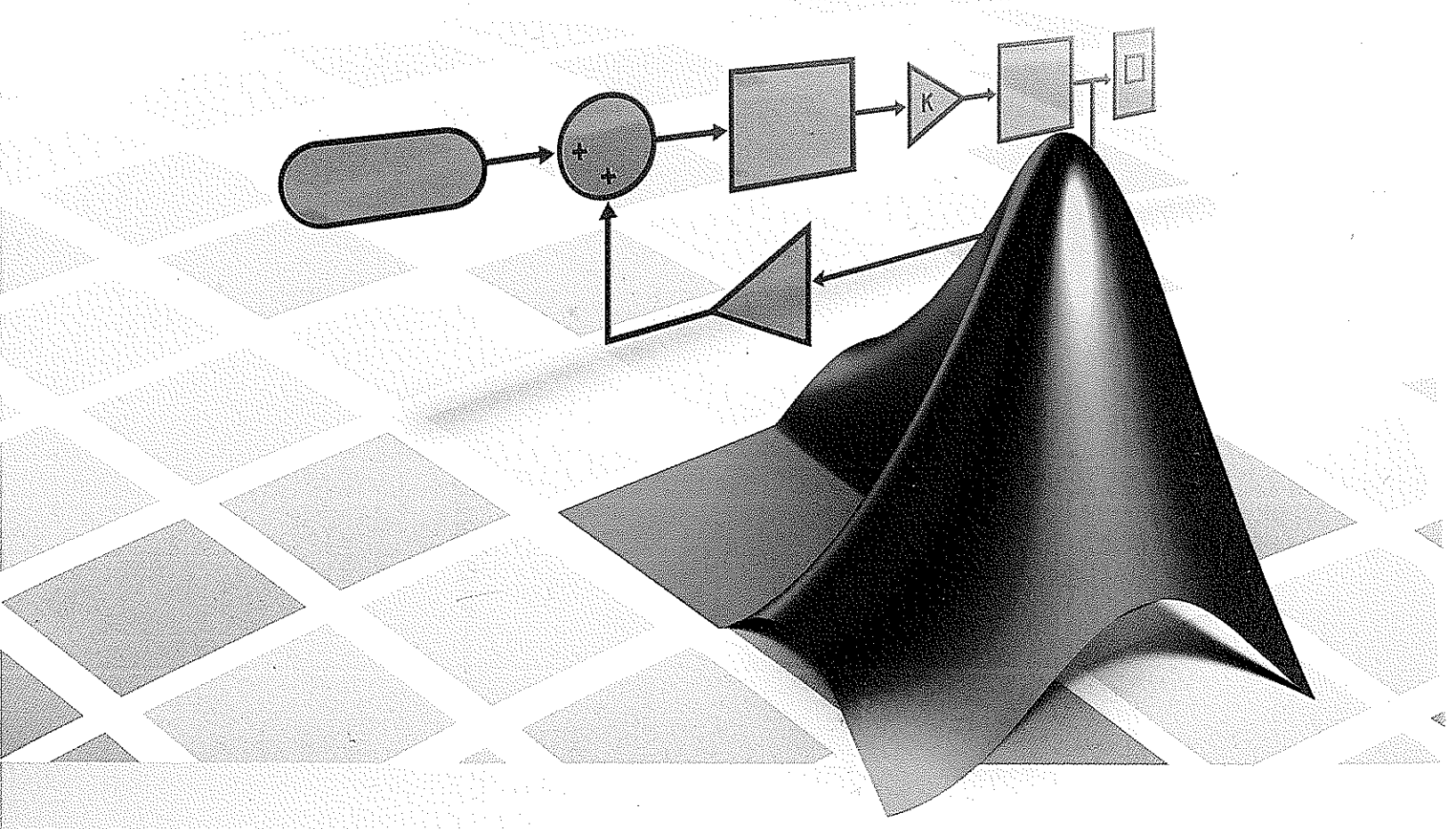
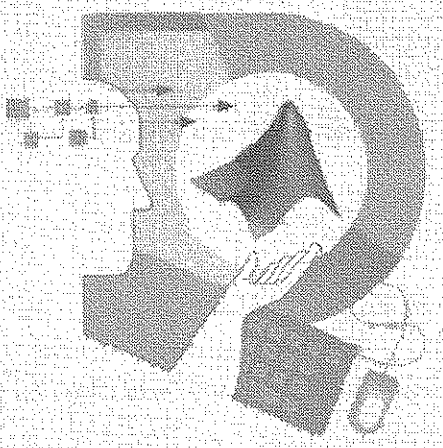
Statistical Methods in MATLAB®



The MathWorks

Statistical Methods in MATLAB®

Table of Contents



The MathWorks
Training Services

ST01 2009V1 Jan

Table of Contents

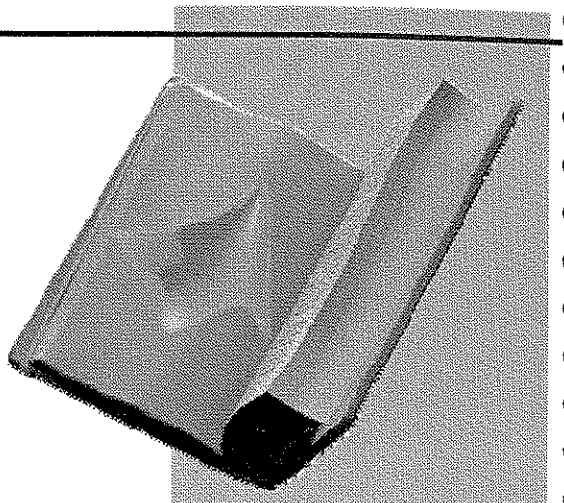


Table of Contents

1. Introduction

Table of Contents

2. Data and Statistics

Table of Contents

3. Probability and Distributions

Table of Contents

4. Regression Analysis

Table of Contents

5. Multivariate Statistics

Table of Contents

6. Random Numbers and Simulation

Table of Contents

7. Inferential Statistics

Table of Contents

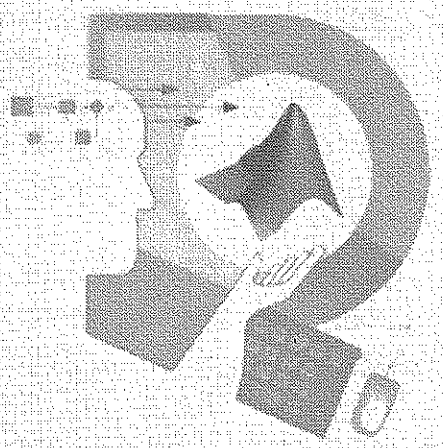
8. Conclusion

Table of Contents

Appendices

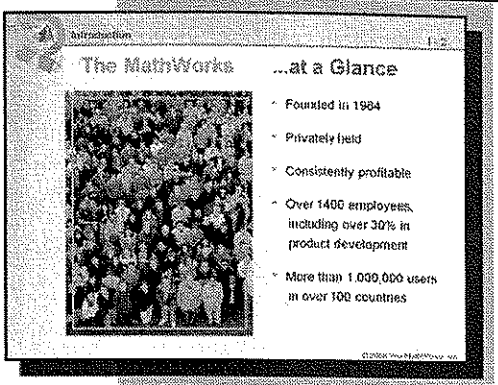
Statistical Methods in MATLAB®

Introduction

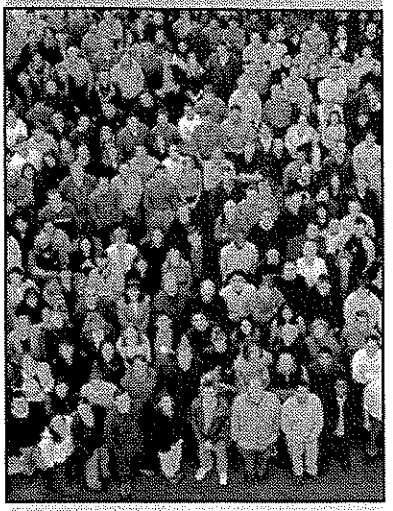


The MathWorks
Training Services

The MathWorks at a Glance



The MathWorks is the leading developer and supplier of technical computing software in the world. Founded in 1984, it now employs over 1,400 people worldwide, over 30% of whom are directly involved in product development. The company is privately held and has been profitable every year since its inception.



The MathWorks

Jack Little and Cleve Moler, the founders of The MathWorks, recognized the need among engineers and scientists for more powerful and productive computation environments beyond those provided by languages such as Fortran and C. In response to that need, they combined their expertise in mathematics, engineering, and computer science to develop MATLAB®, a high-performance technical computing environment. MATLAB combines comprehensive math and graphics functions with a powerful high-level language. In addition to MATLAB, The MathWorks now develops and markets Simulink®, a product for simulating linear and nonlinear dynamic systems. The MathWorks also develops and markets an extensive family of add-on products to meet the application-specific needs of scientists, engineers, and educators.

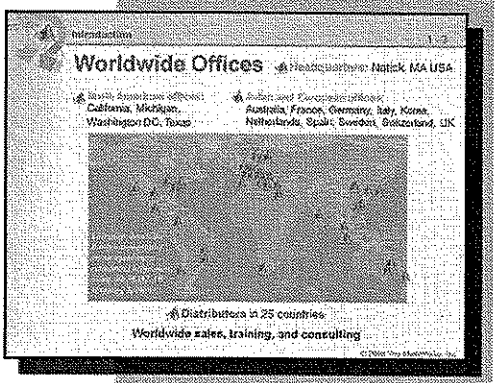
Jack Little
President

Cleve Moler
Chief Scientist

The guiding principle at The MathWorks is "Do the Right Thing." This means doing what is best for our staff members, customers, business partners, and communities for the long term, and believing that "right" answers exist. It means measuring our success not merely in financial terms, but also by how consistently we act according to this principle. Our mission and core values statements express what "doing the right thing" means in our day-to-day work.

The MathWorks also has a social mission: "We will be active members of our communities, promote social responsibility, and encourage environmental awareness." The MathWorks people actively participate in realizing the social mission.

Worldwide Offices



This map shows the Earth's topography on an equidistant cylindrical projection, using the Mapping Toolbox.

The MathWorks customers include 1,000,000 of the world's technical leaders, in over 100 countries on all seven continents (including Antarctica). These people work at the world's most innovative technology companies, government research labs, financial institutions, and at more than 3,500 universities. They rely on The MathWorks because MATLAB and Simulink have become the standard throughout science and industry.

The MathWorks supports its customers through a worldwide network of offices, distributors, and resellers.

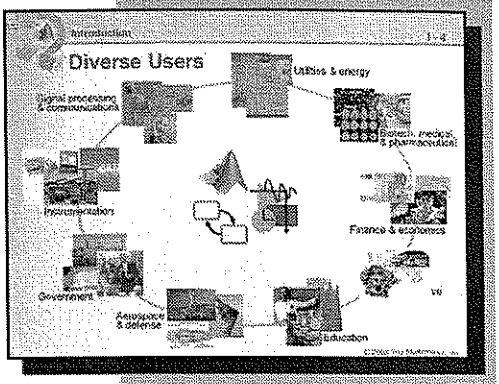
Headquarters (Natick, MA USA)
508-647-7000
`www.mathworks.com`
`support@mathworks.com`

Worldwide Offices

Region/Country	Phone	Web
Australia The MathWorks Australia Pty. Ltd	+61 2 8669 4700	`www.mathworks.au` `support@mathworks.au`
France The MathWorks S.A.S.	+33 (0)1 41 14 67 14	`www.mathworks.fr` `support@mathworks.fr`
Germany The MathWorks GmbH	+49 241 470 750	`www.mathworks.de` `support@mathworks.de`
Italy The MathWorks s.r.l.	+39 011 2274 700	`www.mathworks.it` `support@mathworks.it`
Republic of Korea The MathWorks LLC	+82 (0)2 6006 5114	`www.mathworks.co.kr` `support@mathworks.co.kr`
Netherlands The MathWorks, BV	+31 (0) 182 696 700	`www.mathworks.nl` `support@mathworks.nl`
Spain The MathWorks, S.L.	+34 91 799 18 80	`www.mathworks.es` `support@mathworks.es`
Sweden The Mathworks AB	+46 8 5053 17 00	`www.mathworks.se` `support@mathworks.se`
Switzerland The MathWorks GmbH	+41 (0) 31 950 60 20	`www.mathworks.ch` `support@mathworks.ch`
United Kingdom The MathWorks Ltd.	+44 1223 226 700	`www.mathworks.co.uk` `support@mathworks.co.uk`

Diverse Users



In the last few years, MathWorks tools have been used to:

- Improve the safety and efficiency of spacecraft docking by developing adaptive neurocontrol technology for a computer-aided joystick control system
- Advance the mapping of the human genome by developing algorithms for DNA sequencing instruments
- Avert financial crises in emerging economies by building an econometric model to predict significant volatility
- Advance the diagnosis and treatment of gastrointestinal tract disorders by improving visual imaging of the small intestine
- Test the dynamic position of ships in heavy seas with simulations using scale models in the laboratory
- Verify the legality of currency by scanning images of the notes and identifying the small fibers that they contain
- Improve the quality of next-generation network audio products by simulating signals transmitted over a network
- Enable temperate crops to be grown in dry coastal regions by designing a greenhouse that converts seawater into fresh water
- Improve race car performance by designing a system for the automatic testing of suspension systems
- Teach computer programming to undergraduates by developing a test and measurement laboratory that poses authentic engineering problems to the students
- Create images of unexplored underwater archeology and geology sites by mapping plankton density relative to water masses
- Translate the distorted human voice in high-pressure environments by lowering the frequency and pitch of the sound made by the larynx

Key Industries

Aerospace and defense

Automotive

Biotech, medical, and pharmaceutical

Chemical and petroleum

Computers and office equipment

Education

Electronics and semiconductors

Finance and economics

Government

Industrial equipment and machinery

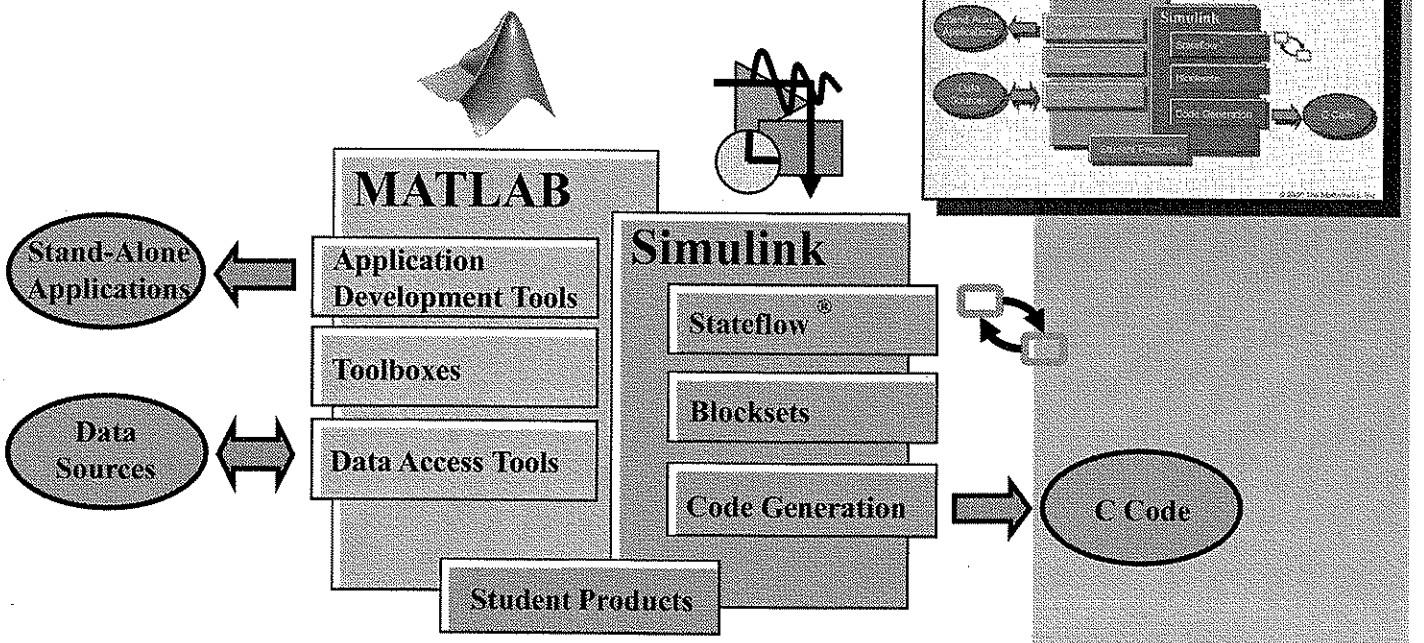
Instrumentation

Marine

Signal processing and communications

Utilities and energy

The MathWorks Family



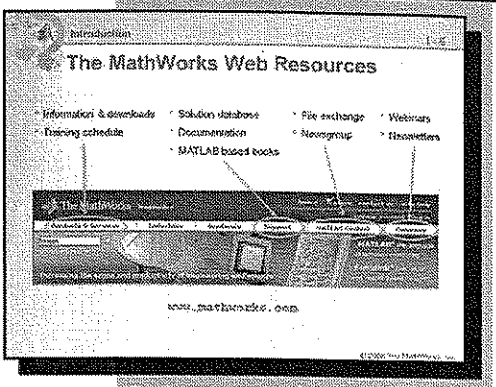
Some key characteristics of the MATLAB language:

- A user-friendly, intuitive syntax, favoring brevity and simplicity without compromising intelligibility
- The highest quality numerical algorithms, based on close historical ties with the numerical analysis research community
- Powerful, easy-to-use graphics and visualization capabilities
- A high-level language, making it possible to carry out computations in a line or two that would require hundreds of lines of code in Fortran or C
- Easy extensibility, by the user or via packages of application-specific M-files and GUIs known as toolboxes
- Real and complex vectors and matrices (including sparse matrices) as fundamental data types

Some Key Characteristics of Simulink:

- A complete environment for modeling, simulating, and implementing dynamic and embedded systems
- Design and test linear, nonlinear, discrete-time, continuous-time, hybrid, and multirate systems
- Applications in controls, DSP, communications, and systems engineering
- Open architecture allows integration of models from other environments

The MathWorks Web Resources



The MathWorks Web site at `www.mathworks.com` contains a wealth of resources beyond the materials provided for this course*. Among them:

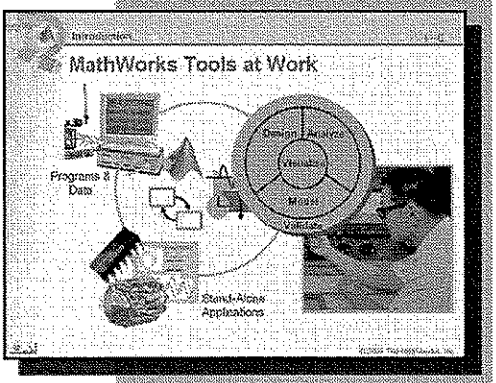
- Information on products and downloads
  `www.mathworks.com/products`
- A searchable tech support database
  `www.mathworks.com/support`
- Live and recorded Webinars on MathWorks tools and applications
  `www.mathworks.com/company/events`
- MATLAB Central file exchange, newsgroups, and contests
  `www.mathworks.com/matlabcentral`
- MATLAB based books
  `www.mathworks.com/support/books`
- Numerical Computing with MATLAB
  `www.mathworks.com/moler`
- MATLAB newsletters
  `www.mathworks.com/company/newsletters`
- The MathWorks consulting services
  `www.mathworks.com/consulting`
- Up-to-date information on training courses, dates, and locations
  `www.mathworks.com/training`
- Complete product documentation
  `www.mathworks.com/access/helpdesk/help/helpdesk.shtml`

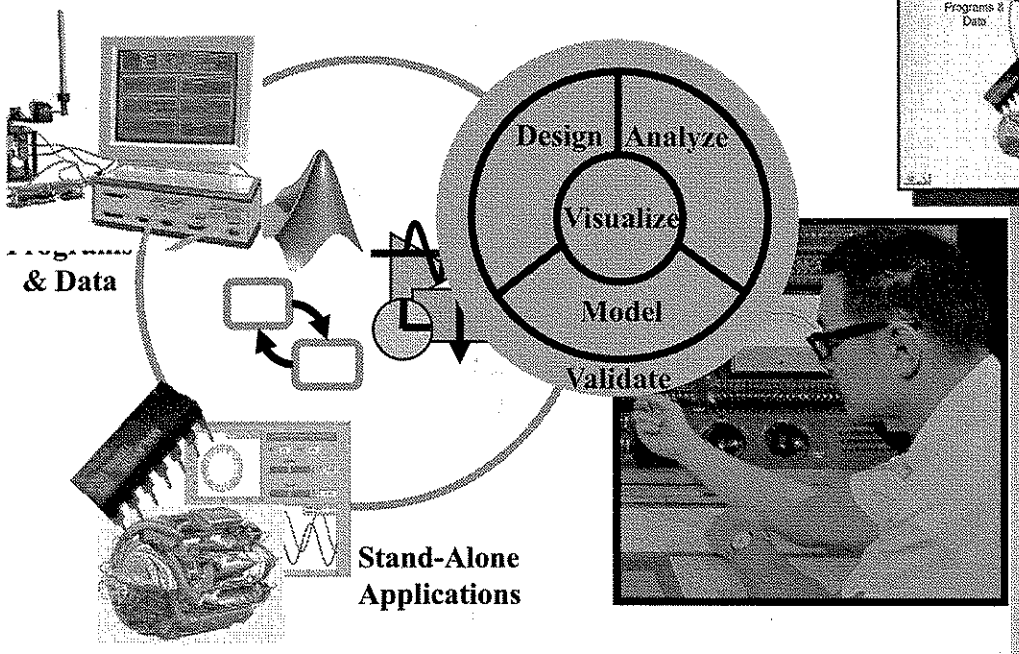
Try

`www.mathworks.com`

* For international Web sites, see page 1-3.

MathWorks Tools at Work





The job of any scientist or engineer involves design, modeling, and analysis. In all aspects of that job, data visualization plays a central role. A continuous validation of results circumscribes the process.

Day-to-day decisions are based on interactions with programs and data. The ultimate goal may be to deploy designs in hardware or stand-alone applications. Documentation is crucial at every stage.

MathWorks tools offer a unified, integrated environment for the entire scientific and engineering process. They combine broad analytic capabilities, built-in documentation features, and easy-to-use interfaces to external programs and data. In this environment, implementation is a natural extension of development.

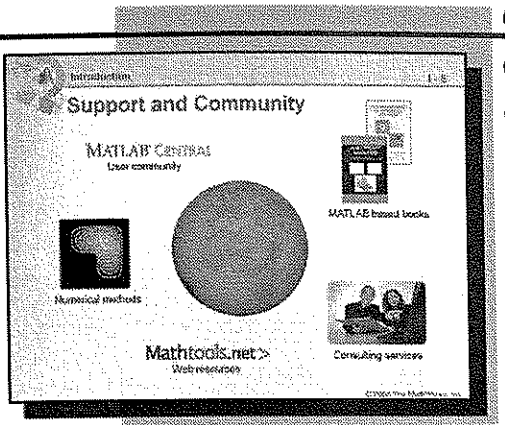
In the context of **this course**, the components of the scientific and engineering process will take on specific meanings. For example,

- We will **design** a Monte Carlo simulation of a stochastic process.
- We will **analyze** trends in multidimensional data.
- We will **model** the distribution of a random variable.
- We will **visualize** grouped and clustered data.

The MathWorks Model-Based Calibration Toolbox allows engineers with the task of developing engine management algorithms to run simulations that refine tradeoffs until the desired performance is reached. The big advantage to using this toolbox is that the engine to be simulated only needs to be run once on a dynamometer for characterization. The rest of the work can be done at the desktop, which saves a lot of expensive dyno time.

Automotive Design & Production, Dec. 2003

Support and Community

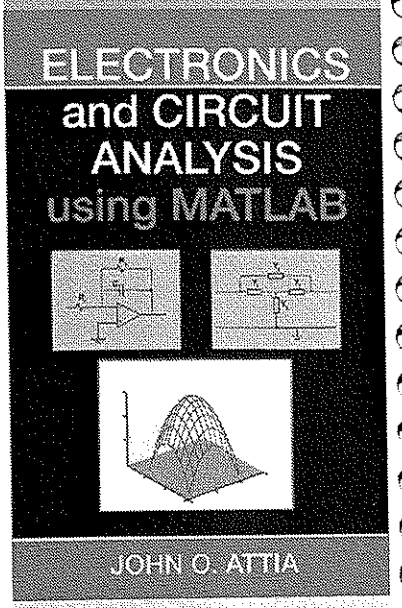


You are connected to a world of creative MATLAB users, both inside and outside of The MathWorks, for information, support, and community.

MATLAB Central* provides a single location for users of MathWorks products to exchange information directly with each other in forums specific to dozens of application areas. A file exchange contains thousands of user-created extensions of MathWorks products with helpful ideas to get you started on your next project. A newsgroup, `comp.soft-sys.matlab`, is read by thousands of users worldwide, including people from the MathWorks development and technical support departments.

Mathtools.net (`www.mathtools.net`) provides links to technical computing resources on MATLAB, programming, applications, industries, and education.

MATLAB based books* include more than 700 books in 20 languages. The texts present theory, real-world examples, and exercises using MATLAB, Simulink, and other MathWorks products. They provide reference for researchers in academia and industry, tools for practicing engineers, and course material for instructors in engineering, science, and mathematics.



MATLAB based books are available in a wide variety of application areas.

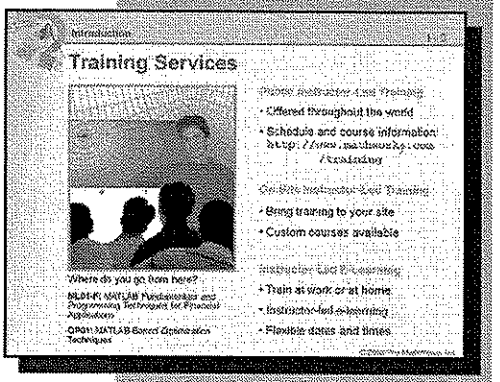
Numerical Computing with MATLAB*, written by MATLAB creator Cleve Moler, is a Web-based text explaining the numerical methods behind MATLAB. It provides more than 70 M-files and more than 200 exercises. Those adopting the textbook for a course can register for access to curriculum tools and materials, including a solutions manual and a set of slides for use in classroom lectures.

MathWorks consulting services* provides specific, project-based services including start-up services, application development, model-based and system-level design, embedded-systems development, enterprise-wide integration, and product migration.

---

* See 1 – 6 for URL.

Training Services



MathWorks Training Services can help you use our products to succeed in your work. Our training courses are developed around the core responsibilities of engineers, scientists, and educators. Our trainers focus on your goals and how to utilize MathWorks tools to achieve them.

The MathWorks offers introductory and intermediate courses in MATLAB, Simulink, and Stateflow, as well as advanced courses in subjects such as control design, signal and image processing, test and measurement, optimization, statistics, and financial analysis.

Where do you go from here?

Additional courses related to this course

OP01: MATLAB-Based Optimization Techniques

ML01-F: MATLAB Fundamentals and Programming Techniques for Financial Applications

Public instructor-led training

Public instructor-led courses are offered in North America, France, Germany, Italy, the Republic of Korea, the Netherlands, Spain, Switzerland, and the United Kingdom. In addition, many of our distributors offer training at other international sites. So, whether you are in Winnipeg or Wien, you can find hands-on training near you.

On-site instructor-led training

The MathWorks also offers custom training courses, which can be taught at your own facility. Our engineers can incorporate company- or industry-specific examples into a curriculum of your choosing.

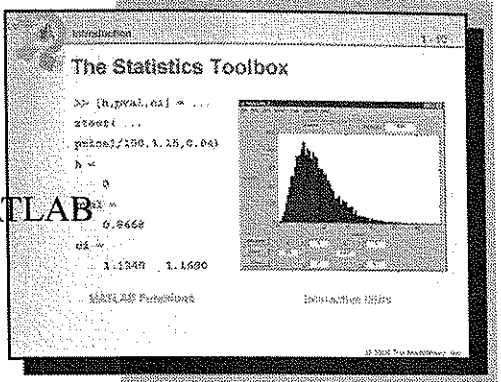
Instructor-led e-learning

All of our training courses (except those requiring specialized hardware) are also offered over the Web. An instructor in one of our offices can share his/her desktop with you and communicate over the phone. You get the same quality instruction with the convenience of being in your home or office. Our e-learning courses are also typically run with fewer students, on a more flexible schedule.

For course information and the latest training schedule and locations:
`www.mathworks.com/training`

Don't see it on the schedule? Contact us and we'll make it happen:
`training@mathworks.com`

The Statistics Toolbox



The Statistics Toolbox is a collection of tools built on the MATLAB numeric computing environment.

The toolbox supports a wide range of common statistical tasks, from calculating sample statistics, to statistical plotting, data analysis, and hypothesis testing.

The toolbox provides two principal categories of tools:

- Building-block probability and statistics functions
- Interactive graphical user interfaces (GUIs)

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

```
>> type function_name
```

You can change the way any toolbox function works by copying and renaming the M-file, then modifying your copy by typing

```
>> edit function_name
```

You can also extend the toolbox by adding your own M-files.

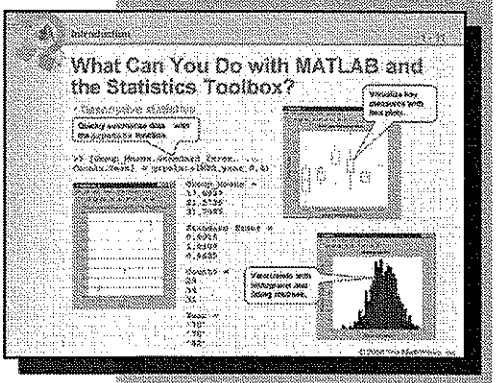
The second category of tools is made up of interactive GUIs. GUI-based tools provide an unified environment for accessing a number of related toolbox functions. The Statistics Toolbox contains GUIs for polynomial fitting and prediction, probability density function exploration, and random number generation, among others.

GUIs are also easily modified or extended using the MATLAB Graphical User Interface Design Environment (GUIDE) tool.

A Sampling of Statistics Toolbox functions

```
anoval
boxplot
cdf
cluster
corrcoef
cov
hist
kurtosis
manoval
mean
median
moment
multcompare
nlinfit
normpdf
perms
polyfit
prctile
random
regress
skewness
std
tblread
tblwrite
ttest
var
ztest
```

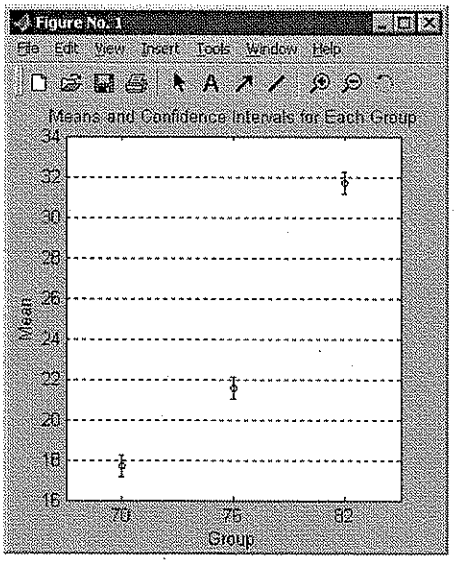
What Can You Do with MATLAB and the Statistics Toolbox?



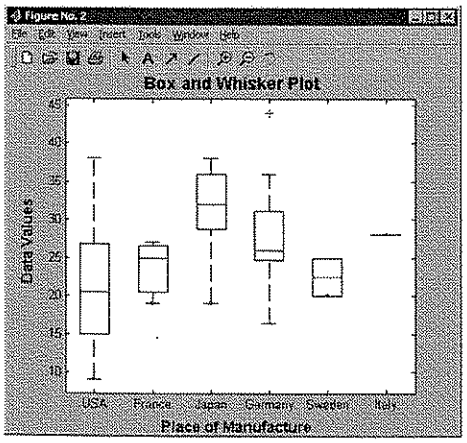
Descriptive statistics

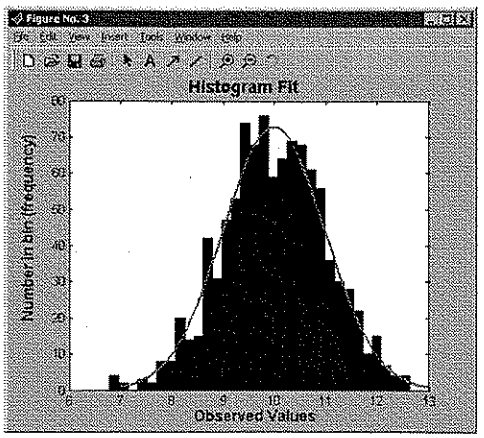
- Quickly summarize data with the `grpstats` function.

```
>> [Group_Means,Standard_Error, ...
Counts,Year] = grpstats(MPG,year,0.6)

Group_Means =
       17.6897
       21.5735
       31.7097

Standard Error =
       0.9915
       1.0100
       0.9685

Counts =
       29
       34
       31

Year =
       '70'
       '76'
       '82'
```



- Visualize key measures with box plots.
- View trends with histograms and fitting routines.

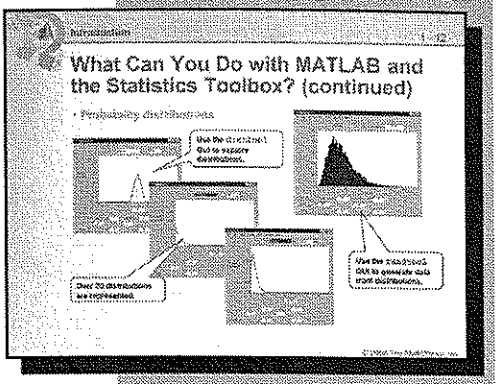




Try

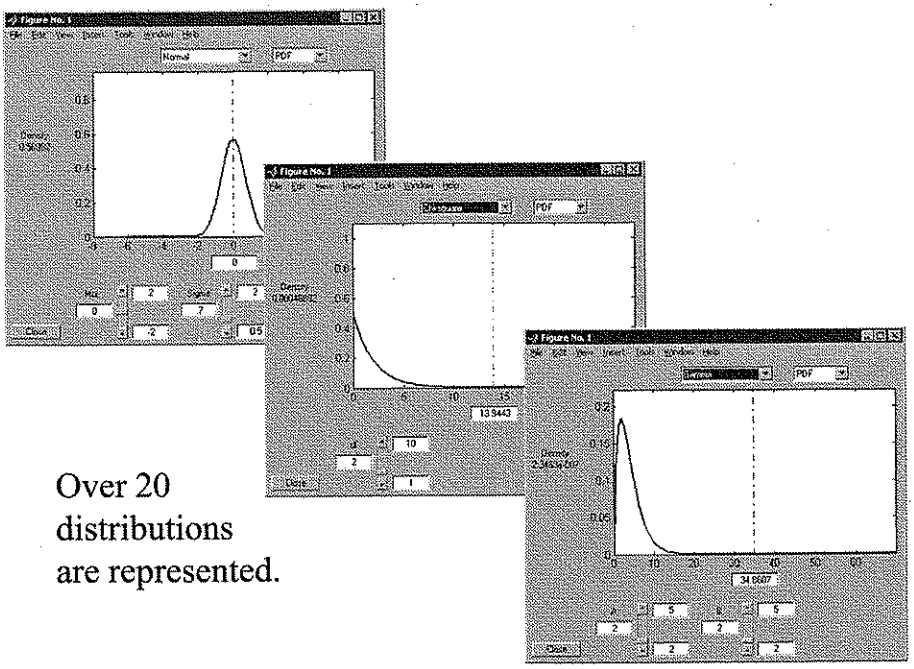
```
>> group = ...
unidrnd(4,100,1);

>> true_mean = 1:5;

>> true_mean = ...
true_mean(...
ones(100,1),:);

>> x = ...
normrnd(true_mean,1);

>> means = ...
grpstats(x,group)


>> x1 = ...
normrnd(5,1,100,1);

>> x2 = ...
normrnd(6,1,100,1);

>> x = [x1 x2];

>> boxplot(x,1)


>> hist(x1)

>> hold on

>> [mu,sigma] = ...
normfit(x1);

>> x = 2:.01:8;

>> y = ...
normpdf(x,mu,sigma);

>> plot(x,50*y,'r')
```

What Can You Do with MATLAB and the Statistics Toolbox?



Probability distributions

- Use the `disttool` GUI to explore distributions.



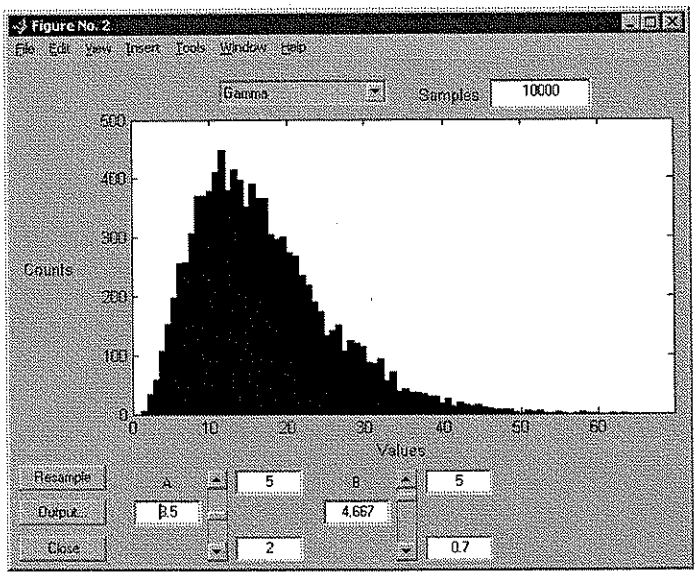
Over 20 distributions are represented.

Try

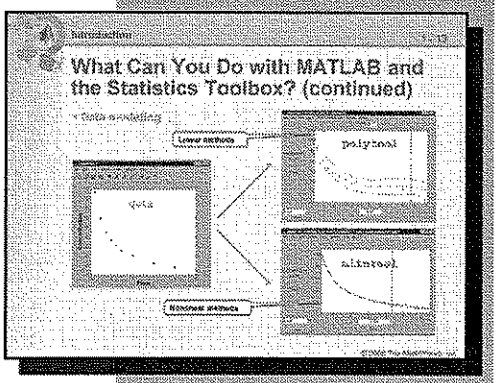
```
>> disttool
```

```
>> randtool
```

- Use the `randtool` GUI to generate data from distributions.

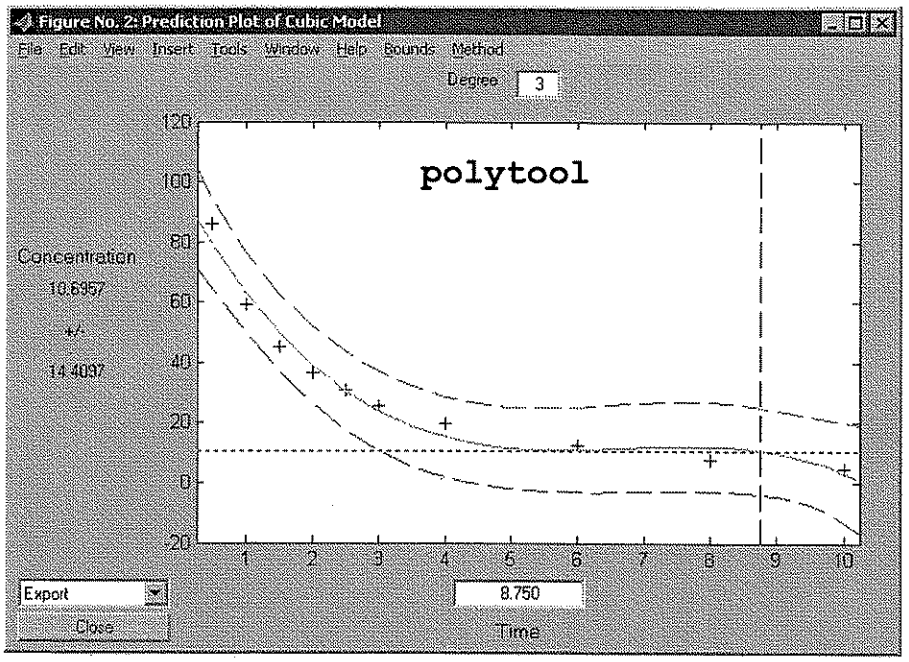


What Can You Do with MATLAB and the Statistics Toolbox?

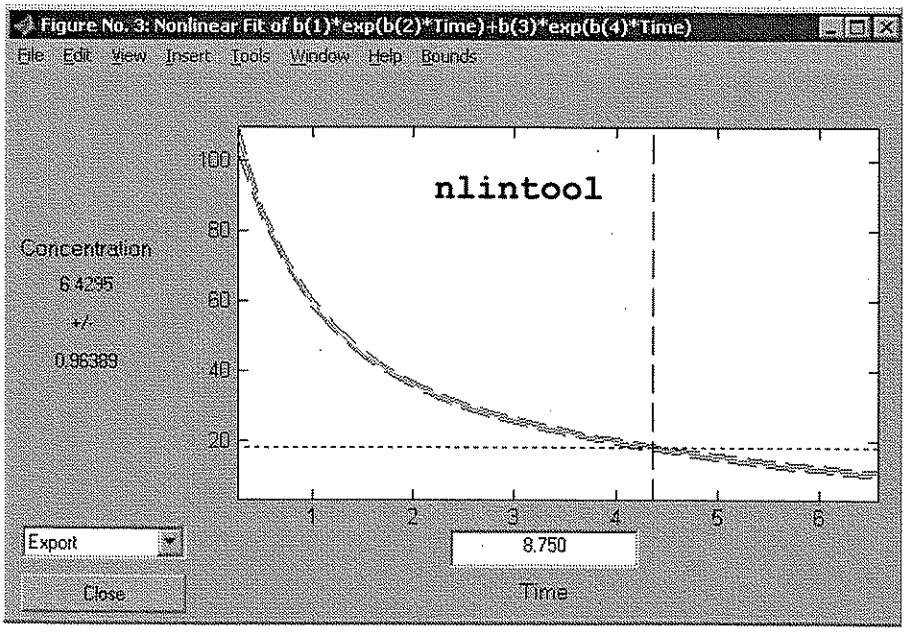


Data modeling

- Linear methods



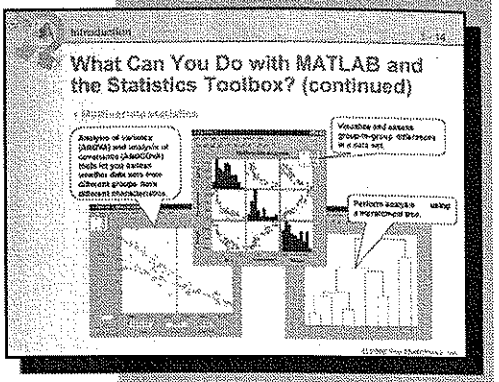
- Nonlinear methods



Try

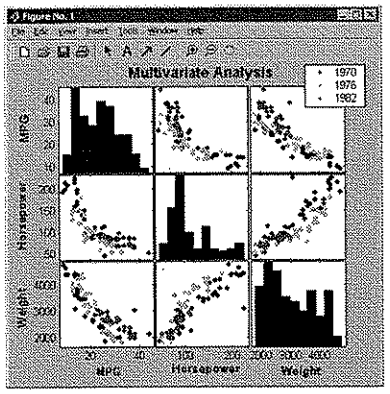
```
>> load polydata

>> polytool(x,y)

>> robustdemo

>> playshow glmdemo


>> clear

>> load reaction

>> rstool(...
reactants,rate,...
'quadratic',...
0.01,xn,yn)

>> rsmdemo

>> nlintool(...
reactants,rate,...
'hougen',beta,...
0.01,xn,yn)
```

What Can You Do with MATLAB and the Statistics Toolbox?



Multivariate statistics

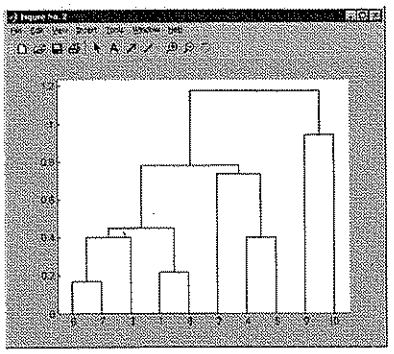
- Visualize and assess group-to-group differences in a data set.



Try

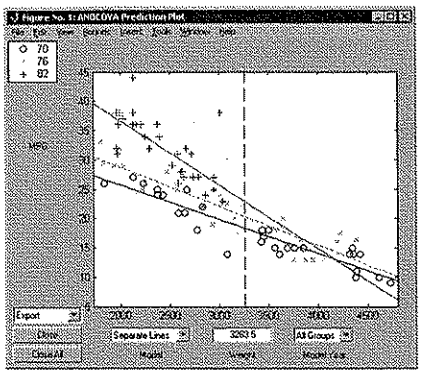
```
>> load carsmall

>> x = ...
[MPG Horsepower ...
Displacement Weight];

>> gplotmatrix ...
(x,[],Model_Year,...
[],'+xo')
```

- Perform analysis using a hierarchical cluster tree.



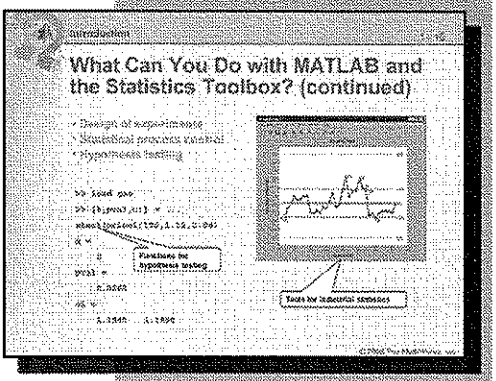
```
>> X = ...
[rand(10,2)+1; ...
rand(10,2)+2];

>> Y = pdist(X);

>> Z = linkage(Y);

>> dendrogram(Z);
```

- Analysis of variance (ANOVA) and analysis of covariance (ANOCOVA) tools let you assess whether data sets from different groups have different characteristics.




```
>> aoctool
```

What Can You Do with MATLAB and the Statistics Toolbox?

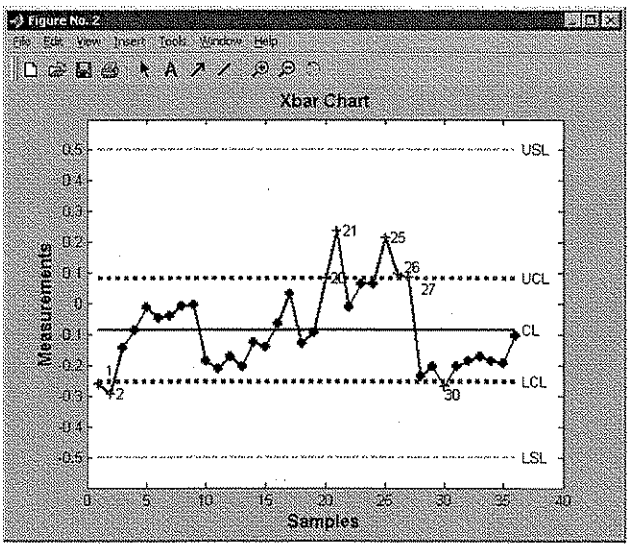


Design of experiments

Statistical process control

Hypothesis testing

- Tools for industrial statistics



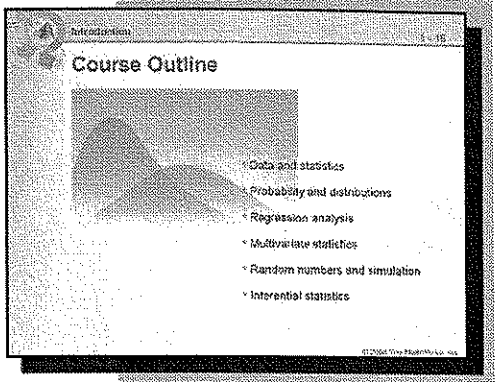
- Functions for hypothesis testing

```
>> load gas
>> [h,pval,ci] = ...
ztest(price1/100,1.15,0.04)
h =
     0
pval =
     0.8668
ci =
     1.1340    1.1690
```

Try

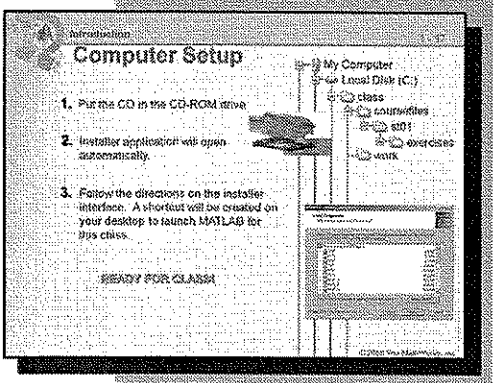
```
>> load parts

>> conf = 0.99;

>> spec = ...
[-0.5 0.5];

>> xbarplot( ...
runout,conf,spec)



>> load gas

>> [h,pval,ci] = ...
ztest( ...
price1/100,1.15,.04)

>> [h,pval,ci] = ...
ttest( ...
price2/100,1.15)

>> [h,sig,ci] = ...
ttest2( ...
price1,price2)
```

Course Outline



- Data and statistics
- Probability and distributions
- Regression analysis
- Multivariate statistics
- Random numbers and simulation
- Inferential statistics

Computer Setup

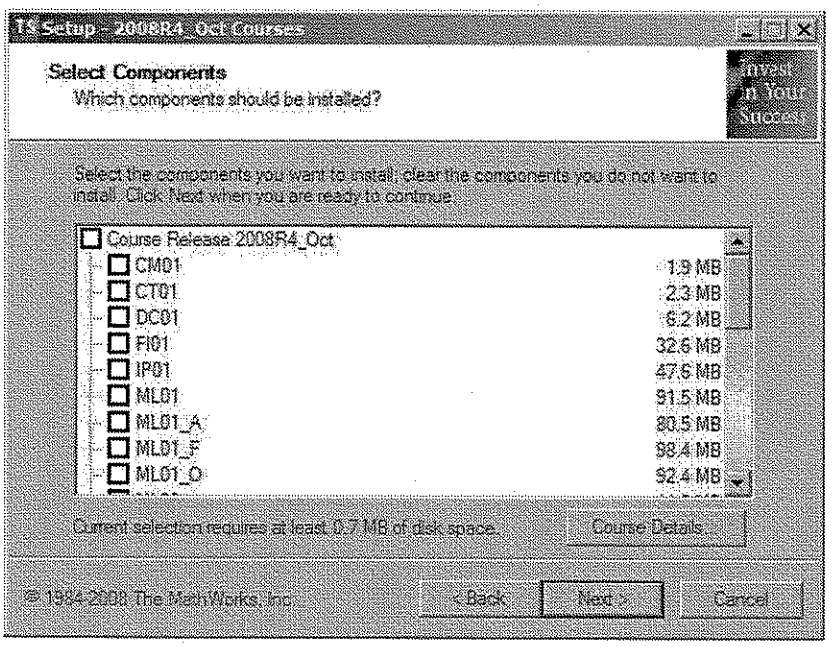


To get ready for class, you need to install the examples and exercises on your course CD. Follow these steps:

1. Put the CD in the CD-ROM drive.

2. Installer application will open automatically. If installer application does not open automatically, open CD-ROM drive in Windows Explorer. Run file `CoursesInstaller_20XXRX_MMM.exe`.

3. Follow the prompts in the installer through the installation process. A shortcut will be created on your desktop to launch MATLAB for this class.

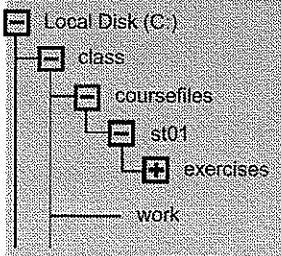


In the installer, you have these options:

- Choose a class root directory for your course files.
- Choose the courses for which you need to install the files. (Examples and exercises for all of our courses are on the CD.)
- Create a shortcut on the desktop to launch MATLAB for this class. (This shortcut runs a `startup.m` file when launching MATLAB, that is customized for the installed course files)

Try

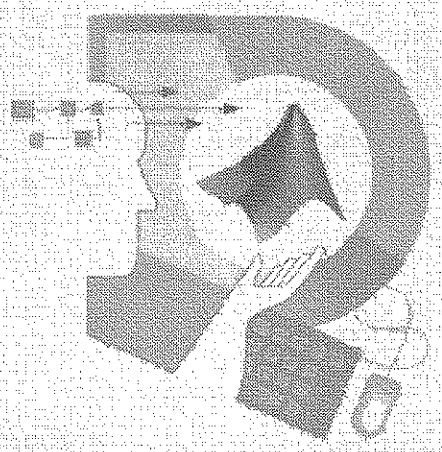
Typical setup



The installation creates a subdirectory of your chosen root directory (the default is `C:\class`) called `coursefiles`. This directory has subdirectories for each of the courses you install, labeled by course number. These individual course directories contain the examples and exercises for the courses. Each course directory has a subdirectory called `exercises` that contains all exercises and their solutions. Finally, there is a subdirectory called `work`, which is empty. During class, put all your work in this subdirectory, so that it is on the path and easy to find. You might want to set your current directory to the `work` directory for convenience.

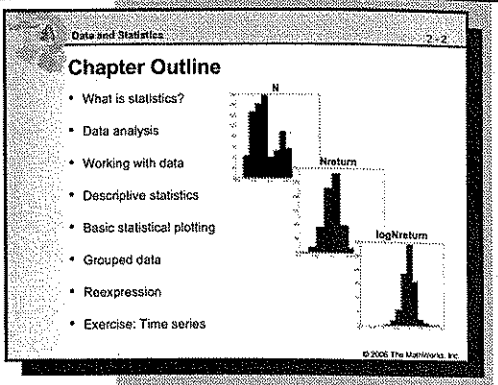
Statistical Methods in MATLAB®

Data and Statistics



The MathWorks
Training Services

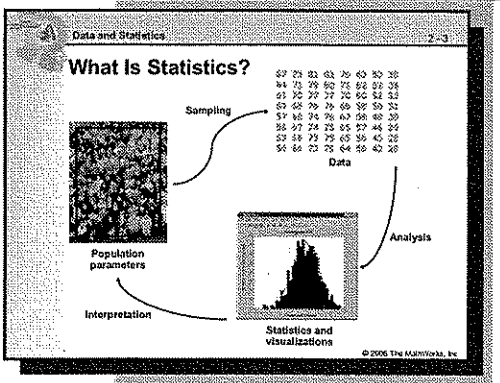
Chapter Outline



- What is statistics?
- Data analysis
- Working with data
- Descriptive statistics
- Basic statistical plotting
- Grouped data
- Reexpression

- Exercise: Time series

What Is Statistics?



Statistical methods play an important role in the larger process of *mathematical modeling*. Mathematical modeling allows us to make considered measurements of a system, and then to draw reasonable conclusions from our observations.

The process of mathematical modeling typically involves four sequential components:

1. The cause and effect relationships of the system that we want to understand (the object of study).

2. The formalization (modeling) of that system in appropriate mathematical language, based on measurement and observation.

3. Computation and visualization within the formalism of the model, in an effort to discover patterns and relationships not apparent in the original observations.

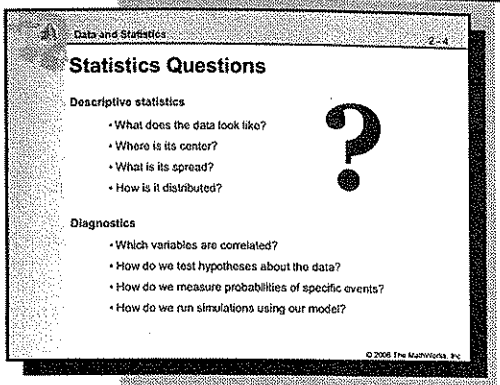
4. Interpretation of the computations and visualizations, in the form of predictions about the behavior of the original system.

Scientific method involves a repeated cycling through the modeling process, comparing model outputs with observations, systematically refining the model in turn.

To create a statistical model, data is gathered from a *population* through a process of *sampling*. Animal populations, stock portfolios, and the items on an assembly line all might represent experimental populations in this context. Much of statistics is concerned with sampling methodology and the design of effective experiments.

After sampling, numerical *statistics* are computed from the data. The statistics are displayed in appropriate visualizations in an effort to discover their patterns and trends. The goal is to make reliable estimates of population *parameters* corresponding to the statistics.

Statistics Questions



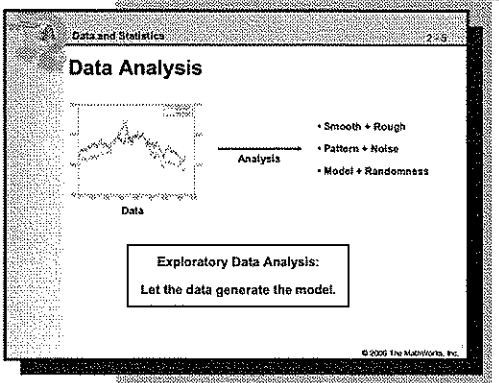
Descriptive statistics

- What does the data look like?
- Where is its center?
- What is its spread?
- How is it distributed?

Diagnostics

- Which variables are correlated?
- How do we test hypotheses about the data?
- How do we measure probabilities of specific events?
- How do we run simulations using our model?

Data Analysis



Analysis, literally, means resolution into component parts. Data analysis attempts to resolve experimental data into components that are, apparently, either significant or insignificant characteristics of the system under study. The breakdown is expressed in a variety of ways:

- Data = smooth + rough
- Data = pattern + noise
- Data = model + randomness

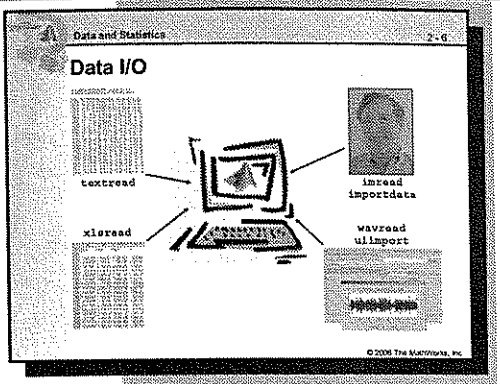
The first two decompositions describe the belief that patterns exist in the system being observed and that those patterns can be discerned in the experimental data. The "rough" and the "noise" represent both the limits of our measurement techniques and the limits of our understanding of the system. Measurement limitations are usually fixed by equipment and methodology, but a reduction of the "noise" of model inaccuracy can often be achieved by systematic application of the mathematical modeling process.

The third decomposition is more methodological: When building statistical models, we can include randomness as a formal representation of our known limitations. We can then make realistic comparisons between our model output and the measured behavior of the system under study.

A poor approach to data analysis is to begin in the "confirmatory mode," where a predetermined model is merely tested to see if it fits the sampled data. Small differences between the "smooth" of the model and the "rough" of the data do not necessarily mean that any real understanding of data patterns has been achieved.

Instead, an "exploratory mode" to data analysis is advocated. Beginning with a simple model, the "rough" of the residual distance between model and data is repeatedly mined for more "smooth." Data visualization is an extremely important part of this process. The goal is to let the data itself generate the model's characteristics.

Data I/O



MATLAB provides a variety of routines for reading experimental data into the MATLAB environment. In MATLAB, data takes the form of workspace variables, that is, arrays of various sorts. Workspace variables can then be manipulated and analyzed using functions and graphical user interfaces (GUIs) from MATLAB and the Statistics Toolbox.

MATLAB also provides a variety of routines for writing the results of your statistical analyses to file formats that may be used outside of the MATLAB environment.

File I/O routines in MATLAB are divided into two categories:

- High-level routines

  These routines work with data in file formats that MATLAB recognizes: text, delimiter-separated ASCII, spreadsheets, various sound and image formats, scientific data formats (CDF and HDF), HTML and XML, ZIP, etc.

  There is a family of high-level `*read` commands in MATLAB that reads in data files in recognizable formats. Each command takes a filename input argument and then a variety of formatting input arguments. These commands go by the names `textread`, `dlmread`, `xlsread`, `imread`, `auread`, `urlread`, etc.

  At the highest level is the MATLAB `importdata` command, which reads in files in recognizable formats by making formatting choices automatically. A GUI version of `importdata`, which suggests formatting options to the user, is the Import Wizard, invoked by the `uiimport` command.

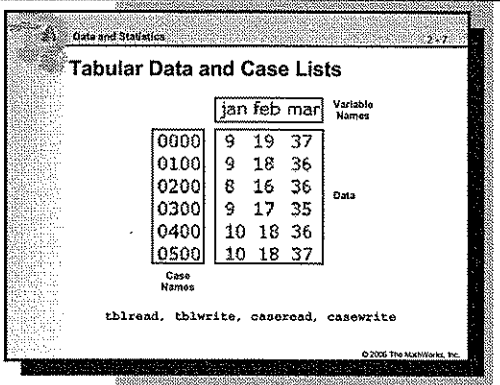
- Low-level routines

  MATLAB also includes a set of low-level file I/O functions based on the I/O functions of the ANSI Standard C Library. Files are opened with `fopen`, read from and written to with `fread`, `fwrite`, `fscanf`, `fprintf`, etc., and then closed with `fclose`.

Try

```
>> help fileformats
>> help iofun

>> jan = ...
textread( ...
'all_temps.txt', ...
'%*u%u%*[^\n]', ...
'headerlines',4);

>> [data,head] = ...
xlsread( ...
'stockdata.xls');
>> plot(data(:,3))
>> legend(head{3})

>> I = ...
importdata( ...
'tlane.jpg');
>> image(I)

>> which hey3.wav
>> uiimport
    (Browse:hey3.wav)
>> sound( ...
hey3.data, hey3.fs)
```

Tabular Data and Case Lists



The Statistics Toolbox provides additional data I/O routines for working with typical statistical file formats. These include the following functions for reading and writing tables:

- `tblread`

```
>> [data,varnames,casenames] = tblread
```

displays the File Open dialog box for selection of a tabular data input 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 specification of the name of a file in the current directory, or the complete path name of any file.

- `tblwrite`

```
>> tblwrite(data,'varnames','casenames')
```

displays the File Open dialog box for specification of the tabular data output file. The file format is the same as for `tblread`. The input `data` is a numeric matrix with a value for each variable-case pair. The input `'varnames'` is a string matrix containing the variable names. The input `'casenames'` is a string matrix containing the names of each case in the first column.

```
>> tblwrite(data,'varnames','casenames',...
   'filename')
```

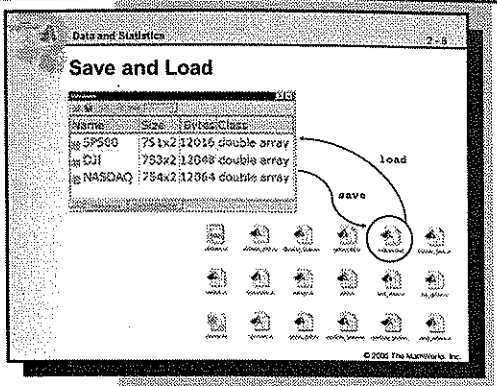
allows specification of a file as in `tblread`.

The Statistics Toolbox functions `caseread` and `casewrite` are used for moving lists of cases in to and out of the MATLAB workspace. The output of `caseread` is a padded character array (string matrix) with one case in each row.

Try

```
>> type sat.dat
>> [data, ...
varnames, ...
casenames] = ...
tblread('sat.dat')

>> tblwrite( ...
data, ...   varnames,
... casenames, ...
'sattest.dat')
>> type sattest.dat

>> strmat = ...
strvcat( ...
'January', ...
'February', ...
'March')
>> casewrite( ...
strmat,'months.dat')

>> type months.dat
>> names = ...
caseread( ...
'months.dat')
```

Save and Load



Two data I/O routines are especially useful when working with MATLAB variables:

- The `save` command writes workspace variables to a binary MATLAB data file (MAT-file) with a `.mat` extension.

- The `load` command reads variables from a MAT-file back into the MATLAB workspace.

Though quite specialized, `save` and `load` are used for day-to-day management of your MATLAB computations.

Three years of data on major stock indices have been saved as a MAT file and can be loaded into the workspace by typing

```
>> load indices
```

The workspace will now contain three new variables:

```
DJI
NASDAQ
SP500
```

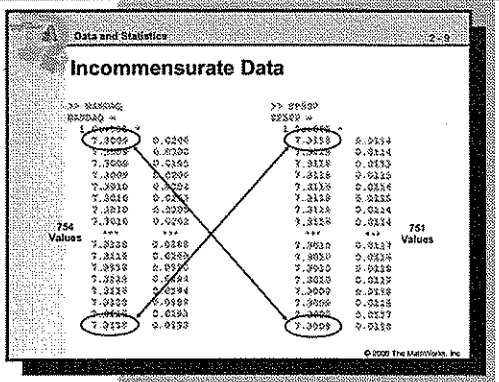
Note: Current and historical data on securities can be downloaded from online sources using the Datafeed Toolbox. If the toolbox is installed, it may be opened by typing

```
>> dftool
```

Try

```
>> doc save
>> doc load

>> load indices


>> dftool
```

Incommensurate Data



The three data sets

```
DJI
NASDAQ
SP500
```

are not immediately comparable because of their *incommensurate sets* of sample dates. Note, for example, that the `NASDAQ` and `SP500` data begin and end on the same date, but contain a different number of values. Note, also, that the dates in the `NASDAQ` data are reversed with respect to the dates in the `SP500` data.

To make the two sets of time series data commensurate, we use MATLAB *set operations* such as

```
setdiff
intersect
ismember
setxor
union
unique
```

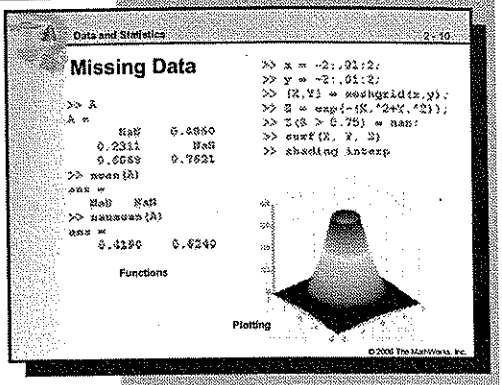
Note: The commensurate indices `S` and `N` created at right, together with the array `dates`, are stored in the file `SNindices.mat`, for future reference. They may be retrieved by typing

```
>> load SNindices
```

Try

```
>> datestr( ...
SP500(1,1))

>> datestr( ...
SP500(end,1))

>> datestr( ...
NASDAQ(1,1))

>> datestr( ...
NASDAQ(end,1))

>> SP500 = ...
flipud(SP500);

>> [xorSN,iS,iN]...
= setxor( ...
SP500(:,1), ...
NASDAQ(:,1),...
'rows')

>> datestr(xorSN)

>> NASDAQ(iN,:) ... =
[];

>> SP500(iS,:) ...  =
[];

>> N = NASDAQ(:,2);

>> S = SP500(:,2);

>> dates = ...
datestr( ...
NASDAQ(:,1));

>> t = ...
1:length(dates);

>> plotyy(t,S,t,N)
```

Missing Data



Incommensurate data sets are often made commensurate by filling in missing data values with

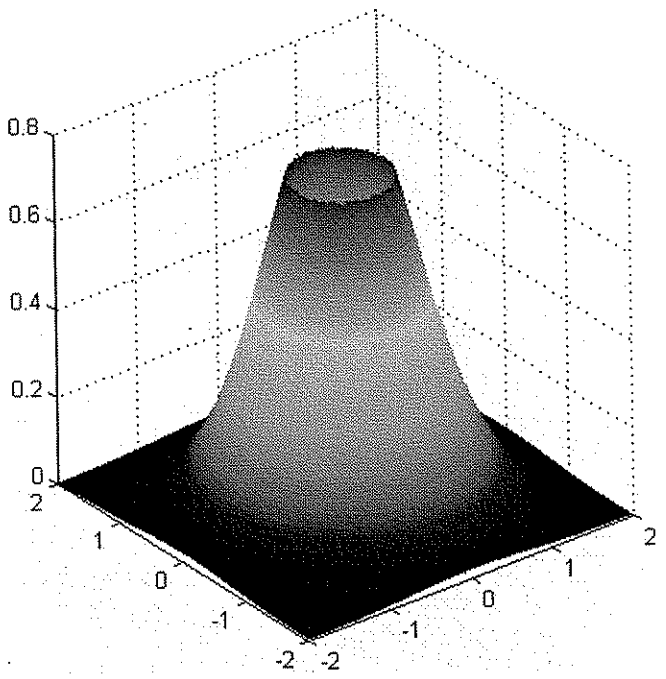
`NaN`

the MATLAB representation for undefined numerical results. Removing `NaN`s can destroy the array structure of otherwise commensurate data sets.

The Statistics Toolbox has a number of functions that ignore `NaN` values. These include

```
nanmax
nanmean
nanmedian
nanmin
nanstd
nansum
```

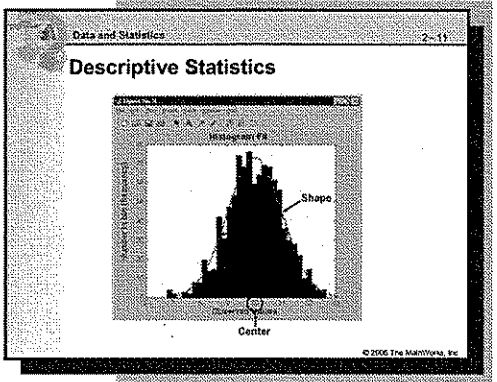
Note that plotting commands ignore `NaN` values automatically.



Try

```
>> A = rand(3,2)
>> A([1,5]) = ...
[nan,nan]
>> mean(A)
>> nanmean(A)
>> nanmax(A)

>> s = 1:3;
>> plot( ...
s,A(:,1),'o')

>> x = -2:.01:2;
>> y = -2:.01:2;
>> [X,Y] = ...
meshgrid(x,y);
>> Z = exp( ...
-(X.^2+Y.^2));
>> Z(Z>0.75) = nan;
>> surf(X,Y,Z)
>> shading interp
```

Descriptive Statistics



A *statistic* is a numerical value calculated from a population sample. If the sample is randomly selected, the statistic is a *random variable*. When sampled values of a random variable are sorted from low to high, they form what is known as the *sample distribution.*

The distribution of a variable has three important characteristics:

- Center
- Spread
- Shape

The *center* of a distribution is its overall location on the continuum from low to high values. It is the value around which all of the other values of the distribution are scattered.

The *spread* of a distribution refers to its variability or dispersion around the center. It is related to the volatility of the underlying population parameter and the limitations of the sampling process.

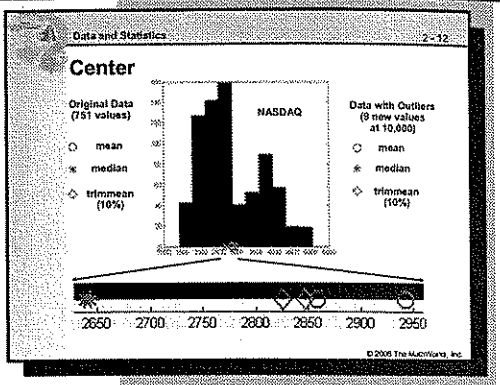
We will look at measures of center and spread in the next few pages.

The *shape* of a distribution is more subtle, referring to its type (often categorized by the type of underlying process that leads to the distribution), whether it is symmetrical or skewed, whether it is single-peaked or multi-peaked, whether it has outliers or gaps, etc. We will look at distribution shapes in the next chapter.

When working with *summary statistics* of the characteristics of a distribution, we want to use measures with a high degree of *resistance*. Resistant statistics are less affected by changes, no matter how large, in a small proportion of the sampled values. They are also resistant to departures from the standard normal distribution, making them better measures for a wider variety of distributions.

We combine summary statistics with appropriate *visual summaries*, using the many plot types in MATLAB and the Statistics Toolbox.

Center



MATLAB and the Statistics Toolbox contain several functions for measuring distribution centers. These include

`mean`
`median`
`trimmean`
`geomean`
`harmmean`

Different measures display different degrees of resistance to atypical deviations from the center of the data (*outliers*).

The `mean` (a.k.a. the *arithmetic mean*, or the *average*) is useful for symmetric distributions, but it is notoriously nonresistant to the presence of outliers. The mean answers the question "If all of the data had the same value, what would that value have to be in order to achieve the same sum?"

The `median` is an *order statistic*, giving the midpoint of the sorted data. It is much more resistant to changes in a few data values. It is an especially useful center for nonsymmetric (*skewed*) distributions.

`trimmean` takes a second argument (a percentage) indicating the amount of outlying data to be trimmed (half of the percentage at each end). It is resistant to outliers to the degree specified.

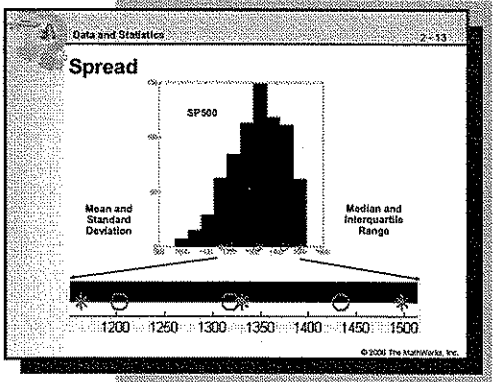
The *geometric mean* `geomean` is appropriate for exponentially distributed data, or for data that is normally distributed after a logarithmic reexpression. It is more resistant than the mean, though not as much as the median. The geometric mean answers the question "If all of the data had the same value, what would that value have to be in order to achieve the same product?"

The *harmonic mean* `harmmean` is sometimes appropriate for data, such as rates, that arise as ratios. This measure is very sensitive to changes in the data, and may be used to add significance to outliers.

Try

```
>> load SNindices
>> hist(N)
>> hold on
>> plot( ...
mean(N),0,'ro')
>> plot( ...
median(N),0,'r*')
>> plot( ...
trimmean(N,10),0,'rd')
>> N(752:760) = ...
10000;
>> plot( ...
mean(N),0,'mo')
>> plot( ...
median(N),0,'m*')
>> plot( ...
trimmean(N,10),0,'md')
>> N(752:760)=[];
>> hold off
```

Spread



MATLAB and the Statistics Toolbox also contain several functions for measuring distribution spread. These include

```
range
std
var
mad
iqr
prctile
```

The `range` is simply the difference between the maximum and minimum data values.

Like the `mean`, the *standard deviation* `std` is typically used to measure the spread of symmetrical, normally distributed data. Because it is given as the square root of the *variance* `var`, the sum of the squares of the residual distances of data values from the mean, the standard deviation tends to amplify the effect of outliers.

The *mean absolute deviation* `mad` is also based on the center at `mean`, but it makes itself more resistant to outliers than the standard deviation by avoiding squaring when calculating residual distances.

The *interquartile range* `iqr` (a.k.a. the *midspread*) is based on the more robust center at the median (the 50th percentile point). It gives the distance between the 25th and 75th percentile in the data. The interquartile range is used to construct the *five number summary* of a set of data, consisting of the median, the two ends of the interquartile range, and the maximum and minimum data values within a distance of 1.5*`iqr` from the 75th and 25th percentiles, respectively. In a normal distribution, 95% of the data is within these maximum and minimum values. The five number summary is used to create boxplots.

The function `prctile` is used in conjunction with `iqr`:

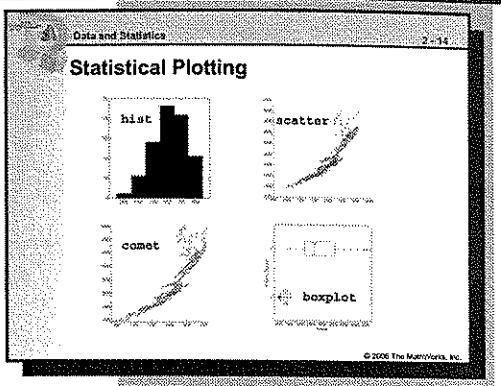
```
>> Y = prctile(X,p)
```

returns a value that is greater than p percent of the values in X.

Try

```
>> load SNindices
>> hist(S)
>> hold on
>> std(S)
>> iqr(S)
>> plot( ...
mean(S),0,'ro',...
[mean(S)-std(S),...
mean(S)+std(S)],...
0,'ro')
>> hold on
>> plot( ...
median(S),0,'m*',...
[prctile(S,50)-...
iqr(S),...
prctile(S,50)+...
iqr(S)],...
0,'m*')
>> hold off
```

Statistical Plotting



Data analyses that rely entirely on numerical summaries based on statistics often miss important characteristics of a distribution. In exploratory data analysis, visual displays and summaries of the data are vital. Only after a combined numerical and graphical analysis of the data is it appropriate to construct models and proceed to hypothesis testing.

We will look at a number of statistical plotting routines in MATLAB and the Statistics Toolbox. Here we introduce three basic plot types:

```
hist
scatter
boxplot
```

Histograms produced by the `hist` command are the basic visualization tool for univariate data. Histograms order the data and display the distance between the values, providing a quick view of the distribution of a particular variable. The `hist` command takes a variety of optional input arguments for controlling the display.

Scatter plots produced by the `scatter` command are the basic visualization tool for multivariate data. They are used to identify relationships and correlations among different variables in a data set.

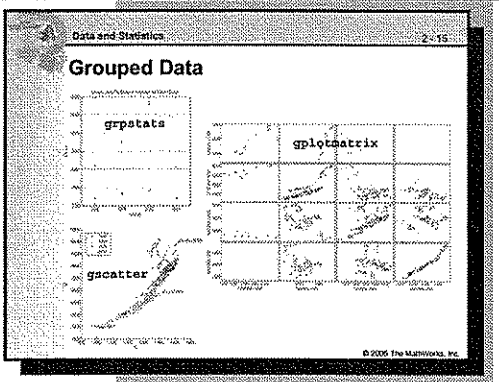
Box plots produced by the `boxplot` command, as discussed on the previous page, display the five number summary of a set of data. Box plots have the advantage of providing a visual representation of the spreads in the five number summary. They are especially useful for visualizing distributions with nonnormal tails.

Box plots can include notches to help determine whether different groups have significantly different medians. The notches are drawn so that groups with overlapping notches are not significantly different, and groups with nonoverlapping notches are significantly different. The width of the notched interval is proportional to the interquartile range divided by the square root of the number of observations. Side-by-side comparison of two notched box plots is the graphical equivalent of a *t-test* (see Chapter 7).

Try

```
>> load SNindices
>> hist(S)
>> hist(S,50)
>> hist(S,...
[1000:100:1500])

>> scatter(S,N)
>> scatter( ...
S,N,4*pi,'m')
>> comet(S,N)
>> S2 = ...
repmat(S',100,1);
>> N2 = ...
repmat(N',100,1);
>> comet(S2(:),N2(:))

>> boxplot([S,N])
>> boxplot( ...
[S,N],0,'*',0,1)
>> boxplot( ...
[S,N],1,'*',0,1)
>> boxplot( ...
[S,N],0,'*',1,1)
>> boxplot(...
[S,N],0,'*',0,0)
```

Grouped Data



The functions for computing descriptive statistics, when passed a *matrix* of input data, typically treat the columns independently, computing separate statistics for each column. Sometimes it is convenient to add a column to data that provides grouping information, so that data with the same value in the grouping column are considered to be part of the same *group*. Although MATLAB makes it simple to apply functions to any subset of an array, the Statistics Toolbox provides a number of functions for working with groups directly.

- `grpstats`

```
>> [means,sem,counts,name] = ...
   grpstats(X,group)
```

returns the means, the *standard error* of the means (the `std` of the sample distribution), the number of elements in each group, and the names of the groups, of each column of `X` by group, where `X` is a matrix of observations.

`group` is an array that defines the grouping. Two elements of `X` are in the same group if their corresponding group values are the same. `group` can be a vector, string array, or cell array of strings.

- `gscatter, gplotmatrix`

```
>> 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 with the same marker and color.

- `gname`

```
>> gname('cases')
```

displays cross-hairs on a figure and waits for mouse or keyboard input. Click once near each point to label that point. Input `cases` is a string matrix with each row the case name of a data point.

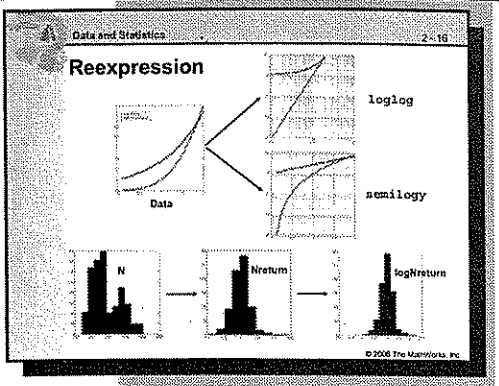
Try

```
>> load SNindices

>> ygroup = ...
dates(:,8:11);

>>[mean,sem,...
count,name] = ...
grpstats( ...
[S,N],ygroup,.05)

>> gscatter( ...
S,N,ygroup)

>> gname(dates)
```

...then click on points in the scatter plot (Esc to end).

```
>> I00 = ...
ismember( ...
cellstr(ygroup), ...
'2000');

>> mgroup00 = ...
dates(I00,4:11);

>> gscatter( ...
S(I00),N(I00), ...
mgroup00)

>> doc gplotmatrix

>> edit ysyn

>> ySyN

>> gname(mgroup)
```

Reexpression



Normal distributions (symmetrical and single-peaked) and linear relationships (plotted as straight lines) are standards in statistics, though they rarely arise in practice. One way to handle nonnormality and nonlinearity is through *reexpression*. Reexpression is simply the use of a scale of measurement other than the one on which a variable was originally recorded. This is accomplished by applying a *transformation* to the data.

A familiar example is data displaying a concave upward nonlinear trend. Both power functions $y = x^a$ and exponential functions $y = a^x$ are possible models. Notice, however, what happens when we take the logarithm of these equations:

Power	Exponential
$\ln y = a \ln x$	$\ln y = x \ln a$
$Y = aX$	$Y = xA$

In the case of the power function, the reexpressed variables $Y = \ln y$ and $X = \ln x$ now show a linear relationship. In the case of the exponential function, it is the transformed variable Y and the original variable x that show a linear relationship.

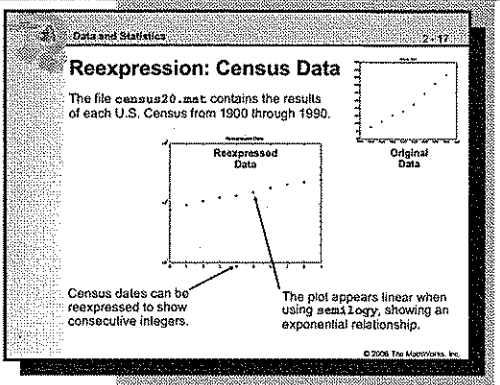
Generally speaking, *concave transformations* (like roots and logarithms) spread out values at the left of a variable's distribution and pull in outliers at the right. Similarly, *convex transformations* of a variable (like powers and exponentials) are useful when a distribution is bunched to the right. These transformations both have the advantage of putting skewed distributions on a scale on which they are symmetrically distributed. We must remember, however, that symmetrical distributions do not necessarily have the shape of a normal distribution. Symmetrizing variables is, nevertheless, usually a good first step in a successful data analysis.

Another form of reexpression, called *icing the tails*, is used for rescaling distributions with outliers, or stretched tails, to both the left and the right. In this case, a transformation that is convex to the left and concave to the right is required. Sine functions, odd-numbered roots, and logistic functions all serve as possible candidates.

Try

```
>> x = 0.1:0.1:2;
>> yp = x.^3;
>> ye = exp(x);
>> plot( ...
x,yp,'r',x,ye,'b')
>> loglog( ...
x,yp,'r',x,ye,'b')
>> grid on
>> semilogy( ...
x,yp,'r',x,ye,'b')
>> grid on

>> load SNindices
>> i = ...
2:length(dates);
>> Nreturn = ...
(N(i)-N(i-1)) ...
./N(i-1);
>> logNreturn = ...
log(N(i)./N(i-1));
>> hist(N)
>> hist(Nreturn)
>> hist(logNreturn)
>> normplot( ...
logNreturn)
```

Reexpression: Census Data



The file `census20.mat` contains the results of each U.S. Census from 1900 through 1990. When the file is loaded, two variables appear in the workspace: `cdate` and `pop`.

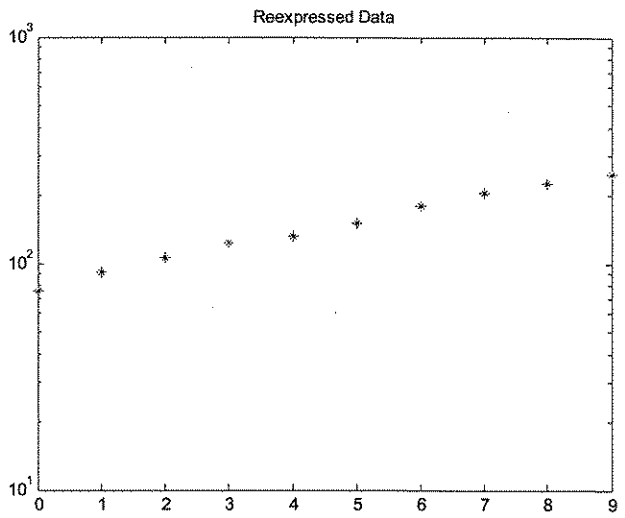
Since the values in `cdate` are very high, this data should be reexpressed. One method is to use the formula

```
>> rcdate = (cdate - 1900) / 10;
```

The variable `rcdate` now contains integers from 0 to 9.

Plotting the data, we see that it follows a concave upward nonlinear trend. This is typical of exponential growth, power functions, and many other relationships. To test for exponential growth, we create a semilog plot of the data:

```
>> semilogy(rcdate, pop, '*');
```



The plot seems to be roughly a straight line with positive slope, indicating that the data may be reasonably fit by an exponential function. If the exponential has the form $y = Ae^{ax}$, then we will have $\ln y = \ln A + ax$. We see that the slope of our line is a and its y-intercept is $\ln A$.

Other attempts to reexpress the data using `loglog` or `semilogx` do not result in straight lines, so the corresponding fits will not represent the data as well.

Try

```
>> censusgui
```

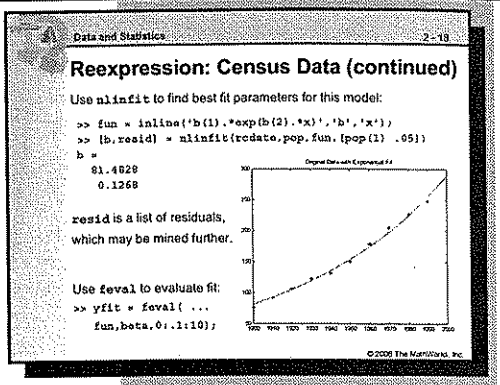
Then try:

```
>> load census20
>> rcdate = ...
   (cdate-1900)/10;
>> semilogy ...
   (rcdate,pop,'*')
```

Compare to:

```
>> loglog ...
   (rcdate,pop,'*')
>> semilogx ...
   (rcdate,pop,'*')
```

Reexpression: Census Data (continued)



To fit an exponential to the U.S. Census data, we may use the Statistics Toolbox function `nlinfit` (discussed in more detail in Chapter 4). First, create an inline function:

```
>> fun = inline('b(1).*exp(b(2).*x)','b','x');
```

Then, call `nlinfit` with the data, inline function, and initial guesses for the parameters:

```
>> [b,resid] = ...
nlinfit(rcdate,pop,fun,[pop(1) .05])

b =
   81.4828
    0.1268
```

The outputs correspond to the A and the a in the functional form $y = Ae^{ax}$. Notice that $Ae^{a(x+1)} = Ae^{ax}e^{a} = ye^{a} = ye^{0.1268} \approx y(1.1352)$, suggesting a growth rate of about 13% percent every 10 years.

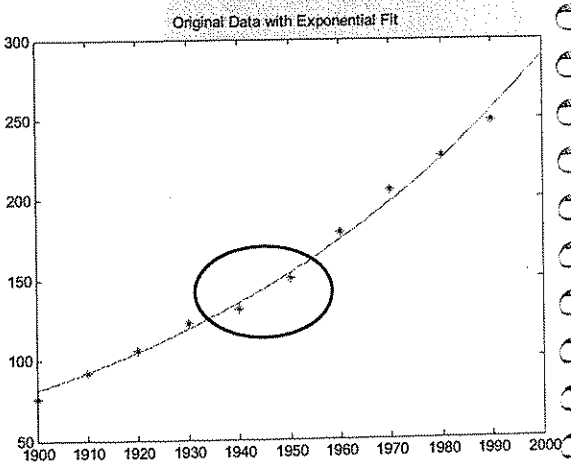
To plot a fitting curve along with the data, call `feval` to evaluate the fit, then reexpress the data in the original terms:

```
>> t = 0:.1:10;
>> yfit = feval(fun,b,t);
>> xfit = (t*10) + 1900;
>> plot(cdate,pop,'*',xfit,yfit,'r-')
```

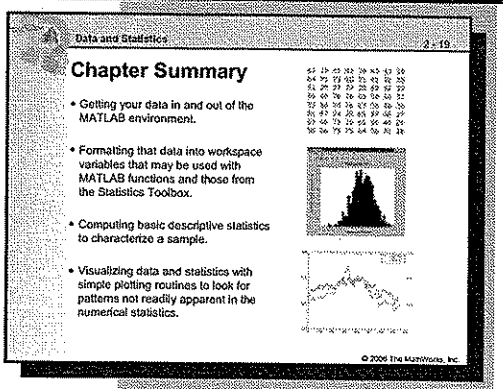
Further analysis could be done on the residuals. For example, the residuals of the 1940 and 1950 censuses are negative, while the residuals of all surrounding censuses are positive. This suggests that the population in both 1940 and 1950 was lower than the trend.

Try

```
>> fun = ...
inline(['b(1).*' ...
'exp(b(2).*x)'], ...
'b','x');

>> [b,resid] = ...
nlinfit(rcdate,...
pop,fun,[pop(1) .05])

>> t = 0:.1:10;
>> yfit = feval...
(fun,b,t);
>> xfit = t*10+1900;
>> plot(cdate,pop,...
'*',xfit,yfit,'r-');
```



Chapter Summary



In this chapter we covered the basics of exploratory data analysis in MATLAB:

- Getting your data in and out of the MATLAB environment.
- Formatting that data into workspace variables that may be used with MATLAB functions and those from the Statistics Toolbox.
- Computing basic descriptive statistics to characterize a sample.
- Visualizing data and statistics with simple plotting routines to look for patterns not readily apparent in the numerical statistics.

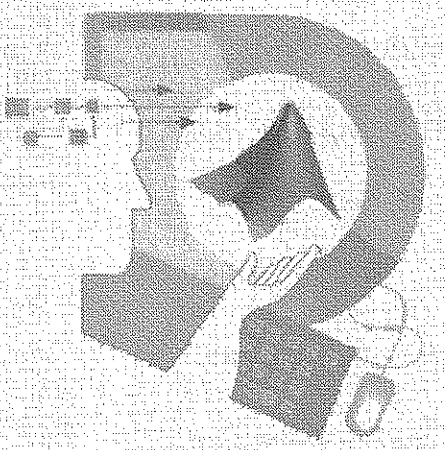
We will make use of these techniques and develop the themes of exploratory data analysis in the chapters that follow.

In the next chapter we introduce probability distributions as models of the sample distributions that we have begun discussing here. Fitting a theoretical distribution to a sample is a key step in the data modeling process. With a model distribution in hand, we will be able to generate random data, run simulations, and move toward the prediction and decision-making techniques of inferential statistics.

Exercise: Time Series

Statistical Methods in MATLAB®

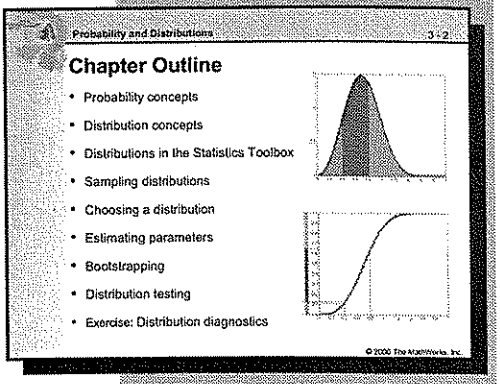
Probability and Distributions



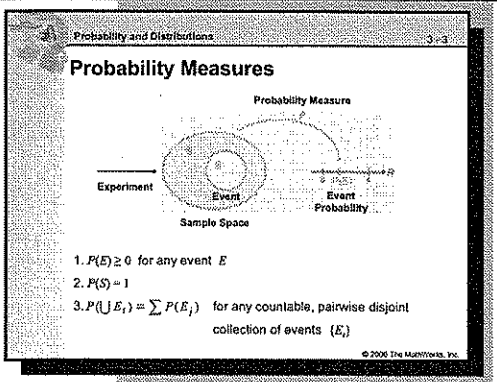
The MathWorks
Training Services

Chapter Outline

- Probability concepts
- Distribution concepts
- Distributions in the Statistics Toolbox
- Sampling distributions
- Choosing a distribution
- Estimating parameters
- Bootstrapping
- Distribution testing
- Exercise: Distribution diagnostics



Probability Measures



Consider an experiment with a *sample space* (set of possible outcomes) S. The probability of an *event* E (a subset of the sample space) expresses the likelihood that the outcome of the experiment will lie in E. It is a *measure* of the set E.

A *probability measure* P for S is a real-valued function defined on the collection of all events E satisfying

1. $P(E) \geq 0$ for any event E

2. $P(S) = 1$

3. $P(\bigcup E_i) = \sum P(E_i)$ for any countable, pairwise disjoint collection of events $\{E_i\}$

The last condition is known as *countable additivity*, and states that the probability of a union of a finite or countably infinite collection of disjoint events is the sum of the corresponding probabilities.

Intuitively, the probability of an event is supposed to express the long-term relative frequency of the event. Suppose we repeat some experiment indefinitely. Let $N_n(E)$ denote the number of times E occurs (the outcome of the experiment lies in E) in n runs. The number N_n is the experimental *frequency* of the event. The number

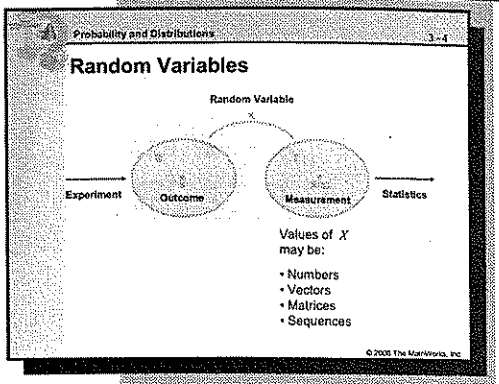
$$P_n(E) = N_n(E) / n$$

is the event's *relative frequency*. We expect the relative frequency of an event to converge to the probability of the event:

$$P_n(E) \rightarrow P(E) \text{ as } n \rightarrow \infty$$

This is the *law of large numbers*. Of all of the possible probability measures satisfying 1. through 3. above, we expect the true probability to satisfy this law. It then follows that if we have data from n runs of an experiment, the observed relative frequency $P_n(E)$ can be used as an approximation of $P(E)$. This approximation is called the *empirical probability* of the event E.

Random Variables



Suppose that we perform a random experiment, one whose outcome cannot be predicted with certainty beforehand. Let S be the sample space of the experiment's possible outcomes. Then any function X from S to another set T is called a (T-valued) *random variable*. Statistics are random variables, computed from experimental outcomes.

We think of random variables as *measurements* of interest in the context of a random experiment. The measurements are random in the sense that their values depend on the unpredictable outcome of the experiment. Each time the experiment is run, an outcome s in S occurs, and the random variable X takes on the value $X(s)$.

Often, a random variable X takes values in a subset T of all real-valued k-element vectors for some k. If $k > 1$, then

$$X = [X_1, X_2, ..., X_k]$$

where X_i is a real-valued random variable for each i. In this case, we refer to X as a *random vector*.

A random variable may also take on matrix values. For example, if an experiment selects n objects from a population and records various measurements for each object, then the outcome of the experiment is a vector of vectors

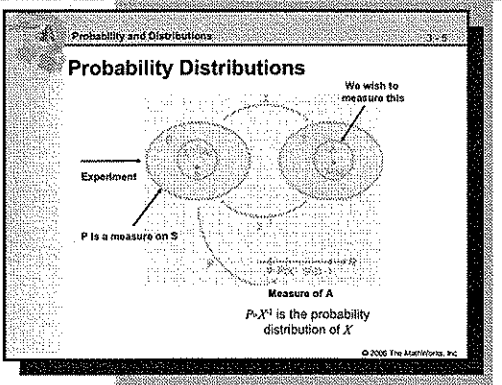
$$X = [C_1, C_2, ..., C_n]$$

where C_i is the vector of measurements for the i^{th} object and the i^{th} column of the matrix X.

There are other possibilities. Random variables sometimes take values that are infinite sequences. When the sequence is time based, the random variable is often called a *stochastic process*.

In all cases, however, a random variable is simply a function from the sample space S to another set T. That is, it is a measurement made on possible experimental outcomes.

Probability Distributions



Suppose that X is a random variable from a sample space S to a set T. How do we measure the probability that X is in A, where A is some subset of T?

Let S_A be the set of all experimental outcomes in the sample space S that produce a value of X in A, so that

$$S_A = X^{-1}(A) = \{s \in S : X(s) \in A\}$$

If P is a probability measure on S, then $P(S_A)$, as a function of A, is a probability measure on T. This measure is called the *probability distribution* of the random variable X (relative to the choice of the measure P on S). The probability $P(S_A)$, the probability that X takes its value in the specified range A, is often written $P(X \in A)$.

We see that any random variable X defines

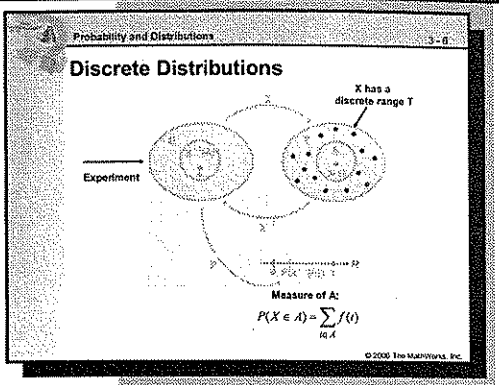
- A range of measurement outcomes T (the possible values of X)
- A collection of subranges or events (the subsets of A of T, corresponding to subsets $S_A = X^{-1}(A) = \{s \in S : X(s) \in A\}$ in S)
- A probability measure on these events (the distribution of X, defined by a measure P on S, given by $P(X^{-1}(A)) = P(S_A)$)

The outcome s of an experiment can itself be thought of as a random variable. If we take X to be the identity function on S, then X is a random variable with

$$P(S_A) = P(A)$$

In this sense, *any* probability measure on S can be thought of as the distribution of a random variable. Particular distributions are often described by defining the measure P on the sample space S.

Discrete Distributions



Consider a random experiment with sample space S. Let X be a random variable on S taking on a countable (finite or countably infinite) range of possible values T. Then X is a *discrete random variable* described by a *discrete distribution*.

A *probability density function* for the random variable X is a real-valued function on the discrete set T for which

$$f(t) = P(\{s \in S : X(s) = t\})$$

where P is a probability measure on S. We write: $f(t) = P(X = t)$.

Discrete density functions always satisfy the following conditions:

1. $f(t) \geq 0$ for any t in T
2. $\sum_{t \in T} f(t) = 1$
3. $\sum_{t \in A} f(t) = P(X \in A)$ for any subset A of T

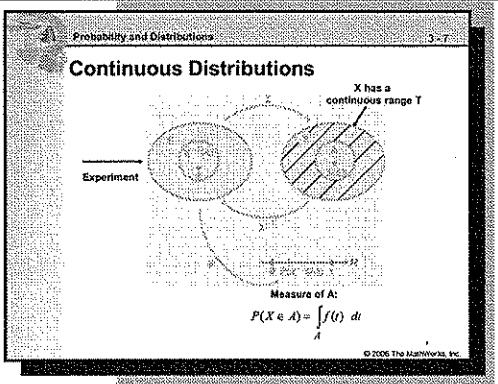
Condition 3 is particularly important since it implies that the probability distribution of a discrete random variable is completely determined by its density function. Conversely, any function that satisfies conditions 1 and 2 is a discrete density, and condition 3 can be used to construct a discrete probability distribution for X.

If X has a discrete distribution with density f, then we define the *expected value* of X to be

$$E(X) = \sum_{x \in T} xf(x)$$

The expected value of X is also called the *mean of the distribution* and is frequently denoted by μ.

Continuous Distributions



Consider a sample space S with probability measure P. Let X be a random variable on S taking on a range of possible values T. Then X is a *continuous random variable* described by a *continuous distribution* if

$$P(X = t) = 0 \quad \text{for each } t \text{ in } T$$

Continuous distributions are in contrast with discrete distributions, for which all of the probability density is concentrated on a discrete set. In a continuous distribution, the probability density is spread continuously over T, with no density at any particular value of X.

A real-valued function f defined on T is said to be a *probability density function* for X if f satisfies the following conditions:

1. $f(t) \geq 0$ for any t in T
2. $\int_T f(t)\ dt = 1$
3. $\int_A f(t)\ dt = P(X \in A)$ for any subset A of T

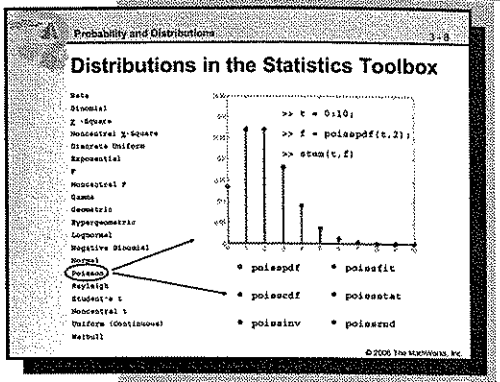
Condition 3 is particularly important since it implies that the probability distribution of a continuous random variable is completely determined by the density function. Conversely, any function that satisfies conditions 1 and 2 is a continuous density, and condition 3 can be used to construct a continuous probability distribution for X.

If X has a continuous distribution with density f then we define the *expected value* of X to be

$$E(X) = \int_T x f(x)\ dx$$

The expected value of X is also called the *mean of the distribution* and is frequently denoted by μ.

Distributions in the Statistics Toolbox



The Statistics Toolbox supports computations involving 20 different common distributions. For each distribution `dist` (where `dist` is `norm` for the normal distribution, `bino` for the binomial distribution, etc.), the toolbox provides *a selection of* the following functions (depending on `dist`):

- `distpdf`
  The distribution's *probability density function*. Computes densities $f(t)$ for specified distribution parameter settings.

- `distcdf`
  The distribution's *cumulative distribution function*. For distributions of real-valued random variables X, computes a sum or integral over the pdf: $F(t) = P(X \le t)$.

- `distinv`
  The distribution's *inverse cumulative distribution function*. Given probabilities P between 0 and 1, computes values of t with cdf equal to P (or, for discrete distributions where an exact inverse may not exist, the first t such that the cdf equals or exceeds P.) Used in random number generation and hypothesis testing. (See Chapters 6 and 7.)

- `distfit`
  For *parameter estimation* when fitting the distribution to a data set. Computes parameter estimates and confidence intervals.

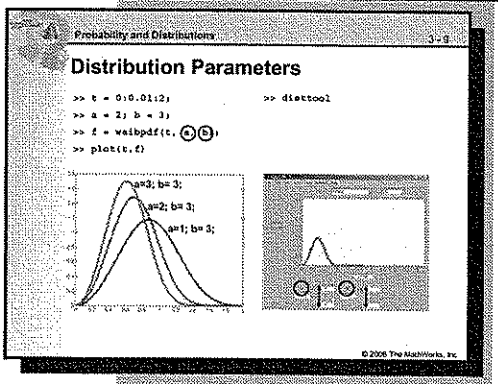
- `diststat`
  Computes the *mean and variance* of the distribution as a function of the parameters.

- `distrnd`
  For *random number generation* from the distribution. Based on the uniform random number generator `rand` in MATLAB, `rnd` functions compute random numbers either directly or by applying appropriate inversion or rejection methods. (See Chapter 6.)

Distributions supported by the Statistics Toolbox

```
Beta
Binomial
χ -Square
Noncentral χ-Square
Discrete Uniform
Exponential
F
Noncentral F
Gamma
Geometric
Hypergeometric
Lognormal
Negative Binomial
Normal
Poisson
Rayleigh
Student's t
Noncentral t
Uniform (Continuous)
Weibull
```

Distribution Parameters



Each distribution in the Statistics Toolbox is actually a *family* of distributions, related to each other by the functional form of their common pdf, but differing from each other in the specific expression of certain characteristic parameters.

A familiar example is the family of normal distributions. The pdf for the normal family is

$$f(t) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(t-\mu)^2}{2\sigma^2}}$$

Plotted, this function always produces a "bell-shaped curve." The shape of the curve, however, will depend on the choice of the two defining parameters: the mean μ and the standard deviation σ.

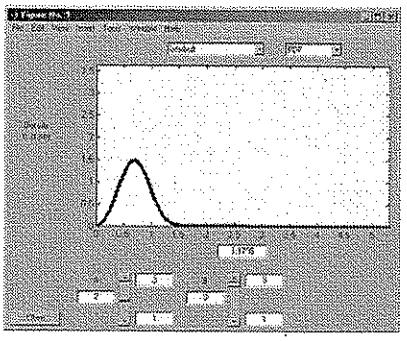
Each distribution has its own set of defining parameters which allow you to "tune" its shape to a particular data set. Each distribution function mentioned on the previous page (with the exception of the `distfit` functions) has required input arguments that specify these parameter values.

One recommended way to find out more about the parameters that define a particular distribution `dist` is to read the documentation:

```
>> doc distpdf
```

Another way is to vary the parameters interactively and see their effect on the shape of the distribution. The Distribution Tool in the Statistics Toolbox makes the dependence immediate and intuitive:

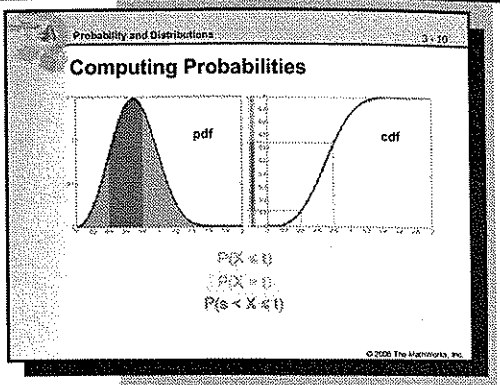
```
>> disttool
```



Try

```
>> t = 0:0.01:2;
>> a = 2; b = 3;
>> f = ...
weibpdf(t,a,b);
>> plot(t,f)

>> hold on
>> [m,v] = ...
weibstat(a,b)
>> M = ...
weibpdf(m,a,b);
>> plot( ...
[m m],[0 M],'r')

>> hold on
>> F = ...
weibcdf(t,a,b);
>> plot(t,F,'m')

>> disttool
```

Computing Probabilities



The probability that a discrete random variable X takes on any particular value t is computed with its pdf f:

$$P(X = t) = f(t)$$

Of course, $P(X = t) = 0$ for any particular value t of a continuous random variable X.

More typically, we will want to compute the probability that a random variable X (discrete or continuous) will lie in a particular *range* of values. That is, we will want the probability of an event.

Event probabilities are computed with the a random variable's cdf. For discrete distributions with pdf f, the cdf F is given by

$$F(t) = \sum_{s \le t} f(s)$$

For continuous distributions with pdf f, the cdf F is given by

$$F(t) = \int_{-\infty}^{t} f(s)\, ds$$

Then

$$P(X \le t) = F(t)$$

$$P(X > t) = 1 - F(t)$$

$$P(s < X \le t) = F(t) - F(s)$$

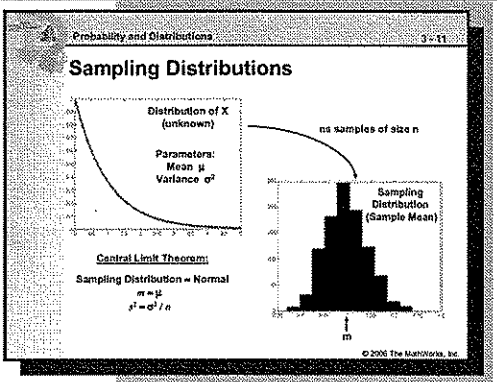
Suppose, for example, that a quality assurance inspector tests 200 circuit boards a day, 2% of which have defects. The number of defects found will have a binomial distribution. At right we find

- The probability that no defective boards are found.
- The probability that more than 5 defective boards are found.
- The probability that between 5 and 10 defective boards are found.

Try

```
>> binopdf( ...
0,200,0.02)

>> 1 - ...
binocdf(...
5,200,0.02)

>> binocdf(...
10,200,0.02) ...
- binocdf(...
5,200,0.02)

>> mean = ...
binostat(200,0.02)
```

Sampling Distributions



One common form of random experiment is to take a sample from a given population. Depending on the outcome (sample), the statistics we compute to estimate population parameters–means, proportions, correlations, etc.–will vary. Statistics are just random variables on the sample space.

The *sampling distribution* of a statistic is the distribution of its values when considered over all possible samples of the same size from the same population. It is the underlying probability distribution of the statistic, and it is generally unknown.

Providing an accurate description of the sampling distribution is crucial to producing reliable simulations (Chapter 6), performing hypothesis tests, and computing confidence intervals (Chapter 7).

We may estimate sampling distributions by taking many samples, and computing the statistic many times. The *sample mean* may be used to estimate the population parameter measured by the statistic. The sample standard deviation is often called the *standard error*. The accuracy of these estimates depends on both the the sampling method and the number of samples used to create the distribution.

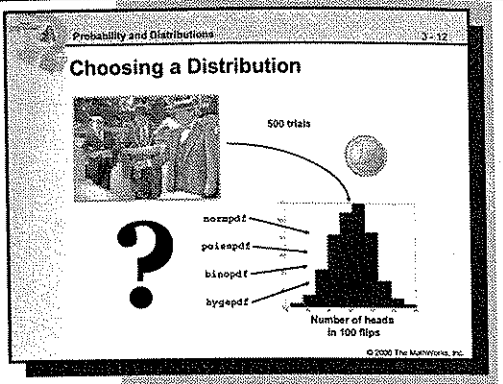
We say that a statistic is *unbiased* if the sample mean is equal to the true value of the population parameter it estimates. Unbiased statistics reflect sampling procedures with no systematic tendency to overestimate or underestimate the parameter.

The *Central Limit Theorem* describes the behavior of the sample mean with increasing sample size. Suppose that a statistic X has an underlying probability distribution with density $f(t)$, mean μ, and variance σ^2. The Central Limit Theorem says that, *regardless of f*, the sample mean will have an approximate normal distribution, with mean μ and variance σ^2/n, where n is the sample size. As n increases, the distribution of the sample mean approaches a normal distribution regardless of how X is distributed. This is important, because we can use this information to estimate the error when we use the sample mean to estimate the population mean μ.

Try

```
>> mu = 1;

>> t = 0:.01:5;
>> f = exppdf(t,mu);
>> plot(t,f)
>> [M,S] = ...
expstat(mu)

>> n = 500;
>> ns = 1000;
>> samples = ...
exprnd(mu,n,ns);
>> means = ...
mean(samples);
>> hist(means)
>> m = mean(means)
>> M
>> v = var(means)
>> (S^2)/n
```

Choosing a Distribution



The goal of data modeling is to find a distribution that accurately represents the data source. Given a sampling distribution, we typically follow a two-stage process:

- Identify a suitable distribution family.
- Tune the distribution parameters to fit the data.

The sampling distributions of many common statistics, like the sample mean, are well known. If we do not know the distribution of our statistic, however, we must pick from among the available pdfs. This step is often the most difficult, as it requires some knowledge, or some assumptions, about the nature of the underlying process producing the data.

It is important to become familiar with the assumptions that produce the standard models. Here are a few examples:

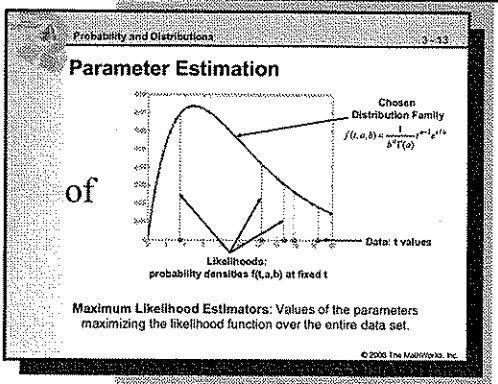
- The *Normal Distribution* is based on the Central Limit Theorem, and is used to model sample means.
- The *Binomial Distribution* models the number of "successes" in repeated trials of a two-outcome experiment where the probability of success for any one trial is constant.
- The *Exponential Distribution* is used primarily to model the duration of time until some outcome.
- The *Poisson Distribution* models numbers of events over a fixed interval of time, distance, etc.
- The *Hypergeometric Distribution* models sampling from a population without replacement.

You may learn more about the background behind the distributions represented in the Statistics Toolbox, and their typical modeling contexts, by reading about their pdfs in the documentation.

Try

```
>> edit coinflips

>> n = 100;
>> p = 0.5;
>> N = 500;
>> T = ...
coinflips(n,p,N);
>> S = sum(T);
>> hist(S)

>> figure
>> s = ...
binornd( ...
n,p,[1 500]);
>> hist(s)
```

Parameter Estimation



Once a distribution family has been chosen, the second stage of the data modeling process attempts to identify the *one* member of the family that best fits the available data. This means choosing best values for the distribution's parameters.

The *likelihood function* of a distribution is just the pdf $f(t, a)$, but viewed as a function of the parameters a rather than the value t. *Maximum likelihood estimators* (MLEs) are values of the parameters that maximize the likelihood function over an entire data set. Intuitively, the MLEs are the values of the parameters with the highest probability of producing the data. In practice, MLEs are found using either optimization methods or formulas from statistical sampling theory that give the optimum values in closed form.

MLEs are computed using the Statistics Toolbox `distfit` functions (with the exception of `normfit`, which computes minimum variance unbiased estimators, or MVUEs). The function `mle` computes MLEs for all of the supported distributions, taking the distribution name as a first argument. The parameter estimation functions will also return confidence intervals for the given estimates. Confidence intervals are discussed further in Chapter 7.

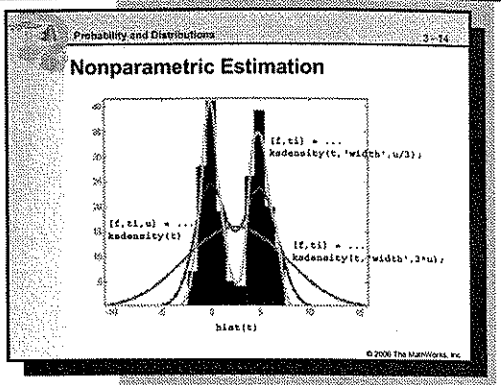
Try

```
>> N = 1000;
>> x = ...
normrnd(5,1,N,1);
>> n = 100;
>> hist(x,n)
>> hold on
>> [nout,xout] = ...
hist(x,n);
>> m = min(xout);
>> M = max(xout);

>> [mu,sigma] = ...
normfit(x);
>> t = m:0.01:M;
>> y = ...
normpdf(t,mu,sigma);
>> s = ((M-m)/n)*N;
>> plot(t,s*y,'r')
```

Note: Here we scale so that the area under the pdf is equal to the area of the histogram (the bin size times the number of data points).

```
>> [estimators, ...
intervals] = ...
mle('norm',x)
```

Nonparametric 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.

```
>> [f,ti] = ksdensity(t)
```

computes a probability density estimate of the sample in the vector `t`. The output `f` is the vector of density values evaluated at the points in `ti`. By default, the density is evaluated at 100 equally spaced points covering the range of the data in `t`.

```
>> [...] = ksdensity(...,'param1',val1,...)
```

specifies parameter name/value pairs to control the density estimation. Valid parameters include `width` and `kernel`.

The parameter `width` controls the bandwidth of the kernel smoothing window. This determines the smoothness of the probability density curve. The default setting is optimal for estimating normal densities. If it is set too high, `ksdensity` smoothes the data so much that important features may be obscured. The wider the window, the more the density will look just like the kernel function. Smaller bandwidths produce rougher curves, but provide better indications of major features, such as multiple modes.

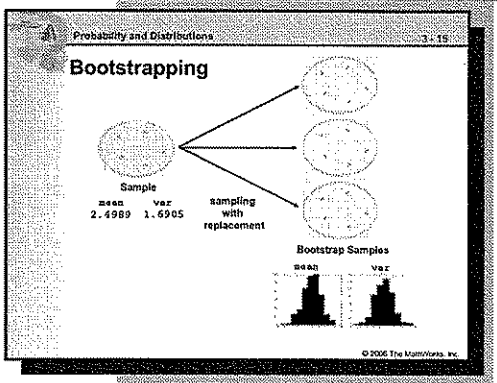
The parameter `kernel` allows you to specify a kernel function by supplying either the function name or a function handle. The four preselected functions, `'normal'`, `'epanechinikov'`, `'box'`, and `'triangle'` are all scaled to have standard deviation equal to one, so the bandwidth parameter means roughly the same thing regardless of the kernel function.

Smooth density estimates are especially useful when comparing distributions. While it is difficult to overlay two histograms to compare them, you can easily overlay smooth density estimates.

Try

```
>> t = ...
[randn(100,1); ...
5+randn(100,1)];
>> hist(t)
>> hold on
>> [f,ti] = ...
ksdensity(t);
>> plot( ...
ti,200*f,'r');

>> hold on
>> [f,ti,u] = ...
ksdensity(t);
>> [f,ti] = ...
ksdensity( ...
t,'width',3*u);
>> plot( ...
ti,200*f,'m');
>> [f,ti] = ...
ksdensity( ...
t,'width',u/3);
>> plot( ...
ti,200*f,'g');
```

Bootstrapping



Resampling methods are sometimes used to acquire information about the uncertainty of statistical estimators. They depend on a single sample (usually a large sample), which is assumed to capture relevant characteristics of the original population. Resampling from the sample is then meant to parallel the original data generating process.

Bootstrapping is a procedure that chooses random samples *with replacement* from a data set. Sampling with replacement means that many different samples are generated from the original sample, even when the number of elements in each bootstrap sample equals the number of elements in the original data set. A range of sample statistics may be generated, allowing us to measure the uncertainty of the quantity that we are estimating.

```
>> 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 the function `bootfun` for analysis. The argument `nboot` must be a positive integer. Each input data set must contain the same number of rows. Bootstrap samples will contain the same number of rows as the data sets, with values randomly chosen (with replacement).

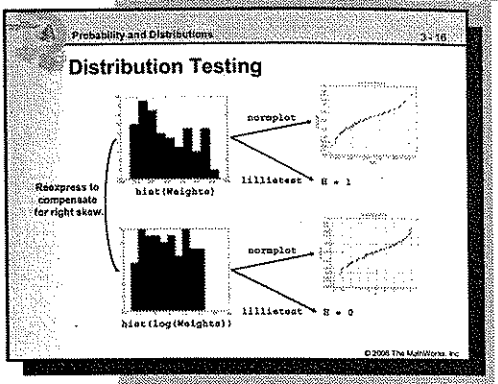
Each row of the output, `bootstat`, contains the results of applying `bootfun` to one set of bootstrap samples. If `bootfun` returns multiple outputs, only the first is stored in `bootstat`. If the first output from `bootfun` is a matrix, the matrix is reshaped to a row vector for storage in `bootstat`.

Bootstrapping is the underlying technique used in Monte Carlo simulation, which we discuss in Chapter 6.

Try

```
>> [m,v] = ...
raylstat(2)
>> d = ...
raylrnd( ...
2,10000,1);
>> sample_m = ...
mean(d)
>> sample_v = ...
var(d)
>> bootmean = ...
bootstrp( ...
500,'mean',d);
>> hist(bootmean)
>> bootvar = ...
bootstrp( ...
500,'var',d);
>> figure
>> hist(bootvar)
```

Distribution Testing



Distribution testing can provide some assurance that the assumptions we make about our data and its distribution are not being violated, and give early warning of any problems.

For example, we often want to see if data–either original data or reexpressed data–is normally distributed. The Statistics Toolbox provides a variety of tools for testing normality.

One method is the normal *probability plot*, implemented by the `normplot` command. The plot shows the empirical probability versus the data value for each point in the sample. A line connects the 25th and 75th percentiles of the data. If all the data points fall near the line, the assumption of normality is reasonable.

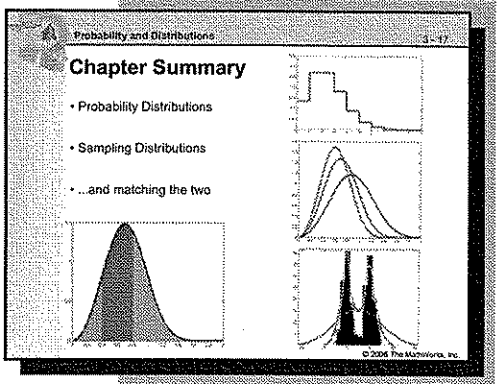
Another method is to run an *hypothesis test*. (Hypothesis testing is discussed more fully in Chapter 7.) The commands `jbtest`, `kstest`, and `lillietest` all run tests for normality. Each evaluates the hypothesis that the data has a normal distribution with unspecified mean and variance, against the alternative that the data set is not normal. The tests return 1 if they can reject the hypothesis that the data has a normal distribution, 0 if they cannot.

Similar tests exists for other distributions. For example, the command `weibplot` produces a probability for the Weibull distribution. The command `kstest2` compares the distributions of values in two data sets, one of which could be generated randomly from a known distribution. The null hypothesis for this test is that the data sets have the same continuous distribution. The alternative hypothesis is that they have different continuous distributions.

Try

```
>> load carsmall
>> hist(Weight)
>> normplot(Weight)
>> lillietest( ...
Weight)

>> reex_Weight = ...
log(Weight);
>> hist( ...
reex_Weight)
>> normplot( ...
reex_Weight)
>> lillietest( ...
reex_Weight)
```

Chapter Summary



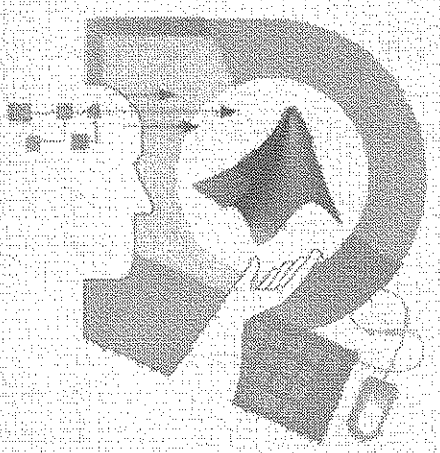
This chapter provided us with a broad overview of the important features of distributions of a single random variable. We discussed the probability distributions that model the underlying processes producing data, and the sampling distributions with which we must work in practice. Much of the chapter was concerned with matching the two: finding probability distributions that can serve as the best models of the characteristic features of our data.

Exercise: Distributions Diagnostics

In the next chapter we broaden our scope to consider relationships among several variables. Chapter 4 considers the two-variable case, and discusses the basic terminology and techniques. Chapter 5 generalizes these approaches to the case of multivariate statistics.

Statistical Methods in MATLAB®

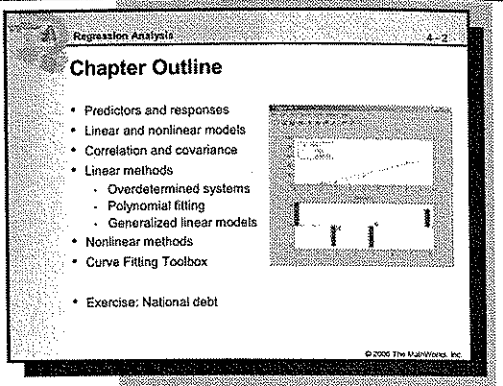
Regression Analysis



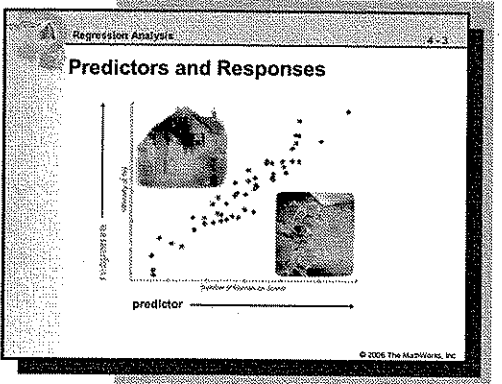
The MathWorks
Training Services

Chapter Outline

- Predictors and responses
- Linear and nonlinear models
- Correlation and covariance
- Linear methods
  - Overdetermined systems
  - Polynomial fitting
  - Generalized linear models
- Nonlinear methods
- Curve Fitting Toolbox
- Exercise: National debt



Predictors and Responses



Many data analysis and modeling problems involve more than a single variable. In this chapter we consider some of the basic issues and analysis techniques in the context of bivariate data. In the next chapter we generalize to multivariate data.

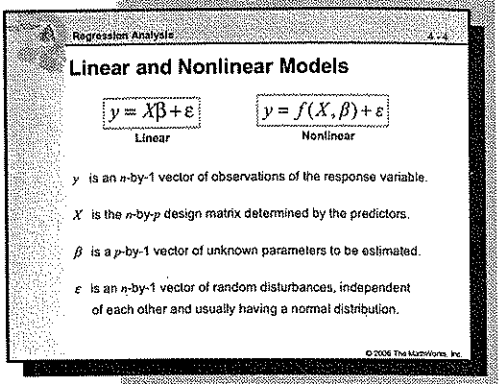
Often we want to compare the distributions of the same variable from different groups. Side-by-side boxplots, histograms, and nonparametric pdfs allow us to visualize these comparisons. In this chapter we will concentrate, however, on the relationships among several variables from the same group.

To analyze these relationships we must first ask if we are simply looking for patterns and correlations, or if we are trying to identify the changes in certain variables that explain, or cause, changes in the other variables. In the latter context, we call certain variables *predictors* and other variables *responses*. Our description of the relationship between predictor and response variables then takes on added significance as a model of some underlying process.

We must remember that calling one variable a predictor and another a response does not compel any variable to *cause* changes in the other. For example, the number of firemen responding to a fire is a good predictor of the intensity of the fire. To say, however, that the firemen are causing the fire's intensity is obviously wrong headed. Generally speaking, when we observe associations and correlations among variables, we must remind ourselves that this is not, in itself, good evidence of causation.

Another issue to keep in mind is that of *lurking variables*. When we are looking at bivariate data, we may observe relationships that are strongly influenced by other variables outside the scope of our investigation. We might, for example, observe a strong correlation between a person's overall health and the make of the car they drive. The association, however, might be mostly a response to the person's income level. In the next chapter, we will see how to add these additional variables to our analyses. In this chapter, we will simply want to to be cautious when drawing conclusions about the population our data describes.

Linear and Nonlinear Models



Linear models are used to 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.

β is a p-by-1 vector of unknown parameters to be estimated.

ε is an n-by-1 vector of random disturbances, independent of each other and usually having a normal distribution.

Think of linear models as ones in which the response variables are expressed as simple linear combinations of the predictor variables plus, perhaps, some noise. In the bivariate case, this means that the two variables will fall approximately along a line when plotted.

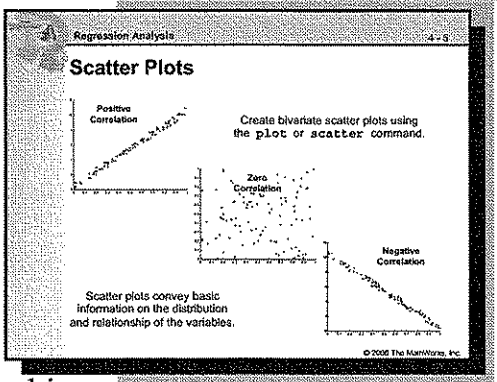
MATLAB and the Statistics Toolbox use the general form of the linear model to solve a variety of specific regression and analysis of variance (ANOVA) problems. (ANOVA is discussed in Chapter 7)

Not all relationships are well described by a linear regression model. The Statistics Toolbox also has functions for fitting a variety of *nonlinear models* of the form

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

where f is any function of X and β.

Scatter Plots

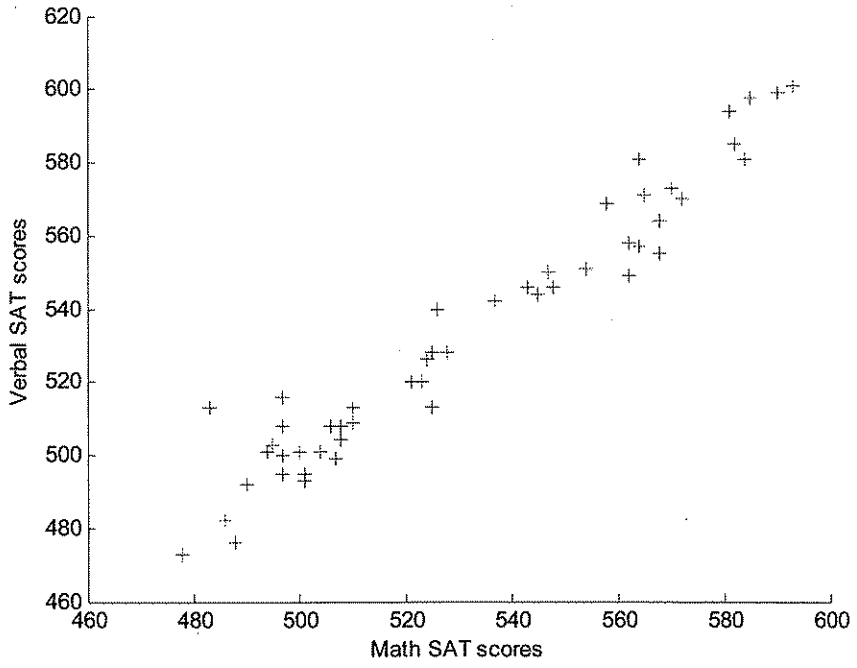


A *scatter plot* displays ordered pairs of data. Plotting paired data provides us with basic information about the distribution of each variable, and the relationship between the two.

A scatter plot is an excellent starting point for analyzing bivariate data, since the scatter plot can indicate a clear relationship between the data, or else indicate that no relationship exists.

For example, the data file `satdata.mat` contains the average score for all 50 states and the District of Columbia on the math and verbal sections of a 1998 Standard Aptitude Test (SAT). A scatter plot can be used to determine whether there is any relationship between the scores.

```
>> load satdata
>> scatter(m,v,'+')
>> xlabel('Math SAT scores')
>> ylabel('Verbal SAT scores')
```



The scatter plot suggests that there is a strong *correlation* between the math and verbal scores, and indicates that a linear fit may be possible. We say there is a *positive correlation* since an increase in the predictor produces an increase in the response. Variables showing *negative correlation* move in opposite directions.

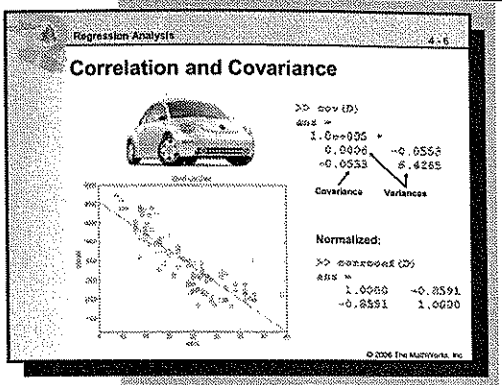
Try

```
>> load satdata
>> scatter(m,v,'+')
```

For *grouped* scatter plots, recall the `gscatter` command from Chapter 2

```
>> doc gscatter
```

Correlation and Covariance



MATLAB and the Statistics Toolbox provide a number of numerical measures of the correlation between variables.

Suppose, as in Chapter 3, that we have a random experiment with sample space S. Suppose that X and Y are both real-valued random variables on S with expected values $E(X)$, $E(Y)$ and variances var(X), var(Y), respectively. The *covariance* of X and Y is defined by

$$\text{cov}(X, Y) = E\{[X - E(X)][Y - E(Y)]\}$$

The *correlation* (or *correlation coefficient*) of X and Y is just a normalized version of covariance, defined by

$$\text{cor}(X, Y) = \text{cov}(X, Y) / [\text{std}(X)\ \text{std}(Y)]$$

Both the covariance and the correlation measure the linear relationship between two random variables. The two measures always have the same sign, indicating the direction of correlation.

The MATLAB functions `cov` and `corrcoef` compute sample statistics to estimate these quantities.

```
>> C = cov(D)
```

computes a covariance matrix `C`. If `D` is a matrix of data, where each row is an observation and each column is a variable, `C` is a symmetric square matrix with size equal to the number of variables. Variances appear on the diagonal, and off-diagonal elements in the (i, j)th and (j, i)th positions give the covariance of variables i and j.

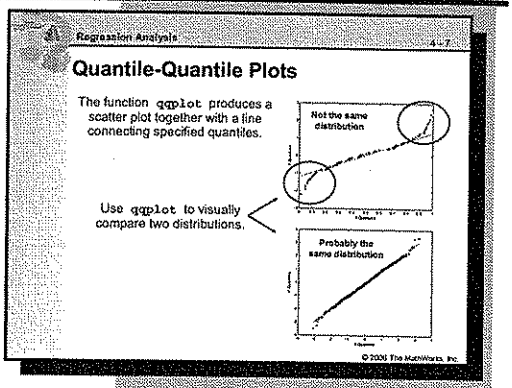
```
>> C = corrcoef(D)
```

computes a matrix of correlations with the same I/O set up as `cov`.

Try

```
>> load carsmall

>> plot( ...
MPG,Weight,'o')
>> lsline

>> D = [MPG,Weight];
>> cov(D)
>> corrcoef(D)

>> edit nanremover
>> D2 = ...
nanremover(D);
>> cov(D2)
>> corrcoef(D2)
```

Quantile-Quantile Plots



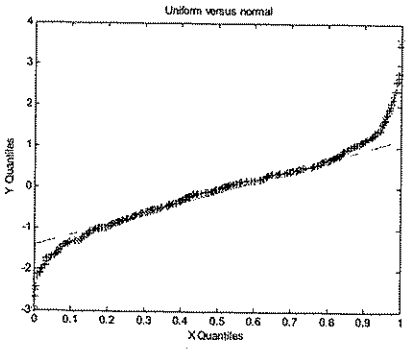
A very useful type of scatter plot is the quantile-quantile plot. Roughly speaking, a quantile of order p is a value where the sample `cdf` crosses p. Quantile-quantile plots are just scatter plots together with a line connecting two specified quantiles in the data sets. By default, the quantiles chosen are the first and third quartiles. This line gives a robust linear fit of the order statistics of the two samples.

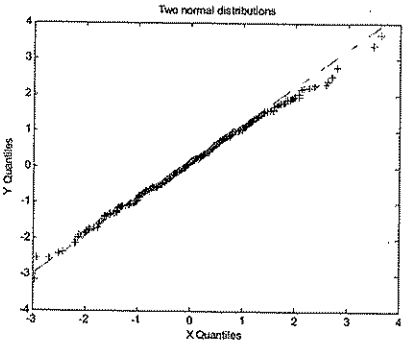
Quantile-quantile plots are typically used to visually compare two distributions. If the distributions come from the same family, the ordered scatter should follow the line.

The Statistics Toolbox implements quantile-quantile plots with the `qqplot` command.

Try the following code, which generates one uniformly distributed data set and two normally distributed ones.

```
>> x = rand(500,1);
>> y = randn(500,1);
>> z = randn(500,1);
>> qqplot(x,y)
>> figure
>> qqplot(y,z)
```





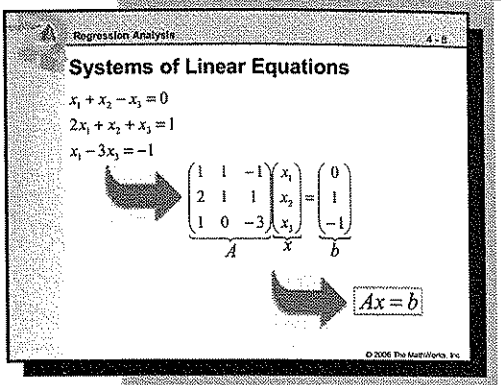
The first plot is nonlinear, confirming that the data sets are not from the same distribution family. The second plot is roughly linear, so it suggests that the data sets do probably come from the same family.

Recall the `normplot` and `weibplot` functions from Chapter 3, which can be used to test a single data set against a normal distribution or a Weibull distribution.

Try

```
>> x = rand(500,1);
>> y = randn(500,1);
>> z = randn(500,1);
>> qqplot(x,y)
>> qqplot(y,z)

>> normplot(x)
>> normplot(y)

>> load satdata
>> qqplot(m,v)
>> corrcoef(m,v)
```

Systems of Linear Equations



We now introduce some basic MATLAB commands for solving systems of linear equations. We will see that the basic methodology presented here is behind most of the linear regression methods in MATLAB and the Statistics Toolbox.

Recall that any system of linear equations can be written succinctly in matrix form. For example,

$$\begin{aligned} x_1 + x_2 - x_3 &= 0 \\ 2x_1 + x_2 + x_3 &= 1 \\ x_1 - 3x_3 &= -1 \end{aligned}$$

can be rewritten as

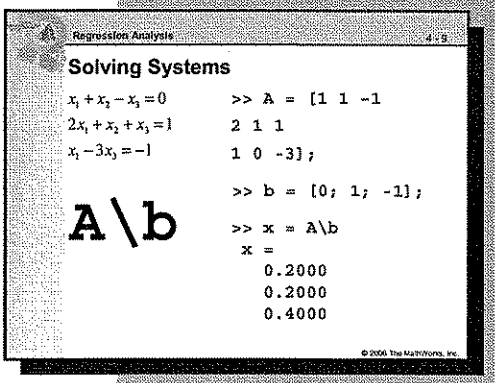
$$\underbrace{\begin{pmatrix} 1 & 1 & -1 \\ 2 & 1 & 1 \\ 1 & 0 & -3 \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}}_{x} = \underbrace{\begin{pmatrix} 0 \\ 1 \\ -1 \end{pmatrix}}_{b}$$

or as

$$Ax = b$$

MATLAB uses a left division operator (\), the *backslash*, to solve systems of linear equations for the unknown vector of variables x.

Solving Systems



We may compare the solution of a matrix equation like $Ax = b$ to the solution of a scalar equation like $7x = 10$:

$$7x = 10 \Rightarrow x = 7^{-1} * 10 = 10/7 \qquad (1)$$

$$Ax = b \Rightarrow x = A^{-1} * b = A \backslash b \qquad (2)$$

In (2) we use the MATLAB backslash operator as a shorthand for left multiplication by A^{-1}. While scalar multiplication is commutative, so that $7^{-1}*10 = 10*7^{-1} = 10/7$, matrix multiplication is not, so that $A^{-1}*b = A\backslash b$ and $b*A^{-1} = b/A$ are generally different.

For example, we can solve the system

$$x_1 + x_2 - x_3 = 0$$
$$2x_1 + x_2 + x_3 = 1$$
$$x_1 - 3x_3 = -1$$

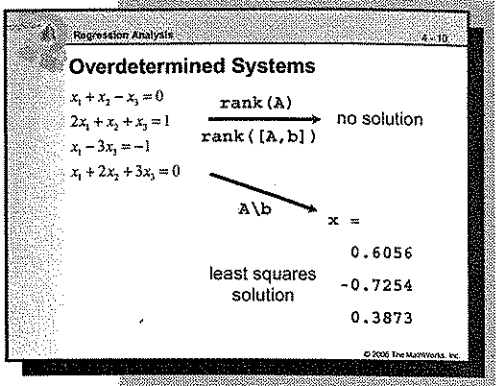
by typing

```
>> A = [1 1 -1; 2 1 1; 1 0 -3];
>> b = [0; 1; -1];
>> x = A\b
 x =
   0.2000
   0.2000
   0.4000
```

Try

```
>> A = ...
[1 1 -1
2 1 1
1 0 -3];
>> b = [0;1;-1];

>> x = A\b

>> edit slashspeed
>> slashspeed
>> edit slashspeed2
>> slashspeed2
```

Overdetermined Systems



If we add a fourth equation to our system, it becomes *overdetermined*:

$$x_1 + x_2 - x_3 = 0$$
$$2x_1 + x_2 + x_3 = 1$$
$$x_1 - 3x_3 = -1$$
$$x_1 + 2x_2 + 3x_3 = 0$$

Overdetermined systems have no solutions. When a system has more constraints (equations) than degrees of freedom (variables), it becomes overdetermined – unless some of the equations are simply linear combinations of the others.

When a linear system is overdetermined, the MATLAB backslash operator will still return values:

```
>> x = A\b
 x =
   0.6056
  -0.7254
   0.3873
```

For overdetermined systems, the backslash operator solves an optimization problem

$$\min_x \|Ax - b\|$$

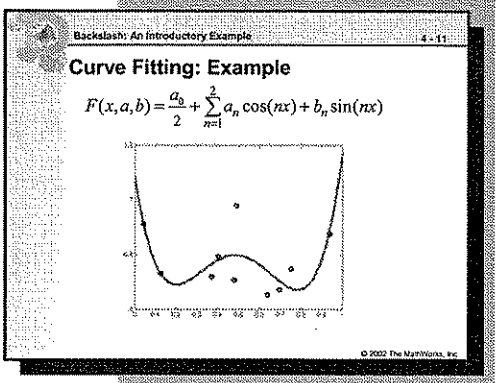
That is, backslash finds an x such that the norm of the difference between Ax and b is as small as possible. This x is called a *least squares solution* to the system.

Any problem expressed in terms of an A and a b for which we want to minimize the norm of the difference Ax-b can be solved with backslash. Many regression (curve fitting) problems take this form, including the example on the following page.

Try

```
>> A = ...
[1 1 -1
2 1 1
1 0 -3
1 2 3];
>> b = [0;1;-1;0];

>> rank(A)
>> rank([A,b])

>> x = A\b
```

Curve Fitting: Example



Data Vectors: *xdata, ydata*

```
>> m = 10;
>> xdata = rand(m,1);
>> ydata = rand(m,1);
```

Functional Form: Fourier series of order 2

$$F(x,a,b)=\frac{a_0}{2}+\sum_{n=1}^{2}(a_n\cos(nx)+b_n\sin(nx))$$

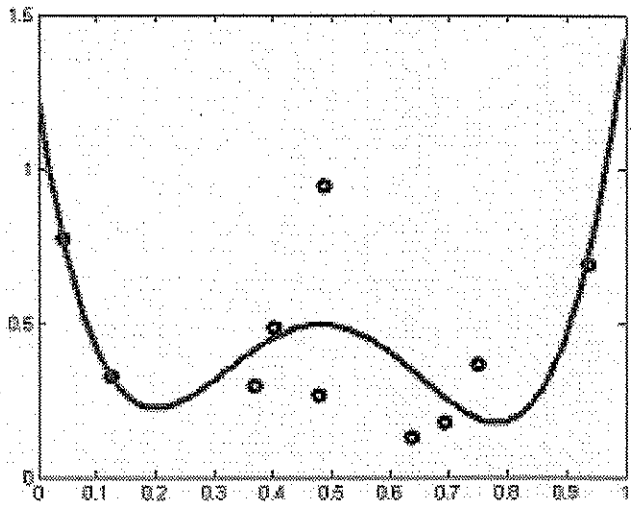
Problem: Find vectors *a*, *b* to minimize the sum of squares.

$$\min_{a,b}\sum_{i=1}^{m}[F(x_i,a,b)-y_i]^2$$

Solution:

```
>> M = ...
[0.5*ones(m,1),cos(xdata),cos(2*xdata), ...
 sin(xdata), sin(2*xdata)];

>> c = M\ydata;
```

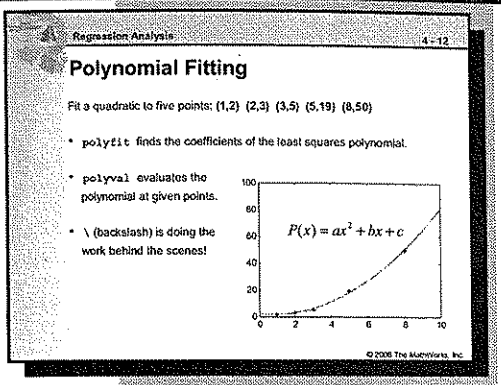
where $c=(a_0, a_1, a_2, b_1, b_2)$.



Try

```
>> m = 10;

>> xdata=rand(m,1);

>> ydata=rand(m,1);

>> M = ...
[.5*ones(m,1), ...
cos(xdata), ...
cos(2*xdata), ...
sin(xdata), ...
sin(2*xdata)];

>> c = M\ydata;

>> x = 0:.01:1;

>> X = ...
[.5*ones(size(x)),...
cos(x), ...
cos(2*x), ...
sin(x), ...
sin(2*x)];

>> y = X*c;

>> plot( ...
xdata,ydata,'o')

>> hold on

>> plot(x,y,'r')
```

Polynomial Fitting



If we specifically want to fit polynomials to our data (as a particular kind of least squares fit), MATLAB offers a number of functions and graphical user interfaces (GUIs).

MATLAB uses the `polyfit` and `polyval` functions to determine the coefficients of a best fit polynomial of arbitrary degree. The code for `polyfit` sets up a call to the backslash operator, which performs the actual fit.

Here is an example of the use of these commands:

```
>> x = [1 2 3 5 8];
>> y = [2 3 5 19 50];

>> p = polyfit(x,y,2)  % Get polynomial
p =
    0.9406   -1.5280    2.2308
```

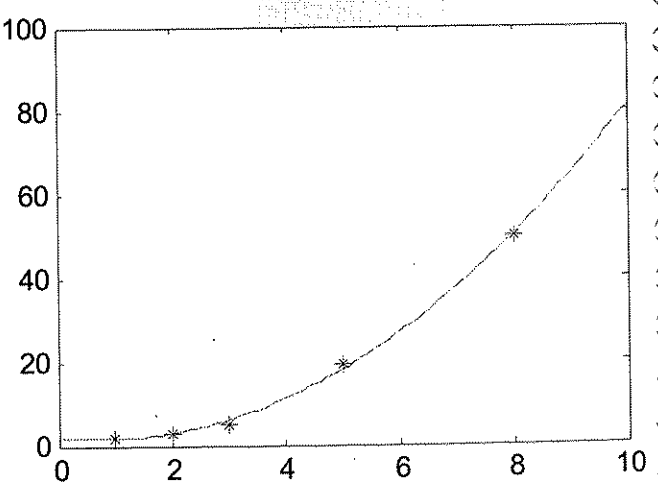
The polynomial's coefficients are listed from highest to lowest degree, so p = $0.9406x^2 - 1.5280x + 2.2308$.

The `polyval` function can be used with any polynomial and data values; one way to use it to calculate values from a polynomial fit:

```
>> yf = polyval(p, x)
y_fit =
1.6434    2.9371    6.1119   18.1049   50.2028
```

A fitted curve can also be plotted along with data:

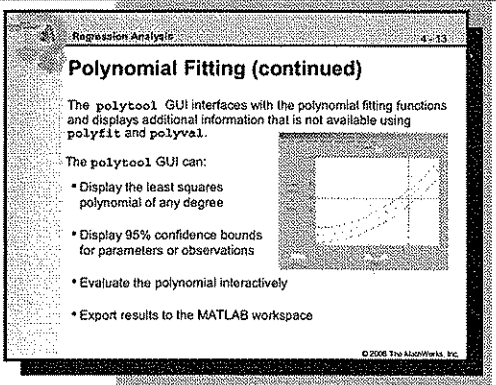
```
>> x_pl = 0:.1:10;
>> y_pl = polyval(p, x_pl);
>> plot(x, y, '*', x_pl, y_pl, 'r')
```



Try

```
>> x = [1 2 3 5 8];
>> y = [2 3 5 19 50];

>> p = polyfit(x,y,2)
>> yf = polyval(p,x)

>> x_pl = 0:.1:10;
>> y_pl = ...
polyval(p, x_pl);

>> plot(...
x,y,'*',x_pl,y_pl,'r')
```

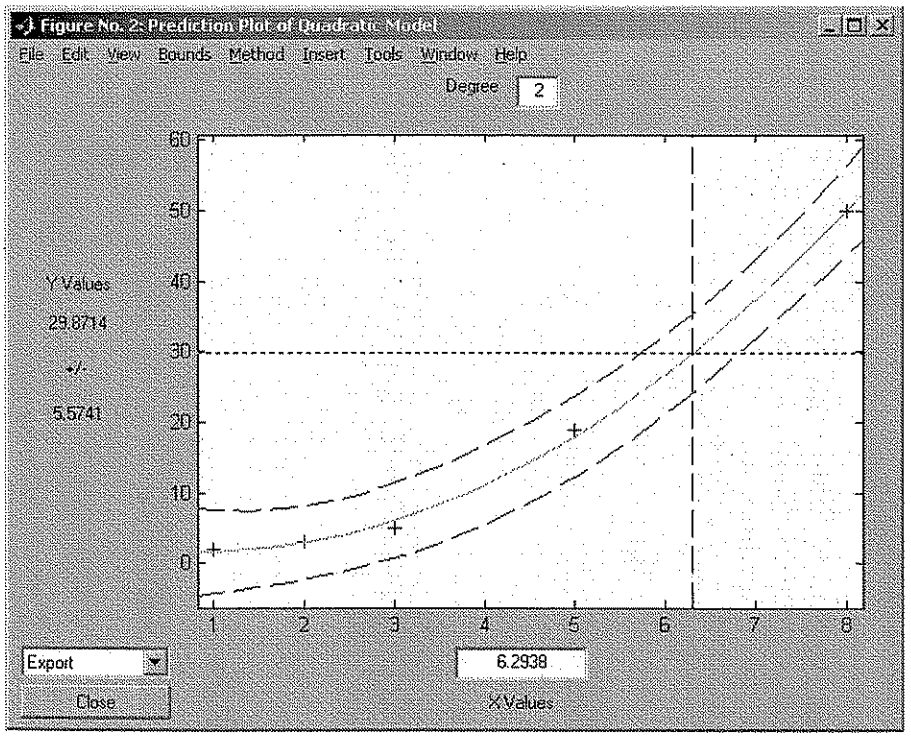
Polynomial Fitting (continued)



The `polytool` GUI is available in the Statistics Toolbox as a graphic interface to the polynomial fitting functions. The `polytool` GUI displays additional information that is not available using `polyfit` and `polyval`.

The `polytool` GUI can

- Display the least squares polynomial of any degree
- Display 95% confidence bounds for parameters or observations
- Evaluate the polynomial interactively
- Export results to the MATLAB workspace



Try

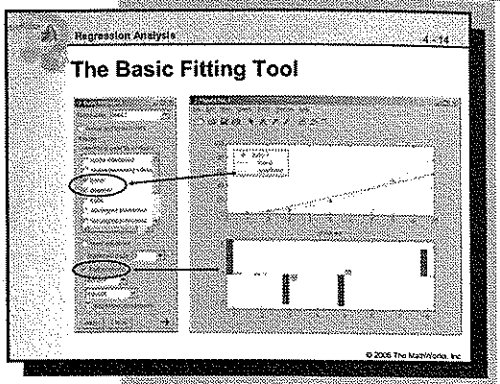
```
>> x = [1 2 3 5 8];
>> y = [2 3 5 19 50];
>> polytool(x,y)
```

Also try

```
>> [p,s] = ...
polyfit(x,y,2);
>> [yfit,delta] = ...
polyconf(p,x,s,0.05)
```

The Statistics Toolbox function `polyconf` can be used from the command line to generate the confidence intervals `polytool` displays, or to create confidence bounds of different levels.

The Basic Fitting Tool



The Basic Fitting Tool is available in MATLAB as a graphic interface for fitting both polynomials and splines to data. It uses `polyfit` and `polyval` to perform its calculations.

The Basic Fitting Tool can

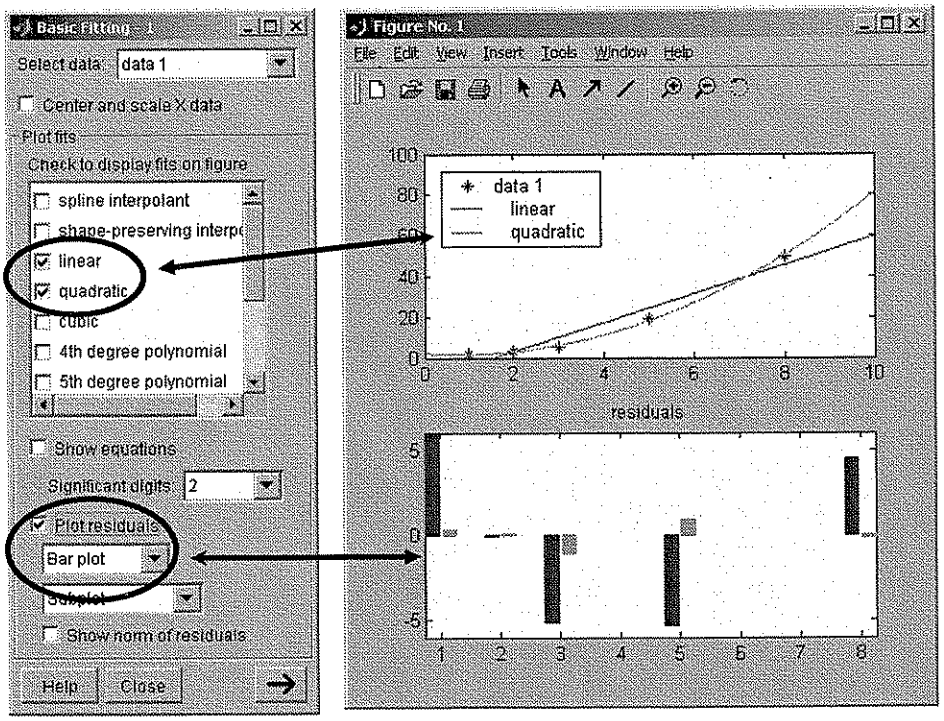
- Display the least squares polynomial of first through tenth degree
- Evaluate the polynomial interactively
- Export results to the MATLAB workspace
- View residuals and their norm in a subplot

Try

```
>> x = [1 2 3 5 8];
>> y = [2 3 5 19 50];
>> plot(x,y,'*');
>> axis([0 10 0 100]);
```

Then select

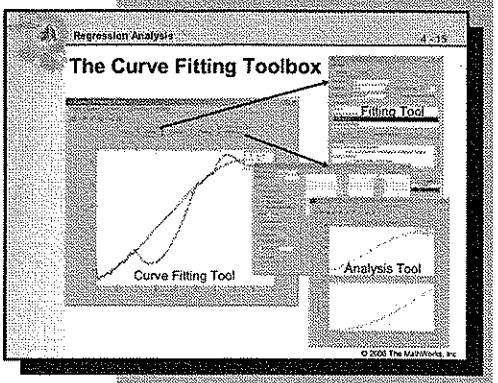
Tools → Basic Fitting



Although the Basic Fitting Tool uses `polyfit` and `polyval`, it is not able to display confidence intervals, since the functions required to do this are in the Statistics Toolbox.

The Basic Fitting Tool can calculate interpolating cubic splines as well as best fit polynomials. For other nonlinear fitting options, use the Statistics Toolbox function `nlintool`, or the Curve Fitting Toolbox. Each will be described later in this chapter.

The Curve Fitting Toolbox

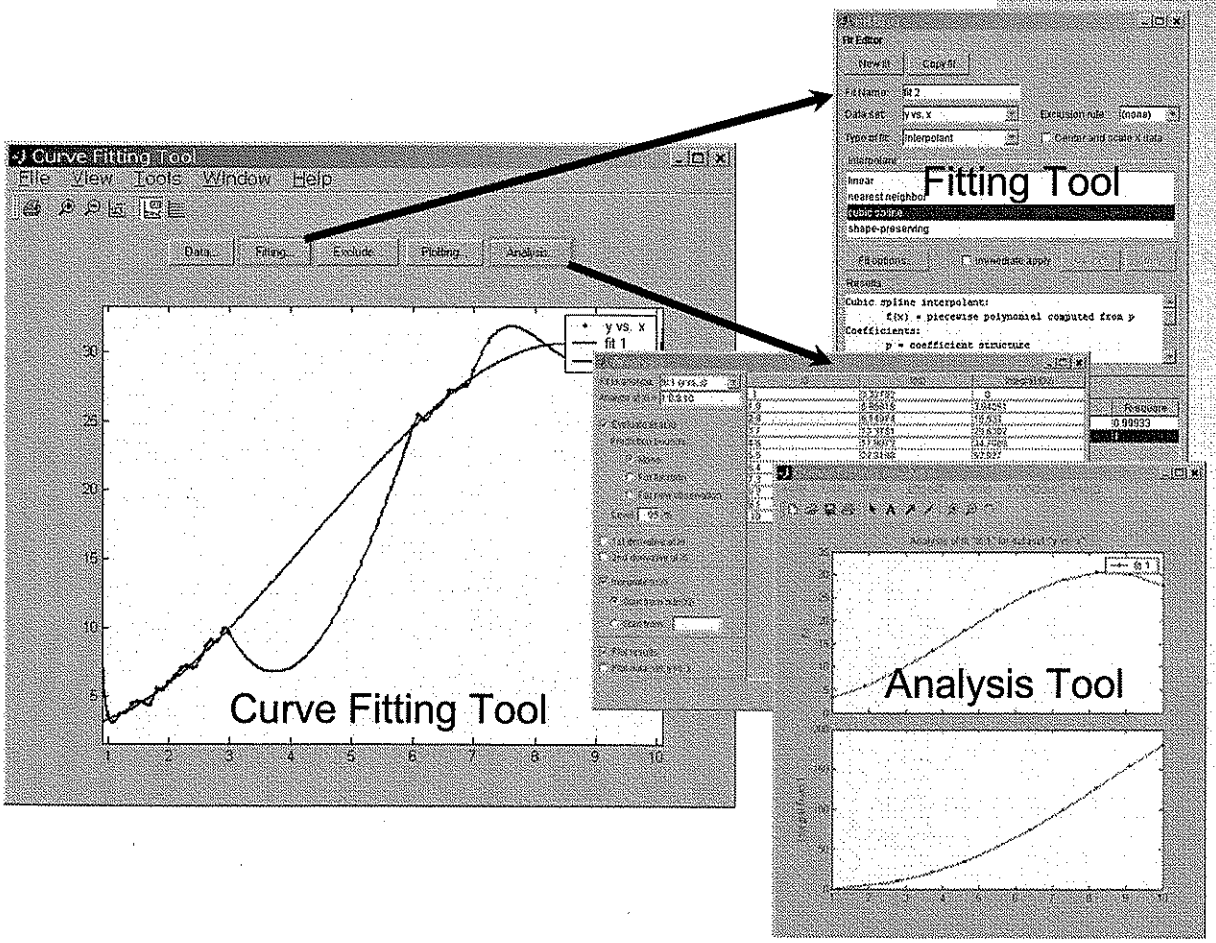


The Curve Fitting Toolbox is a collection of graphical user interfaces (GUIs) and M-file functions designed to simplify and unify tasks associated with curve fitting.

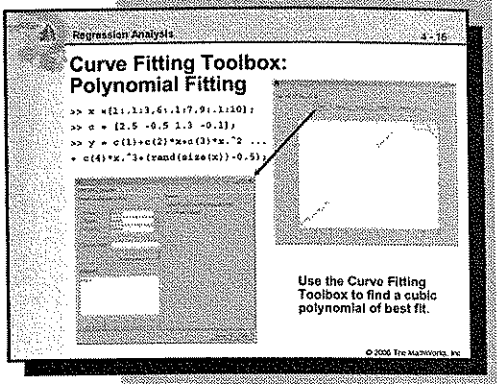
The toolbox serves as a central access point for curve-fitting routines otherwise implemented by functions in MATLAB, the Optimization Toolbox, the Statistics Toolbox, and the Spline Toolbox. It also offers many additional features related to data analysis. Preprocessing routines are available for data scaling, sectioning, smoothing, and removal of outliers. Postprocessing routines include those for interpolation, extrapolation, differentiation, and integration of fits. An extensive library of linear and nonlinear parametric models allows for optimal fitting across families. All tasks are accomplished quickly through the `cftool` GUI.

Try (requires the Curve Fitting Toolbox)

```
>> cftool
```



Curve Fitting Toolbox: Polynomial Fitting



The following example illustrates the use of the `cftool` GUI from the Curve Fitting Toolbox. We show polynomial fitting to compare with the methods introduced previously.

Suppose we want to fit a cubic polynomial model to the following data:

```
>> x = [1:.1:3,6:.1:7,9:.1:10];
>> c = [2.5 -0.5 1.3 -0.1];
>> y = c(1)+c(2)*x+c(3)*x.^2+c(4)*x.^3 ...
       +(rand(size(x))-0.5);
```

The data includes two intervals of missing values.

Open the `cftool` by typing

```
>> cftool
```

Then

1. Import the data into the `cftool` by clicking the **Data** button.

2. On the **Data** tab, choose the `x` and `y` data from the workspace.

3. Assign the data a name in the **Data Set Name** box.

4. Click the **Create Data Set** button. The **Smooth** tab in the **Data** window allows for smoothing of the data using a variety of methods.

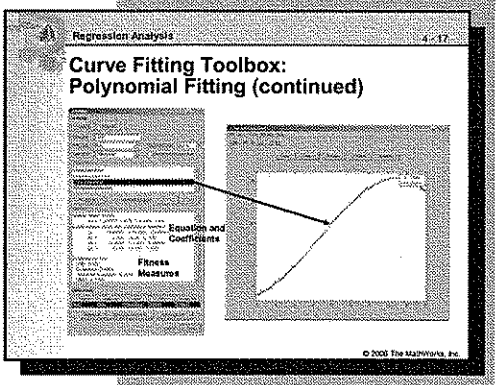
5. Once the data set has been imported, dismiss the import window by clicking on the **Close** button.

A plot of the data is displayed in the `cftool` window.

Try

```
>> x = ...
[1:.1:3,...
 6:.1:7, ...
 9:.1:10];

>> c = ...
[2.5 -0.5 1.3 -0.1];

>> y = ...
c(1) + ...
c(2)*x + ...
c(3)*x.^2 +...
c(4)*x.^3 + ...
rand(size(x))-0.5;
```

Curve Fitting Toolbox: Polynomial Fitting (continued)



To fit a curve:

1. Click the **Fitting** button.

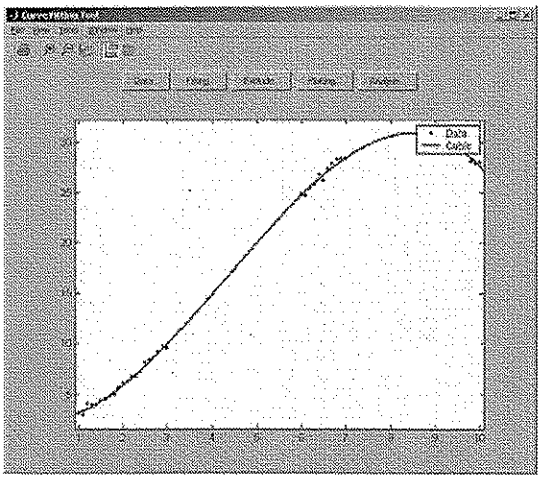
2. In the Fit Editor, click the **New Fit** button.

3. Assign a name to the fit in the **Fit Name** box.

4. Select the data and choose `Polynomial` as the **Type of Fit**.

5. Select `Cubic Polynomial` as the **Polynomial**.

6. Select **Immediate Apply** to see the plot.

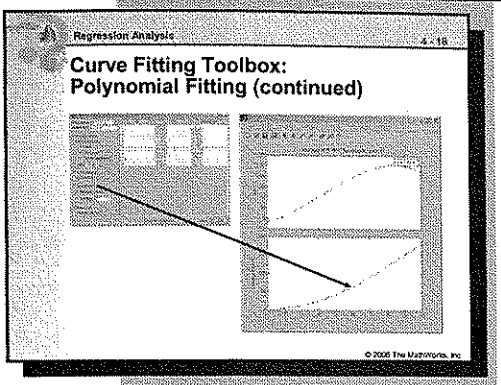


The window displays the fitted curve, its coefficients, and a measure of the goodness of fit. Values may be saved to the workspace using the **Save to Workspace** button. Values of interest may be selected using the **Table Options** button.

Compare the cubic polynomial fit with a cubic spline fit by repeating the steps above, except

- Replace `Polynomial` with `Interpolant` as the **Type of Fit**.
- Choose `Cubic Spline` as the **Interpolant**.

Curve Fitting Toolbox: Polynomial Fitting (continued)

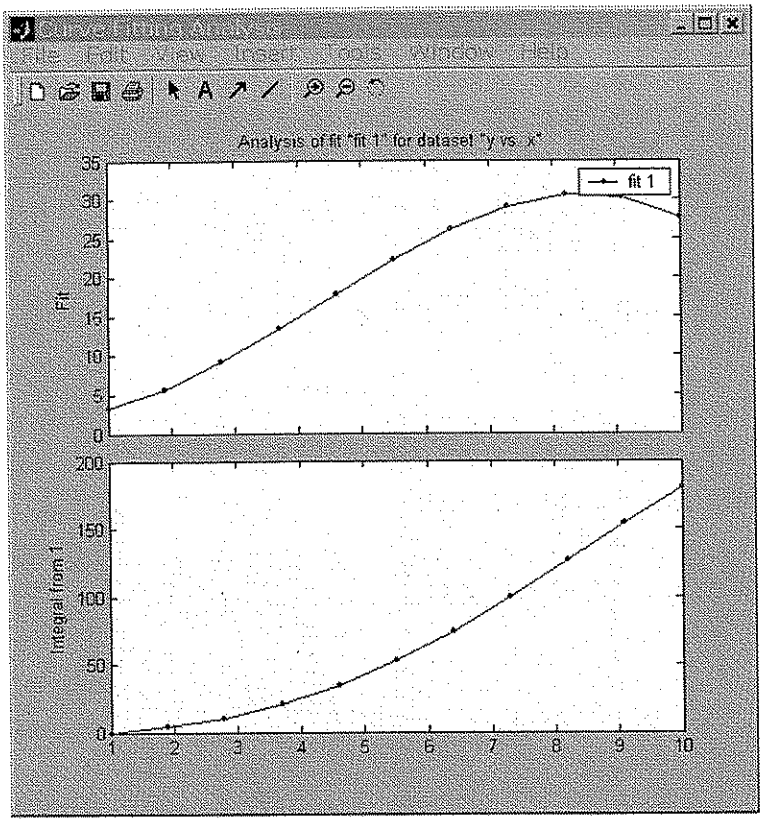


The Curve Fitting Toolbox gives you the option to easily exclude intervals of data, fitting only selected points. Data exclusion rules are specified by clicking the **Exclude** button.

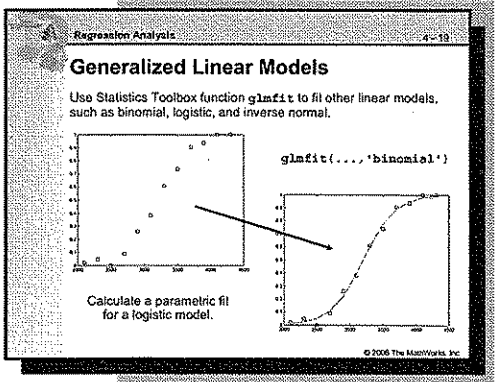
The **Analysis** button gives you access to a variety of data analysis tools. You can

- Interpolate and extrapolate from the fit over specific ranges.
- Compute and plot first and second derivatives of the fit.
- Compute and plot the integral of the fit over specific ranges.

The results of the analysis can be saved to the workspace using the **Save to Workspace** button.



Generalized Linear Models



The Statistics Toolbox function `glmfit` can be used to calculate parameters for generalized linear models.

Using `glmfit` allows parameters to be calculated for responses that are normal, binomial, Poisson, gamma, or inverse normal.

The Statistics Toolbox also includes the `glmval` function, which is used much like the MATLAB `polyval` function. To use `glmval`, provide the output parameters from `glmfit`, the values on which to calculate the fit, and a string describing which generalized linear model you are working with.

The following example uses `glmfit` and `glmval` to calculate a best-fit logistic curve for the proportion of cars with poor gas mileage, based on vehicle weight.

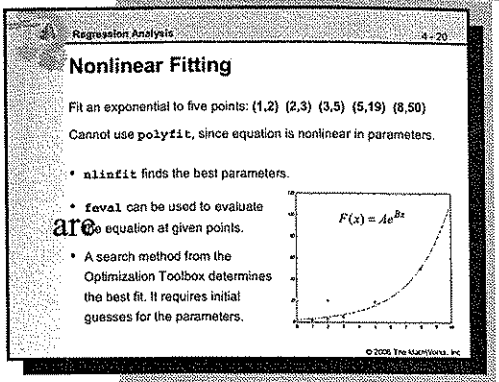
```
>> w = (2100:200:4300)';
>> poor = [1 2 0 3 8 8 14 17 19 15 17 21]';
>> tot = ...
[48 42 31 34 31 21 23 23 21 16 17 21]';

% Calculate proportion of poor mileage cars
>> p = poor ./ tot;

% Calculate generalized linear model
>> [bl,dl,sl] = ...
glmfit(w,[poor tot],'binomial');

% Evaluate model for the original weights
>> yfit = glmval(bl,w,'logit');

% Plot the fit alongside the original data
>> plot(w,p,'bs')
>> hold on
>> plot(w,yfit,'r-')
```

The `glmfit` and `glmval` functions can be very useful in cases where linear, polynomial, and spline models do not provide a satisfactory fit. Here, using the binomial model meets the requirement that the proportion of failing cars lies between 0 and 1.

Try

```
>> load carinfo

>> [bl,dl,sl] = ...
glmfit(w,...
[poor tot], ...
'binomial');

>> yfit = ...
glmval(bl,w,'logit');

>> plot(w,p,'bs')
>> hold on
>> plot(w,yfit,'r-')
```

Nonlinear Fitting



As we have seen, the MATLAB backslash operator is used (often behind the scenes) to compute best fits for models that are linear in the parameters, such as

$$y = Ax^3 + Be^x + C\sin x$$

The backslash operator *cannot* be used for models that are nonlinear, such as

$$y = Ae^{Bx}$$

The Statistics Toolbox provides the `nlinfit` function to compute best fits for nonlinear equations. The basic syntax is

```
>> beta = nlinfit(x, y, fun, b0)
```

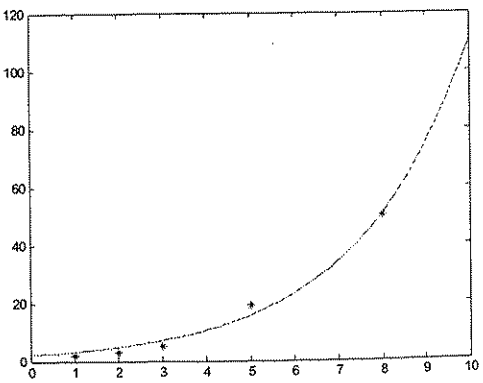
Here `x` and `y` are the given data, `fun` is a function handle or inline function describing the fit, and `b0` is a vector of initial guesses for the parameters. The result `beta` gives the parameters of best fit.

Here is an example:

```
>> x = [1 2 3 5 8]; y = [2 3 5 19 50];
>> fun = inline('b(1).*exp(b(2).*x)','b','x');
>> beta = nlinfit(x, y, fun, [1;.5])
beta =
    2.2092
    0.3911
```

```
% Use values of beta to plot fitting curve
>> x_pl = 0:.1:10;
>> y_pl = feval(fun, beta, x_pl);
>> plot(x, y, '*', x_pl, y_pl, 'r')
```

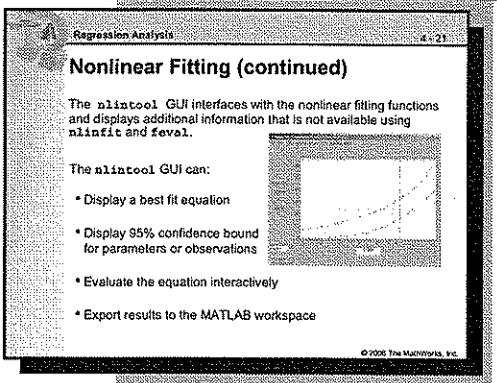
At right is the plot of the fitted curve, $y = (2.2092)\, e^{0.3911x}$, along with the five data points. `nlinfit` calls the Optimization Toolbox and uses a Levenberg-Marquardt method to optimize the parameter values.



Try

```
>> x = [1 2 3 5 8];
>> y = [2 3 5 19 50];

>> fun = ...
inline( ...
'b(1).*exp(b(2).*x)',
'b','x');
>> beta = ...
nlinfit( ...
x,y,fun,[1;.5])

>> x_pl = 0:.1:10;
>> y_pl = ...
feval(fun,beta,x_pl);
>> plot( ...
x,y,'*',...
x_pl,y_pl,'r')
```

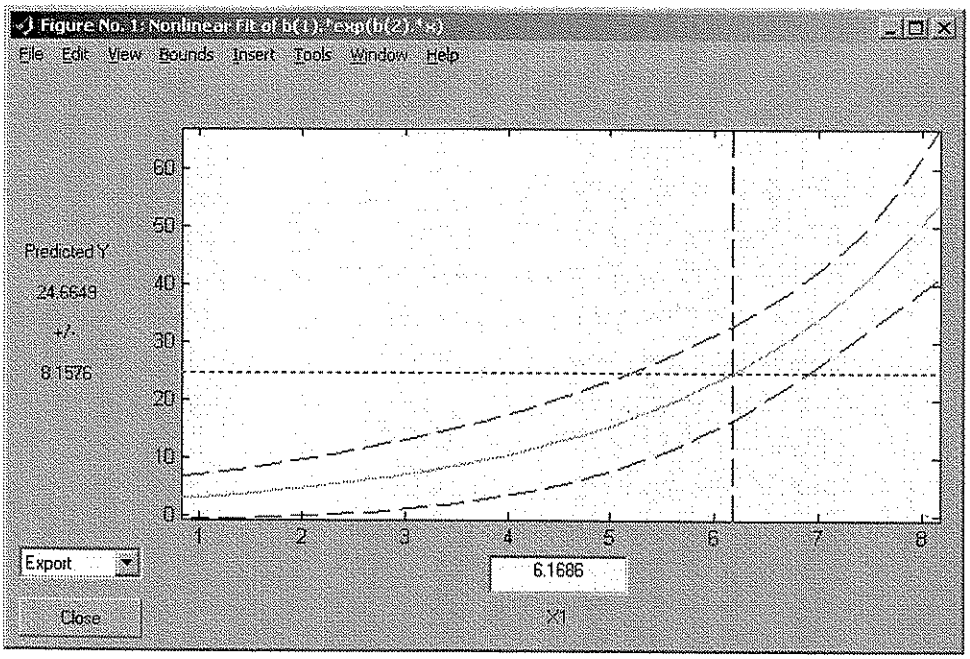
Nonlinear Fitting (continued)



The `nlintool` GUI is available in the Statistics Toolbox as a graphic interface for fitting parameters in nonlinear models. The `nlintool` GUI displays additional information that is not available using `nlinfit` and `feval`.

The `nlintool` GUI can

- Display the best fit, as computed by `nlinfit`
- Display 95% confidence bounds for parameters or observations
- Evaluate the nonlinear equation interactively
- Export results to the MATLAB workspace



The Statistics Toolbox functions `nlparci` and `nlpredci` can be used from the command line to generate the confidence intervals `nlintool` displays.

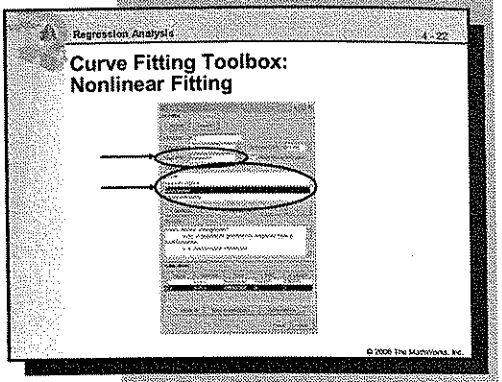
Try

```
>> x = [1 2 3 5 8];
>> y = [2 3 5 19 50];

>> fun = ...
inline( ...
'b(1).*exp(b(2).*x)',
'b','x');
>> nlintool( ...
x,y,fun,[1;.5])
```

Also try

```
>> [beta,r,J] = ...
nlinfit( ...
x,y,fun,[1;.5]);

% Parameter C.I.
>> ci = ...
nlparci(beta,r,J)

% Prediction C.I.
>> x_fit = 0:.1:10;
>> [y_fit, ci] = ...
nlpredci(...
fun,x_fit,beta,r,J);
```

Curve Fitting Toolbox: Nonlinear Fitting



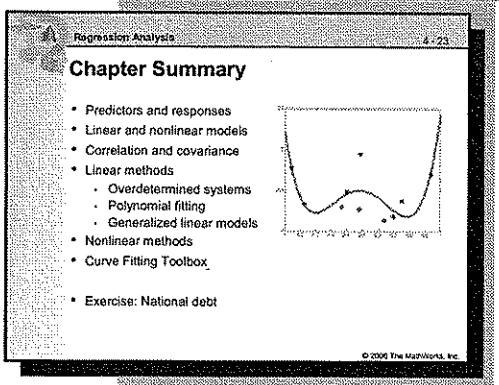
For nonlinear fitting over a variety of functional families, the Curve Fitting Toolbox provides a library of models, including

- Exponential
- Rational (to degree 5/5)
- Peak (Gaussian)
- Distribution (Weibull)
- Fourier
- Power
- Splines (cubic and smoothing)
- Interpolant (linear, nearest-neighbor, cubic-spline, shape-preserving)
- Custom-developed models

Library models use optimized solver parameters, and some nonlinear models compute optimized starting points. You can override default parameters, or develop your own set of custom models. You can save and reload an unlimited number of custom models for repeated use or further modification.

The Curve Fitting Toolbox also offers a variety of fitting methods, including linear least squares and nonlinear least squares via trust region, Levenberg-Marquardt, or Gauss-Newton algorithms. You may choose to assign both weights and bounds on the coefficients. You may also choose two forms of robust fitting: bisquare or least absolute residuals.

Chapter Summary



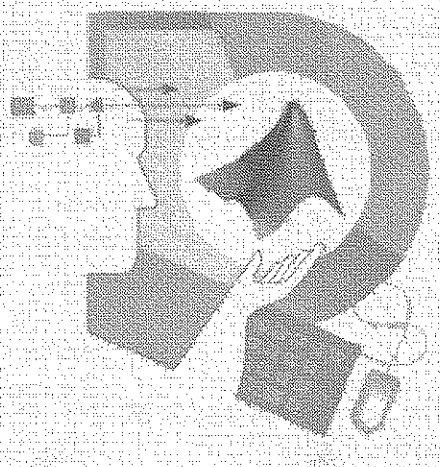
- Predictors and responses
- Linear and nonlinear models
- Correlation and covariance
- Linear methods
  - Overdetermined systems
  - Polynomial fitting
  - Generalized linear models
- Nonlinear methods
- Curve Fitting Toolbox
- Exercise: National debt

Exercise: National Debt

In the next chapter we generalize some of the bivariate techniques here to multivariate data, with a continued emphasis on modeling data relationships. We also introduce some new techniques that are particularly useful when working with large, multivariate data sets.

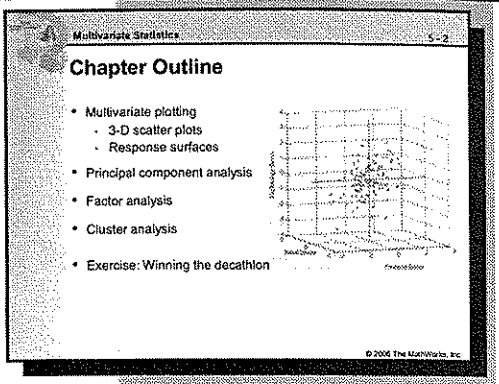
Statistical Methods in MATLAB®

Multivariate Statistics



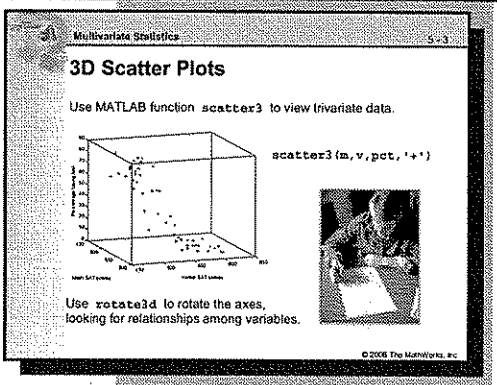
The MathWorks
Training Services

Chapter Outline



- Multivariate plotting
  - 3-D scatter plots
  - Response surfaces
- Principal component analysis
- Factor analysis
- Cluster analysis
- Exercise: Winning the decathlon

3-D Scatter Plots



Often, data is presented in three variables instead of two. Creating a scatter plot for trivariate data requires the `scatter3` function. As with two-dimensional scatter plots, the scatter shows basic information about the distribution of the variables, and can display relationships among them.

The file `randudata.mat` contains 3000 consecutive "random" numbers from the random number generator used by IBM computers throughout the 1960s. A 3-D scatter plot quickly shows that the data are not, in some intuitive sense, "random." A clear pattern is visible. (We discuss definitions of "random" and the number generators used by MATLAB and the Statistics Toolbox in the next chapter.)

Try

```
>> load randudata
>> scatter3(list(1,:),list(2,:),list(3,:),'+')
```

Choose the Rotate 3-D tool at the top of the figure, then click and drag on the axes to rotate the plot. What patterns do you see?

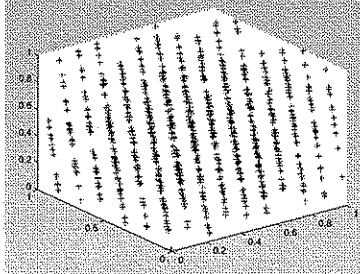
We may also use 3-D scatter plots to mine for additional information about the Standard Aptitude Test (SAT), introduced in the last chapter in the discussion of 2-D scatter plots (p. 4-5). We now plot the math and verbal SAT scores using a third coordinate: the percentage of students taking the test.

```
>> load satdata
>> scatter3(m,v,pct,'+')
>> grid off, box on, rotate3d on
>> xlabel('Math SAT scores')
>> ylabel('Verbal SAT scores')
>> zlabel('Percentage taking test')
```

Rotate the plot. The scatter indicates that the states that fared the best on the SAT were also the states where the fewest students took the test, indicating that other measures are needed before deciding which states truly fared well.

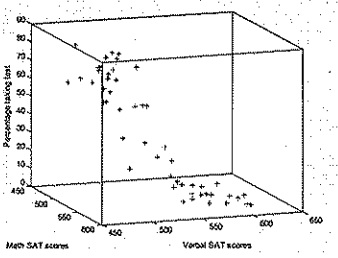
Try

```
>> load randudata
>> scatter3( ...
list(1,:), ...
list(2,:), ...
list(3,:),'+')
```



Also try

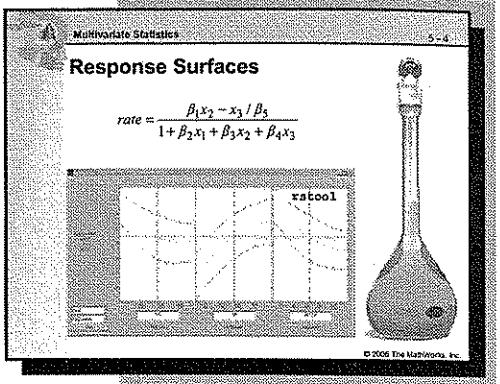
```
>> load satdata
>> scatter3( ...
m,v,pct,'+')
>> grid off
>> box on
>> rotate3d on
```



There is also a 3-D version of `gscatter`:

```
>> doc gscatter3
```

Response Surfaces



An example of a multidimensional nonlinear model is the Hougen-Watson model for chemical reaction kinetics. The form of the model is given by

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

We see that there are three predictor variables, x_1, x_2, and x_3, and one response variable, *rate*. We also see that the parameters β_1, β_2, β_3, β_4, and β_5 enter the model nonlinearly.

Given the 4-D nature of the model, we cannot view reaction data with `scatter3` unless we hold one of the inputs constant. The complete data set takes the geometric form of a 4-D scatter around an interpolating least squares *response surface*, which we are only able to view in 2-D or 3-D slices.

The Statistics Toolbox function `rstool` is useful for fitting response surface models to multidimensional data. The GUI provides an interactive environment for exploring multidimensional data and the response surfaces that result from a multiple regression using a linear additive model. (For further information on multiple linear regression, see the discussion in the toolbox documentation.)

The file `reaction.mat` contains reaction rate data in which the predictors x_1, x_2, and x_3 represent partial pressures of three reactants: hydrogen, n-pentane, and isopentane. Try

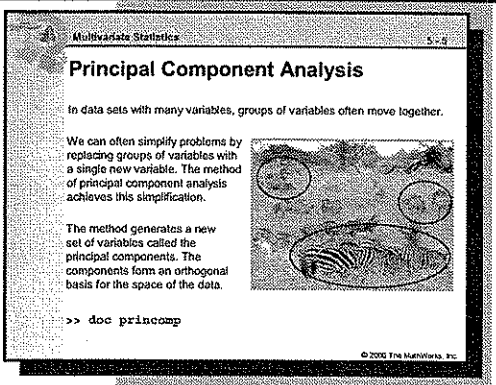
```
>> load reaction
>> rstool( ...
reactants,rate,'quadratic',0.01,xn,yn)
```

You will see a "vector" of three plots. The response variable in all three plots is the reaction rate. The first plot has hydrogen as the predictor 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.

Try

```
>> load reaction

>> edit hougen
>> constH = ...
repmat( ...
reactants(1,1),13,1);
>> constHreact = ...
[constH, ...
reactants(:,2),
reactants(:,3)];
>> constHrates = ...
hougen( ...
beta,constHreact);
>> scatter3( ...
reactants(:,2),
reactants(:,3),
constHrates)

>> rstool( ...
reactants,rate, ...
'quadratic', ...
0.01,xn,yn)
```

Principal Component Analysis



In data sets with many variables, groups of variables often move together. One reason for this is that more than one variable may be measuring the same driving force in the system. In many systems there are only a few such driving forces. Nevertheless, an abundance of measurement methods may allow us to create many system variables. When this happens, we can often take advantage of the redundancy in the information.

We attempt to simplify such problems by replacing groups of variables with single variables. *Principal component analysis* is a method for achieving this simplification. The method generates a new set of variables, called the principal components. Each principal component is a linear combination of the original variables. All of 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.

The principal component basis is formed by starting with a single axis in data space. Each datum is projected on the axis to form the fist component. The axis is chosen to make the variance of this new variable as large as possible. Subsequent steps project the data onto new axes, orthogonal to the others. The axes are always chosen to maximize the variance of the new component. The full set of components is as large as the original set of variables, but it is common for the sum of the variances of the first few components to exceed 80% of the total variance of the original data. By examining just these few variables, we may develop a good overall understanding of the driving forces behind the original data.

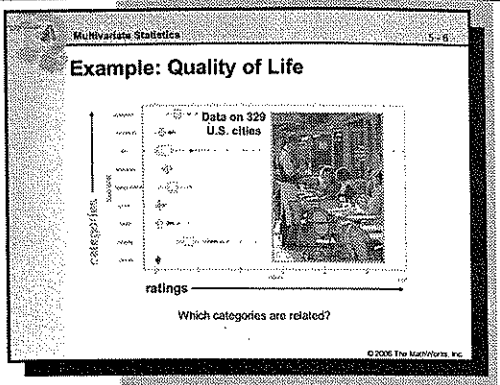
```
>> [PC,Z,vars,T2] = princomp(X)
```

takes a data matrix `X` and returns the principal components in `PC`, the Z-scores in `Z`, the eigenvalues of the covariance matrix of `X` in `vars` , and T2 statistics in `T2`. The Z-scores are the data formed by transforming the original data into the space of the principal components. The values of the vector `vars` are the variance of the columns of `Z`. `T2` is a measure of the multivariate distance of each observation from the center of the data set.

Try

```
>> doc princomp
```

Example: Quality of Life



Let us look at a sample principal component analysis that uses nine different indices of the quality of life in 329 U.S. cities:

- Climate
- Housing
- Health
- Crime
- Transportation
- Education
- Arts
- Recreation
- Economics

For each index, higher is better; so, for example, a higher index for crime means a lower crime rate.

Type

```
>> load cities
```

The cities data set contains three variables:

- `categories` % String matrix containing the names of the indices.
- `names` % String matrix containing the 329 city names.
- `ratings` % Data matrix with 329 rows and 9 columns.

Ordinarily we might graph pairs of the variables, but here there are 36 two-variable plots. Perhaps principal component analysis can reduce the number of variables we need to consider.

Try

```
>> load cities

>> categories

>> first5 = ...
names(1:5,:)

>> boxplot( ...
ratings,0,'+',0)
>> set(gca,...
'YTicklabel', ...
categories)
```

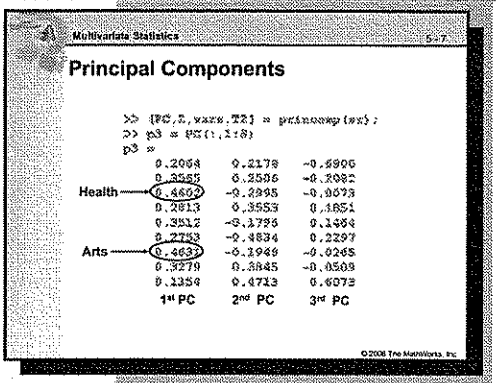
Normalize the data:

```
>> stdr = ...
std(ratings);
>> sr = ... ratings./
...
repmat(stdr,329,1);
```

Find the components:

```
>> [PC,Z,vars,T2] ...
= princomp(sr);
```

Principal Components



The first output of the `princomp` function, `PC`, contains the nine principal components. These are the linear combinations of the nine original variables.

Look at the first three principal components:

```
>> p3 = PC(:,1:3)
```

The largest weights in the first column (first principal component) are the third and seventh elements, corresponding to the variables health and arts. All the elements of the first principal component are the same sign, making it a weighted average of all the variables.

To see that the principal components are orthogonal, note that left multiplication by their transpose yields the identity matrix:

```
>> I = p3'*p3
```

Try

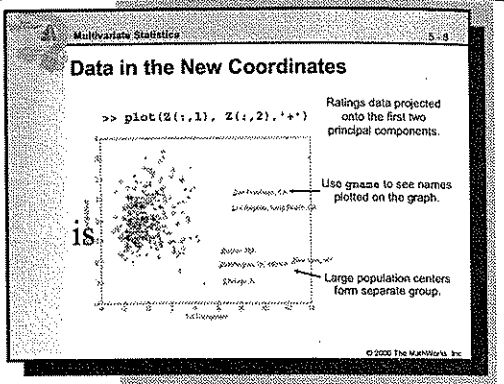
First three components:

```
>> p3 = PC(:,1:3)
```

Components are orthogonal:

```
>> I = p3'*p3
```

Data in the New Coordinates



The second output, `Z`, is the data in the new coordinate system defined by the principal components. This output the same size as the input data matrix.

A plot of the first two columns of `Z` shows the ratings data projected onto the first two principal components:

```
>> plot(Z(:,1), Z(:,2),'+')
>> xlabel('1st Component');
>> ylabel('2nd Component');
```

Note the outlying points to the right of the plot.

The function `gname` is useful for graphically identifying a few points in a plot like this. (`gname` was discussed on p. 2-15.) Try:

```
>> gname(names)
```

The `gname` labels show that the outlying cities to the right of the plot are the biggest population centers in the United States. Perhaps we should consider them as a completely separate group.

If we call `gname` without arguments,

```
>> gname
```

it labels each point with its row number. We can create an index variable containing the row numbers of all the metropolitan areas:

```
>> metro = [43 65 179 213 234 270 314];
```

We can then remove these rows from the ratings matrix.

To practice, repeat the analysis using the revised ratings variable as the new data matrix and the revised names variable as the labels.

Try

```
>> plot( ...
Z(:,1),Z(:,2),'+')
>> xlabel( ...   '1st
Component');
>> ylabel( ...
'2nd Component');

>> gname(names);
```

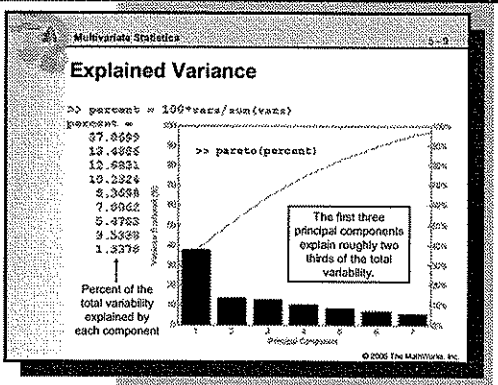
... then click on points in the plot. Press `Enter` when you have finished.

```
>> gname;

>> metro = ...     [43
65 179 213 ... 234
270 314];
>> names(metro,:)
```

Remove metro data:

```
>> rsubset = ...
ratings;
>> nsubset = ...
names;
>>> nsubset(metro,:)
... = [];
>> rsubset(metro,:)
... = [];
>>> size(rsubset)
```

Explained Variance



The third output, `vars`, is a vector containing the part of the variance explained by the corresponding column of `Z`:

```
>> vars
```

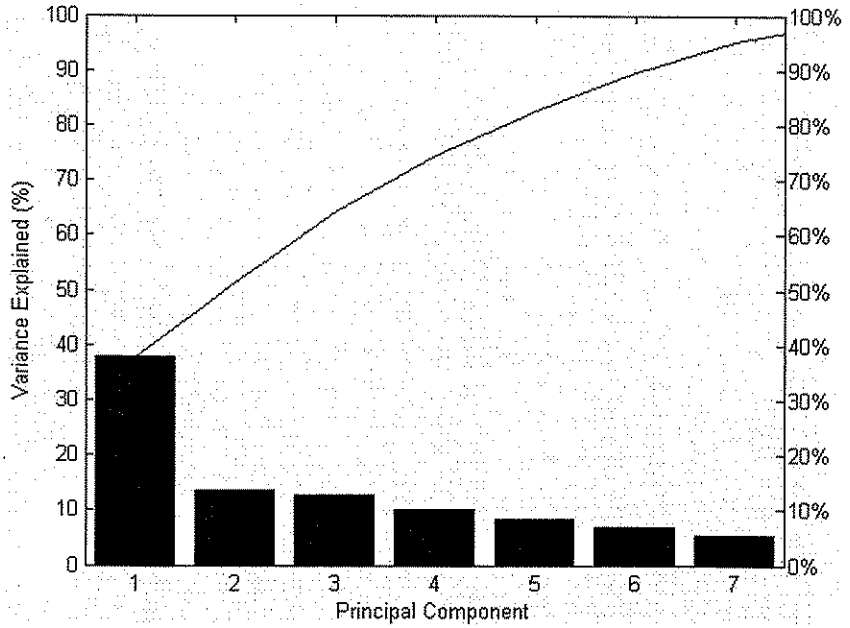
You can easily calculate the percent of the total variability explained by each principal component:

```
>> percent = 100*vars/sum(vars)
```

A "Scree" plot is a *pareto chart* of the percent variability explained by each principal component:

```
>> pareto(percent)
>> xlabel('Principal Component')
>> ylabel('Variance Explained (%)')
```

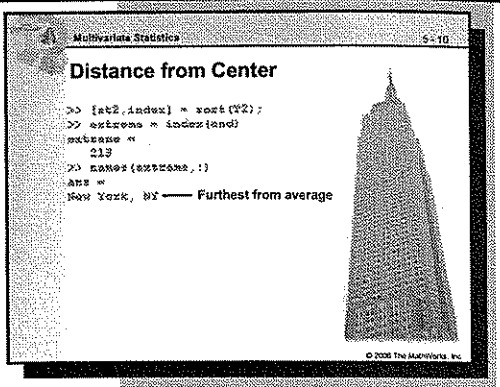
We can see that the first three principal components explain roughly two thirds of the total variability in the standardized ratings.



Try

```
>> vars

>> percent = ...
100*vars/sum(vars)

>> doc pareto
>> pareto(percent)
>> xlabel( ...
'Principal ...
Component')
>> ylabel( ...
'Variance ...
Explained (%)')
```

Distance from Center



The last output of the `princomp` function, `T2`, is Hotelling's T2 statistic, a measure of the multivariate distance of each observation from the center of the data set. This is an analytic way to find the most extreme points in the data.

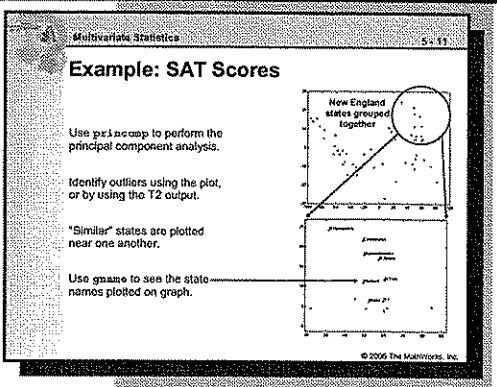
```
>> [st2, index] = sort(T2); % Sort in ascending order.
>> extreme = index(end)
>> names(extreme,:)
```

It is perhaps not surprising that the ratings for New York City are the furthest from the average U.S. town.

Try

```
>> [st2,index] = ...
sort(T2);
>> extreme = ...
index(end)
>> names(extreme,:)
```

Example: SAT Scores



The script file `satpcascript.m` contains code that performs a principal component analysis on the SAT data we considered earlier (pp. 4-5, 5-3). After the initial analysis, the T2 output is used to calculate outliers, and `gname` is used to click on points in the plot to see the states being plotted.

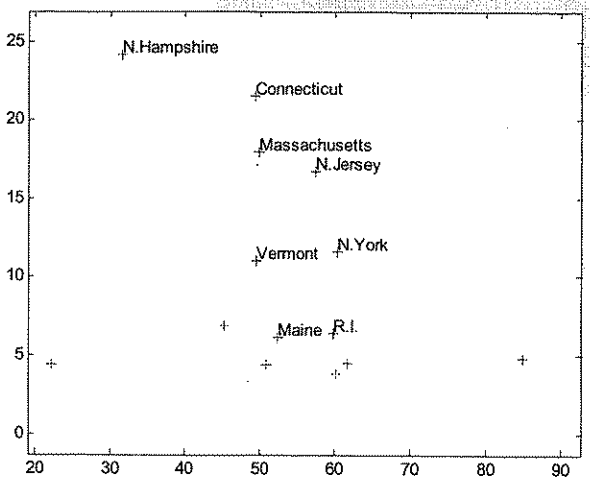
```
% Script for Principal Component Analysis

clear all
load satdata

% Build matrix to pass to PRINCOMP.
mat = [m v pct];
[pcs,newdata,vars,t2] = princomp(mat);

% Plot new data, according to principal components.
plot(newdata(:,1),newdata(:,2),'+')

% Get list of states with most unusual results.
[t2sorted,ix] = sort(t2);
outix = ix(end-4:end);
outstates = names(outix,:)

% Use gname to plot names on component list.
% Try the upper right corner. Hit RETURN when done.

gname(names)
```

Try

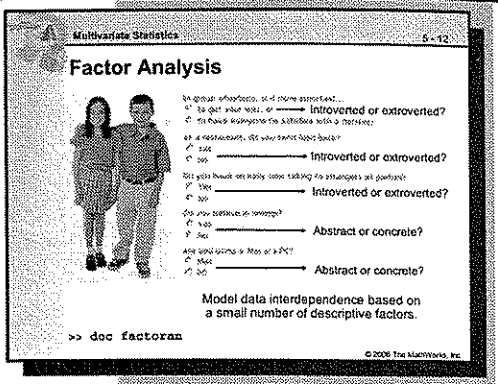
```
>> satpcascript
```

Using `gname` is one way to find outliers, but it is also used to find states with very similar data.

For example, at right is the scatter plot after PCA, zoomed into the upper-right corner. The selected states in this corner are almost all New England states, with similar SAT scores and student participation levels.



Factor Analysis



Multivariate data often includes a large number of measured variables, and sometimes those variables overlap, in the sense that groups of them may be dependent. For example, in a decathlon, each athlete competes in 10 events, but several of them can be thought of as "speed" events, others as "strength" events, etc. Thus, you can think of a competitor's 10 event scores as largely dependent on a smaller set of 3 or 4 types of athletic ability. *Factor analysis* is a way to fit a model to multivariate data to estimate just this sort of interdependence.

Try

```
>> doc factoran
```

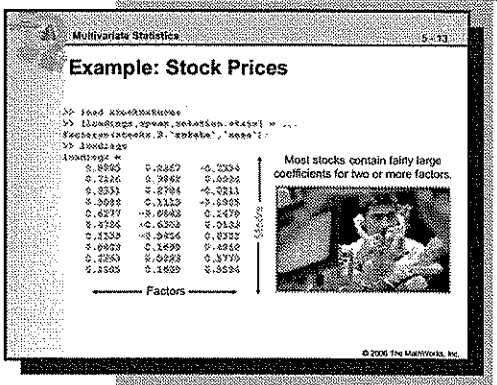
A common example of factor analysis is a personality test. Sixty questions are asked, and those questions are distilled to four or five factors based on the given answers. A person is then categorized as "introverted" or "extroverted", "abstract" or "concrete", and so forth. These tests are often based on statistical measures that guarantee the significance of the chosen factors.

Both factor analysis and principal component analysis 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 data correlations.

In the factor analysis model, the measured variables depend on a smaller number of unobserved *latent factors*. Because each may affect several variables in common, these factors are also 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.

The Statistics Toolbox uses the `factoran` command to perform factor analyses.

Example: Stock Prices



Over the course of 100 weeks, the percent change in stock prices for ten companies is recorded. Of the 10 companies, the first four can be classified as primarily "technology," the next three as "financial," the last three as "retail." It seems reasonable that 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 price.

```
>> load stockreturns
```

loads the variable `stocks` with 100 rows of weekly data and 10 columns representing the different companies.

```
>> [loadings,spvar,rotation,stats] = ...
factoran(stocks,3,'rotate','none');
```

specifies a model fit with three common factors. By default, `factoran` computes rotated estimates of the loadings to try and make their interpretation simpler. We temporarily override the default.

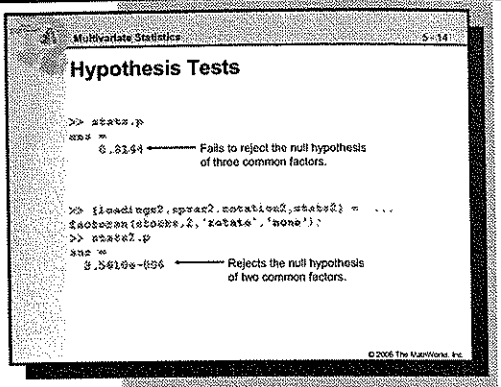
The first two outputs are the estimated loadings and the estimated specific variances. Each row of the `loadings` matrix represents one of the 10 stocks, and each column corresponds to a common factor. With unrotated estimates, interpretation of the factors in this fit is difficult because most of the stocks contain fairly large coefficients for two or more factors. *Factor rotation* (discussed next) helps to simplify the structure in the `loadings` matrix, so that it will be easier to assign meaningful interpretations to the factors.

From the estimated specific variances `spvars`, we can see that the model indicates that a particular stock price varies quite a lot beyond the variation due to the common factors. 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. The data fall in between.

Try

```
>> load stockreturns

>> [loadings, ...
spvar, ...
rotation, ... stats]
= ... factoran( ...
stocks,3, ...
'rotate','none');

>> loadings
>> spvar
>> rotation
```

Hypothesis Tests



The fourth output returned by `factoran`, `stats`, is a structure containing information about the null hypothesis that the number of common factors is equal to the number of common factors specified for the fitted model. (We discuss hypothesis testing generally in Chapter 7.)

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 covariances in the data:

```
>> stats.p
ans =
   0.8144
```

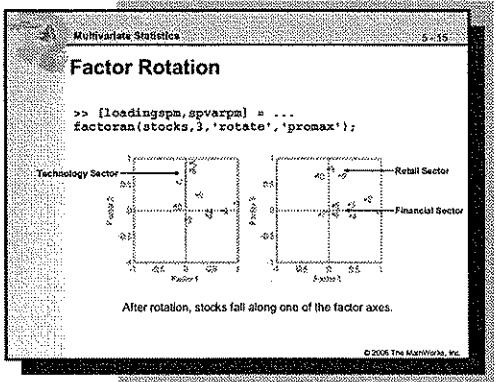
To determine if fewer than three factors can provide an acceptable fit, you can try a model with two common factors. The *p*-value for this second fit is highly significant, and rejects the hypothesis of two factors, indicating that the simpler model is not sufficient to explain the pattern in these data:

```
>> [loadings2,spvar2,rotation2,stats2] =  ...
factoran(stocks,2,'rotate','none');
>> stats2.p
ans =
   3.5610e-006
```

Try

```
>> stats.p

>> [loadings2, ...
spvar2, ...
rotation2,... stats2]
=  ... factoran( ...
stocks,2, ...
'rotate','none');
>> stats2.p
```

Factor Rotation



As the results illustrate, the estimated loads 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, i.e., 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 example, here we rotate the estimated loadings by using the `promax` criterion, a common oblique method:

```
>> [loadingspm,spvarpm] = ...
factoran(stocks,3,'rotate','promax');
```

Promax rotation has created a simpler structure in the loadings, one in which most of the stocks have a large load on only one factor.

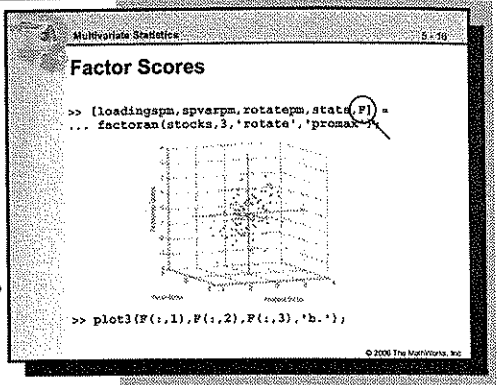
To see this more clearly, you can plot each stock using the factor loadings as coordinates. "Simple" structure in the loadings appears in these plots as stocks that fall along one of the factor axes. Because there are three factors, it is easier to see the loadings if you make a pair of two-dimensional plots. The script `factplot.m` creates these plots.

The plots show that `promax` has succeeded in rotating 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.

Try

```
>> [loadingspm, ...
spvarpm] = ...
factoran( ...
stocks,3, ...
'rotate','promax');

>> edit factplot
>> factplot
```

Factor Scores



Sometimes it is useful to be able to classify an observation based on its *factor scores*. For example, if you accepted the three-factor model and the interpretation of the rotated factors, you might want to categorize each week in terms of how favorable it was for each of the three stock sectors, based on the data from the 10 observed stocks.

Since the data in this example are the raw stock price changes, and not their correlation matrix, you can have `factoran` return estimates–the factor scores–of the value of each of the three rotated common factors for each week. The scores are returned in a fifth output from `factoran`. You can then plot the estimated scores to see how the different stock sectors were affected during each week. The script `factplot2.m` carries this out.

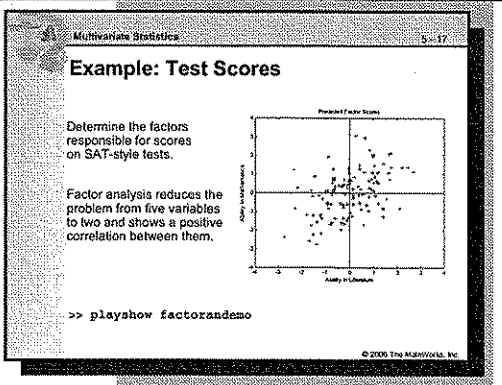
Oblique rotation often creates factors that are correlated. The plot shows some evidence of correlation between the first and third factors, and you can investigate further by computing the estimated factor correlation matrix:

```
>> inv(rotationpm'*rotationpm)
```

Try

```
>> edit factplot2
>> factplot2
```

Example: Test Scores



The SAT data we looked at previously (pp. 4-5, 5-3, 5-11) does not have enough variables to perform a factor analysis. Five variables are necessary to generate more than one factor.

The file `examgrades.mat`, however, contains grades from five different tests. The first two cover mathematics, the second two cover literature, and the fifth is a comprehensive exam. We can use factor analysis to determine whether there is a single common factor that can explain the scores:

```
>> load examgrades
>> [loadings,spvar,rotation,stats] = ...
factoran(grades,1);
>> stats.p
```

The *p*-value (3.32 %) is too small to be acceptable, so we try again with two factors:

```
>> [loadings,spvar,rotation,stats] = ...
factoran(grades,2);
>> stats.p
```

This *p*-value (70.61 %) is large enough to be acceptable. Therefore, it is reasonable to expect that two common factors can explain the covariance found in the exam grades.

Also try:

```
>> [loadings,spvar,rotation,stats,F] = ...
factoran(grades,2,'rotate','promax', ...
'maxit',200);
>> scatter(F(:,1),F(:,2),'+');
```

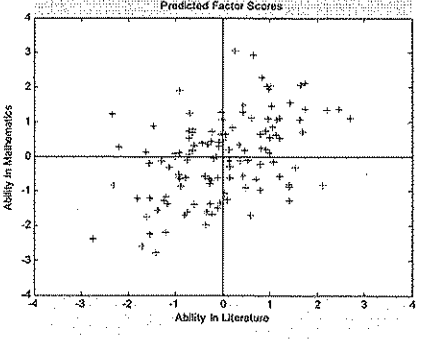
The plot shows a positive correlation between students' ability in mathematics and their ability in literature.

For a detailed demo of the factor analysis in this example, type

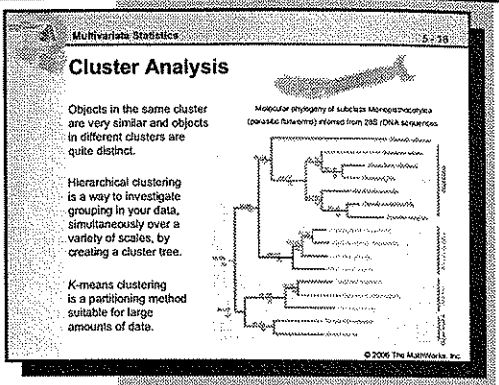
```
>> playshow factorandemo
```

Try

```
>> playshow ...
factorandemo
```



Cluster Analysis



Cluster analysis, also called *segmentation analysis* or *taxonomy analysis*, is a way to 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, data might contain a number of observations of subjects in a study where each observation is a set of variables.

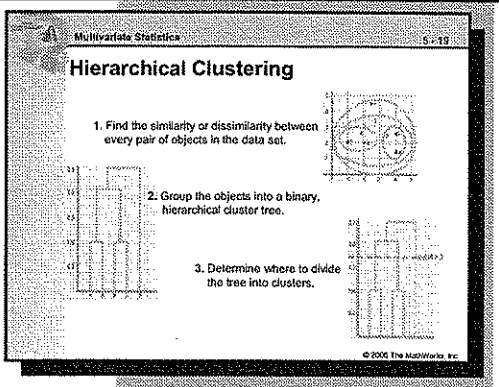
Cluster analysis can help in creating balanced treatment and control groups for a designed study. If you find that each cluster contains roughly equal numbers of treatment and control subjects, then statistical differences found between the groups can be attributed to the experiment and not to any initial difference between the groups.

Two types of cluster analysis are available in the Statistics Toolbox, *hierarchical clustering* and *K-means 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 Statistics Toolbox includes a function, `clusterdata`, which performs all the necessary steps for you. It incorporates the `pdist`, `linkage`, and `cluster` functions, which may be used separately.

K-means clustering can best be described as a partitioning method. The function `kmeans` partitions observations in your data into *K* mutually exclusive clusters, and returns a vector of indices indicating to which of the clusters it has assigned each observation. Unlike the hierarchical clustering methods, `kmeans` does not create a tree structure to describe the groupings in the 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 the data, and not just their proximities. These differences often mean that `kmeans` is more suitable for clustering large amounts of data.

Hierarchical Clustering



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 metrics to compute this measurement.

2. **Group the objects into a binary, hierarchical cluster tree.**

   In this step, you link together 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.

3. **Determine where to divide the hierarchical tree into clusters.**

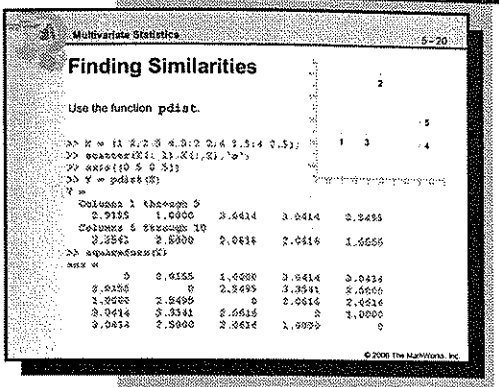
   In this step, you divide the objects in the hierarchical tree into clusters using the `cluster` function. The `cluster` function can create clusters by detecting natural groupings in the hierarchical tree or by cutting off the tree at an arbitrary point.

The Statistics Toolbox also 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. However, the `clusterdata` function does not give you access to the options each of the individual routines offers. For example, if you use the `pdist` function you can choose the metric, whereas if you use the `clusterdata` function you cannot.

Try

```
>> doc pdist
>> doc linkage
>> doc cluster
>> doc clusterdata
```

Finding Similarities



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 $m*(m-1)/2$ pairs in the data set. The result of this computation is commonly known as a *distance matrix* or *dissimilarity matrix*.

There are many metrics to calculate this distance information. By default, the `pdist` function calculates the Euclidean distance between objects. You can, however, specify several other options.

You can, optionally, normalize the values in the data set before calculating the distance information. In many data sets, variables may 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 scale.

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

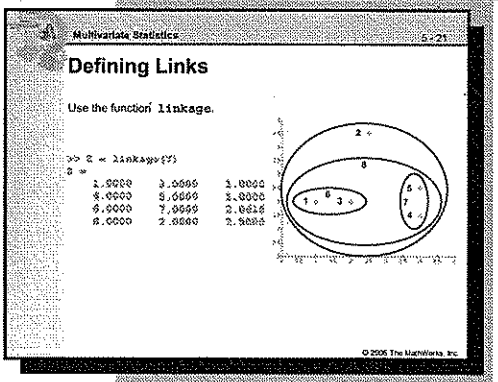
```
>> X = [1 2;2.5 4.5;2 2;4 1.5;4 2.5];
```

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.

Try

```
>> X = ...
[1 2;2.5 4.5; ...
2;4 1.5;4 2.5];
>> scatter( ...
X(:,1),X(:,2),'o')
>> axis([0 5 0 5])

>> Y = pdist(X)
>> squareform(Y)
```

Defining Links



Once the proximity between objects in the data set has been computed, you can determine which objects in the data set should be grouped together 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 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 `X` of x and y coordinates, the `linkage` function generates a hierarchical cluster tree, returning the linkage information in a matrix `Z`:

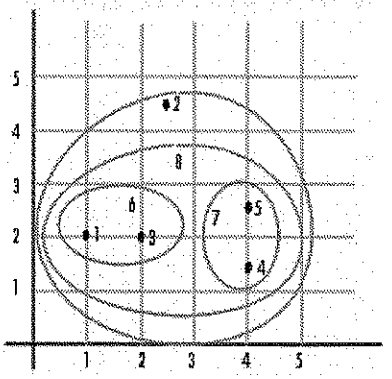
```
>> Z = linkage(Y)
```

In the output, each row identifies a link. The first two columns identify the objects that have been linked, e.g, object 1, object 2. The third column contains the distance between these objects. For the sample data set `X`, the `linkage` function begins by grouping together 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.

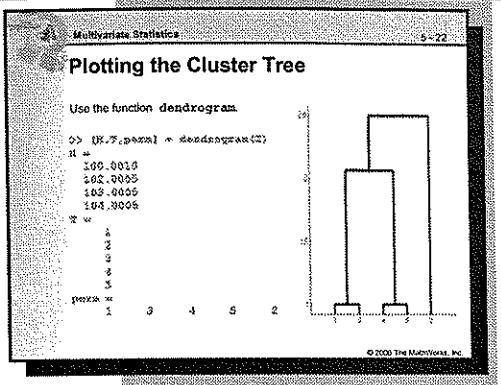
The third row indicates that the `linkage` function grouped together objects 6 and 7. If our original sample data set contained only five objects, what are objects 6 and 7? Object 6 is the newly formed binary cluster created by the grouping of objects 1 and 3. When the linkage function groups two objects together 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. Object 7 is the index for the cluster formed by objects 4 and 5. As a 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.

Try

```
>> Z = linkage(Y)
```

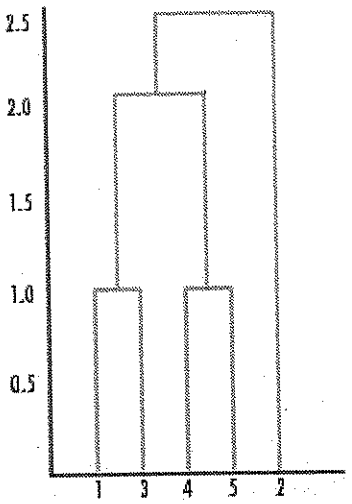


Plotting the Cluster Tree



The hierarchical, binary cluster tree created by the `linkage` function is most easily understood when viewed graphically. The Statistics Toolbox includes the `dendrogram` function to plot this hierarchical tree information as a graph.

```
>> dendrogram(Z)
```



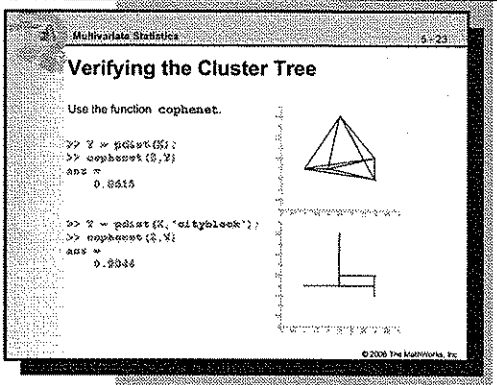
In the figure, the numbers along the horizontal axis represent the indices of the objects in the original data set. The links between objects are represented as upside down U-shaped lines. The height of the U indicates the distance between the objects. For example, the link representing the cluster with objects 1 and 3 has a height of 1.

Try

```
>> dendrogram(Z)
```

```
>> doc dendrogram
```

Verifying the Cluster Tree



After linking the objects in a data set into a hierarchical cluster tree, you may want to verify that the tree represents significant similarity groupings.

One way to measure the validity of the cluster information generated by the linkage function is to compare it with the original proximity 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 better the clustering solution.

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)
```

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, use the `cophenet` function to evaluate the clustering using a different distance metric:

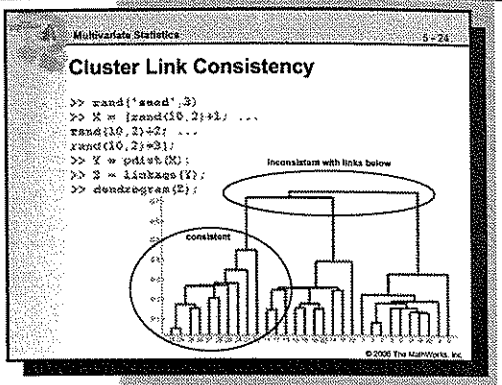
```
>> Y = pdist(X,'cityblock'))
>> c = cophenet(Z,Y)
```

The cophenetic correlation coefficient shows a stronger correlation when the City Block metric is used.

Try

```
>> c = ...
cophenet(Z,Y)

>> Y = ...     pdist(
... X,'cityblock')
>> c = ...
cophenet(Z,Y)
```

Cluster Link Consistency



In addition to verifying that the tree represents significant similarity groupings, you may want more information about the links between the objects.

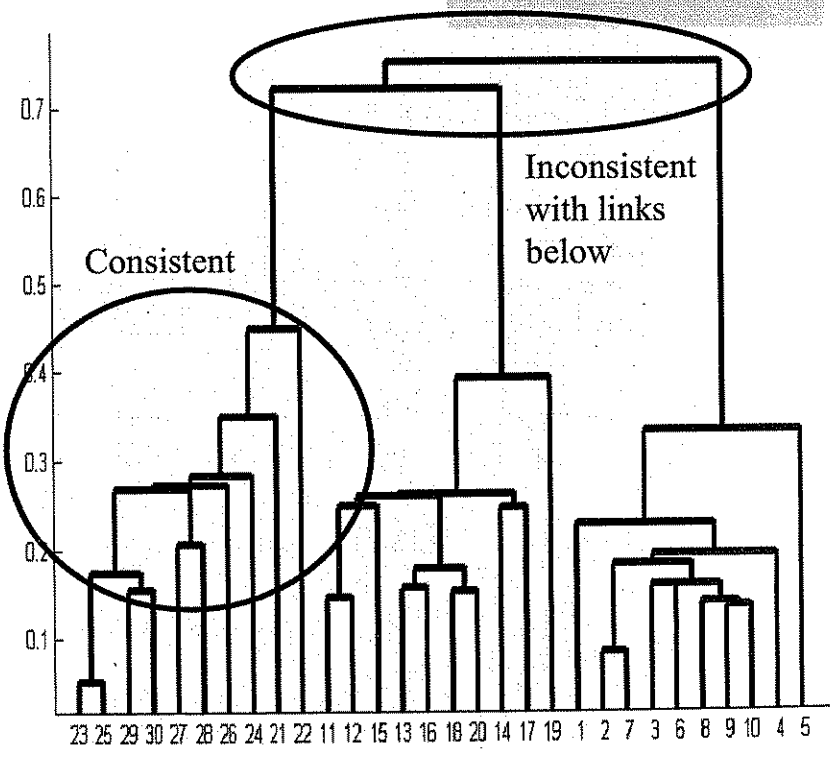
One way to determine the natural cluster divisions in a data set is to compare the height of each link in a cluster tree with the heights of neighboring links below it in the tree. If a link is approximately the same height as neighboring links, it indicates that there are similarities between the objects joined at this level of the hierarchy. These links are said to exhibit a high level of *consistency*. If the height of a link differs from neighboring links, it indicates that there are dissimilarities between the objects at this level in the cluster tree. This link is said to be *inconsistent* with the links around it. In cluster analysis, inconsistent links can indicate the border of a natural division in a data set. The cluster function uses a measure of inconsistency to determine where to divide a data set into clusters.

Try

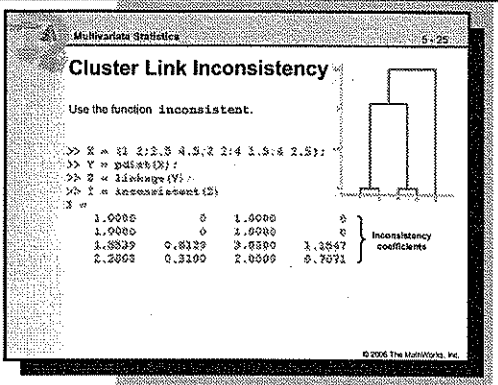
```
>> rand('seed',3)
>> X = ...
[rand(10,2)+1; ...
rand(10,2)+2; ...
rand(10,2)+3];
>> Y = pdist(X);
>> Z = linkage(Y);
>> dendrogram(Z);
```

To illustrate, the following example creates a data set of random numbers with three deliberate natural groups. In the dendrogram, note how objects tend to collect into three groups. These three groups are connected by three longer links. These longer links are inconsistent when compared with the links below them :

```
>> rand('seed',3)
>> X = [rand(10,2)+1; ...
rand(10,2)+2; ...
rand(10,2)+3];
>> Y = pdist(X);
>> Z = linkage(Y);
>> dendrogram(Z);
```



Cluster Link Inconsistency



The relative consistency of each link in a hierarchical cluster tree can be quantified using the *inconsistency coefficient*. This value compares the height of a link in a cluster hierarchy with the average height of neighboring links. If the object is consistent with those around it, it has a low inconsistency coefficient. If the object is inconsistent with those around it, it will have a higher inconsistency coefficient.

To generate a listing of the inconsistency coefficient for each link the cluster tree, use the `inconsistent` function. The inconsistent function compares each link in the cluster hierarchy with adjacent links two levels below it in the cluster hierarchy. This is called the *depth* of the comparison. Using the inconsistent function, you can 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.

For example, returning to the sample data set of coordinates, we can use the `inconsistent` function to calculate the inconsistency values for the links created by the `linkage` function:

```
>> I = inconsistent(Z)
```

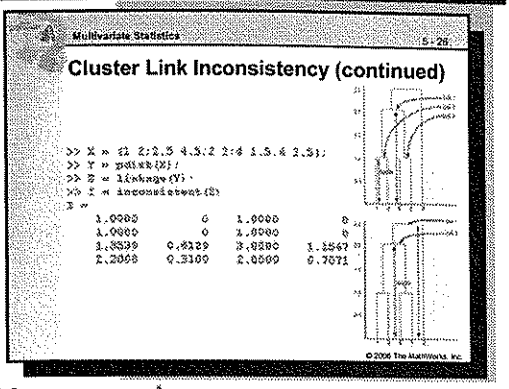
The `inconsistent` function returns data about the links in an (m-1) x 4 matrix where each column provides data about the links:

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

Try

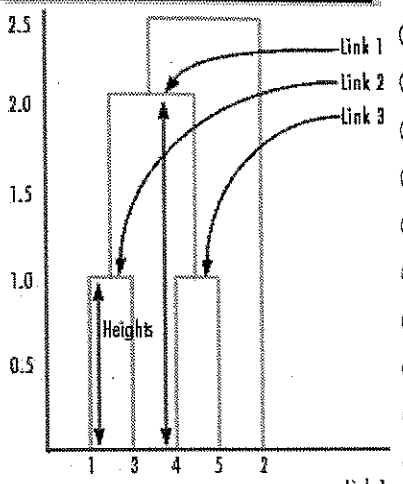
```
>> X = ...
[1 2;2.5 4.5; ...    2
2;4 1.5;4 2.5];
>> Y = pdist(X);
>> Z = linkage(Y);

>> I = ...
inconsistent(Z)
```

Cluster Link Inconsistency (continued)

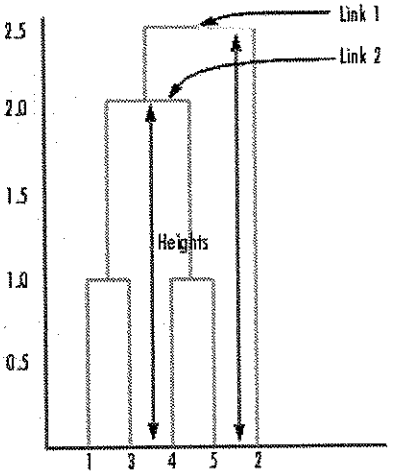


In the sample output `I`, the first row represents the link between objects 1 and 3. (This cluster is assigned the index 6 by the `linkage` function.) Because this a leaf node, the inconsistency coefficient is zero. The second row represents the link between objects 4 and 5, also a leaf node. (This cluster is assigned the index 7 by the `linkage` function.)

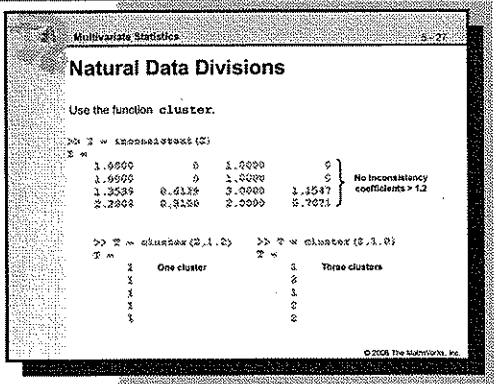
The third row evaluates the link that connects these two leaf nodes, objects 6 and 7. (This cluster is called object 8 in the `linkage` output). Column three indicates that three links are considered in the calculation: the link itself and the two links directly below it in the hierarchy. Column one 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 two represents the standard deviation between the links. The last column contains the inconsistency value for these links, 0.8165. The figure at right illustrates the links and heights included in this calculation.



Row four in the output matrix describes the link between object 8 and object 2. Column three 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 figure at right illustrates the links and heights in this calculation.



Natural Data Divisions



In the hierarchical cluster tree, the data set may naturally align itself into clusters. This can be particularly evident in a dendrogram 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 points where the similarities between objects change. You can use this to determine where the cluster function draws 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,1.2)
```

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 1.0, the `cluster` function divides the sample data set into three clusters:

```
>> T = cluster(Z,1.0)
```

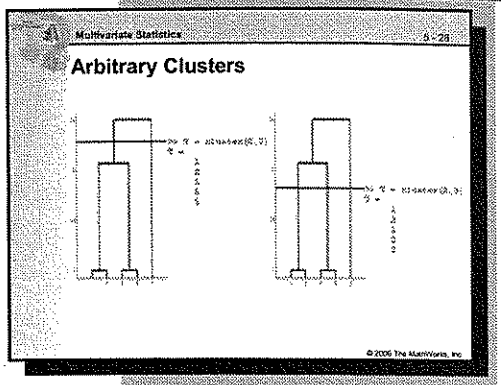
The output indicates that objects 1 and 3 were placed in cluster 1, objects 4 and 5 in cluster 2, and object 2 in cluster 3.

Try

```
>> T = ...
cluster(Z,1.2)
```

```
>> T = ...
cluster(Z,1.0)
```

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. In this case, the value of the cutoff argument specifies the point in the cluster hierarchy at which to create the clusters.

For example, you can specify that you want the `cluster` function to divide 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,2)
```

To help you visualize how the `cluster` function determines how to create these clusters, the figure at right shows the dendrogram of the hierarchical cluster tree. When you specify a value of 2, the `cluster` function draws an imaginary horizontal line across the dendrogram that bisects two vertical lines. All the objects below the line belong to one of these two clusters.

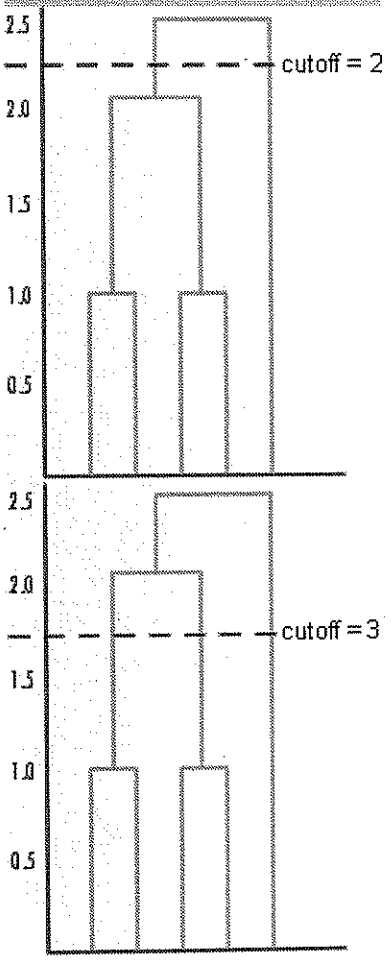
If you specify a cutoff value of 3, the `cluster` function cuts off the hierarchy at a lower point, bisecting three lines:

```
>> T = cluster(Z,3)
```

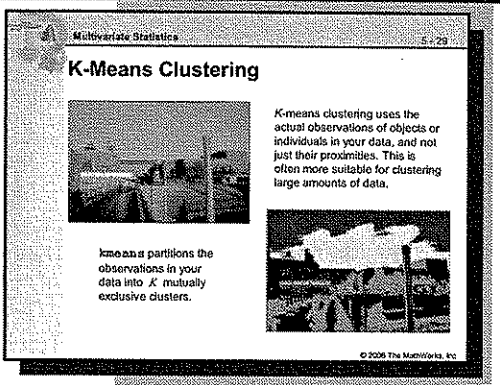
This time, objects 1 and 3 are grouped in a cluster, objects 4 and 5 are grouped in a cluster, and object 2 is placed into a cluster, as seen in the figure at right.

Try

```
>> T = cluster(Z,2)

>> T = cluster(Z,3)
```



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`, `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 metrics, depending on the kind of data you are clustering.

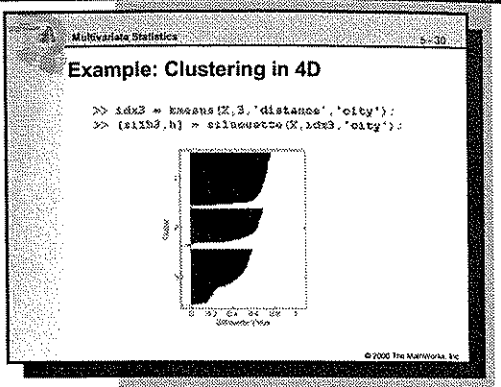
Each cluster in the partition is defined by its member objects and by its *centroid*, or center. The centroid for each cluster is the point to which the sum of distances from all objects in that cluster is minimized. `kmeans` computes cluster centroids differently for each distance measure, to minimize 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.

Try

```
>> doc kmeans
>> doc silhouette
```

Example: Clustering in 4-D



Load some sample data:

```
>> load kmeansdata;
```

Even though these data are 4-D, and cannot be easily visualized, `kmeans` enables you to investigate if group structures exist. Call `kmeans` with K, the desired number of clusters, equal to 3. Specify the city block metric, and use the default starting method of initializing centroids from randomly selected data points:

```
>> idx3 = kmeans(X,3,'distance','city');
```

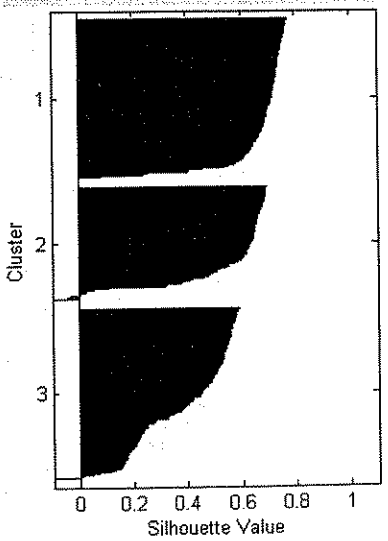
To get an idea of how well separated the resulting clusters are, you can make a *silhouette plot* using the cluster indices output from `kmeans`. The silhouette plot displays a measure of how close each point in one cluster is to points in the neighboring clusters. This measure ranges from +1, indicating 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. These values are returned in the first output:

```
>> [silh3,h] = silhouette(X,idx3,'city');
>> xlabel('Silhouette Value')
>> ylabel('Cluster')
```

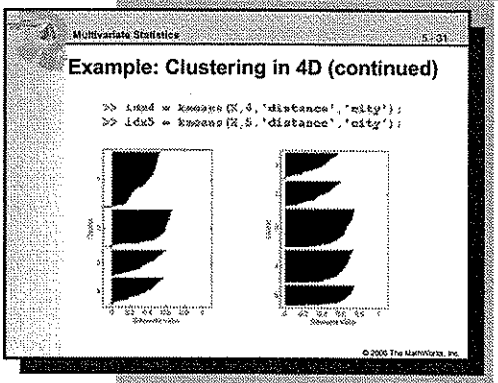
From the silhouette plot, you can see that most points in the third cluster have a large silhouette value, greater than 0.6, indicating that that cluster is somewhat separated from neighboring clusters. However, the second cluster contains many points with low silhouette values, and the first contains a few points with negative values, indicating that those two clusters are not well separated.

Try

```
>> load kmeansdata;

>> idx3 = ... kmeans(
...          X,3, ...
'distance','city');

>> [silh3,h] = ...
silhouette( ...
X,idx3,'city');
>> xlabel( ...
'Silhouette Value')
>> ylabel('Cluster')
```



Example: Clustering in 4-D (continued)



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');
```

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 may 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');
```

A more quantitative way to compare the two solutions is to look at the average silhouette values for the two cases:

```
>> mean(silh3)
>> mean(silh4)
```

It is a good idea to experiment with a range of values for K:

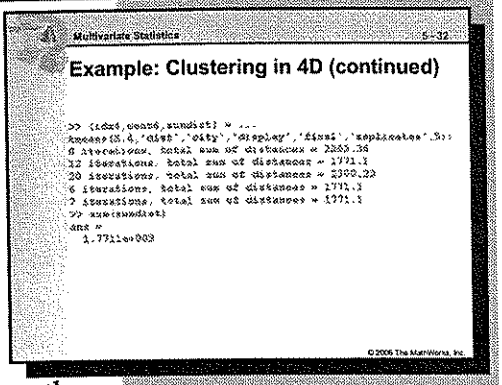
```
>> idx5 = ...
kmeans(X,5,'dist','city','replicates',5);
>> [silh5,h] = silhouette(X,idx5,'city');
>> mean(silh5)
```

This plot indicates that five is probably not the right number of clusters, since two clusters contain points with mostly low values.

Try

```
>> idx4 = ... kmeans(
...          X,4, ...
'dist','city', ...
'display','iter');

>> [silh4,h] = ...
silhouette( ...
X,idx4,'city');
>> xlabel( ...
'Silhouette Value')
>> ylabel('Cluster')

>> mean(silh3)
>> mean(silh4)

>> idx5 = ... kmeans(
...      X,5, ...
'dist','city', ...
'replicates',5);
>> [silh5,h] = ...
silhouette( ...
X,idx5,'city');
>> xlabel( ...
'Silhouette Value')
>> ylabel('Cluster')
>> mean(silh5)
```

Example: Clustering in 4-D (continued)



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);
```

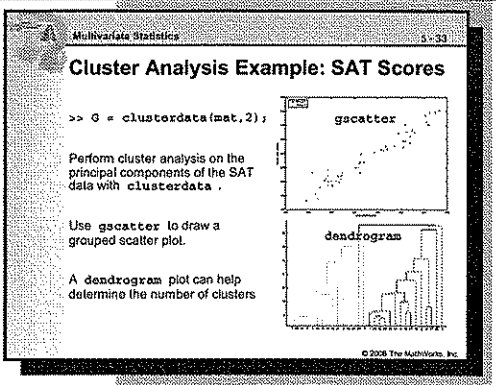
The output shows that, even for this relatively simple problem, nonglobal minima do exist. Each of these five replicates began from a different randomly selected set of initial centroids, and `kmeans` found two different local minima. However, the final solution that `kmeans` returns is the one with the lowest total sum of distances, over all replicates:

```
>> sum(sumdist)
```

Try

```
>> [idx4, ...
cent4, ...
sumdist] = ...
kmeans( ...
X,4, ...
'dist','city', ...
'display','final',...
'replicates',5);

>> sum(sumdist)
```

Cluster Analysis Example: SAT Scores



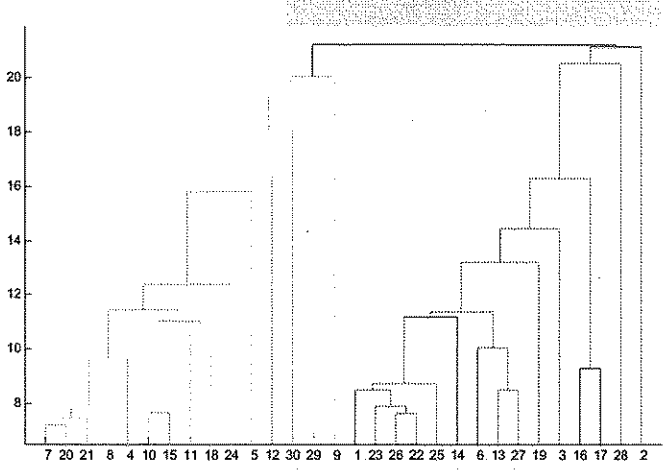
The file `satcluster.m` contains code for cluster analysis on the SAT data set. It divides the data into two clusters based on the results of principal component analysis.

```
% Script file for clustering with SAT data

clear all,close all
load satdata

% Perform principal component analysis
mat = [m v pct];
[pcs,newdata] = princomp(mat);

% Perform clustering
Y = pdist(newdata);
Z = linkage(Y);
G = cluster(Z,2);  % OR G = clusterdata(newdata,2);

% Grouped scatter plot based on clustering
gscatter(m,v,G,'br','+x')

% Use dendrogram to detect number of clusters
figure
dendrogram(Z,'colorthreshold',Z(end,end)-eps)
```

Try

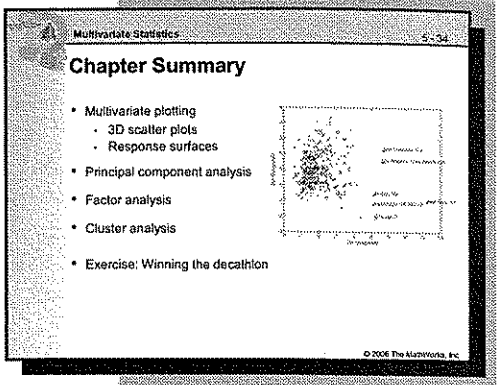
```
>> satcluster
```

For hierarchical clustering, a dendrogram plot can be used to determine the appropriate number of clusters. In this case, the dendrogram shows two main clusters.



Dendrograms and silhouette plots are good for exploratory statistics, in that they let the data determine the best number of clusters, rather than the user forcing a particular number of clusters on the data.

Chapter Summary



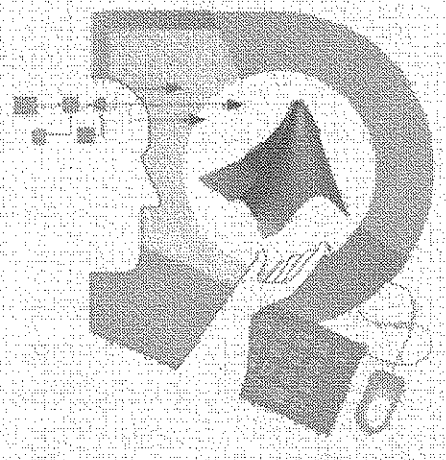
- Multivariate plotting
  - 3-D scatter plots
  - Response surfaces
- Principal component analysis
- Factor analysis
- Cluster analysis
- Exercise: Winning the decathlon

Exercise: Winning the Decathlon

In the next chapter we return to random number generation from probability and sampling distributions, investigating the nature of the different processes and their application in simulation.

Statistical Methods in MATLAB®

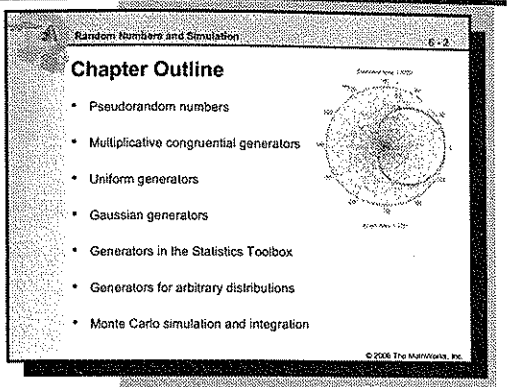
Random Numbers and Simulation



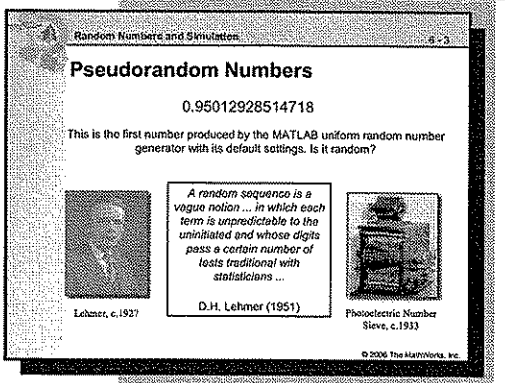
The MathWorks
Training Services

Chapter Outline

- Pseudorandom numbers
- Multiplicative congruential generators
- Uniform generators
- Gaussian generators
- Generators in the Statistics Toolbox
- Generators for arbitrary distributions
- Monte Carlo simulation and integration



Pseudorandom Numbers



Here is an interesting number:

0.95012928514718

This is the first number produced by the MATLAB uniform random number generator with its default settings. Start up a fresh MATLAB, set `format long`, type `rand`, and you get the number.

Try

```
>> ! matlab
>> format long
>> rand
```

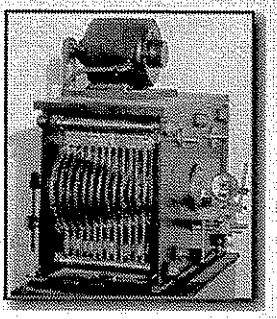
If all MATLAB users, all around the world, all on different computers, keep getting this same number, is it really "random?" No, it isn't. Computers are (in principle) deterministic machines and should not exhibit random behavior. If your computer doesn't access some external device, like a gamma ray counter or a clock, then it must really be computing *pseudorandom* numbers.

A working definition of randomness was given in 1951 by Berkeley Professor D. H. Lehmer, a pioneer in computing and, especially, computational number theory:

> *A random sequence is a vague notion ... in which each term is unpredictable to the uninitiated and whose digits pass a certain number of tests traditional with statisticians ...*

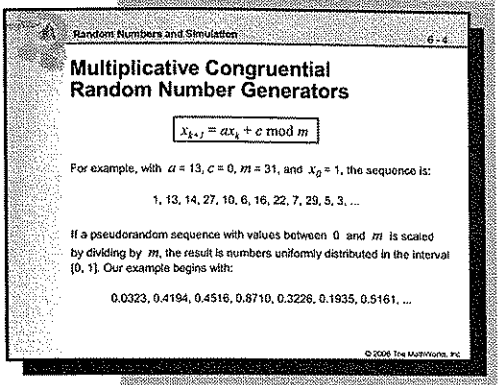
Lehmer, c. 1927

Precise mathematical definitions of randomness are modeled on notions of *information content*, *noncomputability*, and *stochasticity*, among other things. The various definitions of randomness do not always agree on which sequences are random and which are not, however. As a result, a standard definition of "random" has not emerged. This is in contrast to the notion of, for example, "computable", where all of the mathematical models that have been proposed are provably equivalent.



Photoelectric Number Sieve, c.1933

Multiplicative Congruential Random Number Generators



Lehmer invented the *multiplicative congruential algorithm*, which is the basis for many of the random number generators in use today.

Lehmer's generators involve three integer parameters, a, c, and m, and an initial value, x_0, called the *seed*. A sequence of integers is defined by

$$x_{k+1} = ax_k + c \bmod m.$$

The operation "mod m" means divide by m and keep the remainder. Numbers that are equal mod m are said to be *congruent*.

For example, with $a = 13$, $c = 0$, $m = 31$, and $x_0 = 1$, the sequence begins with

1, 13, 14, 27, 10, 6, 16, 22, 7, 29, 5, 3, ...

What's the next value? It looks pretty unpredictable, but you've been initiated. So you can compute 13*3 mod 31, which is 8.

The first 30 terms in the sequence are a permutation of the integers from 1 to 30 and then the sequence repeats itself. We say that the generator has a *period* of $m - 1$.

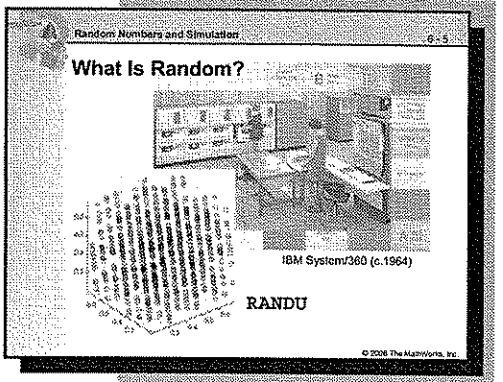
If a pseudorandom integer sequence with values between 0 and m is scaled by dividing by m, the result is floating-point numbers uniformly distributed in the interval [0, 1]. Our example above begins with

0.0323, 0.4194, 0.4516, 0.8710, 0.3226, 0.1935, 0.5161, ...

There are only a finite number of values, 30 in this case. The smallest value is 1/31; the largest is 30/31. Each one is equally probable in a long run of the sequence.

Exercise: Write a Random Number Generator

What Is Random?



In the 1960's the Scientific Subroutine Library on IBM System/360 computers included a random number generator named RND or RANDU. It was a multiplicative congruential generator with parameters $a = 65539$, $c = 0$, and $m = 2^{31}$.

With a 32-bit integer word size, arithmetic mod 2^{31} can be done quickly. Furthermore, since $a = 2^{16} + 3$, the multiplication by a can be done with a shift and an addition. Such considerations were important on the computers of that era, but they gave the resulting sequence a very undesirable property.

The following relations are all taken mod 2^{31}:

$$\begin{aligned} x_{k+2} &= (2^{16} + 3)x_{k+1} \\ &= (2^{16} + 3)^2 x_k \\ &= (2^{32} + 6 * 2^{16} + 9) x_k \\ &= [6 * (2^{16} + 3) - 9] x_k \end{aligned}$$

Hence

$$x_{k+2} = 6 x_{k+1} - 9 x_k$$

As a result, there is an extremely high correlation among three successive random integers of the sequence generated by RANDU.

The IBM generator is implemented in the file `randSSL.m`.

The GUI `MontePI` can use the IBM generator to compute the value of π using a simple form of *Monte Carlo integration*. It generates random points in a cube and counts the fraction that lie within the inscribed sphere. It then computes π using

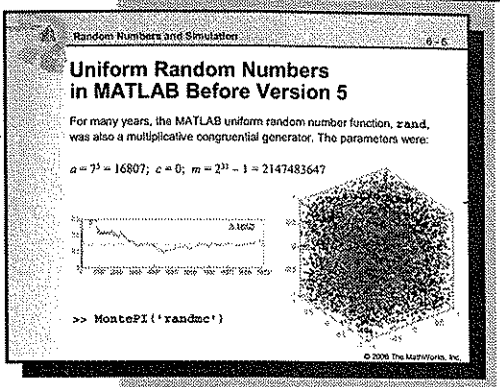
$$6(V_{sphere} / V_{cube}) = 6([4/3\pi r^3] / [(2r)^3]) = \pi$$

When using random numbers generated by `randSSL`, the resulting pattern of points is far from random. Nevertheless, the integration produces reasonable approximations of π.

Try

```
>> edit randSSL

>> r = ...
randSSL(100,1);
>> minr = min(r);
>> maxr = max(r);
>> meanr = mean(r);
>> varr = var(r);
>> hist(r,25)

>> edit MontePI
>> MontePI('randSSL')
   (repeatedly)
>> MontePI('rand')
   (repeatedly)
```

Uniform Random Numbers in MATLAB Before Version 5



For many years, the MATLAB uniform random number function, `rand`, was also a multiplicative congruential generator. The parameters were

$a = 7^5 = 16807$
$c = 0$
$m = 2^{31} - 1 = 2147483647$

These values were recommended in a 1988 paper by S. K. Park and K. W. Miller, "Random number generators: Good ones are hard to find" (Comm ACM, vol. 32). The old MATLAB multiplicative congruential generator is available in the M-file `randmc.m`.

The Monte Carlo integration

```
>> MontePI('randmc')
```

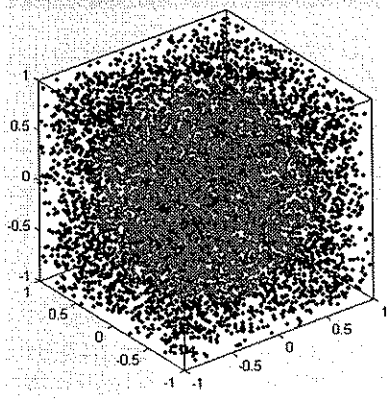
shows that the points do not suffer the correlation of the SSL generator. They generate a much better "random" cloud in the cube.

Like our exercise generator, `randmc` generates all real numbers of the form k / m for $k = 1 \ldots m - 1$. The smallest and largest are 0.00000000046566 and 0.99999999953434. The sequence repeats itself after m – 1 values, which is a little over 2 billion numbers. A few years ago, that was regarded as plenty. But today, a 800 MHz Pentium laptop can exhaust the period in less than half an hour. Of course, to do anything useful with 2 billion numbers takes more time, but we would still like to have a longer period.

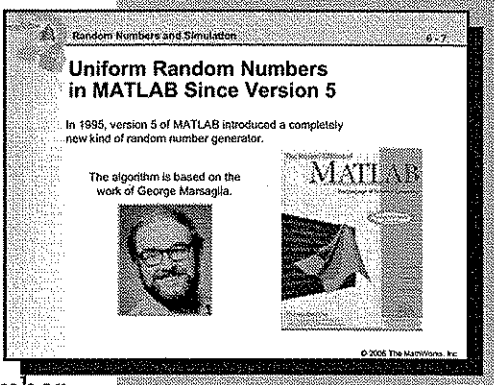
Try

```
>> edit randmc

>> r = ...
randmc(100,1);
>> minr = min(r);
>> maxr = max(r);
>> meanr = mean(r);
>> varr = var(r);
>> hist(r,25)

>> MontePI('randmc')
```

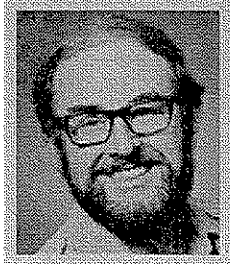
(repeatedly)



Uniform Random Numbers in MATLAB Since Version 5



In 1995, version 5 of MATLAB introduced a completely new kind of random number generator. The algorithm is based on work of George Marsaglia, a professor at Florida State University and the author of the classic analysis of random number generators, "Random numbers fall mainly in the planes," (Proc. Nat. Acad. Sci., vol 61, 1968).



George Marsaglia

Marsaglia's generator does not use Lehmer's algorithm. In fact, there are no multiplications or divisions at all. It is specifically designed to produce floating-point values, not just scaled integers.

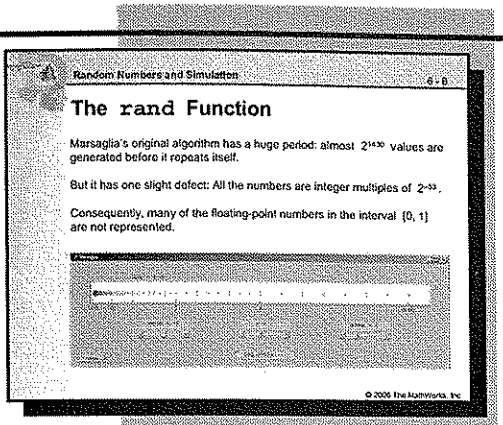
In place of a single seed, the new generator has 35 words of internal memory or *state*. Thirty-two of the words form a cache of floating-point numbers, z, between 0 and 1. The remaining three words contain an integer index i, which varies between 1 and 32, a single random integer j, and a "borrow" flag b. This entire state vector is built up a bit at a time during an initialization phase. Different values of j yield different initial states.

The generation of the ith floating-point number in the sequence involves a "subtract with borrow" step, where one number in the cache is replaced by the difference of two others:

$$z_i = z_{i+20} - z_{i+5} - b$$

The three indices, i, $i+20$, and $i+5$, are all interpreted mod 32 (by using just their last five bits). The quantity b is left over from the previous step; it is either zero or a small positive value. If the computed z_i is positive, b is set to zero for the next step. But if the computed z_i would be negative, it is made positive by adding 1.0 before it is saved and b is set to 2^{-53} for the next step. The quantity 2^{-53}, which is half of the MATLAB constant `eps`, is called one *ulp* because it is one *unit in the last place* for floating-point numbers slightly less than 1.

The `rand` Function

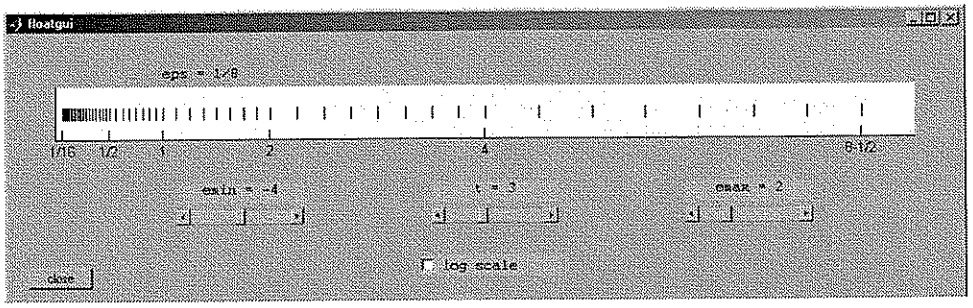


By itself, the generator described on the previous page is almost perfectly satisfactory. Marsaglia has shown that it has a huge period–almost 2^{1430} values are generated before it repeats itself. But it has one slight defect. All the numbers are the results of floating-point additions and subtractions of numbers in the initial cache, so they are all integer multiples of 2^{-53}. Consequently, many of the floating-point numbers in the interval [0, 1] are not represented.

Try

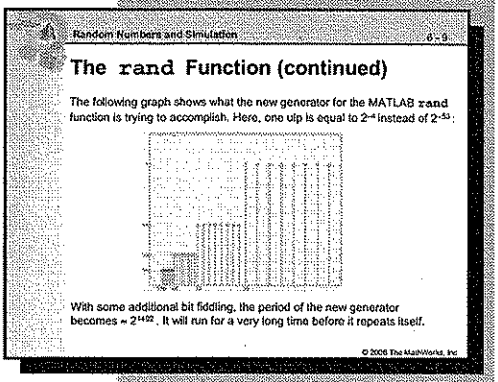
```
>> floatgui
```

The floating-point numbers between 1/2 and 1 are equally spaced with a spacing of one ulp, and our subtract-with-borrow generator will eventually generate all of them. But numbers less than 1/2 are more closely spaced and the generator would miss most of them. It would generate only half of the possible numbers in the interval [1/4, 1/2], only a quarter of the numbers in [1/8, 1/4] and so on.

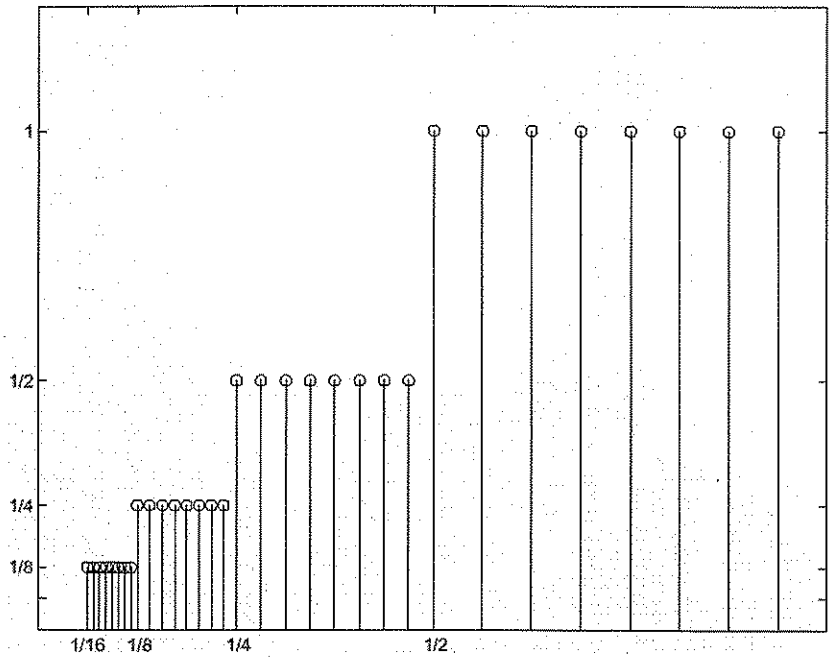


This is where the quantity j in the state vector comes in. It is the result of a separate, independent, random number generator based on bitwise logical operations. The floating-point fraction of each z_i is XORed with j to produce the result returned by the generator. This breaks up the even spacing of the numbers less than 1/2. It is now theoretically possible to generate all the floating-point numbers between 2^{-53} and $1 - 2^{-53}$. We're not sure they are all actually generated, but we don't know of any that can't be.

The `rand` Function (continued)



The following graph shows what the new generator is trying to accomplish. Here, one ulp is equal to 2^{-4} instead of 2^{-53}:



The graph depicts the relative frequency of each of the floating-point numbers. A total of 32 floating-point numbers are shown. Eight of them are between 1/2 and 1 and they are all equally like to occur. There are also eight numbers between 1/4 and 1/2, but, since this interval is only half as wide, each of them should occur only half as often. As we move to the left, each subinterval is half as wide as the previous one, but it still contains the same number of floating-point numbers, so their relative frequencies must be cut in half. Imagine this picture with 2^{53} numbers in each of 2^{32} smaller intervals and you will see what the new generator is doing.

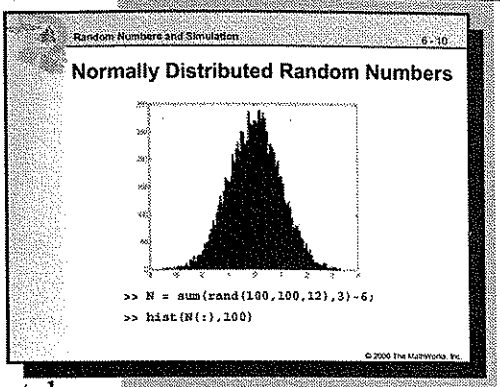
With the additional bit fiddling, the period of the new generator becomes something like 2^{1492}. It will run for a very long time before it repeats itself.

This is the generator implemented by the MATLAB `rand` function.

Try

```
>> doc rand
>> edit mrand
```

Normally Distributed Random Numbers



Most algorithms for generating normally distributed random numbers use transformations of uniform distributions.

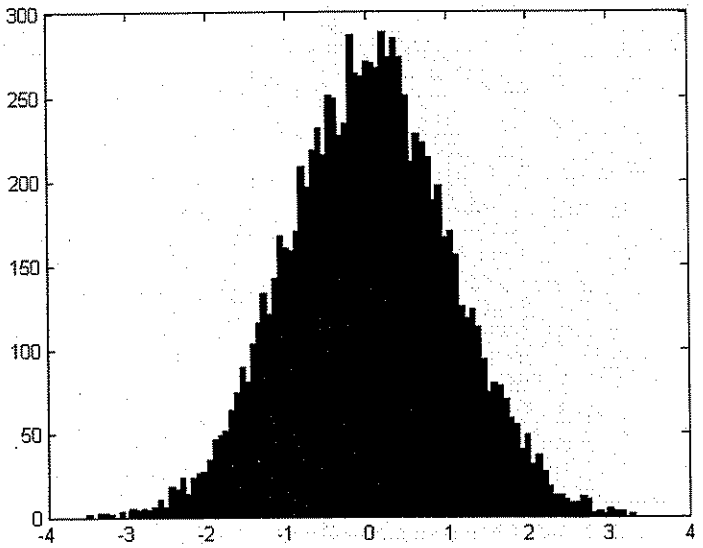
The simplest way to generate an *m*-by-*n* matrix with approximately normally distributed elements is to use the expression:

```
>> sum(rand(m,n,12),3) - 6
```

This works because `R = rand(m,n,p)` generates a three-dimensional uniformly distributed array and `sum(R,3)` sums along the third dimension. The result is a two-dimensional array with elements drawn from a distribution with mean $p/2$ and variance $p/12$ that approaches a normal distribution as $p \to \infty$.

If we take $p = 12$, we get a pretty good approximation to the normal distribution and we get the variance to be equal to one without any additional scaling.

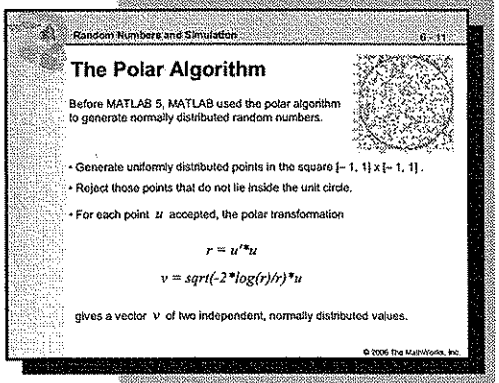
There are two difficulties with this approach. It requires twelve uniform random numbers to generate one normal random number, so it is slow. And the finite p approximation causes it to poorly approximate the tails of the normal distribution.



Try

```
>> N = ...
sum(rand( ...
100,100,12),3)-6;
>> hist(N(:),100)
```

The Polar Algorithm



Before MATLAB 5, MATLAB used the *polar algorithm* to generate normally distributed random numbers.

This algorithm finds uniformly distributed random points in the unit circle by first generating uniformly distributed random points in the square $[-1, 1] \times [-1, 1]$, then rejecting those points that do not lie inside the circle. Points in the square are represented by vectors u with two uniformly distributed components. For each point accepted, the polar transformation

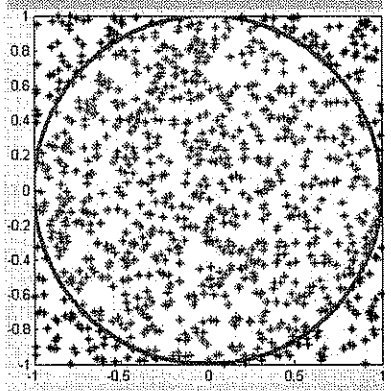
$$r = u'*u$$
$$v = sqrt(-2*log(r)/r)*u$$

gives a vector v of two independent, normally distributed values.

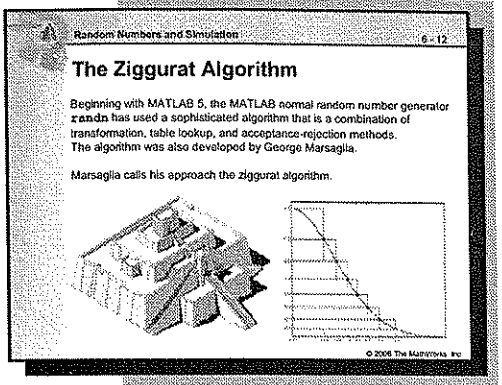
The polar algorithm does not involve any approximations, so it has the proper behavior in the tails of the distribution. But it is moderately expensive. Over 21% of the uniform numbers are rejected when they fall outside of the circle and the square root and logarithm calculations contribute significantly to the cost.

Try

```
>> edit polaralg
>> polaralg( ...
0,1,1000)
```

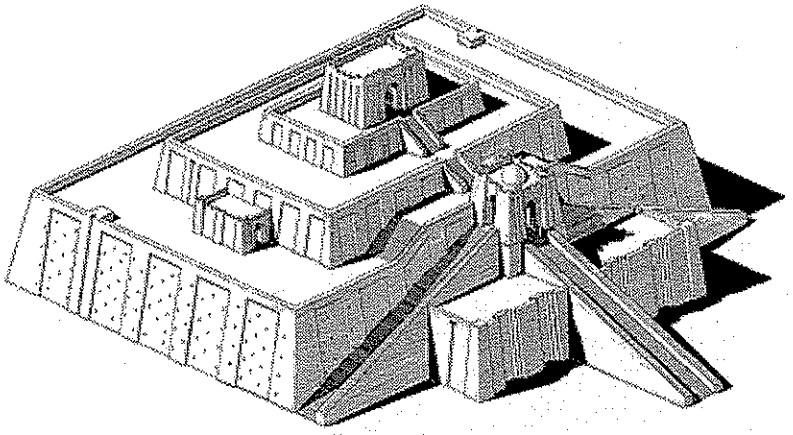


The Ziggurat Algorithm



Beginning with MATLAB 5, the MATLAB normal random number generator `randn` has used a sophisticated algorithm that is a combination of transformation, table lookup, and acceptance-rejection methods. The algorithm was also developed by George Marsaglia. Marsaglia calls his approach the *ziggurat algorithm*.

Ziggurats are ancient Mesopotamian terraced temple mounds that, mathematically, are two-dimensional step functions. A one-dimensional ziggurat underlies Marsaglia's algorithm.



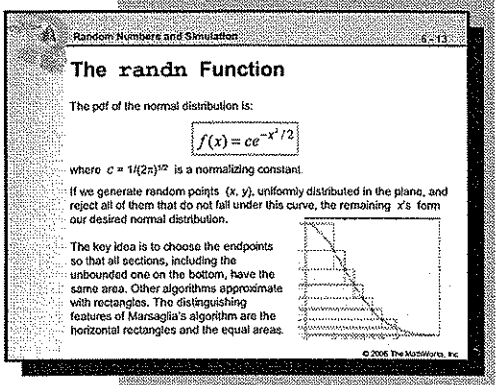
The Ziggurat of Ur

Marsaglia has refined his ziggurat algorithm over the years. An early version is described in volume II of Knuth's classic *The Art of Computer Programming*. The MATLAB version is described by Marsaglia and W. W. Tsang in the SIAM Journal of Scientific and Statistical Programming, volume 5, 1984, available in the online electronic journal, Journal of Statistical Software:

`http://www.jstatsoft.org/v05/i08/ziggurat.pdf`

We will describe this recent version in the following pages because it is the most elegant. The version used in MATLAB is more complicated, but is based on the same ideas and is just as effective.

The `randn` Function

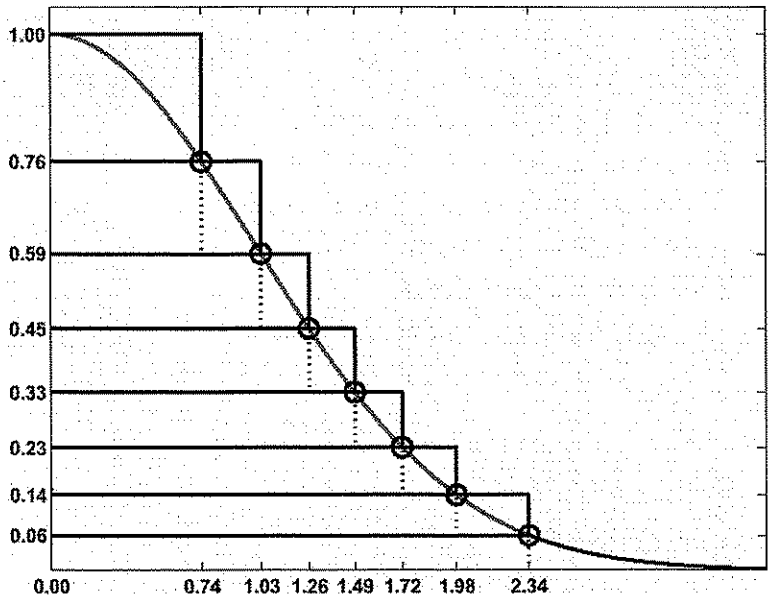


The probability density function of the normal distribution is the bell-shaped curve:

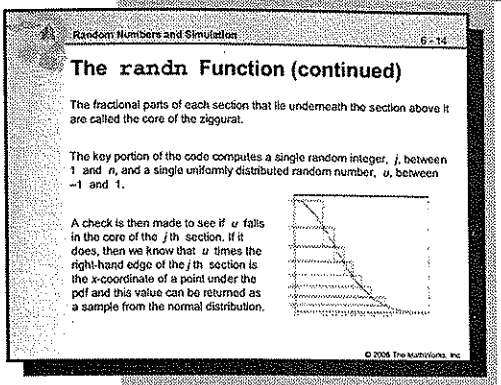
$$f(x) = ce^{-x^2/2}$$

where $c = 1/(2\pi)^{1/2}$ is a normalizing constant that we can ignore. If we generate random points (x, y), uniformly distributed in the plane, and reject all of them that do not fall under this curve, the remaining x's form our desired normal distribution.

The ziggurat algorithm covers the area under the pdf by a slightly larger area with n sections. The figure below has $n = 8$; actual code might use $n = 128$. The top $n - 1$ sections of the ziggurat are rectangles. The bottom section is a rectangle together with an infinite tail under the graph of $f(x)$. The right-hand edges of the rectangles are at the points z_k, $k = 2, \ldots, n$, shown with circles in the figure. With $f(z_1) = 1$ and $f(z_{n+1}) = 0$, the height of the kth section is $f(z_k) - f(z_{k+1})$. The key idea is to choose the z_k's so that all n sections, including the unbounded one on the bottom, have the same area. There are other algorithms that approximate the area under the pdf with rectangles. The distinguishing features of Marsaglia's algorithm are the facts that the rectangles are horizontal and have equal areas.



The `randn` Function (continued)

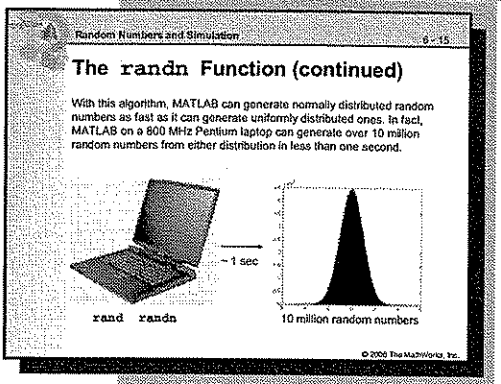


For a specified number, n, of sections, it is possible to solve a transcendental equation to find z_n, the point where the infinite tail meets the first rectangular section. In our figure with $n = 8$, it turns out that $z_n = 2.34$. In actual code with $n = 128$, $z_n = 3.4426$. Once z_n is known, it is easy to compute the common area of the sections and the other right-hand end points, z_k. It is also possible to compute $\sigma_k = z_{k-1} / z_k$, which is the fraction of each section that lies underneath the section above it. Call these fractional sections the *core* of the ziggurat. The right-hand edge of the core is the dotted line in our figure. The computation of these z_k's and σ_k's is done in an initialization section of code that is run only once.

After the initialization, normally distributed random numbers can be computed very quickly. The key portion of the code computes a single random integer, j, between 1 and n, and a single uniformly distributed random number, u, between -1 and 1. A check is then made to see if u falls in the core of the jth section. If it does, then we know that uz_j is the x-coordinate of a point under the pdf and this value can be returned as a sample from the normal distribution.

Most of the σ_j's are greater than 0.98, and the test is true over 97% of the time. One normal random number can usually be computed from one random integer, one random uniform, an if test and a multiplication. No square roots or logarithms are required. The point determined by j and u will fall outside of the core less than 3% of the time. This happens when $j = 1$ because the top section has no core, when j is between 2 and n – 1 and the random point is in one of the little rectangles covering the graph of $f(x)$, or when $j = n$ and the point is in the infinite tail. In these cases, additional computations involving logarithms, exponentials, and more uniform samples are required.

The `randn` Function (continued)

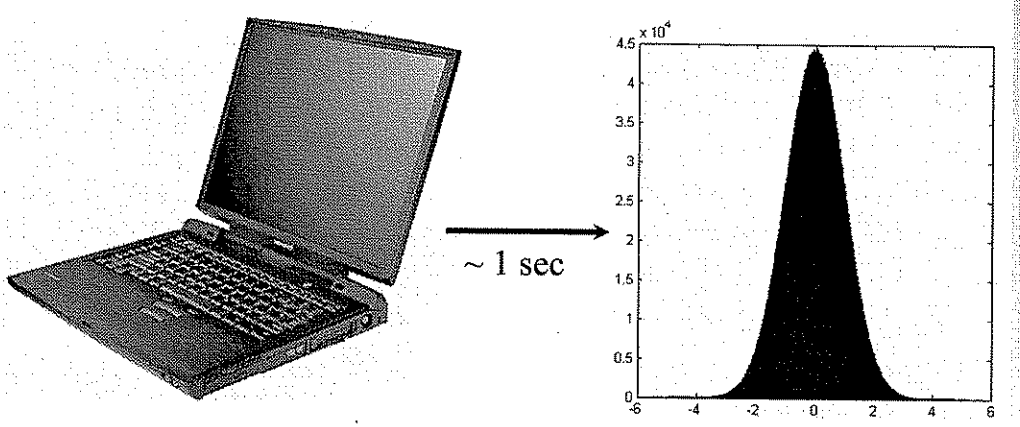


It is important to realize that, even though the ziggurat step function only approximates the probability density function, the resulting distribution is exactly normal. Decreasing n decreases the amount of storage required for the tables and increases the fraction of time that extra computation is required, but does not affect the accuracy. Even with $n = 8$, we would have to do the more costly corrections almost 23% of the time, instead of less than 3%, but we would still get an exact normal distribution.

With this algorithm, MATLAB can generate normally distributed random numbers as fast as it can generate uniformly distributed ones. In fact, MATLAB on a 800 MHz Pentium laptop can generate over 10 million random numbers from either distribution in less than one second.

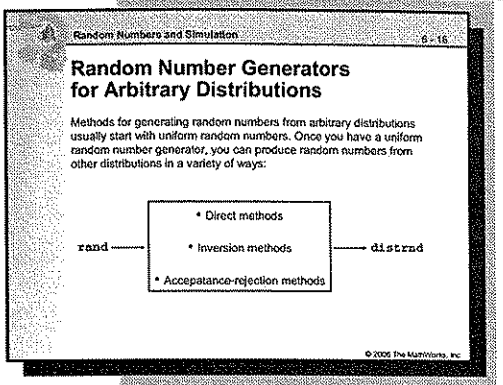
Try

```
>> doc randn
>> edit mrandn
```



`rand` `randn` 10 million random numbers

Random Number Generators for Arbitrary Distributions



Methods for generating random numbers from arbitrary distributions usually start with uniform random numbers. Once you have a uniform random number generator, you can produce random numbers from other distributions.

• Direct Methods

Direct methods follow from the definition of the distribution. Consider, for example, 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.

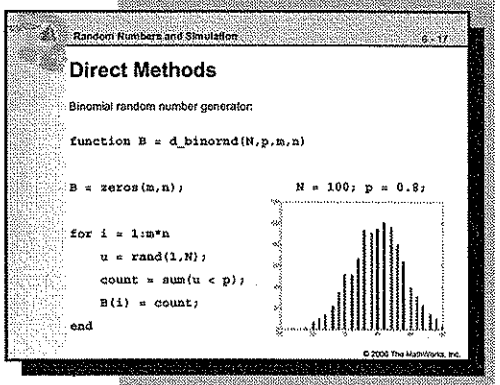
• Inversion Methods

Inversion methods are due to a fundamental theorem that relates the uniform distribution to other continuous distributions. If F is the cdf of a continuous distribution with inverse cdf F^{-1}, and U is a uniform random number, then $F^{-1}(U)$ has distribution F. You can thus generate a random number from a distribution by applying the inverse cdf for that distribution to a uniform random number. This approach, however, is usually not the most efficient.

• Acceptance-Rejection Methods

The functional form of some distributions makes it difficult or time consuming to generate random numbers using direct or inversion methods. Rejection methods can provide a good solution in these cases. These methods also begin with uniform random numbers, but they require the availability of an additional random number generator. If the goal is to generate random numbers from a distribution with pdf f, rejection methods use a generator for another distribution with pdf g satisfying $f(x) \le cg(x)$ for all x. Given a generator for g, rejection methods can combine it with a uniform generator to produce a generator for f.

Direct Methods

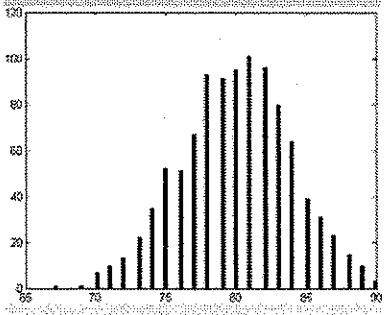


Here is a simple implementation of a random number generator for the binomial distribution with parameters N (number of trials) and p (probability of success). It uses the uniform random number generator `rand` and a direct method of generating N uniform random numbers and then counting the number that are less than p.

```
function B = d_binornd(N,p,m,n)

B = zeros(m,n);

for i = 1:m*n
    u = rand(1,N);
    count = sum(u < p);
    B(i) = count;
end
```

Try

```
>> edit d_binornd

>> N = 100; p = 0.8;
>> B = ...
d_binornd( ...
N,p,1,1000);
>> hist(B,N)
```

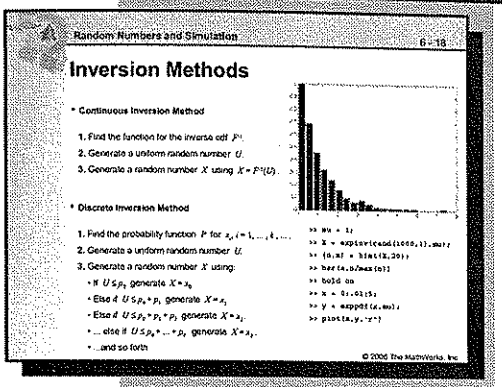


The Statistics Toolbox function `binornd` also uses a direct method, but it is based on the definition of the binomial distribution as a sum of Bernoulli random variables.

The method above is easily converted to a random number generator for the Poisson distribution with parameter λ by recalling that the Poisson distribution is just the limiting case of the binomial distribution as N approaches infinity and p approaches zero while Np is held fixed at λ. (Input λ to the generator and internally set N to some large number and p to λ / N.)

The Statistics Toolbox function `poissrnd` actually uses two methods: a waiting time method for small values of λ, and a method due to Ahrens and Dieter for larger values of λ.

Inversion Methods



Inversion methods are based on the fact that the outputs of continuous cdfs F are uniform on the interval (0, 1). Thus if U is a uniform random variable on (0, 1), we can obtain a random variable X from the distribution with cdf F using

$$X = F^{-1}(U)$$

• Continuous Inversion Method

1. Find the function for the inverse cdf F^{-1}.
2. Generate a uniform random number U.
3. Generate a random number X using $X = F^{-1}(U)$.

The same method can be adapted to the case of discrete distributions. Suppose we wish to generate a random variable X from a distribution described by

$$P(X = x_i) = p_i \;; \quad x_0 < x_1 < x_2 < \ldots \quad \sum_i p_i = 1$$

We can generate a uniform random number U and then use

$$X = x_i \quad \text{if} \quad F(x_{i-1}) < U < F(x_i)$$

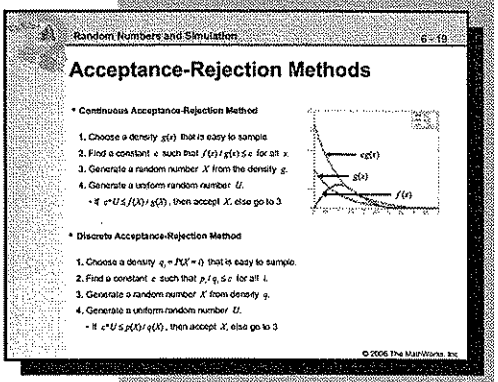
• Discrete Inversion Method

1. Find the probability function P for $x_i, i = 1, \ldots, k, \ldots$.
2. Generate a uniform random number U.
3. Generate a random number X using:
   - If $U \le p_0$ generate $X = x_0$
   - Else if $U \le p_0 + p_1$ generate $X = x_1$
   - Else if $U \le p_0 + p_1 + p_2$ generate $X = x_2$
   - ... else if $U \le p_0 + \ldots + p_k$ generate $X = x_k$
   - ...and so forth.

Try

```
>> mu = 1;
>> X = ...
expinv( ...
rand(1000,1),mu);
>> [n,x] = ...
hist(X,20);
>> bar(x,n/max(n))
>> hold on
>> x = 0:.01:5;
>> y = exppdf(x,mu);
>> plot(x,y,'r')

>> edit di_rnd
>> X = ...
di_rnd(1,1000);
>> hist(X,3)
```

Acceptance-Rejection Methods

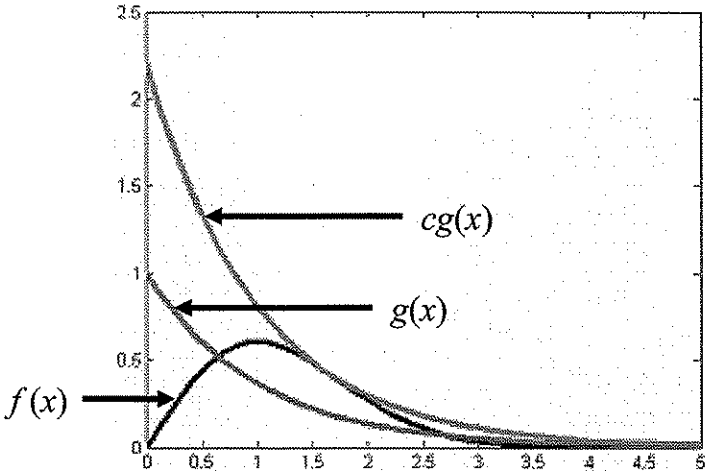


Acceptance-rejection methods may be used when we have a method for generating random numbers from one pdf g, but not from the pdf f in which we are interested. We first generate a random number X from g and then accept it as a random number from f with probability $f(X) / g(X)$.

Exercise: Acceptance-Rejection Methods

• Continuous Acceptance-Rejection Method

1. Choose a density $g(x)$ that is easy to sample.
2. Find a constant c such that $f(x) / g(x) \le c$ for all x.
3. Generate a random number X from the density g.
4. Generate a uniform random number U.
   - If $c^*U \le f(X) / g(X)$, then accept X, else go to 3.

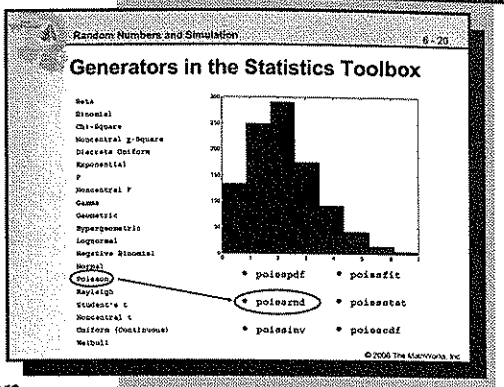


This method can also be adapted to discrete distributions. In this case we assume that we have a method for generating random variables from a distribution with $q_i = P(X = i)$ and we would like to generate random variables from a distribution with $p_i = P(X = i)$.

• Discrete Acceptance-Rejection Method

1. Choose a density $q_i = P(X = i)$ that is easy to sample.
2. Find a constant c such that $p_i / q_i \le c$ for all i.
3. Generate a random number X from density q.
4. Generate a uniform random number U.
   - If $c^*U \le p(X) / q(X)$, then accept X, else go to 3.

Random Number Generators in the Statistics Toolbox



The Statistics Toolbox supports random number generation from twenty different common distributions. For each distribution, random numbers may be generated with the `distrnd` function, where `dist` is the name of the distribution.

- `distrnd`

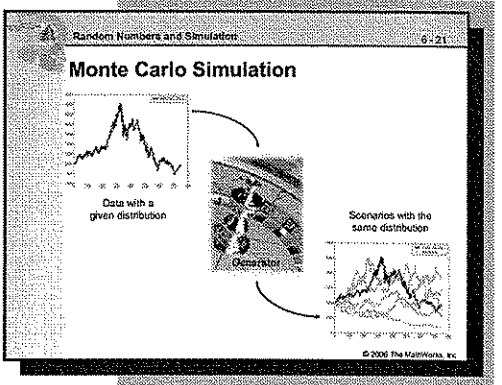
`betarnd`	- Beta random numbers
`binornd`	- Binomial random numbers
`chi2rnd`	- Chi square random numbers
`exprnd`	- Exponential random numbers
`frnd`	- F random numbers
`gamrnd`	- Gamma random numbers
`geornd`	- Geometric random numbers
`hygernd`	- Hypergeometric random numbers
`iwishrnd`	- Inverse Wishart random matrix
`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`	- Random numbers from specified distribution
`raylrnd`	- Rayleigh random numbers
`trnd`	- T random numbers
`unidrnd`	- Discrete uniform random numbers
`unifrnd`	- Uniform random numbers
`weibrnd`	- Weibull random numbers
`wishrnd`	- Wishart random matrix

Distributions supported by the Statistics Toolbox

`Beta`
`Binomial`
`Chi-Square`
`Noncentral χ-Square`
`Discrete Uniform`
`Exponential`
`F`
`Noncentral F`
`Gamma`
`Geometric`
`Hypergeometric`
`Lognormal`
`Negative Binomial`
`Normal`
`Poisson`
`Rayleigh`
`Student's t`
`Noncentral t`
`Uniform (Continuous)`
`Weibull`

Random numbers are generated using a variety of techniques. To find out more about the algorithm for a particular generator, check the documentation or open it in the editor.

Monte Carlo Simulation



On p. 2-3 we outlined the role of statistics in the larger process of mathematical modeling. When the components of a mathematical model accurately mimic the behavior of a system, we may use the model to make reliable predictions.

In many circumstances, however, collecting data, building a model, and testing it against a system of interest may be difficult, time-consuming, costly, or even dangerous. In such cases, *probabilistic models*, based on random simulations guided by the statistical properties of known data (rather than an analysis of the entire system), often provide reasonable alternatives. Scenarios generated by probabilistic models are often called *Monte Carlo simulations*, after the casino in Monaco. Monte Carlo methods for analysis and prediction of system behavior make essential use of random number generation from known distributions.

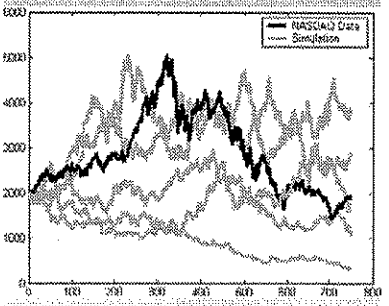
We've already seen an example of Monte Carlo simulation, in the exercise at the end of Chapter 3. There we looked at some historical data on the NASDAQ index. We saw that daily returns in the data, when reexpressed logarithmically, followed an approximately normal distribution. Using the statistical properties of this distribution, we were able to generate random returns with similar characteristics and so play out a variety of possible scenarios over time. Cumulatively, these scenarios expanded our single data set into a distribution of possible data sets, from which we could begin to estimate probabilities of market moves, etc. (See pp. 3-17 – 3-20.)

An alternative to distribution fitting and random number generation is the method of *bootstrapping*, discussed on p. 3-15.

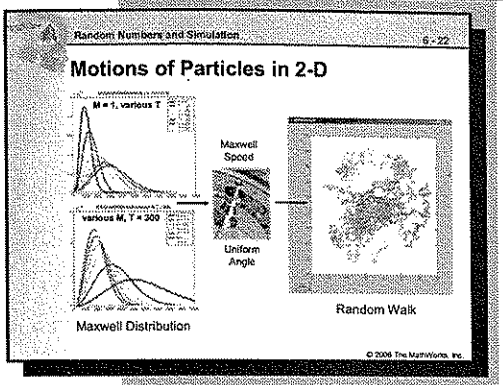
Try

```
>> edit randsim
>> randsim
```
(repeatedly)

```
>> edit bootsim
>> bootsim
```
(repeatedly)



Motion of Particles In 2-D



Another example of a Monte Carlo simulation is modeling the motion of particles in 2-D. The precise interactions of particles on a molecular level may be unknown, but their overall statistical characteristics are readily modeled.

Suppose that $X_1, X_2, \ldots$ are independent, identically distributed real-valued random variables with common density function f, mean μ, and variance σ^2. The n^{th} partial sum is the random variable

$$Y_n = X_1 + X_2 + \cdots + X_n$$

The random process $Y_0, Y_1, Y_2 \ldots$ is called a *random walk*, since we can think of Y_n as the position at time n for a walker who makes successive random steps $X_1, X_2, \ldots$. The mean and variance of Y_n, respectively, are $n\mu$ and $n\sigma^2$. A graph of the values of Y_n as a function of n is called a *path* of the random walk.

The file `gasmotion.m` uses a random walk technique to model the motion of a 2D perfect gas.

At any step, an angle of motion for each particle is randomly generated from a uniform distribution scaled to the interval $[0, 2\pi]$.

The speeds of particles in an ideal gas follow a Maxwell distribution with pdf

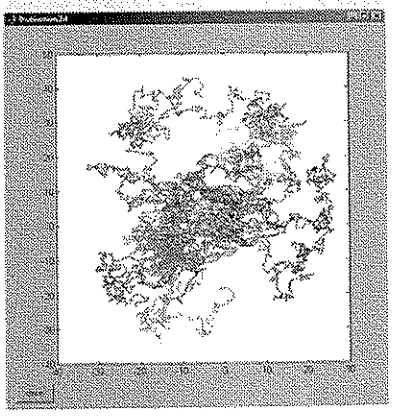
$$f(v) = 4\pi\left(\frac{M}{2\pi RT}\right)^{3/2} v^2 e^{\left(\frac{-Mv^2}{2RT}\right)}$$

Since this distribution type is not directly supported by the Statistics Toolbox, and since the cdf does not have a closed-form inverse, we generate random speeds using an acceptance-rejection method.

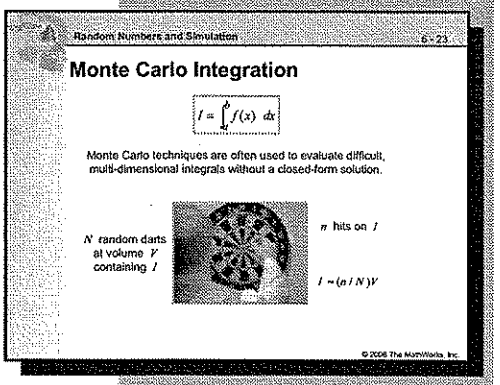
We then use the `cumsum` command to form the random walk from the random variables for angle and speed.

Try

```
>> edit gasmotion
>> edit Mxrnd

>> gasmotion( ...
100,10,0.18,10,1);
>> gasmotion( ...
100,10,0.18,500,1);

>> [v,x,y] = ...
gasmotion( ...
100,10,0.18,500,0);
>> plot(x,y)
>> figure; hist(v)
>> figure; hist(v(:))
```



Monte Carlo Integration



Monte Carlo techniques are often used to evaluate difficult, multi-dimensional integrals without a closed-form solution.

One method is based upon an idea similar to the acceptance-rejection method for generating random variables from arbitrary distribution functions. Suppose we wish to evaluate

$$I = \int_a^b f(x)\ dx$$

If we put a bounding box around the function $f(x)$, then the integral of $f(x)$ can be understood to be the fraction of the bounding box that is also within $f(x)$. So if we choose a uniform random point within the bounding box, the probability that the point is within $f(x)$ is given by the fraction of the area that $f(x)$ occupies. The method is to take a large number of random points within the box and count the number that are within $f(x)$ to get the area

$$I \approx \frac{n}{N} V$$

where n is the number of points within $f(x)$, N is the total number of points generated, and V is the volume of the bounding box. The method is inefficient: many points are usually required to make the approximation converge towards I with any degree of precision.

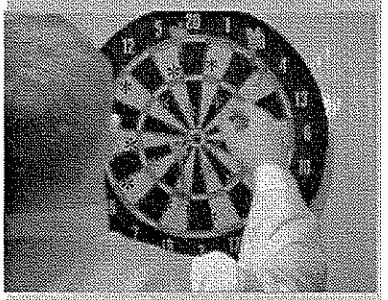
A more efficient approach is to note that we can write I as

$$I = \int_a^b f(x)g(x)\ dx = \frac{1}{V} \int_a^b Vf(x)g(x)\ dx$$

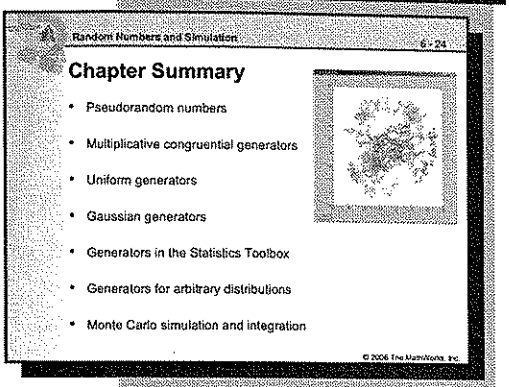
where $g(x)$ is equal to 1 if x is in the domain of integration and 0 otherwise. This can be interpreted as the mean value of the function $h(x) = Vf(x)g(x)$ for the random variable x which is uniformly distributed within *the domain* (rather than V). This gives a more quickly convergent approximation procedure:

$$I \approx \frac{1}{N} \sum_{i=1}^{N} h(x_i) = \frac{V}{N} \sum_{i=1}^{N} g(x_i)$$

Exercise: Microphone Coverage



Chapter Summary

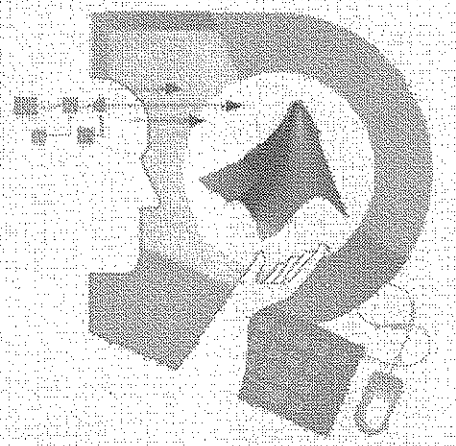


- Pseudorandom numbers
- Multiplicative congruential generators
- Uniform generators
- Gaussian generators
- Generators in the Statistics Toolbox
- Generators for arbitrary distributions
- Monte Carlo simulation and integration

In the next chapter we discuss hypothesis testing as a procedure for assessing assertions about arbitrary population characteristics.

Statistical Methods in MATLAB®

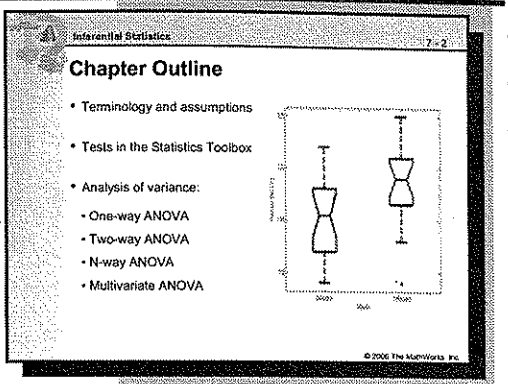
Inferential Statistics



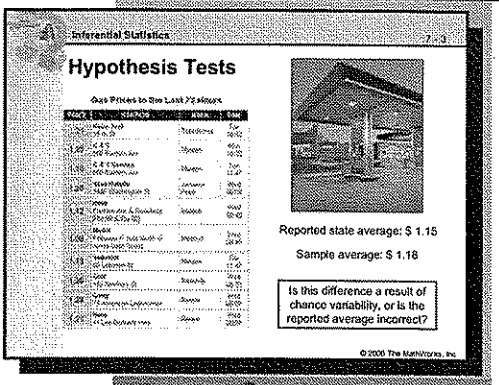
The MathWorks
Training Services

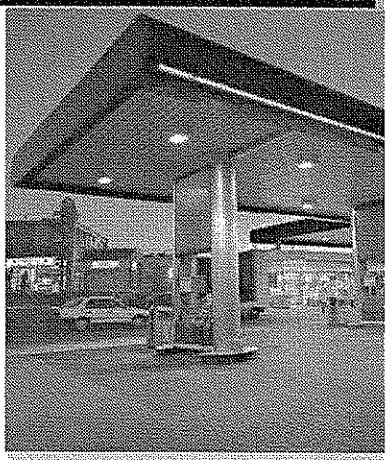
Chapter Outline

- Terminology and assumptions
- Tests in the Statistics Toolbox
- Analysis of variance:
  - One-way ANOVA
  - Two-way ANOVA
  - N-way ANOVA
  - Multivariate ANOVA



Hypothesis Tests





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.

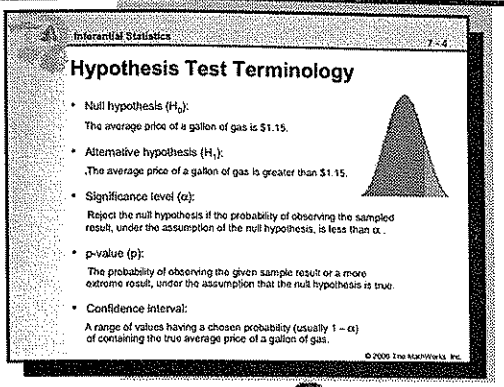
A simpler approach is to find out the price of gas at a small number of randomly chosen stations around the state and compare the average price to $1.15.

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.

Gas Prices in the Last 72 Hours

PRICE	STATION	AREA	TIME
1.21	**Race Trak** 15 th St	Tuscaloosa	Tue 10:52
1.20	**C & T** 856 Eastern Ave	Malden	Mon 18:20
1.10	**C & T Service** 856 Eastern Ave	Malden	Tue 11:47
1.20	**Stan Hatoffs** 3440 Washington St	Jamaica Plain	Wed 00:09
1.12	**Hess** Eastern Ave & Broadway (Rte 99 & Rte 60)	Malden	Wed 08:49
1.09	**Mobil** Fellsway (1 mile North of Ames Dept Store)	Medford	Wed 08:49
1.13	**Superior** 60 Lebanon St	Malden	Tue 11:47
1.25	**Gulf** 153 Newbury St	Peabody	Wed 09:33
1.28	**Getty** 79 American Legion Hwy	Revere	Wed 00:09
1.21	**Hess** 41 Lee Burbank Hwy	Revere	Wed 00:09

Hypothesis Test Terminology



To get started, there are some terms to define:

- 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 H0: $\mu = 1.15$.

- 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 H1: $\mu > 1.15$. The other possibilities are H1: $\mu < 1.15$ and H1: $\mu \neq 1.15$.

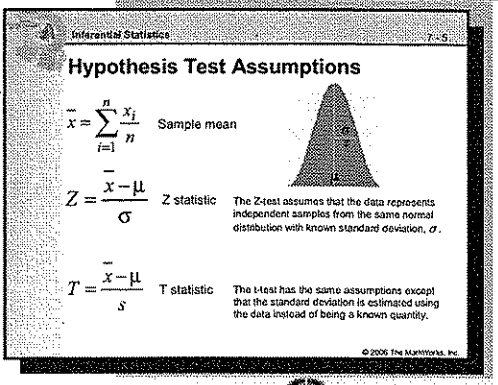
- The *significance level* is related to the degree of certainty you require in order to reject the null hypothesis in favor of the alternative. By taking a small sample you cannot be certain about your conclusion. So you decide in advance to reject the null hypothesis if the probability of observing 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 α.

- The *p-value* is the probability of observing a value as extreme as or more extreme than the given sample result, under the assumption that the null hypothesis is true. If the *p*-value is less than α , then you reject the null hypothesis. For example, if $\alpha = 0.05$ and the *p*-value is 0.03, you reject the null hypothesis.

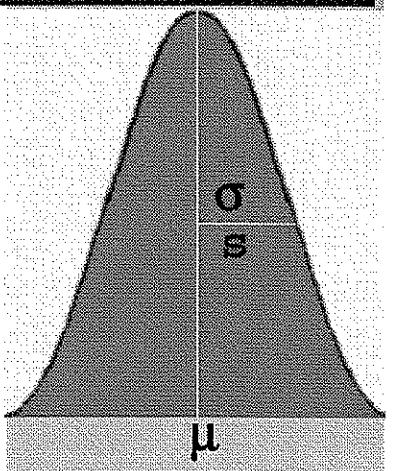
  The converse is not true. If the *p*-value is greater than α , you have insufficient evidence to reject the null hypothesis.

- 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 our example, 1.15 is inside a 95% confidence interval for the mean, μ . That is equivalent to being unable to reject the null hypothesis at a significance level of 0.05. Conversely, if the 100(1- α) confidence interval does not contain 1.15, then you reject the null hypothesis at the α level.

Hypothesis Test Assumptions



The difference between hypothesis test procedures often arises from differences in the assumptions that you are 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, σ. 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

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

where

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

is the sample mean.

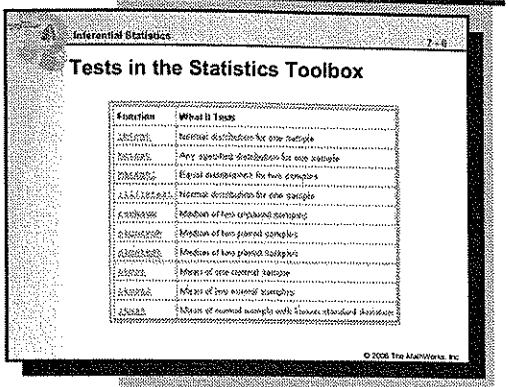
The signal is the difference between μ and the hypothesized mean. The noise is the standard deviation posited or estimated.

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, ν, equal to one less than the number of data.

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 or a more extreme 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 very 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, as we saw in Chapter 3, there are functions for testing whether the normal assumption is reasonable.

Tests in the Statistics Toolbox



The Statistics Toolbox has functions for performing the following tests:

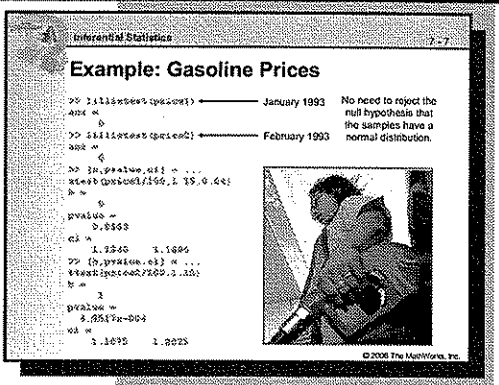
Function	What it Tests
jbtest	Normal distribution for one sample
kstest	Any specified distribution for one sample
kstest2	Equal distributions for two samples
lillietest	Normal distribution for one sample
ranksum	Median of two unpaired samples
signrank	Median of two paired samples
signtest	Median of two paired samples
ttest	Mean of one normal sample
ttest2	Mean of two normal samples
ztest	Mean of normal sample with known standard deviation

Examples of the distribution testing functions `jbtest`, `kstest`, and `lillietest` were introduced in Chapter 3 (p. 3-16).

Try

```
>> doc jbtest

>> doc kstest

>> doc kstest2

>> doc lillietest

>> doc ranksum

>> doc signrank

>> doc signtest

>> doc ttest

>> doc ttest2

>> doc ztest
```

Example: Gasoline Prices



This example uses the gasoline price data in the MAT-file `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)
>> lillietest(price2)
```

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)
```

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:

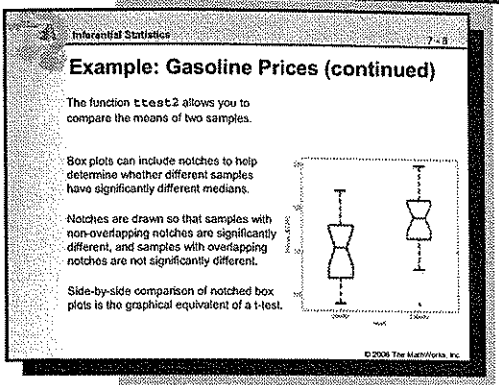
```
>> [h,pvalue,ci] = ttest(price2/100,1.15)
```

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

Try

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

>> lillietest( ...
price1)
>> lillietest( ...
price2)

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

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

Example: Gasoline Prices (continued)



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)
```

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. We 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)
```

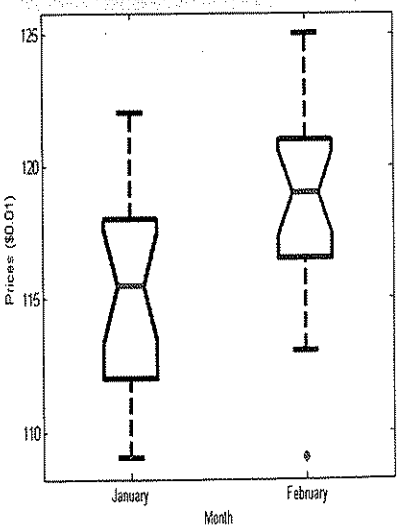
As might be expected, the rank sum test leads to the same conclusion but it is less sensitive to the difference between samples (higher *p*-value). The box plot below gives the same conclusion graphically. Note that the notches have little, if any, overlap:

```
>> boxplot(prices,1)
>> set(gca,'XtickLabel', ...
str2mat('January','February'))
>> xlabel('Month')
>> ylabel('Prices ($0.01)')
```

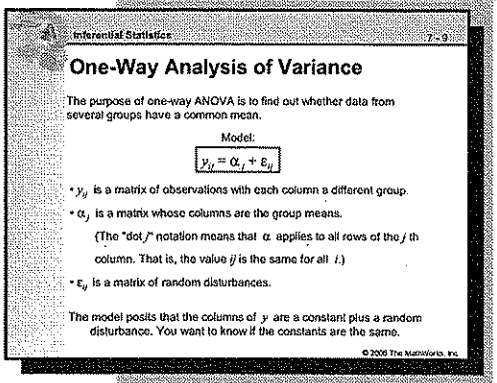
Refer to Chapter 2 (p. 2-14) for more information about box plots.

Try

```
>> [h,sig,ci] = ...
ttest2( ...
price1,price2)

>> [p,h,stats] = ...
ranksum( ...
price1,price2)

>> boxplot(prices,1)
>> set( ...
gca, ...
'XtickLabel', ...
str2mat( ...
'January', ...
'February'))
>> xlabel('Month')
>> ylabel( ...
'Prices ($0.01)')
```



One-Way Analysis of Variance



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

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

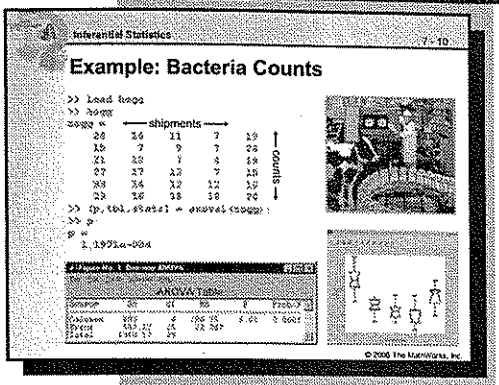
$$y_{ij} = \alpha_{.j} + \varepsilon_{ij}$$

where

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

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

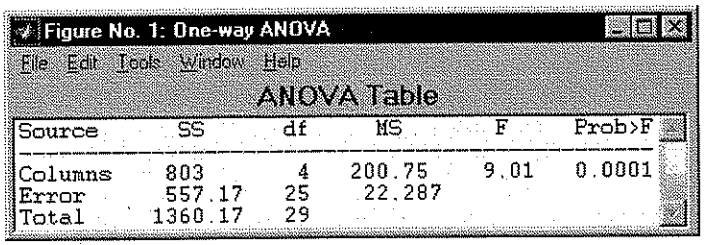
Example: Bacteria Counts



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
>> [p,tbl,stats] = anova1(hogg);
>> p
```

The standard ANOVA table has columns for the sums of squares, degrees of freedom, mean squares, F statistic, and p-value.



Source	SS	df	MS	F	Prob>F
Columns	803	4	200.75	9.01	0.0001
Error	557.17	25	22.287		
Total	1360.17	29			

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

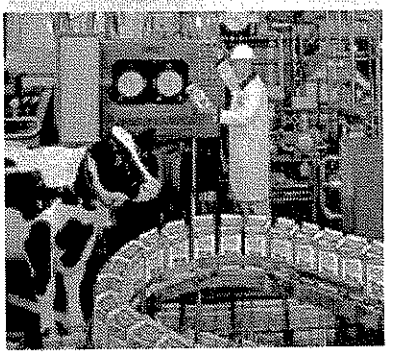
In this case the p-value is about 0.0001, a very small value. This is a strong indication that the bacteria counts from the different tankers are not the same. An F statistic as extreme as the observed F would occur by chance only once in 10,000 times if the counts were truly equal.

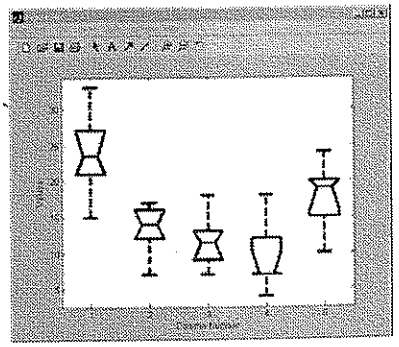
The p-value returned by anova1 depends on assumptions about the random disturbances ε_{ij} in the model equation. For the p-value to be correct, these disturbances need to be independent, normally distributed, and have constant variance.

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`.

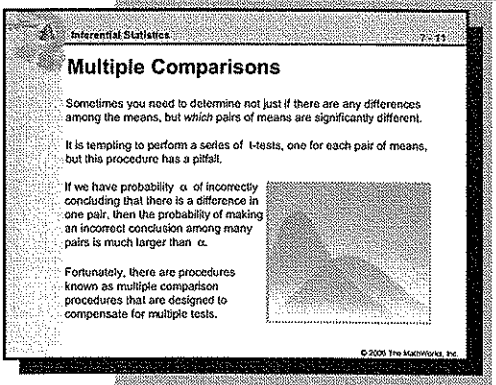
Try

```
>> load hogg
>> hogg

>> [p, ...
tbl, ...
stats] = ...
anova1(hogg);
>> p
```





Multiple Comparisons

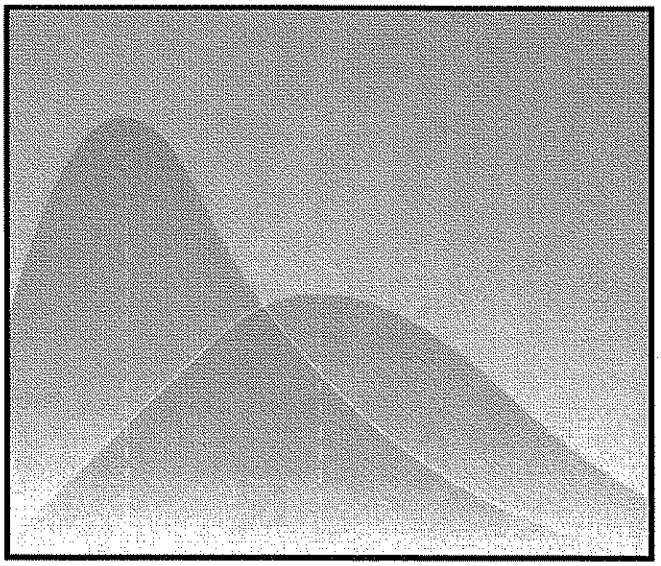


Sometimes you need to determine not just if there are any differences among the means, but specifically which pairs of means are significantly different. It is tempting to perform a series of *t*-tests, one for each pair of means, but this procedure has a pitfall.

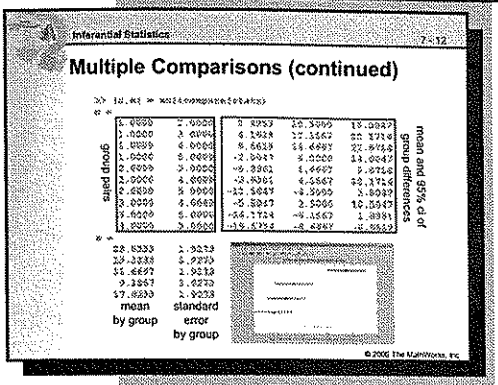
In a *t*-test, we compute a *t* statistic and compare it to a critical value. The critical value is chosen so that when the means are really the same (any apparent difference is due to random chance), the probability that the t statistic will exceed the critical value is small, say 5%. When the means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



Multiple Comparisons (continued)



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

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

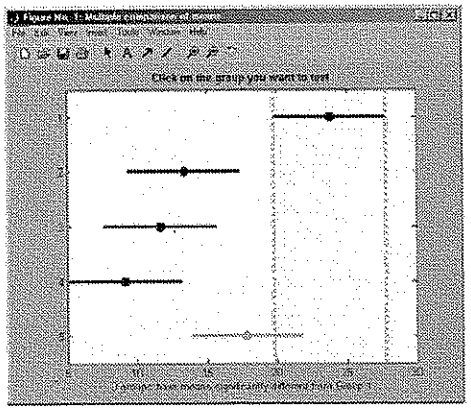
The first output from `multcompare` has one row for each pair of groups, with an estimate of the difference in group means and a confidence interval for that group. For example, the second row has the values:

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 we can conclude that the means of groups 1 and 3 are different.

The second output contains the mean and standard error by group.

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

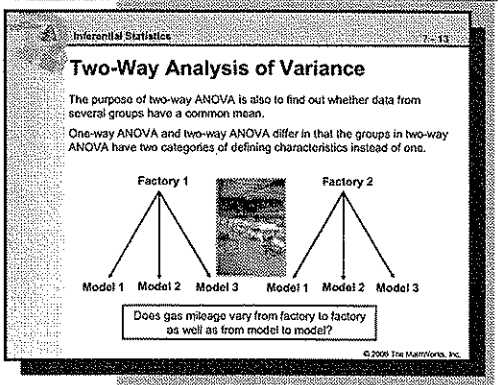


The graph shows that group 1 is significantly different from groups 2, 3, and 4. By using the mouse to select group 4, you can determine that it is also significantly different from group 5. Other pairs are not significantly different.

Try

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

Two-Way Analysis of Variance

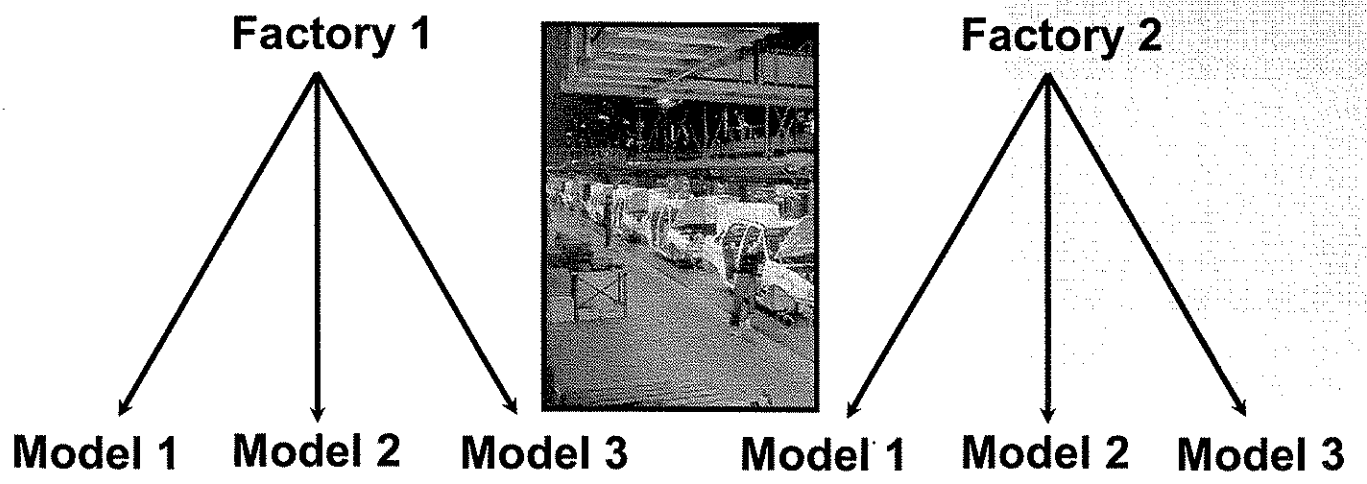


The purpose of two-way ANOVA is also to find out whether data from several groups have a common mean. One-way ANOVA and two-way ANOVA differ in that the groups in two-way ANOVA have two categories of defining characteristics instead of one.

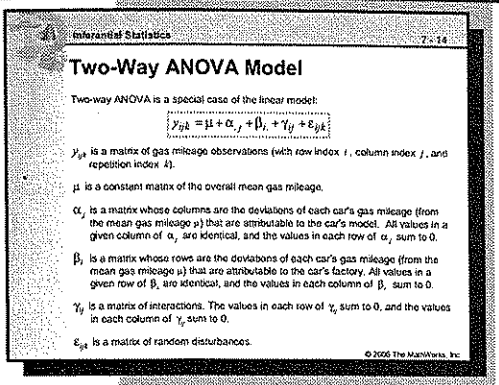
Suppose an automobile company has two factories, and each factory makes the same three models of car. It is reasonable to ask if the gas mileage in the cars varies from factory to factory as well as from model to model. We use two predictors, factory and model, to explain differences in mileage.

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 model.



Two-Way ANOVA 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,

y_{ijk} is a matrix of gas mileage observations (with row index i, column index j, and repetition index k).

μ is a constant matrix of the overall mean gas mileage.

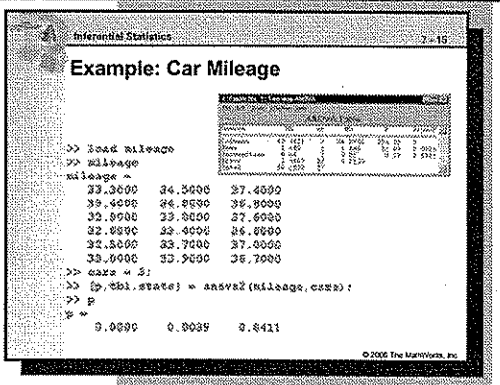
$\alpha_{.j}$ is a matrix whose columns are the deviations of each car's gas mileage (from the mean gas mileage μ) that are attributable to the car's model. All values in a given column of $\alpha_{.j}$ are identical, and the values in each row of $\alpha_{.j}$ sum to 0.

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

γ_{ij} is a matrix of interactions. The values in each row of γ_{ij} sum to 0, and the values in each column of γ_{ij} sum to 0.

ε_{ijk} is a matrix of random disturbances.

Example: Car Mileage



The purpose of the example is to determine the effect of car model and factory on the mileage rating of cars.

```
>> load mileage
>> mileage
>> cars = 3;
>>[p,tbl,stats] = anova2(mileage,cars);
>> p
```

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, F statistics, and p-values.

Figure No. 1: Two-way ANOVA

File Edit Tools Window Help

ANOVA Table

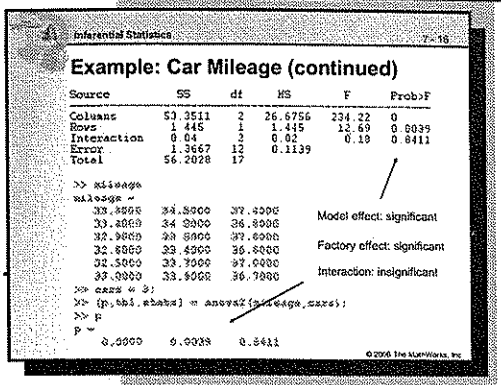
Source	SS	df	MS	F	Prob>F
Columns	53.3511	2	26.6756	234.22	0
Rows	1.445	1	1.445	12.69	0.0039
Interaction	0.04	2	0.02	0.18	0.8411
Error	1.3667	12	0.1139		
Total	56.2028	17			

You can use the F statistics to do hypotheses tests to find out if the mileage is the same across models, factories, and model-factory pairs (after adjusting for the additive effects). The function `anova2` returns the p-value from these tests.

Try

```
>> load mileage
>> mileage
>> cars = 3;
>>[p,tbl,stats] = ...
anova2(mileage,cars);
>> p
```

Example: Car Mileage (continued)



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 models is significantly different.

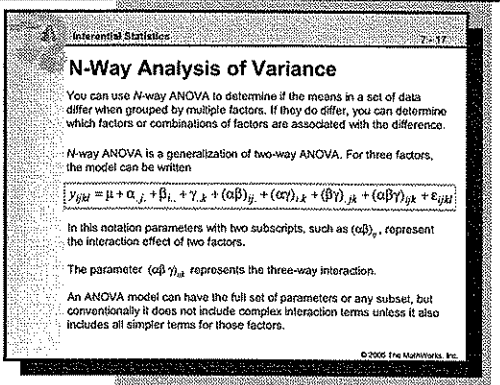
The *p*-value for the factory effect is 0.0039, which is also highly significant. This indicates that one factory is out-performing the other in the gas mileage of the cars it produces. The observed *p*-value indicates that an *F* statistic as extreme as the observed *F* would occur by chance about four out of 1000 times if the gas mileage were truly equal from factory to factory.

There does not appear to be any interaction between factories and models. The *p*-value, 0.8411, means that the observed result is quite likely (84 out 100 times) given that there is no interaction.

The *p*-values returned by `anova2` depend on assumptions about the random disturbances ε_{ijk} in the model equation. For the *p*-values to be correct these disturbances need to be independent, normally distributed, and have constant variance.

In addition, `anova2` requires that data be balanced, which in this case means there must be the same number of cars for each combination of model and factory.

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.

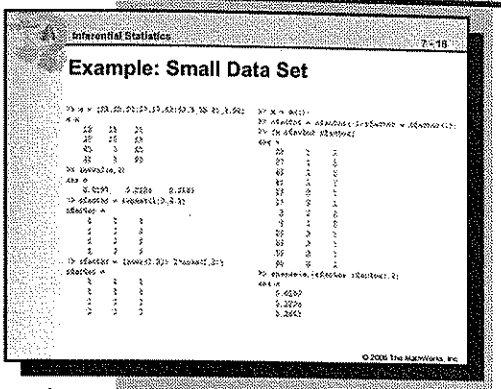
N-way ANOVA is a generalization of two-way ANOVA. For three factors, the model can be written

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

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 for 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.

Example: Small Data Set



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

```
>> m = [23,15,20;27,17,63;43,3,55;41,9,90]
>> anova2(m,2)
```

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 we could create them as follows.

```
>> cfactor = repmat(1:3,4,1)
>> rfactor = [ones(2,3); 2*ones(2,3)]
```

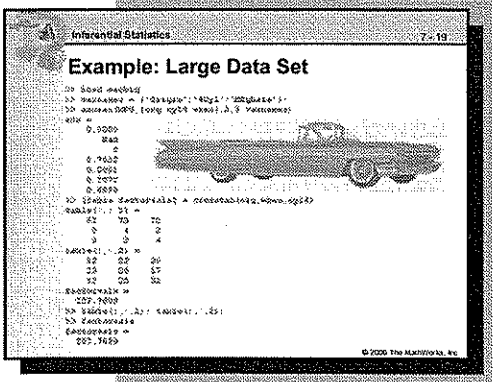
The `cfactor` matrix shows that each column of `m` represents a different level of the column factor. The `rfactor` matrix shows that the top two rows of `m` represent one level of the row factor, and bottom two rows of `m` represent a second level of the row factor. In other words, each value `m(i,j)` represents an observation at column factor level `cfactor(i,j)` and row factor level `rfactor(i,j)`. To solve the above problem with `anovan`, we need to reshape `m`, `cfactor`, and `rfactor` to be vectors:

```
>> m = m(:);
>> cfactor = cfactor(:);
>> rfactor = rfactor(:);
>> [m cfactor rfactor]
>> anovan(m,{cfactor rfactor},2)
```

Try

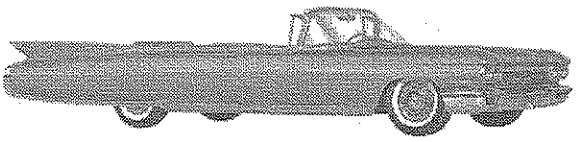
```
>> m = ...
[23,15,20
27,17,63
43,3,55
41,9,90]
>> anova2(m,2)

>> cfactor = ...
repmat(1:3,4,1)
>> rfactor = ...
[ones(2,3)  2*ones(2,
3)]

>> m = m(:);
>> cfactor = ...
cfactor(:);
>> rfactor = ...
rfactor(:);
>> [m ...
cfactor ...
rfactor]
>> anovan( ...
m, ...
{cfactor rfactor},...
2)
```

Example: Large Data Set



In the previous example we used `anova2` to study a small data set measuring car mileage. Now we study a larger set of car data with mileage and other information on 406 cars made between 1970 and 1982. First we load the data set and look at the variable names:

```
>> load carbig
>> whos
```



We will focus our attention 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 we 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)
```

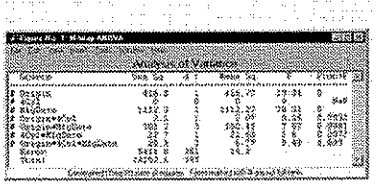
Note that many terms are marked by a # symbol as not having full rank, and one of them has zero degrees of freedom and is missing a *p*-value. This can happen when there are missing factor combinations and the model has higher-order terms. In this case, the cross-tabulation below shows that there are 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,factorvals] = crosstab(org,when,cyl4)
>> table(:,:,1)
>> table(:,:,2)
>> factorvals
```

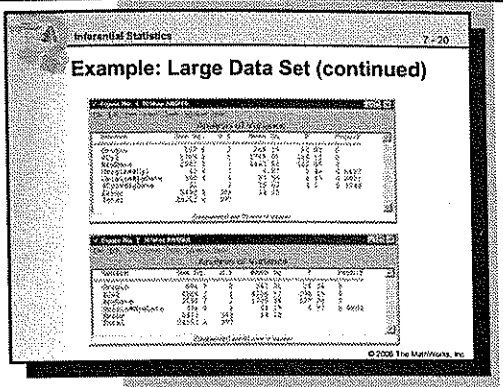
Consequently it is impossible to estimate the three-way interaction effects, and including the three-way interaction term in the model makes the fit singular.

Try

```
>> load carbig
>> whos

>> varnames = ...
{'Origin';'4Cyl'; ...
'MfgDate'};
>> anovan( ...
MPG, ...
{org cyl4 when}, ...
3,3,varnames)

>> [table, ...
factorvals] = ...
crosstab( ...
org,when,cyl4)
>> table(:,:,1)
>> table(:,:,2)
>> factorvals
```



Example: Large Data Set (continued)



Using even the limited information available in the ANOVA table, we can see that the three-way interaction has a *p*-value of 0.699, so it is not significant. We decide to request only two-way interactions this time:

```
>> [p,tbl,stats,termvec] = ...
anovan(MPG,{org cyl4 when},2,3,varnames);
>> termvec'
```

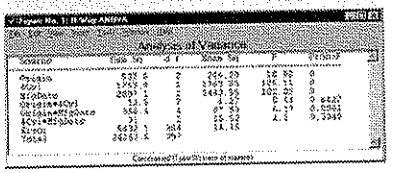
Now all terms are estimable. The *p*-values for interaction term 4 (`Origin*4Cyl`) and interaction term 6 (`4Cyl*MfgDate`) are much larger than a typical cutoff value of 0.05, indicating these terms are not significant. We could choose to omit these terms and pool their effects into the error term. The output `termvec` variable returns a vector of codes, each of which is a bit pattern representing a term. We 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:

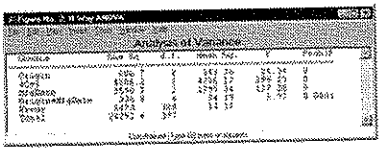
```
>> termvec([4 6]) = []
>> termvec
>> anovan( ...
MPG,{org cyl4 when},termvec,3,varnames)
```

Now we 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.

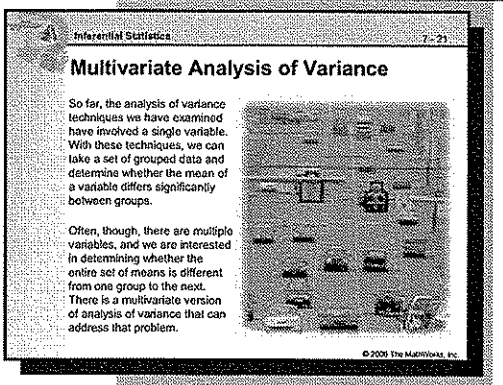
Try

```
>> [p,tbl, ...
stats,termvec] = ...
anovan( ...
MPG, ...
{org cyl4 when}, ...
2,3,varnames);
>> termvec'

>> termvec([4 6]) ...
= []
>> termvec
>> anovan( ...
MPG, ...
{org cyl4 when}, ...
termvec,3,varnames)
```





Multivariate Analysis of Variance



So far, the analysis of variance techniques we have examined have involved a single variable. With these techniques, we can take a set of grouped data and determine whether the mean of a variable differs significantly between groups.

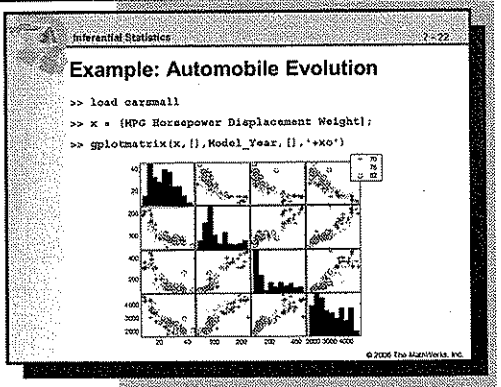
Often, though, there are multiple variables, and we are interested in determining whether the entire set of means is different from one group to the next. There is a multivariate version of analysis of variance that can address that problem.

The Statistics Toolbox includes the `manova1` function for One-Way Multivariate Analysis of Variance (MANOVA).

Try

```
>> doc manova1
```

Example: Automobile Evolution

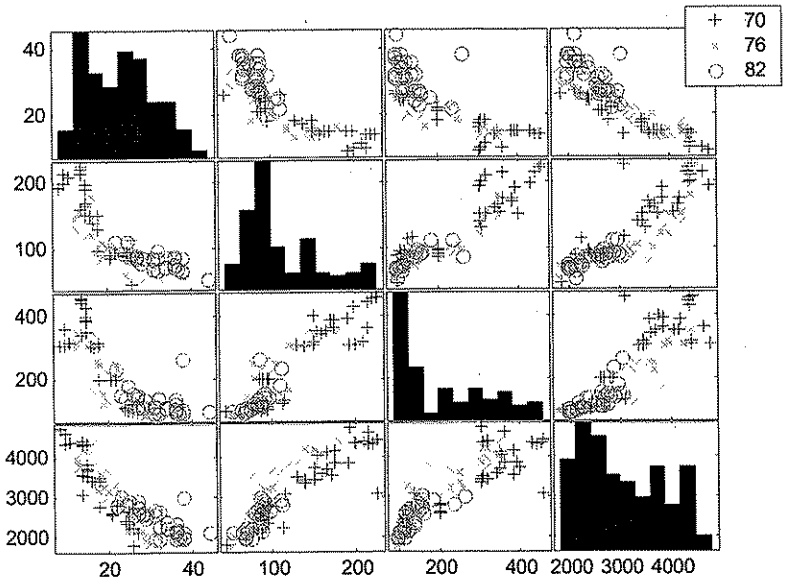


The `carsmall` data set has measurements on a variety of car models from the years 1970, 1976, and 1982. Suppose we are interested in whether the characteristics of the cars have changed over time. First we load the data:

```
>> load carsmall
```

Four of the variables in this MAT-file (`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. We can create a grouped plot matrix of these variables using the `gplotmatrix` function:

```
>> x = [MPG Horsepower Displacement Weight];
>> gplotmatrix(x,[],Model_Year,[],'+xo')
```

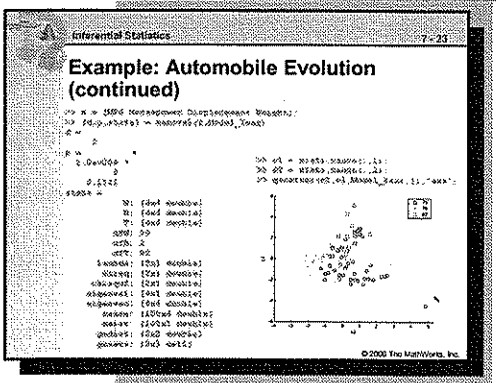


It appears the cars do differ from year to year. The upper right plot, for example, is a graph of `MPG` versus `Weight`. The 1982 cars appear to have higher mileage than the older cars, and they appear to weigh less on average. But as a group, are the three years significantly different from one another? The `manova1` function can answer that question.

Try

```
>> load carsmall
>> x = [MPG ...
Horsepower ...
Displacement ...
Weight];

>> gplotmatrix ...
(x,[],...
Model_Year,[],'+xo')
```

Example: Automobile Evolution (continued)




```
>> [d,p,stats] = manova1(x,Model_Year)
```

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.

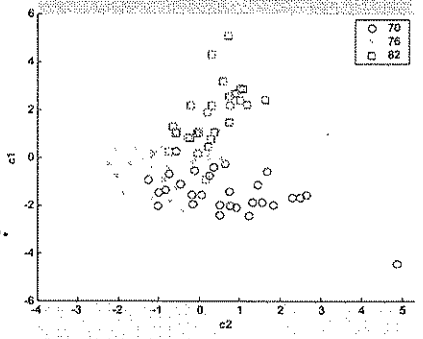
The third output, `stats`, is a structure containing several fields. These fields include

- Matrix analogs to the within, between, and total sums of squares in ordinary one-way analysis of variance.
- The ingredients of the test for the dimensionality of group means.
- Information used to perform a canonical analysis on the data.
- Two Mahalanobis distances: `mdist`, which measures the distance from each point to its group mean, and `gmdist`, which measures the distances between each pair of group means.

If we had more groups, we might have found it instructive to use the `manovacluster` function to draw a diagram that presents clusters of the groups, formed using the distances between their means.

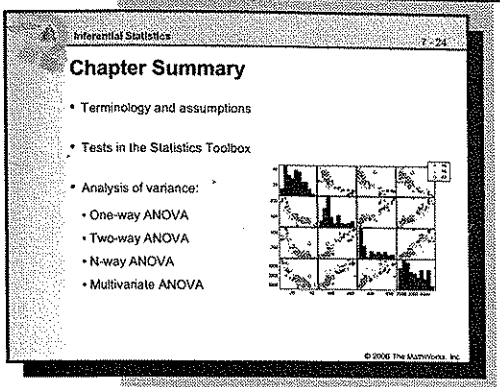
Try

```
>> [d,p,stats] = ...
manova1(x, ...
Model_Year)

>> c1 = ...
stats.canon(:,1);
>> c2 = ...
stats.canon(:,2);
>> gscatter( ...
c2,c1, ...
Model_Year, ...
[], ...
'oxs')
```



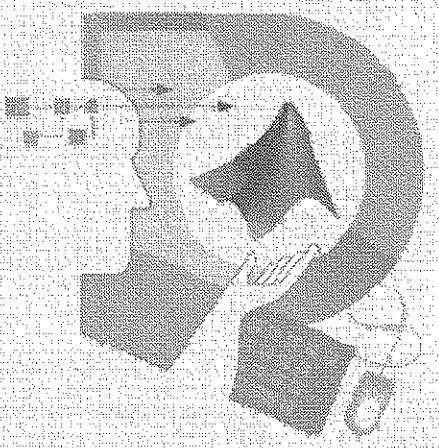
Chapter Summary

- Terminology and assumptions
- Tests in the Statistics Toolbox
- Analysis of variance:
  - One-way ANOVA
  - Two-way ANOVA
  - N-way ANOVA
  - Multivariate ANOVA



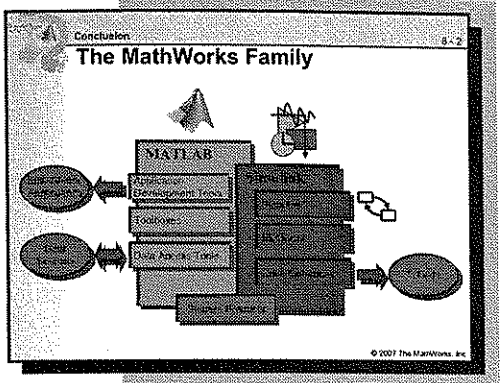
Statistical Methods in MATLAB®

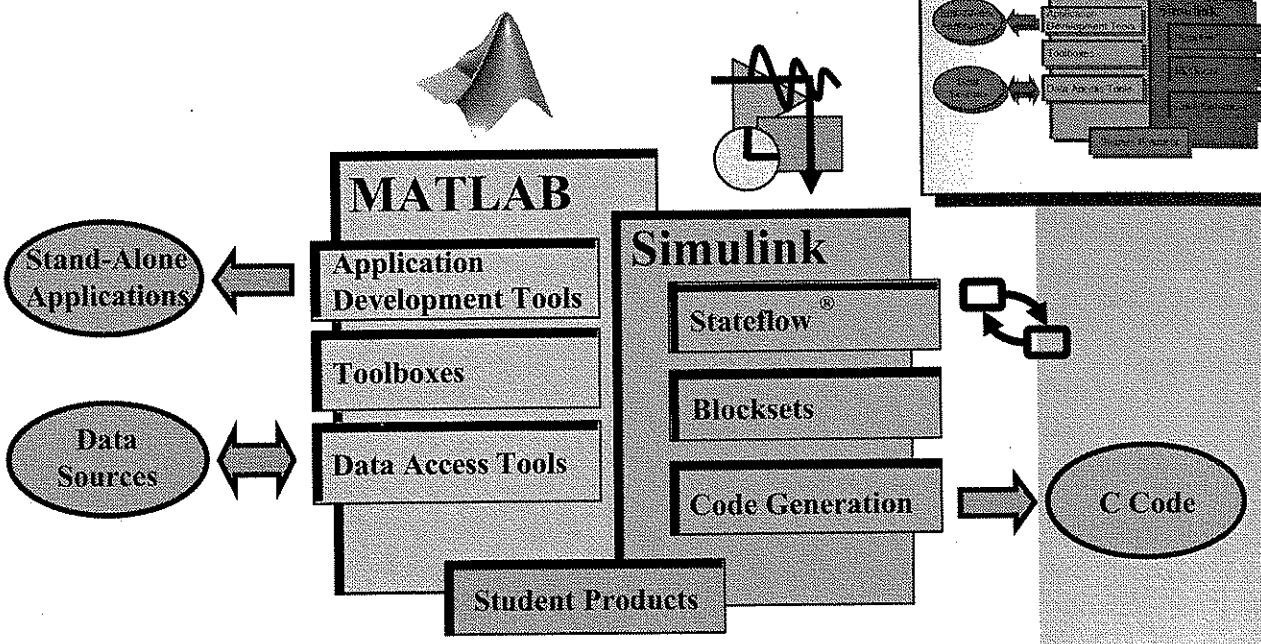
Conclusion



The MathWorks
Training Services

The MathWorks Family





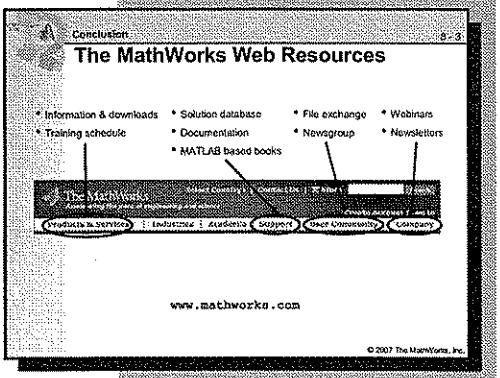
Key characteristics of the MATLAB language

- A user-friendly, intuitive syntax, favoring brevity and simplicity without compromising intelligibility
- The highest quality numerical algorithms, based on close historical ties with the numerical analysis research community
- Powerful, easy-to-use graphics and visualization capabilities
- A high-level language, making it possible to carry out computations in a line or two that would require hundreds of lines of code in languages such as Fortran or C
- Easy extensibility, by the user or via packages of application-specific M-files and GUIs known as toolboxes
- Real and complex vectors and matrices (including sparse matrices) as fundamental data types

Key Characteristics of Simulink

- A complete environment for modeling, simulating, and implementing dynamic and embedded systems
- Design and test linear, nonlinear, discrete-time, continuous-time, hybrid, and multirate systems
- Applications in controls, DSP, communications, and systems engineering
- Open architecture allows integration of models from other environments

The MathWorks Web Resources



The MathWorks Web site at `www.mathworks.com` contains a wealth of resources beyond the materials provided for this course.*

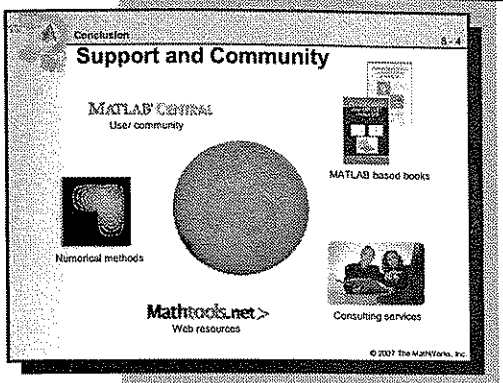
- Information on products and downloads
  `www.mathworks.com/products`
- A searchable tech support database
  `www.mathworks.com/support`
- Live and recorded Webinars on MathWorks tools and applications
  `www.mathworks.com/company/events`
- MATLAB Central file exchange, newsgroups, and contests
  `www.mathworks.com/matlabcentral`
- MATLAB based books
  `www.mathworks.com/support/books`
- Numerical Computing with MATLAB
  `www.mathworks.com/moler`
- MATLAB newsletters
  `www.mathworks.com/company/newsletters`
- The MathWorks consulting services
  `www.mathworks.com/consulting`
- Up-to-date information on training courses, dates, and locations
  `www.mathworks.com/training`
- Complete product documentation
  `www.mathworks.com/access/helpdesk/help/helpdesk.shtml`

Try

`www.mathworks.com`

* For international Web sites, see the Introduction.

Support and Community

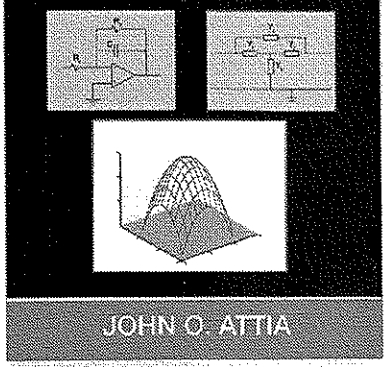


You are connected to a world of creative MATLAB users, both inside and outside of The MathWorks, for information, support, and community.

MATLAB Central* provides a single location for users of MathWorks products to exchange information directly with each other in forums specific to dozens of application areas. A file exchange contains thousands of user-created extensions of MathWorks products with helpful ideas to get you started on your next project. A newsgroup, `comp.soft-sys.matlab`, is read by thousands of users worldwide, including people from the MathWorks development and technical support departments.

Mathtools.net (`www.mathtools.net`) provides links to technical computing resources on MATLAB, programming, applications, industries, and education.

MATLAB based books* include more than 700 books in 20 languages. The texts present theory, real-world examples, and exercises using MATLAB, Simulink, and other MathWorks products. They provide reference for researchers in academia and industry, tools for practicing engineers, and course material for instructors in engineering, science, and mathematics.



MATLAB based books are available in a wide variety of application areas.

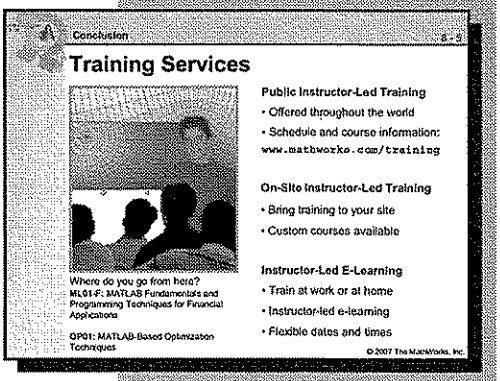
Numerical Computing with MATLAB*, written by MATLAB creator Cleve Moler, is a Web-based text explaining the numerical methods behind MATLAB. It provides more than 70 M-files and more than 200 exercises. Those adopting the textbook for a course can register for access to curriculum tools and materials, including a solutions manual and a set of slides for use in classroom lectures.

MathWorks consulting services* provides specific, project-based services including startup services, application development, model-based and system-level design, embedded-systems development, enterprise-wide integration, and product migration.

---

* See previous page for URL.

Training Services



MathWorks Training Services can help you use our products to succeed in your work. Our training courses are developed around the core responsibilities of engineers, scientists, and educators. Our trainers focus on your goals and how to use MathWorks tools to achieve them.

The MathWorks offers introductory and intermediate courses in MATLAB, Simulink, and Stateflow, as well as advanced courses in subjects such as control design, signal and image processing, test and measurement, optimization, statistics, and financial analysis.

Public instructor-led training
Public instructor-led courses are offered in North America, Europe, and Asia. In addition, many of our distributors offer training at other international sites. So, whether you are in Winnipeg or Wien, you can find classroom training near you.

On-site instructor-led training
The MathWorks also offers custom training courses, which can be taught at your own facility. Our engineers can incorporate company- or industry-specific examples into a curriculum of your choosing.

Instructor-led e-learning
All of our training courses (except those requiring specialized hardware) are also offered over the Web. An instructor in one of our offices can share his/her desktop with you and communicate over the phone. You get the same quality instruction with the convenience of being in your home or office. Our e-learning courses are also typically run with fewer students, on a more flexible schedule.

Where do you go from here?

Related courses

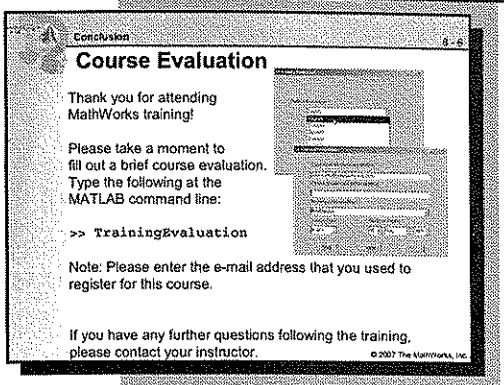
OP01: MATLAB-Based Optimization Techniques

ML01-F: MATLAB Fundamentals and Programming Techniques for Financial Applications

For course information and the latest training schedule and locations:
`www.mathworks.com/training`

Don't see it on the schedule? Contact us and we'll make it happen:
`training@mathworks.com`

Course Evaluation



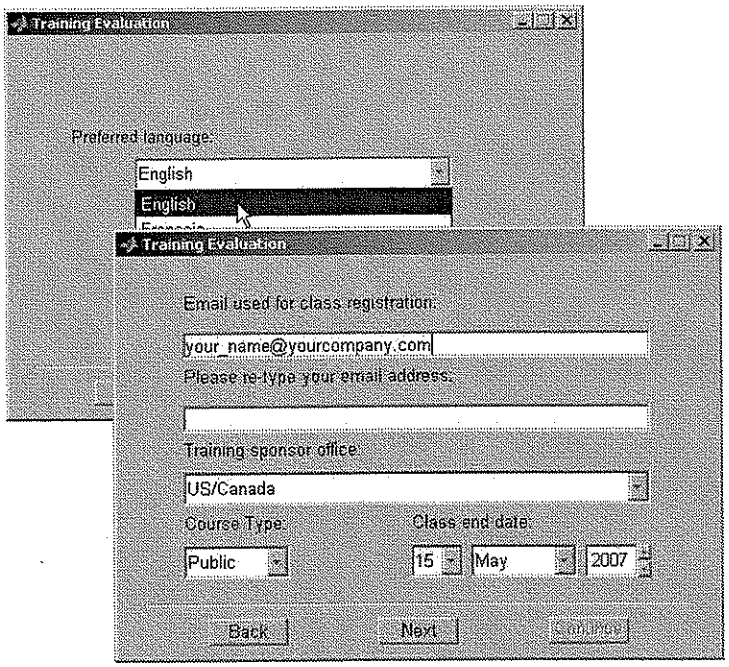
Thank you for attending MathWorks training!

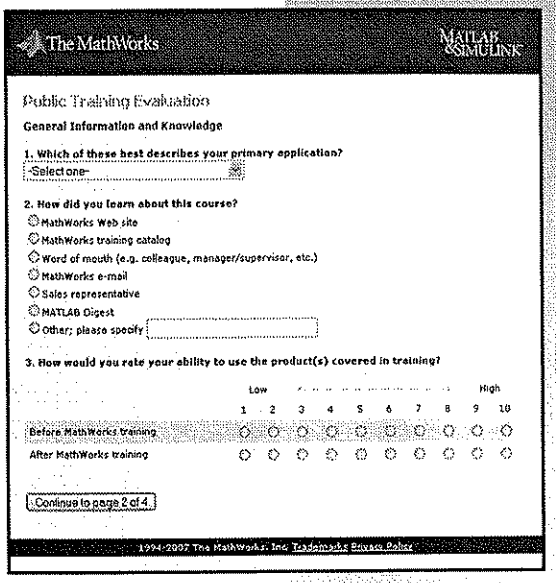
Please take a moment to fill out a brief course evaluation.

Type the following at the MATLAB command line:

```
>> TrainingEvaluation
```

Note: Please enter the e-mail address that you used to register for this course.





If you have any further questions following the training, please contact your instructor.

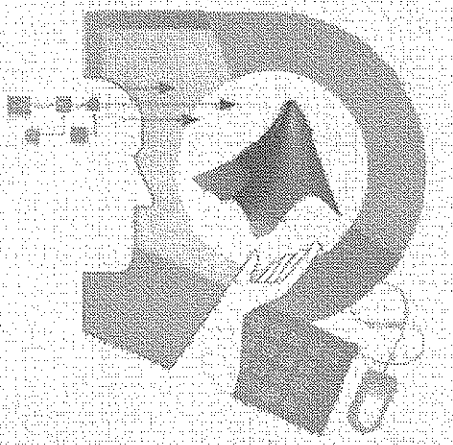
If needed, an alternate method of launching the evaluation is through the following URL:

http://www.mathworks.com/services/training/eval

The following is the information you will need to fill in the form at the above URL:
Training Type: Public, or On-Site
Preferred Language: EN (English), FR (French), DE (German), IT (Italy), KO (Korean), ES (Spanish), ZH (Simplified Chinese) Training Office: AU (Australia), BNL (Benelux), CH (Switzerland), CN (China), DE (Germany), FR (France), IT (Italy), KR (Korea), Natick (North America), Pan-Euro (Pan-European), SE (Nordic), UK (United Kingdom)
Date Format: YYYY-MM-DD
Email Address: [email address you used to register for the course]

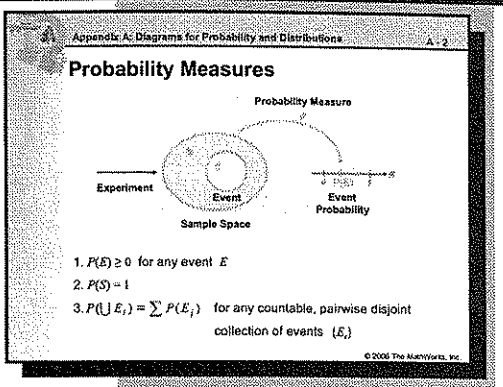
Statistical Methods in MATLAB®

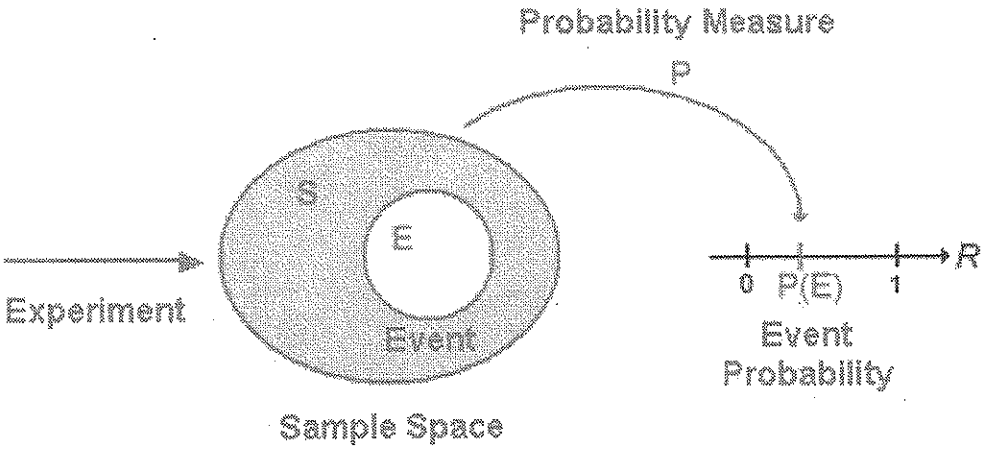
Appendix A: Diagrams for Probability and Distributions



The MathWorks Training Services

Probability Measures



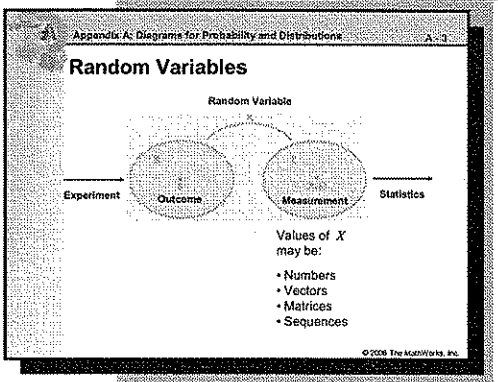


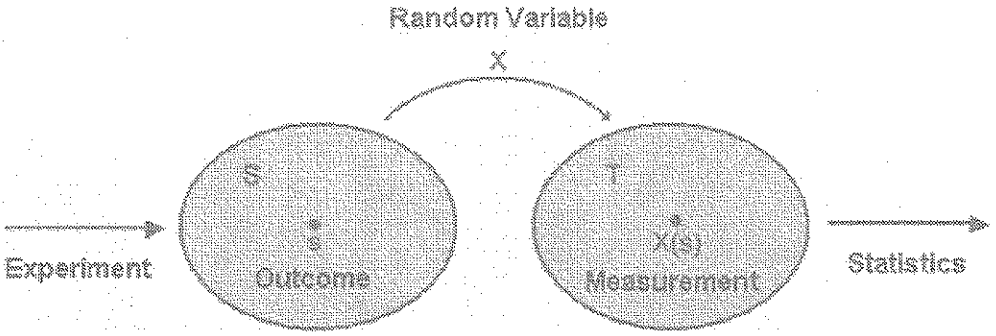
1. $P(E) \geq 0$ for any event E

2. $P(S) = 1$

3. $P(\bigcup E_i) = \sum P(E_i)$ for any countable, pairwise disjoint collection of events $\{E_i\}$

Random Variables

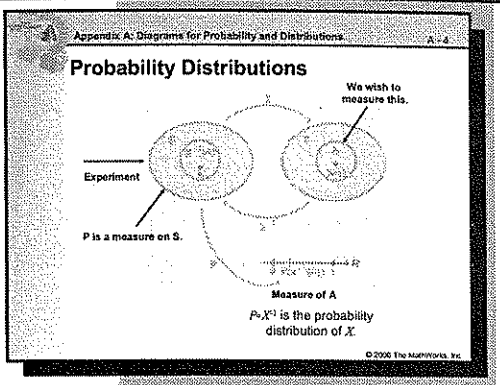


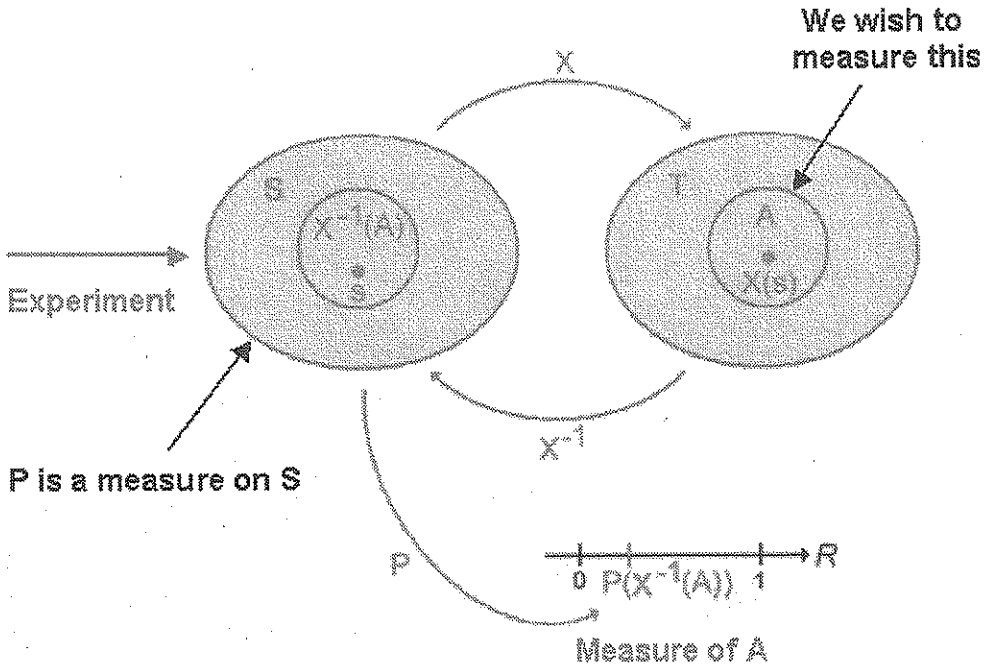


Values of X may be:

- Numbers
- Vectors
- Matrices
- Sequences

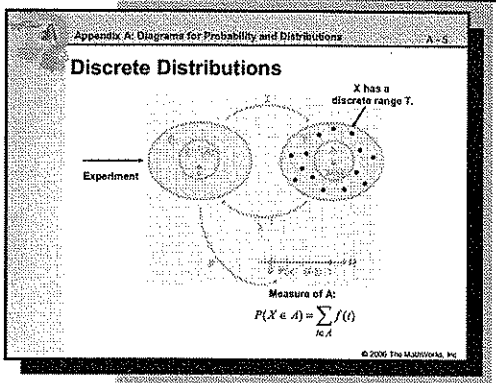
Probability Distributions

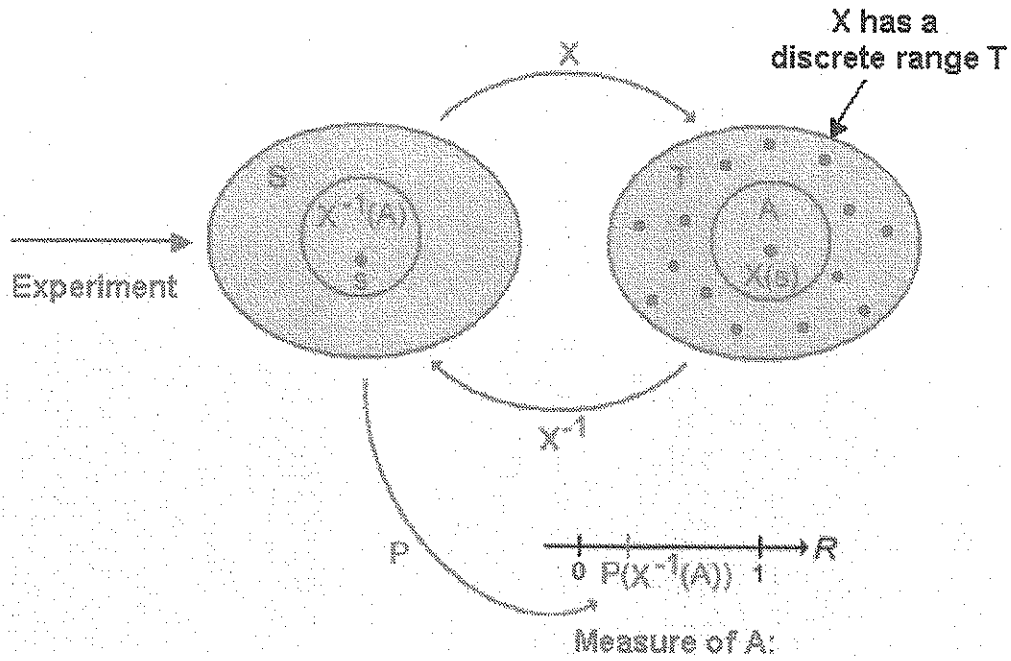




$P \circ X^{-1}$ is the probability distribution of X

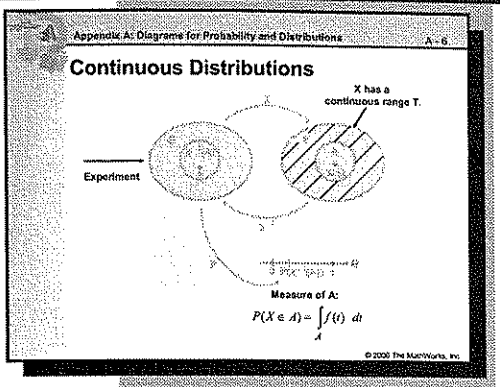
Discrete Distributions

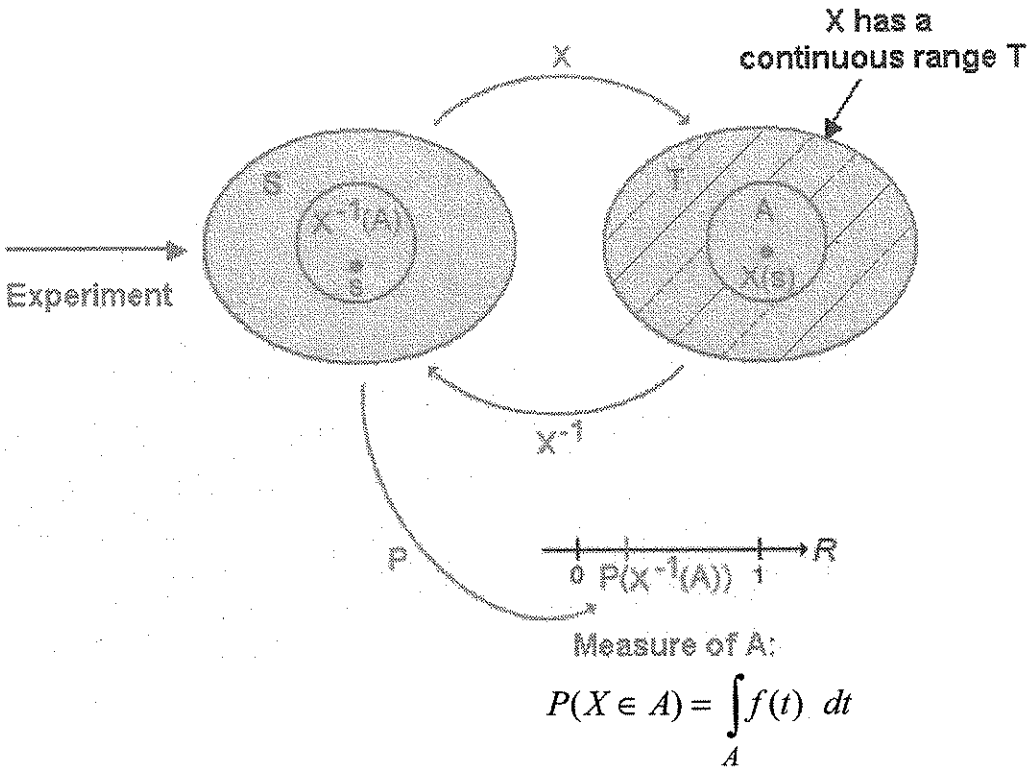




$$P(X \in A) = \sum_{t \in A} f(t)$$

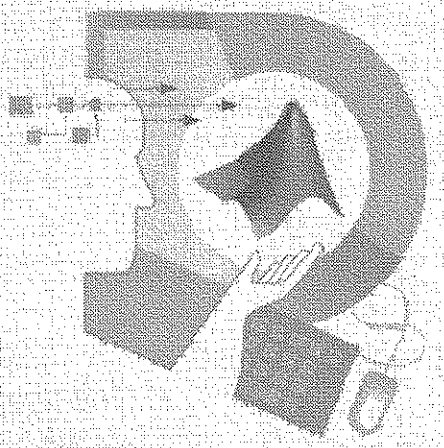
Continuous Distributions





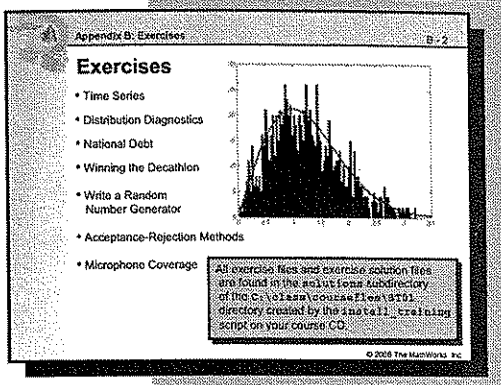
Statistical Methods in MATLAB®

Appendix B: Exercises



The MathWorks
Training Services

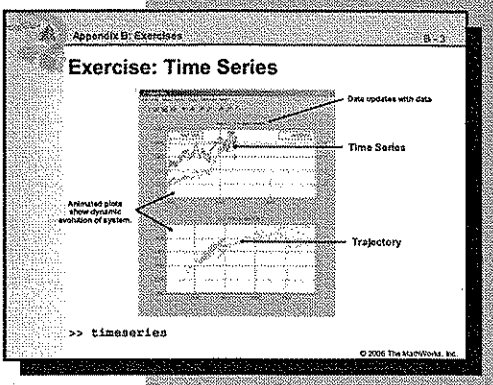
Exercises



Exercises are referenced in the notes with the icon. Exercises correspond to particular sections of the course, and allow you to practice the techniques of that section. Exercises may also require you to bring together material from earlier in the section, or from previous sections.

All exercise files and exercise solution files are found in the `solutions` subdirectory of the `C:\class\coursefles\ST01` directory created by the `install_training` script on your course CD.

Exercise: Time Series



Many data sets involve variables that depend on time.

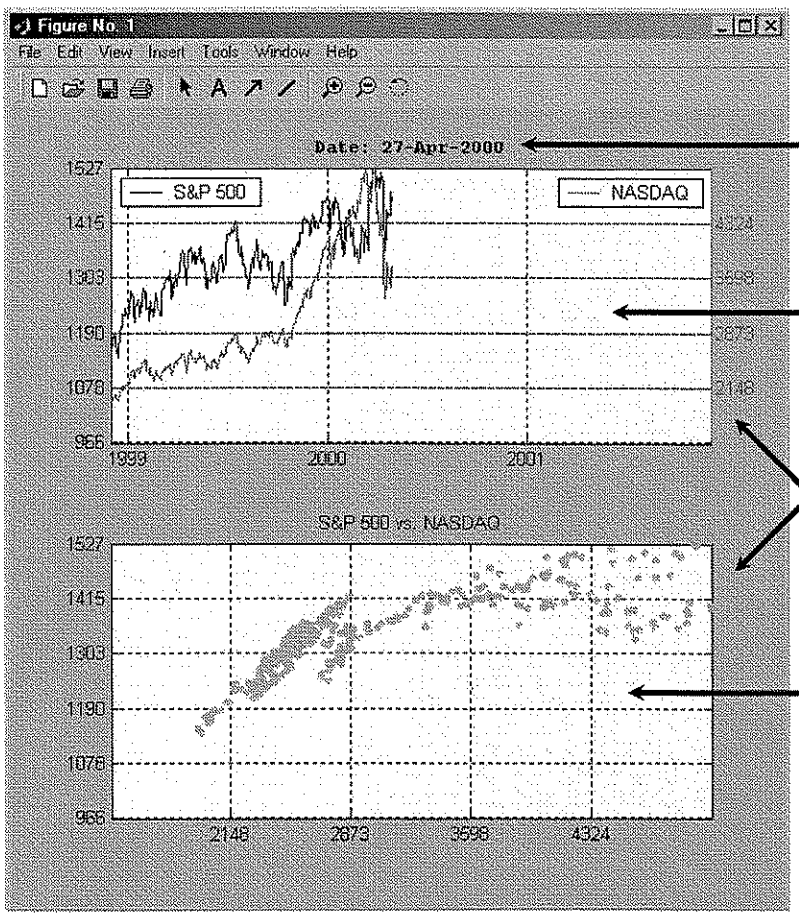
In this exercise, we will look at two statistical plot types used in the analysis of time-dependent variables: *time series plots* and *trajectory plots*. These plots arise in the study of time-varying or *dynamical systems*, to visualize underlying processes that may be described through differential equations or (for discrete time variables) difference equations. Sequences of data values measured over time are often called *stochastic processes*.

This exercise practices the use of the statistical plotting routines covered in this chapter and reviews the use of M-file scripts and Handle Graphics.

The exercise is to use the data on the S&P 500 and NASDAQ indices to reproduce, in part or in whole, the display below.

Try

```
>> timeseries
```



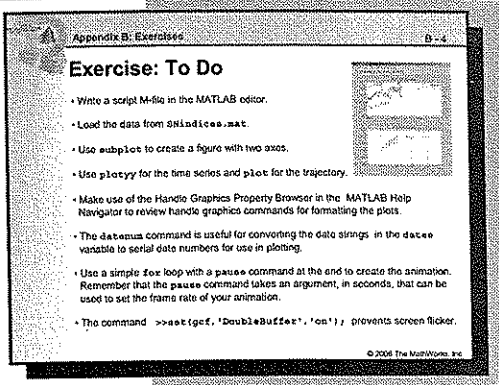
Date updates with data

Time series

Animated plots show dynamic evolution of system.

Trajectory

Exercise: To Do



Remember that there are many approaches to the solution of these exercises. Try not to look at the solution until you have gone through some trial and error on your own. If you don't know how to do something, browse the documentation. Begin with some small part of the problem, and work toward a complete solution one step at a time.

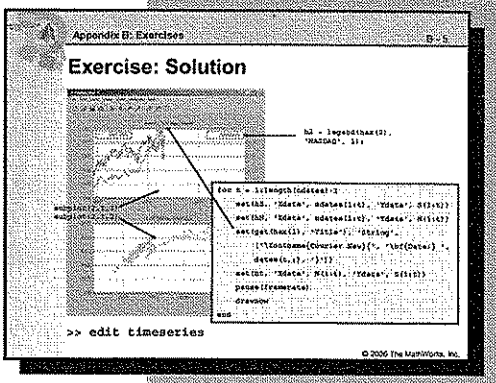
Some hints:

- Write a script M-file in the MATLAB editor.
- Load the data from `SNindices.mat`.
- Use `subplot` to create a figure with two axes.
- Use `plotyy` for the time series and `plot` for the trajectory. (The function `gscatter` will write to a new figure window, not the subplot, unless you copy and edit its function M-file.)
- Make use of the Handle Graphics® Property Browser in the MATLAB Help Navigator to review Handle Graphics commands for formatting the plots.
- The `datenum` command is useful for converting the date strings in the `dates` variable to serial date numbers for use in plotting.
- Use a simple `for`-loop with a `pause` command at the end to create the animation. Remember that the `pause` command takes an argument, in seconds, that can be used to set the frame rate of your animation. (For animations involving more demanding calculations, you may want to look at the `getframe` and `movie` commands in MATLAB.)
- The command

```
>> set(gcf,'DoubleBuffer','on');
```

prevents screen flicker.

Exercise: Solution



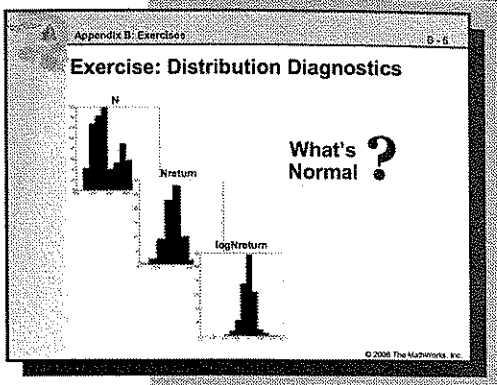
Here are the contents of the script M-file `timeseries.m`.

Try

```
>> edit timeseries
```

```
% Time series and trajectory animation of the
% S&P 500 and NASDAQ data in the file SNindices.mat

% Set frame rate for playback:
framerate = 0.01;

load SNindices

Smin = min(S); Smax = max(S);
Nmin = min(N); Nmax = max(N);
ndates = datenum(dates);
dmin = min(ndates); dmax = max(ndates);

subplot(2, 1, 1)
% Plot first point from each data set and set axes:
[hax, hS, hN] = plotyy(ndates(1), S(1), ndates(1), N(1));

set(hax(1), 'XGrid', 'on', 'YGrid', 'on')
set(hax(2), 'XGrid', 'on', 'YGrid', 'on')
set(hax(1), 'XLim', [dmin dmax], 'YLim', [Smin Smax])
set(hax(2), 'XLim', [dmin dmax], 'YLim', [Nmin Nmax])
set(hax(1), 'XTick', [datenum('01-Jan-1999'):365:datenum('01-Jan-2001')])
set(hax(2), 'XTick', [datenum('01-Jan-1999'):365:datenum('01-Jan-2001')])
set(hax(1), 'XTickLabel', {'1999'; '2000'; '2001'})
set(hax(2), 'XTickLabel', {'1999'; '2000'; '2001'})
set(hax(1), 'YTick', round([Smin:(Smax-Smin)/5:Smax]))
set(hax(2), 'YTick', round([Nmin:(Nmax-Nmin)/5:Nmax]))
h1 = legend(hax(1), 'S&P 500', 2);
set(h1, 'Color', 'w')
h2 = legend(hax(2), 'NASDAQ', 1);
set(h2, 'Color', 'w')

subplot(2, 1, 2)
% Plot first point and set axes:
ht = plot(N(1), S(1), 'co', 'MarkerFaceColor', 'c', 'MarkerSize', 2);

set(gca, 'XGrid', 'on', 'YGrid', 'on')
set(gca, 'XLim', [Nmin Nmax], 'YLim', [Smin Smax])
set(gca, 'XTick', round([Nmin:(Nmax-Nmin)/5:Nmax]))
set(gca, 'YTick', round([Smin:(Smax-Smin)/5:Smax]))
title('S&P 500 vs. NASDAQ')

% Prevent flickering in display:
set(gcf, 'DoubleBuffer', 'on');

% Keep axes settings and display initial segments of data in order:
for t = 1:length(ndates)-1
    set(hS, 'Xdata', ndates(1:t), 'Ydata', S(1:t))
    set(hN, 'Xdata', ndates(1:t), 'Ydata', N(1:t))
    set(get(hax(1), 'Title'), 'String', {'\fontname{Courier New}{', '\bf{Date:} ', dates(t,:), '}'})
    set(ht, 'Xdata', N(1:t), 'Ydata', S(1:t))
    pause(framerate)
    drawnow
end

clear h1 h2 hS hN hax ht Smin Smax Nmin Nmax dmin dmax t ndates framerate
```

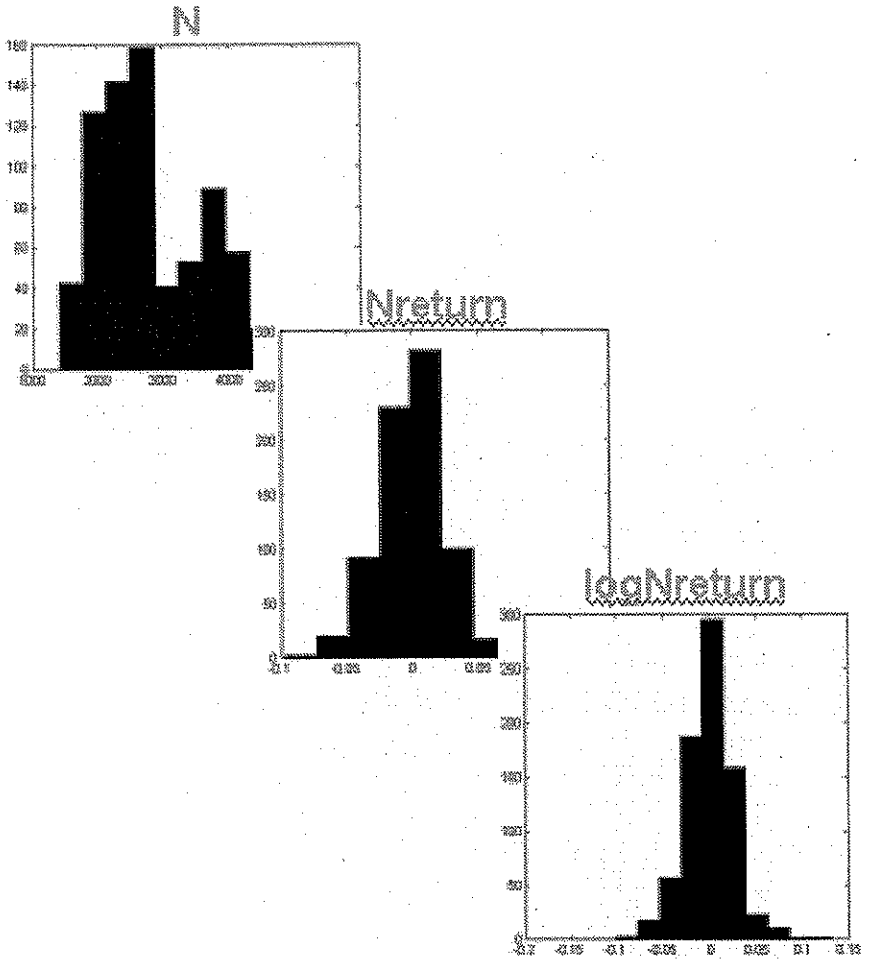
Exercise: Distribution Diagnostics



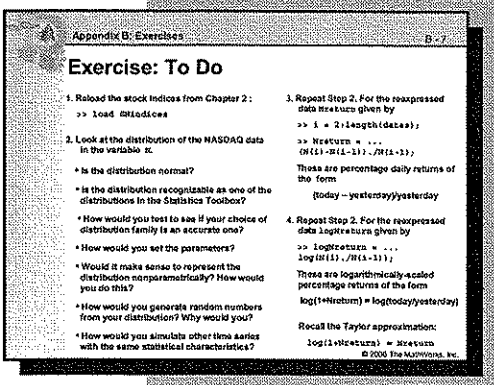
In this exercise we return to the stock indices from Chapter 2 and consider how different reexpressions of the data are distributed. The goal will be to find a reexpression that leads to a random variable with a normal distribution.

The advantage of working with normally distributed data is twofold:

- You are probably familiar with normal distributions and have some intuition about the behavior of normally distributed variables.
- The Statistics Toolbox has more functions for working with normal distributions and their diagnostics than for any other distribution.



Exercise: To Do



1. Reload the stock indices from Chapter 2 by typing

```
>> load SNindices
```

2. Look at the distribution of the NASDAQ data in the variable `N`.

- Is the distribution normal?
- Is the distribution recognizable as one of the supported distributions in the Statistics Toolbox?
- How would you test to see if your choice of distribution family is an accurate one?
- How would you set the parameters?
- Would it make sense to represent the distribution nonparametrically? How would you do this?
- How would you generate random numbers from your distribution? Why would you do this?
- How would you simulate other time series with the same statistical characteristics?

3. Repeat Step 2. For the reexpressed data `Nreturn` given by

```
>> i = 2:length(dates);
>> Nreturn = (N(i)-N(i-1))./N(i-1);
```

These are percentage daily returns of the form

(today – yesterday)/yesterday

4. Repeat Step 2. For the reexpressed data `logNreturn` given by

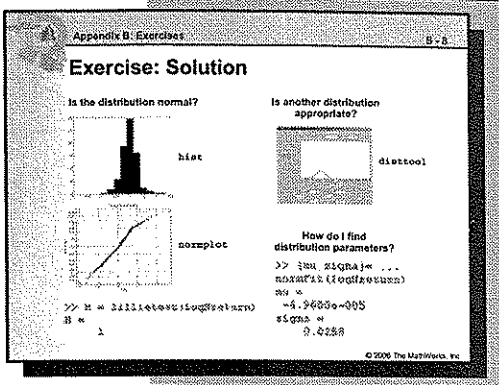
```
>> logNreturn = log(N(i)./N(i-1));
```

These are logarithmically-scaled percentage returns of the form

$$\log(1 + \texttt{Nreturn}) = \log(\text{today}/\text{yesterday})$$

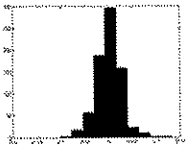
Recall the Taylor approximation: $\log(1 + \texttt{Nreturn}) \approx \texttt{Nreturn}$.

Exercise: Solution

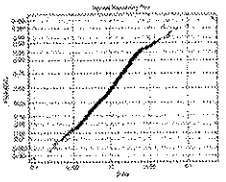


We discuss solution approaches for the log returns in Step 4. The raw NASDAQ data and the raw return data in Steps 2 and 3. may be treated similarly.

```
>> hist(logNreturn)
```
produces a normal-looking distribution.



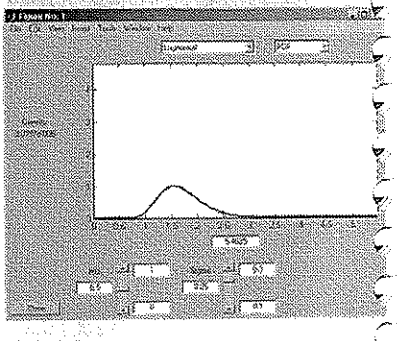
```
>> normplot(logNreturn)
```
also indicates a normal-looking distribution, though noncharacteristically fattened tails become apparent (the distribution is *leptokurtotic*).



```
>> jbtest(logNreturn)
>> kstest(logNreturn)
>> lillietest(logNreturn)
```
all reject the hypothesis that the distribution is normal at the default 5% significance level.

```
>> H = lillietest(logNreturn)
H =
     1
```

Many of the supported distributions in `disttool` have normal-looking distributions, though none of them is based on assumptions that strongly suggest the construction of `logNreturn`.
Note that lognormal distributions, which may come to mind, are appropriate only for strictly positive random variables. Our return series `Nreturn` can take on values between –1 and +∞, and one of the reasons for working with `logNreturn` is that the reexpressed variable can take on values over the entire range from – ∞ to +∞.

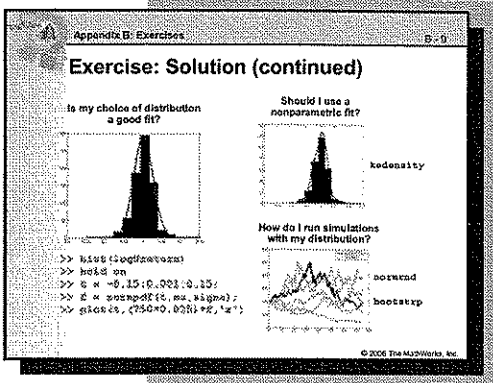


For definiteness, we proceed under the assumption that the returns and log returns are normally distributed, with the caveat that this assumption only applies to return values near the mean.

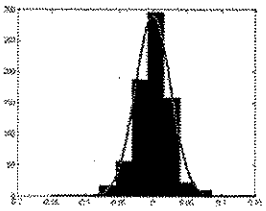
```
>> [mu,sigma] = normfit(logNreturn)
```
returns estimators for the distribution's parameters.

```
>> [mu,sigma]= ...
normfit(logNreturn)
mu =
 -4.9600e-005
sigma =
    0.0258
```

Exercise: Solution (continued)

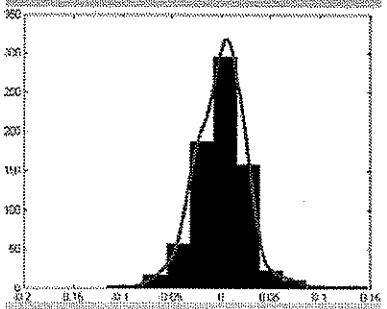



```
>> hist(logNreturn)
>> hold on
>> t = -0.15:.001:0.15;
>> f = normpdf(t,mu,sigma);
>> plot(t,(750*0.025)*f,'r')
```



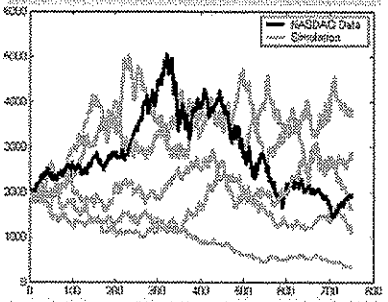
show the fitted normal distribution and the sampling distribution together. We scale the pdf so that its area matches the area of the histogram: (number of data points)*(bin size).

Given the imperfections of our normal model, and our inability to identify other likely parametric models from the toolbox, a nonparametric pdf might be a good choice here. This could be constructed using `ksdensity`, as on p. 3-14. We complete the solution, however, under the continuing assumption of normality.



We could generate random numbers from our distribution with the `normrnd` command. This would be one approach to producing simulations of the time series over the same time period. The script `randsim` shows the general approach:

```
>> edit randsim
>> randsim (repeatedly)
```

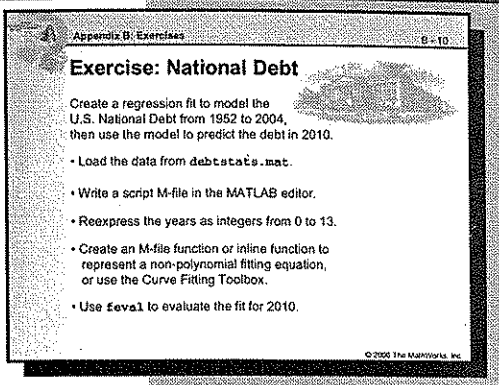


The script uses the approximation `logNreturn` ≈ `Nreturn` from Step 4 at the bottom of p. 3-18, then uses the formula for `Nreturn` in Step 3 to solve for `N(i)`. Values for `logNreturn` are replaced with random values generated from the fitted distribution.

Another approach to simulation is to use bootstrapping from the data on `logNreturn`. Rather than generating random numbers from the fitted distribution, we randomly select values from the existing sample. A simple version of this approach is explored in the script `bootsim`:

```
>> edit bootsim
>> bootsim (repeatedly)
```

Exercise: National Debt



The file `debtstats.mat` contains estimates of the U.S. National Debt for every 4 years from 1952 to 2004. When loaded, it creates workspace variables for the years and the debt for that year, in billions.

This exercise is fairly open ended; the challenge is to model the data given here with one or more regressions.

As a final task, you are asked to use your model (or models) to predict the U.S. National Debt for the year 2010.

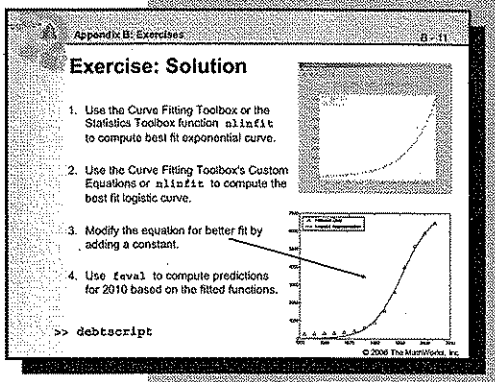
Some hints:

- Load the data from `debtstats.mat`.
- Reexpress the years so that they are integers from 0 to 13.
- The equation for an exponential model is given by $y = Ae^{Bx}$.
- The equation for a logistic model is given by $y = \frac{A}{1+e^{(Bx+C)}}$.
- For a polynomial fit, use the Basic Fitting Tool, `polytool`, or `polyfit`.
- For a nonlinear fit, use the Curve Fitting Toolbox, `nlintool`, or `nlinfit`.
- When using `nlintool` or `nlinfit`, you must first create the fitting function using an M-file or the `inline` function, and provide an initial guess for the parameters in the optimization.
- When using the Curve Fitting Toolbox, try the supported models before creating a custom equation.
- To check whether a logarithmic transformation is useful, try plotting the data using `semilogx`, `semilogy`, or `loglog`.
- Use `feval` to evaluate the models for the year 2010, after making appropriate inverse reexpressions.

Try

```
>> load debtstats
```

Exercise: Solution



Here are the contents of the script M-file `debtscript.m`:

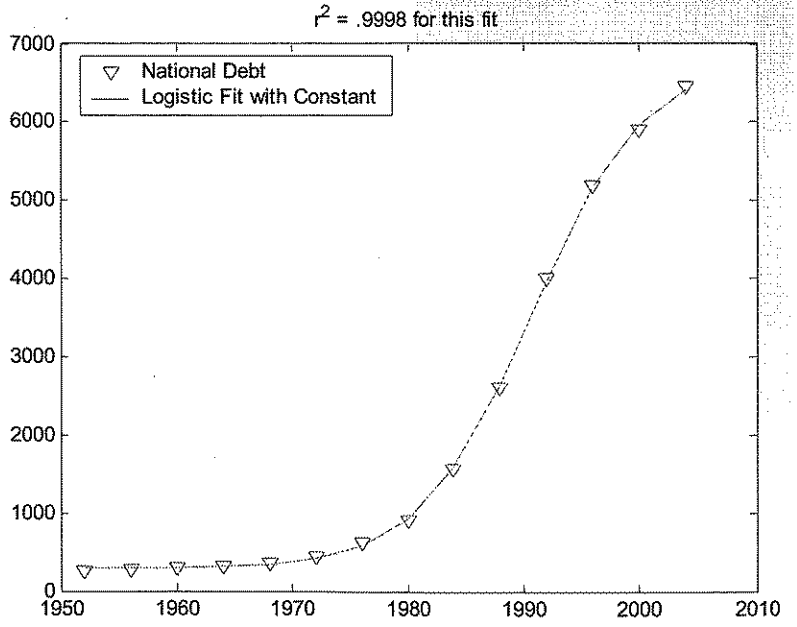
```
% Script to work with U.S. National Debt statistics
clear all
load debtstats

% Reexpress year data, make it 0 to 13.
ryear = (yr - 1952) / 4;

% Generate exponential fit
xfun = inline('b(1).*exp(b(2).*x)','b','x');
[bx,rx] = nlinfit(ryear,debt,xfun,[debt(1) .2]);
yxfit = feval(xfun,bx,ryear);
plot(yr,debt,'*',yr,yxfit,'r-')

% Generate logistic fit
figure
lfun = inline('b(1)./(1 + exp((b(2).*x) + b(3)))','b','x');
[bl,rl] = nlinfit(ryear,debt,lfun,[1e4 -1 6]);
ylfit = feval(lfun,bl,ryear);
plot(yr,debt,'^',yr,ylfit,'k-')

% Logistic fit seems off; try logistic fit with added constant
figure
lcfun = inline('b(1)./(1 + exp((b(2).*x) + b(3))) +
b(4)','b','x');
[bcl,rcl] = nlinfit(ryear,debt,lcfun,[1e4 -1 6 250]);
ylcfit = feval(lcfun,bcl,ryear);
plot(yr,debt,'v',yr,ylcfit,'m-')

% Use fits to predict debt in 2010.  Reexpress first.

predyr = 2010;
rpred = (predyr - 1952) / 4;

pxfit = feval(xfun,bx,rpred)
plfit = feval(lfun,bl,rpred)
plcfit = feval(lcfun,bcl,rpred)
```

Try

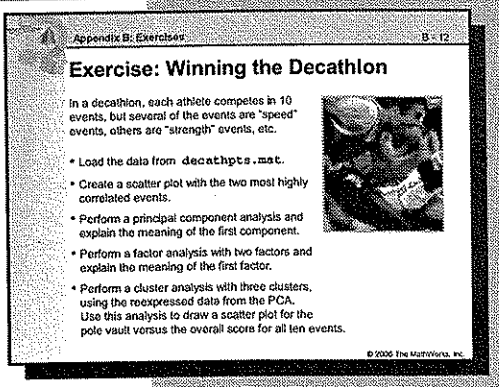
```
>> edit debtscript
>> debtscript
```

The third fit is shown at right, and does a nice job of fitting the data.

This task can also be accomplished using the models of the Curve Fitting Toolbox and its `cftool` GUI.

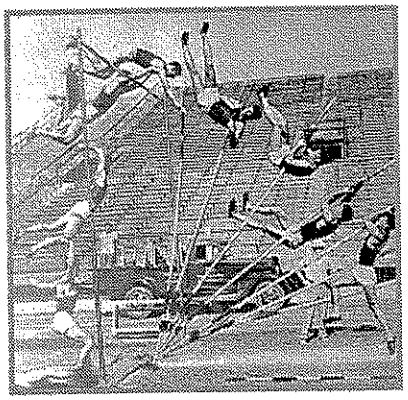


Exercise: Winning the Decathlon



The file `decathpts.mat` contains scores for the 10 events of the decathlon for the top 86 decathletes from January through October 2002. The events are, in order

- 100 meter dash
- Long jump
- Shot put
- High jump
- 400 meter dash
- 110 meter hurdles
- Discus throw
- Pole vault
- Javelin throw
- 1500 meter dash



Each row of the matrix `scores` represents a decathlete, while each column represents a particular event.

Each type of analysis performed in this chapter can be applied to the decathalon data.

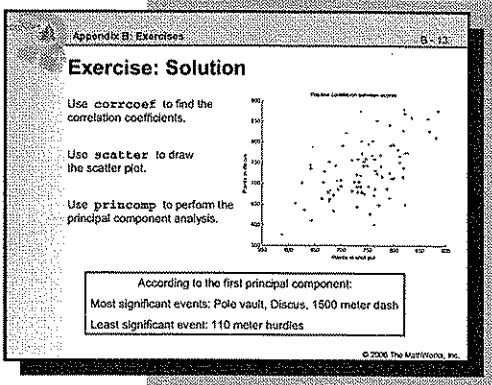
Some things to try:

- Using `corrcoef(scores)`, find the pair of events with the highest positive correlation and the pair with the highest negative correlation, then plot the data for these events with a scatter plot.
- Perform a principal component analysis on the data. Based on the first principal component, which events are most correlated and may be measuring similar physical abilities?
- Perform a factor analysis with two factors on the data and no rotation. Which events are positively associated in the loads for the first factor, and which are negatively associated?
- Perform a cluster analysis on the data, using three clusters. What happens when you try a hierarchical clustering? What happens when you try a K-means clustering? Create a scatter plot with the results from the pole vault and the total score for all ten events.

Try

```
>> load decathpts
```

Exercise: Solution



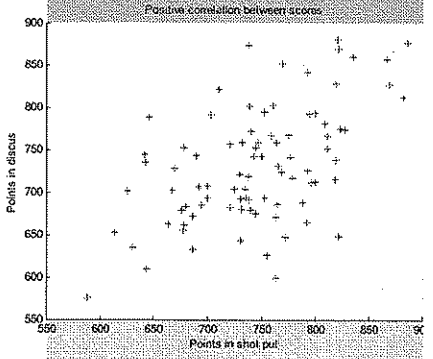
The script M-file `decathscript.m` contains the full solution, including the conversion formulas to take raw time and distance data and convert to decathlon point scores.

Here are the solutions to the presented tasks from the script:

```
% Load data and calculate total scores
load decathpts
totals = sum(scores,2);

% Find events with maximum correlation
corrs = corrcoef(scores);
corrs = triu(corrs,1);  % Above diagonal
[vals,event1] = max(corrs);
[val,event2] = max(vals);
event1 = event1(event2);

% 2-D scatter plot of shot put versus discus throw
scatter(scores(:,event1),scores(:,event2),'+');
xlabel('Points in shot put')
ylabel('Points in discus')
title('Positive correlation between scores')

% Principal Component Analysis
[pcs,newdata,var,t2] = princomp(scores);
pcs(:,1) % Display first component
```

Try

```
>> decathscript
```



The first principal component indicates that the events that distinguish decathletes from one another are the pole vault (event number 8) and the discus throw (event number 7). These events have the highest component scores. The next-most-distinguishing event is the 1500 meter dash. The least distinguishing event, according to the first principal component, is the 110 meter hurdles. This event also has the smallest standard deviation in score, which can be seen by typing `std(scores)`. One interesting observation is that this analysis suggests that the ten events in the decathlon are *not* equally weighted in their contribution to the final score.

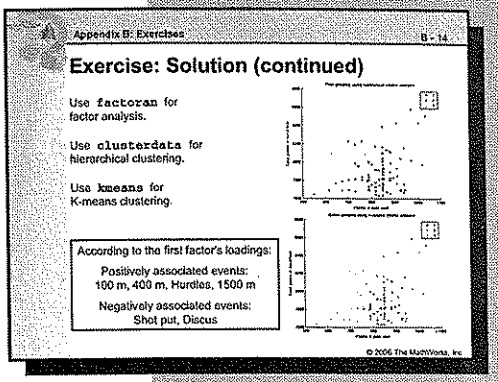
Another way to visualize this is to use the `plotmatrix` command, which displays histograms for each column as well as scatter plots for pairs of columns:

```
>> plotmatrix(scores)
```

Try

```
>>
plotmatrix(scores)
```

Exercise: Solution (continued)




```
% Factor Analysis
[Loadings,specVar,rot,stats,preds] = ...
factoran(scores,2,'rot','none');
Loadings(:,1) % Display first factor
```

The first factor indicates several events which are associated with one another. Four events have a high positive association: the 100 meter dash (event number 1), the 400 meter dash (event number 5), the 110 meter hurdles (event number 6), and the 1500 meter dash (event number 10). Two events have a negative association with this factor: the shot put (event number 3) and the discus throw (event number 7). This suggests that "speed" is the most distinguishing factor that can be found from the data.

```
% Cluster Analysis - try hierarchical first
Ghier = clusterdata(newdata,3);
figure
gscatter(scores(:,8),totals,Ghier,'brg','+x*')
xlabel('Points in pole vault')
ylabel('Total points in decathlon')
title('Poor grouping using hierarchical clusters')
```

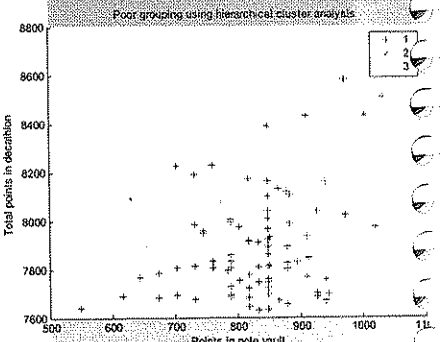
Hierarchical cluster analysis is very sensitive to outliers. In this case, two outliers generate their own clusters while the remaining data is gathered into a single cluster. For large data sets with outliers, K-means clustering is highly recommended.

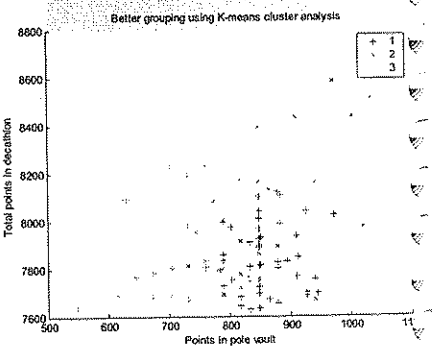
```
% Cluster Analysis - now try K-means
Gkmns = kmeans(newdata,3,'replicates',10);
figure
gscatter(scores(:,8),totals,Gkmns,'brg','+x*')
xlabel('Points in pole vault')
ylabel('Total points in decathlon')
title('Better grouping using K-means cluster analysis')
```

K-means clustering does a nice job of clustering the data. Answers vary due to the initial random assignment of data to clusters in `kmeans`, but 11 of the top 13 overall scores are placed in the same cluster most of the time. The plot also indicates a clear distinction between decathletes who performed poorly on this event, decathletes who performed well on this event but not well overall, and decathletes who performed well both in the pole vault and in the overall contest.

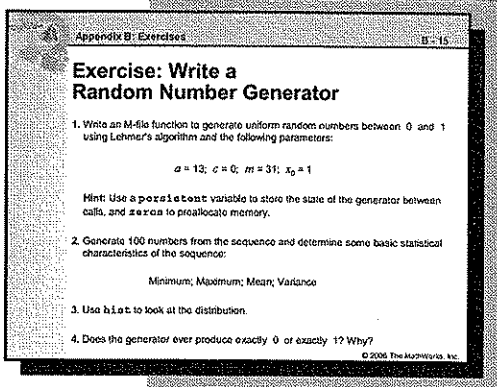
Try

```
>> decathscript
```





Exercise: Write a Random Number Generator



1. Write an M-file function to generate uniform random numbers between 0 and 1 using Lehmer's algorithm and the following parameters:

- $a = 13$
- $c = 0$
- $m = 31$
- $x_0 = 1$

Hint: Use a `persistent` variable to store the state of the generator between calls, and `zeros` to preallocate memory.

2. Generate 100 numbers from the sequence and determine some basic statistical characteristics of the sequence:

- Minimum
- Maximum
- Mean
- Variance

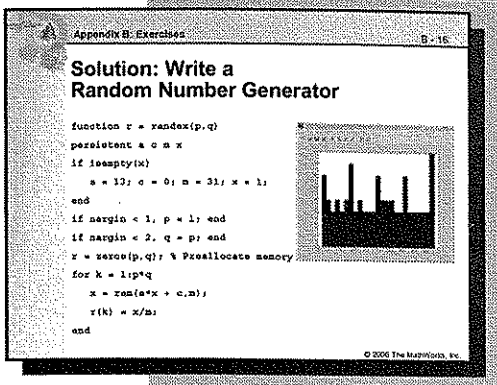
3. Use `hist` to look at the distribution.

4. Does the generator ever produce exactly 0 or exactly 1? Why?

Try

```
>> doc rem
>> doc persistent
```

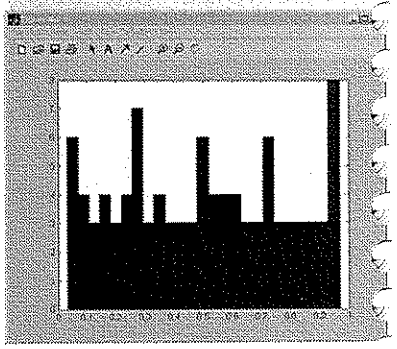
Solution: Write a Random Number Generator



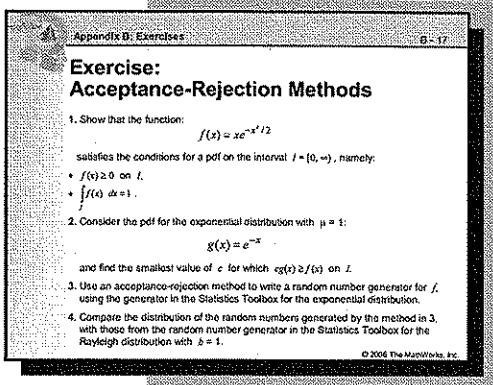

```
function r = randex(p,q)

% Multiplicative congruential
% uniform random number generator
%
% r = randex
% generates a single random number.
%
% r = randex(m,n)
% generates an m-by-n random matrix.
%
% clear randutx
% reinitializes the generator.
%
% Seeds must be set by editing this file.

persistent a c m x

if isempty(x)
   a = 13;
   c = 0;
   m = 31;
   x = 1;
end

if nargin < 1, p = 1; end

if nargin < 2, q = p; end

r = zeros(p,q); % Preallocate memory
for k = 1:p*q
   x = rem(a*x + c, m);
   r(k) = x/m;
end
```

Try

```
>> edit randex

>> r = ...
randex(100,1);
>> minr = min(r);
>> maxr = max(r);
>> meanr = mean(r);
>> varr = var(r);

>> hist(r,25)
```



Exercise: Acceptance-Rejection Methods


Appendix B: Exercises B - 17

Exercise:
Acceptance-Rejection Methods

1. Show that the function:

$$f(x) = xe^{-x^2/2}$$

satisfies the conditions for a pdf on the interval $I = [0, \infty)$, namely:

- $f(x) \geq 0$ on I.
- $\int_I f(x)\ dx = 1$.

2. Consider the pdf for the exponential distribution with $\mu = 1$:

$$g(x) = e^{-x}$$

and find the smallest value of c for which $cg(x) \geq f(x)$ on I.

3. Use an acceptance-rejection method to write a random number generator for f, using the generator in the Statistics Toolbox for the exponential distribution.

4. Compare the distribution of the random numbers generated by the method in 3. with those from the random number generator in the Statistics Toolbox for the Rayleigh distribution with $b = 1$.

© 2006 The MathWorks, Inc.

1. Show that the function:

$$f(x) = xe^{-x^2/2}$$

satisfies the conditions for a pdf on the interval $I = [0, \infty)$, namely

- $f(x) \geq 0$ on I
- $\int_I f(x)\ dx = 1$

2. Consider the pdf for the exponential distribution with $\mu = 1$:

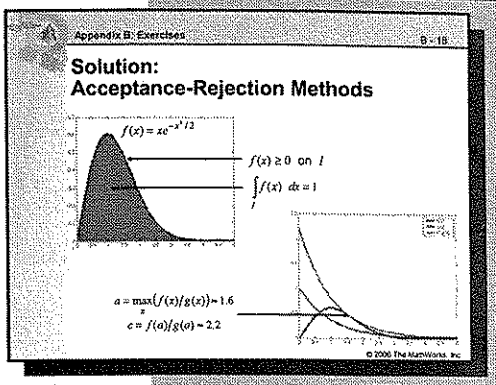
$$g(x) = e^{-x}$$

and find the smallest value of c for which $cg(x) \geq f(x)$ on I.

3. Use an acceptance-rejection method to write a random number generator for f, using the random number generator in the Statistics Toolbox for the exponential distribution.

4. Compare the distribution of the random numbers generated by the method in 3. with those from the random number generator in the Statistics Toolbox for the Rayleigh distribution with $b = 1$.

Solution: Acceptance-Rejection Methods



1.

- Since $x \geq 0$ on I and the exponential function is always positive, the product $f(x)$ is non-negative on I.

- Let $u = x^2/2$, $du = x\,dx$. Then

$$\int_{x=0}^{x=\infty} xe^{-x^2/2}\,dx = \int_{u=0}^{u=\infty} e^{-u}\,du$$

$$= \lim_{b\to\infty} \int_0^b e^{-u}\,du$$

$$= \lim_{b\to\infty} \left[-e^{-u}\right]_0^b$$

$$= \lim_{b\to\infty} \left(-e^{-b}+1\right)$$

$$= 1$$

2. We have

$$ce^{-x} \geq xe^{-x^2/2}$$

when

$$c \geq xe^{x-x^2/2}$$

If

$$h(x) = xe^{x-x^2/2}$$

then

$$h'(x) = e^{x-x^2/2}(1+x-x^2)$$

The maximum value of $h(a)$ occurs when $(1 + a - a^2) = 0$, at

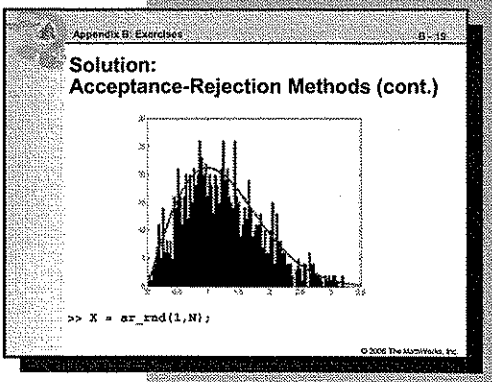
$$a = (1+\sqrt{5})/2$$

This gives a minimum c of $h(a) \approx 2.2$.

Try

```
>> f = ...
inline( ...
'x.*exp(-
(x.^2)/2)');
>> x = 0:0.1:5;
>> plot(x,f(x))

>> format long
>> quad(f,0,5)
>> quad(f,0,10)
>> quad(f,0,100)
>> tol = 10^-10;
>> quad(f,0,100,tol)

>> g = ...
inline('exp(-x)');
>> x = 0:0.1:5;
>> plot(x,f(x))
>> hold on
>> plot(x,g(x),'r')
>> h = ...
inline( ...
'x.*exp( ...
x-(x.^2)/2)');
>> a = ...
(1+sqrt(5))/2;
>> c = h(a);
>>
plot(x,c*g(x),'m')
>> legend( ...
'f(x)','g(x)', ...
'c*g(x)')
```

Solution: Acceptance-Rejection Methods (continued)



3. The following code implements the method.

```
function X = ar_rnd(m,n)

f = inline('x.*exp(-(x.^2)/2)');
g = inline('exp(-x)');

h = inline('x.*exp(x-(x.^2)/2)');
a = (1 + sqrt(5))/2;
c = h(a);

X = zeros(m,n);

for i = 1:m*n
    accept = 0;
    while accept == 0
       r = exprnd(1);
       u = rand;
       if c*u <= f(r)/g(r)
           X(i) = r;
           accept = 1;
       end
    end
end
```

4. The pdf f is actually the pdf for a Rayleigh distribution with $b = 1$.

Compare

```
>> X = ar_rnd(1,1000); hist(X,100)
```

with

```
>> Y = raylrnd(1,1,1000); hist(Y,100)
```

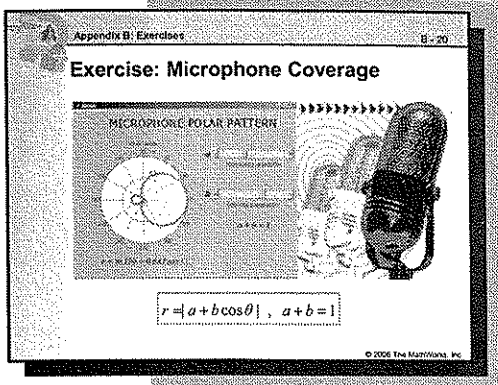
Try

```
>> edit ar_rnd
>> N = 1000;
>> X = ...
ar_rnd(1,N);
>> n = 100;
>> hist(X,n)
>> hold on
>> [nout,xout] =
... hist(X,n);
>> m = min(xout);
>> M = max(xout);

>> f = ...
inline( ...
'x.*exp(-
(x.^2)/2)');
>> x = m:0.1:M;
>> y = f(x);
>> s = ((M-m)/n)*N;
>> plot(x,s*y,'r')
```

See p. 3-13 for scaling info.

```
>> Y = ...
raylrnd(1,1,1000);
>> hist(Y,100)
>> hold on
>> x = 0:0.1:3.5;
>> y =
raylpdf(x,1);
>> plot(x,35*y,'r')
```

Exercise: Microphone Coverage



The *polar pattern* of a microphone is a plot of its sensitivity as a function of direction. Measurements are made by placing a microphone on a turntable rotated through 360 degrees, at a fixed distance from a fixed-frequency signal. Output levels are measured in decibels relative to a fixed reference level. The graph of the output level versus angle is the polar pattern.

The magnitude of the polar response r may be described by the polar equation

$$r = |a + b\cos\theta|$$

where $a + b = 1$. The specific values of a and b determine the shape of the response pattern.

Try

```
>> micgui
```

• Omni-directional microphones

For omni-directional mics, $a = 1$, so $b = 0$. Omni mics are called *pressure mics*, since sound pressure alone conveys no information about angular direction. Omni mics have excellent low frequency response, are less susceptible to breath and wind noise, and have smoother frequency response than unidirectional mics.

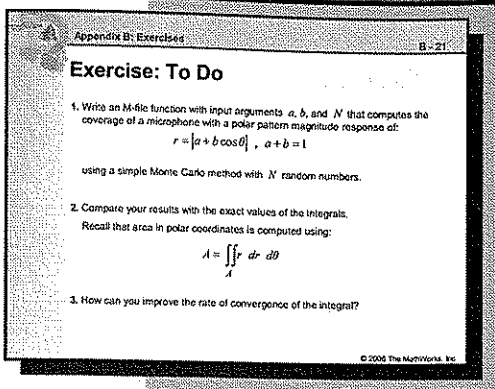
• Bi-directional (Figure Eight) microphones

For bi-directional mics, $a = 0$, so $b = 1$. Bi-directional mics pick up sound from the front and rear, rejecting sounds from the sides. These mics are called *velocity mics*, because the pattern responds to the air particle velocity information. Bi-directional mics are used in recording and sound reinforcement applications where their pattern provides pick up for two adjacent sources.

• Uni-directional (Cardioid) microphones

When a pressure element ($0 < a < 1$) is combined with a velocity element ($0 < b < 1$), a uni-directional pattern is formed. Sound can be picked up from the front and mostly rejected from the rear. Cardioid mics are the most popular of all patterns, and are widely used in both sound reinforcement and recording.

Exercise: To Do



The area (or, in 3D, the volume) of the polar pattern gives a rough measure of a microphone's *coverage*. The integrals involved are not difficult, and closed-form solutions are readily computed. The setting, however, provides a simple context for evaluating methods of Monte Carlo integration.

1. Write an M-file function with input arguments a, b, and N that computes the coverage of a microphone with a polar pattern magnitude response of

$$r = |a + b\cos\theta| \quad , \quad a + b = 1$$

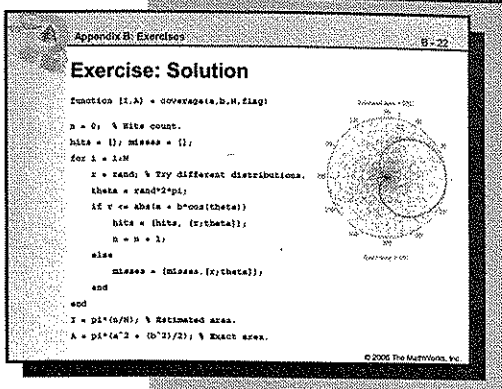
using a simple Monte Carlo method with N random numbers.

2. Compare your results with the exact values of the integrals. Recall that area in polar coordinates is computed using

$$A = \iint_A r \; dr \; d\theta$$

3. How can you improve the rate of convergence of the integral?

Exercise: Solution

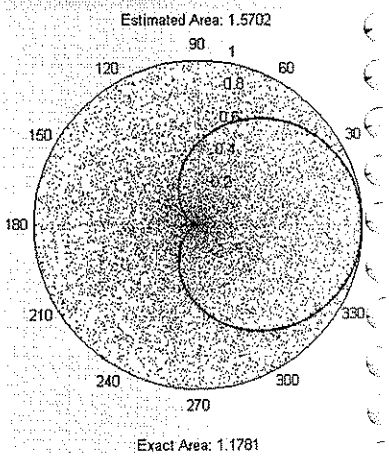


1.

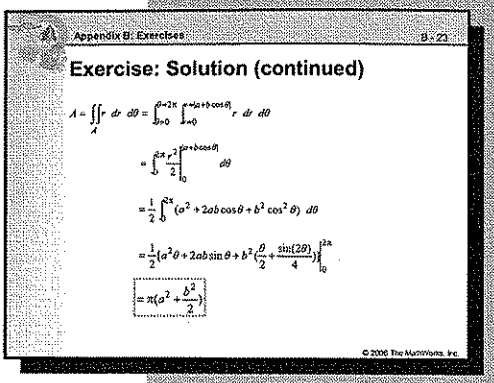
```
function [I,A] = coverage(a,b,N,flag)

n = 0;  % Hits count.
hits = [];
misses = [];

% Compute integral:
for i = 1:N
    r = rand; % Try different distributions.
    theta = rand*2*pi;
    if r <= abs(a + b*cos(theta))
        hits = [hits, [r;theta]];
        n = n + 1;
    else
        misses = [misses,[r;theta]];
    end
end

I = pi*(n/N); % Estimated area.
A = pi*(a^2 + (b^2)/2); % Exact area.

% Plot results if flag == 1:
if flag == 1
    t = 0:0.01:2*pi;
    r = abs(a + b*cos(t));
    h0 = polar(t,r,'r');
    set(h0,'LineWidth',2)
    title(['Estimated Area: ',num2str(I)])
    xlabel(['Exact Area: ',num2str(A)])
    hold on
    h1 = polar(hits(2,:),hits(1,:),'r.');
    set(h1,'MarkerSize',1)
    h2 = polar(misses(2,:),misses(1,:),'b.');
    set(h2, 'MarkerSize', 1)
end
```

Try

```
>> edit coverage

>> [I,A] = ...
coverage( ...
.5,.5,10^4,1)
>> [I,A] = ...
coverage( ...
.5,.5,10^5,0)
```



Exercise: Solution (continued)



2.

$$A = \iint_A r \; dr \; d\theta = \int_{\theta=0}^{\theta=2\pi} \int_{r=0}^{r=|a+b\cos\theta|} r \; dr \; d\theta$$

$$= \int_0^{2\pi} \left. \frac{r^2}{2} \right|_0^{|a+b\cos\theta|} d\theta$$

$$= \frac{1}{2} \int_0^{2\pi} (a^2 + 2ab\cos\theta + b^2\cos^2\theta) \; d\theta$$

$$= \frac{1}{2}\left[a^2\theta + 2ab\sin\theta + b^2\left(\frac{\theta}{2} + \frac{\sin(2\theta)}{4}\right)\right]\Bigg|_0^{2\pi}$$

$$= \pi\left(a^2 + \frac{b^2}{2}\right)$$

3. Notice that we are using a uniform random number generator for both `r` and `theta` in the code (lines 17 and 18 in `coverage.m`). The choice for r is inaccurate because the bands of area between r and $r + dr$ at any angle increase quadratically with r. A better choice would be to use a random number generator for r that weights the distribution on [0, 1] more heavily for larger r. Use the `disttool` to investigate some possibilities, or write a random number generator for a quadratic distribution on [0, 1] using acceptance-rejection techniques.

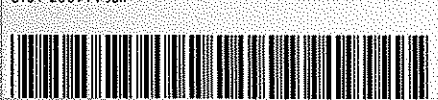
Try

... editing line 17 in `coverage.m`

The MathWorks

3 Apple Hill Drive
Natick, MA 01760 USA
508-647-7000

ST01 2009V1 Jan



www.mathworks.com/training