Confounding'
'X1'	'a'	'X1 + X2*X5 + X4*X7'
'X2'	'b'	'X2 + X1*X5 + X3*X8'
'X3'	'c'	'X3 + X2*X8 + X4*X6'
'X4'	'd'	'X4 + X1*X7 + X3*X6'
'X5'	'ab'	'X5 + X1*X2'
'X6'	'cd'	'X6 + X3*X4'
'X7'	'ad'	'X7 + X1*X4'

---

'X8'	'bc'	'X8 + X2*X3'
'X1*X2'	'ab'	'X5 + X1*X2'

Here all the main effects are confounded with one or more two-factor interactions.

**References**

[1] Box, G. A. F., W. G. Hunter, and J. S. Hunter (1978), *Statistics for Experimenters*, Wiley, New York.

**See Also**

ff2n, fullfact, hadamard

# friedman

---

**Purpose** Friedman's nonparametric two-way Analysis of Variance (ANOVA)

**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  $\mu$  is an overall location parameter,  $\alpha_i$  represents the column effect,  $\beta_j$  represents the row effect, and  $\varepsilon_{ijk}$  represents the error. This test ranks the data within each level of B, and tests for a difference across levels of A. The `p` that `friedman` returns is the p-value for the null hypothesis that  $\alpha_i = 0$ . If the p-value is near zero, this casts doubt on the null hypothesis. A sufficiently

small p-value suggests that at least one column-sample median is significantly different than the others; i.e., there is a main effect due to factor A. The choice of a critical p-value to determine whether a result is “statistically significant” is left to the researcher. It is common to declare a result significant if the p-value is less than 0.05 or 0.01.

`friedman` also displays a figure showing an ANOVA table, which divides the variability of the ranks into two or three parts:

- The variability due to the differences among the column effects
- The variability due to the interaction between rows and columns (if `reps` is greater than its default value of 1)
- The remaining variability not explained by any systematic source

The ANOVA table has six columns:

- The first shows the source of the variability.
- The second shows the Sum of Squares (SS) due to each source.
- The third shows the degrees of freedom (df) associated with each source.
- The fourth shows the Mean Squares (MS), which is the ratio SS/df.
- The fifth shows Friedman’s chi-square statistic.
- The sixth shows the p-value for the chi-square statistic.

`p = friedman(X, reps, 'displayopt')` enables the ANOVA table display when `'displayopt'` is `'on'` (default) and suppresses the display when `'displayopt'` is `'off'`.

`[p, table] = friedman(...)` returns the ANOVA table (including column and row labels) in cell array `table`. (You can copy a text version of the ANOVA table to the clipboard by selecting **Copy Text** from the **Edit** menu.)

`[p, table, stats] = friedman(...)` returns a `stats` structure that you can use to perform a follow-up multiple comparison test. The `friedman` test evaluates the hypothesis that the column effects are all the same against the alternative that they are not all the same. Sometimes it is preferable to perform a test to determine which pairs of column effects are significantly different, and which are not. You can use the `multcompare` function to perform such tests by supplying the `stats` structure as input.

## Assumptions

Friedman's test makes the following assumptions about the data in X:

- All data come from populations having the same continuous distribution, apart from possibly different locations due to column and row effects.
- All observations are mutually independent.

The classical two-way ANOVA replaces the first assumption with the stronger assumption that data come from normal distributions.

## Examples

Let's repeat the example from the `anova2` function, this time applying Friedman's test. Recall that the data below come from a study of popcorn brands and popper type (Hogg 1987). The columns of the matrix `popcorn` are brands (Gourmet, National, and Generic). The rows are popper type (Oil and Air). The study popped a batch of each brand three times with each popper. The values are the yield in cups of popped popcorn.

```
load popcorn
popcorn
popcorn =
  5.5000  4.5000  3.5000
  5.5000  4.5000  4.0000
  6.0000  4.0000  3.0000
  6.5000  5.0000  4.0000
  7.0000  5.5000  5.0000
  7.0000  5.0000  4.5000

p = friedman(popcorn,3)
p =
  0.0010
```

Source	SS	df	MS	Chi-sq	Prob>Chi-sq
Columns	99.75	2	49.875	13.76	0.001
Interaction	0.0833	2	0.0417		
Error	16.1667	12	1.3472		
Total	116	17			

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`.

You could also test popper type by permuting the popcorn array as described on “Friedman’s Test” on page 4-60 and repeating the test.

## References

[1] Hogg, R. V. and J. Ledolter, *Engineering Statistics*, MacMillan Publishing Company, 1987.

[2] Hollander, M. and D. A. Wolfe, *Nonparametric Statistical Methods*, Wiley, 1973.

## See Also

`anova2`, `multcompare`, `kruskalwallis`

# frnd

---

**Purpose** Random numbers from the F distribution

**Syntax**

```
R = frnd(V1,V2)
R = frnd(V1,V2,v)
R = frnd(V1,V2,m,n)
```

**Description** `R = frnd(V1,V2)` generates random numbers from the F distribution with numerator degrees of freedom `V1` and denominator degrees of freedom `V2`. `V1` and `V2` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input for `V1` or `V2` is expanded to a constant array with the same dimensions as the other input.

`R = frnd(V1,V2,v)` generates random numbers from the F distribution with parameters `V1` and `V2`, where `v` is a row vector. If `v` is a 1-by-2 vector, `R` is a matrix with `v(1)` rows and `v(2)` columns. If `v` is 1-by-`n`, `R` is an `n`-dimensional array.

`R = frnd(V1,V2,m,n)` generates random numbers from the F distribution with parameters `V1` and `V2`, where scalars `m` and `n` are the row and column dimensions of `R`.

## Reproducing the Output of frnd

`frnd` uses the MATLAB functions `rand` and `randn` to generate random numbers. When you call `frnd`, you change the current states of `rand` and `randn`, and thereby alter the output of subsequent calls to `frnd` or any other functions that depend on `rand` or `randn`. If you want to reproduce the output of `frnd`, reset the states of `rand` and `randn` to the same fixed values each time you call `frnd`. For an example of how to do this, and a list of the Statistics Toolbox functions that depend on `rand` or `randn`, see “Reproducing the Output of Random Number Functions” on page 2-46.

---

**Note** The results in the following examples depend on the current states of `rand` and `randn`. If you run the code in these examples, your results may differ from the answers shown here.

---

**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**

fcdf, finv, fpdf, fstat

# fstat

---

**Purpose** Mean and variance for the F distribution

**Syntax** `[M,V] = fstat(V1,V2)`

**Description** `[M,V] = fstat(V1,V2)` returns the mean 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

$$\frac{2v_2^2(v_1+v_2-2)}{v_1(v_2-2)^2(v_2-4)}$$

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** `fcdf`, `finv`, `frnd`, `frnd`

**Purpose** Interactive contour plot of a function

**Syntax** `fsurfht('fun',xlims,ylim)`  
`fsurfht('fun',xlims,ylim,p1,p2,p3,p4,p5)`

**Description** `fsurfht('fun',xlims,ylim)` 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 `ylim` in the form `[ymin ymax]`.

`fsurfht('fun',xlims,ylim,p1,p2,p3,p4,p5)` allows for five optional parameters that you can supply to the function `fun`.

The intersection of the vertical and horizontal reference lines on the plot defines the current  $x$ -value and  $y$ -value. You can drag these reference lines and watch the calculated  $z$ -values (at the top of the plot) update simultaneously. Alternatively, you can type the  $x$ -value and  $y$ -value into editable text fields on the  $x$ -axis and  $y$ -axis.

**Example** Plot the Gaussian likelihood function for the `gas.mat` data.

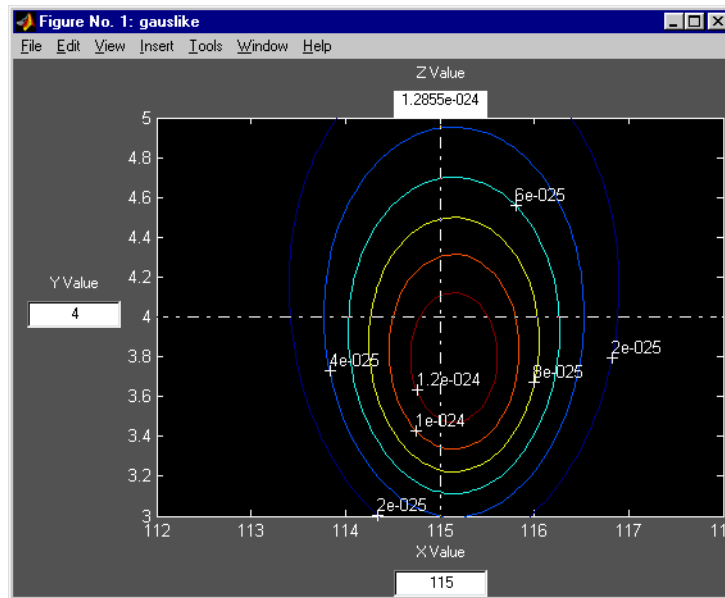
```
load gas
```

Create a function containing the following commands, and name it `gauslike.m`.

```
function z = gauslike(mu,sigma,p1)
n = length(p1);
z = ones(size(mu));
for i = 1:n
z = z .* (normpdf(p1(i),mu,sigma));
end
```

The `gauslike` function calls `normpdf`, treating the data sample as fixed and the parameters  $\mu$  and  $\sigma$  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
```

**Purpose** Full-factorial experimental design

**Syntax** `design = fullfact(levels)`

**Description** `design = fullfact(levels)` give the factor settings for a full factorial design. Each element in the vector `levels` specifies the number of unique values in the corresponding column of `design`.

For example, if the first element of `levels` is 3, then the first column of `design` contains only integers from 1 to 3.

**Example** If `levels = [2 4]`, `fullfact` generates an eight-run design with two levels in the first column and four in the second column.

```
d = fullfact([2 4])
```

```
d =
```

```
 1 1
 2 1
 1 2
 2 2
 1 3
 2 3
 1 4
 2 4
```

**See Also** `ff2n`, `dcovary`, `daugment`, `cordexch`

# gamcdf

---

**Purpose** Gamma cumulative distribution function (cdf)

**Syntax** `P = gamcdf(X,A,B)`  
`[P,PLO,PUP] = gamcdf(X,A,B,PCOV,ALPHA)`

**Description** `gamcdf(X,A,B)` computes the gamma cdf at each of the values in `X` using the corresponding parameters in `A` and `B`. `X`, `A`, and `B` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs. The parameters in `A` and `B` must be positive.

The gamma cdf is

$$p = F(x|a, b) = \frac{1}{b^a \Gamma(a)} \int_0^x t^{a-1} e^{-\frac{t}{b}} dt$$

The result, `p`, is the probability that a single observation from a gamma distribution with parameters `a` and `b` will fall in the interval `[0 x]`.

`[P,PLO,PUP] = gamcdf(X,A,B,PCOV,ALPHA)` produces confidence bounds for `P` when the input parameters `A` and `B` are estimates. `PCOV` is a 2-by-2 matrix containing the covariance matrix of the estimated parameters. `ALPHA` has a default value of 0.05, and specifies 100\*(1-ALPHA)% confidence bounds. `PLO` and `PUP` are arrays of the same size as `P` containing the lower and upper confidence bounds.

`gammainc` is the gamma distribution with `b` fixed at 1.

## Examples

```
a = 1:6;  
b = 5:10;  
prob = gamcdf(a.*b,a,b)  
  
prob =  
  
0.6321    0.5940    0.5768    0.5665    0.5595    0.5543
```

The mean of the gamma distribution is the product of the parameters, `ab`. In this example, the mean approaches the median as it increases (i.e., the distribution becomes more symmetric).



## See Also

[cdf](#), [gamfit](#), [gaminv](#), [gamlike](#), [gampdf](#), [gamrnd](#), [gamstat](#)

# gamfit

---

**Purpose** Parameter estimates and confidence intervals for gamma distributed data

**Syntax**

```
phat = gamfit(data)
[phat,pci] = gamfit(data)
[phat,pci] = gamfit(data,alpha)
[...] = gamfit(data,alpha,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 non-negative 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. You can create `options` using the function `statset`. Enter `statset('gamfit')` to see the names and default values of the parameters that `gamfit` accepts in the `options` structure.

**Example** Note that the 95% confidence intervals in the example below bracket the true parameter values of 2 and 4.

```
a = 2; b = 4;
data = gamrnd(a,b,100,1);
[p,ci] = gamfit(data)
```

```
p =
    2.1990    3.7426
```

```
ci =  
  1.6840    2.8298  
  2.7141    4.6554
```

**Reference**

[1] Hahn, G. J. and S.S. Shapiro. *Statistical Models in Engineering*. John Wiley & Sons, New York. 1994. p. 88.

**See Also**

gamcdf, gaminv, gamlike, gampdf, gamrnd, gamstat, mle, statset

**Purpose** Inverse of the gamma cumulative distribution function (cdf)

**Syntax**  
`X = gaminv(P,A,B)`  
`[X,XLO,XUP] = gamcdf(P,A,B,PCOV,ALPHA)`

**Description** `X = gaminv(P,A,B)` computes the inverse of the gamma cdf with parameters A and B for the corresponding probabilities in P. P, A, and B can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs. The parameters in A and B must all be positive, and the values in P must lie on the interval [0 1].

The gamma inverse function in terms of the gamma cdf is

$$x = F^{-1}(p|a,b) = \{x:F(x|a,b) = p\}$$

where

$$p = F(x|a,b) = \frac{1}{b^a \Gamma(a)} \int_0^x t^{a-1} e^{-\frac{t}{b}} dt$$

`[X,XLO,XUP] = gamcdf(P,A,B,PCOV,ALPHA)` produces confidence bounds for P when the input parameters A and B are estimates. PCOV is a 2-by-2 matrix containing the covariance matrix of the estimated parameters. ALPHA has a default value of 0.05, and specifies 100\*(1-ALPHA)% confidence bounds. PLO and PUP are arrays of the same size as P containing the lower and upper confidence bounds.

**Algorithm** There is no known analytical solution to the integral equation above. `gaminv` uses an iterative approach (Newton's method) to converge on the solution.

**Examples** This example shows the relationship between the gamma cdf and its inverse function.

```
a = 1:5;  
b = 6:10;  
x = gaminv(gamcdf(1:5,a,b),a,b)  
  
x =  
    1.0000    2.0000    3.0000    4.0000    5.0000
```

**See Also**

`gamcdf`, `gamfit`, `gamlike`, `gampdf`, `gamrnd`, `gamstat`, `icdf`

# gamlike

---

**Purpose** Negative gamma log-likelihood function

**Syntax** `logL = gamlike(params,data)`  
`[logL,AVAR] = gamlike(params,data)`

**Description** `logL = gamlike(params,data)` returns the negative of the gamma log-likelihood function for the parameters, `params`, given data. The length of output vector `logL` is the length of vector `data`.

`[logL,AVAR] = gamlike(params,data)` also returns `AVAR`, which is the asymptotic variance-covariance matrix of the parameter estimates when the values in `params` are the maximum likelihood estimates. `AVAR` is the inverse of Fisher's information matrix. The diagonal elements of `AVAR` are the asymptotic variances of their respective parameters.

`[...] = gamlike(params,data,censoring)` accepts a boolean vector of the same size as `data` that is 1 for observations that are right-censored and 0 for observations that are observed exactly.

`[...] = gamfit(params,data,censoring,freq)` accepts a frequency vector of the same size as `data`. `freq` typically contains integer frequencies for the corresponding elements in `data`, but may contain any non-negative values.

`gamlike` is a utility function for maximum likelihood estimation of the gamma distribution. Since `gamlike` returns the negative gamma log-likelihood function, minimizing `gamlike` using `fminsearch` is the same as maximizing the likelihood.

**Example** This example continues the example for `gamfit`.

```
a = 2; b = 3;
r = gamrnd(a,b,100,1);
[logL,info] = gamlike(gamfit(r),r)

logL =

    267.5648

info =

    0.0788   -0.1104
   -0.1104    0.1955
```

**See Also**

betalike, gamcdf, gamfit, gaminv, gampdf, gamrnd, gamstat, mle, normlike, wbllike

# gampdf

---

**Purpose** Gamma probability density function (pdf)

**Syntax** `Y = gampdf(X,A,B)`

**Description** `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  $[0 \infty)$ .

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  $\chi^2$  functions are special cases of the gamma function.

**Examples** The exponential distribution is a special case of the gamma distribution.

```
mu = 1:5;
y = gampdf(1,1,mu)

y =
    0.3679    0.3033    0.2388    0.1947    0.1637

y1 = exppdf(1,mu)

y1 =
    0.3679    0.3033    0.2388    0.1947    0.1637
```

**See Also** `gamcdf`, `gamfit`, `gaminv`, `gamlike`, `gamrnd`, `gamstat`, `pdf`



**Purpose** Random numbers from the gamma distribution

**Syntax**

```
R = gamrnd(A,B)
R = gamrnd(A,B,v)
R = gamrnd(A,B,m,n)
```

**Description** `R = gamrnd(A,B)` generates random numbers from the gamma distribution with parameters `A` and `B`. `A` and `B` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input for `A` or `B` is expanded to a constant array with the same dimensions as the other input.

`R = gamrnd(A,B,v)` generates random numbers from the gamma distribution with parameters `A` and `B`, where `v` is a row vector. If `v` is a 1-by-2 vector, `R` is a matrix with `v(1)` rows and `v(2)` columns. If `v` is 1-by-`n`, `R` is an `n`-dimensional array.

`R = gamrnd(A,B,m,n)` generates gamma random numbers with parameters `A` and `B`, where scalars `m` and `n` are the row and column dimensions of `R`.

### Reproducing the Output of gamrnd

`gamrnd` uses the MATLAB function `rand` to generate random numbers. When you call `gamrnd`, you change the current state of `rand`, and thereby alter the output of subsequent calls to `gamrnd` or any other functions that depend on `rand`. If you want to reproduce the output of `gamrnd`, reset the state of `rand` to the same fixed value each time you call `gamrnd`. For an example of how to do this, and a list of the Statistics Toolbox functions that depend on `rand`, see “Reproducing the Output of Random Number Functions” on page 2-46.

---

**Note** The results in the following examples depend on the current state of `rand`. If you run the code in these examples, your results may differ from the answers shown here.

---

### Examples

```
n1 = gamrnd(1:5,6:10)
n1 =
    9.1132    12.8431    24.8025    38.5960   106.4164
```

# gamrnd

---

```
n2 = gamrnd(5,10,[1 5])
```

```
n2 =  
    30.9486    33.5667    33.6837    55.2014    46.8265
```

```
n3 = gamrnd(2:6,3,1,5)
```

```
n3 =  
    12.8715    11.3068     3.0982    15.6012    21.6739
```

## See Also

gamcdf, gamfit, gaminv, gamlike, gampdf, gamstat

**Purpose** Mean and variance for the gamma distribution

**Syntax** `[M,V] = gamstat(A,B)`

**Description** `[M,V] = gamstat(A,B)` returns the mean and variance for the gamma distribution with parameters specified by A and B. A and B can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of M and V. A scalar input for A or B is expanded to a constant array with the same dimensions as the other input.

The mean of the gamma distribution with parameters  $a$  and  $b$  is  $ab$ . The variance is  $ab^2$ .

**Examples**

```
[m,v] = gamstat(1:5,1:5)
```

```
m =  
    1     4     9    16    25
```

```
v =  
    1     8    27    64   125
```

```
[m,v] = gamstat(1:5,1./(1:5))
```

```
m =  
    1     1     1     1     1
```

```
v =  
    1.0000    0.5000    0.3333    0.2500    0.2000
```

**See Also** `gamcdf`, `gamfit`, `gaminv`, `gamlike`, `gampdf`, `gamrnd`

# geocdf

---

**Purpose** Geometric cumulative distribution function (cdf)

**Syntax** `Y = geocdf(X,P)`

**Description** `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

$$y = F(x|p) = \sum_{i=0}^{\text{floor}(x)} pq^i$$

where  $q = 1 - p$ .

The result,  $y$ , is the probability of observing up to  $x$  trials before a success, when the probability of success in any given trial is  $p$ .

**Examples** Suppose you toss a fair coin repeatedly. If the coin lands face up (heads), that is a success. What is the probability of observing three or fewer tails before getting a heads?

```
p = geocdf(3,0.5)
```

```
p =  
0.9375
```

**See Also** `cdf`, `geoinv`, `geopdf`, `geornd`, `geostat`

**Purpose** Inverse of the geometric cumulative distribution function (cdf)

**Syntax** `X = geoinv(Y,P)`

**Description** `X = geoinv(Y,P)` returns the smallest positive integer `X` such that the geometric cdf evaluated at `X` is equal to or exceeds `Y`. You can think of `Y` as the probability of observing `X` successes in a row in independent trials where `P` is the probability of success in each trial.

`Y` and `P` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input for `P` or `Y` is expanded to a constant array with the same dimensions as the other input. The values in `P` and `Y` must lie on the interval `[0 1]`.

**Examples** The probability of correctly guessing the result of 10 coin tosses in a row is less than 0.001 (unless the coin is not fair).

```
psychic = geoinv(0.999,0.5)
```

```
psychic =
```

```
9
```

The example below shows the inverse method for generating random numbers from the geometric distribution.

```
rndgeo = geoinv(rand(2,5),0.5)
```

```
rndgeo =
```

```
0    1    3    1    0
0    1    0    2    0
```

**See Also** `geocdf`, `geopdf`, `geornd`, `geostat`, `icdf`

# geomean

---

**Purpose** Geometric mean of a sample

**Syntax** `m = geomean(X)`  
`geomean(X, dim)`

**Description** `geomean` 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 sample average is greater than or equal to the geometric mean.

```
x = exprnd(1,10,6);
geometric = geomean(x)

geometric =

    0.7466    0.6061    0.6038    0.2569    0.7539    0.3478

average = mean(x)

average =

    1.3509    1.1583    0.9741    0.5319    1.0088    0.8122
```

**See Also** `mean`, `median`, `harmmean`, `trimmean`

**Purpose** Geometric probability density function (pdf)

**Syntax**  $Y = \text{geopdf}(X, P)$

**Description**  $\text{geopdf}(X, P)$  computes the geometric pdf at each of the values in  $X$  using the corresponding probabilities in  $P$ .  $X$  and  $P$  can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other input. The parameters in  $P$  must lie on the interval  $[0, 1]$ .

The geometric pdf is

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

where  $q = 1 - p$ .

**Examples** Suppose you toss a fair coin repeatedly. If the coin lands face up (heads), that is a success. What is the probability of observing exactly three tails before getting a heads?

```
p = geopdf(3, 0.5)
```

```
p =  
    0.0625
```

**See Also** `geocdf`, `geoinv`, `geornd`, `geostat`, `pdf`

# geornd

---

**Purpose** Random numbers from the geometric distribution

**Syntax**

```
R = geornd(P)
R = geornd(P,v)
R = geornd(P,m,n)
```

**Description** The geometric distribution is useful when you want to model the number of successive failures preceding a success, where the probability of success in any given trial is the constant  $P$ .

$R = \text{geornd}(P)$  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  $P$ .

$R = \text{geornd}(P,v)$  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.

$R = \text{geornd}(P,m,n)$  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  $[0\ 1]$ .

## Reproducing the Output of geornd

`geornd` uses the MATLAB function `rand` to generate random numbers. When you call `geornd`, you change the current state of `rand`, and thereby alter the output of subsequent calls to `geornd` or any other functions that depend on `rand`. If you want to reproduce the output of `geornd`, reset the state of `rand` to the same fixed value each time you call `geornd`. For an example of how to do this, and a list of the Statistics Toolbox functions that depend on `rand`, see “Reproducing the Output of Random Number Functions” on page 2-46.

---

**Note** The results in the following examples depend on the current state of `rand`. If you run the code in these examples, your results may differ from the answers shown here.

---

**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**

geocdf, geoinv, geopdf, geostat

# geostat

---

**Purpose** Mean and variance for the geometric distribution

**Syntax** [M,V] = geostat(P)

**Description** [M,V] = geostat(P) returns the mean 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**

geocdf, geoinv, geopdf, geornd

**Purpose** Interactively draw a line in a figure

**Syntax**  
`gline(fig)`  
`h = gline(fig)`  
`gline`

**Description** `gline(fig)` allows you to draw a line segment in the figure `fig` by clicking the pointer at the two end-points. A rubber band line tracks the pointer movement.

`h = gline(fig)` returns the handle to the line in `h`.

`gline` with no input arguments draws in the current figure.

**See Also** `refline`, `gname`

# glmdemo

---

**Purpose** Demo of generalized linear models

**Syntax** glmdemo

**Description** glmdemo begins a slide show demonstration of generalized linear models. The slides indicate when generalized linear models are useful, how to fit generalized linear models using the `glmfit` function, and how to make predictions using the `glmval` function.

---

**Note** To run this demo from the command line, type `playshow glmdemo`.

---

**See Also** `glmfit`, `glmval`

**Purpose** Generalized linear model fitting

**Syntax**

```
b = glmfit(DATA,Y,'distr')
b = glmfit(DATA,Y,'distr','link','estdisp',offset,pwts,'const')
[b,dev,stats] = glmfit(...)
```

**Description** `b = glmfit(DATA,Y,'distr')` fits the generalized linear model for response  $Y$ , predictor variable matrix  $DATA$ , and distribution ' $distr$ '. The following distributions are available: 'binomial', 'gamma', 'inverse gaussian', 'lognormal', 'normal' (the default), and 'poisson'. In most cases  $Y$  is a vector of response measurements, but for the binomial distribution  $Y$  is a two-column array having the measured number of counts in the first column and the number of trials (the binomial  $N$  parameter) in the second column.  $DATA$  is a matrix having the same number of rows as  $Y$  and containing the values of the predictor variables for each observation. The output  $b$  is a vector of coefficient estimates. This syntax uses the canonical link (see below) to relate the distribution parameter to the predictors.

`b = glmfit(x,y,'distr','link','estdisp',offset,pwts,'const')` provides additional control over the fit. The ' $link$ ' argument specifies the relationship between the distribution parameter ( $\mu$ ) and the fitted linear combination of predictor variables ( $xb$ ). In most cases ' $link$ ' is one of the following:

<b>'link'</b>	<b>Meaning</b>	<b>Default (Canonical) Link</b>
'identity'	$\mu = xb$	'normal'
'log'	$\log(\mu) = xb$	'poisson'
'logit'	$\log(\mu / (1-\mu)) = xb$	'binomial'
'probit'	$\text{norminv}(\mu) = xb$	
'comploglog'	$\log(-\log(1-\mu)) = xb$	
'logloglink'	$\log(-\log(\mu)) = xb$	
'reciprocal'	$1/\mu = xb$	'gamma'
$p$ (a number)	$\mu^p = xb$	'inverse gaussian' (with $p=-2$ )

Alternatively, you can write functions to define your own custom link. You specify the link argument as a three-element cell array containing functions that define the link function, its derivative, and its inverse. For example, suppose you want to define a reciprocal square root link using inline functions. You could define the variable `mylinks` to use as your '`link`' argument by writing:

```
FL = inline('x.^-.5')
FD = inline('-.5*x.^-1.5')
FI = inline('x.^-2')
mylinks = {FL FI FD}
```

Alternatively, you could define functions named `FL`, `FD`, and `FI` in their own M-files, and then specify `mylinks` in the form

```
mylinks = {@FL @FD @FI}
```

The '`estdisp`' argument can be '`on`' to estimate a dispersion parameter for the binomial or Poisson distribution, or '`off`' (the default) to use the theoretical value of 1.0 for those distributions. The `glmfit` function always estimates dispersion parameters for other distributions.

The `offset` and `pwts` parameters can be vectors of the same length as `Y`, or can be omitted (or specified as an empty vector). The `offset` vector is a special predictor variable whose coefficient is known to be 1.0. As an example, suppose that you are modeling the number of defects on various surfaces, and you want to construct a model in which the expected number of defects is proportional to the surface area. You might use the number of defects as your response, along with the Poisson distribution, the log link function, and the log surface area as an `offset`.

The `pwts` argument is a vector of prior weights. As an example, if the response value  $Y(i)$  is the average of  $f(i)$  measurements, you could use `f` as a vector of prior weights.

The '`const`' argument can be '`on`' (the default) to estimate a constant term, or '`off`' to omit the constant term. If you want the constant term, use this argument rather than specifying a column of ones in the `DATA` matrix.

`[b,dev,stats] = glmfit(...)` returns the additional outputs `dev` and `stats`. `dev` is the deviance 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.

stats is a structure with the following fields:

- stats.dfe = degrees of freedom for error
- stats.s = theoretical or estimated dispersion parameter
- stats.sfit = estimated dispersion parameter
- stats.estimatedisp = 1 if dispersion is estimated, 0 if fixed
- stats.beta = vector of coefficient estimates (same as b)
- stats.se = vector of standard errors of the coefficient estimates b
- stats.coeffcorr = correlation matrix for b
- stats.t = t statistics for b
- stats.p = p-values for b
- stats.resid = vector of residuals
- stats.residp = vector of Pearson residuals
- stats.residd = vector of deviance residuals
- stats.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

You have data on cars weighing between 2100 and 4300 pounds. For each car weight you have the total number of cars of that weight, and the number that can be considered to get “poor mileage” according to some test. For example, 8 out of 21 cars weighing 3100 pounds get poor mileage according to a measurement of the miles they can travel on a gallon of gasoline.

```
w = (2100:200:4300)';
poor = [1 2 0 3 8 8 14 17 19 15 17 21]';
total = [48 42 31 34 31 21 23 23 21 16 17 21]';
```

You can compare several fits to these data. First, try fitting logit and probit models:

```
[b1,d1,s1] = glmfit(w,[poor total],'binomial');
[bp,dp,sp] = glmfit(w,[poor total],'binomial','probit');
```

```
d1
d1 =
    6.4842

dp
dp =
    7.5693
```

The deviance for the logit model is smaller than for the probit model. Although this is not a formal test, it suggests that the logit model is preferable.

You can do a formal test comparing two logit models. You already fit one model using  $w$  as a linear predictor, so fit another logit model using both linear and squared terms in  $w$ . If there is no true effect for the squared term, the difference in their deviances should be small compared with a chi-square distribution having one degree of freedom.

```
[b2,d2,s2] = glmfit([w w.^2],[poor total],'binomial');

d1-d2

ans =
    0.7027

chi2cdf(d1-d2,1)

ans =
    0.5981
```

A difference of 0.7072 is not at all unusual for a chi-square distribution with one degree of freedom, so the quadratic model does not give a significantly better fit than the simpler linear model.

The following are the coefficient estimates, their standard errors, t-statistics, and p-values for the linear model:

```
[b1 s1.se s1.t s1.p]

ans =

   -13.3801    1.3940   -9.5986    0.0000
    0.0042    0.0004    9.4474    0.0000
```



This shows that you cannot simplify the model any further. Both the intercept and slope coefficients are significantly different from 0, as indicated by p-values that are 0.0000 to four decimal places.

**See Also**

glmval, glmdemo, nlinfit, regress, regstats

**References**

- [1] Dobson, A. J. *An Introduction to Generalized Linear Models*. 1990, CRC Press.
- [2] McCullagh, P. and J. A. Nelder. *Generalized Linear Models*. 2nd edition, 1990, Chapman and Hall.

# glmval

---

**Purpose** Compute predictions for generalized linear model

**Syntax**

```
yfit = glmval(b,X,'link')  
[yfit,dlo,dhi] = glmval(b,X,'link',stats,clev)  
[yfit,dlo,dhi] = glmval(b,X,'link',stats,clev,N,offset,'const')
```

**Description** `yfit = glmval(b,X,'link')` computes the predicted distribution parameters for observations with predictor values `X` using the coefficient vector `b` and link function `'link'`. Typically, `b` is a vector of coefficient estimates computed by the `glmfit` function. The value of `'link'` must be the same as that used in `glmfit`. The result `yfit` is the value of the inverse of the link function at the linear combination  $X*b$ .

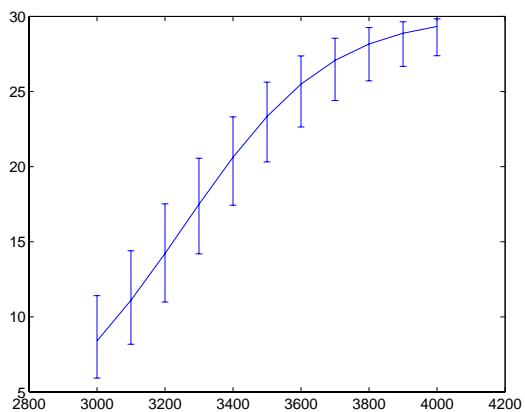
`[yfit,dlo,dhi] = glmval(b,X,'link',stats,clev)` returns confidence bounds for the predicted values when you supply the `stats` structure returned from `glmfit`, and optionally specify a confidence level as the `clev` argument. (The default confidence level is 0.95 for 95% confidence.) The interval `[yfit-dlo, yfit+dhi]` is a confidence bound for the true parameter value at the specified `X` values.

`[yhat,dlo,dhi] = glmval(beta,X,'link',stats,clev,N,offset,'const')` specifies three additional arguments that may be needed if you used certain arguments to `glmfit`. If you fit a binomial distribution using `glmfit`, specify `N` as the value of the binomial `N` parameter for the predictions. If you included an offset variable, specify `offset` as the new value of this variable. Use the same `'const'` value (`'on'` or `'off'`) that you used with `glmfit`.

**Example** Let's model the number of cars with poor gasoline mileage using the binomial distribution. First, use the binomial distribution with the default logit link to model the probability of having poor mileage as a function of the weight and squared weight of the cars. Then you compute a vector `wnew` of new car weights at which you want to make predictions. Next you compute the expected number of cars, out of a total of 30 cars of each weight, that would have poor mileage. Finally you graph the predicted values and 95% confidence bounds as a function of weight.

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

```
[b2,d2,s2] = glmfit([w w.^2],[poor total],'binomial')  
wnew = (3000:100:4000)';  
[yfit,dlo,dhi] = glmval(b2,[wnew wnew.^2],'logit',s2,0.95,30)  
errorbar(wnew,yfit,dlo,dhi);
```

**See Also**

glmfit, glmdemo

# glyphplot

---

**Purpose** Plot stars or Chernoff faces for multivariate data

**Syntax**

```
glyphplot(X)
glyphplot(X, 'Glyph','face')
glyphplot(X, 'Glyph','face', 'Features', F)
glyphplot(X, ..., 'Grid', [ROWS,COLS])
glyphplot(X, ..., 'Grid', [ROWS,COLS], 'Page', page)
glyphplot, ..., 'Centers', C, 'Radius', r)
glyphplot(X, ..., 'ObsLabels', labels)
glyphplot(X, ..., 'Standardize', method)
glyphplot(X, ..., 'PropertyName', PropertyValue, ...)
h = glyphplot(X, ...)
```

**Description** `glyphplot(X)` creates a star plot from the multivariate data in the  $n$ -by- $p$  matrix  $X$ . Rows of  $X$  correspond to observations, columns to variables. A star plot represents each observation as a "star" whose  $i$ -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 12-211.

`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 zeros indicate that the corresponding column of  $X$  should not be plotted. See "Face Features" on page 12-211 for more information.

`glyphplot(X, ..., 'Grid', [rows, cols])` organizes the glyphs into a rows-by-cols grid.

`glyphplot(X, ..., 'Grid', [rows, cols], 'Page', page)` organizes the glyph into one or more pages of a rows-by-cols grid, and displays the page'th page. If `page` is a vector, `glyphplot` displays multiple pages in succession. If

page is 'all', glyphplot displays all pages. If page is 'scroll', glyphplot displays a single plot with a scrollbar.

glyphplot(X, ..., 'Centers', C) creates a plot with each glyph centered at the locations in the N-by-2 matrix C.

glyphplot, ..., 'Centers', C, 'Radius', r) creates a plot with glyphs positioned using C, and scale the glyphs so the largest has radius r.

glyphplot(X, ..., 'ObsLabels', labels) labels each glyph with the text in the character array or cell array of strings labels. By default, the glyphs are labelled 1:N. Pass in '' for no labels.

glyphplot(X, ..., 'Standardize', method) standardizes X before making the plot. Choices for method are

- 'column' — Maps each column of X separately onto the interval [0,1]. This is the default.
- 'matrix' — Maps the entire matrix X onto the interval [0,1].
- 'PCA' — Transforms X to its principal component scores, in order of decreasing eigenvalue, and maps each one onto the interval [0,1].
- 'off' — No standardization. Negative values in X may make a star plot uninterpretable.

glyphplot(X, ..., 'PropertyName', PropertyValue, ...) sets properties to the specified property values for all line graphics objects created by glyphplot.

h = glyphplot(X, ...) returns a matrix of handles to the graphics objects created by glyphplot. For a star plot, h(:,1) and h(:,2) contain handles to the line objects for each star's perimeter and spokes, respectively. For a face plot, h(:,1) and h(:,2) contain object handles to the lines making up each face and to the pupils, respectively. h(:,3) contains handles to the text objects for the labels, if present.

## Face Features

The following table describes the correspondence between the columns of the vector F, the value of the 'Features' input parameter, and the facial features

of the glyph plot. If X has fewer than 17 columns, unused features are displayed at their default value.

Column	Facial Feature
1	Size of face
2	Forehead/jaw relative arc length
3	Shape of forehead
4	Shape of jaw
5	Width between eyes
6	Vertical position of eyes
7	Height of eyes
8	Width of eyes (this also affects eyebrow width)
9	Angle of eyes (this also affects eyebrow angle)
10	Vertical position of eyebrows
11	Width of eyebrows (relative to eyes)
12	Angle of eyebrows (relative to eyes)
13	Direction of pupils
14	Length of nose
15	Vertical position of mouth
16	Shape of mouth
17	Mouth arc length

## Examples

```
load carsmall
X = [Acceleration Displacement Horsepower MPG Weight];
glyphplot(X, 'Standardize', 'column', 'ObsLabels', Model, ...
'grid', [2 2], 'page', 'scroll');
```

```
glyphplot(X, 'Glyph', 'face', 'ObsLabels', Model, 'grid', [2 3],  
          'page', 9);
```

**See Also**

andrewsplot, parallelcoords

# gname

---

**Purpose** Label plotted points with their case names or case number

**Syntax**

```
gname(cases)
gname
h = gname(cases,line_handle)
```

**Description** `gname(cases)` displays a figure window and waits for you to press a mouse button or a keyboard key. The input argument `cases` is a character array or a cell array of strings, in which each row of the character array or each element of the cell array contains the case name of a point. Moving the mouse over the graph displays a pair of cross-hairs. If you position the cross-hairs near a point with the mouse and click once, the graph displays the name of the city corresponding to that point. Alternatively, you can click and drag the mouse to create a rectangle around several points. When you release the mouse button, the graph displays the labels for all points in the rectangle. Right-click a point to remove its label. When you are done labelling points, press the **Enter** or **Escape** key to stop labeling.

`gname` with no arguments labels each case with its case number.

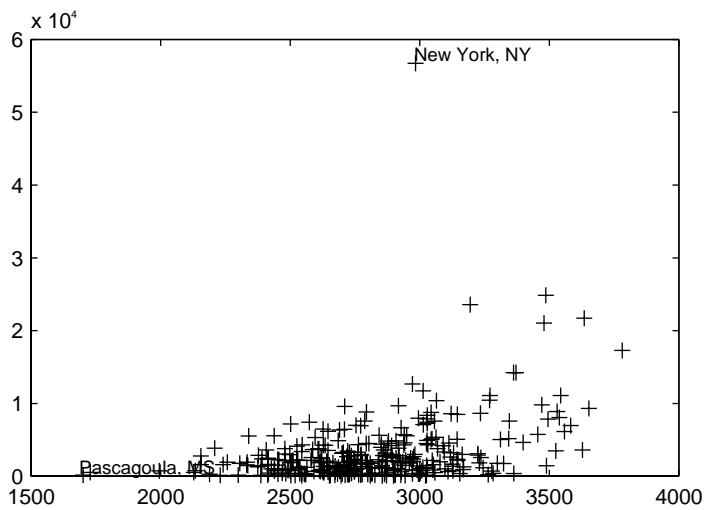
`h = gname(cases,line_handle)` returns a vector of handles to the text objects on the plot. Use the scalar `line_handle` to identify the correct line if there is more than one line object on the plot.

You can use `gname` to label plots created by the `plot`, `scatter`, `gscatter`, `plotmatrix`, and `gplotmatrix` functions.

**Example** This example uses the city ratings data sets to find out which cities are the best and worst for education and the arts.

```
load cities
education = ratings(:,6);
arts = ratings(:,7);
plot(education,arts,'+')
gname(names)
```





Click the point at the top of the graph to display its label, “New York.”

### See Also

`gplotmatrix`, `gscatter`, `gtext`, `plot`, `plotmatrix`, `scatter`

# gplotmatrix

---

**Purpose** Plot matrix of scatter plots by group

**Syntax**

```
gplotmatrix(x,y,g)
gplotmatrix(x,y,g,'clr','sym',siz)
gplotmatrix(x,y,g,'clr','sym',siz,'doleg')
gplotmatrix(x,y,g,'clr','sym',siz,'doleg','dispopt')
gplotmatrix(x,y,g,'clr','sym',siz,'doleg','dispopt','xnam','ynam')
[h,ax,bigax] = gplotmatrix(...)
```

**Description** `gplotmatrix(x,y,g)` 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 `g`.

`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.

`g` is a grouping variable that can be a vector, string array, or cell array of strings. `g` must have the same number of rows as `x` and `y`. Points with the same value of `g` are placed in the same group, and appear on the graph with the same marker and color. Alternatively, `g` 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,g,'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 is `'clr' = 'bgcmyk'`. `'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,g,'clr','sym',siz,'doleg')` controls whether a legend is displayed on the graph (`'doleg' = 'on'`, the default) or not (`'doleg' = 'off'`).

`gplotmatrix(x,y,g,'clr','sym',siz,'doleg','dispopt')` controls what appears along the diagonal of a plot matrix of `x` 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,g,'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 with one row 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. `ax` is a matrix of handles to the axes of the individual plots. `bigax` is a handle to big (invisible) axes framing the entire plot matrix. These are left as the current axes, so a subsequent `title`, `xlabel`, or `ylabel` command will produce labels that are centered with respect to the entire plot matrix.

## Example

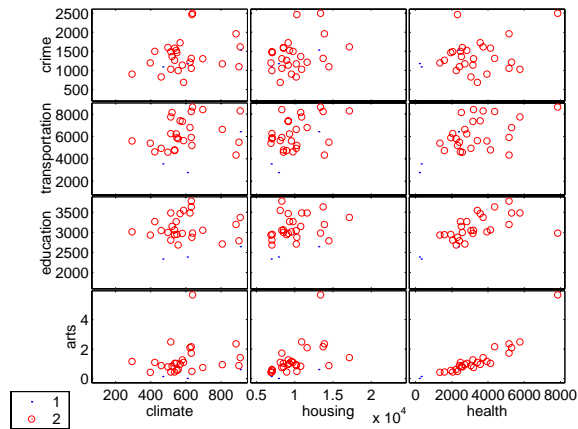
Load the cities data. The ratings array has ratings of the cities in nine categories (category names are in the array `categories`). `group` is a code whose value is 2 for the largest cities. You can make scatter plots of the first three categories against the other four, grouped by the city size code.

```
load discrim
gplotmatrix(ratings(:,1:3),ratings(:,4:7),group)
```

The output figure (not shown) has an array of graphs with each city group represented by a different color. The graphs are a little easier to read if you specify colors and plotting symbols, label the axes with the rating categories, and move the legend off the graphs.

```
gplotmatrix(ratings(:,1:3),ratings(:,4:7),group,...
            'br','o',[],'on','','categories(1:3,:),...
            categories(4:7,:))
```

# gplotmatrix



## See Also

`grpstats`, `gscatter`, `plotmatrix`

**Purpose** Summary statistics by group

**Syntax** `means = grpstats(X,group)`  
`[means,sem,counts,name] = grpstats(X,group)`  
`grpstats(x,group,alpha)`

**Description** `means = grpstats(X,group)` returns the means of each column of `X` by group, where `X` is a matrix of observations. `group` is an array that defines the grouping such that two elements of `X` are in the same group if their corresponding group values are the same. The grouping variable `group` can be a vector, string array, or cell array of strings. It can also 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.

`[means,sem,counts,name] = grpstats(x,group,alpha)` supplies the standard error of the mean in `sem`, the number of elements in each group in `counts`, and the name of each group in `name`. `name` is useful to identify and label the groups when the input group values are not simple group numbers.

`grpstats(x,group,alpha)` plots  $100(1 - \alpha)\%$  confidence intervals around each mean.

**Example** Assign 100 observations to one of four groups. For each observation, you measure five quantities with *true means* from 1 to 5. `grpstats` enables you to compute the means for each group.

```
group = unidrnd(4,100,1);
true_mean = 1:5;
true_mean = true_mean(ones(100,1),:);
x = normrnd(true_mean,1);
means = grpstats(x,group)

means =

    0.7947    2.0908    2.8969    3.6749    4.6555
    0.9377    1.7600    3.0285    3.9484    4.8169
    1.0549    2.0255    2.8793    4.0799    5.3740
    0.7107    1.9264    2.8232    3.8815    4.9689
```

**See Also** `tabulate`, `crosstab`

# gscatter

---

**Purpose** Scatter plot by group

**Syntax**

```
gscatter(x,y,g)
gscatter(x,y,g,'clr','sym',siz)
gscatter(x,y,g,'clr','sym',siz,'doleg')
gscatter(x,y,g,'clr','sym',siz,'doleg','xnam','ynam')
h = gscatter(...)
```

**Description** `gscatter(x,y,g)` creates a scatter plot of `x` and `y`, grouped by `g`, where `x` and `y` are vectors with the same size and `g` can be a vector, string array, or cell array of strings. Points with the same value of `g` are placed in the same group, and appear on the graph with the same marker and color. Alternatively, `g` 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.

`gscatter(x,y,g,'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 is `'clr' = 'bgrcmk'`. `'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,g,'clr','sym',siz,'doleg')` controls whether a legend is displayed on the graph (`'doleg' = 'on'`, the default) or not (`'doleg' = 'off'`).

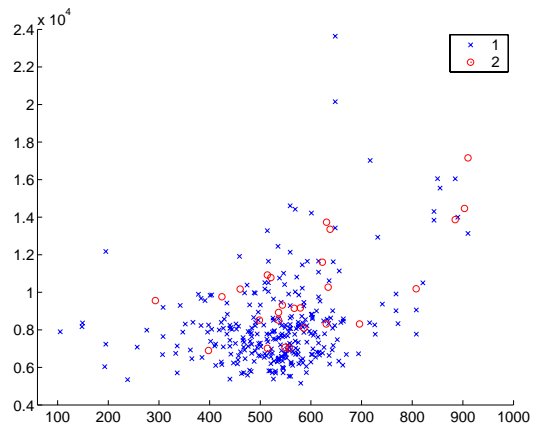
`gscatter(x,y,g,'clr','sym',siz,'doleg','xnam','ynam')` specifies the name to use for the `x`-axis and `y`-axis labels. If the `x` and `y` inputs are simple variable names and `xnam` and `ynam` are omitted, `gscatter` labels the axes with the variable names.

`h = gscatter(...)` returns an array of handles to the lines on the graph.

**Example** Load the `cities` data and look at the relationship between the ratings for climate (first column) and housing (second column) grouped by city size. We'll also specify the colors and plotting symbols.

```
load discrim
```

```
gscatter(ratings(:,1),ratings(:,2),group,'br','xo')
```

**See Also**

gplotmatrix, grpstats, scatter

# harmmean

---

**Purpose** Harmonic mean of a sample of data

**Syntax** `m = harmmean(X)`  
`harmmean(X,dim)`

**Description** `m = harmmean(X)` calculates the harmonic mean of a sample. For vectors, `harmmean(x)` is the harmonic mean of the elements in `x`. For matrices, `harmmean(X)` is a row vector containing the harmonic means of each column. For N-dimensional arrays, `harmmean` operates along the first nonsingleton dimension of `X`.

`harmmean(X,dim)` takes the harmonic mean along dimension `dim` of `X`.

The harmonic mean is

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

**Examples** The sample average 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
```

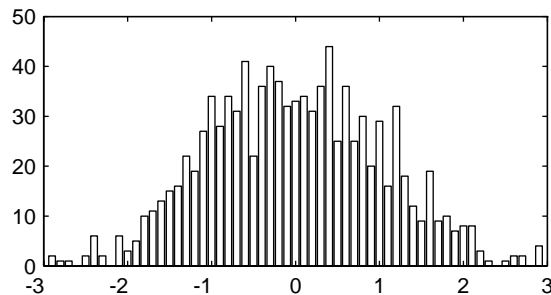
**See Also** `mean`, `median`, `geomean`, `trimmean`



<b>Purpose</b>	Plot histograms
<b>Syntax</b>	<code>hist(y)</code> <code>hist(y,nb)</code> <code>hist(y,x)</code> <code>[n,x] = hist(y,...)</code>
<b>Description</b>	<p><code>hist(y)</code> draws a 10-bin histogram for the data in vector <code>y</code>. The bins are equally spaced between the minimum and maximum values in <code>y</code>.</p> <p><code>hist(y,nb)</code> draws a histogram with <code>nb</code> bins.</p> <p><code>hist(y,x)</code> draws a histogram using the bins in the vector <code>x</code>.</p> <p><code>[n,x] = hist(y,...)</code> do not draw graphs, but return vectors <code>n</code> and <code>x</code> containing the frequency counts and the bin locations such that <code>bar(x,n)</code> plots the histogram. This is useful in situations where more control is needed over the appearance of a graph, for example, to combine a histogram into a more elaborate plot statement.</p> <p>The <code>hist</code> function is a part of the standard MATLAB language.</p>

**Examples** Generate bell-curve histograms from Gaussian data.

```
x = -2.9:0.1:2.9;  
y = normrnd(0,1,1000,1);  
hist(y,x)
```



**See Also** `hist3`, `histc`

# hist3

---

**Purpose** Three-dimensional histogram of bivariate data

**Syntax**

```
hist3(X)
hist3(X, nbins)
hist3(X, ctrs)
hist3(X, 'Edges', edges)
N = hist3(X,...)
[N, C] = hist3(X,...)
hist3(..., 'param1',val1, 'param2',val2, ...)
```

**Description** `hist3(X)` bins the elements of the  $m$ -by- $2$  matrix  $X$  into a 10-by-10 grid of equally-spaced containers, and plots a histogram. Each column of  $X$  corresponds to one dimension in the bin grid.

`hist3(X, nbins)` plots a histogram using an `nbins(1)`-by-`nbins(2)` grid of bins. `hist3(X, 'Nbins', nbins)` is equivalent to `hist3(X, nbins)`.

`hist3(X, ctrs)`, where `ctrs` is a two-element cell array of numeric vectors with monotonically non-decreasing values, uses a two-dimensional grid of bins centered on `ctrs{1}` in the first dimension and on `ctrs{2}` in the second. `hist3` assigns rows of  $X$  falling outside the range of that grid to the bins along the outer edges of the grid, and ignores rows of  $X$  containing 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 two-dimensional grid of bins with edges at `edges{1}` in the first dimension and at `edges{2}` in the second. The  $(i, j)$ -th bin includes the value  $X(k, :)$  if

$$\begin{aligned} \text{edges}\{1\}(i) &\leq X(k,1) < \text{edges}\{1\}(i+1) \\ \text{edges}\{2\}(j) &\leq X(k,2) < \text{edges}\{2\}(j+1) \end{aligned}$$

Rows of  $X$  that fall on the upper edges of the grid, `edges{1}(end)` or `edges{2}(end)`, are counted in the  $(I, j)$ -th or  $(i, J)$ -th bins, where  $I$  and  $J$  are the lengths of `edges{1}` and `edges{2}`. `hist3` does not count rows of  $X$  falling outside the range of the grid. Use `-Inf` and `Inf` in `edges` to include all non-NaN values.

`N = hist3(X, ...)` returns a matrix containing the number of elements of  $X$  that fall in each bin of the grid, and does not plot the histogram.

`[N, C] = hist3(X, ...)` returns the positions of the bin centers in a 1-by-2 cell array of numeric vectors, and does not plot the histogram. `hist3(ax, X, ...)` plots onto an axes with handle `ax` instead of the current axes. See the reference page for axes for more information about handles to plots.

`hist3(..., 'param1',val1, 'param2',val2, ...)` allows you to specify graphics parameter name/value pairs to fine-tune the plot.

### Example

```
% Create the car data and make a histogram on a 7x7 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`

# histfit

---

**Purpose** Histogram with superimposed normal density

**Syntax**

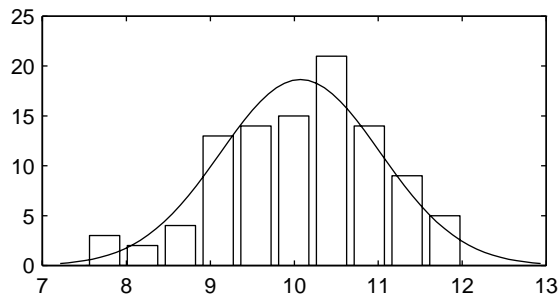
```
histfit(data)
histfit(data,nbins)
h = histfit(data,nbins)
```

**Description** `histfit(data,nbins)` plots a histogram of the values in the vector `data` using `nbins` bars in the histogram. With `nbins` is omitted, its value is set to the square root of the number of elements in `data`.

`h = histfit(data,nbins)` returns a vector of handles to the plotted lines, where `h(1)` is the handle to the histogram, `h(2)` is the handle to the density curve.

**Example**

```
r = normrnd(10,1,100,1);
histfit(r)
```



**See Also** `hist`, `hist3`, `normfit`

**Purpose** Calculate the posterior state probabilities of a sequence

**Syntax**

```
PSTATES = hmmdecode(seq, TRANS, EMIS)
[PSTATES, logpseq] = hmmdecode(seq, TRANS, EMIS)
[PSTATES, logpseq, FORWARD, BACKWARD, S] = hmmdecode(seq, TRANS, EMIS)
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)` element of `PSTATES` gives the probability that the model is in state `i` at the `j`th step, given the sequence `seq`.

---

**Note** The function `hmmdecode` begins with the model in state 1 at step 0, prior to the first emission. `hmmdecode` computes the probabilities in `PSTATES` based on the fact that the model begins in state 1. See “How the Toolbox Generates Random Sequences” on page 11-7 for more information

---

`[PSTATES, logpseq] = hmmdecode(seq, TRANS, EMIS)` returns `logpseq`, the logarithm of the probability of sequence `seq`, given transition matrix `TRANS` and emission matrix `EMIS`.

`[PSTATES, logpseq, FORWARD, BACKWARD, S] = hmmdecode(seq, TRANS, EMIS)` returns the forward and backward probabilities of the sequence scaled by `S`. See “Reference” on page 12-228 for a reference that explains the forward and backward probabilities.

`hmmdecode(..., 'Symbols', SYMBOLS)` specifies the symbols that are emitted. `SYMBOLS` can be a numeric array or a cell array of the names of the symbols. The default symbols are integers 1 through `N`, where `N` is the number of possible emissions.

# hmmdecode

---

See “Calculating Posterior State Probabilities” on page 11-12 for an example of using `hmmdecode`.

## Examples

```
trans = [0.95,0.05;
         0.10,0.90];

emis = [1/6, 1/6, 1/6, 1/6, 1/6, 1/6;
        1/10, 1/10, 1/10, 1/10, 1/10, 1/2];

[seq, states] = hmmgenerate(100,trans,emis);
pStates = hmmdecode(seq,tr,e);

[seq, states] = hmmgenerate(100, trans, emis, 'Symbols',...
{'one','two','three','four','five','six'})

pStates = hmmdecode(seq, tr, e, 'Symbols',...
{'one','two','three','four','five','six'});
```

## Reference

Durbin, R., S. Eddy, A. Krogh, and G. Mitchison, *Biological Sequence Analysis*, Cambridge Univ. Press, 1998.

## See Also

`hmmgenerate`, `hmmestimate`, `hmmviterbi`, `hmmtrain`

<b>Purpose</b>	Estimate the parameters for a hidden Markov model given state information
<b>Syntax</b>	<pre>[TRANS, EMIS] = hmmestimate(seq, states) hmmestimate(..., 'Symbols', SYMBOLS) hmmestimate(..., 'Statenames', STATENAMES) hmmestimate(..., 'Pseudoemissions', PSEUDOE) hmmestimate(..., 'Pseudotransitions', PSEUDOTR)</pre>
<b>Description</b>	<p>[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.</p> <p>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.</p> <p>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.</p> <p>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 <math>i \rightarrow k</math> 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.</p> <p>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 <math>i \rightarrow j</math> 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.</p>

# hmmestimate

---

See “Using hmmestimate” on page 11-9 for an example of using hmmestimate.

## Pseudotransitions and Pseudoemissions

If the probability of a specific transition or emission is very low, the transition might never occur in the sequence states, or the emission might never occur in the sequence seq. In either case, the algorithm returns a probability of 0 for the given transition or emission in TRANS or EMIS. You can compensate for the absence of transition with the 'Pseudotransitions' and 'Pseudoemissions' arguments. The simplest way to do this is to set the corresponding entry of PSEUDO or PSEUDOTR to 1. For example, if the transition  $i \rightarrow j$  does not occur in states, set  $PSEUOTR(i, j) = 1$ . This forces  $TRANS(i, j)$  to be positive. If you have an estimate for the expected number of transitions  $i \rightarrow j$  in a sequence of the same length as states, and the actual number of transitions  $i \rightarrow j$  that occur in seq is substantially less than what you expect, you can set  $PSEUOTR(i, j)$  to the expected number. This increases the value of  $TRANS(i, j)$ . For transitions that do occur in states with the frequency you expect, set the corresponding entry of PSEUDOTR to 0, which does not increase the corresponding entry of TRANS.

If you do not know the sequence of states, use hmmtrain to estimate the model parameters.

### Examples:

```
trans = [0.95,0.05; 0.10,0.90];
e = [1/6, 1/6, 1/6, 1/6, 1/6, 1/6;
1/10, 1/10, 1/10, 1/10, 1/10, 1/2];

[seq, states] = hmmgenerate(1000,trans,emis);

[estimateTR, estimateE] = hmmestimate(seq,states);
```

### See Also

hmmgenerate, hmmdecode, hmmviterbi, hmmtrain



**Purpose** Generate random sequences from a Markov model

**Syntax** `[seq, states] = hmmgenerate(len, TRANS, EMIS)`  
`hmmgenerate(..., 'Symbols', SYMBOLS)`  
`hmmgenerate(..., 'Statenames', STATENAMES)`

**Description** `[seq, states] = hmmgenerate(len, TRANS, EMIS)` takes a known Markov model, specified by transition probability matrix TRANS and emission probability matrix EMIS, and uses it to generate

- A random sequence seq of emission symbols
- A random sequence states of states

The length of both seq and states is len.  $TRANS(i, j)$  is the probability of transition from state  $i$  to state  $j$ .  $EMIS(k, l)$  is the probability that symbol  $l$  is emitted from state  $k$ .

---

**Note** The function `hmmgenerate` begins with the model in state 1 at step 0, prior to the first emission. The model then makes a transition to state  $i_1$ , with probability  $T_{1i_1}$ , and generates an emission  $a_{k_1}$  with probability  $E_{i_1k_1}$ . `hmmgenerate` returns  $i_1$  as the first entry of states, and  $a_{k_1}$  as the first entry of seq. See “How the Toolbox Generates Random Sequences” on page 11-7 for more information

---

`hmmgenerate(..., 'Symbols', SYMBOLS)` specifies the symbols that are emitted. SYMBOLS can be a numeric array or a cell array of the names of the symbols. The default symbols are integers 1 through N, where N is the number of possible emissions.

`hmmgenerate(..., 'Statenames', STATENAMES)` specifies the names of the states. STATENAMES can be a numeric array or a cell array of the names of the states. The default state names are 1 through M, where M is the number of states.

Since the model always begins at state 1, whose transition probabilities are in the first row of TRANS, in the following example, the first entry of the output states is be 1 with probability 0.95 and 2 with probability 0.05.

# hmmgenerate

---

See “Setting Up the Model and Generating Data” on page 11-8 for an example of using `hmmgenerate`.

## Examples

```
trans = [0.95,0.05;
         0.10,0.90];

emis = [1/6, 1/6, 1/6, 1/6, 1/6, 1/6;...
        1/10, 1/10, 1/10, 1/10, 1/10, 1/2;];
[seq, states] = hmmgenerate(100,trans,emis)

[seq, states] = hmmgenerate(100,trans,emis,'Symbols',...
{'one','two','three','four','five','six'},...
'Statenames',{'fair';'loaded'})
```

## See Also

`hmmviterbi`, `hmmdecode`, `hmmestimate`, `hmmtrain`

**Purpose** Maximum likelihood estimate of model parameters for a hidden Markov model

**Syntax**

```
[ESTTR, ESTEMIT] = hmmtrain(seq, TRGUESS, EMITGUESS)
hmmtrain(...,'Algorithm', algorithm)
hmmtrain(...,'Symbols', SYMBOLS)
hmmtrain(...,'Tolerance', tol)
hmmtrain(...,'Maxiterations', maxiter)
hmmtrain(...,'Verbose', true)
hmmtrain(...,'Pseudoemissions', PSEUDOE)
hmmtrain(...,'Pesudotransitions', PSEUDOTR)
```

**Description**

[ESTTR, ESTEMIT] = `hmmtrain(seq,TRGUESS,EMITGUESS)` estimates the transition and emission probabilities for a hidden Markov model from the sequence `seq`, using the Baum-Welch algorithm. `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 `PSEUDO(i, k)` to be a positive number representing an estimate of the expected number of such emissions in the sequence seq.

`hmmtrain(..., 'Pseudotransitions', PSEUDOTR)` specifies pseudocount transition values for the Viterbi training algorithm. Use this argument to avoid zero probability estimates for transitions with very low probability that might not be represented in the sample sequence. `PSEUDOTR` should be a matrix of size M-by-M, where M is the number of states in the hidden Markov model. If the  $i \rightarrow j$  transition does not occur in states, you can set `PSEUDOTR(i, j)` to be a positive number representing an estimate of the expected number of such transitions in the sequence states.

See “Pseudotransitions and Pseudoemissions” on page 12-230 for more information.

If you know the states corresponding to the sequences, use `hmmestimate` to estimate the model parameters.

## Tolerance

The input argument 'tolerance' controls how many steps the `hmmtrain` algorithm executes before the function returns an answer. The algorithm terminates when all of the following three quantities are less than the value that you specify for tolerance:

- The log likelihood that the input sequence seq is generated by the currently estimated values of the transition and emission matrices
- The change in the norm of the transition matrix, normalized by the size of the matrix
- The change in the norm of the emission matrix, normalized by the size of the matrix

The default value of 'tolerance' is .0001. Increasing the tolerance decreases the number of steps the `hmmtrain` algorithm executes before it terminates.

## Maxiterations

The maximum number of iterations, 'maxiterations', controls the maximum number of steps the algorithm executes before it terminates. If the algorithm executes `maxiter` iterations before reaching the specified tolerance, the

algorithm terminates and the function returns a warning. If this occurs, you can increase the value of 'maxiterations' to make the algorithm reach the desired tolerance before terminating.

See “Using hmmtrain” on page 11-10 for an example of using hmmtrain.

**Examples:**

```
tr = [0.95,0.05;  
      0.10,0.90];  
  
e = [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,tr,e);  
seq2 = hmmgenerate(200,tr,e);  
seqs = {seq1,seq2};  
[estTR, estE] = hmmtrain(seqs,tr,e);
```

**See Also**

`hmmgenerate`, `hmmdecode`, `hmmestimate`, `hmmviterbi`

# hmmviterbi

---

**Purpose** Calculate the most probable state path for a hidden Markov model sequence

**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. See “How the Toolbox Generates Random Sequences” on page 11-7 for more information.

---

`hmmviterbi(..., 'Symbols', SYMBOLS)` specifies the symbols that are emitted. SYMBOLS can be a numeric array or a cell array of the names of the symbols. The default symbols are integers 1 through N, where N is the number of possible emissions.

`hmmviterbi(..., 'Statenames', STATENAMES)` specifies the names of the states. STATENAMES can be a numeric array or a cell array of the names of the states. The default state names are 1 through M, where M is the number of states.

See “Computing the Most Likely Sequence of States” on page 11-9 for an example of using `hmmviterbi`.

## Examples

```
trans = [0.95,0.05;
         0.10,0.90];

emis = [1/6, 1/6, 1/6, 1/6, 1/6, 1/6;
        1/10, 1/10, 1/10, 1/10, 1/10, 1/2;];

[seq, states] = hmmgenerate(100,trans,emis);
estimatedStates = hmmviterbi(seq,trans,emis);
```

```
[seq, states] =  
hmmgenerate(100,trans,emis,'Statenames',{'fair';'loaded'});  
estimatesStates =  
hmmviterbi(seq,trans,eemis,'Statenames',{'fair';'loaded'});
```

**See Also**

hmmgenerate, hmmdecode, hmestimate, hmmtrain

# hougen

---

**Purpose** Hougen-Watson model for reaction kinetics

**Syntax** `yhat = hougen(beta, x)`

**Description** `yhat = hougen(beta, x)` returns the predicted values of the reaction rate, `yhat`, as a function of the vector of parameters, `beta`, and the matrix of data, `X`. `beta` must have 5 elements and `X` must have three columns.

`hougen` is a utility function for `rsmdemo`.

The model form is:

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

**Reference** [1] Bates, D., and D. Watts. *Nonlinear Regression Analysis and Its Applications*. Wiley 1988. p. 271–272.

**See Also** `rsmdemo`



**Purpose** Hypergeometric cumulative distribution function (cdf)

**Syntax** `P = 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`, `hygeinv`, `hygepdf`, `hygernd`, `hygestat`

# hygeinv

---

**Purpose** Inverse of the hypergeometric cumulative distribution function (cdf)

**Syntax**  $X = \text{hygeinv}(P, M, K, N)$

**Description**  $\text{hygeinv}(P, M, K, N)$  returns the smallest integer  $X$  such that the hypergeometric cdf evaluated at  $X$  equals or exceeds  $P$ . You can think of  $P$  as the probability of observing  $X$  defective items in  $N$  drawings without replacement from a group of  $M$  items where  $K$  are defective.

**Examples** Suppose you are the Quality Assurance manager for a floppy disk manufacturer. The production line turns out floppy disks in batches of 1,000. You want to sample 50 disks from each batch to see if they have defects. You want to accept 99% of the batches if there are no more than 10 defective disks in the batch. What is the maximum number of defective disks should you allow in your sample of 50?

```
x = hygeinv(0.99, 1000, 10, 50)
```

```
x =  
    3
```

What is the median number of defective floppy disks in samples of 50 disks from batches with 10 defective disks?

```
x = hygeinv(0.50, 1000, 10, 50)
```

```
x =  
    0
```

**See Also** `hygecdf`, `hygepdf`, `hygernd`, `hygestat`, `icdf`

**Purpose** Hypergeometric probability density function (pdf)

**Syntax** `Y = hygepdf(X,M,K,N)`

**Description** `Y = hygecdf(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** `hygecdf`, `hygeinv`, `hygernd`, `hygestat`, `pdf`

# hygernd

---

**Purpose** Random numbers from the hypergeometric distribution

**Syntax**

```
R = hygernd(M,K,N)
R = hygernd(M,K,N,v)
R = hygernd(M,K,N,m,n)
```

**Description** `R = hygernd(M,K,N)` generates hypergeometric random numbers 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 hypergeometric random numbers 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 hypergeometric random numbers with parameters `M`, `K`, and `N`, where scalars `m` and `n` are the row and column dimensions of `R`.

## Reproducing the Output of hygernd

`hygernd` uses the MATLAB function `rand` to generate random numbers. When you call `hygernd`, you change the current state of `rand`, and thereby alter the output of subsequent calls to `hygernd` or any other functions that depend on `rand`. If you want to reproduce the output of `hygernd`, reset the state of `rand` to the same fixed value each time you call `hygernd`. For an example of how to do this, and a list of the Statistics Toolbox functions that depend on `rand`, see “Reproducing the Output of Random Number Functions” on page 2-46.

---

**Note** The result in the following example depends on the current state of `rand`. If you run the code in this example, your results may differ from the answer shown here.

---

**Example**

```
numbers = hygernd(1000,40,50)
```

```
numbers =
```

```
1
```

**See Also**

hygecdf, hygeinv, hygepdf, hygestat

# hygestat

---

**Purpose** Mean and variance for the hypergeometric distribution

**Syntax** [MN,V] = hygestat(M,K,N)

**Description** [MN,V] = hygestat(M,K,N) returns the mean and variance for the hypergeometric distribution with parameters specified by M, K, and N. Vector or matrix inputs for M, K, and N must have the same size, which is also the size of MN and V. A scalar input for M, K, or N is expanded to a constant matrix with the same dimensions as the other inputs.

The mean of the hypergeometric distribution with parameters M, K, and N is  $NK/M$ , and the variance is

$$N \frac{KM - KM - N}{M M M - 1}$$

**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** hygecdf, hygeinv, hygepdf, hygernd

**Purpose** Inverse of a specified cumulative distribution function (icdf)

**Syntax** `X = icdf('name',P,A1,A2,A3)`

**Description** `X = icdf('name',P,A1,A2,A3)` returns a matrix of critical values, *X*, where *'name'* is a string containing the name of the distribution. *P* is a matrix of probabilities, and *A1*, *A2*, and *A3* are matrices of distribution parameters. Depending on the distribution some of the parameters may not be necessary.

*P*, *A1*, *A2*, and *A3* 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.

`icdf` is a utility routine allowing you to access all the inverse cdfs in the Statistics Toolbox using the name of the distribution as a parameter. See “Overview of the Distributions” on page 2-45 for the list of available distributions.

### Examples

```
x = icdf('Normal',0.1:0.2:0.9,0,1)
```

```
x =
    -1.2816    -0.5244         0     0.5244     1.2816
```

```
x = icdf('Poisson',0.1:0.2:0.9,1:5)
```

```
x =
         0         1         3         5         8
```

### See Also

`betainv`, `binoinv`, `cdf`, `chi2inv`, `expinv`, `finv`, `gaminv`, `geoinv`, `hygeinv`, `logninv`, `nbbinv`, `ncfinv`, `nctinv`, `ncx2inv`, `norminv`, `pdf`, `poissinv`, `random`, `raylinv`, `tinv`, `unidinv`, `unifinv`, `wblinv`

# inconsistent

---

**Purpose** Calculate the inconsistency coefficient of a cluster tree

**Syntax** `Y = inconsistent(Z)`  
`Y = inconsistent(Z,d)`

**Description** `Y = inconsistent(Z)` computes the inconsistency coefficient for each link of the hierarchical cluster tree `Z`, where `Z` is an  $(m-1)$ -by-3 matrix generated by the linkage function. The inconsistency coefficient characterizes each link in a cluster tree by comparing its length with the average length of other links at the same level of the hierarchy. The higher the value of this coefficient, the less similar the objects connected by the link.

`Y = inconsistent(Z,d)` computes the inconsistency coefficient for each link in the hierarchical cluster tree `Z` to depth `d`, where `d` is an integer denoting the number of levels of the cluster tree that are included in the calculation. By default, `d=2`.

The output, `Y`, is an  $(m-1)$ -by-4 matrix formatted as follows.

Column	Description
1	Mean of the lengths of all the links included in the calculation.
2	Standard deviation of all the links included in the calculation.
3	Number of links included in the calculation.
4	Inconsistency coefficient.

For each link,  $k$ , the inconsistency coefficient is calculated as:

$$Y(k, 4) = (z(k, 3) - Y(k, 1)) / Y(k, 2)$$

For leaf nodes, nodes that have no further nodes under them, the inconsistency coefficient is set to 0.

**Example**

```
rand('state',12);  
X = rand(10,2);  
Y = pdist(X);
```



```
Z = linkage(Y, 'centroid');  
W = inconsistent(Z,3)  
  
W =  
  
    0.1313         0    1.0000         0  
    0.1386         0    1.0000         0  
    0.1727    0.0482    2.0000    0.7071  
    0.2391         0    1.0000         0  
    0.2242    0.0955    3.0000    1.0788  
    0.2357    0.1027    3.0000    0.9831  
    0.3222    0.1131    3.0000    0.9772  
    0.3376    0.1485    6.0000    1.4883  
    0.4920    0.1341    4.0000    1.1031
```

## See Also

`cluster`, `cophenet`, `clusterdata`, `dendrogram`, `linkage`, `pdist`, `squareform`

# iqr

---

**Purpose** Interquartile range (IQR) of a sample

**Syntax** `y = iqr(X)`  
`iqr(X,dim)`

**Description** `y = iqr(X)` returns the interquartile range of the values in `X`. For vector input, `y` is the difference between the 75th and the 25th percentiles of the sample in `X`. For matrix input, `y` is a row vector containing the interquartile range of each column of `X`. For N-dimensional arrays, `iqr` operates along the first non-singleton 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  $\sigma$  (the second parameter of the normal distribution.)

**Examples** This Monte Carlo simulation shows the relative efficiency of the IQR to the sample standard deviation for normal data.

```
x = normrnd(0,1,100,100);
s = std(x);
s_IQR = 0.7413 * iqr(x);
efficiency = (norm(s - 1)./norm(s_IQR - 1)).^2

efficiency =

    0.3297
```

**See Also** `std`, `mad`, `range`

**Purpose** Generate inverse Wishart random matrix

**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` whose inverse has the Wishart distribution with covariance matrix `inv(SIGMA)` and with `df` degrees of freedom. `SIGMA` can be a vector, a matrix, or a multidimensional array.

`W=iwishrnd(SIGMA,df,DI)` expects `DI` to be the Cholesky factor of the inverse of `SIGMA`. `DI` is an array of the same size as `SIGMA`. If you call `iwishrnd` multiple times using the same value of `SIGMA`, it is more efficient to supply `DI` instead of computing it each time.

`[W,DI]=iwishrnd(SIGMA,df)` returns `DI` so you can provide it as input in future calls to `iwishrnd`.

### Reproducing the Output of `iwishrnd`

`iwishrnd` uses the MATLAB functions `rand` and `randn` to generate random numbers. When you call `iwishrnd`, you change the current states of `rand` and `randn`, and thereby alter the output of subsequent calls to `iwishrnd` or any other functions that depend on `rand` or `randn`. If you want to reproduce the output of `iwishrnd`, reset the states of `rand` and `randn` to the same fixed values each time you call `iwishrnd`. For an example of how to do this, and a list of the Statistics Toolbox functions that depend on `rand` or `randn`, see “Reproducing the Output of Random Number Functions” on page 2-46.

**See Also** `wishrnd`

# jbtest

---

**Purpose** Jarque-Bera test for goodness-of-fit to a normal distribution

**Syntax**  
`H = jbtest(X)`  
`H = jbtest(X,alpha)`  
`[H,P,JBSTAT,CV] = jbtest(X,alpha)`

**Description** `H = jbtest(X)` performs the Jarque-Bera test on the input data vector `X` and returns `H`, the result of the hypothesis test. The result is `H=1` if you can reject the hypothesis that `X` has a normal distribution, or `H=0` if you cannot reject that hypothesis. you reject the hypothesis if the test is significant at the 5% level.

The Jarque-Bera test evaluates the hypothesis that `X` has a normal distribution with unspecified mean and variance, against the alternative that `X` does not have a normal distribution. The test is based on the sample skewness and kurtosis of `X`. For a true normal distribution, the sample skewness should be near 0 and the sample kurtosis should be near 3. The Jarque-Bera test determines whether the sample skewness and kurtosis are unusually different than their expected values, as measured by a chi-square statistic.

The Jarque-Bera test is an asymptotic test, and should not be used with small samples. You may want to use `lillietest` in place of `jbtest` for small samples.

`H = jbtest(X,alpha)` performs the Jarque-Bera test at the `100*alpha%` level rather than the 5% level, where `alpha` must be between 0 and 1.

`[H,P,JBSTAT,CV] = jbtest(X,alpha)` returns three additional outputs. `P` is the p-value of the test, `JBSTAT` is the value of the test statistic, and `CV` is the critical value for determining whether to reject the null hypothesis.

**Example** you can use `jbtest` to determine if car weights follow a normal distribution.

```
load carsmall
[h,p,j] = jbtest(Weight)

h =
     1

p =
    0.026718

j =
```

7.2448

With a p-value of 2.67%, you reject the hypothesis that the distribution is normal. With a log transformation, the distribution becomes closer to normal but is still significantly different at the 5% level.

```
[h,p,j] = jbtest(log(Weight))
```

```
h =  
    1
```

```
p =  
    0.043474
```

```
j =  
    6.2712
```

See `lillietest` for a different test of the same hypothesis.

## Reference

[1] Judge, G. G., R. C. Hill, W. E. Griffiths, H. Lutkepohl, and T.-C. Lee. *Introduction to the Theory and Practice of Econometrics*. New York, Wiley.

## See Also

`hist`, `kstest2`, `lillietest`

# 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 name-value pairs to control the iterative algorithm used by `kmeans`. Valid parameters are the following.

Parameter	Value
'distance'	Distance measure, in $p$ -dimensional space, that <code>kmeans</code> minimizes with respect to. <code>kmeans</code> computes centroid clusters differently for the different supported distance measures:
	'sqEuclidean' Squared Euclidean distance (default). Each centroid is the mean of the points in that cluster.

Parameter (Continued)	Value
	'cityblock' Sum of absolute differences, i.e., L1. 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.
'start'	Method used to choose the initial cluster centroid positions, sometimes known as "seeds". Valid starting values are:
	'sample' Select k observations from X at random (default).
	'uniform' Select k points uniformly at random from the range of X. Not valid with Hamming distance.
	'cluster' Perform a preliminary clustering phase on a random 10% subsample of X. This preliminary phase is itself initialized using 'sample'.

# kmeans

Parameter (Continued)	Value	
	Matrix	k-by-p matrix of centroid starting locations. In this case, you can pass in [ ] for k, and kmeans infers k from the first dimension of the matrix. You can also supply a 3-dimensional array, implying a value for the 'replicates' parameter from the array's third dimension.
'replicates'	Number of times to repeat the clustering, each with a new set of initial cluster centroid positions. kmeans returns the solution with the lowest value for sumd. You can supply 'replicates' implicitly by supplying a 3-dimensional array as the value for the 'start' parameter.	
'maxiter'	Maximum number of iterations. Default is 100.	
'emptyaction'	Action to take if a cluster loses all its member observations. Can be one of:	
	'error'	Treat an empty cluster as an error. (default)
	'drop'	Remove any clusters that become empty. kmeans sets the corresponding return values in C and D to NaN.
	'singleton'	Create a new cluster consisting of the one point furthest from its centroid.
'display'	Controls display of output.	
	'off'	Display no output.
	'iter'	Display information about each iteration during minimization, including the iteration number, the optimization phase (see "Algorithm"), the number of points moved, and the total sum of distances.



Parameter (Continued)	Value	
	'final'	Display a summary of each replication.
	'notify'	Display only warning and error messages. (default)

## Algorithm

kmeans uses a two-phase iterative algorithm to minimize the sum of point-to-centroid distances, summed over all k clusters:

- The first phase uses what the literature often describes as "batch" updates, where each iteration consists of reassigning points to their nearest cluster centroid, all at once, followed by recalculation of cluster centroids. You can think of this phase as providing a fast but potentially only approximate solution as a starting point for the second phase.
- The second phase uses what the literature often describes as "on-line" updates, where points are individually reassigned if doing so will reduce the sum of distances, and cluster centroids are recomputed after each reassignment. Each iteration during this second phase consists of one pass through all the points.

kmeans can converge to a local optimum, in this case, a partition of points in which moving any single point to a different cluster increases the total sum of distances. This problem can only be solved by a clever (or lucky, or exhaustive) choice of starting points.

## See Also

clusterdata, linkage, silhouette

## References

- [1] Seber, G.A.F., *Multivariate Observations*, Wiley, New York, 1984.
- [2] Spath, H., *Cluster Dissection and Analysis: Theory, FORTRAN Programs, Examples*, translated by J. Goldschmidt, Halsted Press, New York, 1985, 226 pp.

# kruskalwallis

---

**Purpose** Kruskal-Wallis nonparametric one-way Analysis of Variance (ANOVA)

**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.

<b>X value</b>	1.4	2.7	1.6	1.6	3.3	0.9	1.1
<b>Rank</b>	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  $X$ .

When  $X$  is a vector, `kruskalwallis` performs a Kruskal-Wallis test on the samples contained in  $X$ , as indexed by input `group` (a 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,  $-27^\circ$ ,  $65^\circ$ , and  $110^\circ$ , 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`. (You can copy a text version of the ANOVA table to the clipboard by using the **Copy Text** item on the **Edit** menu.)

`[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.

# kruskalwallis

---

## Assumptions

The Kruskal-Wallis test makes the following assumptions about the data in  $X$ :

- All samples come from populations having the same continuous distribution, apart from possibly different locations due to group effects.
- All observations are mutually independent.

The classical one-way ANOVA test replaces the first assumption with the stronger assumption that the populations have normal distributions.

## Example

This example compares the material strength study used with the `anova1` function, to see if the nonparametric Kruskal-Wallis procedure leads to the same conclusion. The example studies the strength of beams made from three alloys:

```
strength = [82 86 79 83 84 85 86 87 74 82 78 75 76 77 79 ...
            79 77 78 82 79];

alloy = {'st', 'st', 'st', 'st', 'st', 'st', 'st', 'st', ...
        'al1', 'al1', 'al1', 'al1', 'al1', 'al1', 'al1', ...
        'al2', 'al2', 'al2', 'al2', 'al2', 'al2'};
```

This example uses both classical and Kruskal-Wallis anova, omitting displays:

```
anova1(strength, alloy, 'off')

ans =
    1.5264e-004

kruskalwallis(strength, alloy, 'off')

ans =
    0.0018
```

Both tests find that the three alloys are significantly different, though the result is less significant according to the Kruskal-Wallis test. It is typical that when a data set has a reasonable fit to the normal distribution, the classical ANOVA test is more sensitive to differences between groups.

To understand when a nonparametric test may be more appropriate, let's see how the tests behave when the distribution is not normal. You can simulate this by replacing one of the values by an extreme value (an outlier).

```
strength(20)=120;
anova1(strength,alloy,'off')

ans =
    0.2501

kruskalwallis(strength,alloy,'off')

ans =
    0.0060
```

Now the classical ANOVA test does not find a significant difference, but the nonparametric procedure does. This illustrates one of the properties of nonparametric procedures – they are often not severely affected by changes in a small portion of the data.

## Reference

- [1] Gibbons, J. D., *Nonparametric Statistical Inference*, 2nd Edition, M. Dekker, 1985.
- [2] Hollander, M., and D. A. Wolfe, *Nonparametric Statistical Methods*, Wiley, 1973.

## See Also

anova1, boxplot, friedman, multcompare, ranksum

# ksdensity

---

**Purpose** Compute density estimate using a kernel smoothing method

**Syntax**

```
[f,xi] = ksdensity(x)
f = ksdensity(x,xi)
[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 covering 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.

`[f,xi,u] = ksdensity(...)` also returns the width of the kernel smoothing window.

`[...] = ksdensity(...,'param1',val1,'param2',val2,...)` specifies parameter name/value pairs to control the density estimation. Valid parameters and their possible values are:

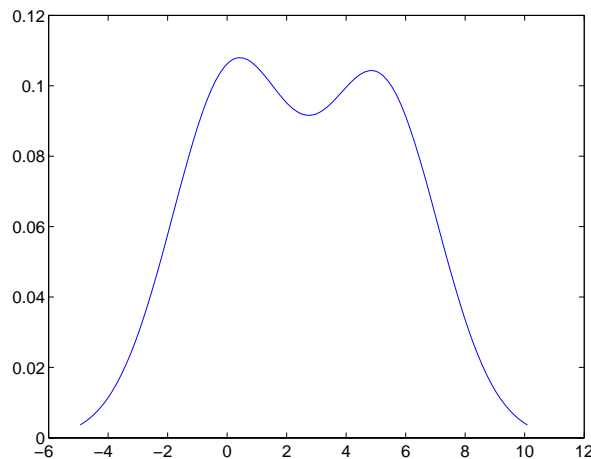
- |                          |  |
|--------------------------|--|
| <code>'censoring'</code> | A logical vector of the same length as <code>x</code> , indicating which entries are censoring times. Default is no censoring.   |
| <code>'kernel'</code>    | The type of kernel smoother to use. Choose the value as <code>'normal'</code> (default), <code>'box'</code> , <code>'triangle'</code> , or <code>'epanechnikov'</code> . Alternatively, you can specify some other function, as a function handle or as a string, e.g., <code>@normpdf</code> or <code>'normpdf'</code> . 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. |
| <code>'npoints'</code>   | The number of equally-spaced points in <code>xi</code> . Default is 100.   |

'support'	<ul style="list-style-type: none"><li>• 'unbounded' allows the density to extend over the whole real line (default).</li><li>• 'positive' restricts the density to positive values.</li><li>• A two-element vector gives finite lower and upper bounds for the support of the density.</li></ul>
'weights'	Vector of the same length as x, assigning weight to each x value. The default is equal weights).
'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.

## 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);
```



## See Also

hist, @ (function handle)

# ksdensity

---

## References

[1] Bowman, A.W. and A. Azzalini, *Applied Smoothing Techniques for Data Analysis*, Oxford University Press, 1997.



**Purpose** Kolmogorov-Smirnov test of the distribution of one sample

**Syntax**

```
H = kstest(X)
H = kstest(X,cdf)
H = kstest(X,cdf,alpha)
H = kstest(X,cdf,alpha,tail)
[H,P] = kstest(...)
[H,P,KSSTAT,] = kstest(...)
[H,P,KSSTAT,CV] = kstest(...)
```

**Description** `H = kstest(X)` performs a Kolmogorov-Smirnov test to compare the values in the data vector `X` with a standard normal distribution (that is, a normal distribution having mean 0 and variance 1). The null hypothesis for the Kolmogorov-Smirnov test is that `X` has a standard normal distribution. The alternative hypothesis that `X` does not have that distribution. The result `H` is 1 if you can reject the hypothesis that `X` has a standard normal distribution, or 0 if you cannot reject that hypothesis. You reject the hypothesis if the test is significant at the 5% level.

For each potential value  $x$ , the Kolmogorov-Smirnov test compares the proportion of values less than  $x$  with the expected number predicted by the standard normal distribution. The `kstest` function uses the maximum difference over all  $x$  values as its test statistic. Mathematically, this can be written as

$$\max(|F(x) - G(x)|)$$

where  $F(x)$  is the proportion of `X` values less than or equal to  $x$  and  $G(x)$  is the standard normal cumulative distribution function evaluated at  $x$ .

`H = kstest(X,cdf)` compares the distribution of `X` to the hypothesized continuous distribution defined by the two-column matrix `cdf`. Column one contains a set of possible  $x$  values, and column two contains the corresponding hypothesized cumulative distribution function values  $G(x)$ . If possible, you should define `cdf` so that column one contains the values in `X`. If there are values in `X` not found in column one of `cdf`, `kstest` will approximate  $G(X)$  by interpolation. All values in `X` must lie in the interval between the smallest and largest values in the first column of `cdf`. If the second argument is empty

(cdf = []), `kstest` uses the standard normal distribution as if there were no second argument.

The Kolmogorov-Smirnov test requires that `cdf` be predetermined. It is not accurate if `cdf` is estimated from the data. To test  $X$  against a normal distribution without specifying the parameters, use `lillietest` instead.

`H = kstest(X, cdf, alpha)` specifies the significance level `alpha` for the test. The default is 0.05.

`H = kstest(X, cdf, alpha, tail)` specifies the type of test in the string `tail`. `tail` can have one of the following values:

- 'unequal'
- 'larger'
- 'smaller'

The tests specified by these values are described in “Tests Specified by tail” on page 12-264.

`[H,P,KSSTAT,CV] = kstest(X, cdf, alpha, tail)` also returns the observed p-value `P`, the observed Kolmogorov-Smirnov statistic `KSSTAT`, and the cutoff value `CV` for determining if `KSSTAT` is significant. If the return value of `CV` is `NaN`, then `kstest` determined the significance calculating a p-value according to an asymptotic formula rather than by comparing `KSSTAT` to a critical value.

## Tests Specified by tail

Let  $S(x)$  be the empirical c.d.f. estimated from the sample vector  $X$ , let  $F(x)$  be the corresponding true (but unknown) population c.d.f., and let `CDF` be the known input c.d.f. specified under the null hypothesis. The one-sample Kolmogorov-Smirnov test tests the null hypothesis that  $F(x) = \text{CDF}$  for all  $x$  against the alternative specified by one of the following possible values of `tail`:

tail	Alternative Hypothesis	Test Statistic
'unequal'	$F(x)$ does not equal CDF (two-sided test)	$\max  S(x) - \text{CDF} $
'larger'	$F(x) > \text{CDF}$ (one-sided test)	$\max[S(x) - \text{CDF}]$
'smaller'	$F(x) < \text{CDF}$ (one-sided test)	$\max[\text{CDF} - S(x)]$

## Examples

**Example 1.** Let's generate some evenly spaced numbers and perform a Kolmogorov-Smirnov test to see how well they fit to a standard normal distribution:

```
x = -2:1:4
x =
    -2    -1     0     1     2     3     4

[h,p,k,c] = kstest(x,[],0.05,0)

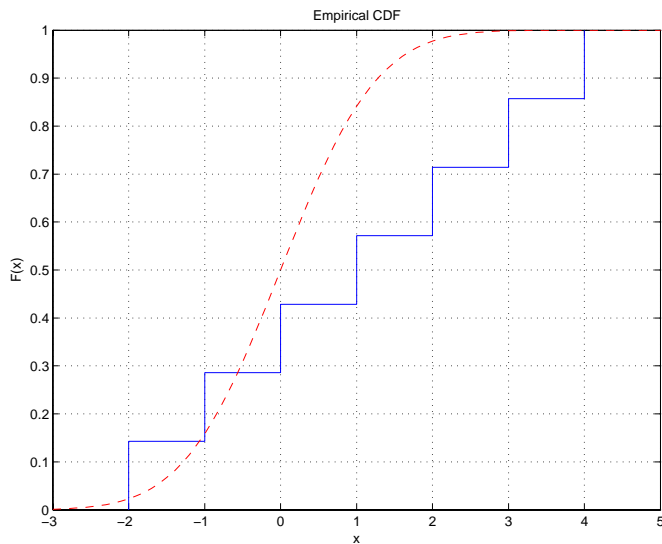
h =
     0
p =
    0.13632
k =
    0.41277
c =
    0.48342
```

You cannot reject the null hypothesis that the values come from a standard normal distribution. Although intuitively it seems that these evenly-spaced integers could not follow a normal distribution, this example illustrates the difficulty in testing normality in small samples.

To understand the test, it is helpful to generate an empirical cumulative distribution plot and overlay the theoretical normal distribution.

```
xx = -3:.1:5;
cdfplot(x)
hold on
plot(xx,normcdf(xx),'r--')
```

# kstest



The Kolmogorov-Smirnov test statistic is the maximum difference between these curves. It appears that this maximum of 0.41277 occurs as the data approaches  $x = 1.0$  from below. You can see that the empirical curve has the value  $3/7$  here, and you can easily verify that the difference between the curves is 0.41277.

```
normcdf(1) - 3/7
ans =
    0.41277
```

You can also perform a one-sided test. Setting `tail = -1` indicates that the alternative is  $F < G$ , so the test statistic counts only points where this inequality is true.

```
[h,p,k] = kstest(x, [], .05, -1)
h =
    0
p =
    0.068181
k =
```

0.41277

The test statistic is the same as before because in fact  $F < G$  at  $x = 1.0$ . However, the p-value is smaller for the one-sided test. If you carry out the other one-sided test, you see that the test statistic changes, and is the difference between the two curves near  $x = -1.0$ .

```
[h,p,k] = kstest(x,[],0.05,1)
```

```
h =  
    0
```

```
p =  
    0.77533
```

```
k =  
    0.12706
```

```
2/7 - normcdf(-1)
```

```
ans =  
    0.12706
```

**Example 2.** Now let's generate random numbers from a Weibull distribution, and test against that Weibull distribution and an exponential distribution.

```
x = wblrnd(1, 2, 100, 1);  
kstest(x, [x wblcdf(x, 1, 2)])
```

```
ans =  
    0
```

```
kstest(x, [x expcdf(x, 1)])
```

```
ans =  
    1
```

## See Also

kstest2, lillietest

# kstest2

---

**Purpose** Kolmogorov-Smirnov test to compare the distribution of two samples

**Syntax**

```
H = kstest2(X1,X2)
H = kstest2(X1,X2,alpha,tail)
[H,P,KSSTAT] = kstest2(X1,X2,cdf,alpha,tail)
```

**Description** H = kstest2(X1,X2) performs a two-sample Kolmogorov-Smirnov test to compare the distributions of values in the two data vectors X1 and X2 of length n1 and n2, respectively, representing random samples from some underlying distribution(s). The null hypothesis for this test is that X1 and X2 are drawn from the same continuous distribution. The alternative hypothesis is that they are drawn from different continuous distributions. The result H is 1 if you can reject the hypothesis that the distributions are the same, or 0 if you cannot reject that hypothesis. You reject the hypothesis if the test is significant at the 5% level.

For each potential value  $x$ , the Kolmogorov-Smirnov test compares the proportion of X1 values less than  $x$  with proportion of X2 values less than  $x$ . The kstest2 function uses the maximum difference over all  $x$  values as its test statistic. Mathematically, this can be written as

$$\max(|F1(x) - F2(x)|)$$

where  $F1(x)$  is the proportion of X1 values less than or equal to  $x$  and  $F2(x)$  is the proportion of X2 values less than or equal to  $x$ . Missing observations, indicated by NaNs are ignored.

H = kstest2(X1,X2,alpha) performs the test at the (100\*alpha)% significance level.

The decision to reject the null hypothesis occurs when the significance level, alpha, equals or exceeds the P-value.

H = kstest2(X1,X2,alpha,tail) accepts a string *tail* that specifies the type of test. *tail* can have one of the following values:

- 'unequal'
- 'larger'
- 'smaller'

The tests specified by these values are described in “Tests Specified by tail” on page 12-269

`[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  $n_1$  and  $n_2$  such that  $(n_1*n_2)/(n_1 + n_2) \geq 4$ .

`[H,P,ksstat] = KSTEST2(...)` also returns the Kolmogorov-Smirnov test statistic  $KSSTAT$  defined above for the test type indicated by *tail*.

## Tests Specified by tail

Let  $S_1(x)$  and  $S_2(x)$  be the empirical distribution functions from the sample vectors  $X_1$  and  $X_2$ , respectively, and  $F_1(x)$  and  $F_2(x)$  be the corresponding true (but unknown) population CDFs. The two-sample Kolmogorov-Smirnov test tests the null hypothesis that  $F_1(x) = F_2(x)$ , for all  $x$ , against the alternative hypothesis specified by *tail*, as described in the following table.

tail	Alternative Hypothesis	Test Statistic
'unequal'	$F_1(x)$ does not equal $F_2(x)$ (two-sided test)	$\max  S_1(x) - S_2(x) $
'larger'	$F_1(x) > F_2(x)$ (one-sided test)	$\max[S_1(x) - S_2(x)]$
'smaller'	$F(x) < F_2(x)$ (one-sided test)	$\max[S_2(x) - S_1(x)]$

## Examples

The following commands compare the distributions of a small evenly-spaced sample and a larger normal sample:

```
x = -1:1:5
y = randn(20,1);
[h,p,k] = kstest2(x,y)

h =
    1
p =
    0.0403
k =
    0.5714
```

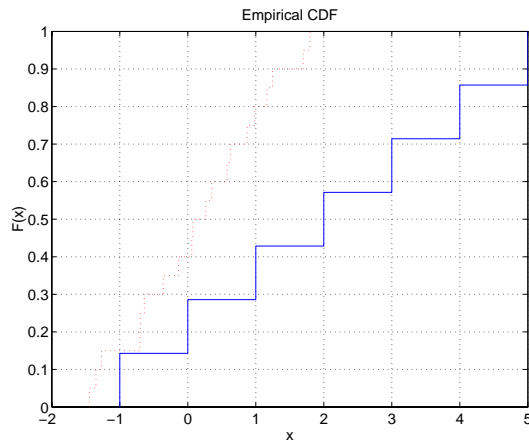
# kstest2

The difference between their distributions is significant at the 5% level ( $p = 4\%$ ). To visualize the difference, you can overlay plots of the two empirical cumulative distribution functions. The Kolmogorov-Smirnov statistic is the maximum difference between these functions. After changing the color and line style of one of the two curves, you can see that the maximum difference appears to be near  $x = 1.9$ . You can also verify that the difference equals the  $k$  value that `kstest2` reports:

```
cdfplot(x)
hold on
cdfplot(y)
h = findobj(gca,'type','line');
set(h(1),'linestyle',':','color','r')

1 - 3/7

ans =
    0.5714
```



## See Also

`kstest`, `lillietest`



**Purpose** Sample kurtosis

**Syntax**

```
k = kurtosis(X)
k = kurtosis(X, flag)
k = kurtosis(X, flag, dim)
```

**Description** `k = kurtosis(X)` returns the sample kurtosis of `X`. For vectors, `kurtosis(x)` is the kurtosis of the elements in the vector `x`. For matrices `kurtosis(X)` returns the sample kurtosis for each column of `X`. For N-dimensional arrays, `kurtosis` operates along the first non-singleton dimension of `X`.

`k = kurtosis(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 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 = 0` to correct for this systematic bias.

`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  $t$ .

---

**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.

---

# kurtosis

---

## Example

```
X = randn([5 4])  
  
X =  
    1.1650    1.6961   -1.4462   -0.3600  
    0.6268    0.0591   -0.7012   -0.1356  
    0.0751    1.7971    1.2460   -1.3493  
    0.3516    0.2641   -0.6390   -1.2704  
   -0.6965    0.8717    0.5774    0.9846  
  
k = kurtosis(X)  
  
k =  
    2.1658    1.2967    1.6378    1.9589
```

## See Also

mean, moment, skewness, std, var

<b>Purpose</b>	Leverage values for a regression
<b>Syntax</b>	<pre>h = leverage(data) h = leverage(data, 'model')</pre>
<b>Description</b>	<p><code>h = leverage(data)</code> finds the leverage of each row (point) in the matrix <code>data</code> for a linear additive regression model.</p> <p><code>h = leverage(data, 'model')</code> finds the leverage on a regression, using a specified model type, where <code>'model'</code> can be one of these strings:</p> <ul style="list-style-type: none"> <li>• <code>'linear'</code> – includes constant and linear terms</li> <li>• <code>'interaction'</code> – includes constant, linear, and cross product terms</li> <li>• <code>'quadratic'</code> – includes interactions and squared terms</li> <li>• <code>'purequadratic'</code> – includes constant, linear, and squared terms</li> </ul> <p>Leverage is a measure of the influence of a given observation on a regression due to its location in the space of the inputs.</p>
<b>Example</b>	<p>One rule of thumb is to compare the leverage to <math>2p/n</math> where <math>n</math> is the number of observations and <math>p</math> is the number of parameters in the model. For the Hald data set this value is 0.7692.</p> <pre>load hald h = max(leverage(ingredients, 'linear'))  h =     0.7004</pre> <p>Since <math>0.7004 &lt; 0.7692</math>, there are no high leverage points using this rule.</p>
<b>Algorithm</b>	<pre>[Q,R] = qr(x2fx(data, 'model')); leverage = (sum(Q' .* Q'))'</pre>
<b>Reference</b>	[1] Goodall, C. R. (1993). <i>Computation using the QR decomposition</i> . Handbook in Statistics, Volume 9. Statistical Computing (C. R. Rao, ed.). Amsterdam, NL Elsevier/North-Holland.
<b>See Also</b>	regstats

# lhsdesign

---

**Purpose** Generate a latin hypercube sample

**Syntax**

```
X = lhsdesign(n,p)
X = lhsdesign(...,'smooth','off')
X = lhsdesign(...,'criterion','c')
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','c')` iteratively generates latin hypercube samples to find the best one according to the criterion 'c', which can be:

'none'	No iteration
'maximin'	Maximize minimum distance between points
'correlation'	Reduce correlation

`X = lhsdesign(...,'iterations',k)` iterates up to `k` times in an attempt to improve the design according to the specified criterion. Default is `K = 5`.

Latin hypercube designs are useful when you need a sample that is random but that is guaranteed to be relatively uniformly distributed over each dimension.

**See Also** `lhsnorm`, `unifrnd`

**Purpose** Generate a latin hypercube sample with a normal distribution

**Syntax**  
`X = lhsnorm(mu,SIGMA,n)`  
`X = lhsnorm(mu,SIGMA,n,'onoff')`

**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,'onoff')` controls the amount of smoothing in the sample. If `'onoff'` 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 `'onoff'` is `'on'` (the default), each column has points uniformly distributed on the probability scale. For example, in place of  $0.5/n$  you use a value having a uniform distribution on the interval  $(0/n, 1/n)$ .

**See Also** `lhsdesign`, `mvnrnd`

# lillietest

---

**Purpose** Lilliefors test for goodness of fit to a normal distribution

**Syntax**

```
H = lillietest(X)
H = lillietest(X,alpha)
[H,P,LSTAT,CV] = lillietest(X,alpha)
```

**Description** `H = lillietest(X)` performs the Lilliefors test on the input data vector `X` and returns `H`, the result of the hypothesis test. The result `H` is 1 if you can reject the hypothesis that `X` has a normal distribution, or 0 if you cannot reject that hypothesis. you reject the hypothesis if the test is significant at the 5% level.

The Lilliefors test evaluates the hypothesis that `X` has a normal distribution with unspecified mean and variance, against the alternative that `X` does not have a normal distribution. This test compares the empirical distribution of `X` with a normal distribution having the same mean and variance as `X`. It is similar to the Kolmogorov-Smirnov test, but it adjusts for the fact that the parameters of the normal distribution are estimated from `X` rather than specified in advance.

`H = lillietest(X,alpha)` performs the Lilliefors test at the  $100 \cdot \alpha\%$  level rather than the 5% level. `alpha` must be between 0.01 and 0.2.

`[H,P,LSTAT,CV] = lillietest(X,alpha)` returns three additional outputs. `P` is the p-value of the test, obtained by linear interpolation in a set of table created by Lilliefors. `LSTAT` is the value of the test statistic. `CV` is the critical value for determining whether to reject the null hypothesis. If the value of `LSTAT` is outside the range of the Lilliefors table, `P` is returned as NaN but `H` indicates whether to reject the hypothesis.

**Example** Do car weights follow a normal distribution? Not exactly, because weights are always positive, and a normal distribution allows both positive and negative values. However, perhaps the normal distribution is a reasonable approximation.

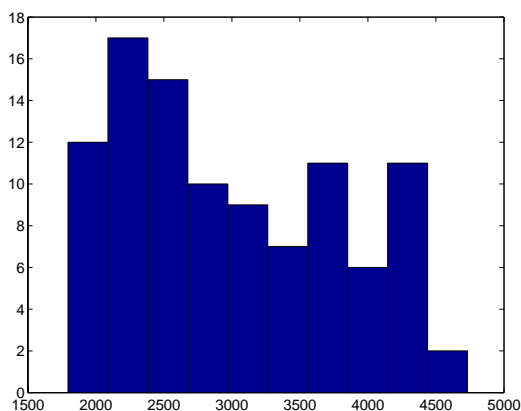
```
load carsmall
[h p l c] = lillietest(Weight);
[h p l c]

ans =
    1.0000    0.0232    0.1032    0.0886
```

The Lilliefors test statistic of 0.10317 is larger than the cutoff value of 0.0886 for a 5% level test, so you reject the hypothesis of normality. In fact, the p-value of this test is approximately 0.02.

To visualize the distribution, you can make a histogram. This graph shows that the distribution is skewed to the right – from the peak near 2250, the frequencies drop off abruptly to the left but more gradually to the right.

```
hist(Weight)
```



Sometimes it is possible to transform a variable to make its distribution more nearly normal. A log transformation, in particular, tends to compensate for skewness to the right.

```
[h p l c] = lillietest(log(Weight))
```

```
ans =
      0      0.13481      0.077924      0.0886
```

Now the p-value is approximately 0.13, so you do not reject the hypothesis.

## Reference

[1] Conover, W. J. (1980). *Practical Nonparametric Statistics*. New York, Wiley.

## See Also

hist, jbtest, kstest2

# linkage

---

**Purpose** Create hierarchical cluster tree

**Syntax** `Z = linkage(Y)`  
`Z = linkage(Y, 'method')`

**Description** `Z = linkage(Y)` creates a hierarchical cluster tree, using the Single Linkage algorithm. The input `Y` is a distance vector of length  $((m - 1) \cdot m / 2)$ -by-1, where  $m$  is the number of objects in the original data set. You can generate such a vector with the `pdist` function. `Y` can also be a more general dissimilarity matrix conforming to the output format of `pdist`.

`Z = linkage(Y, 'method')` computes a hierarchical cluster tree using the algorithm specified by `'method'`, where `'method'` can be any of the following character strings, whose definitions are explained in “Mathematical Definitions” on page 12-279.

<code>'single'</code>	Shortest distance (default)
<code>'complete'</code>	Furthest distance
<code>'average'</code>	Unweighted average distance (UPGMA) (also known as group average)
<code>'weighted'</code>	Weighted average distance (WPGMA)
<code>'centroid'</code>	Centroid distance (UPGMC)
<code>'median'</code>	Weighted center of mass distance (WPGMC)
<code>'ward'</code>	Inner squared distance (minimum variance algorithm)

---

**Note** When `'method'` is `'centroid'`, `'median'`, or `'ward'`, the output of `linkage` is meaningful only if the input `Y` contains Euclidean distances.

---

The output, `Z`, is an  $(m-1)$ -by-3 matrix containing cluster tree information. The leaf nodes in the cluster hierarchy are the objects in the original data set, numbered from 1 to  $m$ . They are the singleton clusters from which all higher



clusters are built. Each newly formed cluster, corresponding to row  $i$  in  $Z$ , is assigned the index  $m+i$ , where  $m$  is the total number of initial leaves.

Columns 1 and 2,  $Z(i, 1:2)$ , contain the indices of the objects that were linked in pairs to form a new cluster. This new cluster is assigned the index value  $m+i$ . There are  $m-1$  higher clusters that correspond to the interior nodes of the hierarchical cluster tree.

Column 3,  $Z(i, 3)$ , contains the corresponding linkage distances between the objects paired in the clusters at each row  $i$ .

For example, consider a case with 30 initial nodes. If the tenth cluster formed by the linkage function combines object 5 and object 7 and their distance is 1.5, then row 10 of  $Z$  will contain the values (5, 7, 1.5). This newly formed cluster will have the index  $10+30=40$ . If cluster 40 shows up in a later row, that means this newly formed cluster is being combined again into some bigger cluster.

### Mathematical Definitions

The '*method*' argument is a character string that specifies the algorithm used to generate the hierarchical cluster tree information. These linkage algorithms are based on different ways of measuring the distance between two clusters of objects. If  $n_r$  is the number of objects in cluster  $r$  and  $n_s$  is the number of objects in cluster  $s$ , and  $x_{ri}$  is the  $i$ th object in cluster  $r$ , the definitions of these various measurements are as follows:

- *Single linkage*, also called *nearest neighbor*, uses the smallest distance between objects in the two clusters.

$$d(r, s) = \min(\text{dist}(x_{ri}, x_{sj})), i \in (1, \dots, n_r), j \in (1, \dots, n_s)$$

- *Complete linkage*, also called *furthest neighbor*, uses the largest distance between objects in the two clusters.

$$d(r, s) = \max(\text{dist}(x_{ri}, x_{sj})), i \in (1, \dots, n_r), j \in (1, \dots, n_s)$$

- *Average linkage* uses the average distance between all pairs of objects in cluster  $r$  and cluster  $s$ .

$$d(r, s) = \frac{1}{n_r n_s} \sum_{i=1}^{n_r} \sum_{j=1}^{n_s} \text{dist}(x_{ri}, x_{sj})$$

# linkage

---

- *Centroid linkage* uses the Euclidean distance between the centroids of the two clusters,

$$d(r, s) = \|\bar{x}_r - \bar{x}_s\|_2$$

where

$$\bar{x}_r = \frac{1}{n_r} \sum_{i=1}^{n_r} x_{ri}$$

$x_s$  is defined similarly. The input  $Y$  should contain Euclidean distances.

- *Median linkage* uses the Euclidean distance between weighted centroids of the two clusters,

$$d(r, s) = \|\tilde{x}_r - \tilde{x}_s\|_2$$

where  $x_r$  and  $x_s$  are weighted centroids for the clusters  $r$  and  $s$ . If cluster  $r$  was created by combining clusters  $p$  and  $q$ ,  $x_r$  is defined recursively as

$$\tilde{x}_r = \frac{1}{2}(\tilde{x}_p + \tilde{x}_q)$$

$x_s$  is defined similarly. The input  $Y$  should contain Euclidean distances.

- *Ward's linkage* uses the incremental sum of squares; that is, the increase in the total within-cluster sum of squares as a result of joining clusters  $r$  and  $s$ . The within-cluster sum of squares is defined as the sum of the squares of the distances between all objects in the cluster and the centroid of the cluster. The equivalent distance is given by

$$d^2(r, s) = n_r n_s \frac{\|\bar{x}_r - \bar{x}_s\|_2^2}{(n_r + n_s)}$$

where  $\|\cdot\|_2$  is Euclidean distance, and  $x_r$  and  $x_s$  are the centroids of clusters  $r$  and  $s$ , as defined in the Centroid linkage, respectively. The input  $Y$  should contain Euclidean distances.

The centroid and median methods can produce a cluster tree that is not monotonic. This occurs when the distance from the union of two clusters,  $r$  and  $s$ , to a third cluster is less than the distance from either  $r$  or  $s$  to that

third cluster. In this case, sections of the dendrogram change direction. This is an indication that you should use another method.

## Example

```
X = [3 1.7; 1 1; 2 3; 2 2.5; 1.2 1; 1.1 1.5; 3 1];  
Y = pdist(X);  
Z = linkage(Y)
```

```
Z =  
    2.0000    5.0000    0.2000  
    3.0000    4.0000    0.5000  
    8.0000    6.0000    0.5099  
    1.0000    7.0000    0.7000  
   11.0000    9.0000    1.2806  
   12.0000   10.0000    1.3454
```

## See Also

`cluster`, `clusterdata`, `cophenet`, `dendrogram`, `inconsistent`, `kmeans`, `pdist`, `silhouette`, `squareform`

# logncdf

---

**Purpose** Lognormal cumulative distribution function

**Syntax** `P = logncdf(X,MU,SIGMA)`  
`[P, PLO, PUP] = logncdf(X, MU, SIGMA, PCOV, alpha)`

**Description** `P = logncdf(X,MU,SIGMA)` computes the lognormal cdf at each of the values in `X` using the corresponding means in `MU` and standard deviations in `SIGMA`. `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)` 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 `logncdf` computes confidence bounds for `P` using a normal approximation to the distribution of the estimate

$$\frac{\hat{X} - \hat{\mu}}{\hat{\sigma}}$$

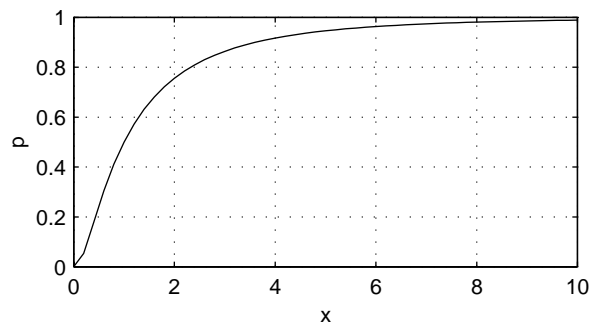
and then transforming those bounds to the scale of the output `P`. The computed bounds give approximately the desired confidence level when you estimate `MU`, `SIGMA`, and `PCOV` from large samples, but in smaller samples other methods of computing the confidence bounds might be more accurate.

The lognormal cdf is

$$p = F(x|\mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \int_0^x \frac{e^{-\frac{(\ln(t)-\mu)^2}{2\sigma^2}}}{t} dt$$

**Example**

```
x = (0:0.2:10);  
y = logncdf(x,0,1);  
plot(x,y); grid;  
xlabel('x'); ylabel('p');
```

**Reference**

[1] Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions, Second Edition*, John Wiley and Sons, 1993, p. 102-105.

**See Also**

`cdf`, `logninv`, `lognpdf`, `lognrnd`, `lognstat`

# lognfit

---

**Purpose** Parameter estimates and confidence intervals for lognormal data

**Syntax**

```
parmhat = lognfit(data)
[parmhat,parmci] = lognfit(data)
[parmhat,parmci] = lognfit(data, alpha)
[...] = lognfit(data, alpha, censoring)
[...] = lognfit(data, alpha, censoring, freq)
[...] = lognfit(data, alpha, censoring, freq, options)
```

**Description** `parmhat = lognfit(data)` returns the estimate of the parameters  $\mu$  and  $\sigma$  of the lognormal distribution, given the data in the vector `data`.

`[parmhat, parmci] = lognfit(data)` returns 95% confidence intervals for the parameter estimates on the  $\mu$  and  $\sigma$  parameters 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  $100(1 - \text{alpha})\%$  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. You can create options using the function `statset`. Enter `statset('lognfit')` to see the names and default values of the parameters that `lognfit` accepts in the options structure. See the reference page for `statset` for more information about these options.

**Example**

This example generates 100 independent samples of lognormal data with  $\mu = 0$  and  $\sigma = 3$ . `parmhat` is an estimate of  $\mu$  and `parmci` is a 99% confidence interval around `parmhat`. Notice that `parmci` contains the true value of  $\mu$ .

```
data = lognrnd(0, 3, 100, 1);  
[parmhat,parmci] = lognfit(data, 0.01)  
  
parmhat =  
    -0.2480    2.8902  
  
parmci =  
    -1.0071    2.4393  
     0.5111    3.5262
```

**See Also**

`logncdf`, `logninv`, `lognlike`, `lognpdf`, `lognrnd`, `lognstat`, `mle`, `statset`

# logninv

---

**Purpose** Inverse of the lognormal cumulative distribution function (cdf)

**Syntax**  
`X = logninv(P, MU, SIGMA)`  
`[X, XLO, XUP] = logninv(P, MU, SIGMA, PCOV, alpha)`

**Description** `X = logninv(P, MU, SIGMA)` computes the inverse lognormal 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, 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)` 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 `logninv` computes confidence bounds for P using a normal approximation to the distribution of the estimate

$$\hat{\mu} + \hat{\sigma}q$$

where  $q$  is the  $P$ th quantile from a normal distribution with mean 0 and standard deviation 1. The computed bounds give approximately the desired confidence level when you estimate MU, SIGMA, and PCOV from large samples, but in smaller samples other methods of computing the confidence bounds might be more accurate.

The lognormal inverse function is defined in terms of the lognormal cdf as

$$x = F^{-1}(p|\mu, \sigma) = \{x:F(x|\mu, \sigma) = p\}$$

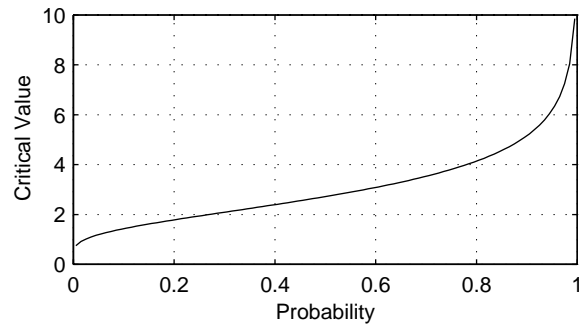
where

$$p = F(x|\mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \int_0^x e^{-\frac{(\ln(t)-\mu)^2}{2\sigma^2}} dt$$

**Example** `p = (0.005:0.01:0.995);`



```
crit = logninv(p,1,0.5);  
plot(p,crit)  
xlabel('Probability');ylabel('Critical Value'); grid
```

**Reference**

[1] Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions, Second Edition*, John Wiley and Sons, 1993. p. 102-105.

**See Also**

icdf, logncdf, lognpdf, lognrnd, lognstat

# lognlike

---

**Purpose** Negative log-likelihood for the lognormal distribution

**Syntax**

```
nlogL = lognlike(params, data)
[nlogL, AVAR] = lognlike(params, data)
[...] = lognlike(params, data, censoring)
[...] = lognlike(params, data, censoring, freq)
```

**Description** `nlogL = lognlike(params, data)` returns the negative of the log-likelihood for the lognormal distribution, evaluated at the parameters `params(1) = mu` and `params(2) = sigma`, given `data`. 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, `AVAR`. If the input parameter value in `params` is the maximum likelihood estimate, `avar` is its asymptotic variance. `AVAR` is based on the observed Fisher's information, not the expected information.

`[...] = lognlike(params, data, censoring)` accepts a Boolean vector, `censoring`, of the same size as `data`, which is 1 for observations that are right-censored and 0 for observations that are observed exactly.

`[...] = lognlike(params, data, censoring, freq)` accepts a frequency vector, `freq`, of the same size as `data`. The vector `freq` typically contains integer frequencies for the corresponding elements in `data`, but can contain any nonnegative values. Pass in `[]` for `censoring` to use its default value.

**See Also** `logncdf`, `lognfit`, `logninv`, `lognpdf`, `lognrnd`

**Purpose** Lognormal probability density function (pdf)

**Syntax** `Y = lognpdf(X,MU,SIGMA)`

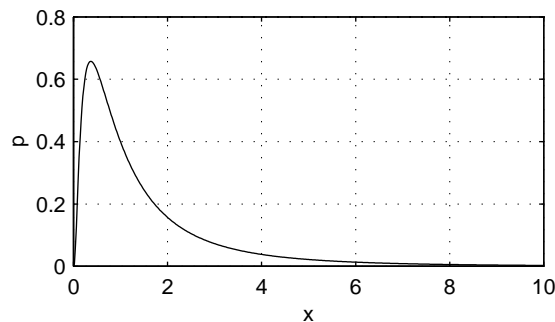
**Description** `Y = logncdf(X,MU,SIGMA)` computes the lognormal cdf at each of the values in `X` using the corresponding means in `MU` and standard deviations in `SIGMA`. `X`, `MU`, and `SIGMA` can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of `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

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

**Example**

```
x = (0:0.02:10);
y = lognpdf(x,0,1);
plot(x,y); grid;
xlabel('x'); ylabel('p')
```



**Reference** [1] Mood, A. M., F.A. Graybill, and D.C. Boes, *Introduction to the Theory of Statistics, Third Edition*, McGraw-Hill, 1974, p. 540-541.

**See Also** `logncdf`, `logninv`, `lognrnd`, `lognstat`, `pdf`

# lognrnd

---

**Purpose** Random matrices from the lognormal distribution

**Syntax**

```
R = lognrnd(MU, SIGMA)
R = lognrnd(MU, SIGMA, v)
R = lognrnd(MU, SIGMA, m, n)
```

**Description** `R = lognrnd(MU, SIGMA)` generates lognormal random numbers with parameters `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 = lognrnd(MU, SIGMA, v)` generates lognormal random numbers 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)` generates lognormal random numbers with parameters `MU` and `SIGMA`, where scalars `m` and `n` are the row and column dimensions of `R`.

## Reproducing the Output of lognrnd

`lognrnd` uses the MATLAB function `randn` to generate random numbers. When you call `lognrnd`, you change the current state of `randn`, and thereby alter the output of subsequent calls to `lognrnd` or any other functions that depend on `randn`. If you want to reproduce the output of `lognrnd`, reset the state of `randn` to the same fixed value each time you call `lognrnd`. For an example of how to do this, and a list of the Statistics Toolbox functions that depend on `randn`, see “Reproducing the Output of Random Number Functions” on page 2-46.

---

**Note** The result in the following example depends on the current state of `randn`. If you run the code in this example, your results might differ from the answer shown here.

---

**Example**

```
r = lognrnd(0,1,4,3)
```

r =

3.2058	0.4983	1.3022
1.8717	5.4529	2.3909
1.0780	1.0608	0.2355
1.4213	6.0320	0.4960

## Reference

[1] Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions, Second Edition*, John Wiley and Sons, 1993, p. 102-105.

## See Also

random, logncdf, logninv, lognpdf, lognstat

# lognstat

---

**Purpose** Mean and variance for the lognormal distribution

**Syntax** [M,V] = lognstat(MU,SIGMA)

**Description** [M,V] = lognstat(MU,SIGMA) returns the mean and variance of the lognormal 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 lognormal distribution with parameters  $\mu$  and  $\sigma$  is

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

and the variance is

$$e^{(2\mu + 2\sigma^2)} - e^{(2\mu + \sigma^2)}$$

**Example** [m,v]= lognstat(0,1)

m =  
1.6487

v =  
4.6708

**Reference** [1] Mood, A. M., F.A. Graybill, and D.C. Boes, *Introduction to the Theory of Statistics, Third Edition*, McGraw-Hill 1974 p. 540–541.

**See Also** logncdf, logninv, lognrnd, lognrnd

**Purpose** Least squares fit line(s)

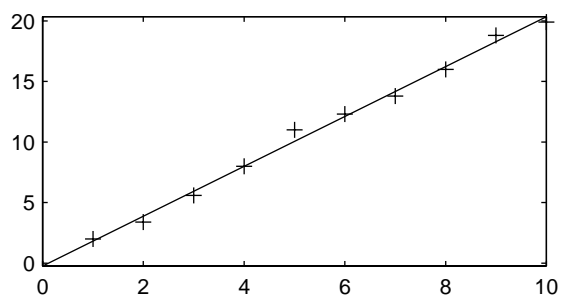
**Syntax** `lsline`  
`h = lsline`

**Description** `lsline` superimposes the least squares line on each line object in the current axes (except `LineStyle` `'-'`, `'--'`, `'.-'`).

`h = lsline` returns the handles to the line objects.

**Example**

```
y = [2 3.4 5.6 8 11 12.3 13.8 16 18.8 19.9]';
plot(y, '+');
lsline;
```



**See Also** `polyfit`, `polyval`

# mad

---

**Purpose** Mean or median absolute deviation (MAD) of a sample of data

**Syntax**

```
y = mad(X)
y = mad(X, dim)
y = mad(X, flag, dim)
```

**Description** `y = mad(X)` returns the mean absolute deviation of the values in `X`. For vector input, `y` is `mean(abs(X - mean(X)))`. For a matrix input, `y` is a row vector containing the mean absolute deviation of each column of `X`. For `N`-dimensional arrays, `mad` operates along the first nonsingleton dimension of `X`.

`Y = mad(X, 1)` computes `Y` based on medians, that is, `median(abs(X - median(X)))`.

`Y = mad(X, 0)` is the same as `mad(X)`, and uses means.

`mad(X, flag, dim)` takes the MAD along dimension `dim` of `X`.

`mad` treats NaNs as missing values and removes them.

**Remarks** The MAD is less efficient than the standard deviation as an estimate of the spread when all the data is from the normal distribution.

For normal data, multiply the MAD by 1.3 as a robust estimate of  $\sigma$  (the scale parameter of the normal distribution).

---

**Note** The default version of MAD, based on means, is also commonly referred to as the average absolute deviation (AAD).

---

**Examples** This example shows a Monte Carlo simulation of the relative efficiency of the MAD to the sample standard deviation for normal data.

```
x = normrnd(0,1,100,100);
s = std(x);
s_MAD = 1.3 * mad(x);
efficiency = (norm(s - 1)./norm(s_MAD - 1)).^2

efficiency =
```



---

0.5972

**Reference** [1] Sachs, L., *Applied Statistics: A Handbook of Techniques*, Springer-Verlag, 1984, p. 253.

**See Also** std, range, iqr

# mahal

---

**Purpose** Mahalanobis distance

**Syntax** `d = mahal(Y,X)`

**Description** `mahal(Y,X)` computes the Mahalanobis distance (in squared units) of each point (row) of the matrix `Y` from the sample in the matrix `X`.

The number of columns of `Y` must equal the number of columns in `X`, but the number of rows may differ. The number of rows in `X` must exceed the number of columns.

The Mahalanobis distance is a multivariate measure of the separation of a data set from a point in space. It is the criterion minimized in linear discriminant analysis.

**Example** The Mahalanobis distance of a matrix `r` when applied to itself is a way to find outliers.

```
r = mvnrnd([0 0],[1 0.9;0.9 1],100);
r = [r;10 10];
d = mahal(r,r);
last6 = d(96:101)
```

```
last6 =

    1.1036
    2.2353
    2.0219
    0.3876
    1.5571
   52.7381
```

The last element is clearly an outlier.

**See Also** `classify`

**Purpose** One-way Multivariate Analysis of Variance (MANOVA)

**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 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. 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.

<b>Field</b>	<b>Contents</b>
<code>W</code>	Within-groups sum of squares and cross-products matrix
<code>B</code>	Between-groups sum of squares and cross-products matrix
<code>T</code>	Total sum of squares and cross-products matrix
<code>dfW</code>	Degrees of freedom for <code>W</code>
<code>dfB</code>	Degrees of freedom for <code>B</code>
<code>dfT</code>	Degrees of freedom for <code>T</code>
<code>lambda</code>	Vector of values of Wilk's lambda test statistic for testing whether the means have dimension 0, 1, etc.
<code>chisq</code>	Transformation of lambda to an approximate chi-square distribution
<code>chisqdf</code>	Degrees of freedom for <code>chisq</code>
<code>eigenval</code>	Eigenvalues of $W^{-1}B$
<code>eigenvec</code>	Eigenvectors of $W^{-1}B$ ; these are the coefficients for the canonical variables <code>C</code> , and they are scaled so the within-group variance of the canonical variables is 1
<code>canon</code>	Canonical variables <code>C</code> , equal to $XC * \text{eigenvec}$ , where <code>XC</code> is <code>X</code> with columns centered by subtracting their means
<code>mdist</code>	A vector of Mahalanobis distances from each point to the mean of its group
<code>gmdist</code>	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 `manovac1uster` to graph a dendrogram showing the clusters among the group means.

### Assumptions

The MANOVA test makes the following assumptions about the data in  $X$ :

- The populations for each group are normally distributed.
- The variance-covariance matrix is the same for each population.
- All observations are mutually independent.

### Example

you can use `manova1` to determine whether there are differences in the averages of four car characteristics, among groups defined by the country where the cars were made.

```
load carbig
[d,p] = manova1([MPG Acceleration Weight Displacement],Origin)

d =
     3

p =
     0
    0.0000
    0.0075
    0.1934
```

There are four dimensions in the input matrix, so the group means must lie in a four-dimensional space. `manova1` shows that you cannot reject the hypothesis that the means lie in a three-dimensional subspace.

### References

[1] Krzanowski, W. J. *Principles of Multivariate Analysis*. Oxford University Press, 1988.

# manova1

---

## See Also

`anova1`, `canoncorr`, `gscatter`, `gplotmatrix`, `manovacluster`

**Purpose** Plot dendrogram showing group mean clusters after MANOVA

**Syntax**

```
manovacluster(stats)
manovacluster(stats,'method')
H = manovacluster(stats)
```

**Description** `manovacluster(stats)` generates a dendrogram plot of the group means after a multivariate analysis of variance (MANOVA). `stats` is the output `stats` structure from `manova1`. The clusters are computed by applying the single linkage method to the matrix of Mahalanobis distances between group means.

See `dendrogram` for more information on the graphical output from this function. The dendrogram is most useful when the number of groups is large.

`manovacluster(stats,'method')` uses the specified method in place of single linkage. `'method'` can be any of the following character strings that identify ways to create the cluster hierarchy. See `linkage` for further explanation.

'single'	Shortest distance (default)
'complete'	Largest distance
'average'	Average distance
'centroid'	Centroid distance
'ward'	Incremental sum of squares

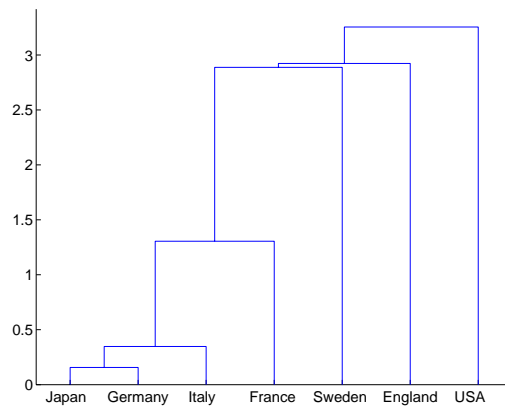
`H = manovacluster(stats,'method')` returns a vector of handles to the lines in the figure.

**Example** Let's analyze the larger car data set to determine which countries produce cars with the most similar characteristics.

```
load carbig
X = [MPG Acceleration Weight Displacement];
[d,p,stats] = manova1(X,Origin);
manovacluster(stats)
```

# manovacluster

---



## See Also

`cluster`, `dendrogram`, `linkage`, `manova1`



**Purpose** Nonmetric and metric multidimensional scaling

**Syntax**

```
Y = mdscale(D, p)
[Y, stress] = mdscale(D, p)
[...] = mdscale(..., 'param1', val1, 'param2', val2, ...)
```

**Description** `Y = mdscale(D, p)` performs non-metric 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 NaNs 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 interpoint distances, also known as stress1. This is the default.

- 'sstress' — Squared stress, normalized with the sum of 4th powers of the interpoint distances.

Choices for metric scaling are:

- 'metricstress' — Stress, normalized with the sum of squares of the dissimilarities.
- 'metricsstress' — Squared stress, normalized with the sum of 4th powers of the dissimilarities.
- 'sammon' — Sammon's nonlinear mapping criterion. Off-diagonal dissimilarities must be strictly positive with this criterion.
- 'strain' — A criterion equivalent to that used in classical multidimensional scaling.
- 'Weights' — A matrix or vector the same size as D, containing nonnegative dissimilarity weights. You can use these to weight the contribution of the corresponding elements of D in computing and minimizing stress. Elements of D corresponding to zero weights are effectively ignored.
- 'Start' — Method used to choose the initial configuration of points for Y. The choices are
  - 'cmdscale' — Use the classical multidimensional scaling solution. This is the default. 'cmdscale' is not valid when there are zero weights.
  - 'random' — Choose locations randomly from an appropriately scaled p-dimensional normal distribution with uncorrelated coordinates.
  - An  $n$ -by- $p$  matrix of initial locations, where  $n$  is the size of the matrix D and  $p$  is the number of columns of the output matrix Y. In this case, you can pass in [] for  $p$  and `mdscale` infers  $p$  from the second dimension of the matrix. You can also supply a three-dimensional array, implying a value for 'Replicates' from the array's third dimension.
- 'Replicates' — Number of times to repeat the scaling, each with a new initial configuration. The default is 1.
- 'Options' — Options for the iterative algorithm used to minimize the fitting criterion. Pass in an options structure created by `statset`. For example,  
`opts = statset('param1', val1, 'param2', val2, ...);`

```
[...] = mdscale(..., 'Options', opts)
```

The choices of statset parameters are

- 'Display' — Level of display output. The choices are 'off' (the default), 'iter', and 'final'.
- 'MaxIter' — Maximum number of iterations allowed. The default is 200.
- 'TolFun' — Termination tolerance for the stress criterion and its gradient. The default is 1e-4.
- 'TolX' — Termination tolerance for the configuration location step size. The default is 1e-4.

## Example

```
% Generate some 4D data, and create a dissimilarity matrix.
load cereal.mat
X = [Calories Protein Fat Sodium Fiber Carbo Sugars Shelf ...
Potass Vitamins];
X = X(strmatch('K',Mfg),:); % take a subset from a single
% manufacturer
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','NorthWest');
```

```
% 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

# mean

---

**Purpose** Average or mean value of vectors and matrices

**Syntax**  
`m = mean(X)`  
`m = mean(X, dim)`

**Description** `m = mean(X)` calculates the sample average

$$\bar{x}_j = \frac{1}{n} \sum_{i=1}^n x_{ij}$$

For vectors, `mean(x)` is the mean value of the elements in vector `x`. For matrices, `mean(X)` is a row vector containing the mean value of each column.

`m = mean(X, dim)` returns the mean values for elements along the dimension of `X` specified by scalar `dim`. For matrices, `mean(X, 2)` is a column vector containing the mean value of each row. The default of `dim` is 1.

The mean function is part of the standard MATLAB language.

**Example** These commands generate five samples of 100 normal random numbers with mean, zero, and standard deviation, one. The sample averages in `xbar` are much less variable ( $0.00 \pm 0.10$ ).

```
x = normrnd(0,1,100,5);
xbar = mean(x)

xbar =

    0.0727    0.0264    0.0351    0.0424    0.0752
```

**See Also** `median`, `std`, `cov`, `corrcoef`, `var`

**Purpose** Median value of vectors and matrices

**Syntax** `m = median(X)`

**Description** `m = median(X)` calculates the median value, which is the 50th percentile of a sample. The median is a robust estimate of the center of a sample of data, since outliers have little effect on it.

For vectors, `median(x)` is the median value of the elements in vector `x`. For matrices, `median(X)` is a row vector containing the median value of each column. Since `median` is implemented using `sort`, it can be costly for large matrices.

The `median` function is part of the standard MATLAB language.

## Examples

```
xodd = 1:5;
modd = median(xodd)

modd =

     3

meven = median(xeven)

meven =

    2.5000
```

This example shows robustness of the median to outliers.

```
xoutlier = [x 10000];
moutlier = median(xoutlier)

moutlier =

     3
```

**See Also** `mean`, `std`, `cov`, `corrcoef`

# mle

---

**Purpose** Maximum likelihood estimation

**Syntax**

```
phat = mle(data)
[phat, pci] = mle(data)
[...] = mle(data, 'distribution', dist)
[...] = mle(data, ..., 'name1', value1, 'name2', value2, ...)
[...] = 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`. `dist` is a character string containing the name of one of the distributions supported by `mle`.

`[...] = mle(data, ..., 'name1', value1, 'name2', value2, ...)` specifies optional argument name/value pairs chosen from the following list, in which argument names are case insensitive and partial matches are allowed.

Name	Value
'censoring'	A boolean vector of the same size as <code>data</code> , containing ones when the corresponding elements of <code>data</code> 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 <code>data</code> , containing non-negative integer frequencies for the corresponding elements in <code>data</code> . The default is one observation per element of <code>data</code> .

<b>Name</b>	<b>Value</b>
'alpha'	A value between 0 and 1 specifying a confidence level of $100 \cdot (1 - \alpha)\%$ 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.

The following table lists the distributions that you can use for the input argument 'distribution'. Arguments are not case sensitive and partial matches are accepted.

<b>Distribution</b>	<b>Value of 'dist'</b>
Beta	'Beta'
Bernoulli	'Bernoulli'
Binomial	'binomial'
Discrete uniform	'Discrete uniform'
Exponential	'Exponential'
Extreme value	'Extreme Value'
Gamma	'Gamma'
Geometric	'Geometric'
Log normal	'lognormal'
Negative binomial	'negative binomial'
Normal	'Normal'
Poisson	'Poisson'

Distribution	Value of 'dist'
Rayleigh	'Rayleigh'
Uniform	'Uniform'
Weibull	'Weibull'

`mle` can also fit a custom distribution that you define using distribution functions, in one of three ways.

`[...] = mle(data, 'pdf', pdf, 'cdf', cdf, 'start', start, ...)` returns MLEs for the parameters of the distribution defined by the probability density and cumulative distribution functions `pdf` and `cdf`. `pdf` and `cdf` are function handles created using the `@` sign. They accept as inputs a vector `data` and one or more individual distribution parameters, and return vectors of probability density values and cumulative probability values, respectively. If the 'censoring' name/value pair is not present, you can omit the 'cdf' name/value pair. `mle` computes the estimates by numerically maximizing the distribution's log-likelihood, and `start` is a vector containing initial values for the parameters.

`[...] = mle(data, 'logpdf', logpdf, 'logsf', logsf, 'start', start, ...)` returns MLEs for the parameters of the distribution defined by the log probability density and log survival functions `logpdf` and `logsf`. `logpdf` and `logsf` are function handles created using the `@` sign. They accept as inputs a vector `data` and one or more individual distribution parameters, and return vectors of logged probability density values and logged survival function values, respectively. This form is sometimes more robust to the choice of starting point than using `pdf` and `cdf` functions. If the 'censoring' name/value pair is not present, you can omit the 'logsf' name/value pair. `start` is a vector containing initial values for the distribution's parameters.

`[...] = mle(data, 'nloglf', nloglf, 'start', start, ...)` returns MLEs for the parameters of the distribution whose negative log-likelihood is given by `nloglf`. `nloglf` is a function handle, specified using the `@` sign, that accepts the four input arguments:

- `params` — a vector of distribution parameter values
- `data` - a vector of data



- `cens` - a boolean vector of censoring values
- `freq` - a vector of integer data frequencies

`nloglf` must accept all four arguments even if you do not supply the 'censoring' or 'frequency' name/value pairs (see above). However, `nloglf` can safely ignore its `cens` and `freq` arguments in that case. `nloglf` returns a scalar negative log-likelihood value and, optionally, a negative log-likelihood gradient vector (see the 'GradObj' `statset` parameter below). `start` is a vector containing initial values for the distribution's parameters.

`pdf`, `cdf`, `logpdf`, `logsf`, or `nloglf` can also be cell arrays whose first element is a function handle as defined above, and whose remaining elements are additional arguments to the function. `mle` places these arguments at the end of the argument list in the function call.

The following optional argument name/value pairs are valid only when 'pdf' and 'cdf', 'logpdf' and 'logcdf', or 'nloglf' are given:

- 'lowerbound' — A vector the same size as `start` containing lower bounds for the distribution parameters. The default is `-Inf`.
- 'upperbound' — A vector the same size as `start` containing upper bounds for the distribution parameters. The default is `Inf`.
- 'optimfun' — A string, either 'fminsearch' or 'fmincon', naming the optimization function to be used in maximizing the likelihood. The default is 'fminsearch'. You can only specify 'fmincon' if the Optimization Toolbox is available.

When fitting a custom distribution, use the 'options' parameter to control details of the maximum likelihood optimization. See `statset('mlecustom')`

for parameter names and default values. `mle` interprets the following `statset` parameters for custom distribution fitting as follows:

Parameter	Value
'GradObj'	'on' or 'off', indicating whether or not <code>fmincon</code> 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 <code>fminsearch</code> .
'DerivStep'	The relative difference used in finite difference derivative approximations when using <code>fmincon</code> , and 'GradObj' is 'off'. 'DerivStep' can be a scalar, or the same size as 'start'. The default is $\text{eps}^{(1/3)}$ . Ignored when using <code>fminsearch</code> .
'FunValCheck'	'on' or 'off', indicating whether or not <code>mle</code> 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 <code>fmincon</code> . <code>mle</code> treats upper and lower bounds as strict inequalities (i.e., open bounds). With <code>fmincon</code> , this is approximated by creating closed bounds inset from the specified upper and lower bounds by <code>TolBnd</code> . The default is $1\text{e-}6$ .

## Example

```
rv = binornd(20,0.75)
rv =
    16
[p,pci] = mle('binomial',rv,0.05,20)
p =
    0.8000
pci =
    0.5634
```

0.9427

**See Also**

betafit, binofit, evfit, expfit, gamfit, lognfit, nbinfit, normfit, mlecov,  
poissfit, raylfit, statset, unifit, wblfit

# mlecov

---

**Purpose** Asymptotic covariance matrix of maximum likelihood estimators

**Syntax**

```
ACOV = mlecov(params, data, ...)  
ACOV = mlecov(params, DATA, 'pdf', PDF, 'cdf', CDF)  
ACOV = mlecov(params, data, 'logpdf', logpdf, 'logsf', logsf)  
ACOV = mlecov(params, data, 'nloglf', nloglf)  
[...] = mlecov(params, data, ..., 'parm1', val1, 'parm2', 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 `data` and returns the negative inverse of that hessian. `ACOV` is a  $p$ -by- $p$  matrix, where  $p$  is the number of elements in `params`.

You must specify a distribution after the input argument `data`, as follows.

`ACOV = mlecov(params, data, 'pdf', pdf, 'cdf', cdf)` enables you to define a distribution by its probability density and cumulative distribution functions, `pdf` and `cdf`, respectively. `pdf` and `cdf` are function handles that you create using the @ sign. They accept a vector of data and one or more individual distribution parameters as inputs and return vectors of probability density function values and cumulative distribution values, respectively. If the 'censoring' name/value pair (see below) is not present, you can omit the 'cdf' name/value pair.

`ACOV = mlecov(params, data, 'logpdf', logpdf, 'logsf', logsf)` enables you to define a distribution by its log probability density and log survival functions, `logpdf` and `logsf`, respectively. `logpdf` and `logsf` are function handles that you create using the @ sign. They accept as inputs a vector of data and one or more individual distribution parameters, and return vectors of logged probability density values and logged survival function values, respectively. If the 'censoring' name/value pair (see below) is not present, you can omit the 'logsf' name/value pair.

`ACOV = mlecov(params, data, 'nloglf', nloglf)` enables you to define a distribution by its log-likelihood function. `nloglf` is a function handle, specified using the @ sign, that accepts the 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 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. `mle` places these arguments at the end of the argument list in the function call.

`[...] = mlecov(params, data, ..., 'parm1', val1, 'parm2', val2, ...)` specifies optional argument name/value pairs chosen from the following.

Name	Value
'censoring'	A boolean vector of the same size as <code>data</code> , containing ones when the corresponding elements of <code>data</code> are right-censored observations and zeros when the corresponding elements are exact observations. The default is that all observations observed exactly. Censoring is not supported for all distributions.

Name	Value
'frequency'	A vector of the same size as data containing non-negative 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: <ul style="list-style-type: none"><li>• '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'.</li><li>• 'DerivStep' — Relative step size used in finite difference derivative calculations. Step size can be a scalar, or the same size as params. The default is <math>\text{eps}^{(1/3)}</math>.</li></ul>

## Example

```
% 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)

function logpdf = betalogpdf(x,a,b)
logpdf = (a-1)*log(x) + (b-1)*log(1-x) - betaln(a,b);
```

## See Also

mle

**Purpose** Central moment of all orders

**Syntax**  
`m = moment(X,order)`  
`m = 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

$$m_n = E(x - \mu)^k$$

where  $E(x)$  is the expected value of  $x$ .

### Example

```
X = randn([6 5])
```

```
X =
```

```
    1.1650    0.0591    1.2460   -1.2704   -0.0562
    0.6268    1.7971   -0.6390    0.9846    0.5135
    0.0751    0.2641    0.5774   -0.0449    0.3967
    0.3516    0.8717   -0.3600   -0.7989    0.7562
   -0.6965   -1.4462   -0.1356   -0.7652    0.4005
    1.6961   -0.7012   -1.3493    0.8617   -1.3414
```

```
m = moment(X,3)
```

```
m =
```

```
   -0.0282    0.0571    0.1253    0.1460   -0.4486
```

**See Also** `kurtosis`, `mean`, `skewness`, `std`, `var`

# multcompare

---

**Purpose** Multiple comparison test of means or other estimates

**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 12-323.

In a one-way analysis of variance, you compare the means of several groups to test the hypothesis that they are all the same, against the general alternative that they are not all the same. Sometimes this alternative may be too general. You may need information about which pairs of means are significantly different, and which are not. A test that can provide such information is called a “multiple comparison procedure.”

When you perform a simple t-test of one group mean against another, you specify a significance level that determines the cutoff value of the t statistic. For example, you can specify the value `alpha = 0.05` to insure that when there is no real difference, you will incorrectly find a significant difference no more than 5% of the time. When there are many group means, there are also many pairs to compare. If you applied an ordinary t-test in this situation, the alpha value would apply to each comparison, so the chance of incorrectly finding a significant difference would increase with the number of comparisons. Multiple comparison procedures are designed to provide an upper bound on the probability that *any* comparison will be incorrectly found significant.

The output `c` contains the results of the test in the form of a five-column matrix. Each row of the matrix represents one test, and there is one row for each pair of groups. The entries in the row indicate the means being compared, the estimated difference in means, and a confidence interval for the difference.

For example, suppose one row contains the following entries.

```
2.0000    5.0000    1.9442    8.2206    14.4971
```



These numbers indicate that the mean of group 2 minus the mean of group 5 is estimated to be 8.2206, and a 95% confidence interval for the true mean is [1.9442, 14.4971].

In this example the confidence interval does not contain 0.0, so the difference is significant at the 0.05 level. If the confidence interval did contain 0.0, the difference would not be significant at the 0.05 level.

The `multcompare` function also displays a graph with each group mean represented by a symbol and an interval around the symbol. Two means are significantly different if their intervals are disjoint, and are not significantly different if their intervals overlap. You can use the mouse to select any group, and the graph will highlight any other groups that are significantly different from it.

`c = multcompare(stats, param1, val1, param2, val2, ...)` specifies one or more of the parameter name/value pairs described in the following table.

Parameter Name	Parameter Values
'alpha'	Scalar between 0 and 1 that determines the confidence levels of the intervals in the matrix <code>c</code> and in the figure (default is 0.05). The confidence level is $100 * (1 - \text{alpha})\%$ .
'displayopt'	Either 'on' (the default) to display a graph of the estimates with comparison intervals around them, or 'off' to omit the graph. See “Examples” on page 12-323.
'ctype'	Specifies the type of critical value to use for the multiple comparison. “Values of <code>ctype</code> ” on page 12-321 describes the allowed values for 'ctype'.

# multcompare

---

Parameter Name	Parameter Values
'dimension'	A vector specifying the dimension or dimensions over which the population marginal means are to be calculated. Use only if you create stats with the function <code>anovan</code> . The default is 1 to compute over the first dimension. See “dimension Parameter” on page 12-322 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 12-322

`[c,m] = multcompare(...)` returns an additional matrix `m`. The first column of `m` contains the estimated values of the means (or whatever statistics are being compared) for each group, and the second column contains their standard errors.

`[c,m,h] = multcompare(...)` returns a handle `h` to the comparison graph. Note that the title of this graph contains instructions for interacting with the graph, and the *x*-axis label contains information about which means are significantly different from the selected mean. If you plan to use this graph for presentation, you may want to omit the title and the *x*-axis label. You can remove them using interactive features of the graph window, or you can use the following commands.

```
title('')  
xlabel('')
```

`[c,m,h,gnames] = multcompare(...)` returns `gnames`, a cell array with one row for each group, containing the names of the groups.

## Values of ctype

The following table describes the allowed values for the parameter 'ctype'.

Values 'ctype'	Meaning
'hsd' or 'tukey-kramer'	Use Tukey's honestly significant difference criterion. This is the default, and it is based on the Studentized range distribution. It is optimal for balanced one-way ANOVA and similar procedures with equal sample sizes. It has been proven to be conservative for one-way ANOVA with different sample sizes. According to the unproven Tukey-Kramer conjecture, it is also accurate for problems where the quantities being compared are correlated, as in analysis of covariance with unbalanced covariate values.
'lsd'	Use Tukey's least significant difference procedure. This procedure is a simple t-test. It is reasonable if the preliminary test (say, the one-way ANOVA F statistic) shows a significant difference. If it is used unconditionally, it provides no protection against multiple comparisons.
'bonferroni'	Use critical values from the t distribution, after a Bonferroni adjustment to compensate for multiple comparisons. This procedure is conservative, but usually less so than the Scheffé procedure.
'dunn-sidak'	Use critical values from the t distribution, after an adjustment for multiple comparisons that was proposed by Dunn and proved accurate by Šidák. This procedure is similar to, but less conservative than, the Bonferroni procedure.
'scheffe'	Use critical values from Scheffé's S procedure, derived from the F distribution. This procedure provides a simultaneous confidence level for comparisons of all linear combinations of the means, and it is conservative for comparisons of simple differences of pairs.

## Values of estimate

The allowable values of the parameter 'estimate' depend on the function that was the source of the stats structure, according to the following table.

Source	Allowable Values of 'estimate'
'anova1'	Ignored. Always compare the group means.
'anova2'	Either 'column' (the default) or 'row' to compare column or row means.
'anovan'	Ignored. Always compare the population marginal means as specified by the dim argument.
'aoctool'	Either 'slope', 'intercept', or 'pmm' to compare slopes, intercepts, or population marginal means. If the analysis of covariance model did not include separate slopes, then 'slope' is not allowed. If it did not include separate intercepts, then no comparisons are possible.
'friedman'	Ignored. Always compare average column ranks.
'kruskalwallis'	Ignored. Always compare average group ranks.

## dimension Parameter

The dimension parameter is a vector specifying the dimension or dimensions over which the population marginal means are to be calculated. For example, if  $\text{dim} = 1$ , 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  $\text{dim} = [1\ 3]$ , 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

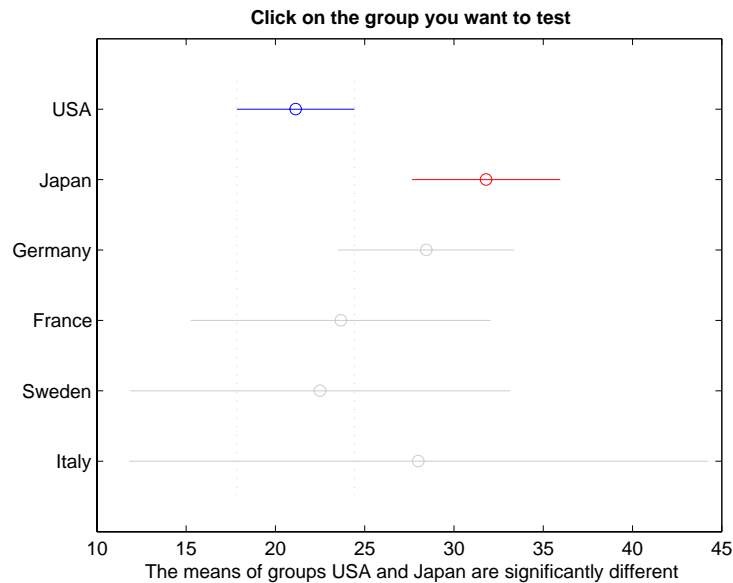
The following example performs a 1-way analysis of variance (ANOVA) and displays group means with their names.

```
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]	[7.0513]

multcompare also displays the following graph of the estimates with comparison intervals around them.



You can click the graphs of each country to compare its mean to those of other countries.

## Example 2

The following continues the example described in the `anova1` reference page, which is related to testing the material strength in structural beams. From the `anova1` output you found significant evidence that the three types of beams are not equivalent in strength. Now you can determine where those differences lie. First you create the data arrays and you perform one-way ANOVA.

```
strength = [82 86 79 83 84 85 86 87 74 82 78 75 76 77 79 ...  
           79 77 78 82 79];  
alloy = {'st', 'st', 'st', 'st', 'st', 'st', 'st', 'st', ...  
        'a1', 'a1', 'a1', 'a1', 'a1', 'a1', 'a1', ...  
        'a2', 'a2', 'a2', 'a2', 'a2', 'a2'};  
[p,a,s] = anova1(strength,alloy);
```

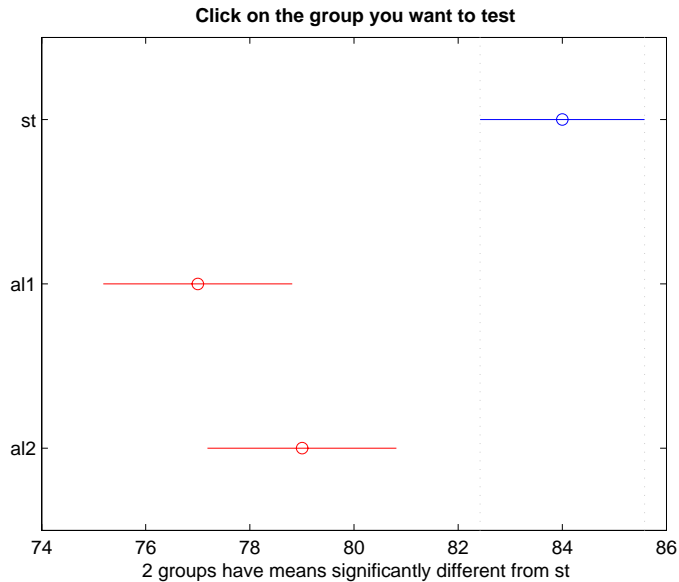
Among the outputs is a structure that you can use as input to `multcompare`.

```
[c,m,h,nms] = multcompare(s);
```

```
[nms num2cell(c)]
```

```
ans =
```

```
'st'      [1]      [2]      [ 3.6064]      [ 7]      [10.3936]
'al1'     [1]      [3]      [ 1.6064]      [ 5]      [ 8.3936]
'al2'     [2]      [3]      [-5.6280]     [-2]      [ 1.6280]
```



The third row of the output matrix shows that the differences in strength between the two alloys is not significant. A 95% confidence interval for the difference is [-5.6, 1.6], so you cannot reject the hypothesis that the true difference is zero.

The first two rows show that both comparisons involving the first group (steel) have confidence intervals that do not include zero. In other words, those differences are significant. The graph shows the same information.

**See Also**

anova1, anova2, anovan, aocool, friedman, kruskalwallis

## References

- [1] Hochberg, Y., and A. C. Tamhane, *Multiple Comparison Procedures*, 1987, Wiley.
- [2] Milliken, G. A., and D. E. Johnson, *Analysis of Messy Data, Volume 1: Designed Experiments*, 1992, Chapman & Hall.
- [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.



**Purpose** Multivariate normal probability density function (pdf)

**Syntax**

```
y = mvnpdf(X)
y = mvnpdf(X,mu)
y = mvnpdf(X,mu,SIGMA)
```

**Description** `y = mvnpdf(X)` returns the  $n$ -by-1 vector  $y$ , containing the probability density of the multivariate normal distribution with zero mean and identity covariance matrix, evaluated at each row of the  $n$ -by- $d$  matrix  $X$ . Rows of  $X$  correspond to observations and columns correspond to variables or coordinates.

`y = mvnpdf(X,mu)` returns the density of the multivariate normal distribution with mean  $\mu$  and identity covariance matrix, evaluated at each row of  $X$ .  $\mu$  is a 1-by- $d$  vector, or an  $n$ -by- $d$  matrix. If  $\mu$  is a matrix, the density is evaluated for each row of  $X$  with the corresponding row of  $\mu$ .  $\mu$  can also be a scalar value, which `mvnpdf` replicates to match the size of  $X$ .

`y = mvnpdf(X,mu,SIGMA)` returns the density of the multivariate normal distribution with mean  $\mu$  and covariance  $SIGMA$ , evaluated at each row of  $X$ .  $SIGMA$  is a  $d$ -by- $d$  matrix, or an  $d$ -by- $d$ -by- $n$  array, in which case the density is evaluated for each row of  $X$  with the corresponding page of  $SIGMA$ , i.e., `mvnpdf` computes  $y(i)$  using  $X(i,:)$  and  $SIGMA(:,:,i)$ . Specify `[]` for  $\mu$  to use its default value when you want to specify only  $SIGMA$ .

If  $X$  is a 1-by- $d$  vector, `mvnpdf` replicates it to match the leading dimension of  $\mu$  or the trailing dimension of  $SIGMA$ .

**Example**

```
mu = [1 -1];
Sigma = [.9 .4; .4 .3];
X = mvnrnd(mu,Sigma,10);
p = mvnpdf(X,mu,Sigma);
```

**See Also** `mvnrnd`, `normpdf`

# mvnrnd

---

**Purpose** Random matrices from the multivariate normal distribution

**Syntax** `R = mvnrnd(mu, SIGMA)`  
`R = mvnrnd(mu, SIGMA, cases)`

**Description** `R = mvnrnd(mu, SIGMA)` returns an  $n$ -by- $d$  matrix  $R$  of random vectors chosen from the multivariate normal distribution with mean  $\mu$ , and covariance  $\text{SIGMA}$ .  $\mu$  is an  $n$ -by- $d$  matrix, and `mvnrnd` generates each row of  $R$  using the corresponding row of  $\mu$ .  $\text{SIGMA}$  is a  $d$ -by- $d$  symmetric positive semi-definite matrix, or a  $d$ -by- $d$ -by- $n$  array. If  $\text{SIGMA}$  is an array, `mvnrnd` generates each row of  $R$  using the corresponding page of  $\text{SIGMA}$ , i.e., `mvnrnd` computes  $R(i, :)$  using  $\mu(i, :)$  and  $\text{SIGMA}(:, :, i)$ . If  $\mu$  is a 1-by- $d$  vector, `mvnrnd` replicates it to match the trailing dimension of  $\text{SIGMA}$ .

`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  $\text{SIGMA}$ .

## Reproducing the Output of mvnrnd

`mvnrnd` uses the MATLAB function `randn` to generate random numbers. When you call `mvnrnd`, you change the current state of `randn`, and thereby alter the output of subsequent calls to `mvnrnd` or any other functions that depend on `randn`. If you want to reproduce the output of `mvnrnd`, reset the state of `randn` to the same fixed value each time you call `mvnrnd`. For an example of how to do this, and a list of the Statistics Toolbox functions that depend on `randn`, see “Reproducing the Output of Random Number Functions” on page 2-46.

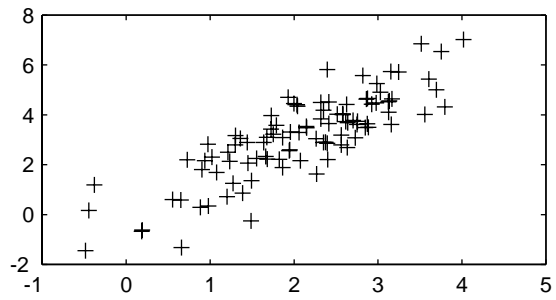
---

**Note** The result in the following example depends on the current state of `randn`. If you run the code in this example, your results may differ from the plot shown here.

---

**Example**

```
mu = [2 3];
sigma = [1 1.5; 1.5 3];
r = mvnrnd(mu, sigma, 100);
plot(r(:,1), r(:,2), '+')
```



**See Also**

lhsnorm, mvnpdf, normrnd

# mvtrnd

---

**Purpose** Random matrices from the multivariate t distribution

**Syntax** `r = mvtrnd(C,df,cases)`

**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.

## Reproducing the Output of mvtrnd

`mvtrnd` uses the MATLAB functions `rand` and `randn` to generate random numbers. When you call `mvtrnd`, you change the current states of `rand` and `randn`, and thereby alter the output of subsequent calls to `mvtrnd` or any other functions that depend on `rand` or `randn`. If you want to reproduce the output of `mvtrnd`, reset the states of `rand` and `randn` to the same fixed values each time you call `mvtrnd`. For an example of how to do this, and a list of the Statistics Toolbox functions that depend on `rand` or `randn`, see “Reproducing the Output of Random Number Functions” on page 2-46.

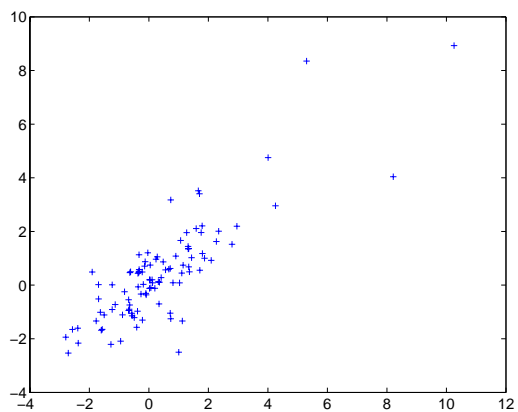
---

**Note** The results in the following example depend on the current states of `rand` and `randn`. If you run the code in this example, your results may differ from the plot shown here.

---

## Example

```
sigma = [1 0.8;0.8 1];  
r = mvtrnd(sigma,3,100);  
plot(r(:,1),r(:,2),'+')
```

**See Also**`mvnrnd`, `trnd`

# nancov

---

**Purpose** Covariance matrix, ignoring NaNs

**Syntax**

```
C = nancov(X)
C = nancov(X,Y)
C = nancov(X,Y)
C = nancov(...,1)
C = nancov(...,'pairwise')
```

**Description** `C = nancov(X)`, where `X` is a vector, returns the sample variance of the values in `X`, treating NaNs as missing values. If `X` is a matrix, in which each row is an observation and each column a variable, `nancov(X)` is the covariance matrix computing using rows of `X` that do not contain any NaN values.

`nancov(X,Y)`, where `X` and `Y` are vectors of equal length, is equivalent to `nancov([X(:) Y(:)])`, where `X(:)` and `Y(:)` are column vectors corresponding to `X` and `Y`, respectively.

`nancov(X)` or `nancov(X,Y)` normalizes the result by `N - 1` where `N` is the number of observations after removing missing values. This makes `nancov(X)` the best unbiased estimate of the covariance matrix if the observations are from a normal distribution.

`nancov(X,1)` or `nancov(X,Y,1)` normalizes the result by `N`. That is, it returns the second moment matrix of the observations about their mean.

`nancov(X,Y,0)` is the same as `nancov(X,Y)`, and `nancov(X,0)` is the same as `nancov(X)`.

`C = NANCOV(...,'pairwise')` computes `C(i,j)` using rows with no NaN values in columns `i` or `j`. The result may not be a positive definite matrix.

`C = NANCOV(...,'complete')` is the default, and it omits rows with any NaN values, even if they are not in column `i` or `j`.

The mean is removed from each column before calculating the result.

**Example** The following example generates random data having nonzero covariance between column 4 and the other columns.

```
X = randn(30,4);      % uncorrelated data
X(:,4) = sum(x,2);    % introduce correlation
X(2,3) = NaN;        % introduce one missing value
C = nancov(x)         % compute sample covariance
```

**See Also**

cov, var, nanvar

# nanmax

---

**Purpose** Maximum ignoring NaNs

**Syntax**

```
M = nanmax(A)
M = nanmax(A,B)
M = nanmax(A,[ ],dim)
[M,ndx] = nanmax(...)
```

**Description** `M = nanmax(A)` returns the maximum with NaNs treated as missing. For vectors, `nanmax(A)` is the largest non-NaN element in A. For matrices, `nanmax(A)` is a row vector containing the maximum non-NaN element from each column. For N-dimensional arrays, `nanmax` operates along the first nonsingleton dimension of X.

`M = nanmax(A,B)` returns an array of the same size as A and B, each of whose entries is the maximum of the corresponding entries of A or B. A scalar input is expanded to an array of the same size as the other input.

`M = nanmax(A,[ ],dim)` operates along the dimension `dim` of X.

`[M,ndx] = nanmax(...)` also returns the indices of the maximum values in the vector `ndx`.

## Example

```
A = magic(3);
A([1 6 8]) = [NaN NaN NaN]

A =
    NaN     1     6
     3     5    NaN
     4    NaN     2

[nmax,maxidx] = nanmax(A)

nmax =
     4     5     6

maxidx =
     3     2     1
```

**See Also** `nanmin`, `nanmean`, `nanmedian`, `nanstd`, `nansum`



**Purpose** Mean ignoring NaNs

**Syntax** `y = nanmean(X)`  
`y = nanmean(X,dim)`

**Description** `y = nanmean(X)` is the mean computed by treating NaNs as missing values.

For vectors, `nanmean(x)` is the mean of the non-NaN elements of `x`. For matrices, `nanmean(X)` is a row vector containing the mean of the non-NaN elements in each column. For N-dimensional arrays, `nanmean` operates along the first nonsingleton dimension of `X`.

`nanmean(X,dim)` takes the mean along dimension `dim` of `X`.

**Example**

```
m = magic(3);  
m([1 6 8]) = [NaN NaN NaN]
```

```
m =
```

```
NaN    1    6  
  3    5   NaN  
  4   NaN    2
```

```
nmean = nanmean(m)
```

```
nmean =
```

```
3.5000    3.0000    4.0000
```

**See Also** `nanmin`, `nanmax`, `nanmedian`, `nanstd`, `nansum`

# nanmedian

---

**Purpose** Median ignoring NaNs

**Syntax** `y = nanmedian(X)`  
`y = nanmedian(X,dim)`

**Description** `y = nanmedian(X)` is the median computed by treating NaNs as missing values. For vectors, `nanmedian(x)` is the median of the non-NaN elements of `x`. For matrices, `nanmedian(X)` is a row vector containing the median of the non-NaN elements in each column of `X`. For N-dimensional arrays, `nanmedian` operates along the first nonsingleton dimension of `X`.  
`y = nanmedian(X,dim)` takes the median along the dimension `dim` of `X`.

**Example**

```
m = magic(4);
m([1 6 9 11]) = [NaN NaN NaN NaN]

m =

     NaN     2     NaN     13
     5     NaN     10     8
     9     7     NaN     12
     4    14     15     1

nmedian = nanmedian(m)

nmedian =

     5.0000     7.0000    12.5000    10.0000
```

**See Also** `nanmin`, `nanmax`, `nanmean`, `nanstd`, `nansum`

**Purpose** Minimum ignoring NaNs

**Syntax**

```
M = nanmin(A)
M = nanmin(A,B)
M = nanmin(A,[],dim)
[M,ndx] = nanmin(...)
```

**Description** `M = nanmin(A)` is the minimum computed by treating NaNs as missing values. For vectors, `M` is the smallest non-NaN element in `A`. For matrices, `M` is a row vector containing the minimum non-NaN element from each column. For `N`-dimensional arrays, `nanmin` operates along the first nonsingleton dimension of `X`.

`M = nanmin(A,B)` returns an array of the same size as `A` and `B`, each of whose entries is the minimum of the corresponding entries of `A` or `B`. A scalar input is expanded to an array of the same size as the other input.

`M = nanmin(A,[],dim)` operates along the dimension `dim` of `X`.

`[M,ndx] = nanmin(...)` also returns the indices of the minimum values in vector `ndx`.

### Example

```
A = magic(3);
A([1 6 8]) = [NaN NaN NaN]

A =

     NaN     1     6
     3     5    NaN
     4    NaN     2

[nmin,minidx] = nanmin(A)

nmin =

     3     1     2

minidx =

     2     1     3
```

**See Also** `nanmax`, `nanmean`, `nanmedian`, `nanstd`, `nansum`

# nanstd

---

**Purpose** Standard deviation ignoring NaNs

**Syntax**

```
Y = nanstd(X)
Y = nanstd(X,1)
Y = nanstd(X,0)
Y = nanstd(X,flag,dim)
```

**Description** `Y = nanstd(X)` is the standard deviation computed by treating NaNs as missing values. For vectors, `nanstd(X)` is the standard deviation of the non-NaN elements of `X`. For matrices, `nanstd(X)` is a row vector containing the standard deviations of the non-NaN elements in each column of `X`. For N-dimensional arrays, `nanstd` operates along the first nonsingleton dimension of `X`.

`nanstd` normalizes `Y` by `N-1`, where `N` is the sample size. The result `Y` is the square root of an unbiased estimator of the variance of the population from which `X` is drawn, as long as `X` consists of independent, identically distributed samples, and data are missing at random.

`Y = nanstd(X,1)` normalizes `Y` by `N`. The result `Y` is the square root of the second moment of the sample about its mean. `nanstd(X,0)` is the same as `nanstd(X)`.

`Y = nanstd(X,flag,dim)` takes the standard deviation along the dimension `dim` of `X`. Set `flag` to 0 to normalize the result by `N-1`; set `flag` to 1 to normalize the result by `N`.

## Example

```
m = magic(3);
m([1 6 8]) = [NaN NaN NaN]

m =

     NaN     1     6
     3     5    NaN
     4    NaN     2

nstd = nanstd(m)

nstd =

     0.7071     2.8284     2.8284
```

**See Also**

nanmax, nanmin, nanmean, nanmedian, nansum

# nansum

---

**Purpose** Sum ignoring NaNs

**Syntax**  
`y = nansum(X)`  
`Y = nansum(X,dim)`

**Description** `y = nansum(X)` is the sum computed by treating NaNs as missing values. For vectors, `nansum(x)` is the sum of the non-NaN elements of `x`. For matrices, `nansum(X)` is a row vector containing the sum of the non-NaN elements in each column of `X`. For N-dimensional arrays, `nansum` operates along the first nonsingleton dimension of `X`.

`Y = nansum(X,dim)` takes the sum along dimension `dim` of `X`.

**Example**

```
m = magic(3);  
m([1 6 8]) = [NaN NaN NaN]  
  
m =  
    NaN     1     6  
     3     5    NaN  
     4    NaN     2  
  
nsum = nansum(m)  
  
nsum =  
     7     6     8
```

**See Also** `nanmax`, `nanmin`, `nanmean`, `nanmedian`, `nanstd`

**Purpose** Variance ignoring NaNs.

**Syntax**

```
Y = nanvar(X)
Y = nanvar(X,1)
Y = nanvar(X,0)
Y = nanvar(X,w)
Y = nanvar(X,w,dim)
```

**Description** `Y = nanvar(X)` returns the sample variance of the values in `X`, treating NaNs as missing values. For a vector input, `Y` is the variance of the non-NaN elements of `X`. For a matrix input, `Y` is a row vector containing the variance of the non-NaN elements in each column of `X`. For `N`-dimensional arrays, `nanvar` operates along the first nonsingleton dimension of `X`.

`nanvar` normalizes `Y` by `N-1`, where `N` is the sample size. The result `Y` is an unbiased estimator of the variance of the population from which `X` is drawn, as long as `X` consists of independent, identically distributed samples, and data are missing at random.

`Y = nanvar(X,1)` normalizes `Y` by `N`. The result `Y` is the second moment of the sample about its mean. `nanvar(X,0)` is the same as `nanvar(X)`.

`Y = nanvar(X,w)` computes the variance using the weight vector `w`. The length of `w` must equal the length of the dimension over which `nanvar` operates, and its elements must be nonnegative. `nanvar` normalizes `w` so that its sum is one.

`Y = nanvar(X,w,dim)` takes the variance along the dimension `dim` of `X`. Set `w` to `[]` to use the default normalization by `N-1`.

**See Also** `var`, `nanstd`, `nanmean`, `nanmedian`, `nanmin`, `nanmax`, `nansum`

# nbincdf

---

**Purpose** Negative binomial cumulative distribution function (cdf)

**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

$$y = F(x|r, p) = \sum_{i=0}^x \binom{r+i-1}{i} p^r q^i I_{(0,1,\dots)}(i)$$

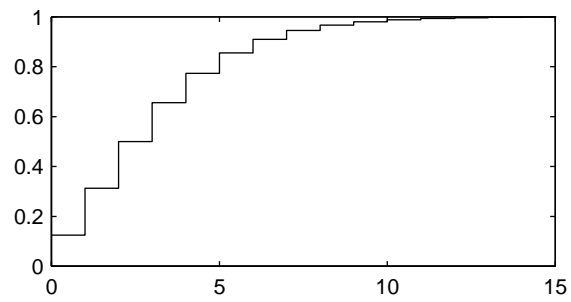
The simplest motivation for the negative binomial is the case of successive random trials, each having a constant probability `P` of success. The number of *extra* trials you must perform in order to observe a given number `R` of successes has a negative binomial distribution. However, consistent with a more general interpretation of the negative binomial, `nbincdf` allows `R` to be any positive value, including nonintegers. When `R` is noninteger, the binomial coefficient in the definition of the cdf is replaced by the equivalent expression

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

**Example**

```
x = (0:15);  
p = nbincdf(x,3,0.5);  
stairs(x,p)
```





**See Also**

`cdf`, `nbinfoit`, `nbinfoinv`, `nbinfopdf`, `nbinfornd`, `nbinfostat`

# nbinfit

---

**Purpose** Parameter estimates and confidence intervals for negative binomial data

**Syntax**

```
parmhat = nbinfit(data)
[parmhat,parmci] = nbinfit(data,alpha)
[...] = nbinfit(data,alpha,options)
```

**Description** `parmhat = nbinfit(data)` returns the maximum likelihood estimates (MLEs) of the parameters of the negative binomial distribution given the data in the vector `data`.

`[parmhat,parmci] = nbinfit(data,alpha)` returns MLEs and  $100*(1-\alpha)$  percent confidence intervals. By default,  $\alpha = 0.05$ , which corresponds to 95% confidence intervals.

`[...] = nbinfit(data,alpha,options)` accepts a structure, `options`, that specifies control parameters for the iterative algorithm the function uses to compute maximum likelihood estimates. You can create options 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`, `nbiniinv`, `nbinpdf`, `nbincrnd`, `nbinstat`, `mle`, `statset`

---

<b>Purpose</b>	Inverse of the negative binomial cumulative distribution function (cdf)
<b>Syntax</b>	$X = \text{nbininv}(Y,R,P)$
<b>Description</b>	<p><math>X = \text{nbininv}(Y,R,P)</math> returns the inverse of the negative binomial cdf with parameters <math>R</math> and <math>P</math> at the corresponding probabilities in <math>P</math>. Since the binomial distribution is discrete, <math>\text{nbininv}</math> returns the least integer <math>X</math> such that the negative binomial cdf evaluated at <math>X</math> equals or exceeds <math>Y</math>. <math>Y</math>, <math>R</math>, and <math>P</math> can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of <math>X</math>. A scalar input for <math>Y</math>, <math>R</math>, or <math>P</math> is expanded to a constant array with the same dimensions as the other inputs.</p> <p>The simplest motivation for the negative binomial is the case of successive random trials, each having a constant probability <math>P</math> of success. The number of <i>extra</i> trials you must perform in order to observe a given number <math>R</math> of successes has a negative binomial distribution. However, consistent with a more general interpretation of the negative binomial, <math>\text{nbininv}</math> allows <math>R</math> to be any positive value, including nonintegers.</p>
<b>Example</b>	<p>How many times would you need to flip a fair coin to have a 99% probability of having observed 10 heads?</p> <pre>flips = nbininv(0.99,10,0.5) + 10 flips = 33</pre> <p>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.</p>
<b>See Also</b>	<code>icdf</code> , <code>nbincdf</code> , <code>nbinfit</code> , <code>nbinpdf</code> , <code>nbinrnd</code> , <code>nbinstat</code>

# 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

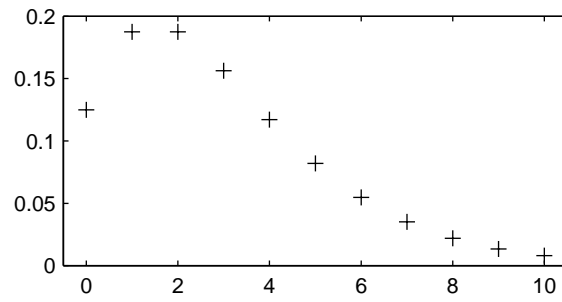
$$y = f(x|r,p) = \binom{r+x-1}{x} p^r q^x I_{(0,1,\dots)}(x)$$

The simplest motivation for the negative binomial is the case of successive random trials, each having a constant probability `P` of success. The number of *extra* trials you must perform in order to observe a given number `R` of successes has a negative binomial distribution. However, consistent with a more general interpretation of the negative binomial, `nbinpdf` allows `R` to be any positive value, including nonintegers. When `R` is noninteger, the binomial coefficient in the definition of the pdf is replaced by the equivalent expression

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

**Example**

```
x = (0:10);  
y = nbinpdf(x,3,0.5);  
plot(x,y, '+')  
set(gca, 'Xlim', [-0.5,10.5])
```



## See Also

`nbincdf`, `nbinfit`, `nbininv`, `nbinrnd`, `nbinstat`, `pdf`

# nbinrnd

---

**Purpose** Random matrices from a negative binomial distribution

**Syntax**

```
RND = nbinrnd(R,P)
RND = nbinrnd(R,P,m)
RND = nbinrnd(R,P,m,n)
```

**Description** `RND = nbinrnd(R,P)` is a matrix of random numbers chosen from a negative binomial distribution with parameters `R` and `P`. `R` and `P` can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of `RND`. A scalar input for `R` or `P` is expanded to a constant array with the same dimensions as the other input.

`RND = nbinrnd(R,P,m)` generates random numbers with parameters `R` and `P`, where `v` is a row vector. If `v` is a 1-by-2 vector, `R` is a matrix with `v(1)` rows and `v(2)` columns. If `v` is 1-by-`n`, `R` is an `n`-dimensional array.

`RND = nbinrnd(R,P,m,n)` generates random numbers with parameters `R` and `P`, where scalars `m` and `n` are the row and column dimensions of `RND`.

The simplest motivation for the negative binomial is the case of successive random trials, each having a constant probability `P` of success. The number of *extra* trials you must perform in order to observe a given number `R` of successes has a negative binomial distribution. However, consistent with a more general interpretation of the negative binomial, `nbinrnd` allows `R` to be any positive value, including nonintegers.

## Reproducing the Output of nbinrnd

`nbinrnd` uses the MATLAB functions `rand` and `randn` to generate random numbers. When you call `nbinrnd`, you change the current states of `rand` and `randn`, and thereby alter the output of subsequent calls to `nbinrnd` or any other functions that depend on `rand` or `randn`. If you want to reproduce the output of `nbinrnd`, reset the states of `rand` and `randn` to the same fixed values each time you call `nbinrnd`. For an example of how to do this, and a list of the Statistics Toolbox functions that depend on `rand` or `randn`, see “Reproducing the Output of Random Number Functions” on page 2-46.

---

**Note** The result in the following example depends on the current states of `rand` and `randn`. If you run the code in this example, your results may differ from the answer shown here.

---

**Example**

Suppose you want to simulate a process that has a defect probability of 0.01. How many units might Quality Assurance inspect before finding three defective items?

```
r = nbinrnd(3,0.01,1,6) + 3
```

```
r =
```

```
496    142    420    396    851    178
```

**See Also**

`nbincdf`, `nbinfit`, `nbininv`, `nbinpdf`, `nbinstat`

# nbinstat

---

**Purpose** Mean and variance of the negative binomial distribution

**Syntax** [M,V] = nbinstat(R,P)

**Description** [M,V] = nbinstat(R,P) returns the mean and variance of the negative binomial distribution with parameters R and P. R and P can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of M and V. A scalar input for R or P is expanded to a constant array with the same dimensions as the other input.

The mean of the negative binomial distribution with parameters  $r$  and  $p$  is  $rq/p$ , where  $q = 1-p$ . The variance is  $rq/p^2$ .

The simplest motivation for the negative binomial is the case of successive random trials, each having a constant probability P of success. The number of *extra* trials you must perform in order to observe a given number R of successes has a negative binomial distribution. However, consistent with a more general interpretation of the negative binomial, nbinstat allows R to be any positive value, including nonintegers.

## Example

```
p = 0.1:0.2:0.9;  
r = 1:5;  
[R,P] = meshgrid(r,p);  
[M,V] = nbinstat(R,P)
```

M =

9.0000	18.0000	27.0000	36.0000	45.0000
2.3333	4.6667	7.0000	9.3333	11.6667
1.0000	2.0000	3.0000	4.0000	5.0000
0.4286	0.8571	1.2857	1.7143	2.1429
0.1111	0.2222	0.3333	0.4444	0.5556

V =

90.0000	180.0000	270.0000	360.0000	450.0000
7.7778	15.5556	23.3333	31.1111	38.8889
2.0000	4.0000	6.0000	8.0000	10.0000
0.6122	1.2245	1.8367	2.4490	3.0612
0.1235	0.2469	0.3704	0.4938	0.6173



**See Also**

`nbincdf`, `nbinfit`, `nbininv`, `nbinpdf`, `nbinrnd`

# ncfcdf

**Purpose** Noncentral F cumulative distribution function (cdf)

**Syntax**  $P = \text{ncfcdf}(X, \text{NU1}, \text{NU2}, \text{DELTA})$

**Description**  $P = \text{ncfcdf}(X, \text{NU1}, \text{NU2}, \text{DELTA})$  computes the noncentral F cdf at each of the values in  $X$  using the corresponding numerator degrees of freedom in  $\text{NU1}$ , denominator degrees of freedom in  $\text{NU2}$ , and positive noncentrality parameters in  $\text{DELTA}$ .  $\text{NU1}$ ,  $\text{NU2}$ , and  $\text{DELTA}$  can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of  $P$ . A scalar input for  $X$ ,  $\text{NU1}$ ,  $\text{NU2}$ , or  $\text{DELTA}$  is expanded to a constant array with the same dimensions as the other inputs.

The noncentral F cdf is

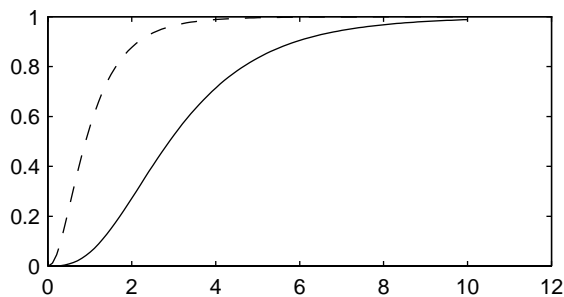
$$F(x|v_1, v_2, \delta) = \sum_{j=0}^{\infty} \left( \frac{\left(\frac{1}{2}\delta\right)^j}{j!} e^{-\frac{\delta}{2}} \right) I\left(\frac{v_1 \cdot x}{v_2 + v_1 \cdot x} \middle| \frac{v_1}{2} + j, \frac{v_2}{2}\right)$$

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

## Example

Compare the noncentral F cdf with  $\delta = 10$  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*, John Wiley and Sons, 1970. pp. 189–200.

**See Also**

`cdf`, `ncfpdf`, `ncfinv`, `ncfrnd`, `ncfstat`

**Purpose** Inverse of the noncentral F cumulative distribution function (cdf)

**Syntax** `X = ncfinv(P,NU1,NU2,DELTA)`

**Description** `X = ncfinv(P,NU1,NU2,DELTA)` returns the inverse of the noncentral F cdf with numerator degrees of freedom NU1, denominator degrees of freedom NU2, and positive noncentrality parameter DELTA for the corresponding probabilities in P. P, NU1, NU2, and DELTA can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of X. A scalar input for P, NU1, NU2, or DELTA is expanded to a constant array with the same dimensions as the other inputs.

**Example** One hypothesis test for comparing two sample variances is to take their ratio and compare it to an F distribution. If the numerator and denominator degrees of freedom are 5 and 20 respectively, then you reject the hypothesis that the first variance is equal to the second variance if their ratio is less than that computed below.

```
critical = finv(0.95,5,20)
```

```
critical =
```

```
2.7109
```

Suppose the truth is that the first variance is twice as big as the second variance. How likely is it that you would detect this difference?

```
prob = 1 - nfcdf(critical,5,20,2)
```

```
prob =
```

```
0.1297
```

If the true ratio of variances is 2, what is the typical (median) value you would expect for the F statistic?

```
ncfinv(0.5,5,20,2)
```

```
ans =
```

```
1.2786
```

**References**

- [1] Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions, Second Edition*, John Wiley and Sons, 1993. p. 102–105.
- [2] Johnson, N., and S. Kotz, *Distributions in Statistics: Continuous Univariate Distributions-2*, John Wiley and Sons, 1970. pp. 189–200.

**See Also**

icdf, nfcdf, ncfpdf, ncfrnd, ncfstat

# ncfpdf

**Purpose** Noncentral F probability density function

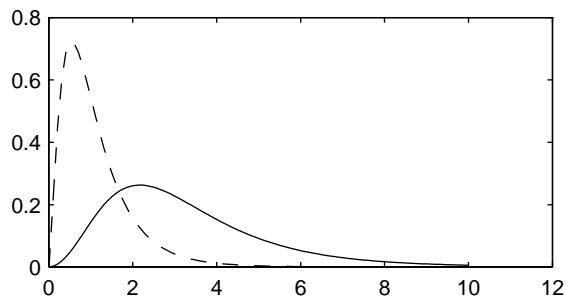
**Syntax**  $Y = \text{ncfpdf}(X, \text{NU1}, \text{NU2}, \text{DELTA})$

**Description**  $Y = \text{ncfpdf}(X, \text{NU1}, \text{NU2}, \text{DELTA})$  computes the noncentral F pdf at each of the values in  $X$  using the corresponding numerator degrees of freedom in  $\text{NU1}$ , denominator degrees of freedom in  $\text{NU2}$ , and positive noncentrality parameters in  $\text{DELTA}$ .  $X$ ,  $\text{NU1}$ ,  $\text{NU2}$ , and  $\text{DELTA}$  can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of  $Y$ . A scalar input for  $\text{NU1}$ ,  $\text{NU2}$ , or  $\text{DELTA}$  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  $\delta$  increases, the distribution flattens like the plot in the example.

**Example** Compare the noncentral F pdf with  $\delta = 10$  to the F pdf with the same number of numerator and denominator degrees of freedom (5 and 20 respectively).

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



**References** [1] Johnson, N., and S. Kotz, *Distributions in Statistics: Continuous Univariate Distributions-2*, John Wiley and Sons, 1970. pp. 189–200.

**See Also** ncfcdf, ncfinv, ncfrnd, ncfstat, pdf

**Purpose** Random matrices from the noncentral F distribution

**Syntax**

```
R = ncfrnd(NU1,NU2,DELTA)
R = ncfrnd(NU1,NU2,DELTA,v)
R = ncfrnd(NU1,NU2,DELTA,m,n)
```

**Description** `R = ncfrnd(NU1,NU2,DELTA)` returns a matrix of random numbers chosen from the noncentral F distribution with parameters `NU1`, `NU2` and `DELTA`. `NU1`, `NU2`, and `DELTA` can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of `R`. A scalar input for `NU1`, `NU2`, or `DELTA` is expanded to a constant matrix with the same dimensions as the other inputs.

`R = ncfrnd(NU1,NU2,DELTA,v)` returns a matrix of random numbers with parameters `NU1`, `NU2`, and `DELTA`, where `v` is a row vector. If `v` is a 1-by-2 vector, `R` is a matrix with `v(1)` rows and `v(2)` columns. If `v` is 1-by-`n`, `R` is an `n`-dimensional array.

`R = ncfrnd(NU1,NU2,DELTA,m,n)` generates random numbers with parameters `NU1`, `NU2`, and `DELTA`, where scalars `m` and `n` are the row and column dimensions of `R`.

### Reproducing the Output of ncfrnd

`ncfrnd` uses the MATLAB functions `rand` and `randn` to generate random numbers. When you call `ncfrnd`, you change the current states of `rand` and `randn`, and thereby alter the output of subsequent calls to `ncfrnd` or any other functions that depend on `rand` or `randn`. If you want to reproduce the output of `ncfrnd`, reset the states of `rand` and `randn` to the same fixed values each time you call `ncfrnd`. For an example of how to do this, and a list of the Statistics Toolbox functions that depend on `rand` or `randn`, see “Reproducing the Output of Random Number Functions” on page 2-46.

---

**Note** The results in the following example depend on the current states of `rand` and `randn`. If you run the code in this example, your results may differ from the answers shown here.

---

**Example** Compute six random numbers from a noncentral F distribution with 10 numerator degrees of freedom, 100 denominator degrees of freedom and a

# ncfrnd

---

noncentrality parameter,  $\delta$ , of 4.0. Compare this to the F distribution with the same degrees of freedom.

```
r = ncfrnd(10,100,4,1,6)
```

```
r =  
    2.5995    0.8824    0.8220    1.4485    1.4415    1.4864
```

```
r1 = frnd(10,100,1,6)
```

```
r1 =  
    0.9826    0.5911    1.0967    0.9681    2.0096    0.6598
```

## References

[1] Johnson, N., and S. Kotz, *Distributions in Statistics: Continuous Univariate Distributions-2*, John Wiley and Sons, 1970. pp. 189–200.

## See Also

ncfcdf, ncfinv, ncfpdf, ncfstat



**Purpose** Mean and variance of the noncentral F distribution

**Syntax** `[M,V] = ncfstat(NU1,NU2,DELTA)`

**Description** `[M,V] = ncfstat(NU1,NU2,DELTA)` returns the mean and variance of 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  $\delta$  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  $v_2 > 4$ .

**Example** `[m,v]= ncfstat(10,100,4)`

m =  
1.4286

v =  
0.4252

**References** [1] Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions, Second Edition*, John Wiley and Sons, 1993. p. 73–74.

[2] Johnson, N., and S. Kotz, *Distributions in Statistics: Continuous Univariate Distributions-2*, John Wiley and Sons, 1970. pp. 189–200.

**See Also** `ncfcdf`, `ncfinv`, `ncfpdf`, `ncfrnd`

# nctcdf

---

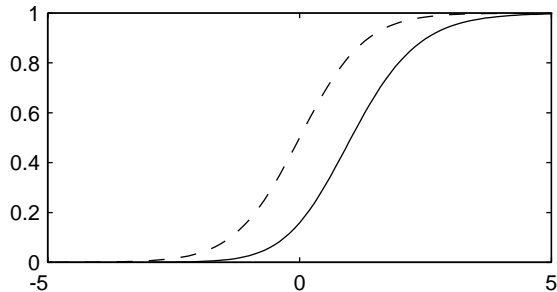
**Purpose** Noncentral T cumulative distribution function

**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.

**Example** Compare the noncentral T cdf with `DELTA = 1` to the T cdf with the same number of degrees of freedom (10).

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



**References** [1] Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions, Second Edition*, John Wiley and Sons, 1993. p. 147–148.

[2] Johnson, N., and S. Kotz, *Distributions in Statistics: Continuous Univariate Distributions-2*, John Wiley and Sons, 1970. pp. 201–219.

**See Also** `cdf`, `nctcdf`, `nctinv`, `nctpdf`, `nctrnd`, `nctstat`

---

<b>Purpose</b>	Inverse of the noncentral T cumulative distribution
<b>Syntax</b>	$X = \text{nctinv}(P, \text{NU}, \text{DELTA})$
<b>Description</b>	$X = \text{nctinv}(P, \text{NU}, \text{DELTA})$ 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.
<b>Example</b>	<pre>x = nctinv([0.1 0.2],10,1)  x =    -0.2914    0.1618</pre>
<b>References</b>	[1] Evans, M., N. Hastings, and B. Peacock, <i>Statistical Distributions, Second Edition</i> , John Wiley and Sons, 1993. p. 147–148. [2] Johnson, N., and S. Kotz, <i>Distributions in Statistics: Continuous Univariate Distributions-2</i> , John Wiley and Sons, 1970. pp. 201–219.
<b>See Also</b>	icdf, nctcdf, nctpdf, nctrnd, nctstat

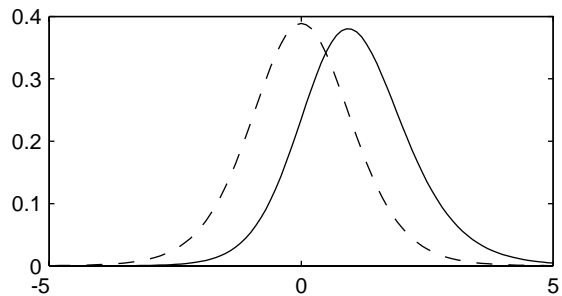
**Purpose** Noncentral T probability density function (pdf)

**Syntax** `Y = nctpdf(X,V,DELTA)`

**Description** `Y = nctpdf(X,V,DELTA)` computes the noncentral T pdf at each of the values in `X` using the corresponding degrees of freedom in `V` and noncentrality parameters in `DELTA`. Vector or matrix inputs for `X`, `V`, and `DELTA` must have the same size, which is also the size of `Y`. A scalar input for `X`, `V`, or `DELTA` is expanded to a constant matrix with the same dimensions as the other inputs.

**Example** Compare the noncentral T pdf with `DELTA = 1` to the T pdf with the same number of degrees of freedom (10).

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



**References** [1] Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions, Second Edition*, John Wiley and Sons, 1993. p. 147–148.

[2] Johnson, N., and S. Kotz, *Distributions in Statistics: Continuous Univariate Distributions-2*, John Wiley and Sons, 1970. pp. 201–219.

**See Also** `nctcdf`, `nctinv`, `nctrnd`, `nctstat`, `pdf`

**Purpose** Random matrices from noncentral T distribution

**Syntax**

```
R = nctrnd(V,DELTA)
R = nctrnd(V,DELTA,v)
R = nctrnd(V,DELTA,m,n)
```

**Description** `R = nctrnd(V,DELTA)` returns a matrix of random numbers chosen from the noncentral T distribution with parameters `V` and `DELTA`. `V` and `DELTA` can be vectors, matrices, or multidimensional arrays. A scalar input for `V` or `DELTA` is expanded to a constant array with the same dimensions as the other input.

`R = nctrnd(V,DELTA,v)` returns a matrix of random numbers with parameters `V` and `DELTA`, where `v` is a row vector. If `v` is a 1-by-2 vector, `R` is a matrix with `v(1)` rows and `v(2)` columns. If `v` is 1-by-`n`, `R` is an `n`-dimensional array.

`R = nctrnd(V,DELTA,m,n)` generates random numbers with parameters `V` and `DELTA`, where scalars `m` and `n` are the row and column dimensions of `R`.

### Reproducing the Output of nctrnd

`nctrnd` uses the MATLAB functions `rand` and `randn` to generate random numbers. When you call `nctrnd`, you change the current states of `rand` and `randn`, and thereby alter the output of subsequent calls to `nctrnd` or any other functions that depend on `rand` or `randn`. If you want to reproduce the output of `nctrnd`, reset the states of `rand` and `randn` to the same fixed values each time you call `nctrnd`. For an example of how to do this, and a list of the Statistics Toolbox functions that depend on `rand` or `randn`, see “Reproducing the Output of Random Number Functions” on page 2-46.

---

**Note** The result in the following example depends on the current states of `rand` and `randn`. If you run the code in these examples, your results may differ from the answer shown here.

---

**Example**

```
nctrnd(10,1,5,1)

ans =
```

1.6576  
1.0617  
1.4491  
0.2930  
3.6297

## References

- [1] Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions, Second Edition*, John Wiley and Sons, 1993. p. 147–148.
- [2] Johnson, N., and S. Kotz, *Distributions in Statistics: Continuous Univariate Distributions-2*, John Wiley and Sons, 1970. pp. 201–219.

## See Also

nctcdf, nctinv, nctpdf, nctstat

**Purpose** Mean and variance for the noncentral t distribution

**Syntax** [M,V] = nctstat(NU,DELTA)

**Description** [M,V] = nctstat(NU,DELTA) returns the mean and variance of 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  $\delta$  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  $v > 2$ .

**Example** [m,v] = nctstat(10,1)

m =  
1.0837

v =  
1.3255

**References** [1] Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions, Second Edition*, John Wiley and Sons, 1993. p. 147–148.

[2] Johnson, N., and S. Kotz, *Distributions in Statistics: Continuous Univariate Distributions-2*, John Wiley and Sons, 1970. pp. 201–219.

**See Also** nctcdf, nctinv, nctpdf, nctrnd

# ncx2cdf

**Purpose** Noncentral chi-square cumulative distribution function (cdf)

**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.

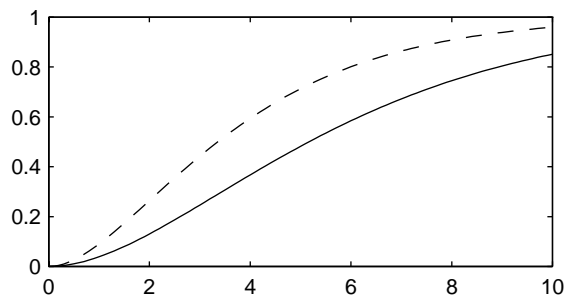
Some texts refer to this distribution as the generalized Rayleigh, Rayleigh-Rice, or Rice distribution.

The noncentral chi-square cdf is

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

## Example

```
x = (0:0.1:10)';  
p1 = ncx2cdf(x,4,2);  
p = chi2cdf(x,4);  
plot(x,p,'--',x,p1,'-')
```



## References

[1] Johnson, N., and S. Kotz, *Distributions in Statistics: Continuous Univariate Distributions-2*, John Wiley and Sons, 1970. pp. 130–148.



**See Also**

`cdf`, `ncx2inv`, `ncx2pdf`, `ncx2rnd`, `ncx2stat`

# ncx2inv

---

**Purpose** Inverse of the noncentral chi-square cdf

**Syntax** `X = ncx2inv(P,V,DELTA)`

**Description** `X = ncx2inv(P,V,DELTA)` returns the inverse of the noncentral chi-square cdf with parameters `V` and `DELTA` at the corresponding probabilities in `P`. `P`, `V`, and `DELTA` can be vectors, matrices, or multidimensional arrays that all have the same size., which is also the size of `X`. A scalar input for `P`, `V`, or `DELTA` is expanded to a constant array with the same dimensions as the other inputs.

**Algorithm** `ncx2inv` uses Newton's method to converge to the solution.

**Example**

```
ncx2inv([0.01 0.05 0.1],4,2)
ans =
    0.4858    1.1498    1.7066
```

**References** Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions, Second Edition*, John Wiley and Sons, 1993. p. 50–52.

Johnson, N., and S. Kotz, *Distributions in Statistics: Continuous Univariate Distributions-2*, John Wiley and Sons, 1970. pp. 130–148.

**See Also** `icdf`, `ncx2cdf`, `ncx2pdf`, `ncx2rnd`, `ncx2stat`

**Purpose** Noncentral chi-square probability density function (pdf)

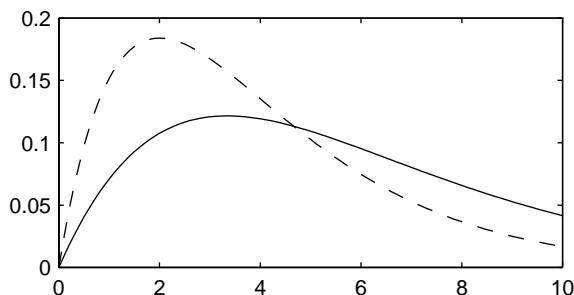
**Syntax**  $Y = \text{ncx2pdf}(X, V, \text{DELTA})$

**Description**  $Y = \text{ncx2pdf}(X, V, \text{DELTA})$  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  $\text{DELTA}$ . Vector or matrix inputs for  $X$ ,  $V$ , and  $\text{DELTA}$  must have the same size, which is also the size of  $Y$ . A scalar input for  $X$ ,  $V$ , or  $\text{DELTA}$  is expanded to a constant array with the same dimensions as the other inputs.

Some texts refer to this distribution as the generalized Rayleigh, Rayleigh-Rice, or Rice distribution.

**Example** As the noncentrality parameter  $\delta$  increases, the distribution flattens as shown in the plot.

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



**References** [1] Johnson, N., and S. Kotz, *Distributions in Statistics: Continuous Univariate Distributions-2*, John Wiley and Sons, 1970. pp. 130–148.

**See Also** `ncx2cdf`, `ncx2inv`, `ncx2rnd`, `ncx2stat`, `pdf`

# ncx2rnd

---

**Purpose** Random matrices from the noncentral chi-square distribution

**Syntax**

```
R = ncx2rnd(V,DELTA)
R = ncx2rnd(V,DELTA,v)
R = ncx2rnd(V,DELTA,m,n)
```

**Description** `R = ncx2rnd(V,DELTA)` returns a matrix of random numbers chosen from the non-central 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`.

## Reproducing the Output of ncx2rnd

`ncx2rnd` uses the MATLAB function `randn` to generate random numbers. When you call `ncx2rnd`, you change the current state of `randn`, and thereby alter the output of subsequent calls to `ncx2rnd` or any other functions that depend on `randn`. If you want to reproduce the output of `ncx2rnd`, reset the state of `randn` to the same fixed value each time you call `ncx2rnd`. For an example of how to do this, and a list of the Statistics Toolbox functions that depend on `randn`, see “Reproducing the Output of Random Number Functions” on page 2-46.

---

**Note** The result in the following example depends on the current state of `randn`. If you run the code in this example, your results may differ from the answer shown here.

---

**Example**

```
ncx2rnd(4,2,6,3)
ans =
```

6.8552	5.9650	11.2961
5.2631	4.2640	5.9495
9.1939	6.7162	3.8315
10.3100	4.4828	7.1653
2.1142	1.9826	4.6400
3.8852	5.3999	0.9282

**References**

- [1] Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions, Second Edition*, John Wiley and Sons, 1993. p. 50–52.
- [2] Johnson, N., and S. Kotz, *Distributions in Statistics: Continuous Univariate Distributions-2*, John Wiley and Sons, 1970. pp. 130–148.

**See Also**

ncx2cdf, ncx2inv, ncx2pdf, ncx2stat

# ncx2stat

---

**Purpose** Mean and variance for the noncentral chi-square distribution

**Syntax** `[M,V] = ncx2stat(NU,DELTA)`

**Description** `[M,V] = ncx2stat(NU,DELTA)` returns the mean and variance of 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  $\nu$  and  $\delta$  is  $\nu + \delta$ , and the variance is  $2(\nu + 2\delta)$ .

**Example** `[m,v] = ncx2stat(4,2)`

```
m =  
    6
```

```
v =  
   16
```

**References** [1] Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions, Second Edition*, John Wiley and Sons, 1993. p. 50–52.

[2] Johnson, N., and S. Kotz, *Distributions in Statistics: Continuous Univariate Distributions-2*, John Wiley and Sons, 1970. pp. 130–148.

**See Also** `ncx2cdf`, `ncx2inv`, `ncx2pdf`, `ncx2rnd`

**Purpose** Nonlinear least-squares regression

**Syntax**

```
beta = nlinfit(X,y,fun,beta0)
[beta,r,J] = nlinfit(X,y,fun,beta0)
[...] = nlinfit(X, y, fun, beta0, options)
```

**Description** `beta = nlinfit(X,y,fun,beta0)` estimates the coefficients of a nonlinear regression function using least squares. `y` is a vector of response (dependent variable) values. Typically, `X` is a design matrix of predictor (independent variable) values, with one row for each value in `y`. However, `X` can be any array that `fun` can accept. `fun` is a function, specified using the `@` sign, of the form

```
yhat = myfun(beta,X)
```

where `beta` is a coefficient vector. `fun` returns a vector `yhat` of fitted `y` values. `beta0` is a vector containing initial values for the coefficients.

`[beta,r,J] = nlinfit(X,y,fun,beta0)` returns the fitted coefficients, `beta`, the residuals, `r`, and the Jacobian, `J`. You can use these outputs with `nlpredci` to produce error estimates on predictions, and with `nlparci` to produce error estimates on the estimated coefficients.

---

**Note** `nlintool` provides a GUI for performing nonlinear fits and computing confidence intervals.

---

`[...] = nlinfit(X, y, fun, beta0, options)` accepts the input argument `options`, a structure that sets parameters for the algorithm used in `nlinfit`. You can create options using `statset`. The fields of options are

- `MaxIter` — Maximum number of iterations allowed. The default is 100.
- `TolFun` — Termination tolerance on the residual sum of squares. The default 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),

# nlinfit

---

- 'iter'
- 'final'
- 'DerivStep' — Relative difference used in finite difference gradient calculation. May be a scalar, or the same size as the parameter vector beta0. The default is  $\text{eps}^{(1/3)}$ .
- 'FunValCheck' — Check for invalid values, such as NaN or Inf, from the objective function. Values are 'off' or 'on' (the default).

nlinfit treats NaNs in y or fun(beta, X) as missing data and ignores the corresponding rows.

## Example

Find the coefficients that best fit the data in `reaction.mat`. The chemistry behind this data set deals with reaction kinetics as a function of the partial pressure of three chemical reactants: hydrogen, n-pentane, and isopentane.

The hougen function uses the Hougen-Watson model for reaction kinetics to return the predicted values of the reaction rate.

```
load reaction
betafit = nlinfit(reactants,rate,@hougen,beta)

betafit =

    1.2526
    0.0628
    0.0400
    0.1124
    1.1914
```

## Reference

[1] Seber, G.A.F, and C. J. Wild, *Nonlinear Regression*, Wiley, 1989.

## See Also

hougen, nlintool, nlparci, nlpredci



<b>Purpose</b>	Fits a nonlinear equation to data and displays an interactive graph
<b>Syntax</b>	<pre>nlintool(x,y,fun,beta0) nlintool(x,y,fun,beta0,alpha) nlintool(x,y,fun,beta0,alpha,'xname','yname')</pre>
<b>Description</b>	<p>nlintool displays a “vector” of plots, one for each column of the matrix of inputs, x. The response variable, y, is a column vector that matches the number of rows in x.</p> <p>nlintool(x,y,fun,beta0) is a prediction plot that provides a nonlinear curve fit to (x,y) data. It plots a 95% global confidence interval for predictions as two red curves. beta0 is a vector containing initial guesses for the parameters.</p> <p>fun is a MATLAB function that returns a vector of fitted y values. It is of the form</p> <pre>yhat = myfun(beta,x)</pre> <p>nlintool(x,y,FUN,beta0,alpha) plots a 100(1 - alpha)% confidence interval for predictions.</p> <p>The default value for alpha is 0.05, which produces 95% confidence intervals.</p> <p>nlintool(x,y,fun,beta0,alpha,'xname','yname') labels the plot using the string matrix, 'xname' for the x variables and the string 'yname' for the y variable.</p> <p>[...] = nlinfit(X, y, fun, beta0, options) accepts the input argument options, a structure that controls parameters for the algorithm used in nlinfit. You can create options using statset. The fields of options are</p> <ul style="list-style-type: none"><li>• MaxIter — Maximum number of iterations allowed. The default is 100.</li><li>• TolFun — Termination tolerance on the residual sum of squares. The default is 1e-8.</li><li>• TolX — Termination tolerance on the estimated coefficients beta. The default is 1e-8.</li><li>• Display — Level of display output during estimation. The choices are<ul style="list-style-type: none"><li>▪ 'off' (the default),</li></ul></li></ul>

# nlintool

---

- 'iter'
- 'final'

nlinfit treats NaNs in  $y$  or  $\text{fun}(\text{beta}, X)$  as missing data and ignores the corresponding rows.

## Example

See “An Interactive GUI for Nonlinear Fitting and Prediction” on page 5-6 for an example and for details about using the graphical interface.

## See Also

nlinfit, rstool

**Purpose** Confidence intervals on estimates of parameters in nonlinear models

**Syntax**

```
ci = nlparci(beta,resid,J)
ci = nlparci(beta,resid,J,alpha)
```

**Description** `nlparci(beta,resid,J)` returns the 95% confidence interval `ci` on 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 Jacobian `J`.

`ci = nlparci(beta,resid,J,alpha)` returns 100\*(1-alpha) percent 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.

**Example** Continuing the example from `nlinfit`:

```
load reaction
[beta,resid,J] = nlinfit(reactants,rate,'hougen',beta);
ci = nlparci(beta,resid,J)

ci =

    -1.0798    3.3445
    -0.0524    0.1689
    -0.0437    0.1145
    -0.0891    0.2941
    -1.1719    3.7321
```

**See Also** `nlinfit`, `nlintool`, `nlpredci`

# nlpredci

---

**Purpose** Confidence intervals on predictions of nonlinear models

**Syntax**

```
yypred = nlpredci(fun,inputs,beta,r,J)
[yypred,delta] = nlpredci(FUN,inputs,beta,r,J)
yypred = nlpredci(FUN,inputs,beta,r,J,alpha,'simopt','predopt')
```

**Description** `yypred = nlpredci(FUN,inputs,beta,r,J)` returns the predicted responses, `yypred`, given the fitted parameters `beta`, residuals `r`, and the Jacobian matrix `J`. `inputs` is a matrix of values of the independent variables in the nonlinear function.

`[yypred,delta] = nlpredci(FUN,inputs,beta,r,J)` also returns the half-width, `delta`, of confidence intervals for the nonlinear least squares predictions. The confidence interval calculation is valid for systems where the length of `r` exceeds the length of `beta` and `J` is of full column rank. The interval `[yypred-delta,yypred+delta]` is a 95% non-simultaneous confidence interval for the true value of the function at the specified input values.

`yypred = nlpredci(FUN,inputs,beta,r,J,alpha,'simopt','predopt')` controls the type of confidence intervals. The confidence level is `100(1 - alpha)%`. `'simopt'` can be `'on'` for simultaneous intervals or `'off'` (the default) for non-simultaneous intervals. `'predopt'` can be `'curve'` (the default) for confidence intervals for the function value at the inputs, or `'observation'` for confidence intervals for a new response value.

`nlpredci` uses the outputs of `nlinfit` for its inputs.

**Example** Continuing the example from `nlinfit`, you can determine the predicted function value at `[100 300 80]` and the half-width of a confidence interval for it.

```
load reaction
[beta,resids,J] = nlinfit(reactants,rate,@hougen,beta);
[yypred,delta] = nlpredci(@hougen,[100 300 80],beta,resids,J)

yypred =
    13

delta =
    1.4277
```

**See Also**      `nlinfit`, `nlintool`, `nlparci`

# normcdf

**Purpose** Normal cumulative distribution function (cdf)

**Syntax** `P = normcdf(X,MU,SIGMA)`  
`[P, PLO, PUP] = normcdf(X, MU, SIGMA, PCOV, alpha)`

**Description** `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 `normcdf` computes confidence bounds for `P` using a normal approximation to the distribution of the estimate

$$\frac{\hat{X} - \hat{\mu}}{\hat{\sigma}}$$

and then transforming those bounds to the scale of the output `P`. The computed bounds give approximately the desired confidence level when you estimate `MU`, `SIGMA`, and `PCOV` from large samples, but in smaller samples other methods of computing the confidence bounds might be more accurate.

The normal cdf is

$$p = F(x|\mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^x e^{-\frac{(t-\mu)^2}{2\sigma^2}} dt$$

The result, `p`, is the probability that a single observation from a normal distribution with parameters `μ` and `σ` will fall in the interval `(-∞ x]`.

The *standard normal* distribution has `μ = 0` and `σ = 1`.

**Examples**

What is the probability that an observation from a standard normal distribution will fall on the interval [-1 1]?

```
p = normcdf([-1 1]);
```

```
p(2) - p(1)
```

```
ans =
```

```
0.6827
```

More generally, about 68% of the observations from a normal distribution fall within one standard deviation,  $\sigma$ , of the mean,  $\mu$ .

**See Also**

[cdf](#), [normfit](#), [norminv](#), [normpdf](#), [normplot](#), [normrnd](#), [normspec](#), [normstat](#)

# normfit

---

**Purpose** Parameter estimates and confidence intervals for normal data

**Syntax**

```
[muhat, sigmahat] = normfit(DATA)
[muhat, sigmahat, mucu, sigmaci] = normfit(DATA)
[muhat, sigmahat, mucu, 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,  $\mu$ , and standard deviation,  $\sigma$ , of the normal distribution given the data in DATA.

[muhat, sigmahat, mucu, sigmaci] = normfit(DATA) returns 95% confidence intervals for the parameter estimates on the  $\mu$  and  $\sigma$  parameters in the arrays mucu and sigmaci. The first row of mucu contains the lower bounds of the confidence intervals for  $\mu$ , and the second row contains the upper bounds. The first row of sigmaci contains the lower bounds of the confidence intervals for  $\sigma$ , and the second row contains the upper bounds .

[muhat, sigmahat, mucu, 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.

[...] = 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 when there is censoring. You can create options using the function statset. Enter



`statset('normfit')` to see the names and default values of the parameters that `normfit` accepts in the options structure. See the reference page for `statset` for more information about these options.

### Example

In this example the data is a two-column random normal matrix. Both columns have  $\mu = 10$  and  $\sigma = 2$ . Note that the confidence intervals below contain the “true values.”

```
data = normrnd(10,2,100,2);
[mu,sigma,muci,sigmaci] = normfit(data)

mu =
    10.1455    10.0527

sigma =
    1.9072    2.1256

muci =
    9.7652    9.6288
   10.5258   10.4766

sigmaci =
    1.6745    1.8663
    2.2155    2.4693
```

### See Also

`normcdf`, `norminv`, `normlike`, `normpdf`, `normplot`, `normrnd`, `normspec`, `normstat`, `mle`, `statset`

# norminv

---

**Purpose** Inverse of the normal cumulative distribution function (cdf)

**Syntax**  
`X = norminv(P,MU,SIGMA)`  
`[X, XLO, XUP] = norminv(P, MU, SIGMA, PCOV, alpha)`

**Description** `X = norminv(P,MU,SIGMA)` computes the inverse of the normal cdf with parameters `MU` and `SIGMA` at the corresponding probabilities in `P`. `P`, `MU`, and `SIGMA` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs. The parameters in `SIGMA` must be positive, and the values in `P` must lie in the interval [0 1].

`[X, XLO, XUP] = norminv(P, MU, SIGMA, PCOV, alpha)` produces confidence bounds for `X` when the input parameters `MU` and `SIGMA` are estimates. `PCOV` is the covariance matrix of the estimated parameters. `alpha` specifies 100(1 - `alpha`)% confidence bounds. The default value of `alpha` is 0.05. `XLO` and `XUP` are arrays of the same size as `X` containing the lower and upper confidence bounds.

The function `norminv` computes confidence bounds for `P` using a normal approximation to the distribution of the estimate

$$\hat{\mu} + \hat{\sigma}q$$

where  $q$  is the  $P$ th quantile from a normal distribution with mean 0 and standard deviation 1. The computed bounds give approximately the desired confidence level when you estimate `MU`, `SIGMA`, and `PCOV` from large samples, but in smaller samples other methods of computing the confidence bounds may be more accurate.

The normal inverse function is defined in terms of the normal cdf as

$$x = F^{-1}(p|\mu, \sigma) = \{x:F(x|\mu, \sigma)= p\}$$

where

$$p = F(x|\mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^x e^{-\frac{(t-\mu)^2}{2\sigma^2}} dt$$

The result,  $x$ , is the solution of the integral equation above where you supply the desired probability,  $p$ .

### Examples

Find an interval that contains 95% of the values from a standard normal distribution.

```
x = norminv([0.025 0.975],0,1)
```

```
x =  
-1.9600    1.9600
```

Note that the interval  $x$  is not the only such interval, but it is the shortest.

```
x1 = norminv([0.01 0.96],0,1)
```

```
x1 =  
-2.3263    1.7507
```

The interval  $x1$  also contains 95% of the probability, but it is longer than  $x$ .

### See Also

icdf, normfit, normfit, normpdf, normplot, normrnd, normspec, normstat

# normlike

---

**Purpose** Negative normal log-likelihood function

**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  $\text{params}(1) = \text{MU}$  and  $\text{params}(2) = \text{SIGMA}$ , given the vector `data`.

`[nlogL,AVAR] = normlike(params,data)` also returns the inverse of Fisher's information matrix, `AVAR`. If the input parameter values in `params` are the maximum likelihood estimates, the diagonal elements of `AVAR` are their asymptotic variances. `AVAR` is based on the observed Fisher's information, not the expected information.

`[...] = normlike(param, data, censoring)` accepts a Boolean vector, `censoring`, of the same size as `data`, which is 1 for observations that are right-censored and 0 for observations that are observed exactly.

`[...] = normlike(param, data, censoring, freq)` accepts a frequency vector, `freq`, of the same size as `data`. The vector `freq` typically contains integer frequencies for the corresponding elements in `data`, but can contain any nonnegative values. Pass in `[]` for `censoring` to use its default value.

`normlike` is a utility function for maximum likelihood estimation.

**See Also** `betalike`, `gamlike`, `mle`, `normfit`, `wbllike`

**Purpose** Normal probability density function (pdf)

**Syntax** `Y = normpdf(X, MU, SIGMA)`

**Description** `normpdf(X, MU, SIGMA)` computes the normal pdf 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.

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  $\mu$  and standard deviation  $\sigma$ . Conversely, if  $y$  is normal with mean  $\mu$  and standard deviation  $\sigma$ , 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

`mvnpdf`, `normfit`, `norminv`, `normplot`, `normrnd`, `normspec`, `normstat`, `pdf`

# normplot

**Purpose** Normal probability plot for graphical normality testing

**Syntax**  
`normplot(X)`  
`h = normplot(X)`

**Description** `normplot(X)` displays a normal probability plot of the data in `X`. For matrix `X`, `normplot` displays a line for each column of `X`.

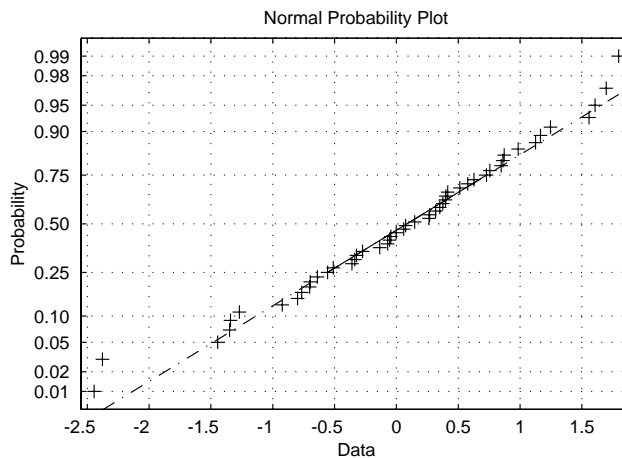
The plot has the sample data displayed with the plot symbol '+' . Superimposed on the plot is a line joining the first and third quartiles of each column of `X` (a robust linear fit of the sample order statistics.) This line is extrapolated out to the ends of the sample to help evaluate the linearity of the data.

If the data does come from a normal distribution, the plot will appear linear. Other probability density functions will introduce curvature in the plot.

`h = normplot(X)` returns a handle to the plotted lines.

**Examples** Generate a normal sample and a normal probability plot of the data.

```
x = normrnd(0,1,50,1);  
h = normplot(x);
```



The plot is linear, indicating that you can model the sample by a normal distribution.

**See Also**

`cdfplot`, `hist`, `normfit`, `normfit`, `norminv`, `normpdf`, `normrnd`, `normspec`, `normstat`

# normrnd

---

**Purpose** Generate random numbers from the normal distribution

**Syntax**

```
R = normrnd(MU, SIGMA)
R = normrnd(MU, SIGMA, v)
R = normrnd(MU, SIGMA, m, n)
```

**Description** `R = normrnd(MU, SIGMA)` generates normal random numbers with mean `MU` and standard deviation `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 normal random numbers 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 = normrnd(MU, SIGMA, m, n)` generates normal random numbers with parameters `MU` and `SIGMA`, where scalars `m` and `n` are the row and column dimensions of `R`.

## Reproducing the Output of normrnd

`normrnd` uses the MATLAB function `randn` to generate random numbers. When you call `normrnd`, you change the current state of `randn`, and thereby alter the output of subsequent calls to `normrnd` or any other functions that depend on `randn`. If you want to reproduce the output of `normrnd`, reset the state of `randn` to the same fixed value each time you call `normrnd`. For an example of how to do this, and a list of the Statistics Toolbox functions that depend on `randn`, see “Reproducing the Output of Random Number Functions” on page 2-46.

---

**Note** The results in the following examples depend on the current state of `randn`. If you run the code in these examples, your results might differ from the answers shown here.

---

**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**

normfit, normfit, norminv, normpdf, normplot, normspec, normstat

# normspec

---

**Purpose** Plot normal density between specification limits

**Syntax**  
`p = normspec(specs, mu, sigma)`  
`[p,h] = normspec(specs, mu, sigma)`

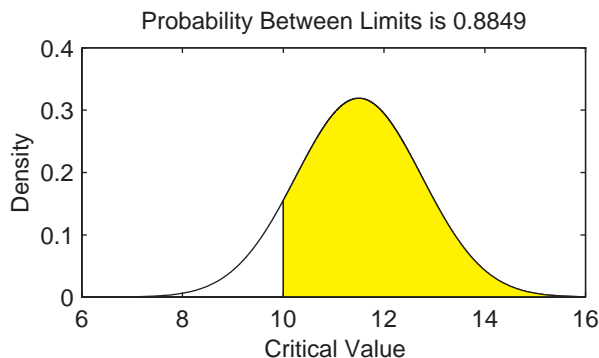
**Description**  
`p = normspec(specs, mu, sigma)` plots the normal density between a lower and upper limit defined by the two elements of the vector `specs`, where `mu` and `sigma` are the parameters of the plotted normal distribution.

`[p,h] = normspec(specs, mu, sigma)` returns the probability `p` of a sample falling between the lower and upper limits. `h` is a handle to the line objects.

If `specs(1)` is `-Inf`, there is no lower limit, and similarly if `specs(2) = Inf`, there is no upper limit.

**Example**  
Suppose a cereal manufacturer produces 10 ounce boxes of corn flakes. Variability in the process of filling each box with flakes causes a 1.25 ounce standard deviation in the true weight of the cereal in each box. The average box of cereal has 11.5 ounces of flakes. What percentage of boxes will have less than 10 ounces?

```
normspec([10 Inf], 11.5, 1.25)
```



**See Also** `capaplot`, `disttool`, `histfit`, `normfit`, `normfit`, `norminv`, `normpdf`, `normplot`, `normrnd`, `normstat`

**Purpose** Mean and variance for the normal distribution

**Syntax** `[M,V] = normstat(MU,SIGMA)`

**Description** `[M,V] = normstat(MU,SIGMA)` returns the mean 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  $\mu$  and  $\sigma$  is  $\mu$ , and the variance is  $\sigma^2$ .

### Examples

```
n = 1:5;
[m,v] = normstat(n'*n,n'*n)

m =
     1     2     3     4     5
     2     4     6     8    10
     3     6     9    12    15
     4     8    12    16    20
     5    10    15    20    25

v =
     1     4     9    16    25
     4    16    36    64   100
     9    36    81   144   225
    16    64   144   256   400
    25   100   225   400   625
```

**See Also** `normfit`, `normfit`, `norminv`, `normpdf`, `normplot`, `normrnd`, `normspec`

# parallelcoords

---

**Purpose** Parallel coordinates plot for multivariate data

**Syntax**

```
parallelcoords(X)
parallelcoords(X, ..., 'Standardize', 'on')
parallelcoords(X, ..., 'Standardize', 'PCA')
parallelcoords(X, ..., 'Standardize', 'PCAStd')
parallelcoords(X, ..., 'Quantile', alpha)
parallelcoords(X, ..., 'Group', group)
parallelcoords(X, ..., 'Labels', labs)
parallelcoords(X, ..., 'PropertyName', PropertyValue, ...)
h = parallelcoords(X, ...)
```

**Description** `parallelcoords(X)` creates a parallel coordinates plot of the multivariate data in the  $n$ -by- $p$  matrix  $X$ . Rows of  $X$  correspond to observations, columns to variables. A parallel coordinates plot is a tool for visualizing high dimensional data, where each observation is represented by the sequence of its coordinate values plotted against their coordinate indices. `parallelcoords` treats 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 the standardized principal component scores.

`parallelcoords(X, ..., 'Quantile', alpha)` plots only the median and the alpha and (1-alpha) quantiles of  $f(t)$  at each value of  $t$ . This is useful if  $X$  contains many observations.

`parallelcoords(X, ..., 'Group', group)` plots the data in different groups with different colors. Groups are defined by `group`, a numeric array containing a group index for each observation. `group` can also be a character matrix or cell array of strings, containing a group name for each observation.

`parallelcoords(X, ..., 'Labels', labs)` labels the coordinate tick marks along the horizontal axis using `labs`, 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 parameter, `h` contains one handle for each of the three lines 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 raw data
load fisheriris
labs = {'Sepal Length', 'Sepal Width', 'Petal Length', ...
       'Petal Width'};
parallelcoords(meas, 'group', species, 'labels', labs);
% plot only the median and quartiles of each group
parallelcoords(meas, 'group', species, 'labels', labs, ...
              'quantile', .25);
```

## See Also

`andrewsplot`, `glyphplot`

# pareto

---

**Purpose** Pareto charts for Statistical Process Control

**Syntax**  
`pareto(y)`  
`pareto(y,names)`  
`h = 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 `names`. `pareto(y)` labels each bar with the index of the corresponding element in `y`.

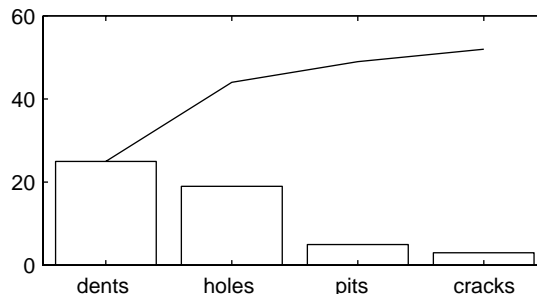
The line above the bars shows the cumulative percentage.

`pareto(y,names)` labels each bar with the row of the string matrix `names` that corresponds to the plotted element of `y`.

`h = pareto(...)` returns a combination of patch and line handles.

**Example** Create a Pareto chart from data measuring the number of manufactured parts rejected for various types of defects.

```
defects = {'pits';'cracks';'holes';'dents'};  
quantity = [5 3 19 25];  
pareto(quantity,defects)
```



**See Also** `bar`, `capaplot`, `ewmaplot`, `hist`, `histfit`, `schart`, `xbarplot`

**Purpose** Principal components analysis (PCA) using the covariance matrix

**Syntax** COEFF = pcacov(X)  
 [COEFF, latent] = pcacov(X)  
 [COEFF, latent, explained] = pcacov(X)

**Description** COEFF = pcacov(X) performs principal components analysis on the p-by-p covariance matrix X, 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 X to have unit variances. To perform principal components analysis on standardized variables, use the correlation matrix  $R = X ./ (SD * SD')$ , where  $SD = \text{sqrt}(\text{diag}(X))$ , in place of X.

[COEFF, latent] = pcacov(X) returns latent, a vector containing the principal component variances, that is, the eigenvalues of X.

[COEFF, latent, explained] = pcacov(X) returns explained, a vector containing the percentage of the total variance explained by each principal component.

**Example**

```
load hald
covx = cov(ingredients);
[COEFF, latent, explained] = pcacov(covx)

COEFF =

    0.0678    -0.6460    0.5673   -0.5062
    0.6785   -0.0200   -0.5440   -0.4933
   -0.0290    0.7553    0.4036   -0.5156
   -0.7309   -0.1085   -0.4684   -0.4844

variances =

    517.7969
     67.4964
     12.4054
      0.2372
```

explained =

86.5974

11.2882

2.0747

0.0397

## References

[1] Jackson, J. E., *A User's Guide to Principal Components*, John Wiley and Sons, Inc. 1991. pp. 1–25.

## See Also

barttest, factoran, pcares, princomp



**Purpose** Residuals from a principal components analysis

**Syntax** `residuals = pcares(X,ndim)`  
`[residuals, reconstructed] = pcares(X, ndim)`

**Description** `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, use `pcares(zscore(X), ndim)`.

`[residuals, reconstructed] = pcares(X, ndim)` returns the reconstructed observations; that is, the approximation to `X` obtained by retaining its first `ndim` principal components.

**Example** This example shows the drop in the residuals from the first row of the Hald data as the number of component dimensions increase 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
```

# pcares

---

## Reference

[1] Jackson, J. E., *A User's Guide to Principal Components*, John Wiley and Sons, Inc. 1991. pp. 1-25.

## See Also

factoran, pcacov, princomp

**Purpose** Probability density function (pdf) for a specified distribution

**Syntax** `Y = pdf('name',X,A1,A2,A3)`

**Description** `pdf('name',X,A1,A2,A3)` returns a matrix of densities, where *'name'* is a string containing the name of the distribution. *X* is a matrix of values, and *A1*, *A2*, and *A3* are matrices of distribution parameters. Depending on the distribution, some of the parameters may not be necessary.

Vector or matrix inputs for *X*, *A1*, *A2*, and *A3* must all have the same size. A scalar input is expanded to a constant matrix with the same dimensions as the other inputs.

`pdf` is a utility routine allowing access to all the pdfs in the Statistics Toolbox using the name of the distribution as a parameter. See “Overview of the Distributions” on page 2-45 for the list of available distributions.

### Examples

```
p = pdf('Normal',-2:2,0,1)
```

```
p =  
    0.0540    0.2420    0.3989    0.2420    0.0540
```

```
p = pdf('Poisson',0:4,1:5)
```

```
p =  
    0.3679    0.2707    0.2240    0.1954    0.1755
```

### See Also

`betapdf`, `binopdf`, `cdf`, `chi2pdf`, `exppdf`, `fpdf`, `gampdf`, `geopdf`, `hygepdf`, `lognpdf`, `nbinspdf`, `ncfpdf`, `nctpdf`, `ncx2pdf`, `normpdf`, `poisspdf`, `raylpdf`, `tpdf`, `unidpdf`, `unifpdf`, `wblpdf`

# pdist

---

**Purpose** Pairwise distance between observations

**Syntax**

```
Y = pdist(X)
Y = pdist(X,distance)
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  $(n - 1) \cdot n/2$ , corresponding to the  $(n - 1) \cdot n/2$  pairs of observations in  $X$ . The distances are arranged in the order (1,2), (1,3), ..., (1, $n$ ), (2,3), ..., (2, $n$ ), ..., ..., ( $n-1$ , $n$ ).  $Y$  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,distance)` computes the distance between objects in the data matrix,  $X$ , using the method specified by `distance`, where `distance` can be any of the following character strings that identify ways to compute the distance.

'euclidean'	Euclidean distance (default)
'seuclidean'	Standardized Euclidean distance. Each coordinate in the sum of squares is inverse weighted by the sample variance of that coordinate.
'mahalanobis'	Mahalanobis distance
'cityblock'	City Block metric
'minkowski'	Minkowski metric
'cosine'	One minus the cosine of the included angle between points (treated as vectors)
'correlation'	One minus the sample correlation between points (treated as sequences of values).

'spearman'	One minus the sample Spearman's rank correlation between observations, treated as sequences of values
'hamming'	Hamming distance, the percentage of coordinates that differ
'jaccard'	One minus the Jaccard coefficient, the percentage of nonzero coordinates that differ
'chebychev'	Chebychev distance (maximum coordinate difference)

`Y = pdist(X,@distfun)` accepts a function handle to a distance function of the form

$$d = \text{distfun}(u,V)$$

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  $V(k, :)$ .

Parameterizing Functions Called by Functions Functions, in the online MATLAB documentation, explains how to provide the additional parameters to the distance function, if necessary.

`Y = pdist(X, 'minkowski', p)` computes the distance between objects in the data matrix,  $X$ , using the Minkowski metric.  $p$  is the exponent used in the Minkowski computation which, by default, is 2.

### Mathematical Definitions of Methods

Given an  $m$ -by- $n$  data matrix  $X$ , which is treated as  $m$  (1-by- $n$ ) row vectors  $x_1, x_2, \dots, x_m$ , 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

$$d_{rs} = \sum_{j=1}^n |x_{rj} - x_{sj}|$$

- Minkowski metric

$$d_{rs} = \left\{ \sum_{j=1}^n |x_{rj} - x_{sj}|^p \right\}^{\frac{1}{p}}$$

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

# perms

---

**Purpose** All permutations

**Syntax** `P = perms(v)`

**Description** `P = perms(v)` where `v` is a row vector of length `n`, creates a matrix whose rows consist of all possible permutations of the `n` elements of `v`. The matrix `P` contains `n!` rows and `n` columns.

`perms` is only practical when `n` is less than 8 or 9.

**Example**

```
perms([2 4 6])
```

```
ans =
```

```
6     4     2
6     2     4
4     6     2
4     2     6
2     4     6
2     6     4
```



**Purpose** Poisson cumulative distribution function (cdf)

**Syntax** `P = poisscdf(X,LAMBDA)`

**Description** `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

$$p = F(x|\lambda) = e^{-\lambda} \sum_{i=0}^{\text{floor}(x)} \frac{\lambda^i}{i!}$$

## 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 ( $\lambda$ ) is two?

```
probability = 1 - poisscdf(4,2)
```

```
probability =
```

```
0.0527
```

About 5% of the time, a normally functioning manufacturing process will produce more than four flaws on a hard disk.

Suppose the average number of flaws ( $\lambda$ ) 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.

# poisscdf

---

## See Also

`cdf`, `poissfit`, `poissinv`, `poisspdf`, `poissrnd`, `poisstat`

**Purpose** Parameter estimates and confidence intervals for Poisson data

**Syntax**

```
lambdahat = poissfit(DATA)
[lambdahat,lambdaci] = poissfit(DATA)
[lambdahat,lambdaci] = poissfit(DATA,alpha)
```

**Description** `poissfit(DATA)` returns the maximum likelihood estimate (MLE) of the parameter of the Poisson distribution,  $\lambda$ , given the data DATA.

`[lambdahat,lambdaci] = poissfit(DATA)` also gives 95% confidence intervals in `lambdaci`.

`[lambdahat,lambdaci] = poissfit(DATA,alpha)` gives 100(1 - alpha)% confidence intervals. For example `alpha = 0.001` yields 99.9% confidence intervals.

The sample average is the MLE of  $\lambda$ .

$$\hat{\lambda} = \frac{1}{n} \sum_{i=1}^n x_i$$

**Example**

```
r = poissrnd(5,10,2);
[l,lci] = poissfit(r)
```

```
l =
    7.4000    6.3000
```

```
lci =
    5.8000    4.8000
    9.1000    7.9000
```

**See Also** `betafit`, `binofit`, `expfit`, `gamfit`, `poisscdf`, `poissinv`, `poisspdf`, `poissrnd`, `poisstat`, `unifit`, `wblfit`

# poissinv

---

**Purpose** Inverse of the Poisson cumulative distribution function (cdf)

**Syntax** `X = poissinv(P,LAMBDA)`

**Description** `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 ( $\lambda$ ) is two, what is the 95th percentile of the number of defects?

```
poissinv(0.95,2)
```

```
ans =
```

```
5
```

What is the median number of defects?

```
median_defects = poissinv(0.50,2)
```

```
median_defects =
```

```
2
```

**See Also** `icdf`, `poisscdf`, `poissfit`, `poisspdf`, `poissrnd`, `poisstat`

**Purpose** Poisson probability density function (pdf)

**Syntax** `Y = poisspdf(X,LAMBDA)`

**Description** `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

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

where  $x$  can be any nonnegative integer. The density function is zero unless  $x$  is an integer.

**Examples** A computer hard disk manufacturer has observed that flaws occur randomly in the manufacturing process at the average rate of two flaws in a 4 Gb hard disk and has found this rate to be acceptable. What is the probability that a disk will be manufactured with no defects?

In this problem,  $\lambda = 2$  and  $x = 0$ .

```
p = poisspdf(0,2)
```

```
p =  
0.1353
```

**See Also** `pdf`, `poisscdf`, `poissfit`, `poissinv`, `poissrnd`, `poisstat`

# poissrnd

---

**Purpose** Random numbers from the Poisson distribution

**Syntax**

```
R = poissrnd(LAMBDA)
R = poissrnd(LAMBDA,v)
R = poissrnd(LAMBDA,m,n)
```

**Description** `R = poissrnd(LAMBDA)` generates Poisson random numbers with mean 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 Poisson random numbers with mean LAMBDA, 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 = poissrnd(LAMBDA,m,n)` generates Poisson random numbers with mean LAMBDA, where scalars m and n are the row and column dimensions of R.

## Reproducing the Output of poissrnd

`poissrnd` uses the MATLAB functions `rand` and `randn` to generate random numbers. When you call `poissrnd`, you change the current states of `rand` and `randn`, and thereby alter the output of subsequent calls to `poissrnd` or any other functions that depend on `rand` or `randn`. If you want to reproduce the output of `poissrnd`, reset the states of `rand` and `randn` to the same fixed values each time you call `poissrnd`. For an example of how to do this, and a list of the Statistics Toolbox functions that depend on `rand` or `randn`, see “Reproducing the Output of Random Number Functions” on page 2-46.

---

**Note** The results in the following examples depend on the current states of `rand` and `randn`. If you run the code in these examples, your results may differ from the answers shown here.

---

**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

poisscdf, poissfit, poissinv, poisspdf, poisstat

# poisstat

---

**Purpose** Mean and variance for the Poisson distribution

**Syntax** `M = poisstat(LAMBDA)`  
`[M,V] = poisstat(LAMBDA)`

**Description** `M = poisstat(LAMBDA)` returns the mean of the Poisson distribution with parameter `LAMBDA`. The size of `M` is the size of `LAMBDA`.

`[M,V] = poisstat(LAMBDA)` also returns the variance `V` of the Poisson distribution.

For the Poisson distribution with parameter  $\lambda$ , both the mean and variance are equal to  $\lambda$ .

**Examples** Find the mean and variance for the Poisson distribution with  $\lambda = 2$ .

```
[m,v] = poisstat([1 2; 3 4])
```

```
m =  
    1    2  
    3    4
```

```
v =  
    1    2  
    3    4
```

**See Also** `poisscdf`, `poissfit`, `poissinv`, `poisspdf`, `poissrnd`



**Purpose** Polynomial evaluation and confidence interval estimation

**Syntax**  
`[Y,DELTA] = polyconf(p,X,S)`  
`[Y,DELTA] = polyconf(p,X,S,alpha)`

**Description** `[Y,DELTA] = polyconf(p,X,S)` uses the optional output `S` generated by `polyfit` to give 95% confidence intervals  $Y \pm \text{DELTA}$ . This assumes the errors in the data input to `polyfit` are independent normal with constant variance.

`[Y,DELTA] = polyconf(p,X,S,alpha)` gives  $100(1 - \text{alpha})\%$  confidence intervals. For example, `alpha = 0.1` yields 90% intervals.

If `p` is a vector whose elements are the coefficients of a polynomial in descending powers, such as those output from `polyfit`, then `polyconf(p,X)` is the value of the polynomial evaluated at `X`. If `X` is a matrix or vector, the polynomial is evaluated at each of the elements.

**Examples** This example gives predictions and 90% confidence intervals for computing time for LU factorizations of square matrices with 100 to 200 columns.

```
n = [100 100:20:200];
for i = n
    A = rand(i,i);
    tic
    B = lu(A);
    t(ceil((i-80)/20)) = toc;
end

[p,S] = polyfit(n(2:7),t,3);
[time,delta_t] = polyconf(p,n(2:7),S,0.1)

time =
    0.0829    0.1476    0.2277    0.3375    0.4912    0.7032

delta_t =
    0.0064    0.0057    0.0055    0.0055    0.0057    0.0064
```

# polyfit

---

**Purpose** Polynomial curve fitting

**Syntax** `[p,S] = polyfit(x,y,n)`

**Description** `p = polyfit(x,y,n)` finds the coefficients of a polynomial  $p(x)$  of degree  $n$  that fits the data,  $p(x(i))$  to  $y(i)$ , in a least-squares sense. The result  $p$  is a row vector of length  $n+1$  containing the polynomial coefficients in descending powers.

$$p(x) = p_1x^n + p_2x^{n-1} + \dots + p_nx + p_{n+1}$$

`[p,S] = polyfit(x,y,n)` returns polynomial coefficients  $p$  and matrix  $S$  for use with `polyval` to produce error estimates on predictions. If the errors in the data,  $y$ , are independent normal with constant variance, `polyval` will produce error bounds which contain at least 50% of the predictions.

You may omit  $S$  if you are not going to pass it to `polyval` or `polyconf` for calculating error estimates.

The `polyfit` function is part of the standard MATLAB language.

## Example

```
[p,S] = polyfit(1:10,[1:10] + normrnd(0,1,1,10),1)
```

```
p =
```

```
1.0300    0.4561
```

```
S =
```

```
-19.6214   -2.8031
```

```
0         -1.4639
```

```
8.0000     0
```

```
2.3180     0
```

## See Also

`polyval`, `polytool`, `polyconf`

**Purpose** Interactive plot for prediction of fitted polynomials

**Syntax**

```
polytool(x,y)
polytool(x,y,n)
polytool(x,y,n,alpha)
polytool(x,y,n,alpha,xname,yname)
h = polytool(...)
```

**Description** `polytool(x,y)` fits a line to the vectors `x` and `y` and displays an interactive plot of the result in a graphical interface. You can use the interface to explore the effects of changing the parameters of the fit and to export fit results to the workspace. See “Polynomial Curve Fitting Demo” on page 4-36 for details.

`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 order: data, fit, lower bounds, upper bounds.

**Algorithm** `polytool` fits by least-squares using the regression model

$$y_i = \beta_0 + \beta_1 x_i + \beta_2 x_i^2 + \dots + \beta_n x_i^n + \varepsilon_i$$

$$\varepsilon_i \sim N(0, \sigma^2) \quad \forall i$$

$$Cov(\varepsilon_i, \varepsilon_j) = 0 \quad \forall i, j$$

# polyval

---

**Purpose** Polynomial evaluation

**Syntax**  $Y = \text{polyval}(p,X)$   
 $[Y,DELTA] = \text{polyval}(p,X,S)$

**Description**  $Y = \text{polyval}(p,X)$  returns the predicted value of a polynomial given its coefficients,  $p$ , at the values in  $X$ .

$[Y,DELTA] = \text{polyval}(p,X,S)$  uses the optional output  $S$  generated by `polyfit` to generate error estimates,  $Y \pm DELTA$ . If the errors in the data input to `polyfit` are independent normal with constant variance,  $Y \pm DELTA$  contains at least 50% of the predictions.

If  $p$  is a vector whose elements are the coefficients of a polynomial in descending powers, then `polyval(p,X)` is the value of the polynomial evaluated at  $X$ . If  $X$  is a matrix or vector, the polynomial is evaluated at each of the elements.

The `polyval` function is part of the standard MATLAB language.

**Examples** Simulate the function  $y = x$ , adding normal random errors with a standard deviation of 0.1. Then use `polyfit` to estimate the polynomial coefficients. Note that predicted  $Y$  values are within  $DELTA$  of the integer  $X$  in every case.

```
[p,S] = polyfit(1:10,(1:10) + normrnd(0,0.1,1,10),1);  
X = magic(3);  
[Y,D] = polyval(p,X,S)
```

```
Y =  
    8.0696    1.0486    6.0636  
    3.0546    5.0606    7.0666  
    4.0576    9.0726    2.0516
```

```
D =  
    0.0889    0.0951    0.0861  
    0.0889    0.0861    0.0870  
    0.0870    0.0916    0.0916
```

**See Also** `polyfit`, `polytool`, `polyconf`

**Purpose** Percentiles of a sample

**Syntax**  
`Y = prctile(X,p)`  
`Y = prctile(X,p,dim)`

**Description** `Y = prctile(X,p)` returns percentiles of the values in `X`. `p` is a scalar or a vector of percent values. When `X` is a vector, `Y` is the same size as `p` and `Y(i)` contains the `p(i)`-th percentile. When `X` is a matrix, the `i`-th row of `Y` contains the `p(i)`-th percentiles of each column of `X`. For `N`-dimensional arrays, `prctile` operates along the first nonsingleton dimension of `X`.

`Y = prctile(X,p,dim)` calculates percentiles along dimension `dim`. The `dim`'th dimension of `Y` has length `length(p)`.

Percentiles are specified using percentages, from 0 to 100. For an  $n$ -element vector `X`, `prctile` computes percentiles as follows:

- 1** The sorted values in `X` are taken to be the  $100*(0.5/n)$ ,  $100*(1.5/n)$ , ...,  $100*((n-0.5)/n)$  percentiles.
- 2** Linear interpolation is used to compute percentiles for percent values between  $100*(0.5/n)$  and  $100*((n-0.5)/n)$ .
- 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
```

# princomp

---

**Purpose** Principal components analysis (PCA)

**Syntax**

```
COEFF = 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  $n \leq p$ , `SCORE(:,n:p)` and `latent(n:p)` are necessarily zero, and the columns of `COEFF(:,n:p)` define directions that are orthogonal to  $X$ .

`[...] = princomp(X, 'econ')` returns only the elements of `latent` that are not necessarily zero, and the corresponding columns of `COEFF` and `SCORE`, that is, when  $n \leq p$ , only the first  $n-1$ . This can be significantly faster when  $p$  is much larger than  $n$ .

**Example**

Compute principal components for the ingredients data in the Hald data set, and the variance accounted for by each component.

```
load hald;
[pc,score,latent,tsquare] = princomp(ingredients);
pc,latent

pc =
    0.0678    -0.6460     0.5673    -0.5062
    0.6785    -0.0200    -0.5440    -0.4933
   -0.0290     0.7553     0.4036    -0.5156
   -0.7309    -0.1085    -0.4684    -0.4844

latent =
    517.7969
     67.4964
     12.4054
      0.2372
```

**Reference**

- [1] Jackson, J. E., *A User's Guide to Principal Components*, John Wiley and Sons, Inc., 1991, p. 592.
- [2] Jolliffe, I.T., *Principal Component Analysis*, 2nd Ed., Springer, 2002.
- [3] Krzanowski, W.J., *Principles of Multivariate Analysis*, Oxford University Press, 1988.
- [4] Seber, G.A.F., *Multivariate Observations*, Wiley, 1984.

**See Also**

barttest, canoncorr, factoran, pcacov, pcares

# probplot

---

**Purpose** Probability plot

**Syntax**

```
probplot(Y)
probplot('distname', Y)
probplot(Y, cens, req)
probplot(ax, Y)
probplot(..., 'noref')
probplot(ax, fun, params)
h = probplot(...)
```

**Description** `probplot(Y)` produces a normal probability plot comparing the distribution of the data `Y` to the normal distribution. `Y` can be a single vector, or a matrix with a separate sample in each column. The plot includes a reference line useful for judging whether the data follow a normal distribution.

`probplot('distname', Y)` creates a probability plot for the specified distribution.

`probplot(Y, cens, freq)` or `probplot('distname', Y, cens, freq)` requires a vector `Y`. `cens` is a vector of the same size as `Y` and contains 1 for observations that are right-censored and 0 for observations that are observed exactly. `freq` is a vector of the same size as `Y`, containing integer frequencies for the corresponding elements in `Y`.

`probplot(ax, Y)` takes a handle `ax` to an existing probability plot, and adds additional lines for the samples in `Y`. `ax` is a handle for a set of axes.

`probplot(..., 'noref')` omits the reference line.

`probplot(ax, fun, params)` takes a function `fun` and a set of parameters, `params`, and adds fitted lines to the axes specified by `ax`. `fun` is a function to compute a cdf function, and is specified with `@`, for example, `@weibcdf`. `params` is the set of parameters required to evaluate `fun`, and is specified as a cell array or vector. The function must accept a vector of `X` values as its first argument, then the optional parameters, and must return a vector of cdf values evaluated at `X`.

`h = probplot(...)` returns handles to the plotted lines.

**See Also** `normplot`, `ecdf`



**Purpose** Procrustes analysis

**Syntax**  
`d = procrustes(X,Y)`  
`[d,Z] = procrustes(X,Y)`  
`[d,Z,transform] = procrustes(X,Y)`

**Description** `d = procrustes(X,Y)` determines a linear transformation (translation, reflection, orthogonal rotation, and scaling) of the points in matrix `Y` to best conform them to the points in matrix `X`. The goodness-of-fit criterion is the sum of squared errors. `procrustes` returns the minimized value of this dissimilarity measure in `d`. `d` is standardized by a measure of the scale of `X`, given by

$$\text{sum}(\text{sum}((X - \text{repmat}(\text{mean}(X,1), \text{size}(X,1), 1)) .^2, 1))$$

i.e., the sum of squared elements of a centered version of `X`. However, if `X` comprises repetitions of the same point, the sum of squared errors is not standardized.

`X` and `Y` must have the same number of points (rows), and `procrustes` matches the `i`th point in `Y` to the `i`th point in `X`. Points in `Y` can have smaller dimension (number of columns) than those in `X`. In this case, `procrustes` adds columns of zeros to `Y` as necessary.

`[d,Z] = procrustes(X,Y)` also returns the transformed `Y` values.

`[d,Z,transform] = procrustes(X,Y)` also returns the transformation that maps `Y` to `Z`. `transform` is a structure with fields:

- `c` Translation component
- `T` Orthogonal rotation and reflection component
- `b` Scale component

That is,  $Z = \text{transform.b} * Y * \text{transform.T} + \text{transform.c}$ .

**Examples** This example creates some random points in two dimensions, then rotates, scales, translates, and adds some noise to those points. It then uses `procrustes` to conform `Y` to `X`, and plots the original `X` and `Y`, and the transformed `Y`.

# procrustes

---

```
X = normrnd(0,1,[10 2]);
S = [0.5 -sqrt(3)/2; sqrt(3)/2 0.5];
Y = normrnd(0.5*X*S + 2,0.05,size(X));
[d,Z,tr] = procrustes(X,Y);
plot(X(:,1),X(:,2),'rx',...
      Y(:,1),Y(:,2),'b.',...
      Z(:,1),Z(:,2),'bx');
```

## See Also

cmdscales, factoran

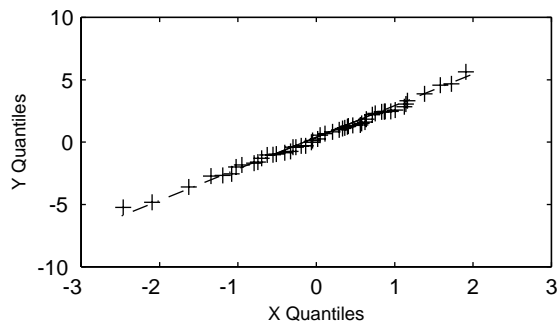
## References

- [1] Seber, G.A.F., *Multivariate Observations*, Wiley, 1984
- [2] Bulfinch, T., *The Age of Fable; or, Stories of Gods and Heroes*, Sanborn, Carter, and Bazin, Boston, 1855.

<b>Purpose</b>	Quantile-quantile plot of two samples
<b>Syntax</b>	<pre>qqplot(X) qqplot(X,Y) qqplot(X,Y,pvec) h = qqplot(...)</pre>
<b>Description</b>	<p><code>qqplot(X)</code> displays a quantile-quantile plot of the sample quantiles of <math>X</math> versus theoretical quantiles from a normal distribution. If the distribution of <math>X</math> is normal, the plot will be close to linear.</p> <p><code>qqplot(X,Y)</code> displays a quantile-quantile plot of two samples. If the samples do come from the same distribution, the plot will be linear.</p> <p>For matrix <math>X</math> and <math>Y</math>, <code>qqplot</code> displays a separate line for each pair of columns. The plotted quantiles are the quantiles of the smaller data set.</p> <p>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.</p> <p>Use <code>qqplot(X,Y,pvec)</code> to specify the quantiles in the vector <code>pvec</code>.</p> <p><code>h = qqplot(X,Y,pvec)</code> returns handles to the lines in <code>h</code>.</p>
<b>Examples</b>	<p>Generate two normal samples with different means and standard deviations. Then make a quantile-quantile plot of the two samples.</p> <pre>x = normrnd(0,1,100,1); y = normrnd(0.5,2,50,1); qqplot(x,y);</pre>

# qqplot

---



## See Also

`normplot`

**Purpose** Quantiles of a sample

**Syntax**  
`Y = quantile(X, p)`  
`Y = quantile(X, p, dim)`

**Description** `Y = quantile(X, p)` returns quantiles of the values in `X`. `p` is a scalar or a vector of cumulative probability values. When `X` is a vector, `Y` is the same size as `p`, and `Y(i)` contains the  $p(i)$ -th quantile. When `X` is a matrix, the  $i$ -th row of `Y` contains the  $p(i)$ -th quantiles of each column of `X`. For  $N$ -dimensional arrays, `quantile` operates along the first nonsingleton dimension of `X`.

`Y = quantile(X, p, dim)` calculates quantiles along dimension `dim`. The `dim`'th dimension of `Y` has length `length(P)`.

Quantiles are specified using cumulative probabilities, from 0 to 1. For an  $n$ -element vector `X`, `quantile` computes quantiles as follows:

- 1** The sorted values in `X` are taken as the  $(0.5/n)$ ,  $(1.5/n)$ , ...,  $((n-0.5)/n)$  quantiles.
- 2** Linear interpolation is used to compute quantiles for probabilities between  $(0.5/n)$  and  $((n-0.5)/n)$ .
- 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]); % a useful summary of x
```

**See Also** `prctile`, `iqr`, `median`

# random

---

**Purpose** Random numbers from a specified distribution

**Syntax** `y = random('name',A1,A2,A3,m,n)`

**Description** `y = random('name',A1,A2,A3,m,n)` returns a matrix of random numbers, where `'name'` is a string containing the name of the distribution, and `A1`, `A2`, and `A3` are matrices of distribution parameters. Depending on the distribution some of the parameters may not be necessary.

Vector or matrix inputs must all have the same size. A scalar input is expanded to a constant matrix with the same dimensions as the other inputs.

The last two parameters, `d` and `e`, are the size of the matrix `y`. If the distribution parameters are matrices, then these parameters are optional, but they must match the size of the other matrix arguments (see second example).

`random` is a utility routine allowing you to access all the random number generators in the Statistics Toolbox using the name of the distribution as a parameter. See “Overview of the Distributions” on page 2-45 for the list of available distributions.

## Examples

```
rn = random('Normal',0,1,2,4)

rn =
    1.1650    0.0751   -0.6965    0.0591
    0.6268    0.3516    1.6961    1.7971

rp = random('Poisson',1:6,1,6)

rp =
     0     0     1     2     5     7
```

## See Also

`betarnd`, `binornd`, `cdf`, `chi2rnd`, `exprnd`, `frnd`, `gamrnd`, `geornd`, `hygernd`, `icdf`, `lognrnd`, `nbinrnd`, `ncfrnd`, `nctrnd`, `ncx2rnd`, `normrnd`, `pdf`, `poissrnd`, `raylrnd`, `trnd`, `unidirnd`, `unifrnd`, `wblrnd`

<b>Purpose</b>	Random sample, with or without replacement
<b>Syntax</b>	<pre>y = randsample(n,k) y = randsample(population,k) y = randsample(...,replace) y = randsample(...,true,w)</pre>
<b>Description</b>	<p><code>y = randsample(n,k)</code> returns a 1-by-k vector <code>y</code> of values sampled uniformly at random, without replacement, from the integers 1 to <code>n</code>.</p> <p><code>y = randsample(population,k)</code> returns <code>k</code> values sampled uniformly at random, without replacement, from the values in the vector <code>population</code>.</p> <p><code>y = randsample(...,replace)</code> returns a sample taken with replacement if <code>replace</code> is <code>true</code>, or without replacement if <code>replace</code> is <code>false</code>. The default is <code>false</code>.</p> <p><code>y = randsample(...,true,w)</code> returns a weighted sample taken with replacement, using a vector of positive weights <code>w</code>, whose length is <code>n</code>. The probability that the integer <code>i</code> is selected for an entry of <code>y</code> is <math>w(i) / \text{sum}(w)</math>. Usually, <code>w</code> is a vector of probabilities. <code>randsample</code> does not support weighted sampling without replacement.</p>
<b>Example</b>	<p>The following command generates a random sequence of the characters A, C, G, and T, with replacement, according to the specified probabilities.</p> <pre>R = randsample('ACGT',48,true,[0.15 0.35 0.35 0.15])</pre>
<b>See Also</b>	<code>rand</code> , <code>randperm</code>

# randtool

---

<b>Purpose</b>	Interactive random number generation using histograms for display
<b>Syntax</b>	randtool
<b>Description</b>	randtool sets up a graphic user interface for exploring the effects of changing parameters and sample size on the histogram of random samples from the supported probability distributions. See “Random Sample Generation Demo” on page 2-4 for detailed information about the demo.
<b>See Also</b>	disttool



**Purpose** Sample range

**Syntax**  
`y = 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.

**Example** The range of a large sample of standard normal random numbers is approximately six. This is the motivation for the process capability indices  $C_p$  and  $C_{pk}$  in statistical quality control applications.

```
rv = normrnd(0,1,1000,5);
near6 = range(rv)

near6 =
    6.1451    6.4986    6.2909    5.8894    7.0002
```

**See Also** `std`, `iqr`, `mad`

# ranksum

---

**Purpose** Wilcoxon rank sum test for equal medians

**Syntax**

```
h = ranksum(x,y)
[p,h] = ranksum(x,y)
[p,h] = ranksum(x,y,'alpha',alpha)
[p,h,stats] = ranksum(...)
```

**Description** `h = ranksum(x,y)` performs a two-sided rank sum test of the hypothesis that two independent samples, in the vectors `x` and `y`, come from distributions with equal medians, and returns the p-value from the test. `p` is the probability of observing the given result, or one more extreme, by chance if the null hypothesis is true, i.e., the medians are equal. Small values of `p` cast doubt on the validity of the null hypothesis. The two sets of data are assumed to come from continuous distributions that are identical except possibly for a location shift, but are otherwise arbitrary. `x` and `y` can be different lengths.

The Wilcoxon rank sum test is equivalent to the Mann-Whitney U test.

`[p,h] = ranksum(x,y)` returns the result of the hypothesis test, performed at the 0.05 significance level, in `h`. If `h = 0`, then the null hypothesis, i.e., medians are equal, cannot be rejected at the 5% level. If `h = 1`, then the null hypothesis can be rejected at the 5% level.

`[p,h] = ranksum(x,y,'alpha',alpha)` returns the result of the hypothesis test performed at the significance level `alpha`.

`[p,h] = ranksum(...,'method',method)` computes the p-value using an exact algorithm, if you set `method` to 'exact' or a normal approximation, if you set `method` to 'approximate'.

If you omit this argument, `ranksum` uses the exact method for small samples and the approximate method for larger samples.

`[p,h,stats] = ranksum(...)` returns `stats`, a structure with one or two fields. The field 'ranksum' contains the value of the rank sum statistic. If the sample size is large, then `p` is calculated using a normal approximation and the field 'zval' contains the value of the normal (Z) statistic.

**Example**

This example tests the hypothesis of equal medians for two independent unequal-sized samples. The theoretical distributions are identical except for a shift of 0.25.

```
x = unifrnd(0,1,10,1);  
y = unifrnd(.25,1.25,15,1);  
[p,h] = ranksum(x,y,0.05)
```

```
p =  
    0.0375
```

```
h =  
    1
```

**See Also**

kruskalwallis, signrank, signtest, ttest2

**References**

- [1] Gibbons, J.D., *Nonparametric Statistical Inference*, 2nd Ed., M. Dekker, 1985.
- [2] Hollander, M. and D. A. Wolfe, *Nonparametric Statistical Methods*, Wiley, 1973.

# raylcdf

---

**Purpose** Rayleigh cumulative distribution function (cdf)

**Syntax** `P = raylcdf(X,B)`

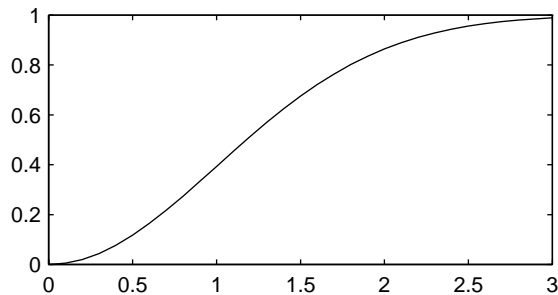
**Description** `P = raylcdf(X,B)` computes the Rayleigh cdf at each of the values in `X` using the corresponding parameters in `B`. `X` and `B` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input for `X` or `B` is expanded to a constant array with the same dimensions as the other input.

The Rayleigh cdf is

$$y = F(x|b) = \int_0^x \frac{t}{b^2} e^{\left(\frac{-t^2}{2b^2}\right)} dt$$

## Example

```
x = 0:0.1:3;  
p = raylcdf(x,1);  
plot(x,p)
```



**Reference** [1] Evans, M., N. Hastings, and B. Peacock, *Statistical Distributions, Second Edition*, Wiley 1993. pp. 134–136.

**See Also** `cdf`, `raylinv`, `raylpdf`, `raylrnd`, `raylstat`

- 
- Purpose** Parameter estimates and confidence intervals for Rayleigh data
- Syntax** `raylfit(data, alpha)`  
`[phat, pci] = raylfit(data, alpha)`
- Description** `raylfit(data, alpha)` returns the maximum likelihood estimates of the parameter of the Rayleigh distribution given the data in the vector `data`.
- `[phat, pci] = raylfit(data, alpha)` returns the maximum likelihood estimate and  $100(1 - \text{alpha})\%$  confidence interval given the data. The default value of the optional parameter `alpha` is 0.05, corresponding to 95% confidence intervals.
- See Also** `raylcdf`, `raylinv`, `raylpdf`, `raylrnd`, `raylstat`, `mle`

# raylinv

---

**Purpose** Inverse of the Rayleigh cumulative distribution function

**Syntax** `X = raylinv(P,B)`

**Description** `X = raylinv(P,B)` returns the inverse of the Rayleigh cumulative distribution function with parameter `B` at the corresponding probabilities in `P`. `P` and `B` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input for `P` or `B` is expanded to a constant array with the same dimensions as the other input.

**Example** `x = raylinv(0.9,1)`

```
x =  
    2.1460
```

**See Also** `icdf`, `raylcdf`, `raylpdf`, `raylrnd`, `raylstat`

**Purpose** Rayleigh probability density function

**Syntax** `Y = raylpdf(X,B)`

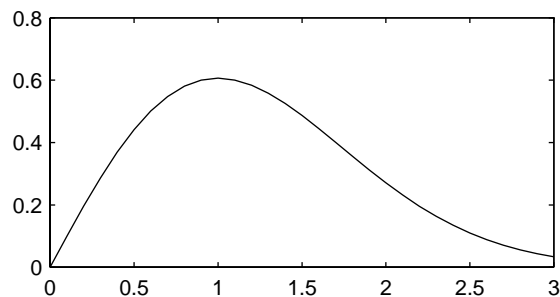
**Description** `Y = raylpdf(X,B)` computes the Rayleigh pdf at each of the values in `X` using the corresponding parameters in `B`. `X` and `B` can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of `Y`. A scalar input for `X` or `B` is expanded to a constant array with the same dimensions as the other input.

The Rayleigh pdf is

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

**Example**

```
x = 0:0.1:3;  
p = raylpdf(x,1);  
plot(x,p)
```



**See Also** `pdf`, `raylcdf`, `raylinv`, `raylrnd`, `raylstat`

# raylrnd

---

**Purpose** Random matrices from the Rayleigh distribution

**Syntax**

```
R = raylrnd(B)
R = raylrnd(B,v)
R = raylrnd(B,m,n)
```

**Description** `R = raylrnd(B)` returns a matrix of random numbers chosen from the Rayleigh distribution with parameter `B`. `B` can be a vector, a matrix, or a multidimensional array. The size of `R` is the size of `B`.

`R = raylrnd(B,v)` returns a matrix of random numbers chosen from the Rayleigh distribution with parameter `B`, where `v` is a row vector. If `v` is a 1-by-2 vector, `R` is a matrix with `v(1)` rows and `v(2)` columns. If `v` is 1-by-`n`, `R` is an `n`-dimensional array.

`R = raylrnd(B,m,n)` returns a matrix of random numbers chosen from the Rayleigh distribution with parameter `B`, where scalars `m` and `n` are the row and column dimensions of `R`.

## Reproducing the Output of raylrnd

`raylrnd` uses the MATLAB function `randn` to generate random numbers. When you call `raylrnd`, you change the current state of `randn`, and thereby alter the output of subsequent calls to `raylrnd` or any other functions that depend on `randn`. If you want to reproduce the output of `raylrnd`, reset the state of `randn` to the same fixed value each time you call `raylrnd`. For an example of how to do this, and a list of the Statistics Toolbox functions that depend on `randn`, see “Reproducing the Output of Random Number Functions” on page 2-46.

---

**Note** The result in the following example depends on the current state of `randn`. If you run the code in this example, your results may differ from the answer shown here.

---

## Example

```
r = raylrnd(1:5)

r =
    1.7986    0.8795    3.3473    8.9159    3.5182
```



**See Also**

random, raylcdf, raylinv, raylpdf, raylstat

# raylstat

---

**Purpose** Mean and variance for the Rayleigh distribution

**Syntax** M = raylstat(B)  
[M,V] = raylstat(B)

**Description** [M,V] = raylstat(B) returns the mean and variance of the Rayleigh distribution with parameter B.

The mean of the Rayleigh distribution with parameter  $b$  is  $b\sqrt{\pi/2}$  and the variance is

$$\frac{4-\pi}{2}b^2$$

**Example** [mn,v] = raylstat(1)

mn =  
1.2533

v =  
0.4292

**See Also** raylcdf, raylinv, raylpdf, raylrnd

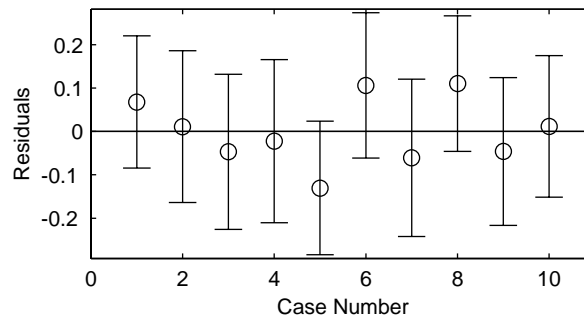
**Purpose** Residual case order plot

**Syntax** `rcoplot(r,rint)`

**Description** `rcoplot(r,rint)` displays an errorbar plot of the confidence intervals on the residuals from a regression. The residuals appear in the plot in case order. Inputs `r` and `rint` are outputs from the `regress` function.

**Example**

```
X = [ones(10,1) (1:10)'];  
y = X * [10;1] + normrnd(0,0.1,10,1);  
[b,bint,r,rint] = regress(y,X,0.05);  
rcoplot(r,rint);
```



The figure shows a plot of the residuals with error bars showing 95% confidence intervals on the residuals. All the error bars pass through the zero line, indicating that there are no outliers in the data.

**See Also** `regress`

# refcurve

---

**Purpose** Add a polynomial curve to the current plot

**Syntax** `h = refcurve(p)`

**Description** `refcurve` adds a graph of the polynomial `p` to the current axes. The function for a polynomial of degree  $n$  is:

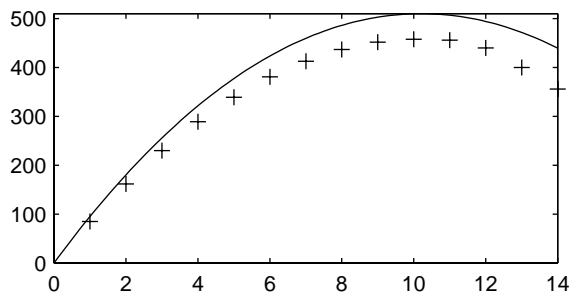
$$y = p_1x^n + p_2x^{(n-1)} + \dots + p_nx + p_{n+1}$$

Note that  $p_1$  goes with the highest order term.

`h = refcurve(p)` returns the handle to the curve.

**Example** Plot data for the height of a rocket against time, and add a reference curve showing the theoretical height (assuming no air friction). The initial velocity of the rocket is 100 m/sec.

```
h = [85 162 230 289 339 381 413 437 452 458 456 440 400 356];  
plot(h, '+')  
refcurve([-4.9 100 0])
```



**See Also** `polyfit`, `polyval`, `refline`

**Purpose** Add a reference line to the current axes

**Syntax**

```
refline(slope,intercept)
refline(slope)
h = refline(slope,intercept)
refline
```

**Description** `refline(slope,intercept)` adds a reference line with the given slope and intercept to the current axes.

`refline(slope)`, where `slope` is a two-element vector, adds the line

$$y = \text{slope}(2) + \text{slope}(1)*x$$

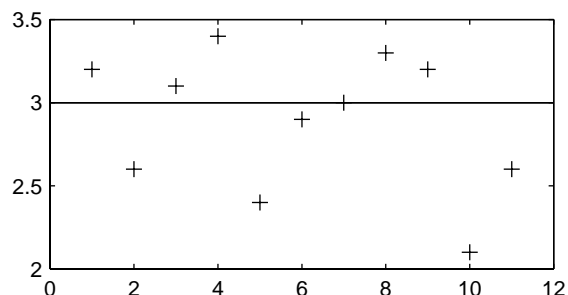
to the figure.

`h = refline(slope,intercept)` returns the handle to the line.

`refline` with no input arguments superimposes the least squares line on each line object in the current figure (except LineStyles `'-'`, `'--'`, `'.-'`). This behavior is equivalent to `lsline`.

### Example

```
y = [3.2 2.6 3.1 3.4 2.4 2.9 3.0 3.3 3.2 2.1 2.6]';
plot(y,'+')
refline(0,3)
```



**See Also** `lsline`, `polyfit`, `polyval`, `refcurve`

# regress

---

**Purpose** Multiple linear regression

**Syntax**

```
b = regress(y,X)
[b,bint,r,rint,stats] = regress(y,X)
[b,bint,r,rint,stats] = regress(y,X,alpha)
```

**Description** `b = regress(y,X)` returns the least squares fit of  $y$  on  $X$  by solving the linear model

$$y = X\beta + \varepsilon$$
$$\varepsilon \sim N(0, \sigma^2 I)$$

for  $\beta$ , where:

- $y$  is an  $n$ -by-1 vector of observations
- $X$  is an  $n$ -by- $p$  matrix of regressors
- $\beta$  is a  $p$ -by-1 vector of parameters
- $\varepsilon$  is an  $n$ -by-1 vector of random disturbances

`[b,bint,r,rint,stats] = regress(y,X)` returns an estimate of  $\beta$  in  $b$ , a 95% confidence interval for  $\beta$  in the  $p$ -by-2 vector  $bint$ . The residuals are returned in  $r$  and a 95% confidence interval for each residual is returned in the  $n$ -by-2 vector  $rint$ . The vector  $stats$  contains the  $R^2$  statistic along with the  $F$  and  $p$  values for the regression.

`[b,bint,r,rint,stats] = regress(y,X,alpha)` gives  $100(1 - \alpha)\%$  confidence intervals for  $bint$  and  $rint$ . For example,  $\alpha = 0.2$  gives 80% confidence intervals.

$X$  should include a column of ones so that the model contains a constant term. The  $F$  statistic and  $p$  value are computed under the assumption that the model contains a constant term, and they are not correct for models without a constant. The  $R$ -square value is one minus the ratio of the error sum of squares to the total sum of squares. This value can be negative for models without a constant, which indicates that the model is not appropriate for the data.

If the columns of  $X$  are linearly dependent, `regress` sets the maximum possible number of elements of  $B$  to zero to obtain a basic solution, and returns zeros in elements of  $bint$  corresponding to the zero elements of  $B$ .

regress treats NaNs in X or y as missing values, and removes them.

## Examples

Suppose the true model is

$$y = 10 + x + \varepsilon$$
$$\varepsilon \sim N(0, 0.01I)$$

where  $I$  is the identity matrix.

```
X = [ones(10,1) (1:10)']
```

```
X =
```

```
    1    1
    1    2
    1    3
    1    4
    1    5
    1    6
    1    7
    1    8
    1    9
    1   10
```

```
y = X * [10;1] + normrnd(0,0.1,10,1)
```

```
y =
```

```
11.1165
12.0627
13.0075
14.0352
14.9303
16.1696
17.0059
18.1797
19.0264
20.0872
```

```
[b,bint] = regress(y,X,0.05)
```

```
b =
```

```
10.0456
 1.0030
```

## regress

---

```
bint =  
  9.9165  10.1747  
  0.9822   1.0238
```

Compare  $b$  to  $[10 \ 1]'$ . Note that `bint` includes the true model values.

### Reference

[1] Chatterjee, S. and A. S. Hadi. *Influential Observations, High Leverage Points, and Outliers in Linear Regression*. Statistical Science, 1986. pp. 379–416.

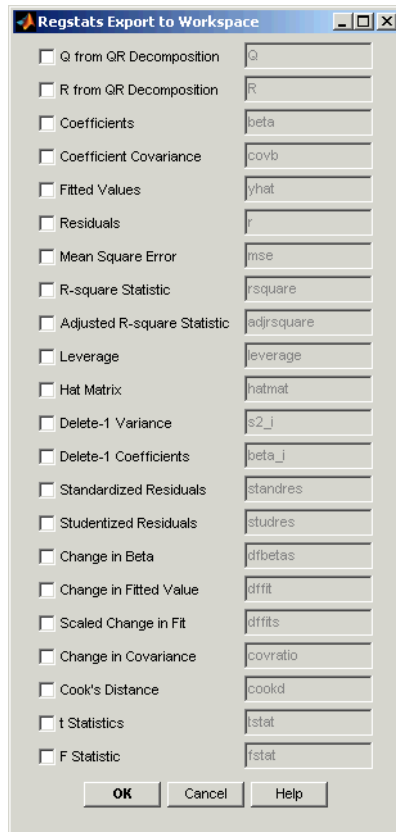


**Purpose** Regression diagnostics for linear models

**Syntax**

```
regstats(responses, DATA)
regstats(responses, DATA, model)
stats = regstats(...)
stats = regstats(responses, data, model, whichstats)
```

**Description** `regstats(responses, data)` fits a multiple linear regression of the measurements in the vector, `responses`, on the values in the matrix, `DATA`. By default, `regstats` uses a linear additive model with a constant term. With this syntax, the function creates a graphical user interface (GUI) that displays a list of diagnostic statistics, as shown in the following figure.



When you select the check boxes, corresponding to the statistics you want to compute, and click **OK**, `regstats` returns the selected statistics as variables in the MATLAB workspace. The names of the variables are displayed on the right-hand side of the GUI. For example, if you select **Coefficients** in the GUI, as shown in the following figure, `regstats` returns the regression coefficients in the variable `beta`.



`regstats(responses,data,model)` enables you to specify the order of the regression model. `model` can be one of the following strings

'linear'	Includes constant and linear terms (default).
'interaction'	Includes constant, linear, and cross product terms.
'quadratic'	Includes interactions and squared terms.
'purequadratic'	Includes constant, linear, and squared terms.

Alternatively, `model` can be a matrix of model terms as accepted by the `x2fx` function. See `x2fx` for a description of this matrix and for a description of the order in which terms appear. You can use this matrix to specify other models including ones without a constant term.

`stats = regstats(...)` creates an output structure `stats`, whose fields contain all the diagnostic statistics for the regression. This syntax does not open the GUI. The following table lists the fields of `stats`:

'Q'	Q from the QR decomposition of the design matrix, which contains the values of the model terms for each observation
'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-square statistic
'adjrsquare'	Adjusted R-square statistic
'leverage'	Leverage
'hatmat'	Hat (projection) matrix
's2_i'	Delete-1 variance
'beta_i'	Delete-1 coefficients
'standres'	Standardized residuals
'studres'	Studentized residuals
'dfbetas'	Scaled change in regression coefficients
'dffit'	Change in fitted values
'dffits'	Scaled change in fitted values
'covratio'	Change in covariance
'cookd'	Cook's distance
'tstat'	t statistics for coefficients
'fstat'	F statistic

Note that the fields names of `stats` correspond to the names of the variables returned to the MATLAB workspace when you use the GUI. For example, `stats.beta` corresponds to the variable `beta` that is returned when you select **Coefficients** in the GUI and click **OK**.

`stats = regstats(responses,DATA,model,whichstats)` returns only the statistics that you specify in `whichstats`. `whichstats` can be a single name such as `'leverage'` or a cell array of names such as `{'leverage' 'standres' 'studres'}`. Set `whichstats` to `'all'` to return all of the statistics.

The F statistic and its p-value are computed under the assumption that the model contains a constant term. They are not correct for models without a constant. The R-square value is one minus the ratio of the error sum of squares to the total sum of squares. This value can be negative for models without a constant, which indicates that the model is not appropriate for the data.

# regstats

## Example

The following commands load the array `hald` and open the `regstats` GUI using the data in `hald`.

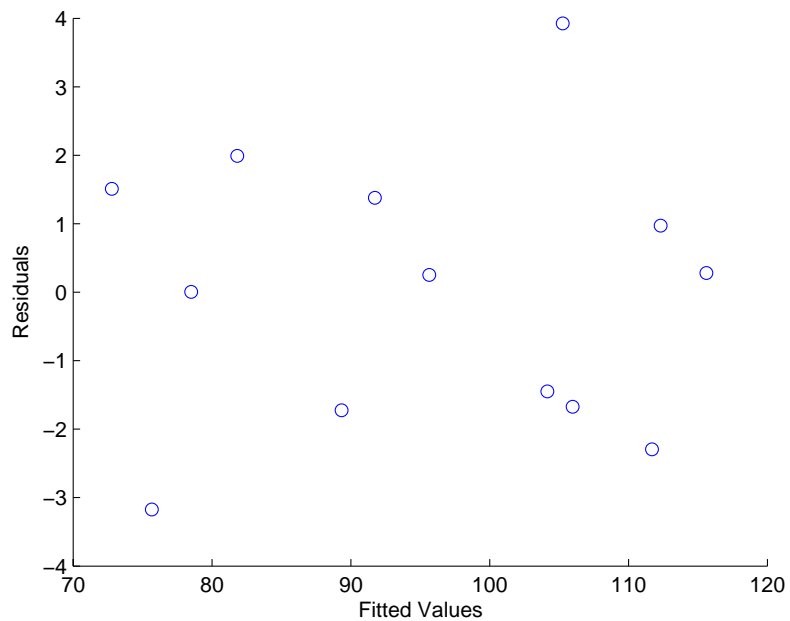
```
load hald
regstats(heat,ingredients,'linear');
```

Select **Fitted Values** and **Residuals** in the GUI.



Click **OK** to export the fitted values and residuals to the MATLAB workspace as variables named `yhat` and `r`, respectively. The following commands create a scatter plot of `yhat` and `r`.

```
scatter(yhat,r)
xlabel('Fitted Values'); ylabel('Residuals');
```



Alternatively, you can create the same plot, without using the regstats GUI, by entering the following commands.

```
whichstats = {'yhat','r'};
stats = regstats(heat,ingredients,'linear', whichstats)

stats =

    source: ''
    yhat: [13x1 double]
    r: [13x1 double]

scatter(stats.yhat,stats.r)
xlabel('Fitted Values'); ylabel('Residuals');
```

The output stats contains the fields yhat and r, corresponding to setting

```
whichstats = {'yhat','r'};
```

stats.yhat and stats.r are the same as the workspace variables yhat and r, respectively, created by the GUI.

## Algorithm

The usual regression model is  $y = X\beta + \varepsilon$ , where:

- $y$  is an  $n$ -by-1 vector of responses
- $X$  is an  $n$ -by- $p$  design matrix
- $\beta$  is an  $p$ -by-1 vector of parameters
- $\varepsilon$  is an  $n$ -by-1 vector of random disturbances

The standard textbook equation for the least squares estimator of  $\beta$  is

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

However, this definition has poor numeric properties. Particularly dubious is the computation of  $(X'X)^{-1}$ , which is both expensive and imprecise.

Let  $X = Q'R$  where  $Q$  and  $R$  come from a QR Decomposition of  $X$ .  $Q$  is orthogonal and  $R$  is triangular. Numerically stable MATLAB code for  $\beta$  is

```
beta = R\(Q'*y);
```

## Regression Diagnostics

This section describes the diagnostic statistics provided by regstats for evaluating multiple linear regression. Many regression diagnostics can be easily calculated using the QR decomposition of  $X$ .

The following sections describe the diagnostics supplied by regstats:

- “QR Decomposition (Q)”
- “QR Decomposition (R)”
- “Regression Coefficients”
- “Fitted Values of the Response”
- “Residuals”
- “Mean Squared Error”
- “R-Square Statistic”
- “Adjusted R-Square Statistic”
- “Covariance Matrix of Estimated Coefficients”
- “Hat (Projection) Matrix”
- “Leverage”
- “Delete-1 Variance”
- “Delete-1 Coefficients”
- “Standardized Residuals”
- “Studentized Residuals”
- “Scaled Change in Regression Coefficients”
- “Change in Fitted Values”
- “Scaled Change in Fitted Values”
- “Change in Covariance”
- “Cook's Distance”
- “Student's t statistics”
- “F statistic”

### QR Decomposition (Q)

Q is the first matrix from the QR Decomposition of  $X$ .

$$[Q,R] = \text{qr}(X,0)$$

This is the so-called economy-sized QR decomposition. Q is n-by-p, and its columns are orthogonal. That is,  $Q'Q = I$  (the identity matrix).

### QR Decomposition (R)

R is the second matrix from the QR Decomposition of X.

$$[Q,R] = \text{qr}(X,0)$$

This is the economy-sized QR decomposition. R is p-by-p and triangular. This makes solving linear systems simple.

### Regression Coefficients

The least squares estimator for the regression coefficients is

$$\text{beta} = R \backslash (Q' * y)$$

If you only want the coefficients and do not need to use Q and R later, then

$$\text{beta} = X \backslash y$$

is the simplest code.

### Fitted Values of the Response

Substituting the least squares estimator for beta into the model equation (leaving out the error term e) gives the fitted values.

$$\text{yhat} = X * \text{beta} = X * (R \backslash (Q' * y))$$

yhat is an n-by-1 vector of fitted (or predicted) values of y.

### Residuals

The residuals are simply the observed values minus the fitted (or predicted) values.

$$r = y - \text{yhat}$$

r is an n-by-1 vector, the same size as y.

### Mean Squared Error

The mean squared error is an estimator of the variance of the random disturbances e. This variance is assumed to be constant for all observations.

$$\text{mse} = \mathbf{r}'\mathbf{r} / (n-p)$$

where

- $\mathbf{r}$  is the vector of residuals.
- $n$  is the number of observations.
- $p$  is the number of unknown coefficients.

## R-Square Statistic

The R-square statistic is

$$\text{rsquare} = 1 - \text{sse}/\text{sst}$$

where

- $\text{sse} = \text{norm}(\mathbf{r})^2$ , where  $\mathbf{r}$  is the vector of residuals.
- $\text{sst} = \text{norm}(\mathbf{y} - \text{mean}(\mathbf{y}))^2$

## Adjusted R-Square Statistic

The adjusted R-square statistic is

$$\text{adjrsquare} = 1 - (1 - \text{rsquare}) * (n / (n - p))$$

where

- $\text{rsquare}$  is the R-square statistic.
- $n$  is the number of observations.
- $p$  is the number of unknown coefficients.

## Covariance Matrix of Estimated Coefficients

The covariance matrix for the estimated coefficients is a  $p$ -by- $p$  matrix. Its diagonal elements are the variances of the individual coefficients in  $\beta$ .

```
Rinv = R \ eye(p); % inverse of R
XtXinv = Rinv*Rinv'; % equivalent to inv(X'*X)
covb = XtXinv*mse;
```

$\text{covb}$  is proportional to  $\text{inv}(\mathbf{X}'\mathbf{X})$ , but the above calculation involving  $\mathbf{R}$  is faster and more stable.



### Hat (Projection) Matrix

The hat matrix is an  $n$ -by- $n$  matrix that projects the vector of observations,  $y$ , onto the vector of fitted values  $\hat{y}$ .

```
hatmat = Q*Q'
yhat = hatmat*y
```

### Leverage

Leverage is a measure of the effect of a particular observation on the fitted regression, due to the position of that observation in the space of the predictor variables,  $X$ . In general, the more extreme a point is in the predictor space, the more leverage it has.

```
leverage = diag(hatmat) = diag(Q*Q')
```

leverage is an  $n$ -by-1 vector containing the leverages of each observation. It is the diagonal of the hat matrix.

### Delete-1 Variance

The delete-1 variance is an  $n$ -by-1 vector. Each element contains the mean squared error of the regression obtained by deleting the corresponding observation.

```
s2_i = ((n-p)*mse - r.*r./(1-h))./(n-p-1)
```

where

- $n$  is the number of observations.
- $p$  is the number of unknown coefficients.
- $mse$  is the mean squared error.
- $r$  is the vector of residuals.
- $h$  is the leverage vector.

### Delete-1 Coefficients

The delete-1 coefficients is a  $p$ -by- $n$  matrix. Each column contains the coefficients of the regression obtained by deleting the corresponding observation.

```
b_i(:,j) = beta - Rinv*(Q(j,:) .* r(j)./(1-h(j)))'
```

where

- $R_{inv}$  is the inverse of the R matrix.
- $r$  is the vector of residuals.
- $h$  is the leverage vector.

## Standardized Residuals

The standardized residuals are the raw residuals, normalized by an estimate of their standard deviation.

```
standres = r ./ sqrt(mse*(1-h))
```

where

- $r$  is the vector of residuals.
- $mse$  is the mean squared error.
- $h$  is the leverage vector.

## Studentized Residuals

The studentized residuals are the raw residuals, normalized by an independent estimate of their standard deviation.

```
studres = r ./ sqrt(s2_i*(1-h))
```

where

- $r$  is the vector of residuals.
- $s2\_i$  is the delete-1 variance.
- $h$  is the leverage vector.

## Scaled Change in Regression Coefficients

The scaled change in regression coefficients is a  $p$ -by- $n$  matrix. Each column contains the scaled change in the estimated coefficients,  $\beta$ , caused by deleting the corresponding observation.

```
d = sqrt(diag(Rinv*Rinv'));  
dfbetas(:,j) = (beta - b_i(:,j)) ./ (sqrt(s2_i(j)).*d(j))
```

where

- `Rinv` is the inverse of the R matrix.
- `b_i` is the matrix of delete-1 coefficients.
- `s2_i` is the vector of delete-1 variances.

### Change in Fitted Values

The change in fitted values is an n-by-1 vector. Each element contains the change in a fitted value caused by deleting the corresponding observation.

```
dffit = r .* (h./(1-h))
```

where

- `r` is the vector of residuals.
- `h` is the leverage vector.

### Scaled Change in Fitted Values

The scaled change in fitted values is an n-by-1 vector. Each element contains the change in a fitted value caused by deleting the corresponding observation, scaled by the standard error.

```
dffits = studres .* sqrt(h./(1-h))
```

where

- `studres` is the vector of studentized residuals.
- `h` is the leverage vector.

### Change in Covariance

The change in covariance is an n-by-1 vector. Each element is the ratio of the generalized variance of the estimated coefficients when the corresponding element is deleted, to the generalized variance of the coefficients using all the data.

```
covr = 1 ./ (((n-p-1+studres.*studres)./(n-p)).^p).*(1-h))
```

where

- `n` is the number of observations.
- `p` is the number of unknown coefficients.
- `studres` is the vector of studentized residuals.

- $h$  is the leverage vector.

## Cook's Distance

Cook's distance is an  $n$ -by-1 vector. Each element is the normalized change in the vector of fitted values,  $\hat{y}$ , due to the deletion of the corresponding observation.

$$\text{cookd} = r \cdot r' \cdot (h ./ (1-h).^2) ./ (p \cdot \text{mse})$$

where

- $r$  is the vector of residuals.
- $h$  is the leverage vector.
- $\text{mse}$  is the mean squared error.
- $p$  is the number of unknown coefficients.

## Student's t statistics

The Student's  $t$  statistics output is a structure containing  $t$  statistics and related information. The structure contains the following fields:

- $\text{beta}$  — Regression coefficient estimates
- $\text{se}$  — Standard errors for the regression coefficient estimates
- $t$  —  $t$  statistics for the regression coefficient estimates, each one for a test that the corresponding coefficient is zero
- $\text{dfe}$  — Degrees of freedom for error
- $\text{pval}$  —  $p$ -values for each  $t$  statistic, which is calculated by the following code:

```
beta = R \ (Q' * y)
se    = sqrt(diag(covb))
t     = beta ./ se
dfe   = n - p
pval  = 2 * (tcdf(-abs(t), dfe))
```

## F statistic

The  $F$  statistic output is a structure containing an  $F$  statistic and related information. The structure contains the following fields:

- $\text{sse}$  — Error sum of squares

- `ssr` — Regression sum of squares
- `dfe` — Error degrees of freedom
- `dfr` — Regression degrees of freedom
- `f` — F statistic value, for a test that all regression coefficients other than the constant term are zero
- `pval` — p-value for the F statistic, which is calculated by the following code:

```
sse = norm(r).^2
ssr = norm(yfit - mean(yfit)).^2
dfe = n-p
dfr = p-1
f = (ssr/dfr) / (sse/dfe)
pval = 1 - fcdf(f, dfr, dfe)
```

**See Also**

leverage, stepwise, regress

**Reference**

- [1] Belsley, D.A., E. Kuh, and R.E. Welsch, *Regression Diagnostics*, New York: Wiley, 1980.
- [2] Cook, R.D., and S. Weisberg, *Residuals and Influence in Regression*, New York: Wiley, 1982.
- [3] Goodall, C. R., “Computation using the QR decomposition,” *Statistical Computing* (C. R. Rao, ed.), Handbook in Statistics, Volume 9. Amsterdam, NL Elsevier/North-Holland, 1993.

# ridge

---

**Purpose** Parameter estimates for ridge regression

**Syntax**  
`b1 = ridge(y,X,k)`  
`b0 = ridge(y,X,k,0)`

**Description** `b1 = ridge(y,X,k)` returns the vector `b1` of regression coefficients, for the linear model  $y = X\beta + \epsilon$ , obtained by performing ridge regression of the response vector `y` on the predictors `x` using ridge parameter `k`. The matrix `X` should not contain a column of ones. The results are computed after centering and scaling the `X` columns so they have mean 0 and standard deviation 1. If `y` is an `n`-by-1 vector of observations, `X` is an `n`-by-`p` matrix, and `k` is a scalar, the result `b1` is a column vector with `p` elements. If `k` has `m` elements, `b1` is `p`-by-`m`.

The ridge estimator of  $\beta$  is  $b_1 = (X'X + kI)^{-1}X'y$ .

where `X` has been centered and scaled. `b0 = RIDGE(y,X,k,0)` performs the regression without centering and scaling. The result `b0` has `p+1` coefficients, with the first being the constant term. `ridge(y,X,k,1)` is the same as `ridge(y,X,k)`.

The relationship between `b1` and `b0` is given by

$$b0 = [\text{mean}(y) - \text{mean}(X) * t; t]$$

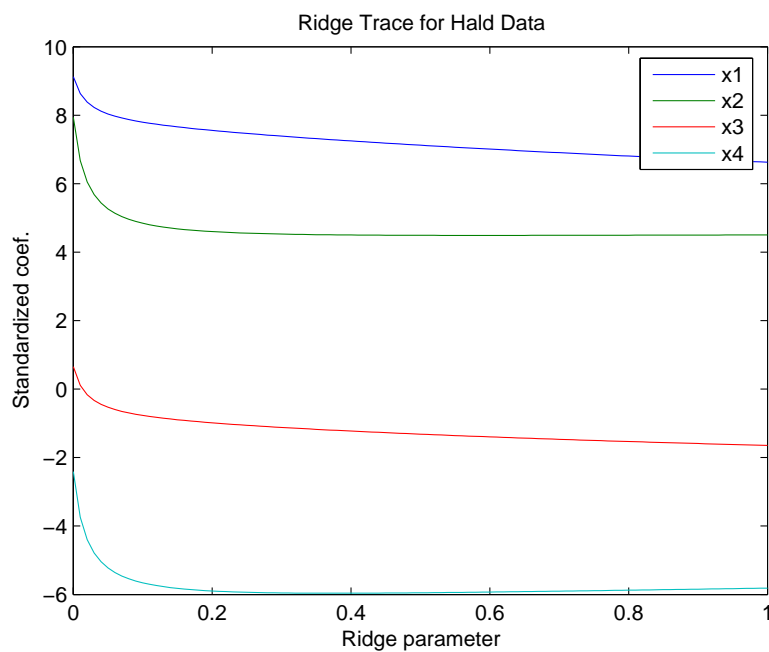
where

$$t = b1 ./ \text{std}(X, 0, 1)$$

When `k = 0`, the result `b` (either `b0` or `b1`) is the least squares estimator. For increasing `k`, the bias of `b` increases, but the variance of `b` falls. For poorly conditioned `X`, the drop in the variance more than compensates for the bias.

**Example** This example creates a ridge trace (a plot of the coefficients as a function of the ridge parameter) for the Hald data.

```
load hald
k = 0:.01:1;
b = ridge(heat, ingredients, k);
plot(k, b');
xlabel('Ridge parameter'); ylabel('Standardized coef. ');
title('Ridge Trace for Hald Data')
legend('x1', 'x2', 'x3', 'x4');
```

**See Also**

regress, stepwise

# robustdemo

---

**Purpose** Demo of robust regression

**Syntax** `robustdemo`  
`robustdemo(X,Y)`

**Description** `robustdemo` demonstrates robust regression and ordinary least squares regression on a sample data set. The function creates a figure window containing a scatter plot of sample data vectors  $X$  and  $Y$ , along with two fitted lines calculated using least squares and the robust bisquare method. The bottom of the figure shows the equations of the lines and the estimated error standard deviations for each fit. If you use the left mouse button to select a point and move it to a new location, both fits will update. If you hold down the right mouse button over any point, the point will be labeled with the leverage of that point on the least squares fit, and the weight of that point in the robust fit.

`robustdemo(X,Y)` performs the same demonstration using the  $X$  and  $Y$  values that you specify.

**Example** See “Robust Regression” on page 4-55 and “Robust Fitting Demo” on page 4-57.

**See Also** `robustfit`, `leverage`



**Purpose** Robust linear regression

**Syntax**

```
b = robustfit(X,Y)
[b,stats] = robustfit(X,Y)
[b,stats] = robustfit(X,Y, 'wfun', tune, 'const')
```

**Description** `b = robustfit(X,Y)` uses robust linear regression to fit  $Y$  as a function of the columns of  $X$ , and returns the vector  $b$  of coefficient estimates. The `robustfit` function uses an iteratively reweighted least squares algorithm, with the weights at each iteration calculated by applying the bisquare function to the residuals from the previous iteration. This algorithm gives lower weight to points that do not fit well. The results are less sensitive to outliers in the data as compared with ordinary least squares regression. `robustfit` prepends a column of ones to  $X$  to account for a constant term.

`[b,stats] = robustfit(X,Y)` also returns a `stats` structure with the following fields:

- `stats.ols_s` – sigma estimate (rmse) from least squares fit
- `stats.robust_s` – robust estimate of sigma
- `stats.mad_s` – estimate of sigma computed using the median absolute deviation of the residuals from their median; used for scaling residuals during the iterative fitting
- `stats.s` – final estimate of sigma, the larger of `robust_s` and a weighted average of `ols_s` and `robust_s`
- `stats.se` – standard error of coefficient estimates
- `stats.t` – ratio of `b` to `stats.se`
- `stats.p` – p-values for `stats.t`
- `stats.coeffcorr` – estimated correlation of coefficient estimates
- `stats.w` – vector of weights for robust fit
- `stats.h` – vector of leverage values for least squares fit
- `stats.dfe` – degrees of freedom for error
- `stats.R` – R factor in QR decomposition of  $X$  matrix

The `robustfit` function estimates the variance-covariance matrix of the coefficient estimates as  $V = \text{inv}(X' * X) * \text{stats}.s^2$ . The standard errors and correlations are derived from  $V$ .

`[b,stats] = robustfit(X,Y,'wfun',tune,'const')` specifies a weight function, a tuning constant, and the presence or absence of a constant term. The weight function `'wfun'` can be any of the names listed in the following table.

**Table 12-1:**

Weight function	Meaning	Tuning constant
'andrews'	$w = (\text{abs}(r) < \pi) .* \sin(r) ./ r$	1.339
'bisquare'	$w = (\text{abs}(r) < 1) .* (1 - r.^2).^2$	4.685
'cauchy'	$w = 1 ./ (1 + r.^2)$	2.385
'fair'	$w = 1 ./ (1 + \text{abs}(r))$	1.400
'huber'	$w = 1 ./ \max(1, \text{abs}(r))$	1.345
'logistic'	$w = \tanh(r) ./ r$	1.205
'talwar'	$w = 1 * (\text{abs}(r) < 1)$	2.795
'welsch'	$w = \exp(-(r.^2))$	2.985

The value  $r$  in the weight function expression is equal to

$$\text{resid}/(\text{tune}*s*\text{sqrt}(1-h))$$

where `resid` is the vector of residuals from the previous iteration, `tune` is the tuning constant, `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.

$$s = \text{MAD}/0.6745$$

The quantity `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 the  $X$  matrix (including the constant term, if any), the smallest  $p-1$  absolute deviations are excluded when computing their median.

In addition to the function names listed above, 'wfun' can be 'ols' to perform unweighted ordinary least squares.

The argument `tune` overrides the default tuning constant from the table. A smaller tuning constant tends to downweight large residuals more severely, and a larger tuning constant downweights large residuals less severely. The default tuning constants, shown in the table, yield coefficient estimates that are approximately 95% as efficient as least squares estimates, when the response has a normal distribution with no outliers. The value of '`const`' can be 'on' (the default) to add a constant term or 'off' to omit it. If you want a constant term, you should set '`const`' to 'on' rather than adding a column of ones to your  $X$  matrix.

As an alternative to specifying one of the named weight functions shown above, you can write your own weight function that takes a vector of scaled residuals as input and produces a vector of weights as output. You can specify '`wfun`' using `@` (for example, `@myfun`) or as an inline function.

## Example

Let's see how a single erroneous point affects least squares and robust fits. First you generate a simple data set following the equation  $y = 10 - 2x$  plus some random noise. Then you change one  $y$  value to simulate an outlier that could be an erroneous measurement.

```
x = (1:10)';
y = 10 - 2*x + randn(10,1);
y(10) = 0;
```

you use both ordinary least squares and robust fitting to estimate the equations of a straight line fit.

```
b1s = regress(y,[ones(10,1) x])

b1s =

    8.6305
   -1.4721

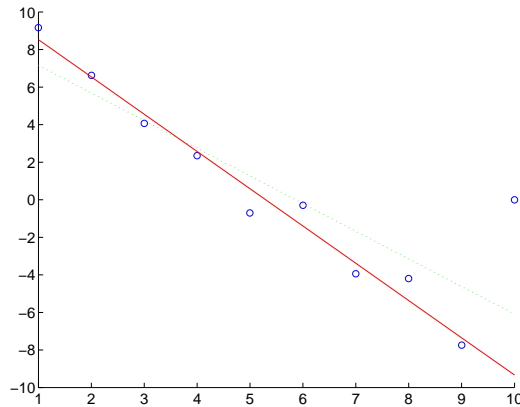
brob = robustfit(x,y)

brob =

    10.5089
   -1.9844
```

A scatter plot with both fitted lines shows that the robust fit (solid line) fits most of the data points well but ignores the outlier. The least squares fit (dotted line) is pulled toward the outlier.

```
scatter(x,y)
hold on
plot(x,bls(1)+bls(2)*x,'g:')
plot(x,brob(1)+brob(2)*x,'r-')
```



## See Also

regress, robustdemo

## References

- [1] DuMouchel, W.H., and F.L. O'Brien (1989), "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.
- [2] Holland, P.W., and R.E. Welsch (1977), "Robust Regression Using Iteratively Reweighted Least-Squares," *Communications in Statistics: Theory and Methods*, A6, 813-827.
- [3] Huber, P.J. (1981), *Robust Statistics*, New York: Wiley.
- [4] Street, J.O., R.J. Carroll, and D. Ruppert (1988), "A Note on Computing Robust Regression Estimates via Iteratively Reweighted Least Squares," *The American Statistician*, 42, 152-154

**Purpose** Rotation of factor analysis or principal components analysis loadings

**Syntax**

```

B = rotatefactors(A)
B = rotatefactors(A, 'Method', 'orthomax', 'Coeff', gamma)
B = rotatefactors(A, 'Method', 'procrustes', 'Target', target)
B = rotatefactors(A, 'Method', 'pattern', 'Target', target)
B = rotatefactors(A, 'Method', 'promax')
[B, T] = rotatefactors(A, ...)
```

**Description** `B = rotatefactors(A)` rotates the  $d$ -by- $m$  loadings matrix  $A$  to maximize the varimax criterion, and returns the result in  $B$ . Rows of  $A$  and  $B$  correspond to variables and columns correspond to factors, for example, the  $(i, j)$ th element of  $A$  is the coefficient for the  $i$ -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

$$\sum(D * \sum(B.^4, 1) - \text{GAMMA} * \sum(B.^2, 1).^2)$$

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.

# rotatefactors

---

`B = rotatefactors(A, 'Method', 'procrustes', 'Target', target)` performs an oblique procrustes rotation of  $A$  to the  $d$ -by- $m$  target loadings matrix `target`.

`B = rotatefactors(A, 'Method', 'pattern', 'Target', target)` performs an oblique rotation of the loadings matrix  $A$  to the  $d$ -by- $m$  target pattern matrix `target`, and returns the result in  $B$ . `target` defines the "restricted" elements of  $B$ , i.e., elements of  $B$  corresponding to zero elements of `target` are constrained to have small magnitude, while elements of  $B$  corresponding to nonzero elements of `target` are allowed to take on any magnitude.

If 'Method' is 'procrustes' or 'pattern', an additional parameter is 'Type', the type of rotation. If 'Type' is 'orthogonal', the rotation is orthogonal, and the factors remain uncorrelated. If 'Type' is 'oblique' (the default), the rotation is oblique, and the rotated factors might be correlated.

When 'Method' is 'pattern', there are restrictions on `target`. If  $A$  has  $m$  columns, then for orthogonal rotation, the  $j$ th column of `target` must contain at least  $m - j$  zeros. For oblique rotation, each column of `target` must contain at least  $m - 1$  zeros.

`B = rotatefactors(A, 'Method', 'promax')` rotates  $A$  to maximize the promax criterion, equivalent to an oblique Procrustes rotation with a target created by an orthomax rotation. Use the four orthomax parameters to control the orthomax rotation used internally by promax.

An additional parameter for 'promax' is 'Power', the exponent for creating promax target matrix. 'Power' must be 1 or greater. The default is 4.

`[B, T] = rotatefactors(A, ...)` returns the rotation matrix  $T$  used to create  $B$ , that is,  $B = A * T$ . `inv(T' * T)` is the correlation matrix of the rotated factors. For orthogonal rotation, this is the identity matrix, while for oblique rotation, it has unit diagonal elements but nonzero off-diagonal elements.

## Examples

```
X = randn(100,10);
L = princomp(X);

% Default (normalized varimax) rotation of the first three
% components from a PCA.
[L1,T] = rotatefactors(L(:,1:3));
```

```
% Equamax rotation of the first three components from a PCA.
[L2, T] = rotatefactors(L(:,1:3),'method','equamax');

% Promax rotation of the first three factors from an FA.
L = factoran(X,3,'Rotate','none');
[L3, T] = rotatefactors(L,'method','promax','power',2);

% Pattern rotation of the first three factors from an FA.
Tgt = [1 1 1 1 1 0 1 0 1; 0 0 0 1 1 1 0 0 0; 1 0 0 1 0 1 1 1 1]';
[L4,T] = rotatefactors(L,'method','pattern','target',Tgt);
inv(T'*T) % the correlation matrix of the rotated factors
```

## References

- [1] Harman, H.H., *Modern Factor Analysis*, Third Ed., University of Chicago Press, 1976.
- [2] Lawley, D.N. and A. E. Maxwell, A.E., *Factor Analysis as a Statistical Method*, Second Ed., American Elsevier Pub. Co., 1971.

## See Also

biplot, factoran, princomp, pcacov, procrustes

# rowexch

---

**Purpose** D-optimal design of experiments – row exchange algorithm

**Syntax**

```
settings = rowexch(nfactors,nruns)
[settings,X] = rowexch(nfactors,nruns)
[settings,X] = rowexch(nfactors,nruns,'model')
[settings,X] = rowexch(...,'param1',value1,'param2',value2,...)
```

**Description** `settings = rowexch(nfactors,nruns)` generates the factor settings matrix, `settings`, for a D-Optimal design using a linear additive model with a constant term. `settings` has `nruns` rows and `nfactors` columns.

`[settings,X] = rowexch(nfactors,nruns)` also generates the associated matrix `X` of term settings, often called the design matrix.

`[settings,X] = rowexch(nfactors,nruns,'model')` produces a design for fitting a specified regression model. The input, `'model'`, can be one of these strings:

<code>'linear'</code>	Includes constant and linear terms (the default)
<code>'interaction'</code>	Includes constant, linear, and cross product terms.
<code>'quadratic'</code>	Includes interactions plus squared terms.
<code>'purequadratic'</code>	Includes constant, linear and squared terms.

`[settings,X] = rowexch(...,'param1',value1,'param2',value2,...)` provides more control over the design generation through a set of parameter/value pairs. Valid parameters are:

<code>'display'</code>	Either <code>'on'</code> or <code>'off'</code> to control display of iteration counter. The default is <code>'on'</code> .
<code>'init'</code>	Initial design as an <code>nruns</code> -by- <code>nfactors</code> matrix. The default is a randomly selected set of points.
<code>'maxiter'</code>	Maximum number of iterations. The default is 10.

**Example** This example illustrates that the D-optimal design for three factors in eight runs, using an interactions model, is a two level full-factorial design.



```
s = rowexch(3,8,'interaction')
```

```
s =  
  -1    -1     1  
   1    -1    -1  
   1    -1     1  
  -1    -1    -1  
  -1     1     1  
   1     1     1  
  -1     1    -1  
   1     1    -1
```

### Algorithm

The `rowexch` function searches for a D-optimal design using a row-exchange algorithm. It first generates a candidate set of points that are eligible to be included in the design, and then iteratively exchanges design points for candidate points in an attempt to reduce the variance of the coefficients that would be estimated using this design. If you need to use a candidate set that differs from the default one, call the `candgen` and `candexch` functions in place of `rowexch`.

### See Also

`bbdesign`, `candexch`, `candgen`, `ccdesign`, `cordexch`, `x2fx`

# rsmdemo

---

**Purpose** Demo of design of experiments and surface fitting

**Syntax** rsmdemo

**Description** rsmdemo creates a GUI that simulates a chemical reaction. To start, you have a budget of 13 test reactions. Try to find out how changes in each reactant affect the reaction rate. Determine the reactant settings that maximize the reaction rate. Estimate the run-to-run variability of the reaction. Now run a designed experiment using the model pop-up. Compare your previous results with the output from response surface modeling or nonlinear modeling of the reaction. The GUI has the following elements:

- A **Run** button to perform one reactor run at the current settings
- An **Export** button to export the  $x$  and  $y$  data to the base workspace
- Three sliders with associated data entry boxes to control the partial pressures of the chemical reactants: Hydrogen, n-Pentane, and Isopentane
- A text box to report the reaction rate
- A text box to keep track of the number of test reactions you have left

**Example** See “Design of Experiments Demo” on page 10-10.

**See Also** rstool, nlintool, cordexch

**Purpose** Interactive fitting and visualization of a response surface

**Syntax**

```
rstool(X,Y)
rstool(X,Y,model)
rstool(X,Y,model,alpha,'xname','yname')
```

**Description** `rstool(X,Y)` opens an interactive GUI for fitting and visualizing a polynomial response surface for a response variable  $Y$  as a function of the multiple predictor variables in  $X$ . Columns of  $X$  correspond to variables, rows to observations.  $Y$  can be a vector, corresponding to a single response, or a matrix, with columns corresponding to multiple responses.  $Y$  must have as many elements (or rows if it is a matrix) as  $X$  has rows. `rstool` displays a "vector" of plots, one for each column of the matrix of inputs  $X$ . `rstool` plots a 95% global confidence interval for predictions as two red curves.

`rstool(x,y,model)` enables you to control the initial regression model, where `model` can be one of the following strings:

- 'linear' – includes constant and first order terms only
- 'purequadratic' – includes constant, linear and squared terms
- 'interaction' – includes constant, linear, and cross product terms
- 'quadratic' – includes interactions and squared terms

Alternatively, `model` can be a matrix of model terms as accepted by the `x2fx` function. See `x2fx` for a description of this matrix and for a description of the order in which terms appear.

`rstool(x,y,model,alpha)` plots  $100(1 - \alpha)\%$  global confidence interval for predictions as two red curves. For example, `alpha = 0.01` gives 99% confidence intervals.

`rstool(x,y,model,alpha,'xname','yname')` labels the graph using the string matrix `'xname'` for the labels to the  $x$ -axes and the string, `'yname'`, to label the  $y$ -axis common to all the plots.

Drag the dashed blue reference line and watch the predicted values update simultaneously. Alternatively, you can get a specific prediction by typing the value of  $x$  into an editable text field. Use the pop-up menu to interactively

change the model. Click the **Export** button to move specified variables to the base workspace.

**Example**

See “Quadratic Response Surface Models” on page 4-42.

**See Also**

`nlintool`, `x2fx`

**Purpose** Chart of standard deviation for Statistical Process Control

**Syntax**

```
schart(DATA,conf)
schart(DATA,conf,specs)
schart(DATA,conf,specs)
[outliers,h] = schart(DATA,conf,specs)
```

**Description** `schart(data)` displays an S chart of the grouped responses in DATA. The rows of DATA contain replicate observations taken at a given time. The rows must be in time order. The graph contains the sample standard deviation  $s$  for each group, a center line at the average  $s$  value, and upper and lower control limits. The limits are placed at a three-sigma distance on either side of the center line, where sigma is an estimate of the standard deviation of  $s$ . If the process is in control, fewer than 3 out of 1000 observations would be expected to fall outside the control limits by random chance. So, if you observe points outside the limits, you can take this as evidence that the process is not in control.

`schart(DATA,conf)` allows control of the confidence level of the upper and lower plotted control limits. The default `conf = 0.9973` produces three-sigma limits.

```
norminv(1 - (1-.9973)/2)
ans =
    3
```

To get  $k$ -sigma limits, use the expression  $1 - 2 * (1 - \text{normcdf}(k))$ . For example, the correct `conf` value for 2-sigma limits is 0.9545, as shown below.

```
k = 2;
1 - 2 * (1 - normcdf(k))
ans =
    0.9545
```

`schart(DATA,conf,specs)` plots the specification limits in the two element vector `specs`.

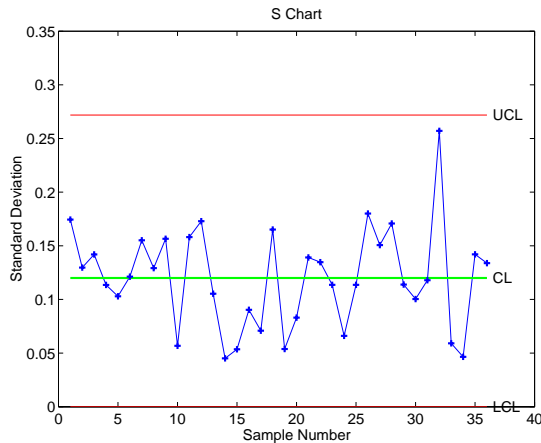
`[outliers,h] = schart(data,conf,specs)` returns `outliers`, a vector of indices to the rows where the mean of DATA is out of control, and `h`, a vector of handles to the plotted lines.

# schart

## Example

This example plots an S chart of measurements on newly machined parts, taken at one hour intervals for 36 hours. Each row of the runout matrix contains the measurements for 4 parts chosen at random. The values indicate, in thousandths of an inch, the amount the part radius differs from the target radius.

```
load parts
schart(runout)
```



All points are within the control limits, so the variability within subgroups is consistent with what would be expected by random chance. There is no evidence that the process is out of control.

## Reference

[1] Montgomery, D., *Introduction to Statistical Quality Control*, John Wiley and Sons 1991. p. 235.

## See Also

capaplot, ewmplot, histfit, xbarplot

**Purpose** Wilcoxon signed rank test for zero median

**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 hypothesis that the data in the vector `x` come from a distribution whose median (and mean, if it exists) is zero, and returns the p-value from the test. `p` is the probability of observing the given result `x`, or one more extreme, by chance if the null hypothesis is true, i.e., the median is zero. Small values of `p` cast doubt on the validity of the null hypothesis. The data are assumed to come from a continuous distribution, symmetric about its median.

`p = signrank(x,m)` performs a two-sided test of the hypothesis that the data in the vector `x` come from a distribution whose median is `m`. `m` must be a scalar.

`p = signrank(x,y)` performs a paired, two-sided test of the hypothesis that the difference between the matched samples in the vectors `x` and `y` comes from a distribution whose median is zero. The differences `x-y` are assumed to come from a continuous distribution, symmetric about its median. `x` and `y` must be the same length.

---

**Note** A hypothesis of zero median for `x-y` is not equivalent to a hypothesis of equal median for `x` and `y`. The `signrank` function tests the former, not the latter.

---

`[p,h] = signrank(...)` returns the result of the hypothesis test, performed at the 5% significance level, in `h`. If `h = 0`, then the null hypothesis, i.e., the median is zero, cannot be rejected at the 5% level. If `h = 1`, then the null hypothesis can be rejected at the 5% level.

# signrank

---

`[p,h] = signrank(...,'alpha', alpha)` returns the result of the hypothesis test performed at the significance level `alpha`.

`[p,h] = signrank(...,'method', method)` computes the p-value using an exact algorithm, if you set `method` to `'exact'` or a normal approximation, if you set `method` to `'approximate'`. If you omit this option, `signrank` uses an exact method for small samples and the approximation for large samples.

`[p,h,stats] = signrank(...)` returns `stats`, a structure with one or two fields. The field `'signedrank'` contains the value of the signed rank statistic. If the sample size is large, then `p` is calculated using a normal approximation and the field `'zval'` contains the value of the normal ( $Z$ ) statistic.

## Example

This example tests the hypothesis of zero median for the difference between two paired samples. The difference between the before and after samples has a symmetric distribution with zero median.

```
before = lognrnd(2, .25, 10, 1);  
after = before + trnd(2, 10, 1);  
[p,h] = signrank(before, after, 0.05)
```

```
p =  
    0.5566
```

```
h =  
    0
```

## See Also

`ranksun`, `signtest`, `ttest`, `ztest`

## References

- [1] Gibbons, J.D., *Nonparametric Statistical Inference*, 2nd Ed., M. Dekker, 1985.
- [2] Hollander, M. and D. A. Wolfe, *Nonparametric Statistical Methods*, Wiley, 1973.



**Purpose** Sign test for zero median

**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 hypothesis that the data in the vector `x` come from a distribution whose median is zero, and returns the `p`-value from the test. `p` is the probability of observing the given result `x`, or a more extreme result, by chance if the null hypothesis is true, i.e., the median is zero. Small values of `p` cast doubt on the validity of the null hypothesis. The data are assumed to come from an arbitrary continuous distribution.

`p = signtest(x,m)` performs a two-sided test of the hypothesis that the data in the vector `x` come from a distribution whose median is `m`. `m` must be a scalar.

`p = signtest(x,y)` performs a paired, two-sided test of the hypothesis that the difference between the matched samples in the vectors `x` and `y` comes from a distribution whose median is zero. The differences `x-y` are assumed to come from an arbitrary continuous distribution. `x` and `y` must be the same length.

---

**Note** A hypothesis of zero median for `x-y` is not equivalent to a hypothesis of equal median for `x` and `y`. The `signtest` function tests the former, not the latter.

---

`[p,h] = signtest(...)` returns the result of the hypothesis test, performed at the 5% significance level, in `h`. If `h = 0`, then the null hypothesis, i.e., the median is zero, cannot be rejected at the 5% level. If `h = 1`, then the null hypothesis can be rejected at the 5% level.

`[p,h] = signtest(...,'alpha', alpha)` returns the result of the hypothesis test performed at the significance level `alpha`.

# signtest

---

`[p,h] = signtest(...,'method', method)` computes the p-value using an exact algorithm, if you set `method` to `'exact'`, or a normal approximation, if you set `method` to `'approximate'`. If you omit this option, `signtest` uses the exact method for small samples and the approximation for large samples .

`[p,h,stats] = signtest(...)` returns `stats`, a structure with one or two fields. The field `'sign'` contains the value of the sign statistic. If the sample size is large, then `p` is calculated using a normal approximation, and the field `'zval'` contains the value of the normal ( $Z$ ) statistic.

## Example

This example tests the hypothesis of zero median for the difference between two paired samples. The difference between the before and after samples has a distribution with zero median.

```
before = lognrnd(2, .25, 10, 1);  
after = before + (lognrnd(0, .5, 10, 1) - 1);  
[p,h] = signtest(before,after,0.05)
```

```
p =  
    0.3438
```

```
h =  
    0
```

## See Also

`ranksun`, `signrank`, `ttest`, `ztest`

## References

- [1] Gibbons, J.D., *Nonparametric Statistical Inference*, 2nd Ed., M. Dekker, 1985.
- [2] Hollander, M. and D. A. Wolfe, *Nonparametric Statistical Methods*, Wiley, 1973.

**Purpose** Silhouette plot for clustered data

**Syntax**

```
silhouette(X,clust)
s = silhouette(X,clust)
[s,h] = silhouette(X,clust)
[...] = silhouette(X,clust,distance)
[...] = 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 numeric vector containing a cluster index for each point, or a character matrix or cell array of strings containing a cluster name for each point. `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,distance)` plots the silhouettes using the inter-point distance measure specified in `distance`. Choices for `distance` are:

'Euclidean'	Euclidean distance
'sqEuclidean'	Squared Euclidean distance (default)
'cityblock'	Sum of absolute differences, i.e., L1
'cosine'	One minus the cosine of the included angle between points (treated as vectors)
'correlation'	One minus the sample correlation between points (treated as sequences of values)
'Hamming'	Percentage of coordinates that differ

# silhouette

---

'Jaccard'            Percentage of non-zero 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 distance function of the form

```
d = distfun(X0,X,p1,p2,...)
```

where `X0` is a 1-by-`p` point, `X` is an `n`-by-`p` matrix of points, and `p1,p2,...` are optional additional arguments. The function `distfun` returns an `n`-by-1 vector `d` of distances between `X0` and each point (row) in `X`. The arguments `p1,p2,...` are passed directly to the function `distfun`.

## Remarks

The silhouette value for each point is a measure of how similar that point is to points in its own cluster compared to points in other clusters, and ranges from -1 to +1. It is defined as

$$S(i) = (\min(b(i,:),2) - a(i)) ./ \max(a(i),\min(b(i,:),2))$$

where  $a(i)$  is the average distance from the  $i$ th point to the other points in its cluster, and  $b(i,k)$  is the average distance from the  $i$ th point to points in another cluster  $k$ .

## Examples

```
X = [randn(10,2)+ones(10,2);  
     randn(10,2)-ones(10,2)];  
cidx = kmeans(X,2,'distance','sqeuclid');  
s = silhouette(X,cidx,'sqeuclid');
```

## See Also

dendrogram, kmeans, linkage, pdist

## References

[1] Kaufman L. and P.J. Rousseeuw, *Finding Groups in Data: An Introduction to Cluster Analysis*, Wiley, 1990

**Purpose** Sample skewness

**Syntax**  
`y = skewness(X)`  
`y = skewness(X, flag)`

**Description** `y = skewness(X)` returns the sample skewness of `X`. For vectors, `skewness(x)` is the skewness of the elements of `x`. For matrices, `skewness(X)` is a row vector containing the sample skewness of each column. For N-dimensional arrays, `skewness` operates along the first nonsingleton dimension of `X`.

`y = skewness(X, flag)` specifies whether to correct for bias (`flag = 0`) or not (`flag = 1`, the default). When `X` represents a sample from a population, the skewness of `X` is biased; that is, it will tend to differ from the population skewness by a systematic amount that depends on the size of the sample. You can set `flag = 0` to correct for this systematic bias.

`skewness(X, flag, dim)` takes the skewness along dimension `dim` of `X`.

`skewness` treats 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  $\mu$  is the mean of  $x$ ,  $\sigma$  is the standard deviation of  $x$ , and  $E(t)$  represents the expected value of the quantity  $t$ .

**Example**

```
X = randn([5 4])

X =
    1.1650    1.6961   -1.4462   -0.3600
    0.6268    0.0591   -0.7012   -0.1356
    0.0751    1.7971    1.2460   -1.3493
```

# skewness

---

0.3516	0.2641	-0.6390	-1.2704
-0.6965	0.8717	0.5774	0.9846

y = skewness(X)

y =

-0.2933	0.0482	0.2735	0.4641
---------	--------	--------	--------

## See Also

kurtosis, mean, moment, std, var

**Purpose** Reformat a distance matrix between upper triangular and square form

**Syntax**

```
Z = squareform(y)
y = squareform(Z)
Z = squareform(y, 'tovector')
Y = squareform(Z, 'tomatrix')
```

**Description**

`Z = squareform(y)`, where `y` is a vector as created by the `pdist` function, converts `y` into a square, symmetric format `Z`, in which `Z(i, j)` denotes the distance between the `i`th and `j`th objects in the original data.

`y = squareform(Z)`, where `Z` is a square, symmetric matrix with zeros along the diagonal, creates a vector `y` containing the `Z` elements below the diagonal. `y` has the same format as the output from the `pdist` function.

`Z = squareform(y, 'tovector')` forces `squareform` to treat `y` as a vector.

`Y = squareform(Z, 'tomatrix')` forces `squareform` to treat `Z` as a matrix.

The last two formats are useful if the input has a single element, so that it is ambiguous whether the input is a vector or square matrix.

## Example

```
y = 1:6

y =

     1     2     3     4     5     6

X = [0 1 2 3; 1 0 4 5; 2 4 0 6; 3 5 6 0]
```

X =

```

     0     1     2     3
     1     0     4     5
     2     4     0     6
     3     5     6     0,
```

Then `squareform(y) = X` and `squareform(X) = y`.

# squareform

---

## See Also

`pdist`



**Purpose** Get parameter values from a statistics options structure

**Syntax**

```
val = statget(options,'param')  
val = statget(options,'param',default)
```

**Description** `val = statget(options,'param')` returns the value of the specified parameter 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 or edit a statistics options structure

**Syntax**

```
options = statset('param1',value1,'param2',value2,...)
statset
options = statset
options = statset(statfun)
options = statset(olddopts,'param1',value1,...)
options = statset(olddopts,newopts)
```

**Description** `options = statset('param1',value1,'param2',value2,...)` creates an options structure called `options`, which you can pass as an input argument to other statistics functions, such as distribution fitting functions, that use iterative algorithms to maximize or minimize a statistical fitting criterion. You can use the `options` argument to override the default parameters for these algorithms. You specify the value of a parameter using a name/value pair, 'param', value, where 'param' is the name of the parameter and value is its value. Parameters that you do not specify are set to []. If you pass `options` as an input to one of these functions, the function uses the parameter values you specify and uses its default value for any parameter that you do not specify. You only need to type only enough leading characters to define the parameter name uniquely. Case is ignored for parameter names.

`statset` with no input or output arguments displays a complete list of parameters with their valid values.

`options = statset` (with no input arguments) creates an options structure `options` where all fields are set to [].

`options = statset(statfun)` creates an options structure `options` with all parameter names and default values relevant to the statistics function `statfun`. You can specify `statfun` either as a string, such as 'evfit', or as a function handle, such as @evfit.

`options = statset(olddopts,'param1',value1,...)` creates a copy of `olddopts`, modifying the specified parameters with the specified values.

`options = statset(olddopts,newopts)` combines an existing options structure, `olddopts`, with a new options structure, `newopts`. Any parameters in

newopts with nonempty values overwrite the corresponding old parameters in oldopts.

## Parameters

The following table lists the valid parameters for the options structure, their meanings, and their allowed values. You can also view these parameters and allowed values by typing `statset` at the command line.

Parameter	Meaning	Allowed values
DerivStep	Relative difference used in finite difference derivative calculations. May be a scalar or the same size as the parameter vector.	Positive scalar or vector
Display	Amount of information displayed by the algorithm	<ul style="list-style-type: none"> <li>• 'off' — displays no information</li> <li>• 'final' — displays the final output</li> <li>• 'notify' — displays output only if the algorithm fails to converge</li> </ul>
FunValCheck	Check for invalid values, such as NaN or Inf, from the objective function	<ul style="list-style-type: none"> <li>• 'off'</li> <li>• 'on'</li> </ul>
GradObject	Objective function can return a gradient vector as a second output.	<ul style="list-style-type: none"> <li>• 'off'</li> <li>• 'on'</li> </ul>
MaxFunEvals	Maximum number of objective function evaluations allowed	Positive integer
MaxIter	Maximum number of iterations allowed	Positive integer

# statset

Parameter	Meaning	Allowed values
TolBnd	Parameter bound tolerance	Positive scalar
TolFun	Termination tolerance for the objective function value	Positive scalar
TolX	Termination tolerance for the parameters	Positive scalar

## Examples

Suppose you want to change the default parameters for the function `evfit`, which fits data to an extreme value distribution. To see the defaults for `evfit`, enter

```
statset('evfit')  
  
ans =  
  
    Display: 'off'  
MaxFunEvals: []  
    MaxIter: []  
    TolBnd: []  
    TolFun: []  
    TolX: 1.0000e-006  
    GradObj: []  
    DerivStep: []  
    FunValCheck: []
```

Note that the only parameters `evfit` uses are `Display` and `TolX`. To change the value of `TolX` to  $1e-8$ , enter

```
my_opts = statset('TolX',1e-8)  
  
my_opts =  
  
    Display: []  
MaxFunEvals: []  
    MaxIter: []  
    TolBnd: []  
    TolFun: []  
    TolX: 1.0000e-008  
    GradObj: []
```

```
    DerivStep: []  
    FunValCheck: []
```

When you pass `my_opts` into `evfit` with the command

```
evfit(data, [], [], [], my_opts)
```

`evfit` uses its default value 'notify' for `Display` and overrides the default value of `TolX` with `1e-8`.

See the reference page for `evfit` for more information about its syntax.

### See Also

`evfit`, `factoran`, `gamfit`, `lognfit`, `nbinfit`, `normfit`, `statget`

# std

---

**Purpose** Standard deviation of a sample

**Syntax** `y = std(X)`

**Description** `y = std(X)` computes the sample standard deviation of the data in `X`. For vectors, `std(x)` is the standard deviation of the elements in `x`. For matrices, `std(X)` is a row vector containing the standard deviation of each column of `X`. For N-dimensional arrays, `std` operates along the first nonsingleton dimension of `X`.

`std` normalizes by  $n-1$  where  $n$  is the sample size. The result `Y` is the square root of an unbiased estimator of the variance of the population from which `X` is drawn, as long as `X` consists of independent, identically distributed samples.

The standard deviation is

$$y = \left( \frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2 \right)^{\frac{1}{2}}$$

where the sample average is  $\bar{x} = \frac{1}{n} \sum x_i$ .

The `std` function is part of the standard MATLAB language.

`Y = std(X, 1)` normalizes `Y` by  $n$ . The result `Y` is the square root of the second moment of the sample about its mean. `std(X, 0)` is the same as `std(X)`.

`Y = std(X, flag, dim)` takes the standard deviation along the dimension `dim` of `X`. Set `flag` to 0 to normalize `Y` by  $n-1$ ; set `flag` to 1 to normalize by  $n$ .

## Examples

In each column, the expected value of `y` is one.

```
x = normrnd(0,1,100,6);
y = std(x)

y =
    0.9536    1.0628    1.0860    0.9927    0.9605    1.0254

y = std(-1:2:1)

y =
    1.4142
```

**See Also**

cov, var

# stepwise

---

**Purpose** Interactive environment for stepwise regression

**Syntax** `stepwise(X,y)`  
`stepwise(X,y,inmodel,penter,premove)`

**Description** `stepwise(X,y)` displays an interactive tool for creating a regression model to predict the vector  $y$ , using a subset of the predictors given by columns of the matrix  $X$ . Initially, no predictors are included in the model, but you can click predictors to switch them into and out of the model.

For each predictor in the model, the interactive tool plots the predictor's least squares coefficient as a blue filled circle. For each predictor not in the model, the interactive tool plots a filled red circle to indicate the coefficient the predictor would have if you add it to the model. Horizontal bars in the plot indicate 90% confidence intervals (colored) and 95% confidence intervals (black).

`stepwise(X,y,inmodel,penter,premove)` specifies the initial state of the model and the confidence levels to use. `inmodel` is either a logical vector, whose length is the number of columns in  $X$ , or a vector of indices, whose values range from 1 to the number of columns in  $X$ , specifying the predictors that are included in the initial model. The default is to include no columns of  $X$ . `penter` specifies the maximum p-value that a predictor can have for the interactive tool to recommend adding it to the model. The default value of `penter` is 0.05. `premove` specifies the minimum p-value that a predictor can have for the interactive tool to recommend removing it from the model. The default value of `premove` is 0.10.

The interactive tool treats a NaN in either  $X$  or  $y$  as a missing value. The tool does not use rows containing missing values in the fit.

**Examples** See “Quadratic Response Surface Models” on page 4-42 and “Stepwise Regression Demo” on page 4-45.

**Reference** [1] Draper, N., and H. Smith, *Applied Regression Analysis, Second Edition*, John Wiley and Sons, Inc., 1981, pp. 307–312.

**See Also** `, regress, rstool, stepwisefit`



**Purpose** Fit regression model using stepwise regression

**Syntax**

```
b = stepwisefit(X,y)
[b,se,pval,inmodel,stats,nextstep,history] = stepwisefit(...)
[...] = stepwisefit(X,y,'Param1',value1,'Param2',value2,...)
```

**Description** `b = stepwisefit(X,y)` uses stepwise regression to model the response variable `y` as a function of the predictor variables represented by the columns of the matrix `X`. The result is a vector `b` of estimated coefficient values for all columns of `X`. The `b` value for a column not included in the final model is the coefficient that you would obtain by adding that column to the model.

```
[b,se,pval,inmodel,stats,nextstep,history] = stepwisefit(...)
```

returns the following additional results:

- `se` is a vector of standard errors for `b`.
- `pval` is a vector of p-values for testing whether `b` is 0.
- `inmodel` is a logical vector, whose length equals the number of columns in `X`, specifying which predictors are in the final model. A 1 in position `j` indicates that the `j`th predictor is in the final model; a 0 indicates that the corresponding predictor is not in the final model.
- `stats` is a structure containing additional statistics.
- `nextstep` is the recommended next step — either the index of the next predictor to move in or out, or 0 if no further steps are recommended.
- `history` is a structure containing information about the history of steps taken.

```
[...] = stepwisefit(X,y,'Param1',value1,'Param2',value2,...)
```

specifies one or more of the name/value pairs described in the following table.

# stepwisefit

Parameter Name	Parameter Value
'inmodel'	Logical vector specifying the predictors to include in the initial fit. The default is a vector of all zeros, specifying no predictors.
'penter'	Maximum p-value for a predictor to be added. The default is 0.05.
'premove'	Minimum p-value for a predictor to be removed. The default is 0.10.
'display'	'on' displays information about each step. 'off' omits the information.
'maxiter'	Maximum number of steps to take (default is no maximum)
'keep'	Logical vector specifying the predictors to keep in their initial state. The default is a vector of all zeros, specifying no predictors.
'scale'	'on' scales each column of X by its standard deviation before fitting. 'off' does not scale (the default).

## Example

```
load hald
stepwisefit(ingredients, heat, 'penter', .08)
Initial columns included: none
Step 1, added column 4, p=0.000576232
Step 2, added column 1, p=1.10528e-006
Step 3, added column 2, p=0.0516873
Step 4, removed column 4, p=0.205395
Final columns included: 1 2
```

```
ans =  
  
      'Coeff'      'Std.Err.'      'Status'      'P'  
      [ 1.4683]      [ 0.1213]      'In'          [2.6922e-007]  
      [ 0.6623]      [ 0.0459]      'In'          [5.0290e-008]  
      [ 0.2500]      [ 0.1847]      'Out'         [ 0.2089]  
      [-0.2365]      [ 0.1733]      'Out'         [ 0.2054]
```

```
ans =  
  
1.4683  
0.6623  
0.2500  
-0.2365
```

## See Also

addedvarplot, , regress, rstool, stepwise

# surfht

---

**Purpose** Interactive contour plot

**Syntax** `surfht(Z)`  
`surfht(x,y,Z)`

**Description** `surfht(Z)` is an interactive contour plot of the matrix  $Z$  treating the values in  $Z$  as height above the plane. The  $x$ -values are the column indices of  $Z$  while the  $y$ -values are the row indices of  $Z$ .

`surfht(x,y,Z)` where  $x$  and  $y$  are vectors specify the  $x$  and  $y$ -axes on the contour plot. The length of  $x$  must match the number of columns in  $Z$ , and the length of  $y$  must match the number of rows in  $Z$ .

There are vertical and horizontal reference lines on the plot whose intersection defines the current  $x$ -value and  $y$ -value. You can drag these dotted white reference lines and watch the interpolated  $z$ -value (at the top of the plot) update simultaneously. Alternatively, you can get a specific interpolated  $z$ -value by typing the  $x$ -value and  $y$ -value into editable text fields on the  $x$ -axis and  $y$ -axis respectively.

**Purpose** Frequency table

**Syntax** TABLE = tabulate(x)  
tabulate(x)

**Description** TABLE = tabulate(x) takes a vector x and returns a matrix, TABLE. The first column of TABLE contains the unique values of x. The second contains the number of instances of each value. The last column contains the percentage of each value.

TABLE = tabulate(ARRAY), where ARRAY is a character array or a cell array of strings, returns TABLE as a cell array. The first column contains the unique string values in ARRAY. The other two columns contain the same information as for a vector input.

tabulate with no output arguments displays a formatted table in the command window.

**Example** tabulate([1 2 4 4 3 4])

Value	Count	Percent
1	1	16.67%
2	1	16.67%
3	1	16.67%
4	3	50.00%

**See Also** pareto

# tblread

---

**Purpose** Read tabular data from the file system

**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 the 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.

[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.

[data,varnames,casenames] = tblread(filename,'delimiter') reads from 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'.

tblread returns the data read in three values.

Return Value	Description
data	Numeric matrix with a value for each variable-case pair.
varnames	String matrix containing the variable names in the first row.
casenames	String matrix containing the names of each case in the first column.

### Example

```
[data,varnames,casenames] = tblread('sat.dat')  
  
data =  
    470    530  
    520    480  
  
varnames =  
Male  
Female  
  
casenames =  
Verbal  
Quantitative
```

### See Also

caseread, tblwrite, tdfread

# tblwrite

---

**Purpose** Writes tabular data to the file system

**Syntax**

```
tblwrite(data,'varnames','casenames')  
tblwrite(data,'varnames','casenames','filename')  
tblwrite(data,'varnames','casenames','filename','delimiter')
```

**Description** `tblwrite(data,'varnames','casenames')` displays the **File Open** dialog box for interactive specification of the tabular data output file. The file format has variable names in the first row, case names in the first column and data starting in the (2,2) position.

'varnames' is a string matrix containing the variable names. 'casenames' is a string matrix containing the names of each case in the first column. data is a numeric matrix with a value for each variable-case pair.

`tblwrite(data,'varnames','casenames','filename')` specifies a file in the current directory, or the complete path name of any file in the string 'filename'.

`tblwrite(data,'varnames','casenames','filename','delimiter')` writes to the file using 'delimiter' as the delimiting character. The following table lists the accepted character values for 'delimiter' and their equivalent string values.

Character	String
' '	'space'
'\t'	'tab'
','	'comma'
';'	'semi'
' '	'bar'

The default value of 'delimiter' is 'space'.



**Example**

Continuing the example from `tblread`:

```
tblwrite(data, varnames, casenames, 'satstest.dat')  
type satstest.dat
```

	Male	Female
Verbal	470	530
Quantitative	520	480

**See Also**

`casewrite`, `tblread`

# tcdf

---

**Purpose** Student's t cumulative distribution function (cdf)

**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

$$p = F(x|v) = \int_{-\infty}^x \frac{\Gamma\left(\frac{v+1}{2}\right)}{\Gamma\left(\frac{v}{2}\right)} \frac{1}{\sqrt{v\pi}} \frac{1}{\left(1 + \frac{t^2}{v}\right)^{\frac{v+1}{2}}} dt$$

The result,  $p$ , is the probability that a single observation from the t distribution with  $v$  degrees of freedom will fall in the interval  $(-\infty, x]$ .

**Examples** Suppose 10 samples of Guinness beer have a mean alcohol content of 5.5% by volume and the standard deviation of these samples is 0.5%. What is the probability that the true alcohol content of Guinness beer is less than 5%?

```
t = (5.0 - 5.5) / 0.5;  
probability = tcdf(t,10 - 1)
```

```
probability =
```

```
0.1717
```

**See Also** `cdf`, `tinvs`, `tpdf`, `trnd`, `tstat`

**Purpose** Read file containing tab-delimited numeric and text values

**Syntax**

```
tdfread
tdfread('filename')
tdfread('filename','delimiter')
```

**Description** tdfread displays the **File Open** dialog box for interactive selection of the data file. The file should consist of columns of values, separated by tabs, and with column names in the first line of the file. Each column is read from the file and assigned to a variable with the specified name. If all values for a column are numeric, the variable is converted to numbers; otherwise the variable is a string matrix. After all values are imported, tdfread displays information about the imported values using the format of the whos command.

tdfread('filename') allows command line specification of the name of a file in the current directory, or the complete pathname of any file.

tdfread('filename','delimiter') indicates that the character specified by 'delimiter' separates columns in the file. Accepted values are:

- ' ' or 'space'
- '\t' or 'tab'
- ',' or 'comma'
- ';' or 'semi'
- '|' or 'bar'

The default delimiter is 'tab'.

### Example

```
type sat2.dat

Test,Gender,Score
Verbal,Male,470
Verbal,Female,530
Quantitative,Male,520
Quantitative,Female,480
tdfread('sat2.dat','')

Name          Size          Bytes  Class
```

# tdfread

---

Gender	4x6	48	char array
Score	4x1	32	double array
Test	4x12	96	char array

Grand total is 76 elements using 176 bytes

## See Also

tblread

**Purpose** Inverse of the Student's t cumulative distribution function (cdf)

**Syntax** `X = tinv(P,V)`

**Description** `X = tinv(P,V)` computes the inverse of Student's t cdf with parameter V for the corresponding probabilities in P. P and V can be vectors, matrices, or multidimensional arrays that have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs. The values in P must lie on the interval [0 1].

The t inverse function in terms of the t cdf is

$$x = F^{-1}(p|v) = \{x:F(x|v) = p\}$$

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 (pdf)

**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

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

## Examples

The mode of the t distribution is at  $x = 0$ . This example shows that the value of the function at the mode is an increasing function of the degrees of freedom.

```
tpdf(0,1:6)
```

```
ans =
```

```
0.3183    0.3536    0.3676    0.3750    0.3796    0.3827
```

The t distribution converges to the standard normal distribution as the degrees of freedom approaches infinity. How good is the approximation for  $v = 30$ ?

```
difference = tpdf(-2.5:2.5,30) - normpdf(-2.5:2.5)
```

```
difference =
```

```
0.0035   -0.0006   -0.0042   -0.0042   -0.0006   0.0035
```

**See Also** `pdf`, `tcdf`, `tinv`, `trnd`, `tstat`

**Purpose** Show classification or regression tree graphically

**Syntax** `treedisp(T)`  
`treedisp(T, 'param1', val1, 'param2', val2, ...)`

**Description** `treedisp(T)` takes as input a decision tree `T` as computed by the `treefit` function, and displays it in a figure window. Each branch in the tree is labeled with its decision rule, and each terminal node is labeled with the predicted value for that node.

For each branch node, the left child node corresponds to the points that satisfy the condition, and the right child node corresponds to the points that do not satisfy the condition.

The **Click to display** pop-up menu at the top of the figure enables you to display more information about each node:

- Identity** The node number, whether the node is a branch or a leaf, and the rule that governs the node
- Variable ranges** The range of each of the predictor variables for that node
- Node statistics** The number of elements of the data set that fit the node and ... (the mean and standard deviation of what?)

After you select the type of information you want, click on any node to display the information for that node.

The **Pruning level** spin button displays the number of levels that have been cut from the tree, and the number of levels in the unpruned tree. For example, **1 of 6** indicates that the unpruned tree has six levels, and that one level has been cut from the tree. Use the spin button to change the pruning level.

`treedisp(T, 'param1', val1, 'param2', val2, ...)` specifies optional parameter name-value pairs. Valid parameters are:

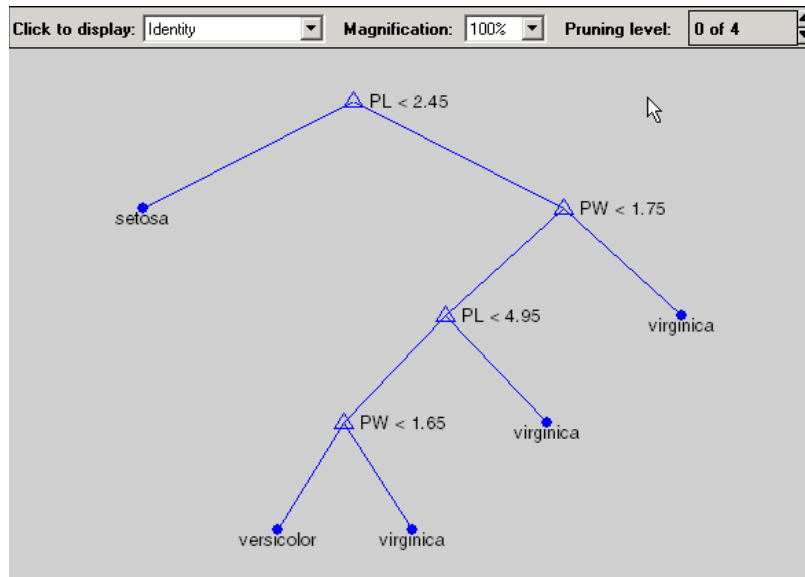
- 'names' A cell array of names for the predictor variables, in the order in which they appear in the `X` matrix from which the tree was created (see `treefit`)
- 'prunelevel' Initial pruning level to display

# treedisp

## 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'});
```



## See Also

treefit, treeprune, treetest



**Purpose** Fit a tree-based model for classification or regression

**Syntax**  
`T = treefit(X,y)`  
`T = treefit(X,y,'param1',val1,'param2',val2,...)`

**Description** `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 parameters are

For all trees:

<code>'catidx'</code>	Vector of indices of the columns of <code>X</code> . <code>treefit</code> treats these columns as unordered categorical values.
<code>'method'</code>	Either <code>'classification'</code> (default if <code>y</code> is text) or <code>'regression'</code> (default if <code>y</code> is numeric)
<code>'splitmin'</code>	A number <code>n</code> such that impure nodes must have <code>n</code> or more observations to be split (default 10)
<code>'prune'</code>	<code>'on'</code> (default) to compute the full tree and a sequence of pruned subtrees, or <code>'off'</code> for the full tree without pruning

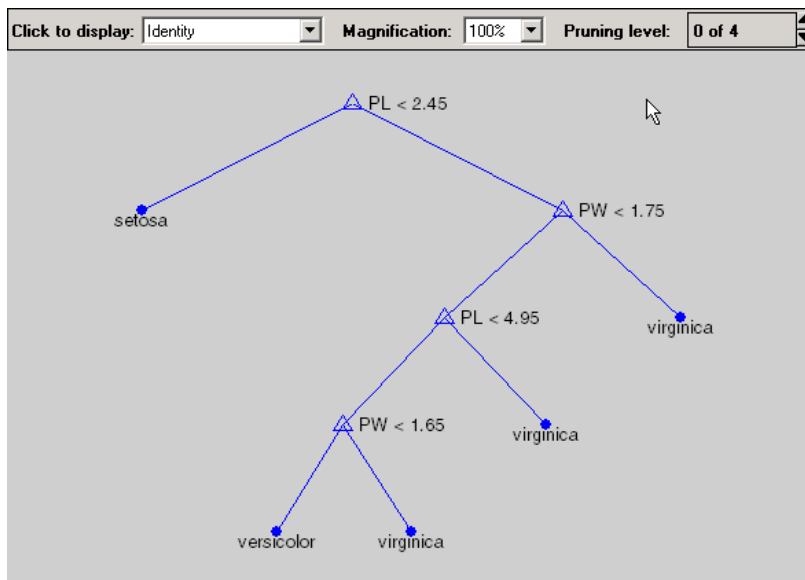
For classification trees only:

'cost'	p-by-p matrix C, where p is the number of distinct response values or class names in the input y. $C(i, j)$ is the cost of classifying a point into class i if its true class is j (default has $C(i, j)=1$ if $i \neq j$ , and $C(i, j)=0$ if $i=j$ ). C can also be a structure S with two fields: S.group containing the group names, 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'});
```

**See Also**

treedisp, treetest

**References**

[1] Breiman, et al., *Classification and Regression Trees*, Chapman and Hall, Boca Raton, 1993.

# treeprune

---

**Purpose** Produce a sequence of subtrees by pruning

**Syntax**

```
T2 = treeprune(T1, 'level', level)
T2 = treeprune(T1, 'nodes', nodes)
T2 = treeprune(T1)
```

**Description** T2 = treeprune(T1, 'level', level) takes a decision tree T1 as created by the treefit function, and a pruning level level, and returns the decision tree T2 pruned to that level. The value level = 0 means no pruning. Trees are pruned based on an optimal pruning scheme that first prunes branches giving less improvement in error cost.

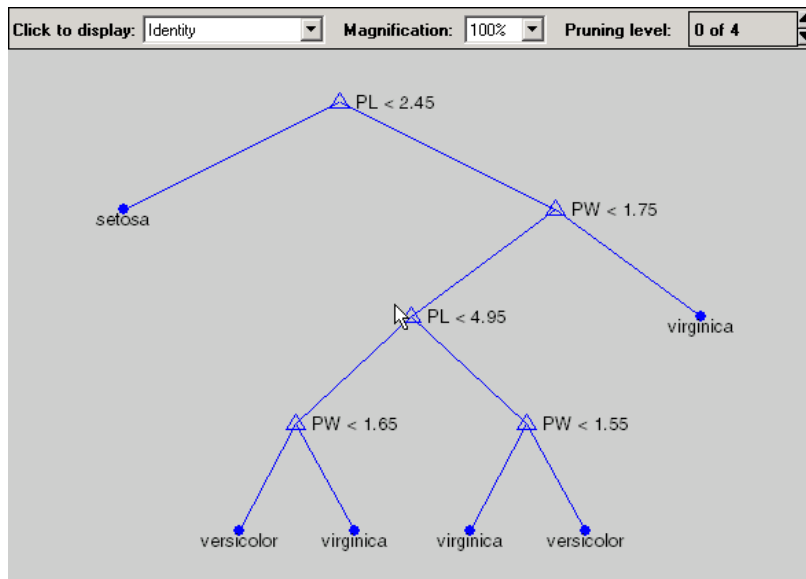
T2 = treeprune(T1, 'nodes', nodes) prunes the nodes listed in the nodes vector from the tree. Any T1 branch nodes listed in nodes become leaf nodes in T2, unless their parent nodes are also pruned. The treedisp function can display the node numbers for any node you select.

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.

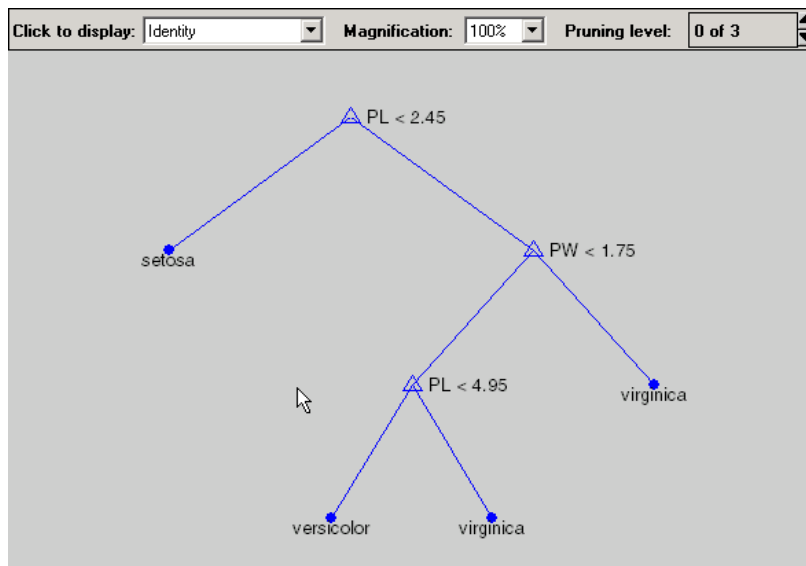
Pruning is the process of reducing a tree by turning some branch nodes into leaf nodes, and removing the leaf nodes under the original branch.

**Examples** Display the full tree for Fisher's iris data, as well as the next largest tree from the optimal pruning sequence.

```
load fisheriris;
t = treefit(meas,species,'splitmin',5);
treedisp(t,'names',{'SL' 'SW' 'PL' 'PW'});
```



```
t1 = treeprune(t,'level',1);  
treedisp(t1,'names',{'SL' 'SW' 'PL' 'PW'});
```



# treeprune

---

## See Also

treefit, treetest, treedisp

---

<b>Purpose</b>	Compute error rate for tree
<b>Syntax</b>	<pre>cost = treetest(T, 'resubstitution') cost = treetest(T, 'test', X, y) cost = treetest(T, 'crossvalidate', X, y) [cost, secost, ntnodes, bestsize] = treetest(...) [...] = treetest(..., 'param1', val1, 'param2', val2, ...)</pre>
<b>Description</b>	<p><code>cost = treetest(T, 'resubstitution')</code> computes the cost of the tree <code>T</code> using a resubstitution method. <code>T</code> is a decision tree as created by the <code>treefit</code> 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 <code>T</code> is a classification tree, the cost of a node is the sum of the misclassification costs of the observations in that node. If <code>T</code> is a regression tree, the cost of a node is the average squared error over the observations in that node. <code>cost</code> is a vector of cost values for each subtree in the optimal pruning sequence for <code>T</code>. 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.</p> <p><code>cost = treetest(T, 'test', X, y)</code> uses the predictor matrix <code>X</code> and response <code>y</code> as a test sample, applies the decision tree <code>T</code> to that sample, and returns a vector cost of cost values computed for the test sample. <code>X</code> and <code>y</code> should not be the same as the learning sample, which is the sample that was used to fit the tree <code>T</code>.</p> <p><code>cost = treetest(T, 'crossvalidate', X, y)</code> uses 10-fold cross-validation to compute the cost vector. <code>X</code> and <code>y</code> should be the learning sample, which is the sample that was used to fit the tree <code>T</code>. 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, <code>treetest</code> 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.</p> <p><code>[cost, secost, ntnodes, bestlevel] = treetest(...)</code> also returns the vector <code>secost</code> containing the standard error of each cost value, the vector <code>ntnodes</code> containing number of terminal nodes for each subtree, and the scalar <code>bestlevel</code> containing the estimated best level of pruning. <code>bestlevel = 0</code></p>

means no pruning, i.e., the full unpruned tree. The best level is the one that produces the smallest tree that is within one standard error of the minimum-cost subtree.

[...] = treetest(..., 'param1', val1, 'param2', val2, ...) specifies optional parameter name-value pairs chosen from the following:

'nsamples'      The number of cross-validations samples (default 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

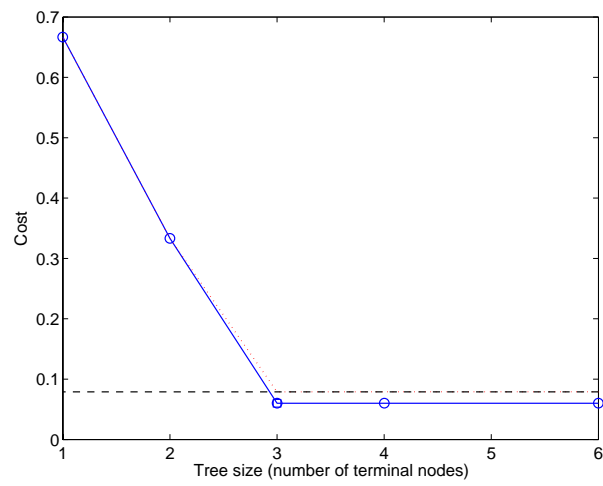
Find the best tree for Fisher's iris data using cross-validation. The solid line shows the estimated cost for each tree size, the dashed line marks 1 standard error above the minimum, and the square marks the smallest tree under the dashed line.

```
% 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. error of minimum cost tree.
[mincost,minloc] = min(c);
plot(n,c,'b-o', n,c+s,'r:', n(best+1),c(best+1),'bs',...
     n,(mincost+s(minloc))*ones(size(n)),'k--');
xlabel('Tree size (number of terminal nodes)')
ylabel('Cost')
```



**See Also**`treetest`, `treedisp`

# treeval

---

**Purpose** Compute fitted value for decision tree applied to data

**Syntax**

```
YFIT = treeval(T,X)
YFIT = treeval(T,X,subtrees)
[YFIT,NODE] = treeval(...)
[YFIT,NODE,CNAME] = treeval(...)
```

**Description** `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(j)` is the fitted response value for a point having the predictor values `X(j,:)`. For a classification tree, `YFIT(j)` is the class number into which the tree would assign the point with data `X(j,:)`. To convert the number into a class name, use the third output argument, `cname` (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, then the output `YFIT` is an `n`-by-`k` matrix, with the `j`th column containing the fitted values produced by the `subtrees(j)` subtree. `subtrees` must be sorted in ascending order.

`[YFIT,NODE] = treeval(...)` also returns an array `NODE` of the same size as `YFIT` containing the node number assigned to each row of `X`. The `treedisp` function can display the node numbers for any node you select.

`[YFIT,NODE,CNAME] = treeval(...)` is valid only for classification trees. It returns a cell array `CNAME` containing the predicted class names.

## Examples

Find the predicted classifications for Fisher's iris data.

```
load fisheriris;
t = treefit(meas,species); % Create decision tree
sfit = treeval(t,meas); % Find assigned class numbers
sfit = t.classname(sfit); % Get class names
mean(strcmp(sfit,species)) % Compute proportion correctly
% classified

ans =
    0.9800
```

**See Also**      treefit, treeprune, treetest

# trimmean

---

**Purpose** Mean of a sample of data excluding extreme values

**Syntax** `m = trimmean(X,percent)`  
`m = trimmean(X,percent,dim)`

**Description** `m = trimmean(X,percent)` calculates the mean of a sample  $X$  excluding the highest and lowest  $(\text{percent}/2)\%$  of the observations. For a vector input,  $m$  is the trimmed mean of  $X$ . For a matrix input,  $m$  is a row vector containing the trimmed mean of each column of  $X$ . For  $N$ -dimensional arrays, `trimmean` operates along the first nonsingleton dimension of  $X$ . `percent` is a scalar between 0 and 100.

`trimmean(X,percent,dim)` takes the trimmed mean along dimension `dim` of  $X$ .

**Remarks** The trimmed mean is a robust estimate of the location of a sample. If there are outliers in the data, the trimmed mean is a more representative estimate of the center of the body of the data than the mean. However, if the data is all from the same probability distribution, then the trimmed mean is less efficient than the sample mean as an estimator of the location of the data.

**Examples** This example shows a Monte Carlo simulation of the efficiency of the 10% trimmed mean relative to the sample mean for normal data.

```
x = normrnd(0,1,100,100);  
m = mean(x);  
trim = trimmean(x,10);  
sm = std(m);  
strim = std(trim);  
efficiency = (sm/strim).^2  
  
efficiency =  
  
0.9702
```

**See Also** `mean`, `median`, `geomean`, `harmmean`

**Purpose** Random numbers from Student's t distribution

**Syntax**

```
R = trnd(V)
R = trnd(V,m)
R = trnd(V,m,n)
```

**Description** `R = trnd(V)` generates random numbers from Student's t distribution with  $V$  degrees of freedom.  $V$  can be a vector, a matrix, or a multidimensional array. The size of  $R$  is the size of  $V$ .

`R = trnd(V,m)` generates random numbers from Student's t distribution with  $V$  degrees of freedom, where  $v$  is a row vector. If  $v$  is a 1-by-2 vector,  $R$  is a matrix with  $v(1)$  rows and  $v(2)$  columns. If  $v$  is 1-by- $n$ ,  $R$  is an  $n$ -dimensional array.

`R = trnd(V,m,n)` generates random numbers from Student's t distribution with  $V$  degrees of freedom, where scalars  $m$  and  $n$  are the row and column dimensions of  $R$ .

### Reproducing the Output of trnd

`trnd` uses the MATLAB functions `rand` and `randn` to generate random numbers. When you call `trnd`, you change the current states of `rand` and `randn`, and thereby alter the output of subsequent calls to `trnd` or any other functions that depend on `rand` or `randn`. If you want to reproduce the output of `trnd`, reset the states of `rand` and `randn` to the same fixed values each time you call `trnd`. For an example of how to do this, and a list of the Statistics Toolbox functions that depend on `rand` or `randn`, see “Reproducing the Output of Random Number Functions” on page 2-46.

---

**Note** The results in the following examples depend on the current states of `rand` and `randn`. If you run the code in these examples, your results may differ from the answers shown here.

---

### Examples

```
noisy = trnd(ones(1,6))
noisy =
    19.7250    0.3488    0.2843    0.4034    0.4816   -2.4190
```

# trnd

---

```
numbers = trnd(1:6,[1 6])
```

```
numbers =
```

```
    -1.9500    -0.9611    -0.9038     0.0754     0.9820     1.0115
```

```
numbers = trnd(3,2,6)
```

```
numbers =
```

```
    -0.3177    -0.0812    -0.6627     0.1905    -1.5585    -0.0433  
     0.2536     0.5502     0.8646     0.8060    -0.5216     0.0891
```

## See Also

tcdf, tinu, tpdf, tstat

**Purpose** Mean and variance for the Student's t distribution

**Syntax** `[M,V] = tstat(NU)`

**Description** `[M,V] = tstat(NU)` returns the mean 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 and variance for 1 to 30 degrees of freedom.

```
[m,v] = tstat(reshape(1:30,6,5))
```

```
m =
```

```
NaN    0    0    0    0
  0    0    0    0    0
  0    0    0    0    0
  0    0    0    0    0
  0    0    0    0    0
  0    0    0    0    0
```

```
v =
```

```
NaN    1.4000    1.1818    1.1176    1.0870
NaN    1.3333    1.1667    1.1111    1.0833
3.0000    1.2857    1.1538    1.1053    1.0800
2.0000    1.2500    1.1429    1.1000    1.0769
1.6667    1.2222    1.1333    1.0952    1.0741
1.5000    1.2000    1.1250    1.0909    1.0714
```

Note that the variance does not exist for one and two degrees of freedom.

**See Also** `tcdf`, `tinv`, `tpdf`, `trnd`

# ttest

---

**Purpose** Hypothesis testing for a single sample mean

**Syntax**

```
h = ttest(x)
h = ttest(x,m)
h = ttest(x,y)
h = ttest(x,m,alpha)
h = ttest(x,m,alpha,tail)
[h,p,ci] = ttest(...)
[h,p,ci,stats] = ttest(...)
```

**Description** `h = ttest(x)` performs a t-test of the hypothesis that the data in the vector `x` come from a distribution with mean zero, and returns the result of the test in `h`. `h==0` indicates that the null hypothesis (mean is zero) cannot be rejected at the 5% significance level. `h==1` indicates that the null hypothesis can be rejected at the 5% level. The data are assumed to come from a normal distribution with unknown variance.

`h = ttest(x,m)` performs a t-test of the hypothesis that the data in the vector `x` come from a distribution with mean `m`.

`h = ttest(x,y)` performs a paired t-test of the hypothesis that two matched (or paired) samples in the vectors `x` and `y` come from distributions with equal means. The difference `x - y` is assumed to come from a normal distribution with unknown variance. `x` and `y` must have the same length.

`h = ttest(...,alpha)` performs the test at the significance level  $(100*\alpha)\%$ . For example, if `alpha = 0.01`, and the result, `h`, is 1 you can reject the null hypothesis at the significance level 0.01. If `h` is 0, you cannot reject the null hypothesis at the `alpha` level of significance.

`h = ttest(...,alpha,tail)` performs the test against the alternative hypothesis specified by `tail`. There are three options for `tail`:

- 'both' — Mean is not 0 (or `m`) (two-tailed test). This is the default.
- 'right' — Mean is greater than 0 (or `m`) (right-tailed test).
- 'left' — Mean is less than 0 (or `m`) (left-tailed test).

Output `p` is the p-value associated with the t-statistic.



$$T = \frac{x - m}{s / \sqrt{n}}$$

where  $s$  is the sample standard deviation and  $n$  is the number of observations in the sample.  $p$  is the probability that the value of the t-statistic is equal to or more extreme than the observed value by chance, under the null hypothesis that the mean of  $x$  is equal to  $m$ .

$ci$  is a  $1 - \alpha$  confidence interval for the true mean.

`[h,p,ci,stats] = ttest(...)` returns a structure with the following fields:

- 'tstat' — Value of the test statistic
- 'df' — Degrees of freedom of the test
- 'sd' — Estimated population standard deviation. For a paired test, this is the standard deviation of  $x - y$ .

## Example

This example generates 100 normal random numbers with theoretical mean zero and standard deviation one. The observed mean and standard deviation are different from their theoretical values, of course, so you test the hypothesis that there is no true difference.

Normal random number generator test.

```
x = normrnd(0,1,1,100);
[h,p,ci] = ttest(x,0)

h =
    0

p =
    0.4474

ci =
   -0.1165    0.2620
```

The result  $h = 0$  means that you cannot reject the null hypothesis. The significance level is 0.4474, which means that by chance you would have observed values of  $T$  more extreme than the one in this example in 45 of 100 similar experiments. A 95% confidence interval on the mean is  $[-0.1165 \ 0.2620]$ , which includes the theoretical (and hypothesized) mean of zero.

# ttest2

---

**Purpose** Hypothesis testing for the difference in means of two samples

**Syntax**

```
[h,significance,ci] = ttest2(x,y)
[h,significance,ci] = ttest2(x,y,alpha)
[h,significance,ci,stats] = ttest2(x,y,alpha)
[...] = ttest2(x,y,alpha,tail)
h = ttest2(x,y,alpha,tail,'unequal')
```

**Description** `h = ttest2(x,y)` performs a t-test to determine whether two samples from a normal distribution (in  $x$  and  $y$ ) could have the same mean when the standard deviations are unknown but assumed equal. The vectors  $x$  and  $y$  can have different lengths.

The result,  $h$ , is 1 if you can reject the null hypothesis that the means are equal at the 0.05 significance level and 0 otherwise.

`significance` is the p-value associated with the t-statistic

$$T = \frac{x - y}{s \sqrt{\frac{1}{n} + \frac{1}{m}}}$$

where  $s$  is the pooled sample standard deviation and  $n$  and  $m$  are the numbers of observations in the  $x$  and  $y$  samples. `significance` is the probability that the observed value of  $T$  could be as large or larger by chance under the null hypothesis that the mean of  $x$  is equal to the mean of  $y$ .

`ci` is a 95% confidence interval for the true difference in means.

`[h,significance,ci] = ttest2(x,y,alpha)` gives control of the significance level `alpha`. For example if `alpha = 0.01`, and the result,  $h$ , is 1, you can reject the null hypothesis at the significance level 0.01. `ci` in this case is a `100(1 - alpha)%` confidence interval for the true difference in means.

`[h,significance,ci,stats] = ttest2(x,y,alpha)` returns a structure `stats` with the following three fields:

- `tstat` — Value of the test statistic
- `df` — Degrees of freedom of the test

- 'sd' — Pooled estimate of the population standard deviation in the equal variance case, or a vector containing the unpooled estimates of the population standard deviations in the unequal variance case

[...] = ttest2(x,y,alpha,tail) allows specification of one- or two-tailed tests, where tail is a flag that specifies one of three alternative hypotheses:

- tail = 'both' specifies the alternative  $\mu_x \neq \mu_y$  (default)
- tail = 'right' specifies the alternative  $\mu_x > \mu_y$
- tail = 'left' specifies the alternative  $\mu_x < \mu_y$

h = ttest2(x,y,alpha,tail,'unequal') performs the test assuming that the two samples come from normal distributions with unknown and possibly unequal variances. This is known as the Behrens-Fisher problem. ttest2 uses Satterthwaite's approximation for the effective degrees of freedom.

## Examples

This example generates 100 normal random numbers with theoretical mean 0 and standard deviation 1. you then generate 100 more normal random numbers with theoretical mean 1/2 and standard deviation 1. The observed means and standard deviations are different from their theoretical values, of course. you test the hypothesis that there is no true difference between the two means. Notice that the true difference is only one half of the standard deviation of the individual observations, so you are trying to detect a signal that is only one half the size of the inherent noise in the process.

```
x = normrnd(0,1,100,1);
y = normrnd(0.5,1,100,1);

[h,significance,ci] = ttest2(x,y)

h =
    1

significance =

    0.0017

ci =
   -0.7352   -0.1720
```

The result h = 1 means that you can reject the null hypothesis. The significance is 0.0017, which means that by chance you would have observed

## ttest2

---

values of  $t$  more extreme than the one in this example in only 17 of 10,000 similar experiments! A 95% confidence interval on the mean is  $[-0.7352 -0.1720]$ , which includes the theoretical (and hypothesized) difference of  $-0.5$ .

---

<b>Purpose</b>	Discrete uniform cumulative distribution (cdf) function
<b>Syntax</b>	<code>P = unidcdf(X,N)</code>
<b>Description</b>	<p><code>P = unidcdf(X,N)</code> computes the discrete uniform cdf at each of the values in <code>X</code> using the corresponding parameters in <code>N</code>. <code>X</code> and <code>N</code> 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 <code>N</code> must be positive integers.</p> <p>The discrete uniform cdf is</p> $p = F(x N) = \frac{\text{floor}(x)}{N} I_{(1, \dots, N)}(x)$ <p>The result, <math>p</math>, is the probability that a single observation from the discrete uniform distribution with maximum <math>N</math> will be a positive integer less than or equal to <math>x</math>. The values <math>x</math> do not need to be integers.</p>
<b>Examples</b>	<p>What is the probability of drawing a number 20 or less from a hat with the numbers from 1 to 50 inside?</p> <pre>probability = unidcdf(20,50)  probability =      0.4000</pre>
<b>See Also</b>	<code>cdf</code> , <code>unidinv</code> , <code>unidpdf</code> , <code>unidrnd</code> , <code>unidstat</code>

# unidinv

---

**Purpose** Inverse of the discrete uniform cumulative distribution function

**Syntax** `X = unidinv(P,N)`

**Description** `X = unidinv(P,N)` returns the smallest positive integer `X` such that the discrete uniform cdf evaluated at `X` is equal to or exceeds `P`. You can think of `P` as the probability of drawing a number as large as `X` out of a hat with the numbers 1 through `N` inside.

`P` and `N` can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of `X`. A scalar input for `N` or `P` is expanded to a constant array with the same dimensions as the other input. The values in `P` must lie on the interval `[0 1]` and the values in `N` must be positive integers.

**Examples** `x = unidinv(0.7,20)`

```
x =  
    14
```

```
y = unidinv(0.7 + eps,20)
```

```
y =  
    15
```

A small change in the first parameter produces a large jump in output. The cdf and its inverse are both step functions. The example shows what happens at a step.

**See Also** `icdf`, `unidcdf`, `unidpdf`, `unidrnd`, `unidstat`

**Purpose** Discrete uniform probability density function (pdf)

**Syntax** `Y = unidpdf(X,N)`

**Description** `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

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

You can think of `y` as the probability of observing any one number between 1 and `n`.

**Examples** For fixed `n`, the uniform discrete pdf is a constant.

```
y = unidpdf(1:6,10)
```

```
y =
    0.1000    0.1000    0.1000    0.1000    0.1000    0.1000
```

Now fix `x`, and vary `n`.

```
likelihood = unidpdf(5,4:9)
```

```
likelihood =
    0    0.2000    0.1667    0.1429    0.1250    0.1111
```

**See Also** `pdf`, `unidcdf`, `unidinv`, `unidrnd`, `unidstat`

# unidrnd

---

**Purpose** Random numbers from the discrete uniform distribution

**Syntax**

```
R = unidrnd(N)
R = unidrnd(N,v)
R = unidrnd(N,m,n)
```

**Description** The discrete uniform distribution arises from experiments equivalent to drawing a number from one to N out of a hat.

`R = unidrnd(N)` generates discrete uniform random numbers 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 discrete uniform random numbers with maximum N, where 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 discrete uniform random numbers with maximum N, where scalars m and n are the row and column dimensions of R.

## Reproducing the Output of unidrnd

`unidrnd` uses the MATLAB function `rand` to generate random numbers. When you call `unidrnd`, you change the current state of `rand`, and thereby alter the output of subsequent calls to `unidrnd` or any other functions that depend on `rand`. If you want to reproduce the output of `unidrnd`, reset the state of `rand` to the same fixed value each time you call `unidrnd`. For an example of how to do this, and a list of the Statistics Toolbox functions that depend on `rand`, see “Reproducing the Output of Random Number Functions” on page 2-46.

---

**Note** The result in the following example depends on the current state of `rand`. If you run the code in this example, your results may differ from the answer shown here.

---

**Example** In the Massachusetts lottery, a player chooses a four digit number. Generate random numbers for Monday through Saturday.

```
numbers = unidrnd(10000,1,6) - 1
```



numbers =

2189

470

6788

6792

9346

## See Also

`unidcdf`, `unidinv`, `unidpdf`, `unidstat`

# unidstat

---

**Purpose** Mean and variance for the discrete uniform distribution

**Syntax** `[M,V] = unidstat(N)`

**Description** `[M,V] = unidstat(N)` returns the mean and variance for the discrete uniform distribution with parameter  $N$ .

The mean of the discrete uniform distribution with parameter  $N$  is  $(N + 1)/2$ .  
The variance is  $(N^2 - 1)/12$ .

**Examples**

```
[m,v] = unidstat(1:6)
```

```
m =
```

```
    1.0000    1.5000    2.0000    2.5000    3.0000    3.5000
```

```
v =
```

```
    0    0.2500    0.6667    1.2500    2.0000    2.9167
```

**See Also**

`unidcdf`, `unidinv`, `unidpdf`, `unidrnd`

**Purpose** Continuous uniform cumulative distribution function (cdf)

**Syntax** `P = unifcdf(X,A,B)`

**Description** `P = unifcdf(X,A,B)` computes the uniform cdf at each of the values in `X` using the corresponding parameters in `A` and `B` (the minimum and maximum values, respectively). `X`, `A`, and `B` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant matrix with the same dimensions as the other inputs.

The uniform cdf is

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

The standard uniform distribution has `A = 0` and `B = 1`.

**Examples** What is the probability that an observation from a standard uniform distribution will be less than 0.75?

```
probability = unifcdf(0.75)
```

```
probability =
```

```
0.7500
```

What is the probability that an observation from a uniform distribution with `a = -1` and `b = 1` will be less than 0.75?

```
probability = unifcdf(0.75, -1, 1)
```

```
probability =
```

```
0.8750
```

**See Also** `cdf`, `unifinv`, `unifit`, `unifpdf`, `unifrnd`, `unifstat`

# unifinv

---

**Purpose** Inverse continuous uniform cumulative distribution function (cdf)

**Syntax** `X = unifinv(P,A,B)`

**Description** `X = unifinv(P,A,B)` computes the inverse of the uniform cdf with parameters A and B (the minimum and maximum values, respectively) at the corresponding probabilities in P. P, A, and B can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs.

The inverse of the uniform cdf is

$$x = F^{-1}(p|a, b) = a + p(a - b)I_{[0, 1]}(p)$$

The standard uniform distribution has A = 0 and B = 1.

**Examples** What is the median of the standard uniform distribution?

```
median_value = unifinv(0.5)
```

```
median_value =
```

```
0.5000
```

What is the 99th percentile of the uniform distribution between -1 and 1?

```
percentile = unifinv(0.99, -1, 1)
```

```
percentile =
```

```
0.9800
```

**See Also** `icdf`, `unifcdf`, `unifit`, `unifpdf`, `unifrnd`, `unifstat`

<b>Purpose</b>	Parameter estimates for uniformly distributed data
<b>Syntax</b>	<pre>[ahat,bhat] = unifit(DATA) [ahat,bhat,ACI,BCI] = unifit(DATA) [ahat,bhat,ACI,BCI] = unifit(DATA,alpha)</pre>
<b>Description</b>	<p>[ahat,bhat] = unifit(DATA) returns the maximum likelihood estimates (MLEs) of the parameters of the uniform distribution given the data in DATA.</p> <p>[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.</p> <p>[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.</p>
<b>Example</b>	<pre>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</pre>
<b>See Also</b>	betafit, binofit, expfit, gamfit, normfit, poissfit, unifcdf, unifinv, unifpdf, unifrnd, unifstat, wblfit

# unifpdf

---

**Purpose** Continuous uniform probability density function (pdf)

**Syntax** `Y = unifpdf(X,A,B)`

**Description** `Y = unifpdf(X,A,B)` computes the continuous uniform pdf at each of the values in `X` using the corresponding parameters in `A` and `B`. `X`, `A`, and `B` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs. The parameters in `B` must be greater than those in `A`.

The continuous uniform distribution pdf is

$$y = f(x|a, b) = \frac{1}{b-a} I_{[a, b]}(x)$$

The standard uniform distribution has `A = 0` and `B = 1`.

**Examples** For fixed `a` and `b`, the uniform pdf is constant.

```
x = 0.1:0.1:0.6;  
y = unifpdf(x)  
  
y =  
    1    1    1    1    1    1
```

What if `x` is not between `a` and `b`?

```
y = unifpdf(-1,0,1)  
  
y =  
    0
```

**See Also** `pdf`, `unifcdf`, `unifinv`, `unifrnd`, `unifstat`

**Purpose** Random numbers from the continuous uniform distribution

**Syntax**

```
R = unifrnd(A,B)
R = unifrnd(A,B,m)
R = unifrnd(A,B,m,n)
```

**Description** `R = unifrnd(A,B)` generates uniform random numbers with parameters `A` and `B`. Vector or matrix inputs for `A` and `B` must have the same size, which is also the size of `R`. A scalar input for `A` or `B` is expanded to a constant matrix with the same dimensions as the other input.

`R = unifrnd(A,B,m)` generates uniform random numbers with parameters `A` and `B`, where `m` is a 1-by-2 vector that contains the row and column dimensions of `R`.

`R = unifrnd(A,B,m,n)` generates uniform random numbers with parameters `A` and `B`, where scalars `m` and `n` are the row and column dimensions of `R`.

### Reproducing the Output of `unifrnd`

`unifrnd` uses the MATLAB function `rand` to generate random numbers. When you call `unifrnd`, you change the current state of `rand`, and thereby alter the output of subsequent calls to `unifrnd` or any other functions that depend on `rand`. If you want to reproduce the output of `unifrnd`, reset the state of `rand` to the same fixed value each time you call `unifrnd`. For an example of how to do this, and a list of the Statistics Toolbox functions that depend on `rand`, see “Reproducing the Output of Random Number Functions” on page 2-46.

---

**Note** The results in the following examples depend on the current state of `rand`. If you run the code in these examples, your results may differ from the answers shown here.

---

### Examples

```
random = unifrnd(0,1:6)

random =

    0.2190    0.0941    2.0366    2.7172    4.6735    2.3010
```

# unifrnd

---

```
random = unifrnd(0,1:6,[1 6])
```

```
random =
```

```
    0.5194    1.6619    0.1037    0.2138    2.6485    4.0269
```

```
random = unifrnd(0,1,2,3)
```

```
random =
```

```
    0.0077    0.0668    0.6868  
    0.3834    0.4175    0.5890
```

## See Also

`unifcdf`, `unifinv`, `unifpdf`, `unifstat`



**Purpose** Mean and variance for the continuous uniform distribution

**Syntax** `[M,V] = unifstat(A,B)`

**Description** `[M,V] = unifstat(A,B)` returns the mean and variance for the continuous uniform distribution with parameters specified by A and B. Vector or matrix inputs for A and B must have the same size, which is also the size of M and V. A scalar input for A or B is expanded to a constant matrix with the same dimensions as the other input.

The mean of the continuous uniform distribution with parameters  $a$  and  $b$  is  $(a + b)/2$ , and the variance is  $(b - a)^2/12$ .

### Examples

```
a = 1:6;
b = 2.*a;
[m,v] = unifstat(a,b)

m =
    1.5000    3.0000    4.5000    6.0000    7.5000    9.0000

v =
    0.0833    0.3333    0.7500    1.3333    2.0833    3.0000
```

**See Also** `unifcdf`, `unifinv`, `unifpdf`, `unifrnd`

# var

---

**Purpose** Variance of a sample

**Syntax**  
`y = var(X)`  
`y = var(X,1)`  
`y = var(X,w)`  
`var(X,w,dim)`

**Description** `y = var(X)` computes the variance of the data in `X`. For vectors, `var(x)` is the variance of the elements in `x`. For matrices, `var(X)` is a row vector containing the variance of each column of `X`.

`y = var(x)` normalizes by  $n-1$  where  $n$  is the sequence length. For normally distributed data, this makes `var(x)` the minimum variance unbiased estimator MVUE of  $\sigma^2$  (the second parameter).

`y = var(x,1)` normalizes by  $n$  and yields the second moment of the sample data about its mean (moment of inertia).

`y = var(X,w)` computes the variance using the vector of positive weights `w`. The number of elements in `w` must equal the number of rows in the matrix `X`. For vector `x`, `w` and `x` must match in length.

`var(X,w,dim)` takes the variance along the dimension `dim` of `X`. Pass in 0 for `w` to use the default normalization by  $N-1$ , or 1 to use  $N$ .

`var` supports both common definitions of variance. Let  $SS$  be the sum of the squared deviations of the elements of a vector `x` from their mean. Then, `var(x) = SS/(n-1)` is the MVUE, and `var(x,1) = SS/n` is the maximum likelihood estimator (MLE) of  $\sigma^2$ .

## Examples

```
x = [-1 1];  
w = [1 3];  
v1 = var(x)  
  
v1 =  
    2  
  
v2 = var(x,1)  
  
v2 =  
    1
```

```
v3 = var(x,w)
```

```
v3 =  
    0.7500
```

**See Also**

cov, std

# wblcdf

---

**Purpose** Weibull cumulative distribution function (cdf)

**Syntax** `P = wblcdf(X, A, B)`  
`[P, PLO, PUP] = wblcdf(X, A, B, PCOV, alpha)`

**Description** `P = wblcdf(X, A, B)` computes the cdf of the Weibull distribution with scale parameter A and shape parameter B, at each of the values in X. X, A, and B can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array of the same size as the other inputs. The default values for A and B are both 1. The parameters A and B must be positive.

`[P, PLO, PUP] = wblcdf(X, A, B, PCOV, alpha)` returns confidence bounds for P when the input parameters A and B are estimates. PCOV is the 2-by-2 covariance matrix of the estimated parameters. alpha has a default value of 0.05, and specifies 100(1 - alpha)% confidence bounds. PLO and PUP are arrays of the same size as P containing the lower and upper confidence bounds.

The function `wblcdf` computes confidence bounds for P using a normal approximation to the distribution of the estimate

$$\hat{b}(\log x - \log \hat{a})$$

and then transforms those bounds to the scale of the output P. The computed bounds give approximately the desired confidence level when you estimate MU, SIGMA, and PCOV from large samples, but in smaller samples other methods of computing the confidence bounds might be more accurate.

The Weibull cdf is

$$p = F(x|a, b) = \int_0^x b a^{-b} t^{b-1} e^{-\left(\frac{t}{a}\right)^b} dt = 1 - e^{-\left(\frac{x}{a}\right)^b} I_{(0, \infty)}(x)$$

**Examples** What is the probability that a value from a Weibull distribution with parameters a = 0.15 and b = 0.8 is less than 0.5?

probability = `wblcdf(0.5, 0.15, 0.8)`

probability =

0.9272

How sensitive is this result to small changes in the parameters?

```
[A, B] = meshgrid(0.1:0.05:0.2,0.2:0.05:0.3);  
probability = wblcdf(0.5, A, B)  
probability =
```

```
0.7484    0.7198    0.6991  
0.7758    0.7411    0.7156  
0.8022    0.7619    0.7319
```

**See Also**

`cdf`, `wblfit`, `wblinv`, `wbllike`, `wblpdf`, `wblplot`, `wblrnd`, `wblstat`

**Purpose** Parameter estimates and confidence intervals for Weibull data

**Syntax**

```
parmhat = wblfit(data)
[parmhat, parmci] = wblfit(data)
[parmhat, parmci] = wblfit(data, alpha)
[...] = wblfit(data, alpha, censoring)
[...] = wblfit(data, alpha, censoring, freq)
[...] = wblfit(data, alpha, censoring, freq, 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

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

`[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. You can create options using the function `statset`. Enter `statset('wblfit')` to see the names and default values of the

parameters that `lognfit` accepts in the options structure. See the reference page for `statset` for more information about these options.

**Example**

```
data = wblrnd(0.5,0.8,100,1);  
[parmhat, parmci] = wblfit(data)  
  
parmhat =  
    0.5861    0.8567  
  
parmci =  
    0.4606    0.7360  
    0.7459    0.9973
```

**See Also**

`wblcdf`, `wblinv`, `wbllike`, `wblpdf`, `wblrnd`, `wblstat`, `mle`, `statset`

# wblinv

---

**Purpose** Inverse of the Weibull cumulative distribution function

**Syntax**  
`X = wblbinv(P, A, B)`  
`[X, XLO, XUP] = wblinv(P, A, B, PCOV, alpha)`

**Description** `X = wblinv(P, A, B)` returns the inverse cumulative distribution function (cdf) for a Weibull distribution with scale parameter A and shape parameter B, evaluated at the values in P. P, A, and B can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array of the same size as the other inputs. The default values for A and B are both 1.

`[X, XLO, XUP] = wblinv(P, A, B, PCOV, alpha)` returns confidence bounds for X when the input parameters A and B are estimates. PCOV is a 2-by-2 matrix containing the covariance matrix of the estimated parameters. alpha has a default value of 0.05, and specifies 100(1 - alpha)% confidence bounds. XLO and XUP are arrays of the same size as X containing the lower and upper confidence bounds.

The function `wblinv` computes confidence bounds for X using a normal approximation to the distribution of the estimate

$$\log \hat{a} - \frac{\log q}{\hat{b}}$$

where  $q$  is the  $p$ th quantile from a Weibull distribution with scale and shape parameters both equal to 1. The computed bounds give approximately the desired confidence level when you estimate MU, SIGMA, and PCOV from large samples, but in smaller samples other methods of computing the confidence bounds might be more accurate.

The inverse of the Weibull cdf is

$$x = F^{-1}(p|a, b) = \left[ a \ln \left( \frac{1}{1-p} \right) \right]^{\frac{1}{b}} I_{[0,1]}(p)$$

## Examples

The lifetimes (in hours) of a batch of light bulbs has a Weibull distribution with parameters  $a = 200$  and  $b = 6$ . What is the median lifetime of the bulbs?

```
life = wblinv(0.5, 200, 6)
```



```
life =  
    188.1486
```

What is the 90th percentile?

```
life = wblinv(0.9, 200, 6)
```

```
life =  
    229.8261
```

**See Also**

wblcdf, wblfit, wbllike, wblpdf, wblrnd, wblstat, icdf

# wbllike

---

**Purpose** Weibull negative log-likelihood function

**Syntax**

```
nlogL = wbllike(params, data)
[nlogL, 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.

**Example** This example continues the example from `wblfit`.

```
r = wblrnd(0.5,0.8,100,1);
[logL, AVAR] = wbllike(wblfit(r),r)
```

logL =

47.3349

AVAR =

0.0048    0.0014

0.0014    0.0040

## Reference

[1] Patel, J.K., C.H. Kapadia, and D.B. Owen, *Handbook of Statistical Distributions*, Marcel-Dekker, 1976.

## See Also

betalike, gamlike, mle, normlike, wblcdf, wblfit, wblinv, wblpdf, wblrnd, wblstat

# wblpdf

---

**Purpose** Weibull probability density function (pdf)

**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

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

Some references refer to the Weibull distribution with a single parameter. This corresponds to `wblpdf` with `A = 1`.

**Examples** The exponential distribution is a special case of the Weibull distribution.

```
lambda = 1:6;  
y = wblpdf(0.1:0.1:0.6, lambda, 1)  
y =  
    0.9048    0.4524    0.3016    0.2262    0.1810    0.1508  
y1 = exppdf(0.1:0.1:0.6, lambda)  
y1 =  
    0.9048    0.4524    0.3016    0.2262    0.1810    0.1508
```

**Reference** [1] Devroye, L., *Non-Uniform Random Variate Generation*, Springer-Verlag, New York, 1986.

**See Also** `pdf`, `wblcdf`, `wblfit`, `wblinv`, `wbllike`, `wblplot`, `wblrnd`, `wblstat`

**Purpose** Weibull probability plot

**Syntax** `wblplot(X)`  
`h = wblplot(X)`

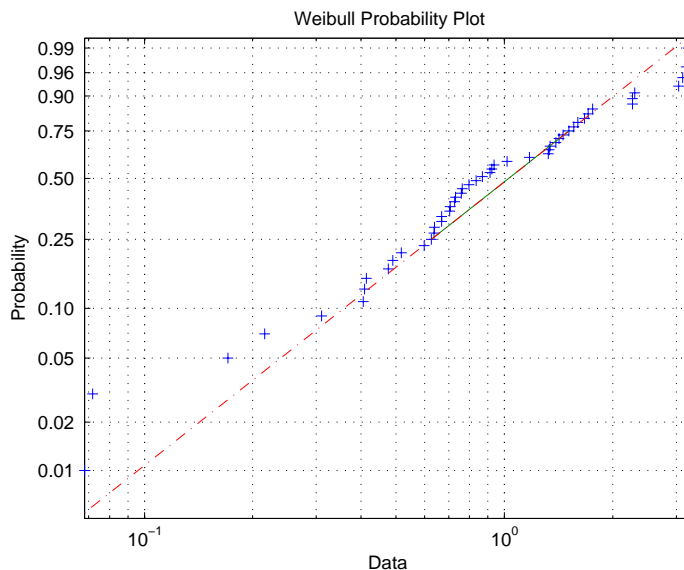
**Description** `wblplot(X)` displays a Weibull probability plot of the data in `X`. If `X` is a matrix, `wblplot` displays a plot for each column.

`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.

**Example**

```
r = wblrnd(1.2,1.5,50,1);  
wblplot(r)
```



**See Also** `normplot`, `wblcdf`, `wblfit`, `wblinv`, `wbllike`, `wblpdf`, `wblrnd`, `wblstat`

# wblrnd

---

**Purpose** Random numbers from the Weibull distribution

**Syntax**

```
R = wblrnd(A,B)
R = wblrnd(A,B,v)
R = wblrnd(A,B,m,n)
```

**Description** `R = wblrnd(A,B)` generates Weibull random numbers 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 Weibull random numbers 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 Weibull random numbers 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`.

## Reproducing the Output of wblrnd

`wblrnd` uses the MATLAB function `rand` to generate random numbers. When you call `wblrnd`, you change the current state of `rand`, and thereby alter the output of subsequent calls to `wblrnd` or any other functions that depend on `rand`. If you want to reproduce the output of `wblrnd`, reset the state of `rand` to the same fixed value each time you call `wblrnd`. For an example of how to do this, and a list of the Statistics Toolbox functions that depend on `rand`, see “Reproducing the Output of Random Number Functions” on page 2-46.

---

**Note** The results in the following examples depend on the current state of `rand`. If you run the code in these examples, your results might differ from the answers shown here.

---

**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
```

**Reference**

[1] Devroye, L., *Non-Uniform Random Variate Generation*, Springer-Verlag, New York, 1986.

**See Also**

wblcdf, wblfit, wblinv, wbllike, wblpdf, wblplot, wblstat

# wblstat

---

**Purpose** Mean and variance for the Weibull distribution

**Syntax** `[M,V] = wblstat(A,B)`

**Description** `[M,V] = wblstat(A,B)` returns the mean 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

$$a^2[\Gamma(1 + 2b^{-1}) - \Gamma(1 + b^{-1})^2]$$

**Examples** `[m,v] = wblstat(1:4,1:4)`

m =

1.0000 1.7725 2.6789 3.6256

v =

1.0000 0.8584 0.9480 1.0346

`wblstat(0.5,0.7)`

ans =

0.6329

**See Also** `wblcdf`, `wblfit`, `wblinv`, `wbllike`, `wblpdf`, `wblplot`, `wblrnd`



**Purpose** Generate Wishart random matrix

**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`.

### Reproducing the Output of wishrnd

`wishrnd` uses the MATLAB functions `rand` and `randn` to generate random numbers. When you call `wishrnd`, you change the current states of `rand` and `randn`, and thereby alter the output of subsequent calls to `wishrnd` or any other functions that depend on `rand` or `randn`. If you want to reproduce the output of `wishrnd`, reset the states of `rand` and `randn` to the same fixed values each time you call `wishrnd`. For an example of how to do this, and a list of the Statistics Toolbox functions that depend on `rand` or `randn`, see “Reproducing the Output of Random Number Functions” on page 2-46.

**See Also** `iwishrnd`

**Purpose** Transform a factor settings matrix to a design matrix

**Syntax**  $D = \text{x2fx}(X)$   
 $D = \text{x2fx}(X, 'model')$

**Description**  $D = \text{x2fx}(X)$  transforms a matrix of system inputs,  $X$ , to a design matrix for a linear additive model with a constant term.

$D = \text{x2fx}(X, 'model')$  allows control of the order of the regression model. 'model' can be one of these strings:

'interaction'	Includes constant, linear, and cross product terms
'quadratic'	Includes interactions and squared terms
'purequadratic'	Includes constant, linear, and squared terms

Alternatively model can be a matrix of terms. In this case, each row of model represents one term. The value in a column is the exponent to which the same column in  $X$  for that term is raised,  $D(i, j) = \text{prod}(x(i, :).^{\text{model}(j, :)})$ . This allows for models with polynomial terms of arbitrary order.

The order of columns for a quadratic model is:

- 1 Constant term
- 2 Linear terms (the input  $X$  columns 1, 2, ...,  $k$ )
- 3 Interaction terms formed by taking pairwise products of  $X$  columns (1, 2), (1, 3), ..., (1,  $k$ ), (2, 3), ..., ( $k-1$ ,  $k$ )
- 4 Squared terms in the order 1, 2, ...,  $k$

Other models use a subset of these terms but keep them in this order.

$\text{x2fx}$  is a utility function for `rstool`, `regstats`, and `cordexch`.

## Examples

### Example 1.

```
x = [1 2 3]';  
model = [0 1 2]';  
D = x2fx(x,model)
```

```
D =
```

1	1	1
1	2	4
1	3	9

The first column is  $x$  to the 0th power. The second column is  $x$  to the 1st power. And the last column is  $x$  squared.

**Example 2.**

```
x = [1 2 3;4 5 6]';
model = 'quadratic';
D = x2fx(x,model)
```

D =

1	1	4	4	1	16
1	2	5	10	4	25
1	3	6	18	9	36

Let  $x_1$  be the first column of  $x$  and  $x_2$  be the second. Then the first column of  $D$  is the constant term, the second column is  $x_1$ , the third column is  $x_2$ , the fourth column is  $x_1x_2$ , the fifth column is  $x_1^2$ , and the last columns is  $x_2^2$ .

**See Also**

rstool, candexch, candgen, cordexch, rowexch,

# xbarplot

---

**Purpose** X-bar chart for Statistical Process Control

**Syntax**

```
xbarplot(DATA)
xbarplot(DATA,conf)
xbarplot(DATA,conf,specs,'sigmaest')
[outlier,h] = xbarplot(...)
```

**Description** `xbarplot(DATA)` displays an x-bar chart of the grouped responses in `DATA`. The rows of `DATA` contain replicate observations taken at a given time, and must be in time order. The graph contains the sample mean  $\bar{x}$  for each group, a center line at the average  $\bar{x}$  value, and upper and lower control limits. The limits are placed at a three-sigma distance on either side of the center line, where sigma is an estimate of the standard deviation of  $\bar{x}$ . If the process is in control, fewer than 3 out of 1000 observations would be expected to fall outside the control limits by random chance. So if you observe points outside the limits, you can take this as evidence that the process is not in control.

`xbarplot(DATA,conf)` allows control of the confidence level of the upper and lower plotted confidence limits. The default `conf = 0.9973` produces three-sigma limits.

```
norminv(1 - (1-.9973)/2)
ans =
     3
```

To get  $k$ -sigma limits, use the expression `1-2*(1-normcdf(k))`. For example, the correct `conf` value for 2-sigma limits is 0.9545, as shown below.

```
k = 2;
1-2*(1-normcdf(k))
ans =
    0.9545
```

`xbarplot(DATA,conf,specs)` plots the specification limits in the two element vector `specs`.

`xbarplot(DATA,conf,specs,'sigmaest')` specifies how `xbarplot` should estimate the standard deviation. Acceptable values are:

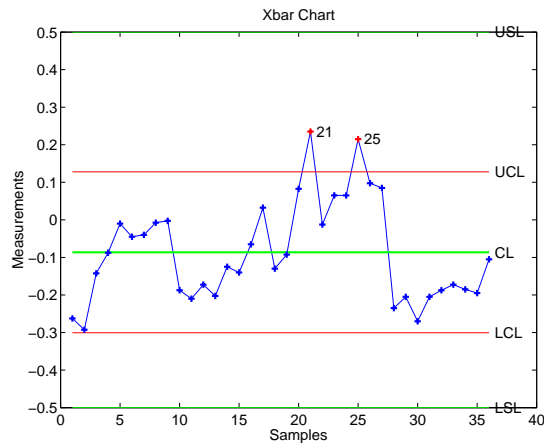
- 's' – use the average of the group standard deviations (default)
- 'v' – use the square root of a pooled variance estimate
- 'r' – use the average range with each group; requires 25 or fewer observations per group

`[outlier,h] = xbarplot(DATA,conf,specs)` returns `outlier`, a vector of indices to the rows where the mean of `DATA` is out of control, and `h`, a vector of handles to the plotted lines.

## Example

Plot an x-bar chart of measurements on newly machined parts, taken at one hour intervals for 36 hours. Each row of the runout matrix contains the measurements for four parts chosen at random. The values indicate, in thousandths of an inch, the amount the part radius differs from the target radius.

```
load parts
xbarplot(runout,0.999,[-0.5 0.5])
```



The points in groups 21 and 25 are out of control, so the mean in those groups was higher than would be expected by random chance alone. There is evidence that the process was not in control when those measurements were collected.

## See Also

`capaplot`, `histfit`, `ewmplot`, `schart`

## zscore

---

**Purpose** Standardized Z score

**Syntax** `Z = zscore(D)`

**Description** `Z = zscore(D)` returns the deviation of each column of D from its mean, normalized by its standard deviation. This is known as the Z score of D.

For column vector V, the Z score is  $Z = (V - \text{mean}(V)) ./ \text{std}(V)$ .

**Purpose** Hypothesis testing for the mean of one sample with known variance

**Syntax**

```
h = ztest(x,m,sigma)
h = ztest(x,m,sigma,alpha)
[h,sig,ci,zval] = ztest(x,m,sigma,alpha,tail)
```

**Description** `h = ztest(x,m,sigma)` performs a Z test at significance level 0.05 to determine whether a sample `x` from a normal distribution with standard deviation `sigma` could have mean `m`.

`h = ztest(x,m,sigma,alpha)` gives control of the significance level `alpha`. For example, if `alpha = 0.01` and the result is `h = 1`, you can reject the null hypothesis at the significance level 0.01. If `h = 0`, you cannot reject the null hypothesis at the `alpha` level of significance.

`[h,sig,ci] = ztest(x,m,sigma,alpha,tail)` allows specification of one- or two-tailed tests, where `tail` is a flag that specifies one of three alternative hypotheses:

- `tail = 'both'` specifies the alternative  $\bar{x} \neq m$  (default)
- `tail = 'right'` specifies the alternative  $x > m$
- `tail = 'left'` specifies the alternative  $x < m$

`zval` is the value of the Z statistic

$$z = \frac{\bar{x} - m}{\sigma / \sqrt{n}}$$

where  $n$  is the number of observations in the sample.

`sig` is the probability that the observed value of Z could be as large or larger *by chance* under the null hypothesis that the mean of  $x$  is equal to  $m$ .

`ci` is a  $1 - \alpha$  confidence interval for the true mean.

**Example** This example generates 100 normal random numbers with theoretical mean zero and standard deviation one. The observed mean and standard deviation are different from their theoretical values, of course. you test the hypothesis that there is no true difference.

## ztest

---

```
x = normrnd(0,1,100,1);
m = mean(x)
m =
    0.0727

[h,sig,ci] = ztest(x,0,1)

h =
    0

sig =
    0.4669

ci =
   -0.1232    0.2687
```

The result,  $h = 0$ , means that you cannot reject the null hypothesis. The significance level is 0.4669, which means that by chance you would have observed values of  $Z$  more extreme than the one in this example in 47 of 100 similar experiments. A 95% confidence interval on the mean is  $[-0.1232 \ 0.2687]$ , which includes the theoretical (and hypothesized) mean of zero.



# Selected Bibliography

---

- [1] Atkinson, A.C., and A.N. Donev, *Optimum Experimental Designs*, Oxford Science Publications 1992.
- [2] Bates, D. and D. Watts. *Nonlinear Regression Analysis and Its Applications*, John Wiley and Sons. 1988. pp. 271–272.
- [3] Bernoulli, J., *Ars Conjectandi*, Basileia: Thurnisius [11.19], 1713
- [4] Box, G.E.P. and N.R. Draper, *Empirical Model-Building and Response Surfaces*, Wiley, New York. 1987.
- [5] Box, G.E.P., W.G. Hunter, and J.S. Hunter. *Statistics for Experimenters*. Wiley, New York. 1978.
- [6] Chatterjee, S. and A.S. Hadi. *Influential Observations, High Leverage Points, and Outliers in Linear Regression*. Statistical Science, 1986. pp. 379–416.
- [7] Dobson, A. J., *An Introduction to Generalized Linear Models*, 1990, CRC Press.
- [8] Durbin, R., S. Eddy, A. Krogh, and G. Mitchison, *Biological Sequence Analysis*, Cambridge Univ. Press, 1998.
- [9] Efron, B., and R.J. Tibshirani. *An Introduction to the Bootstrap*, Chapman and Hall, New York, 1993.
- [10] Evans, M., N. Hastings, and B. Peacock. *Statistical Distributions, Second Edition*. John Wiley and Sons, 1993.
- [11] Hald, A., *Statistical Theory with Engineering Applications*, John Wiley and Sons, 1960. p. 647.
- [12] Hogg, R.V., and J. Ledolter. *Engineering Statistics*. MacMillan Publishing Company, 1987.
- [13] Johnson, N., and S. Kotz. *Distributions in Statistics: Continuous Univariate Distributions*. John Wiley and Sons, 1970.
- [14] McCullagh, P., and J. A. Nelder, *Generalized Linear Models*, 2nd edition, 1990, Chapman and Hall.
- [15] Montgomery, D.C., *Design and Analysis of Experiments*, Wiley, New York. 1984.

---

[16] Moore, J., *Total Biochemical Oxygen Demand of Dairy Manures*. Ph.D. thesis. University of Minnesota, Department of Agricultural Engineering, 1975.

[17] Poisson, S.D., *Recherches sur la Probabilité des Jugements en Matière Criminelle et en Matière Civile, Précédées des Règles Générales du Calcul des Probabilités*. Paris: Bachelier, Imprimeur-Libraire pour les Mathématiques, 1837.

[18] "Student," *On the Probable Error of the Mean*. *Biometrika*, 6:1908. pp. 1-25.

[19] Weibull, W., *A Statistical Theory of the Strength of Materials*. Ingeniors Vetenskaps Akademiens Handlingar, Royal Swedish Institute for Engineering Research. Stockholm, Sweden, No. 153. 1939.



**A**

absolute deviation 3-4  
 addedvarplot 12-20  
 additive effects 4-8  
 alternative hypothesis 6-3  
 analysis of variance 2-62  
   multivariate 7-24  
   N-way 4-11  
   one-way 4-3  
   two-way 4-8  
 andrewsplot 12-22  
 ANOVA 4-2  
 anova1 12-24  
 anova2 **12-30**  
 anovan **12-34**  
 aoctool **12-43**  
 aoctool demo 4-25  
 average linkage 12-279

**B**

bacteria counts 4-3  
 barttest **12-46**  
 baseball odds 12-57, 12-60  
 bbdesign **12-47**  
 Bera-Jarque. *See* Jarque-Bera  
 Bernoulli random variables 12-62  
 beta distribution 2-48  
 betacdf **12-48**  
 betafit **12-49**  
 betainv **12-51**  
 betalike **12-52**  
 betapdf **12-53**  
 betarnd **12-54**  
 betastat **12-56**  
 binocdf **12-57**  
 binofit **12-58**

binoinv **12-60**  
 binomial distribution 2-50  
   negative. *See also* negative binomial  
     distribuiton  
 binopdf **12-61**  
 binornd **12-62**  
 binostat **12-64**  
 biplot 12-65  
 bootstrap **12-67**  
 bootstrap sampling 3-17  
 box plots 8-3  
 Box-Behnken designs 10-9  
   generating 12-47  
 boxplot **12-69**

**C**

candexch **12-74**  
 candgen **12-76**  
 canocorr **12-77**  
 capability studies 9-5  
 capable **12-80**  
 capaplot **12-82**  
 casenames  
   reading from file 12-84  
   writing to file 12-85  
 caseread **12-84**  
 casewrite **12-85**  
 ccdesign **12-86**  
 cdf  
   definition 2-7  
   functions 12-4  
 cdf **12-88**  
 cdfplot **12-89**  
 central composite designs 10-8  
   generating 12-86

- Central Limit Theorem 2-74
  - centroid linkage 12-280
  - Chatterjee and Hadi example 4-35
  - chi2cdf **12-91**
  - chi2inv **12-92**
  - chi2pdf **12-93**
  - chi2rnd **12-94**
  - chi2stat **12-96**
  - chi-square distributions 2-52
  - circuit boards 12-61
  - City Block metric
    - in cluster analysis 12-404
  - classical multidimensional scaling 7-53
    - cmdsca function 12-104
    - overview 7-53
    - reconstructing a map 7-60
    - simple example 7-54
  - classification trees 5-8
    - functions 12-15
    - See also* decision trees
  - classify **12-97**
  - cluster **12-100**
  - cluster analysis 7-30
    - functions 12-14
    - hierarchical clustering 7-30
    - K-means clustering 7-46
  - cluster tree creation 12-278
    - from data 12-102
    - from linkage output 12-100
  - cluster trees
    - inconsistency coefficient 12-246
    - plotting 12-124
  - clusterdata **12-102**
  - cmdsca **12-104**
  - coin 12-194
  - combnk **12-106**
  - comparisons, multiple 4-5
  - complete linkage 12-279
  - confidence intervals
    - hypothesis tests 6-3
    - nonlinear regression 5-5
  - control charts 9-2
    - EWMA charts 9-4
    - S charts 9-3
    - Xbar charts 9-2
  - cophenet **12-107**
  - cophenetic correlation coefficient 12-107
    - defined 7-37
  - cordexch **12-109**
  - corr 12-111
  - corrcoef **12-113**
  - correlation coefficients 12-113
  - cov **12-116**
  - Cp index 9-5, 12-80
  - Cpk index 9-5, 12-80
  - crosstab **12-117**
  - cumulative distribution function (cdf) 2-7
    - empirical 3-15
    - functions 12-4
    - graphing an estimate 8-8
  - curve fitting
    - polynomials
      - demo 4-36
- ## D
- data partitioning
    - K-means clustering 7-46
  - data sets
    - statistical examples 12-18
  - daugment **12-119**
  - dcovary **12-121**
  - decision trees 5-8
    - computing error rate 12-517

- computing response values 12-520
  - creating 12-511
  - creating subtrees 12-514
  - displaying 12-509
  - example 5-8
  - fitting 12-511
  - functions 12-15
  - pruning 12-514
- demos
- design of experiments 10-10
  - list of 12-17
- dendrogram **12-124**, 12-301
- density estimation
- ksdensity function 12-260
- descriptive statistics 3-1
- functions 12-9
- design of experiments 10-1
- Box-Behnken designs 10-9
  - central composite designs 10-8
  - D-optimal designs 10-18
  - fractional factorial designs 10-6
  - full factorial designs 10-4
  - functions 12-13
  - response surface designs 10-8
- dimension reduction
- common factor analysis 12-155
  - PCA from covariance matrix 12-397
  - PCA from raw data matrix 12-420
  - PCA residuals 12-399
- discrete uniform distribution 2-55
- dissimilarity matrix
- creating 7-31
- distance matrix
- creating 7-31
- distribution fitting tool 2-13
- distribution testing
- functions 12-16
- distributions
- probability 2-1
  - supported 2-45
  - visualizing 2-3
- disttool **12-127**
- disttool demo 2-3
- DOE. *See* design of experiments
- D-optimal designs 10-18
- creating from candidate set 12-74
  - generating candidate set 12-76
- dummyvar **12-128**
- E**
- ecdf **12-129**
- ecdfhist 12-131
- efinv 12-137
- empirical cumulative distribution function 3-15
- ecdf function 12-129
- erf 2-74
- error function 2-74
- errorbar **12-133**
- Euclidean distance
- in cluster analysis 12-403
- evcdf 12-134
- evfit 12-135
- evlike 12-138
- evpdf 12-139
- evrnd 12-140
- evstat 12-141
- EWMA charts 9-4
- ewmaplot **12-142**
- expcdf **12-144**
- expfit **12-146**
- expinv **12-148**
- explike 12-150
- exponential distribution 2-56

exppdf **12-151**  
exprnd **12-152**  
expstat **12-154**  
extrapolated 12-425  
extreme value distribution 2-58

## F

F distributions 2-61  
F statistic 4-36  
factor analysis  
    maximum likelihood 12-155  
factoran **12-155**  
factorial designs  
    fractional 10-6  
    full 10-4  
    generating fractional 12-168  
    generating full 12-181  
fcdf **12-164**  
ff2n **12-165**  
file I/O functions 12-17  
finv **12-166**  
floppy disks 12-239  
fpdf **12-167**  
fracfact **12-168**  
fractional factorial designs 10-6  
    generating 12-168  
friedman **12-172**  
Friedman's test 4-60  
frnd **12-176**  
fstat **12-178**  
fsurfht **12-179**  
full factorial designs 10-4  
    generating 12-181  
fullfact **12-181**  
furthest neighbor linkage 12-279

## G

gamcdf **12-182**  
gamfit **12-184**  
gaminv **12-186**  
gamlike **12-188**  
gamma distribution 2-64  
gampdf **12-190**  
gamrnd **12-191**  
gamstat **12-193**  
Gaussian 12-223  
geocdf **12-194**  
geoinv **12-195**  
geomean **12-196**  
geometric distribution 2-66  
geopdf **12-197**  
geornd **12-198**  
geostat **12-200**  
gline **12-201**  
glmdemo **12-202**  
glmdemo demo 4-54  
glmfit **12-203**  
glmval **12-208**  
glyphplot 12-210  
gname **12-214**  
gplotmatrix **12-216**  
group mean clusters, plot 7-29  
grouped plot matrix 7-24  
grpstats **12-219**  
gscatter **12-220**  
Guinness beer 2-79, 12-504

## H

harmmean **12-222**  
hat matrix 4-34  
hidden Markov models 11-1  
    example 11-4



functions 12-16

hierarchical clustering 7-30

- basic procedure 7-31
- computing inconsistency coefficient 12-246
- creating cluster tree 12-278
- creating clusters 7-42
- creating clusters from data 12-102
- creating clusters from linkage output 12-100
- depth of comparison 7-38
- determining proximity 12-402
- evaluating cluster formation 12-107
- finding dissimilarities between objects 7-38
- finding similarities between objects 7-31
- grouping objects 7-34
- inconsistency coefficient 12-246
- plotting cluster trees 12-124

hierarchiical clustering

- cophenetic correlation coefficient 12-107

hist **12-223**

hist3 12-224

histfit **12-226**

hmmdecode 12-227

hmmestimate 12-229

hmmgenerate 12-231

hmmtrain 12-233

hmmviterbi 12-236

Hotelling's T squared 7-11

hougen **12-238**

Hougen-Watson model 5-2

hygecdf **12-239**

hygeinv **12-240**

hygepdf **12-241**

hygernd **12-242**

hygestat **12-244**

hypergeometric distribution 2-67

hypotheses 2-62

hypothesis tests 6-1

functions 12-16

## I

icdf **12-245**

incomplete beta function 2-48

incomplete gamma function 2-64

inconsistency coefficient 12-246

inconsistent **12-246**

inspector 12-407

interaction 4-8

interpolated 12-498

interquartile range (iqr) 3-4

inverse cdf 2-7

inverse cumulative distribution

- functions 12-6

iqr **12-248**

iwishrnd **12-249**

## J

Jarque-Bera test 12-250

jbtest **12-250**

## K

Kaplan-Meier cumulative distribution function

- 12-129

kernel bandwidth 3-12

kernel smoothing function 3-14

K-means clustering 7-46

- cluster separation 7-47
- creating clusters 12-252
- example 7-47
- local minima 7-51
- number of clusters 7-49
- overview 7-46

silhouette plot 12-481  
kmeans **data partitioning**  
    K-means clustering 12-252  
kruskalwallis **12-256**  
Kruskal-Wallis test 4-59  
ksdensity **12-260**  
kstest **12-263**  
kstest2 **12-268**  
kurtosis **12-271**

**L**

latin hypercube sample 12-274  
    normal distribution 12-275  
least squares 12-416  
leverage **12-273**  
lhsdesign **12-274**  
lhsnorm **12-275**  
light bulbs, life of 12-148  
likelihood function 12-53  
Lilliefors test 6-5  
lillietest **12-276**  
linear models 4-1  
    functions 12-12  
    generalized 4-50  
linear transformation  
    Procrustes 12-423  
linkage **12-278**  
logncdf **12-282**  
lognfit 12-284  
logninv **12-286**  
lognlike 12-288  
lognormal distribution 2-69  
lognpdf **12-289**  
lognrnd **12-290**  
lognstat **12-292**  
lottery 12-534

lsline **12-293**  
LU factorizations 12-415

**M**

mad **12-294**  
mahal **12-296**  
Mahalanobis distance 12-296  
    in cluster analysis 12-404  
MANOVA 7-24  
manova1 **12-297**  
manovacluster **12-301**  
Markov chains 11-6  
maximum likelihood  
    factor analysis 12-155  
MDS  
    *See also* multidimensional scaling  
mdscale 12-303  
mean 2-11  
    of probability distribution 2-11  
mean **12-306**  
Mean Squares (MS) 12-24  
measures of  
    central tendency 3-2  
    dispersion 3-4  
median **12-307**  
metric multidimensional scaling  
    *See also* classical multidimensional scaling  
Minkowski metric  
    in cluster analysis 12-404  
mle **12-308**  
mlecov 12-314  
models  
    linear 4-1  
    nonlinear 5-1  
moment **12-317**  
moments of distribution

- functions 12-8
  - Monte Carlo simulation 12-248
  - multcompare **12-318**
  - multidimensional arrays
    - classical (metric) scaling 12-104
  - multidimensional scaling (MDS)
    - classical (metric) 7-53
  - multiple linear regression 4-33
  - multivariate analysis of variance 7-24
    - example 7-24
  - multivariate statistics 7-1
    - analysis of variance 7-24
    - cluster analysis 7-30
    - functions 12-14
    - hierarchical clustering 7-30
    - K-means clustering 7-46
    - Principal Components Analysis 7-2
  - mvnpdf **12-327**
  - mvnrnd **12-328**
  - mvtrnd **12-330**
- N**
- nancov 12-332
  - nanmax **12-334**
  - nanmean **12-335**
  - nanmedian **12-336**
  - nanmin **12-337**
  - NaNs 3-6
  - nanstd **12-338**
  - nansum **12-340**
  - nanvar 12-341
  - nbincdf **12-342**
  - nbinfit **12-344**
  - nbininv **12-345**
  - nbinpdf **12-346**
  - nbinrnd **12-348**
  - nbinstat **12-350**
  - ncfcdf **12-352**
  - ncfinv **12-354**
  - ncfpdf **12-356**
  - ncfrnd **12-357**
  - ncfstat **12-359**
  - nctcdf **12-360**
  - nctinv **12-361**
  - nctpdf **12-362**
  - nctrnd **12-363**
  - nctstat **12-365**
  - ncx2cdf **12-366**
  - ncx2inv **12-368**
  - ncx2pdf **12-369**
  - ncx2rnd **12-370**
  - ncx2stat **12-372**
  - nearest neighbor linkage 12-279
  - negative binomial distribution 2-70
    - confidence intervals 12-344
    - cumulative distribution function (cdf) 12-342
    - definition 2-71
    - inverse cumulative distribution function (cdf) 12-345
    - mean and variance 12-350
    - modeling number of auto accidents 2-71
  - nbincdf function 12-342
  - nbininv function 12-345
  - nbinpdf function 12-346
  - parameter estimates 12-344
  - probability density function (pdf) 12-346
  - random matrices 12-348
- Newton's method 12-186
- nlinfit **12-373**
  - nlintool **12-375**
  - nlintool demo 5-6
  - nlparci **12-377**
  - nlpredci **12-378**

- noncentral F distribution 2-63
- nonlinear regression
  - functions 12-13
- nonlinear regression models 5-1
- nonparametric testing
  - functions 12-16
- normal distribution 2-73
- normal probability plots 8-2, 8-4
- normalizing a dataset 7-32
  - using zscore 12-564
- normcdf **12-380**
- normdemo **12-392**
- normfit **12-382**
- norminv **12-384**
- normlike **12-386**
- normpdf **12-387**
- normplot **12-388**
- normrnd **12-390**
- normstat **12-393**
- notation, mathematical conventions 1-6
- null 6-3
- null hypothesis 6-3
  
- O**
- one-way analysis of variance (ANOVA) 4-2
- options parameters
  - possible values 12-489
- outliers 3-2
  
- P**
- parallelcoords 12-394
- parameter estimation
  - functions 12-3
- pareto **12-396**
- Pascal, Blaise 2-50
  
- PCA. *See* Principal Components Analysis
- pcacov **12-397**
- pcares **12-399**
- pdf
  - definition 2-6
  - functions 12-5
- pdf **12-401**
- pdist **12-402**
- percentiles 3-10
- perms **12-406**
- plots 3-10
- plotting
  - statistical functions 12-10
- poisscdf **12-407**
- poissfit **12-409**
- poissinv **12-410**
- Poisson distribution 2-76
- poisspdf **12-411**
- poissrnd **12-412**
- poisstat **12-414**
- polyconf **12-415**
- polyfit **12-416**
- polynomials
  - curve fitting demo 4-36
- polytool **12-417**
- polytool demo 4-36
- polyval **12-418**
- popcorn 12-32, 12-174
- prctile **12-419**
- Principal Components Analysis (PCA) 7-2
  - component scores 7-6
  - component variances 7-10
  - Hotelling's T squared 7-11
  - principal components 7-6
  - quality of life example 7-3
  - Scree plot 7-10
- princomp **12-420**

- probability density estimation 3-12
    - comparing estimates 3-15
    - function 12-260
    - kernel bandwidth 3-12
    - kernel smoothing function 3-14
  - probability density function (pdf)
    - definition 2-6
    - functions 12-5
  - probability distribution
    - mean and variance 2-11
  - probability distributions 2-1
    - demo 2-3
    - functions 12-2
  - probplot 12-422
  - process control
    - statistical 9-1
  - procrustes **12-423**
  - Procrustes Analysis 12-423
  - p-value 4-10, 6-3
- Q**
- qqplot **12-425**
  - QR decomposition 4-33
  - quality assurance 12-61
  - quantile-quantile plots 8-2, 8-6
- R**
- random **12-428**
  - random number generation 2-9
    - direct 2-9
    - functions 12-6
    - inverted 2-9
    - rejection 2-10
  - random numbers 2-9
  - random sample generation
    - demo 2-4
  - random samples
    - inverse Wishart 12-249
    - latin hypercube 12-274
    - latin hypercube with normal distribution 12-275
    - Wishart 12-559
  - randsample 12-429
  - randtool **12-430**
  - randtool demo 2-4
  - range **12-431**
  - ranksum **12-432**
  - raylcdf **12-434**
  - raylfit 12-435
  - raylinv **12-436**
  - raylpdf **12-437**
  - raylrnd **12-438**
  - raylstat **12-440**
  - rcoplot **12-441**
  - reconstruction
    - map from inter-city distances 7-60
  - refcurve **12-442**
  - references A-1
  - refline **12-443**
  - regress **12-444**
  - regression 2-62
    - nonlinear 5-1
    - robust 4-55
    - stepwise 4-45
  - regression trees 5-8
    - functions 12-15
    - See also* decision trees
  - regstats **12-447**
  - relative efficiency 12-248
  - residuals 4-36
  - response surface designs 10-8
  - Response Surface Methodology (RSM) 4-42

response surface designs  
  Box-Behnken 10-9  
  central composite 10-8  
ridge **12-460**  
robust 3-2  
robust linear fit 12-425  
robustdemo demo 4-57  
robustdemo function **12-462**  
robustfit **12-463**  
rotatefactors 12-467  
rowexch 12-470  
rsmdemo **12-472**  
rsmdemo demo 10-10  
R-square 4-36  
rstool **12-473**  
rstool demo 4-42

## S

S charts 9-3  
scaling arrays  
  classical multidimensional 12-104  
scatter plots 8-10  
  grouped 7-24  
schart **12-475**  
Scree plot 7-10  
segmentation analysis 7-30  
significance level 6-3  
signrank **12-477**  
sigttest **12-479**  
silhouette **12-481**  
similarity matrix  
  creating 7-31  
simulation 12-248  
single linkage 12-279  
skewness 8-3  
skewness **12-483**

squareform **12-485**  
standard normal 12-387  
Standardized Euclidean distance  
  in cluster analysis 12-403  
statget 12-487  
statistical plots 8-1  
Statistical Process Control 9-1  
  capability studies 9-5  
  control charts 9-2  
  functions 12-11  
statistical references A-1  
statistically significant 12-24, 12-256, 12-297  
statset 12-488  
stepwise 4-45, **12-494**  
stepwise regression 4-45  
stepwisefit 12-495  
Sum of Squares (SS) 12-24  
surfht **12-498**  
symmetric 12-182

## T

t distributions 2-79  
  noncentral 2-80  
tab-delimited data  
  reading from file 12-505  
tabular data  
  reading from file 12-500  
tabulate **12-499**  
taxonomy analysis 7-30  
tblread **12-500**  
tblwrite **12-502**  
tcdf **12-504**  
tdfread **12-505**  
tinv **12-507**  
tpdf **12-508**  
treefit **12-511**

treeprune **12-514**  
trees  
    *See also* decision trees  
    *See also* decision trees  
treeshow **12-509**  
treetest **12-517**  
treeval **12-520**  
trimmean **12-522**  
trnd **12-523**  
tstat **12-525**  
ttest **12-526**  
ttest2 **12-528**  
two-way ANOVA 4-8

## U

unbiased 12-544  
unidcdf **12-531**  
unidinv **12-532**  
unidpdf **12-533**  
unidrnd **12-534**  
unidstat **12-536**  
unifcdf **12-537**  
unifinv **12-538**  
unifit **12-539**  
uniform distribution 2-82  
unifpdf **12-540**  
unifrnd **12-541**  
unifstat **12-543**

## V

var **12-544**  
variance 2-11  
    of probability distribution 2-11

## W

ward linkage 12-280  
wblcdf 12-546  
wblfit 12-548  
wblinv 12-550  
wbllike 12-552  
wblpdf 12-554  
wblplot **12-555**  
wblrnd **12-556**  
wblstat 12-558  
Weibull distribution 2-83  
Weibull probability plots 8-7  
Weibull, Waloddi 2-83  
whiskers 8-3  
Wishart random matrix 12-559  
    inverse 12-249  
wishrnd **12-559**

## X

x2fx **12-560**  
Xbar charts 9-2  
xbarplot **12-562**

## Z

zscore **12-564**  
ztest **12-565**

