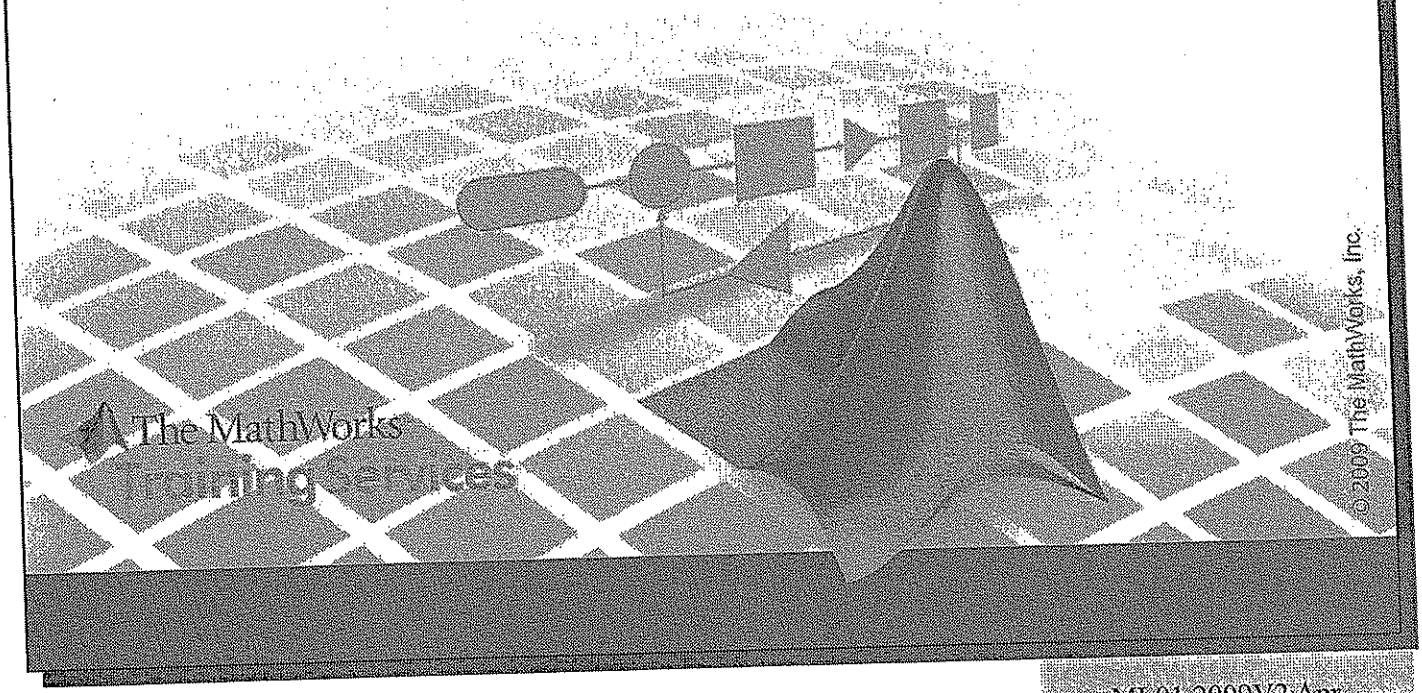
MATLAB® Fundamentals

Table of Contents



ML01 2009V2 Apr

Table of Contents

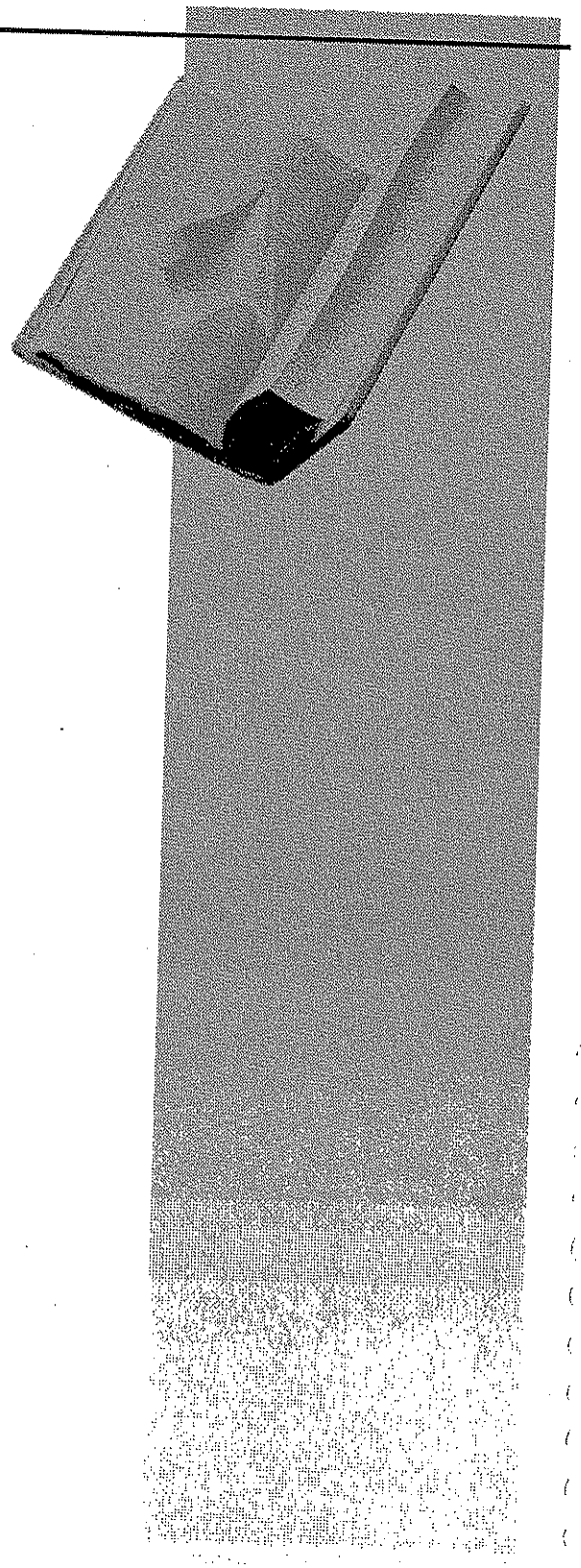


Table of Contents

1. Introduction

Table of Contents

2. Working With the MATLAB® User Interface

Table of Contents

Table of Contents

4. Plotting and Visualization

Table of Contents

5. M-Files

Table of Contents

6. Basic Statistics and Data Analysis

Table of Contents

7. Data Types

Table of Contents

8. M-File Programming

Table of Contents

9. Troubleshooting M-Files

Table of Contents

10. Building Graphical User Interfaces

Table of Contents

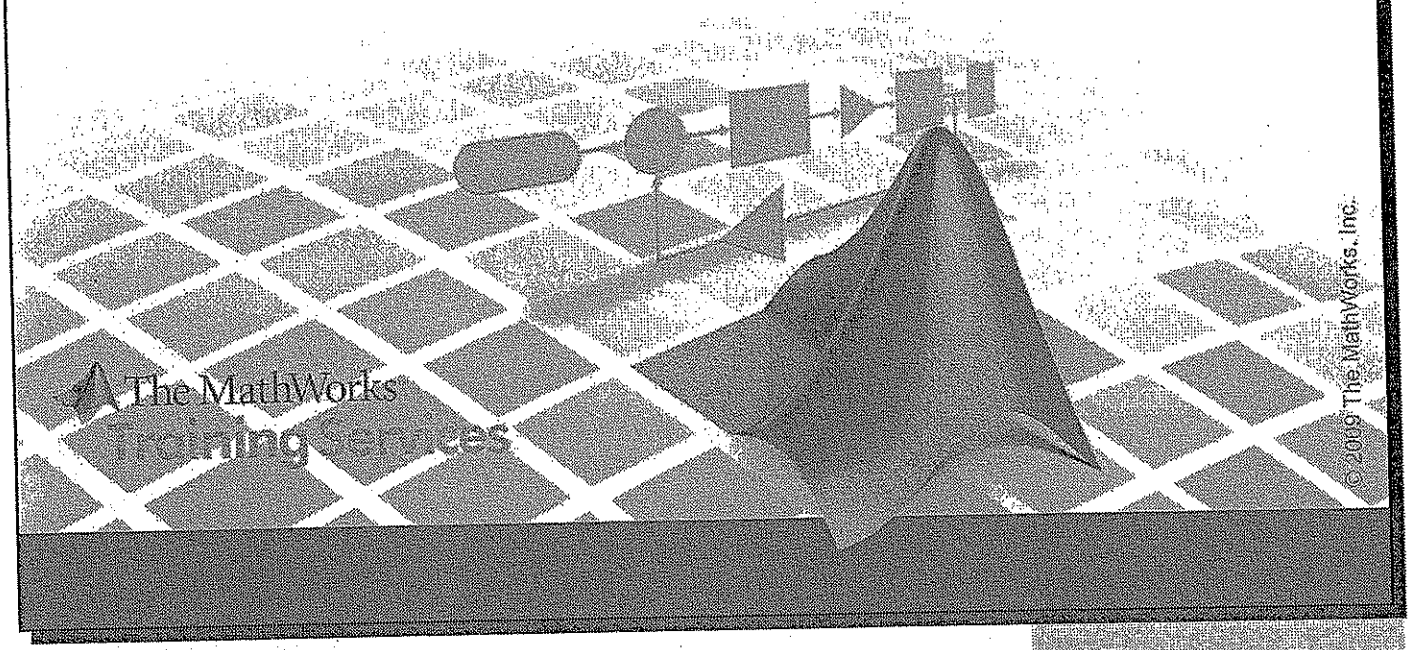
11. Conclusion

Table of Contents

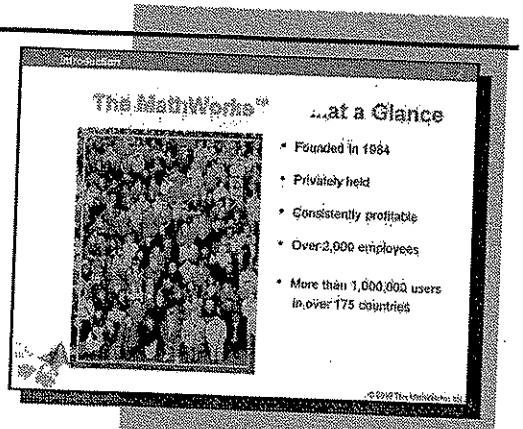
Appendices

MATLAB® Fundamentals

Introduction

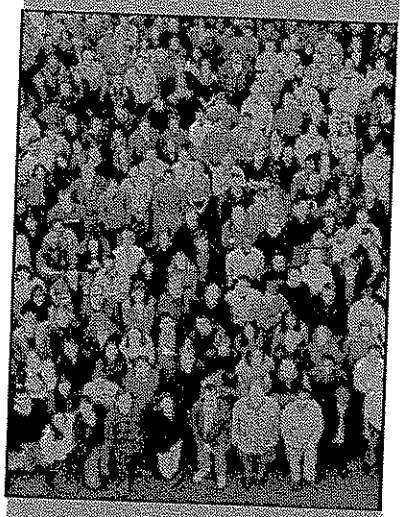


The MathWorks™ at a Glance



The MathWorks™ is the leading developer and supplier of technical computing software in the world. Founded in 1984, it now employs over 2,000 people worldwide. The company is privately held and has been profitable every year since its inception.

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.



The MathWorks

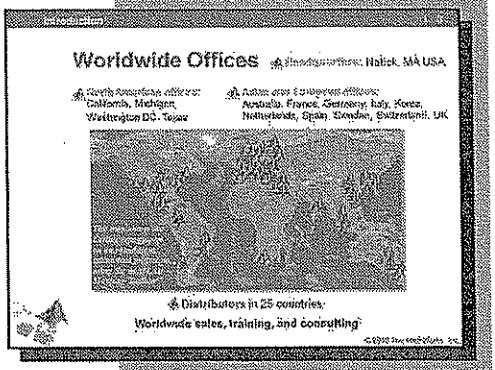
Jack Little
President

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



The MathWorks customers are over 1,000,000 of the world's technical leaders, in over 175 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.

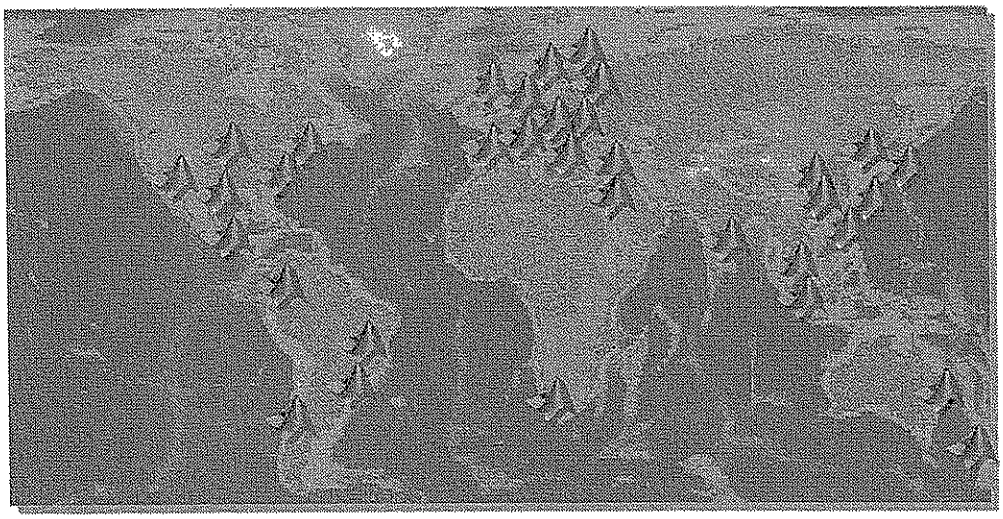
The MathWorks supports its customers through a worldwide network of offices, distributors, and resellers.

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

Worldwide Offices

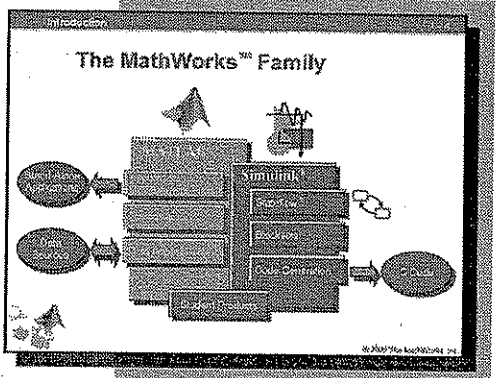
For information on any of our worldwide offices, please go to the following location on The MathWorks Web site and choose a country:

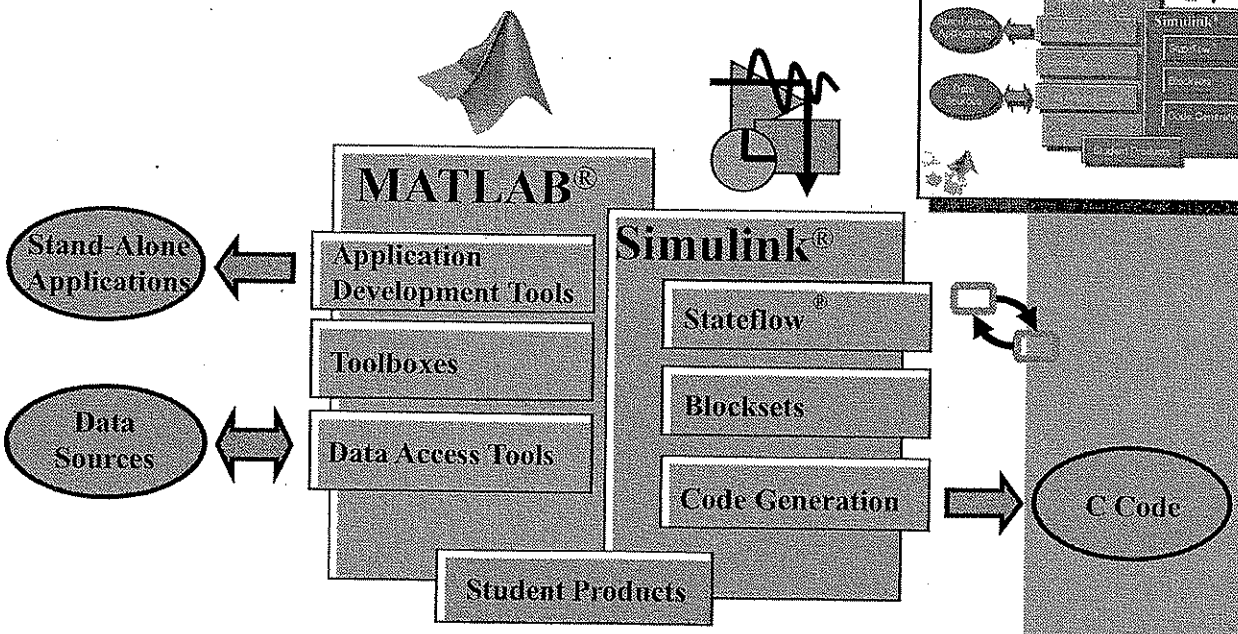
`http://www.mathworks.com/company/worldwide/`



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

The MathWorks™ Family





Key Characteristics of the MATLAB® Language

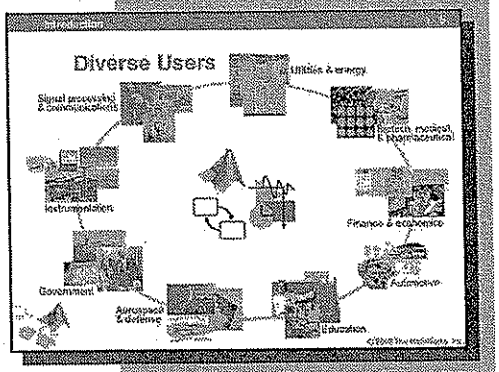
- A user-friendly, intuitive syntax, favoring brevity and simplicity without compromising intelligibility
- The highest quality numerical algorithms, based on close historical ties with the numerical analysis research community
- Powerful, easy-to-use graphics and visualization capabilities
- A high-level 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

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

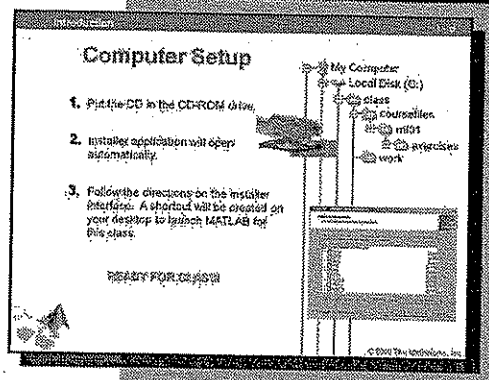
Instrumentation

Marine

Signal processing and communications

Utilities and energy

Computer Setup

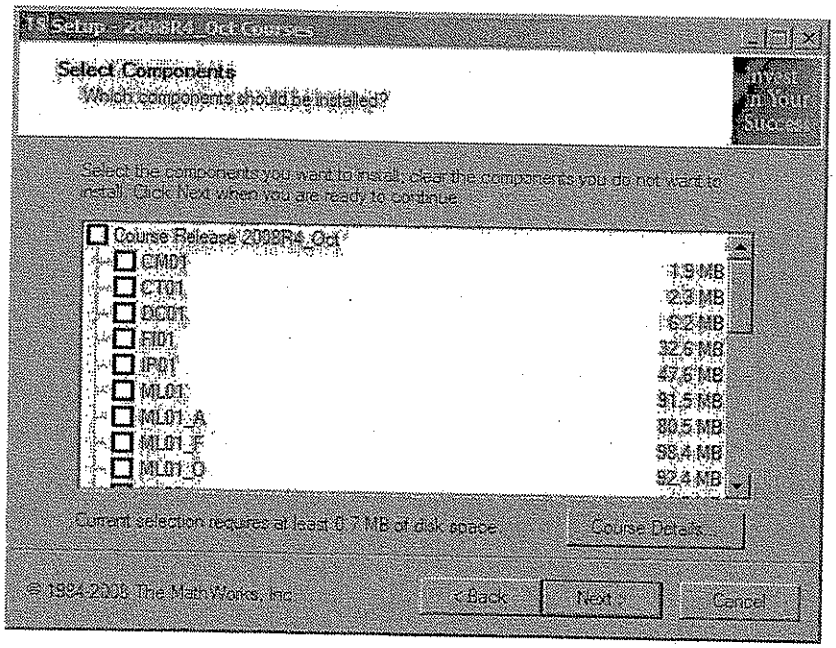


To get ready for class, you need to install the examples and exercises on your course CD. Follow these steps:

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.

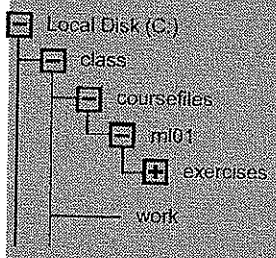


In the installer, you have these options:

- 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



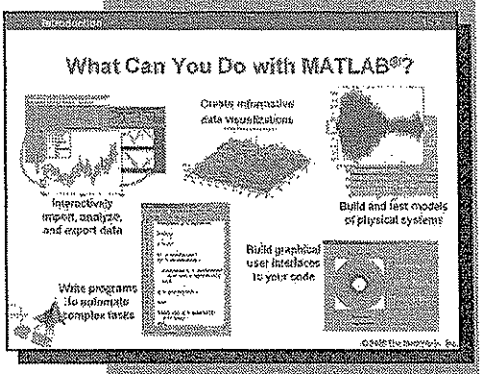
The installation creates a subdirectory of your chosen root directory (the default is `C:\class`) called `coursefiles`. This directory has subdirectories for each of the courses you install, labeled by course number. These individual course directories contain the examples and exercises for the courses. 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.

What Can You Do with MATLAB®?

Lots of useful things!

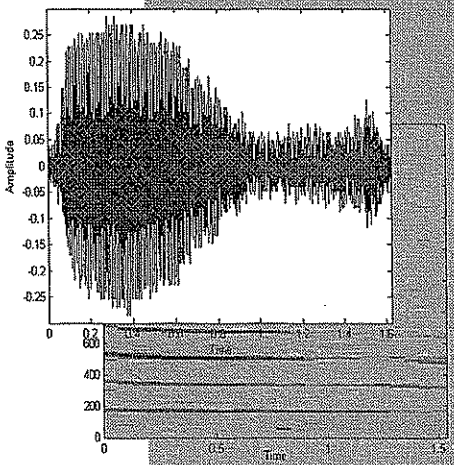
In this course, you will

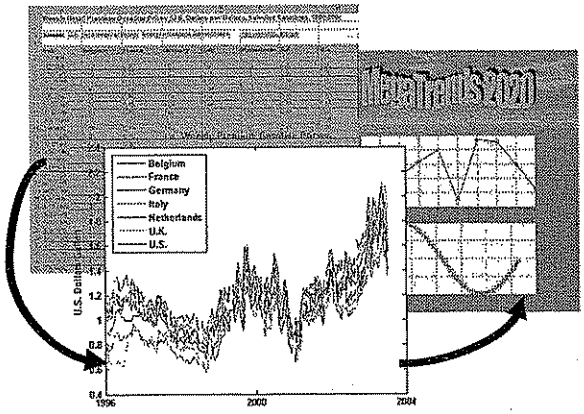
- Interactively import, analyze, and export data.
- Create informative data visualizations.
- Build and test models of physical systems.
- Write programs to automate complex tasks.
- Build graphical user interfaces to your code.

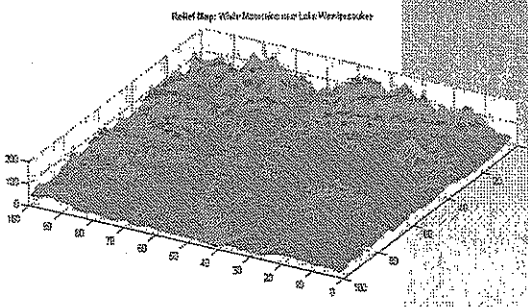


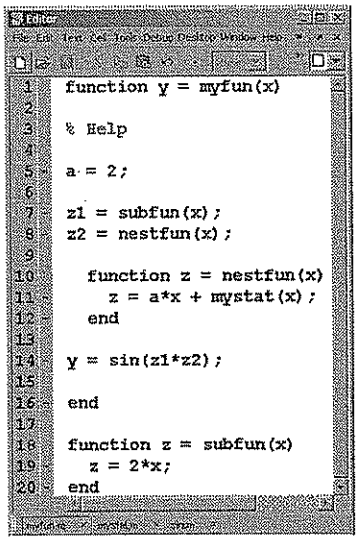
Try

```
>> whalecall
>> roulettegui
```

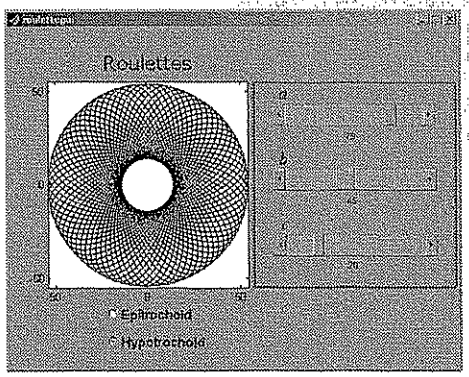




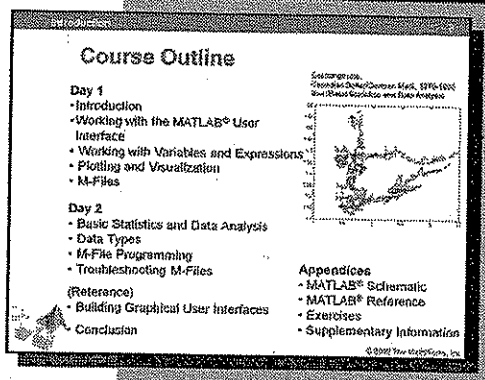




```
function y = myfun(x)

% Help

a = 2;

z1 = subfun(x);
z2 = nestfun(x);

  function z = nestfun(x)
    z = a*x + mystat(x);
  end

y = sin(z1*z2);

end

function z = subfun(x)
  z = 2*x;
end
```



Course Outline



Day 1
- Introduction
- Working with the MATLAB® User Interface
- Working with Variables and Expressions
- Plotting and Visualization
- M-Files

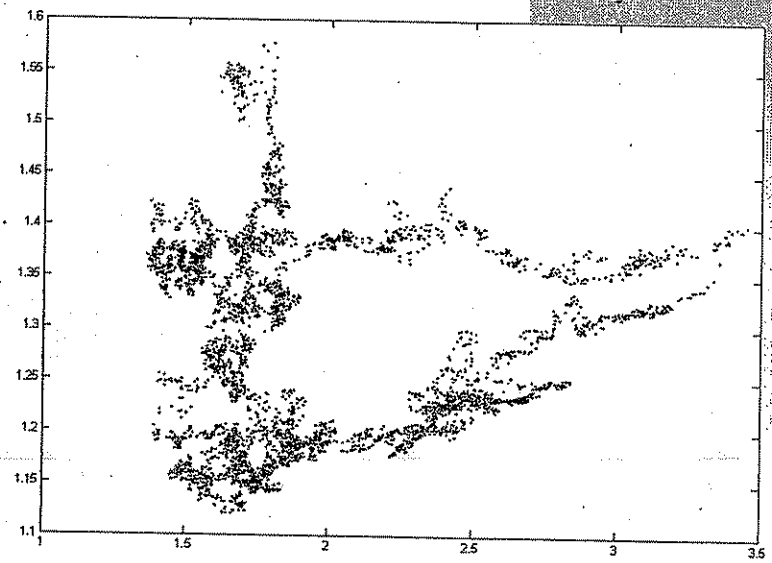
Day 2
- Basic Statistics and Data Analysis
- Data Types
- M-File Programming
- Troubleshooting M-Files

(Reference)
- Building Graphical User Interfaces

- Conclusion

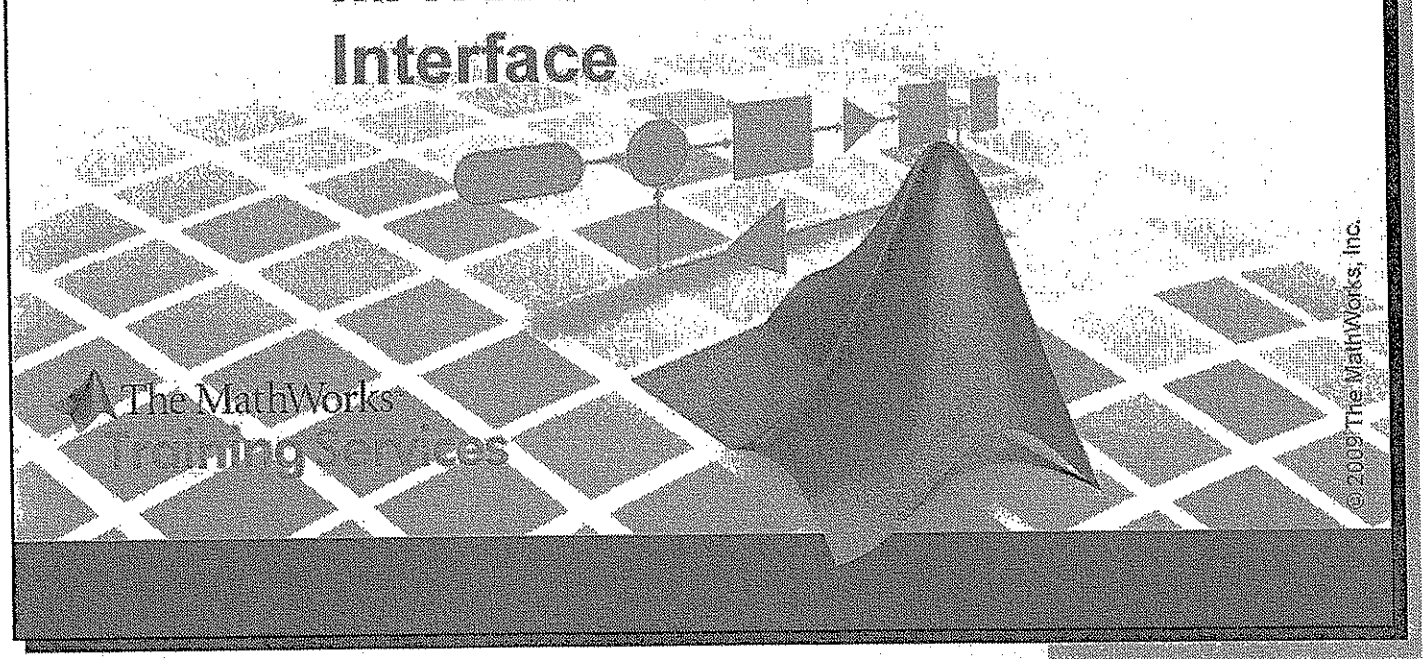
Appendices
- MATLAB® Schematic
- MATLAB® Reference
- Exercises
- Supplementary Information

Exchange rate, Canadian Dollar/German Mark, 1979–1998.
See Chapter 6: "Basic Statistics and Data Analysis."



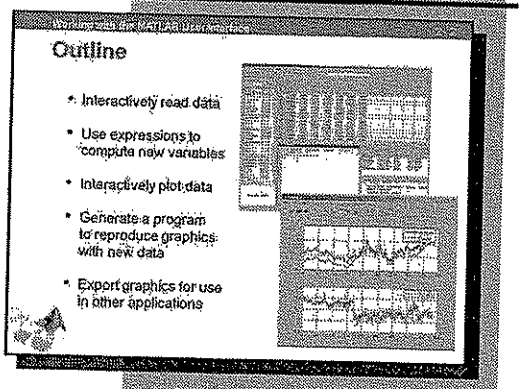
MATLAB® Fundamentals

Working with the MATLAB® User Interface



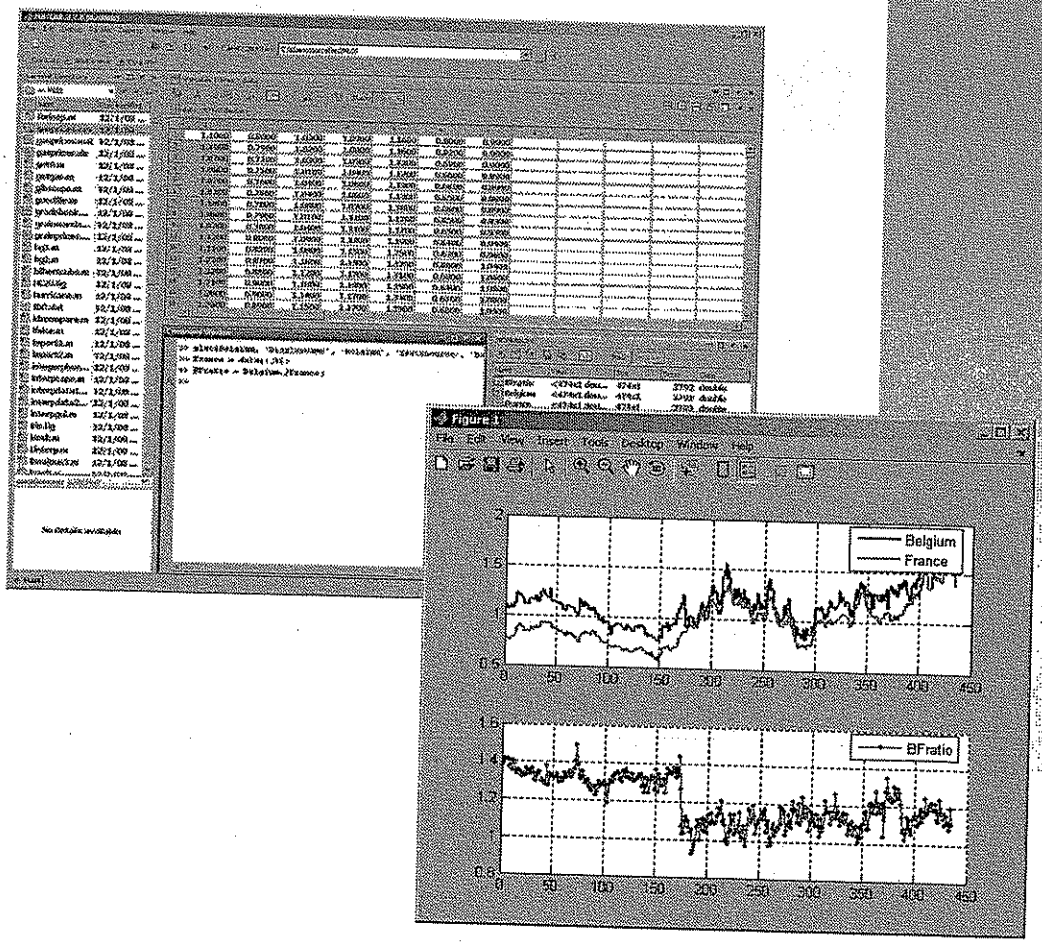
The MathWorks
Training Services

Outline

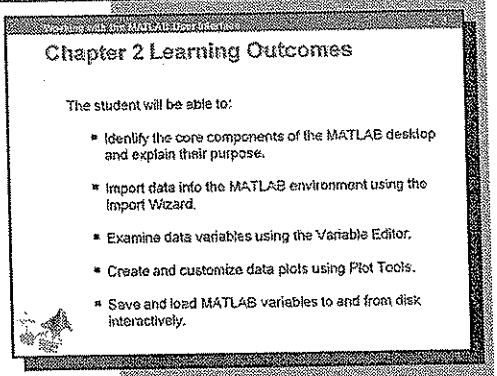


- Interactively read data
- Use expressions to compute new variables
- Interactively plot data
- Generate a program to reproduce graphics with new data
- Export graphics for use in other applications

This chapter introduces the main features of the MATLAB® integrated design environment and its user interfaces. Many themes for the course are established in this chapter, to be explored in detail in later chapters.



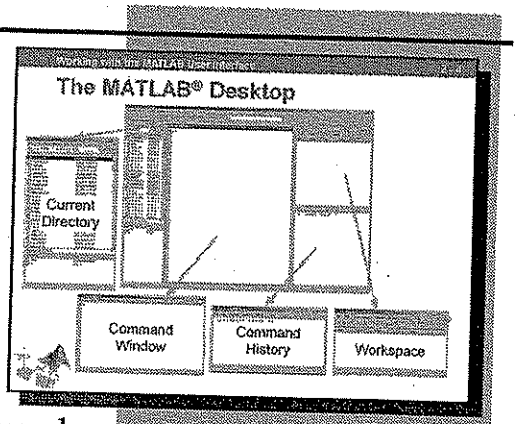
Chapter 2 Learning Outcomes



The student will be able to:

- Identify the core components of the MATLAB desktop environment and explain their purpose.
- Import data into the MATLAB environment using the Import Wizard.
- Examine data variables using the Variable Editor.
- Create and customize data plots using Plot Tools.
- Save and load MATLAB variables to and from disk interactively.

The MATLAB® Desktop



Open MATLAB by double-clicking the MATLAB desktop icon .

By default, MATLAB displays a desktop interface with the following principal components:

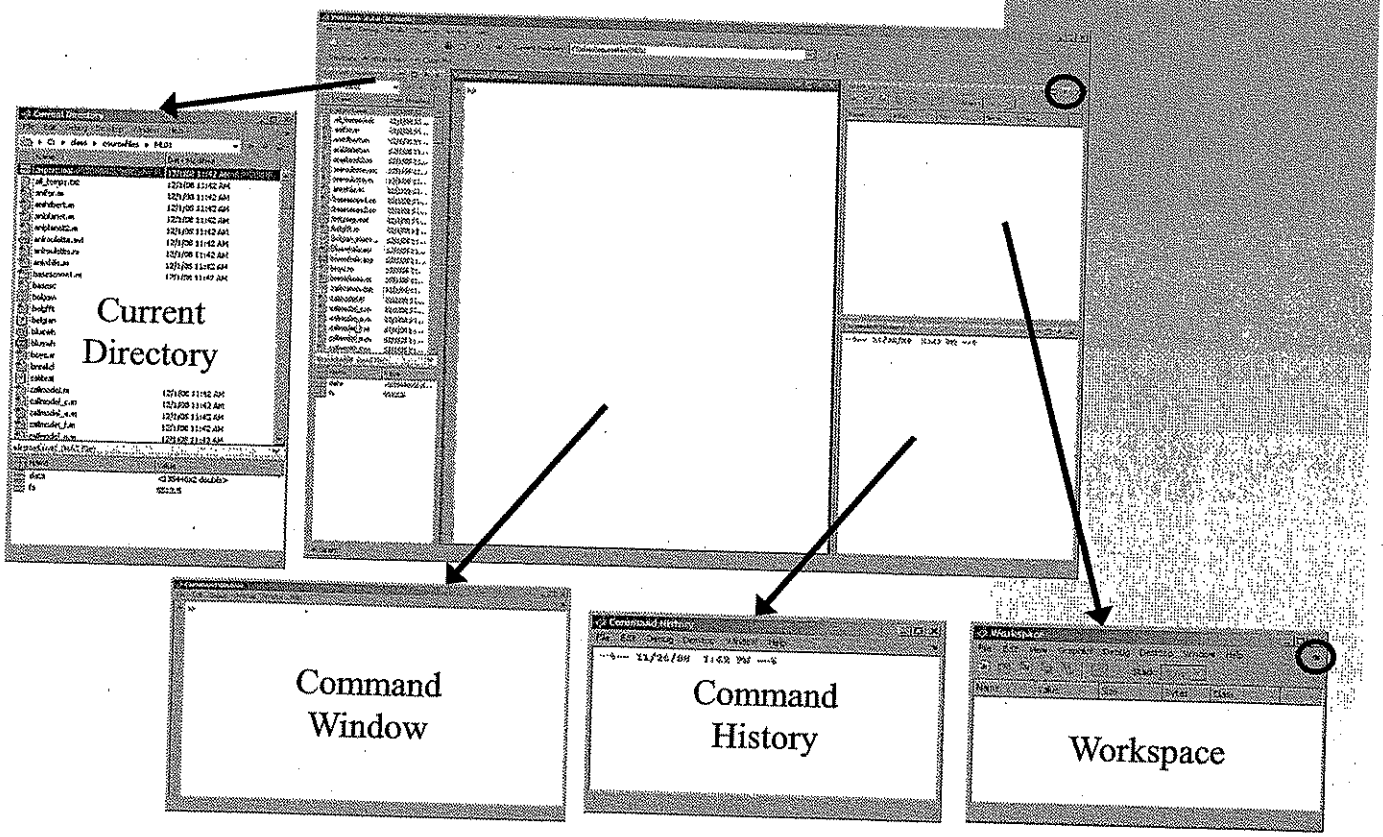
- **The Command Window** — To enter MATLAB commands
- **The Workspace Browser** — To display MATLAB variables
- **The Current Directory Browser** — To display MATLAB files
- **The Command History** — To record your work history

Any component of the interface can be undocked to a separate window, docked back to the desktop, or closed.

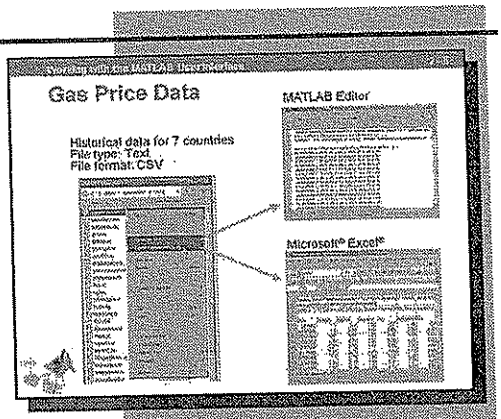
To return to the default desktop layout, choose **Desktop → Desktop Layout → Default** from the menus at the top of the desktop.

Try

Double-click the MATLAB desktop icon to open MATLAB.



Gas Price Data



The file `gasprices.csv` contains historical data on weekly retail premium gasoline prices excluding taxes, in U.S. dollars per gallon, from seven countries over recent years. Updates can be downloaded from the U.S. Department of Energy at

`http://www.eia.doe.gov`

The file is ASCII text (as opposed to a binary file containing non-ASCII characters), and all of the data values following the initial text header are separated by a comma delimiter. We say that the *type* of the file is text, and the *format* is comma-separated value (CSV).

To view the file, change your MATLAB current directory to the directory in which you installed your course files. By default, this is

`C:\class\coursefiles\m101`

A quick way to change the current directory is by clicking the Browse for Folder button next to the current directory displayed at the top of the MATLAB desktop. Once you change the directory, you can see the contents of the directory by clicking the tab for the Current Directory Browser. Scroll to find `gasprices.csv`.

If you right-click the filename in the Current Directory Browser, a context menu appears giving you options to

- **Open as Text**
- **Open Outside MATLAB**

The first option opens the file in the MATLAB (text) editor. The second option opens the file in the default application on your computer used for reading CSV files, such as Microsoft® Excel®. A version of the file, formatted in Excel, is in `gasprices.xls`.

Try

1. Change your current directory to:

   `C:\class\coursefiles\m101`

   Click the Browse for Folder button next to the current directory displayed at the top of the MATLAB desktop.

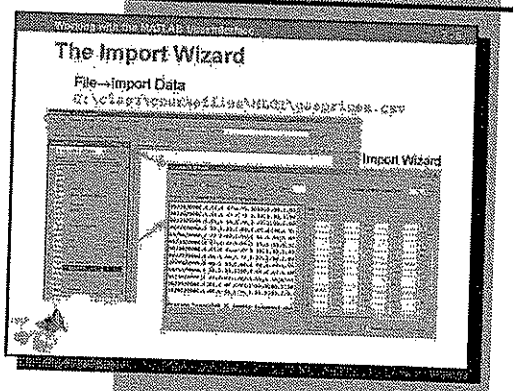
2. Go to the Current Directory Browser, and scroll to find `gasprices.csv`.

3. Right-click the filename, and choose **Open as Text**.

4. Right-click again and choose **Open Outside MATLAB**.

	A	B	C
1	Weekly Retail Premium Gasoline Prices		
2	Data contains omissions (unreported weeks)		
3	Source: U.S. Department of Energy, Energy Inf		
4			
5	Date	Belgium	France
6	1/8/1996	1.10	0.80
7	1/15/1996	1.10	0.79
8	1/22/1996	1.07	0.77
9	1/29/1996	1.05	0.75
10	2/5/1996	1.07	0.76
11	2/12/1996	1.07	0.76
12	2/19/1996	1.10	0.76
13	2/4/1996	1.05	0.75
14	3/11/1996	1.07	0.78
15	3/18/1996	1.12	0.80
16	3/25/1996	1.11	0.82
17	4/1/1996	1.22	0.87
18	4/22/1996	1.22	0.88
19	4/8/1996	1.21	0.90
20	5/13/1996	1.20	0.90
21	5/20/1996	1.20	0.89
22	5/27/1996	1.19	0.88
23	6/3/1996	1.17	0.87
24	6/10/1996	1.11	0.84
25	6/17/1996	1.12	0.85

The Import Wizard



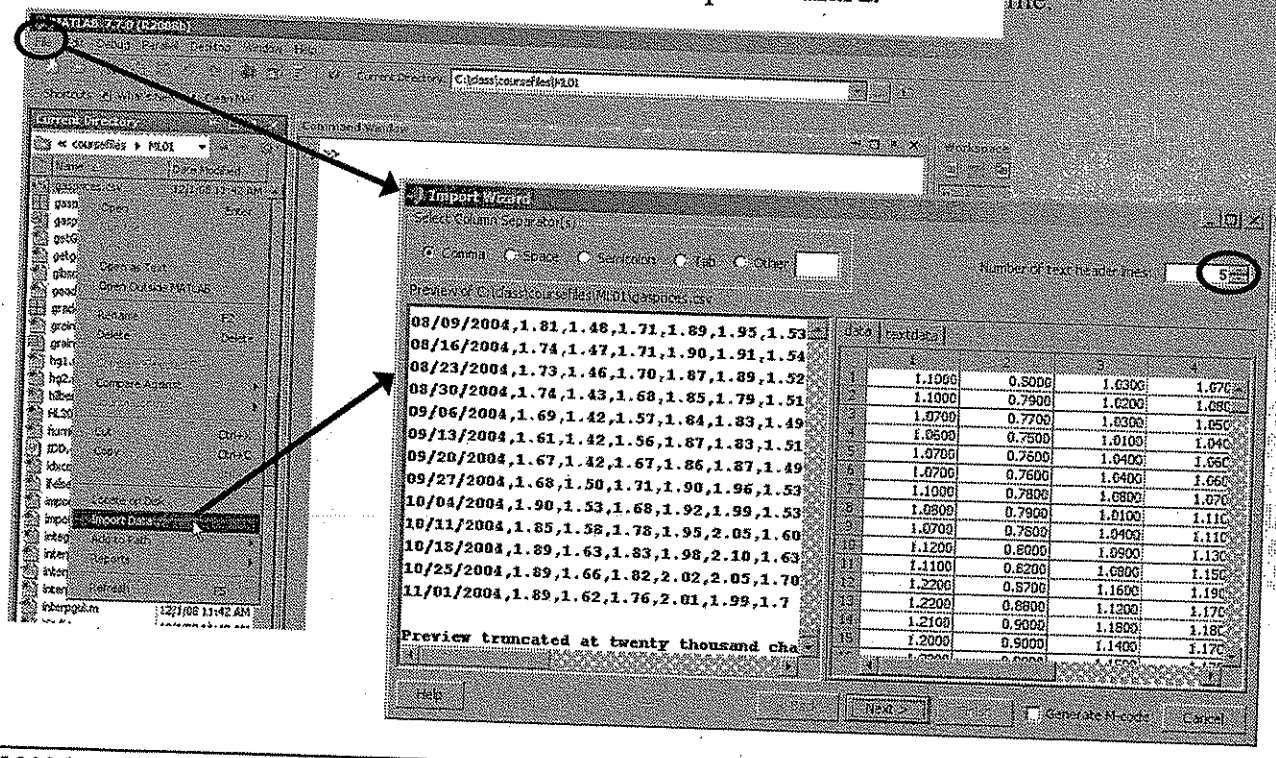
Before using MATLAB for any kind of data analysis, external data must be imported into MATLAB and placed in appropriate "data containers" (MATLAB variables). For many common file formats, this process is automated by a MATLAB graphical user interface (GUI) called the Import Wizard.

To use the Import Wizard, choose **File → Import Data** from the menus at the top of the MATLAB desktop. A dialog opens that allows you to browse for files on your computer. By default, the dialog opens to your MATLAB current directory. Recognized file formats, indicated by their extension (`.csv`, `.xls`, etc.) are displayed. Once you select a file, the Import Wizard opens with a preview of the data and a suggested breakdown into MATLAB variables. A subsequent panel of the GUI, accessed by the **Next** button, allows you to revise the break-down of the data into MATLAB variables, depending on the format of the data. Clicking the **Finish** button imports the data into MATLAB.

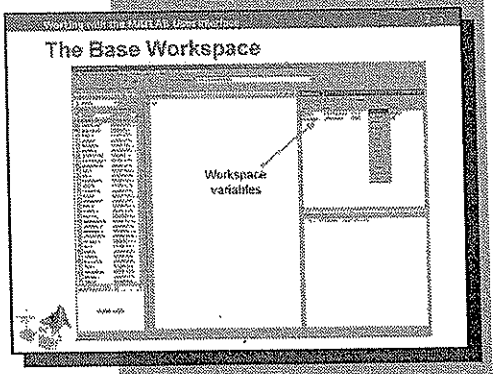
Alternatively, you can open a file in your current directory by right-clicking it in the Current Directory Browser and choosing **Import Data** from the context menu. This opens the file in the Import Wizard.

Try

1. Choose **File → Import Data** from the menus at the top of the MATLAB desktop.
2. In the dialog that opens, browse for `gasprices.csv`.
3. Select the file to open the Import Wizard.
4. Use the GUI to create MATLAB variables containing the data in the file.



The Base Workspace

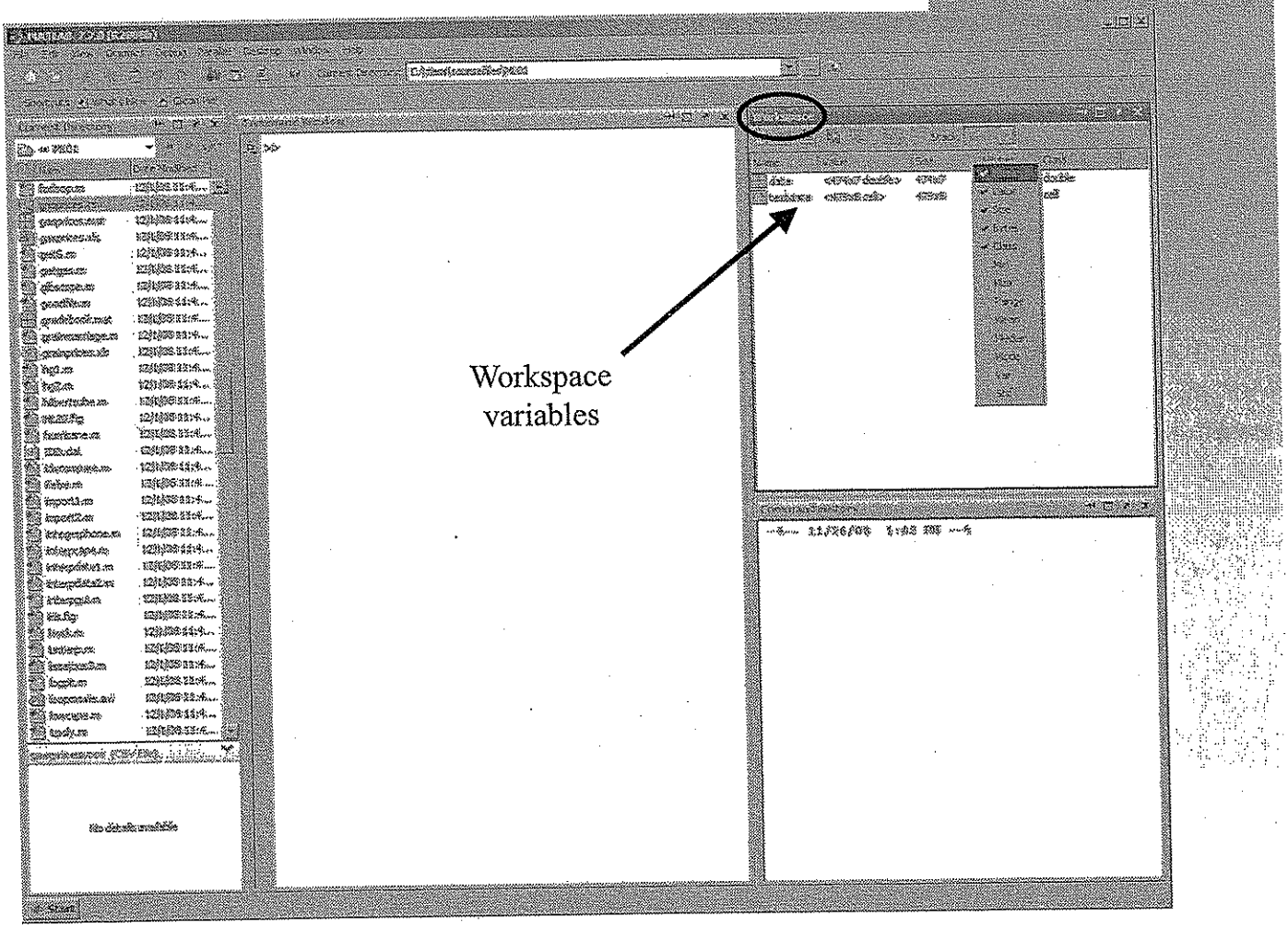


When data is imported into MATLAB (or created in MATLAB), it is stored in MATLAB *variables*. Variables are stored in parts of the MATLAB address space (memory) called *workspaces*. The *base workspace* is the default location for variables created using the Import Wizard, typing in commands at the command prompt >>, or running simple MATLAB programs called *scripts*.

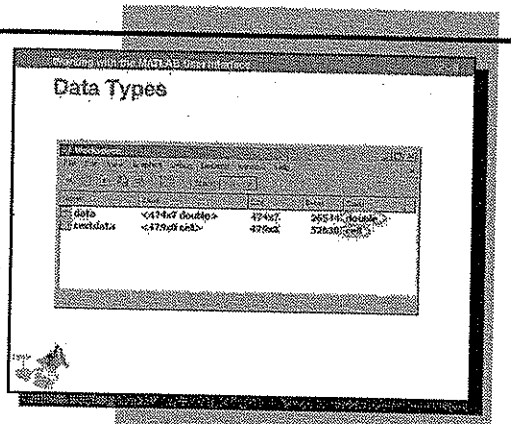
The contents of the base workspace are shown in the **Workspace** browser window. The browser displays the names of the variables currently in memory and (optionally) information about the variables. Right-clicking on the **Name** button at the top of the Workspace browser displays a context menu from which you can choose which information to display about the variables.

Try

Change the properties displayed in the **Workspace** browser to find the maximum and minimum values of the `data` variable.



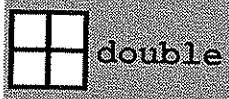
Data Types



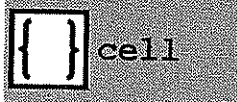
Data is stored in MATLAB variables of various *types*. Different data types store different kinds of information, and they organize that information using different techniques (data architectures). Choosing an appropriate data type for a variable depends on the characteristics of the data and how it will be accessed.

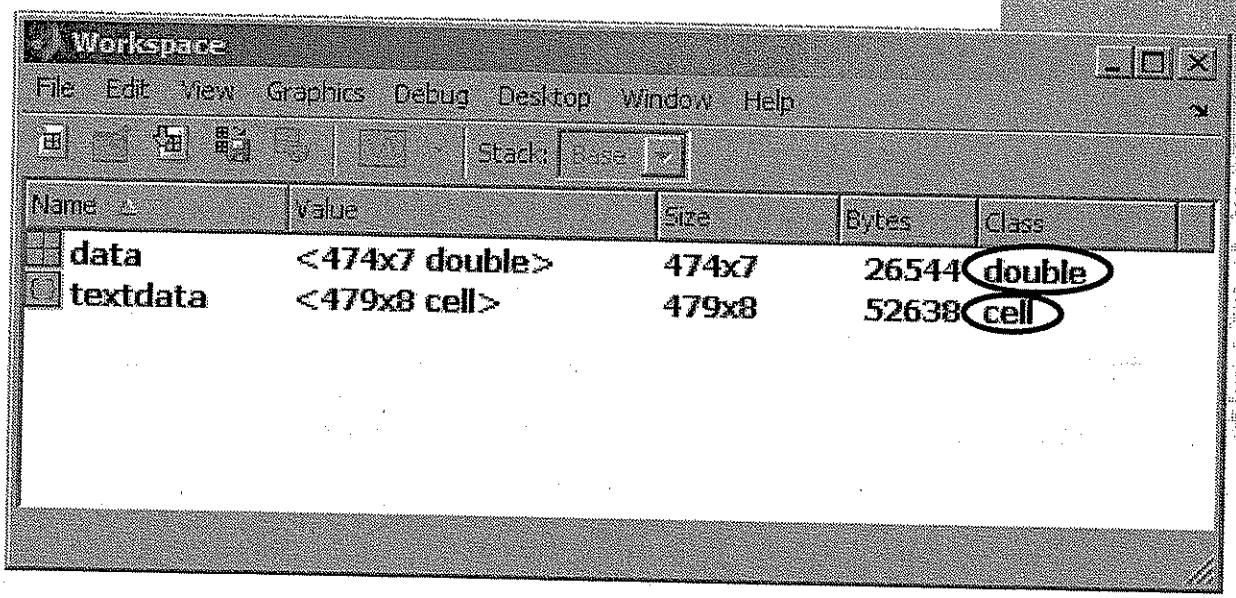
The Import Wizard, by default, chooses reasonable types for the variables that will store your data.

Double arrays are the default type for numerical data in MATLAB. The "double" refers to "double precision," or 64 bits for every number stored in memory. Regardless of the numeric format used to display double arrays in MATLAB (which you can control by choosing **File → Preferences → Command Window** from the menus at the top of the MATLAB desktop), computations involving these variables are always carried out to double precision in memory. As a result, double arrays, while very accurate, occupy large amounts of memory and require intensive computation when they contain large data sets.

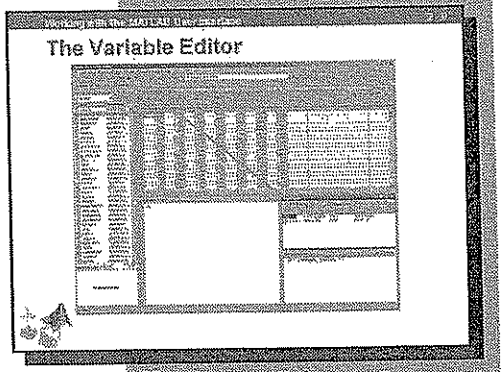


Cell arrays are a more versatile MATLAB type, capable of storing both numerical and character information. When the Import Wizard assigns variables to text data containing character strings of different lengths, it usually chooses a cell array as the "data container."





The Variable Editor



You can view, and edit, the contents of variables in the base workspace using the *Variable Editor*.

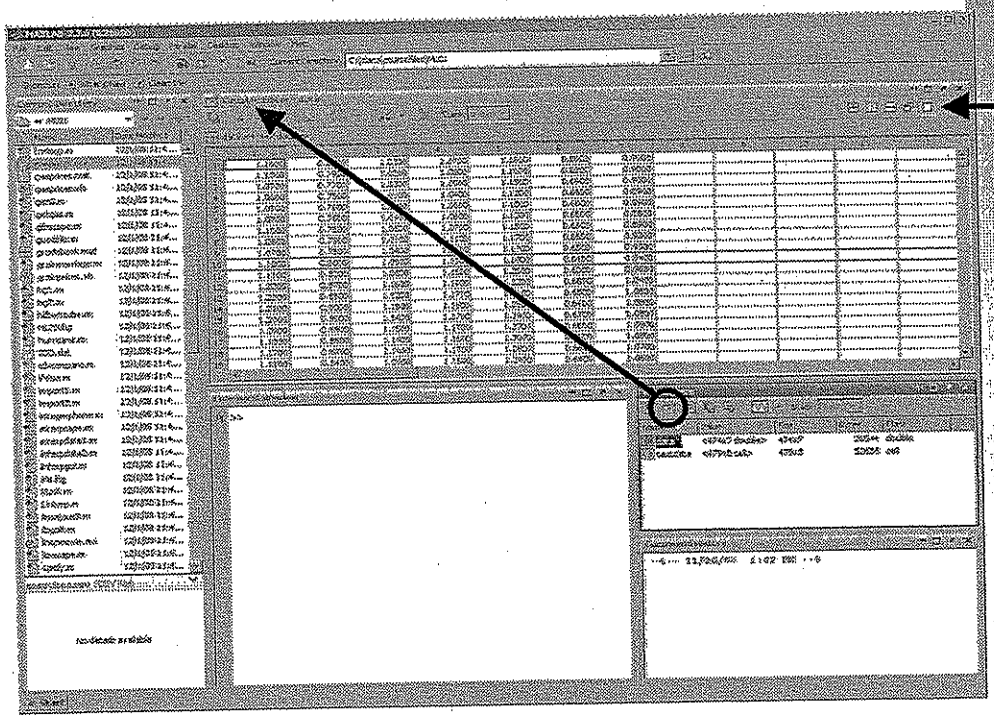
The simplest way to open a variable in the Variable Editor is to double-click the data type icon next to the variable name in the Workspace browser. The Variable Editor opens in a new window on the MATLAB desktop, with the contents of the variable displayed. You can also open the Variable Editor by selecting a variable (or variables) in the Workspace browser and clicking the Open Selection button at the top of the browser.

Try

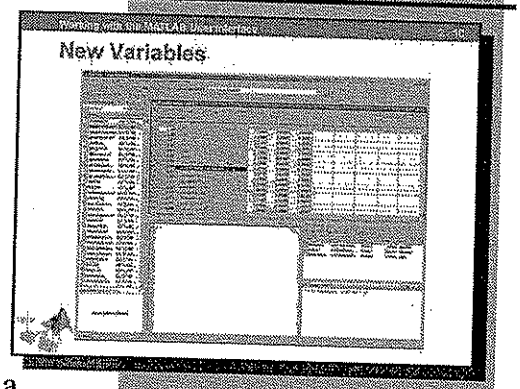
Double-click the icon next to the data variable created by the Import Wizard to open it in the Variable Editor.

When multiple variables are open in the Variable Editor, you can switch the view from one variable to another using the tabs at the bottom of the editor. Alternatively, variables can be tiled for comparison, using the icons at the top right of the editor ⊞ ◫ ⊟ 🗗 ☐.

To edit data in a variable, simply select the portion of the data you wish to edit and type in new values. The Variable Editor acts like a text editor on text data, and like a spreadsheet on numerical data. When you click away from your changes, the new data values are stored.



New Variables
Using the Variable Editor



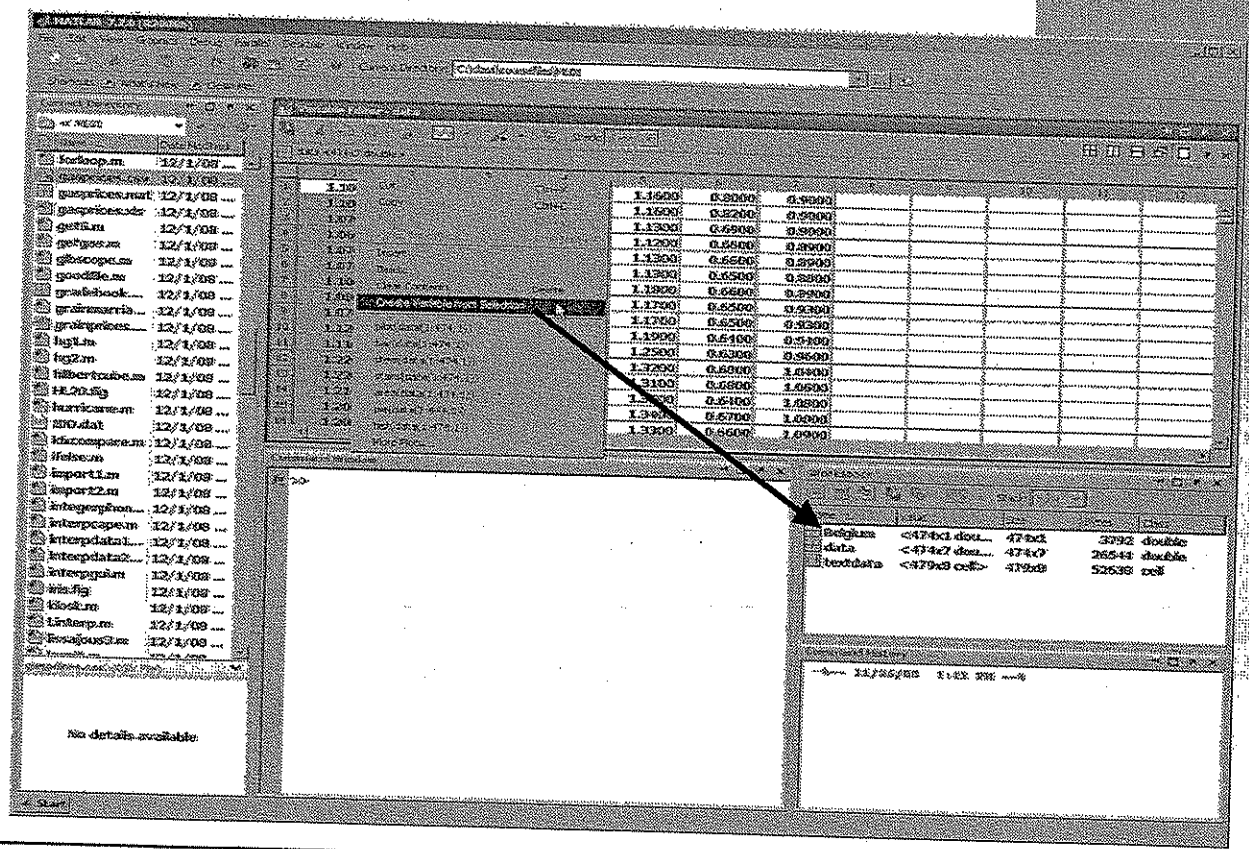
You can also use the Variable Editor to create new variables in the base workspace from data in other variables.

For example, the `data` variable created by the Import Wizard is a 474-by-7 array of values, with each column representing a different variable (country) in the data, and each row representing a different (weekly) observation. This format – observations × variables – is typical of how multivariate data is stored in a single MATLAB array. You might want to use the Variable Editor to extract individual variables (columns) giving gas price data for individual countries.

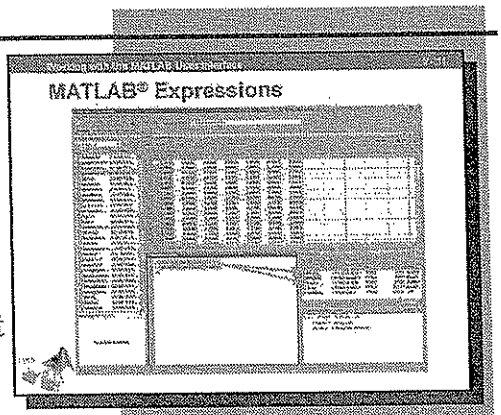
To do this, right-click the numbered column header for the variable you want to select and choose **Create Variable from Selection** from the context menu. A variable called `unnamed` is created in the base workspace. It can be renamed by right-clicking it in the Workspace browser and choosing **Rename** from the context menu.

Try

1. In the Variable Editor, right-click on the first column header in `data`, and choose **Create Variable from Selection** from the context menu.
2. In the Workspace browser, rename the variable `Belgium`.



MATLAB® Expressions



In most data analysis applications, you will want to create variables containing values *computed* from your data. Although you can create new variables with the Variable Editor, you cannot use it to perform calculations, such as the ratio of the first two columns (Belgian and French gas prices). Expressions communicate to MATLAB the computations you want to perform. You enter expressions at the command prompt >> in the Command Window. When evaluated by MATLAB, you can assign the value of an expression to a new variable in the base workspace.

Try

```
>> data(:,2)
>> France = ...
data(:,2);
>> BFratio = ...
Belgium./France;
```

The expression

```
>> data(:,2)
```

is an example of *indexing* into a MATLAB variable. The expression points MATLAB to all (`:`) rows (first argument) in column 2 (second argument) of the variable `data`. Evaluated this way, MATLAB assigns the value of the expression–the 474 values in the second column of `data`–to a temporary variable in the base workspace called `ans`. You can change the name of `ans` in the Workspace browser, but a better practice is to assign the value of the expression to a named variable when it is evaluated.

The expression

```
>> France = data(:,2);
```

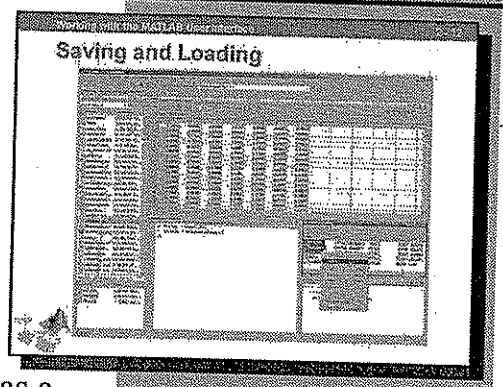
is called an *assignment equality*. MATLAB evaluates the expression to the right of an assignment equality and assigns its value to the variable named on the left of the equality. If the variable already exists, it is over-written. If it does not exist, it is created. The semicolon at the end of the expression simply tells MATLAB to compute the value, assign it, but not display it. Adding a semicolon keeps the display in the Command Window focused on *actions taken*, rather than *results returned*. It also prevents the values of large variables from scrolling down the screen.

The expression

```
>> BFratio = Belgium./France;
```

assigns the value of the element-by-element division (`./`) on the left to the new variable `BFratio` on the right.

Saving and Loading Variables to and from Disk



MATLAB variables remain in the workspace until they are explicitly cleared (for example, by selecting them in the Workspace browser and pressing the delete key) or MATLAB is closed. It is possible to save some (or all) of the variables in the current workspace to disk in a MATLAB-specific format known as a MAT-file. The variables stored in this file can then be loaded into the Workspace later. This provides a quick and easy way to save and load data for use in MATLAB.

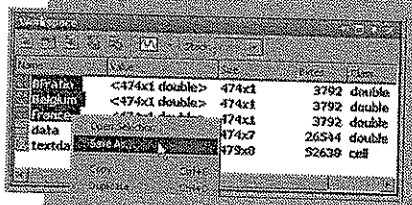
After selecting variables in the Workspace browser, right-clicking produces a context menu containing a Save As... option. The Workspace toolbar includes a Save button ( ) which achieves the same result, as does selecting **File → Save**. The variables will be saved in a MAT-file (`.mat` extension). When a MAT-file is selected in the Current Directory browser, the preview pane shows the variables contained in the file, in a view similar to that of the Workspace browser.

Note that MAT-files save only the data, not the commands or process that created them.

Try

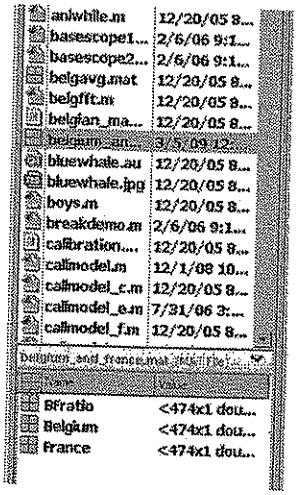
1. Select the variables `Belgium`, `France` and `BFratio` in the Workspace browser (using shift-click or CTRL-click).

2. Save them to a MAT-file called `belgium_and_france.mat`.

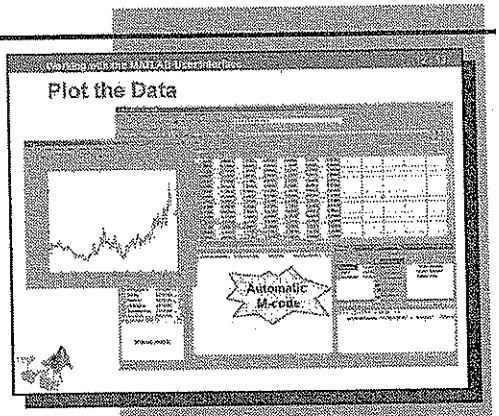


3. Select `France` and `BFratio` in the Workspace browser and press the **Delete** key to clear them from the workspace.

4. Double-click `belgium_and_france.mat` in the Current Directory browser.



Plot the Data



The first step in data analysis is often to create appropriate visualizations of the data. MATLAB offers many plotting routines for both vector and matrix data, and you can access them interactively through the MATLAB desktop.

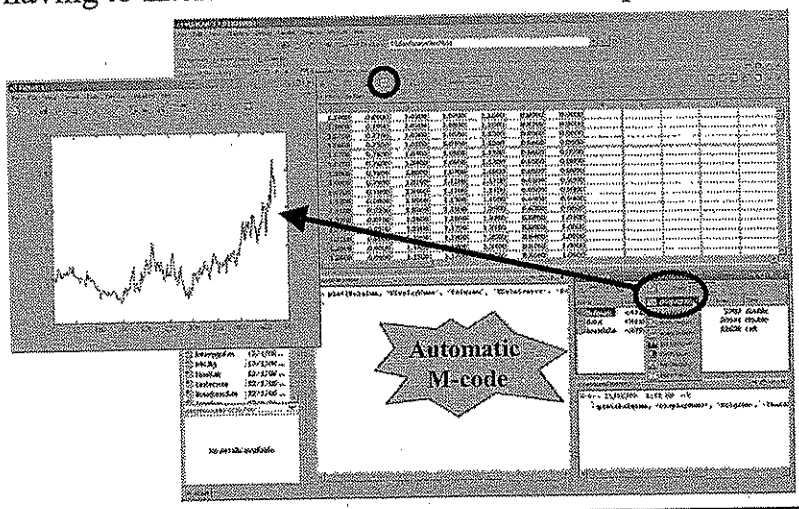
You can plot variables in the base workspace by selecting them in the Workspace browser and choosing a plotting option from the Plot Selection menu ▾ at the top of the browser. Some common plot types, appropriate for the selected data, are listed. Selecting **More Plots** opens a dialog that allows you to browse for further plot types.

Try

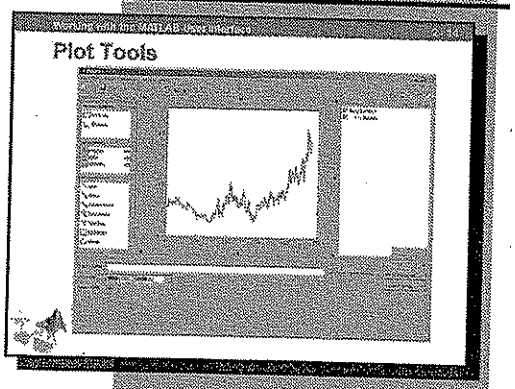
1. Plot the `Belgium` data.
2. Plot all seven countries in `data` simultaneously.

You can plot subsets of your data by selecting a portion of a variable displayed in the Variable Editor and using the Plot Selection menu at the top of the editor. You can select entire columns or rows in a variable by clicking on their numerical headers, or you can drag-select (left-click, hold, and select) ranges of rows and columns. Clicking a value in the Variable Editor, holding down the **Shift** key, and clicking another value, selects the range of data between the two corners.

When you use the Plot Selection menu from either the Workspace Browser or the Variable Editor, MATLAB automatically generates the command that would produce the same result if typed at the command prompt >>, and displays it in the Command Window. This "automatic M-code" is useful for learning MATLAB graphics programming commands. It is used to write MATLAB programs that automate tasks without having to interact with the MATLAB desktop.



Plot Tools



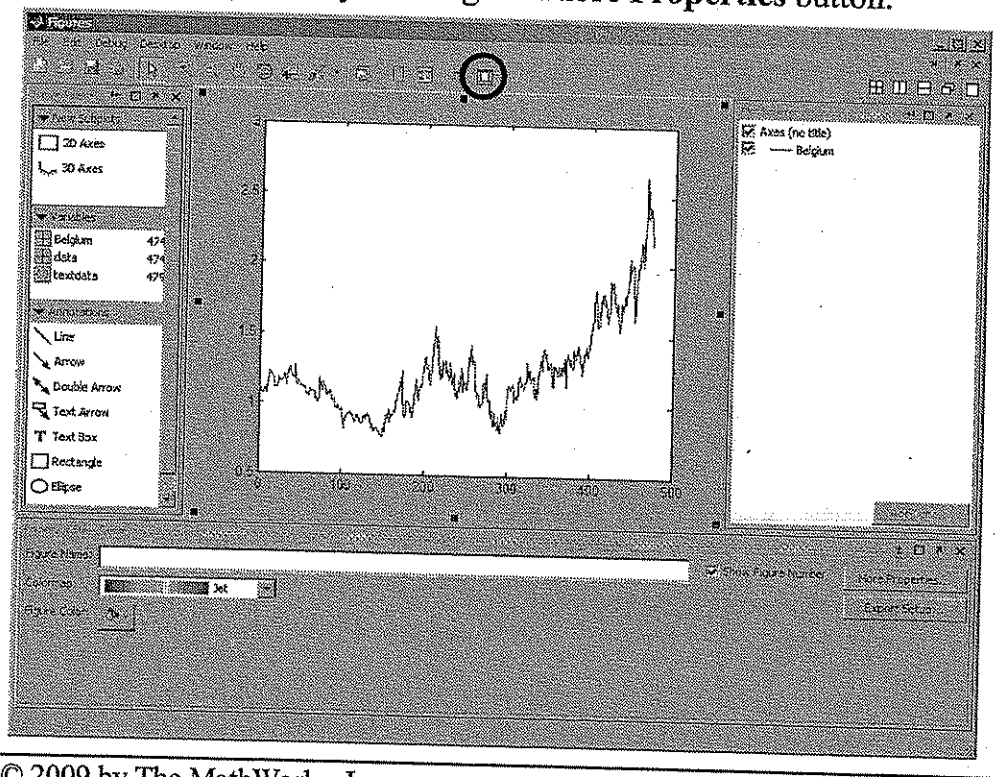
When MATLAB creates an initial plot of your data, you can format the plot so that it conveys exactly the information you want to highlight. The Show Plot Tools button at the top of any MATLAB figure window opens up a variety of tools around the figure to help you format your plot with maximum accuracy.

The **Figure Palette** pane to the left of the figure allows you to

- Add axes to create *subplots* within the figure window.
- Drag and drop variables from the base workspace into any axes.
- Add a variety of annotations to a plot.

The **Plot Browser** pane to the right of the figure allows you to select or deselect individual axes and plots for editing.

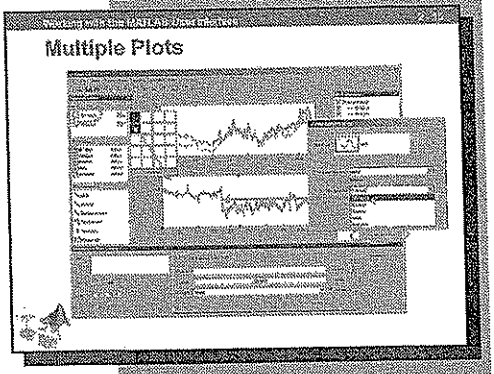
When a plot is selected in the Plot Browser, its individual Property Editor becomes available at the bottom of the figure. The Property Editor shows some of the common properties of that particular plot type, and allows you to change their values interactively. You can view and edit a complete list of the low-level graphics properties of a plot in the Property Inspector by clicking the **More Properties** button.



Try

Open the plot tools on your plot of the `Belgium` data.

Multiple Plots



Multiple data sets can be plotted in the same figure window either overlayed on a single set of axes or by dividing the figure window into multiple axes.

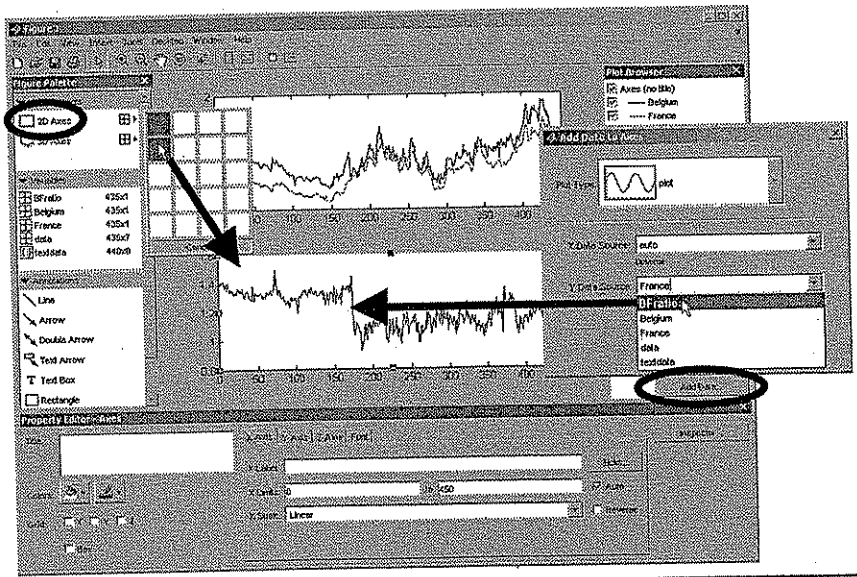
All variables in the base workspace appear in the **Variables** list in the Figure Palette of the Plot Tools. You can drag and drop a variable into a set of axes in the figure to create a new plot, or add it to an existing plot.

Alternatively, you can enter expressions describing new plots directly in the Plot Tools. To do so, toggle down the Edit Plot button in the Figure Toolbar at the top of the figure and click the axes you want to use. Next, click the **Add Data** button at the bottom of the Plot Browser to the right of the figure. A dialog opens that allows you to choose the **X Data Source** and **Y Data Source** for the new plot. Pop-up menus allow you to choose workspace variables for either data source, but you can also enter arbitrary MATLAB expressions.

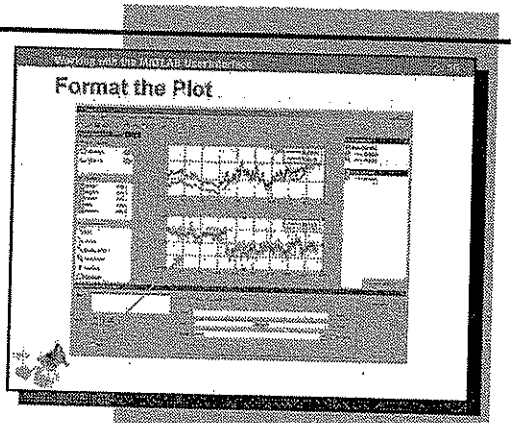
If you need subplots within the figure to display new variables, click the 2-D Axes button ⊞▸ in the New Subplots area of the Figure Palette, and move your mouse to highlight the array of subplots you would like to see in the figure. Click once to add the new axes to the figure.

Try

1. Add the `France` data to the plot of the `Belgium` data.
2. Add a new subplot below the `Belgium-France` plot.
3. Add the calculation for the `BFratio` to the new subplot.



Format the Plot

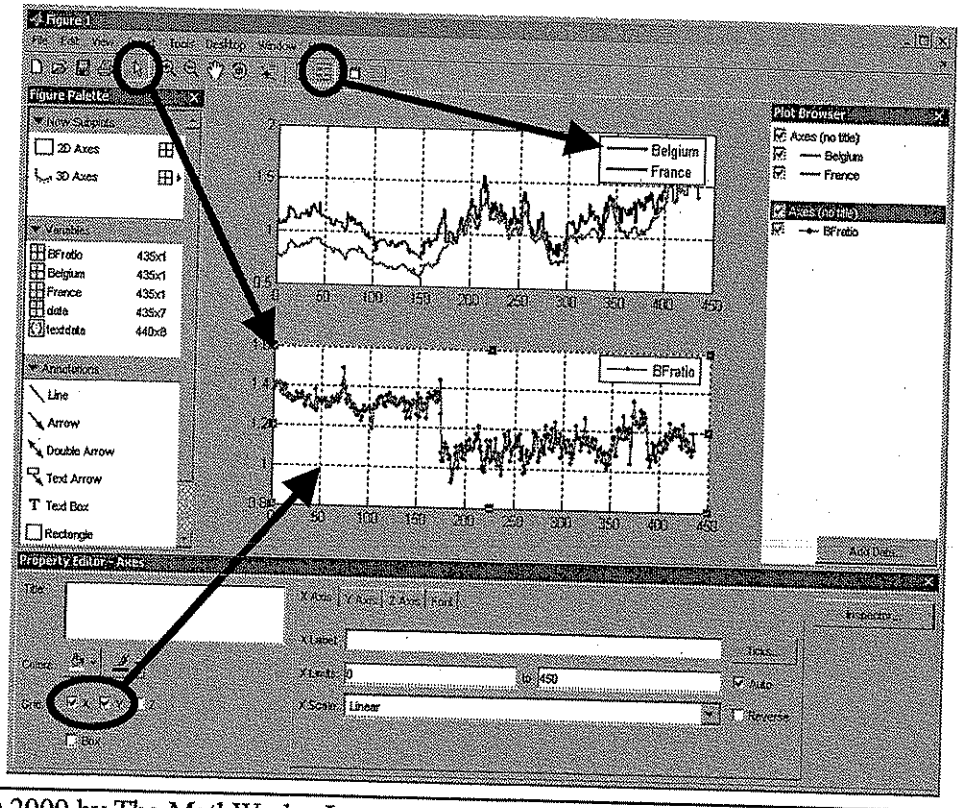


After you have created a basic layout of axes within a figure and have plotted variables within the axes, formatting can enhance the data visualization and emphasize key features.

In addition to the Plot Tools, the Figure Toolbar at the top of the figure offers many more viewing and formatting options:

- Zoom in and out on a portion of a plot.
- Pan along an axis of a plot.
- Rotate a plot in 3-D.
- Use the cursor to display individual data values .
- Add a colorbar displaying color levels in a plot.
- Add a legend to a plot.

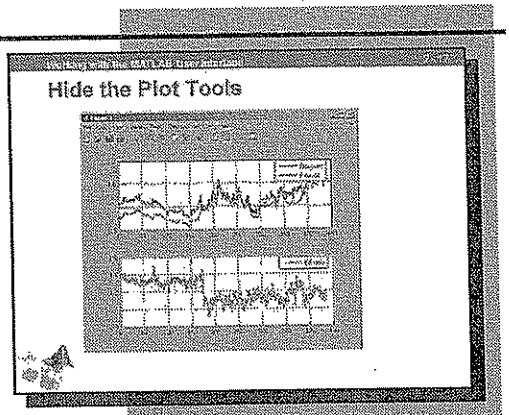
For fine control of the properties of individual graphics objects within a plot, select the object with the Edit Plot button and use the Plot Tools Property Editor at the bottom of the figure.



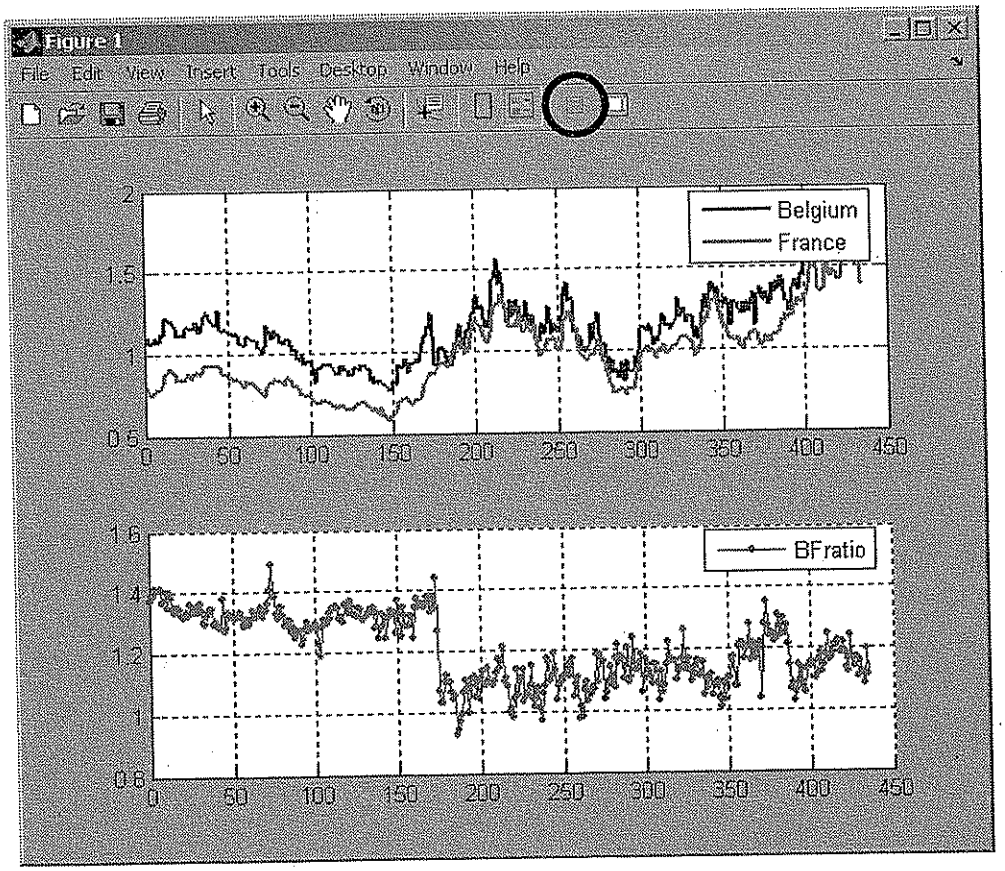
Try

1. Use a heavier line width in the `Belgium/France` plot.
2. Use a contrasting color and marker type in the subplot of the `BFratio`.
3. Add legends to both plots.
4. Add grids to both plots.

Hide the Plot Tools



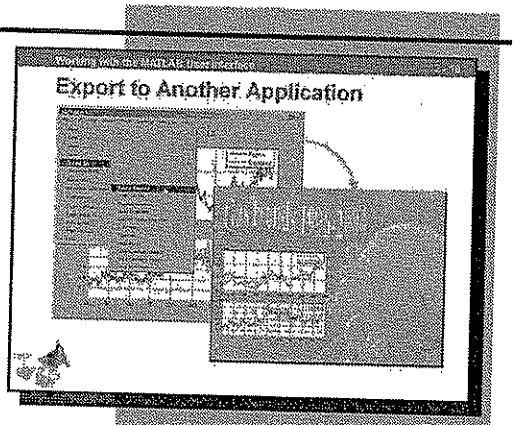
When plotting and formatting are complete, hide the Plot Tools by clicking the Hide Plot Tools button ▫ at the top of the figure. Considered formatting can result in an attention-getting, informative, persuasive presentation of your data.



Try

Hide the Plot Tools.

Export to Another Application



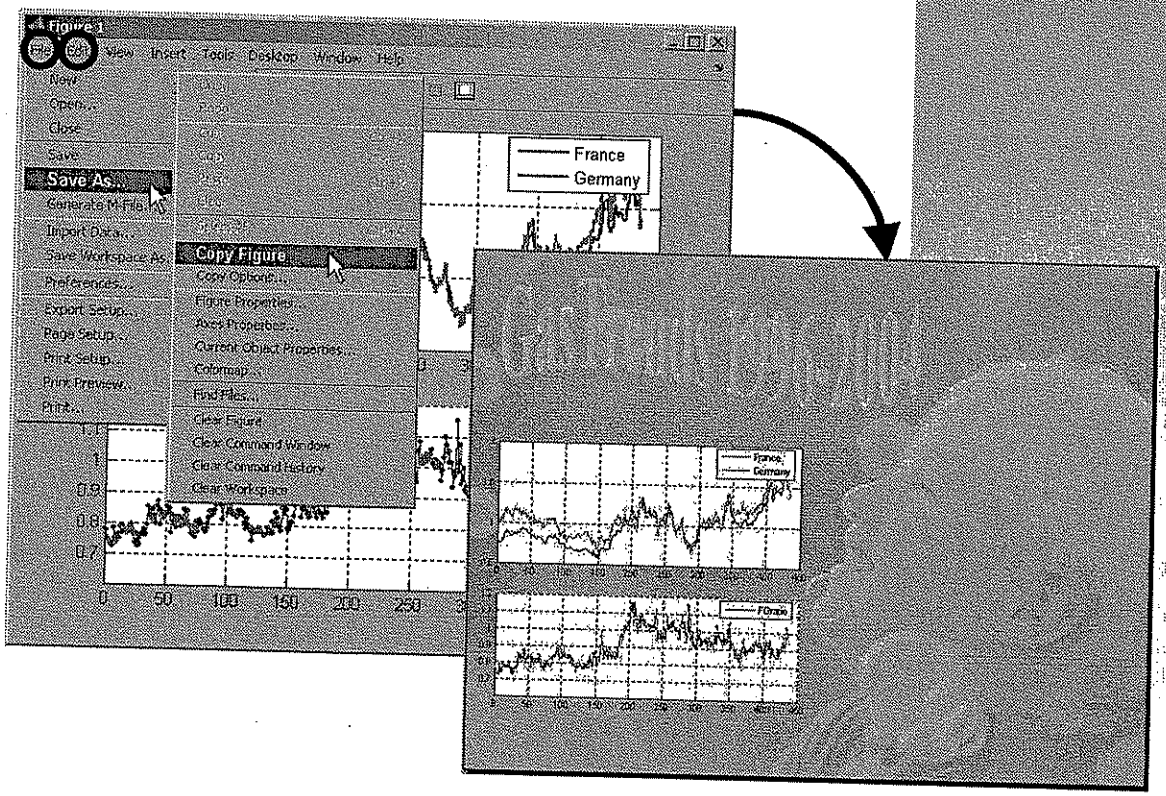
MATLAB figures can be saved, copied, and then exported to applications such as word processors, presentation software, and Web pages.

If you choose **File → Save As** from the menus at the top of a MATLAB figure, a dialog appears that allows you to save the file to a selected directory on your computer. By default, MATLAB uses a binary file format with a `.fig` extension that can be opened quickly in MATLAB. To save to a format appropriate for other applications, use the **Save as Type** pop-up menu in the dialog and select a format.

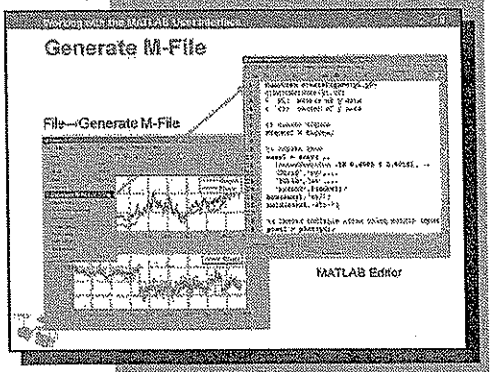
Try

Export your formatted figure to another application such as Word® or PowerPoint®.

On a Windows platform, you can also copy a figure to the Windows clipboard, a part of memory in the Windows address space. Choose **Edit → Copy Figure** from the menus at the top of a MATLAB figure. Choosing **Edit → Copy Options** allows you to set copy preferences in the MATLAB Preferences dialog. When it is copied to the clipboard, you can paste the figures into many applications using **Edit → Paste Special**.



Generate M-File



MATLAB generates automatic M-code when an initial plot is selected from the Plot Selection menu in the Workspace browser or the Variable Editor. You can also generate automatic M-code when formatting is complete.

Because a description of a formatted plot is not usually captured in a single MATLAB expression, MATLAB generates an *M-file* containing a sequence of MATLAB expressions. When they are evaluated in order, these expressions reproduce the plot.

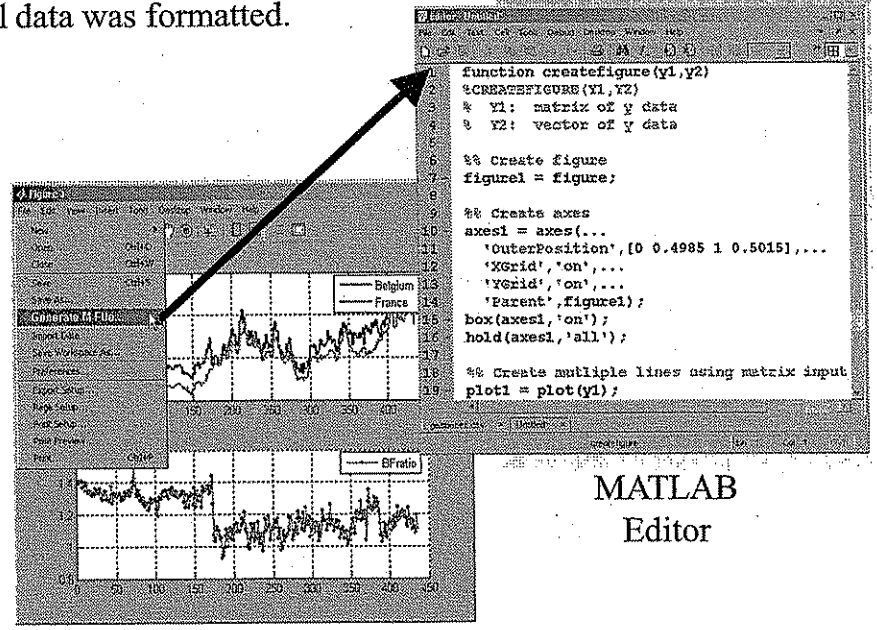
To generate an M-file from your formatted plot, choose **File → Generate M-File** from the menus at the top of the figure. The M-file opens up in the MATLAB Editor. The MATLAB Editor is just a text editor, but its ability to recognize keywords and constructions in the M language makes it the preferred tool for writing MATLAB programs.

M-files come in two types: *scripts* and *functions*. Scripts do the same thing every time the M-code is evaluated. Functions allow you to pass inputs to the code and have the evaluation depend on the inputs. A script suffices for reproducing a particular plot the same way every time, but the automatic M-code generated by MATLAB is a function. This function allows you to pass *new data* to the code and have it formatted exactly the way the original data was formatted.

Try

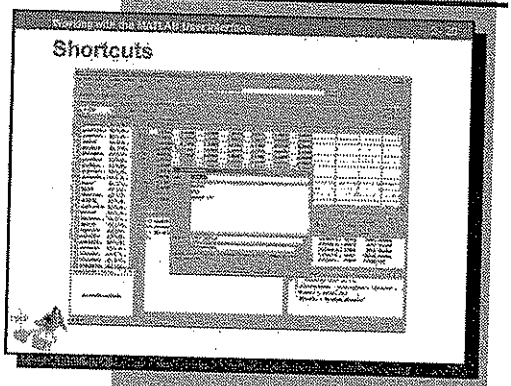
Choose **File → Generate M-File** from the menus at the top of the figure.

File → Generate M-File



MATLAB Editor

Shortcuts



When you are finished with a particular task in MATLAB, you may want to clean up the desktop before moving on to another task. Some useful commands are

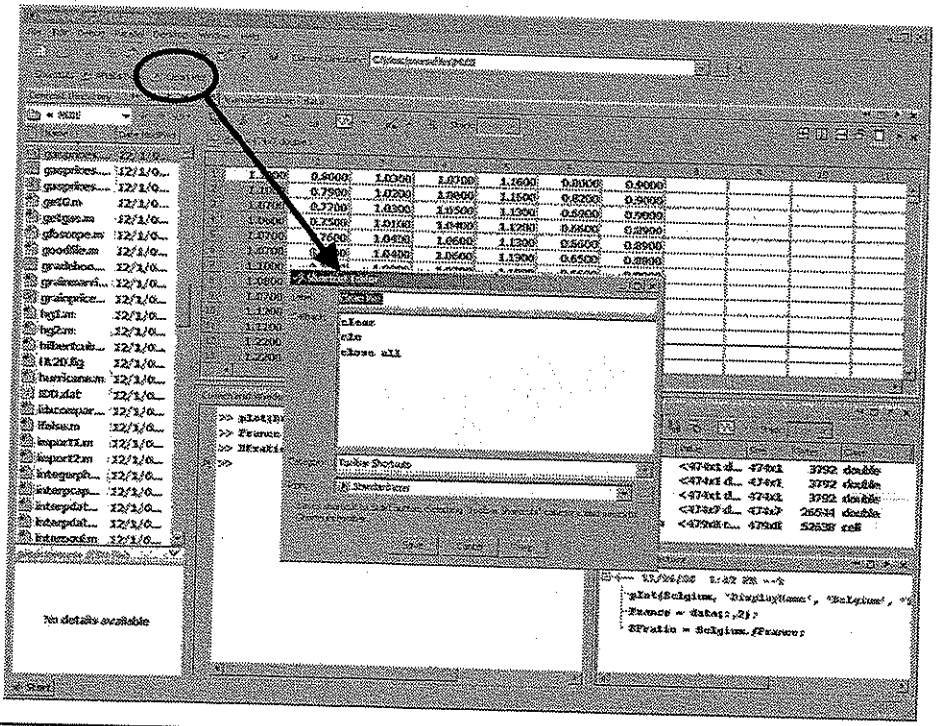
`>> clear` Clears base workspace variables from memory

`>> clc` Clears the Command Window (moves prompt to the top)

`>> close` Closes the current figure

After you begin to use MATLAB regularly, you will probably find other sequences of commands that you execute repeatedly. You could put the commands in an M-file, and call the M-file by name from the command prompt, but MATLAB *shortcuts* allow you to execute the commands at the push of a button.

Shortcut buttons are created by right-clicking in the **Shortcuts** bar at the top of the MATLAB desktop and choosing **New Shortcut** from the context menu. A dialog allows you to enter the sequence of commands that will be executed by the button push (known as a *callback*, as is any code triggered by an event in the user interface). You can also choose a label and an icon for the shortcut button.



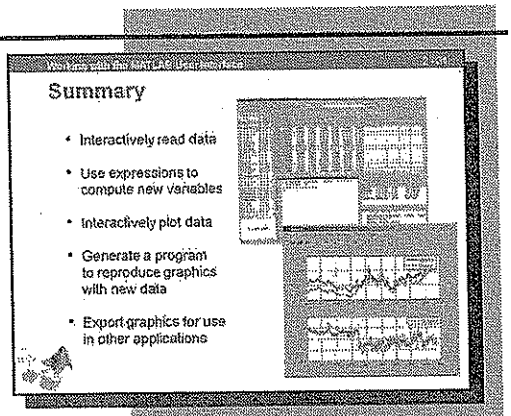
Try

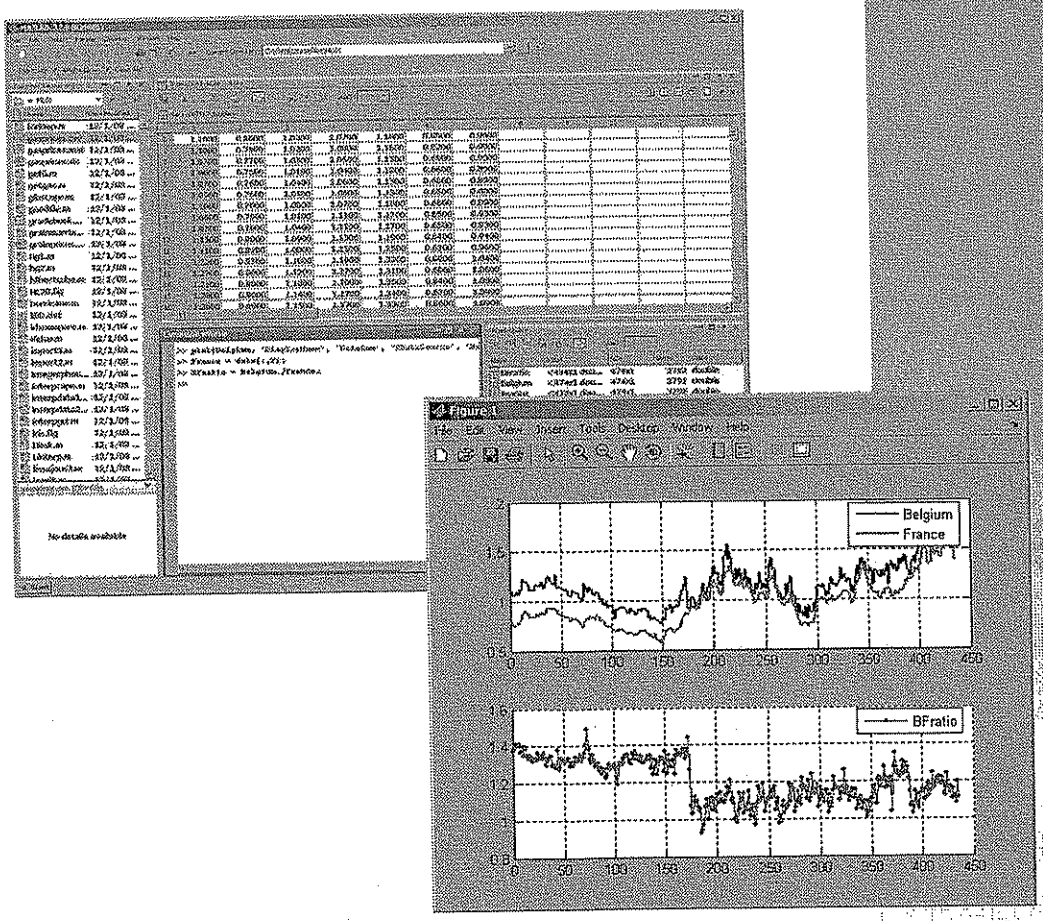
Create a shortcut button that executes the callback

```
clear
clc
close all
```

Summary

- Interactively read data
- Use expressions to compute new variables
- Interactively plot data
- Generate a program to reproduce graphics with new data
- Export graphics for use in other applications

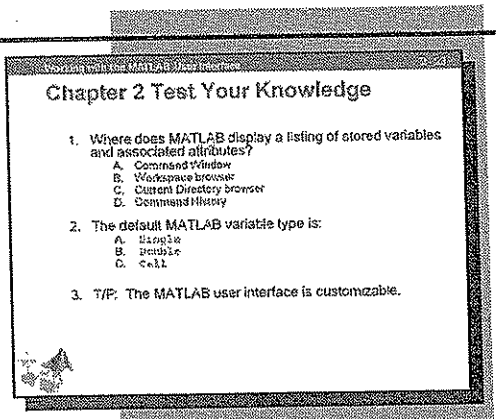




This page intentionally left blank

Chapter 2 Test Your Knowledge

Name:_____



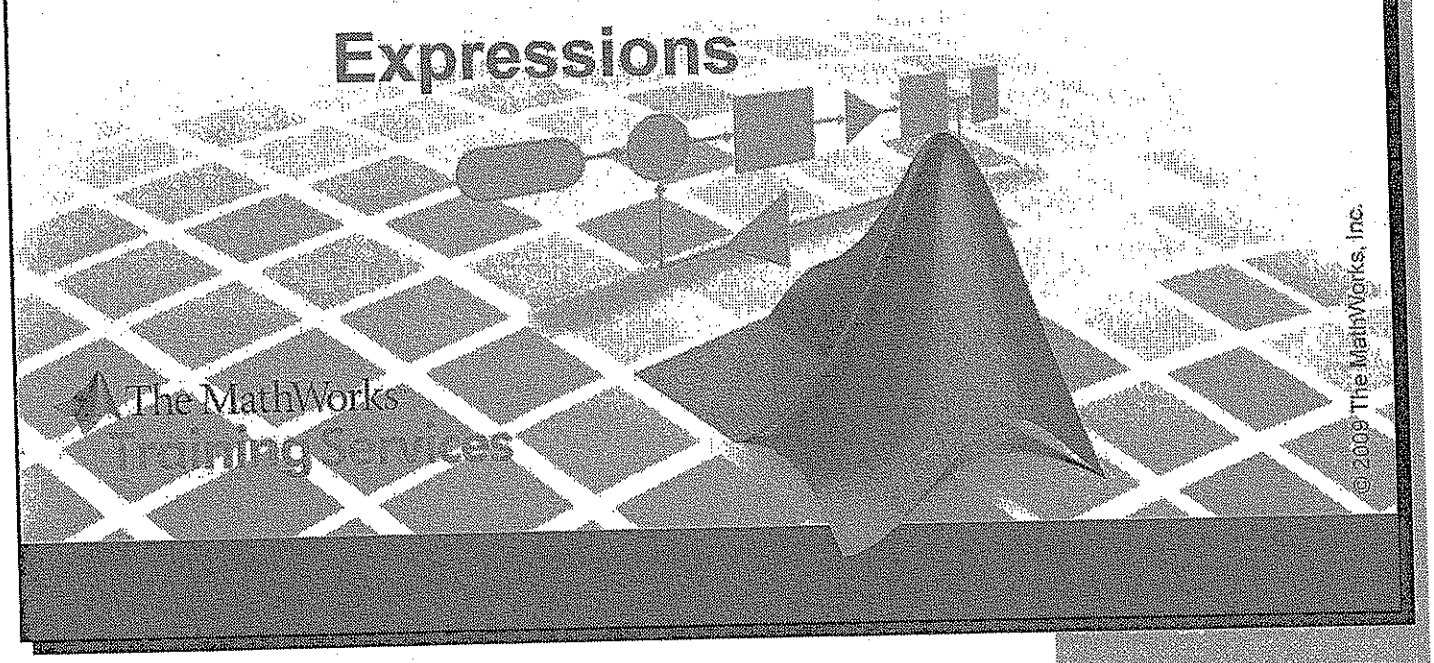
1. Where does MATLAB display a listing of stored variables and associated attributes?
   A. Command Window
   B. Workspace browser
   C. Current Directory browser
   D. Command History

2. The default MATLAB variable type is:
   A. `Single`
   B. `Double`
   C. `Cell`

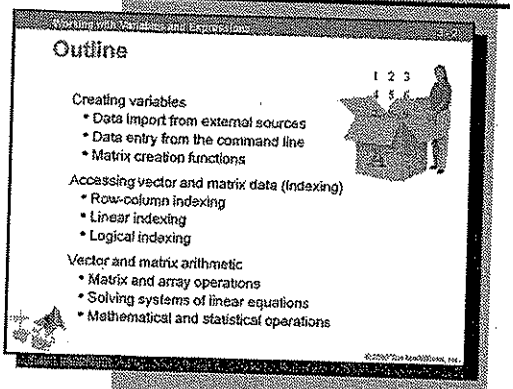
3. T/F: The MATLAB desktop is customizable.

MATLAB® Fundamentals

Working with Variables and Expressions



Outline

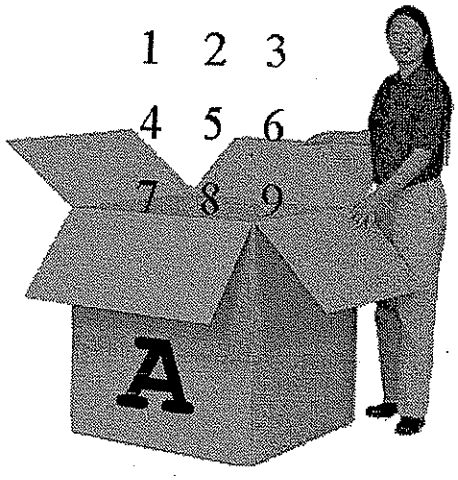


Creating variables
- Data import from external sources
- Data entry from the command line
- Matrix creation functions

Accessing vector and matrix data (indexing)
- Row-column indexing
- Linear indexing
- Logical indexing

Vector and matrix arithmetic
- Matrix and array operations
- Solving systems of linear equations
- Mathematical and statistical operations

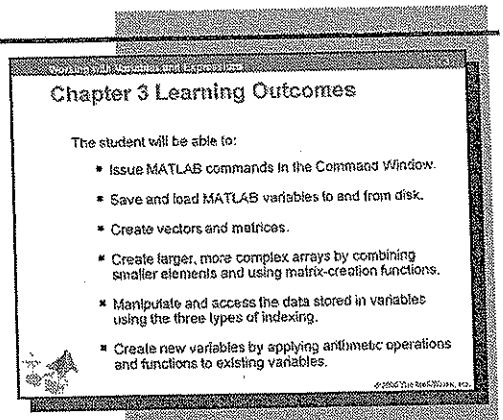
This chapter introduces variables in the MATLAB® environment as data containers. Two essential operations are emphasized: *creating* variables and *accessing* the data the variables contain. The chapter also introduces MATLAB operations for computing with data.



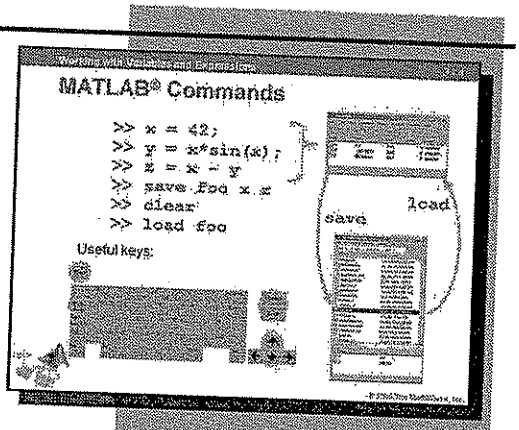
Chapter 3 Learning Outcomes

The student will be able to:

- Issue MATLAB commands in the Command Window.
- Save and load MATLAB variables to and from disk.
- Create vectors and matrices.
- Create larger, more complex arrays by combining smaller elements and using matrix-creation functions.
- Manipulate and access the data stored in variables using the three types of indexing.
- Create new variables by applying arithmetic operations and functions to existing variables.



MATLAB® Commands



MATLAB commands are entered in the Command Window at the MATLAB *prompt* (>>). Commands are executed once the **Enter** key is pressed. There is no required line-ending character in MATLAB. However, three dots (`...`) at the end of a line signify that the command continues onto the next line (thus providing a way to split long commands onto multiple lines).

Any text output from a command is displayed in the Command Window. A semicolon (`;`) at the end of a command suppresses any output.

Although the Command Window shows a record of the commands issued and the output returned, this record is not editable: only the current line can be edited. Furthermore, calculations are performed only as they are entered – there is no retroactive recalculation. Any incorrect calculations must be reentered:

```
>> x = 81;
>> y = sqrt(x);
>> x = 18;
```

Currently `y` still holds the value of 9. To update `y`, the second command must be reentered.

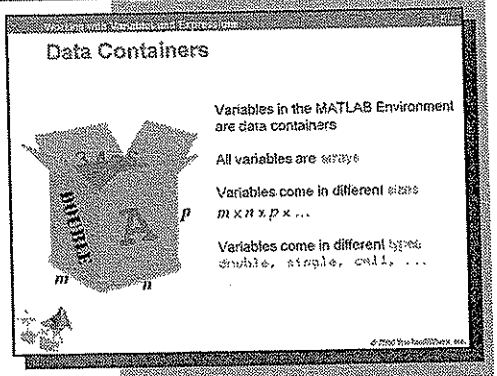
A number of keyboard shortcuts are available to assist in entering commands:

Home, **End**	Move to the beginning, end of an expression
↑, ↓	Move up, down the Command History
Esc	Deletes the expression at the prompt
Tab	Displays commands that complete a partial expression

Try

```
>> x = 42;
>> y = x*sin(x);
>> z = x - y
>> save foo x y
>> clear
>> load foo
```

Data Containers



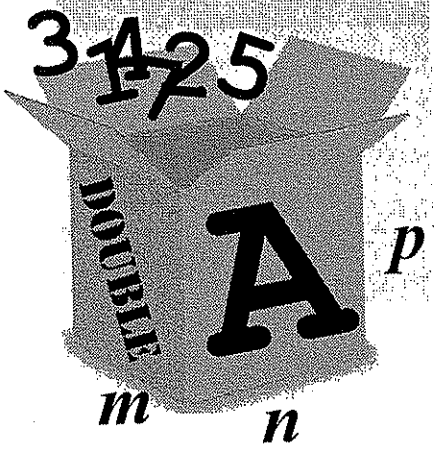
MATLAB is a powerful tool for both data analysis and data modeling. Before MATLAB can do anything with your data, however, the data must reside in the MATLAB environment. MATLAB organizes data in memory in the form of *variables*. You can think of MATLAB variables as data containers.

All MATLAB variables share certain attributes. The specific form of those attributes distinguishes one variable from another.

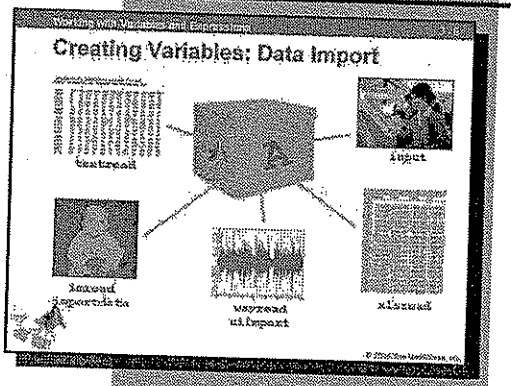
First of all, all MATLAB variables are *arrays*. Arrays ensure that the data in MATLAB variables is always organized, at least at a high level, along a series of linear dimensions. Technically, we say that the fundamental *data type* in MATLAB is a matrix (MATLAB = MATrix + LABoratory). Even scalar variables in MATLAB are treated as 1-by-1 arrays.

The *size* of an array–the number of linear dimensions and the length of the data along each of those dimensions–is an important attribute of any MATLAB variable. MATLAB allows for array size of up to 2^{31} elements; this size is dependant on the hardware architecture of the machine running MATLAB as well as the amount of available memory.

Another import attribute of any MATLAB variable is its *type*. MATLAB supports many different data types, which allow you to organize different kinds of data into different architectures. Choosing an appropriate type for your data container lets you access the data in MATLAB using succinct, intuitive programming commands.



Creating Variables: Data Import



An important component of the MATLAB environment is the ability to read and write data from/to external sources.

MATLAB has extensive capabilities for interfacing directly, in real time, with data from external programs and instrumentation. This course, however, concentrates on reading and writing data that has already been stored in external *files*.

Files come in a variety of standard formats, and MATLAB has specialized routines for working with each of them. To see a list of supported file formats, type

```
>> help fileformats
```

To see a list of associated input/output (I/O) functions, type

```
>> help iofun
```

For convenience, MATLAB provides a graphical user interface to the various I/O functions, called the Import Wizard. You access the Import Wizard by choosing **File → Import Data** or by typing

```
>> uiimport
```

The command `importdata` is a programmatic version of the Import Wizard. It accepts the default choices suggested by the Import Wizard without opening the graphical user interface. Use `importdata` in M-files to read in data from any of the supported file formats.

MATLAB also has a large selection of low-level file I/O functions, modeled after those in the C programming language. These functions allow you to work with unsupported formats by instructing MATLAB to open a file in memory, position itself within the file, read or write specific formatted data, and then close the file.

Try

Supported file formats
```
>> help fileformats
>> help iofun
```

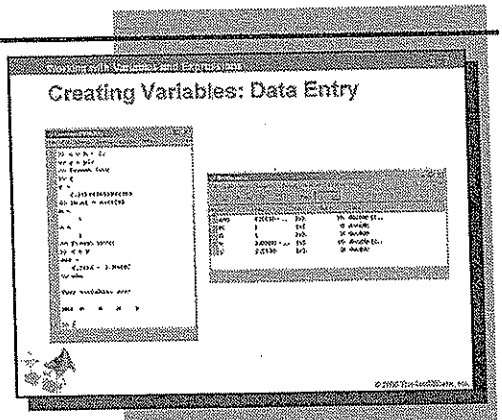
ASCII text data
```
>> fid = fopen( ...
'all_temps.txt','r');
>> jan = textscan( ...
fid,'%*d%d%*[^\n]', ...
'HeaderLines',4);
>> fid=fclose(fid);
```

Spreadsheet data
```
>> [data,text] = ...
xlsread( ...
'stockdata.xls');
>> plot(data(:,3))
>> legend(text(1,4))
```

Image data
```
>> I = ...
importdata( ...
'eli.jpg');
>> image(I)
```

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

Creating Variables: Data Entry



Data can also be entered manually into MATLAB.

The basic mechanism for data entry in MATLAB is the *assignment equality*, in which MATLAB computes data values to the right of the equality and then assigns them to a variable named on the left of the equality. For example,

```
>> x = 1 - i;
>> y = pi;
```

When a computation is carried out but *not* explicitly assigned to a variable, MATLAB assigns the output to the temporary variable `ans`.

The Workspace browser shows that a complex value like `x` takes twice the memory as real values like `y`. MATLAB stores and manipulates the real and imaginary parts of complex data in separate arrays.

By default, numerical data like `x` and `y` are created as *double arrays*. Each data value is stored and manipulated in memory to double (64 bit, 8 byte) precision. Double-precision storage does not affect how the data is displayed in the Command Window. To control the display, you can use the `format` command, or choose **File → Preferences → Command Window** from the menus at the top of the desktop.

Use the `size` command to view the size of a variable:

```
>> [m,n] = size(y)
```

assigns the two outputs of `size`–the number of rows and columns–to separate variables, `m` and `n`, in the base workspace. You can use this syntax with any MATLAB command providing multiple outputs.

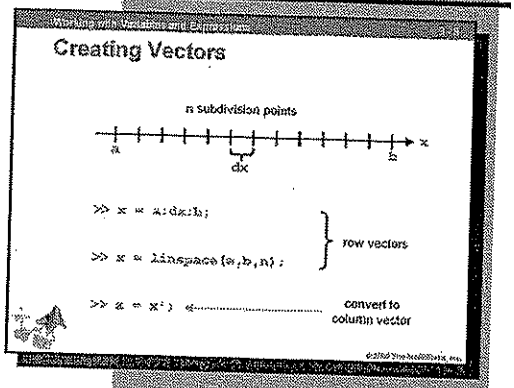
Try

```
>> x = 1 - i;
>> y = pi;
>> format long
>> y
>> [m,n] = size(y)
>> format short
>> x + y
>> who
>> whos
```

Enter
```
>> x
```
and use ↑.

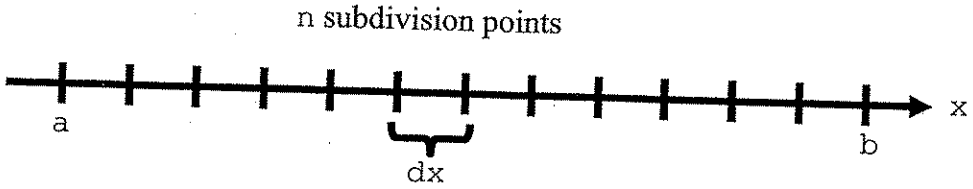
Enter
```
>> fo
```
and use **Tab**.

Creating Vectors



Vectors, or arrays with one dimension of length 1, are the basis for computing and displaying sequences of data in MATLAB.

There are two basic methods for creating equally spaced vectors. The method you choose depends on the information available. Suppose you want to create a vector `x` that begins with the value `a` and ends with the value `b`:



If you know the size of the subdivisions `dx` between values, use

```
>> x = a:dx:b;
```

If you know the number of values (subdivision points) `n`, use

```
>> x = linspace(a,b,n);
```

Both methods create a *row vector*, with dimensions `1`-by-`n`. In the first case (using `dx`), `n` is `1` plus the maximum `k` for which `a + k*dx` ≤ `b`.

To create a *column vector*, with dimensions `n`-by-`1`, simply *transpose* the data with the MATLAB transpose operator:

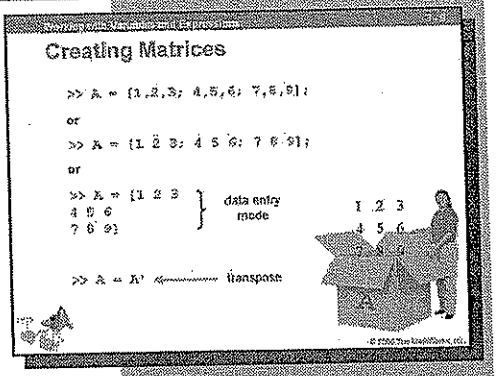
```
>> x = x';
```

Try

Time series

```
>> fs = 8e3;
>> t = 0:1/fs:2;
>> f1 = 300;
>> f2 = 400;
>> f3 = 500;
>> x = ...
sin(2*pi*f1*t) + ...
sin(2*pi*f2*t) + ...
sin(2*pi*f3*t);
>> [m,n] = size(x)
>> soundsc(x,fs)
```

Creating Matrices



To create an m-by-n matrix A in MATLAB, you need to know:

- The values of the m*n elements in A
- The position (index) of each value in A

If you are entering values sequentially from the command prompt, you are entering one row of the matrix after another. You tell MATLAB where the matrix begins and ends, where the rows begin and end, and where each value begins and ends. MATLAB has special *delimiters* for each of these data entry markers.

```
>> A = [1,2,3; 4,5,6; 7,8,9];
```

assigns a 3-by-3 matrix to the variable A. The square brackets delimit the matrix, the semicolons delimit the rows, and the commas delimit the values. Note that the semicolon outside of the brackets is interpreted quite differently by MATLAB; the semicolon suppresses the display.

The commas in the above assignment can be replaced by spaces:

```
>> A = [1 2 3; 4 5 6; 7 8 9];
```

In general, MATLAB ignores spaces, but you can use them both as delimiters and to improve the readability of your expressions.

You can replace the row delimiters (;) by hard returns:

```
>> A = [1 2 3
4 5 6
7 8 9];
```

This mode is especially useful for long data entry, where it is important to check each row against neighboring rows.

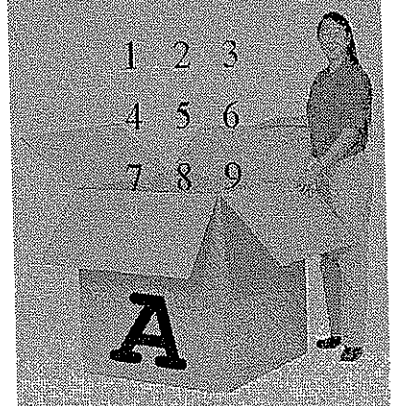
The MATLAB transpose operator works with arrays as well as vectors.

```
>> A = A'
```

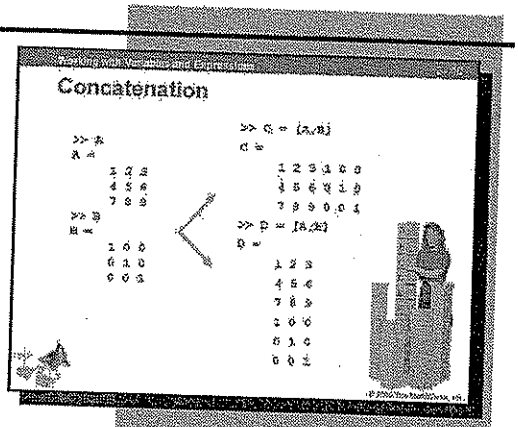
interchanges the rows and columns in A.

Try

```
>> A = [1 2 3
4 5 6
7 8 9]
>> A = A'
>> [m,n] = size(A)
```



Concatenation



The square brackets ([ ]) used to delimit the start and end of a matrix are together called the *concatenation operator*. Their function is actually more general.

Recall that MATLAB views even scalar variables as small, 1-by-1 arrays.

When you enter

```
>> A = [1,2,3; 4,5,6; 7,8,9];
```

you are asking MATLAB to *concatenate* (piece together) 9 separate 1-by-1 arrays into a larger 3-by-3 array. The delimiters tell MATLAB which arrays go side by side (,) and which go one above the other (;).

The arrays being concatenated do not need to be 1-by-1 – they can be of any dimension, so long as arrays with the same number of rows are concatenated side-by-side, and arrays with the same number of columns are concatenated one-above-the-other. Thus if

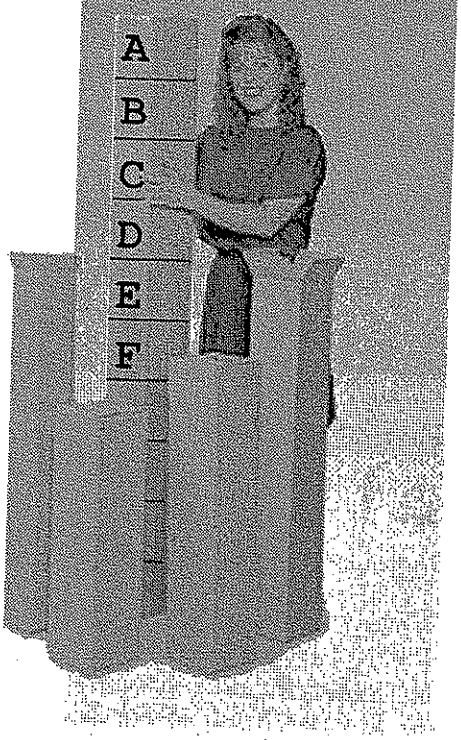
```
>> A = [1,2,3; 4,5,6; 7,8,9];
>> B = [1,0,0; 0,1,0; 0,0,1];
```

then

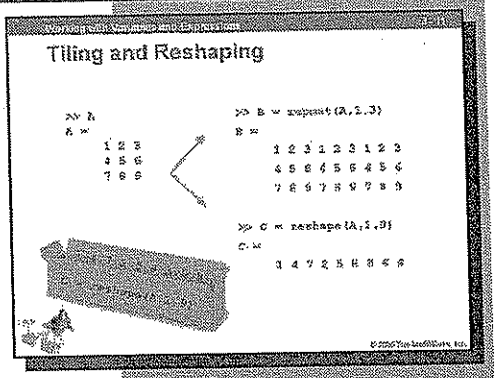
```
>> C = [A,B]
C =
     1     2     3     1     0     0
     4     5     6     0     1     0
     7     8     9     0     0     1

>> D = [A;B]
D =
     1     2     3
     4     5     6
     7     8     9
     1     0     0
     0     1     0
     0     0     1
```

Try

```
>> A = ...
[1 2 3
4 5 6
7 8 9]
>> B = [1 0 0
0 1 0
0 0 1]
>> C = [A,B]
>> D = [A;B]
```



Tiling and Reshaping



One kind of concatenation that occurs frequently when building MATLAB variables is the *tiling* of a single array, horizontally and vertically. The MATLAB `repmat` function (for *rep*licate *mat*rix) automates this operation.

```
>> repmat(A,m,n)
```

replicates the array `A` in an `m`-by-`n` tiling. Thus if

```
>> A = [1,2,3; 4,5,6; 7,8,9];
```

then

```
>> B = repmat(A,1,3)
B =
     1  2  3  1  2  3  1  2  3
     4  5  6  4  5  6  4  5  6
     7  8  9  7  8  9  7  8  9
```

Concatenation is just one method of reshaping your data into a variable of a different size. Another useful MATLAB function for this purpose is `reshape`.

```
>> reshape(A,m,n)
```

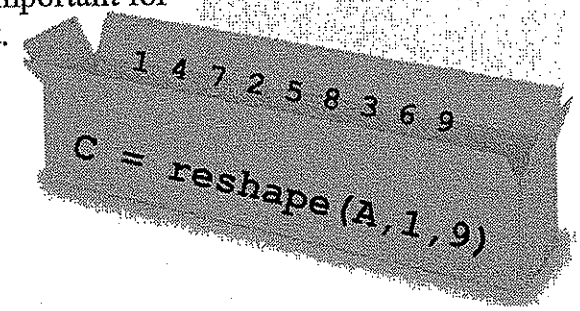
puts the data in `A` into a new variable with dimensions `m`-by-`n`. To do this, of course, `A` must contain `m*n` elements. Thus

```
>> C = reshape(A,1,9)
C =
     1  4  7  2  5  8  3  6  9
```

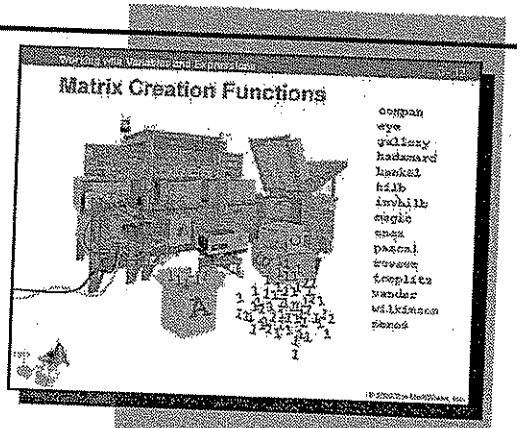
Notice that the `reshape` function does not specify the *order* of the elements in the reshaped array. No order is specified because every MATLAB array has an implicit linear ordering, down the columns from left to right, which is the ordering that MATLAB uses to store array elements in memory. Understanding this ordering is important for certain operations with variables, like data input and output.

Try

```
>> A = ...
[1 2 3
4 5 6
7 8 9]
>> B = repmat(A,1,3)
>> C = ...
reshape(A,1,9)
```



Matrix Creation Functions



MATLAB has a large number of *matrix creation functions* that output variables with certain characteristics and avoid tedious data entry. Often, the matrices created serve as building blocks, or starting points, for other MATLAB variables.

The basic syntax for matrix creation functions is

```
>> A = function_name(m,n);
```

or

```
>> A = function_name(n);
```

In the first case, the function creates a matrix `A` with dimensions `m`-by-`n`, populated by elements whose characteristics are determined by the particular function. In the second case, a square `n`-by-`n` matrix is created.

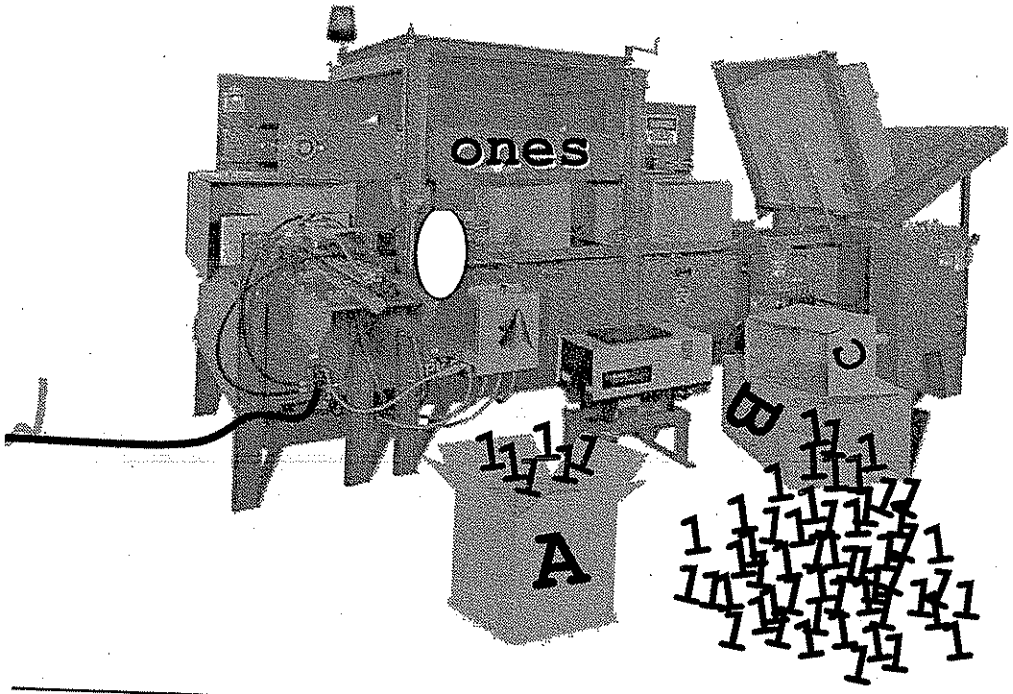
Typing

```
>> help elmat
```

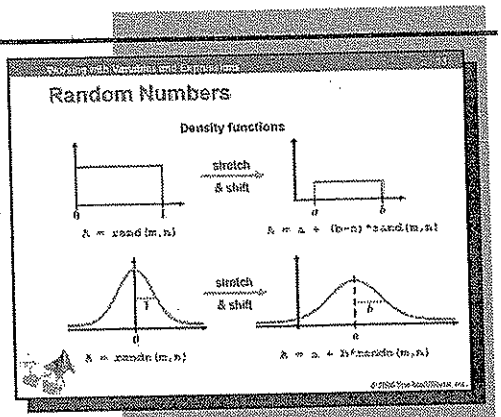
lists some of the elementary matrix functions in MATLAB, including a number of matrix creation functions.

Try

```
>> help elmat
>> help gallery

>> A = ones(2,4)
>> B = zeros(3)
>> C = eye(3)
>> D = magic(3)

Leslie population matrix
>> a = [0 2 0];
>> b = [0.6 0.8];
>> L = gallery(...
'leslie',a,b)
```



Random Numbers



An extremely useful pair of matrix creation functions is the two fundamental random number generators in MATLAB.

```
>> rand(m,n)
```

returns an m-by-n array of uniformly distributed random numbers between 0 and 1. To generate uniformly distributed random numbers between arbitrary bounds a and b, simply stretch and shift the density:

```
>> a + (b-a)*rand(m,n)
```

Similarly,

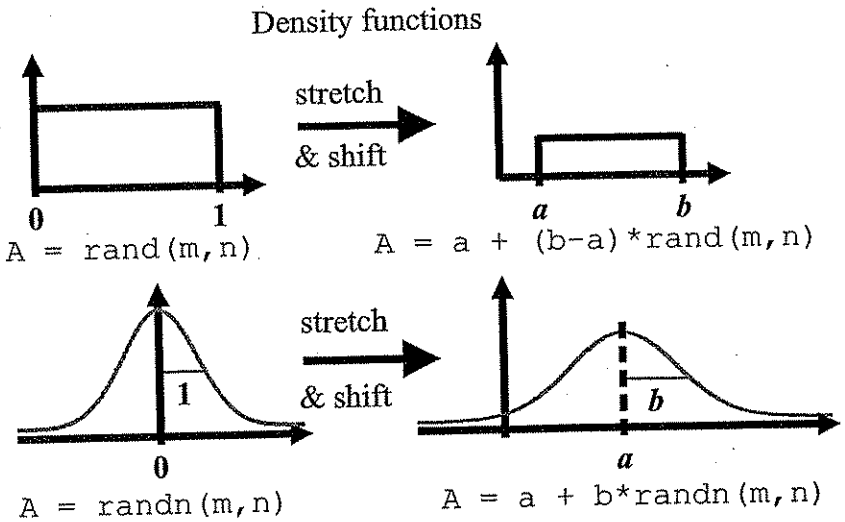
```
>> randn(m,n)
```

returns an m-by-n array of normally distributed random numbers from the standard normal distribution with mean 0 and standard deviation 1.

```
>> a + b*randn(m,n)
```

returns normally distributed random numbers from the normal distribution with mean a and standard deviation b.

You can generate random numbers from other distributions with `rand` and the inverse cumulative distribution function. The Statistics Toolbox™ adds dozens more random number generators to MATLAB.



Try

```
>> x = rand(1,1e6);
>> hist(x,100)
>> y = randn(1,1e6);
>> hist(y,100)
```

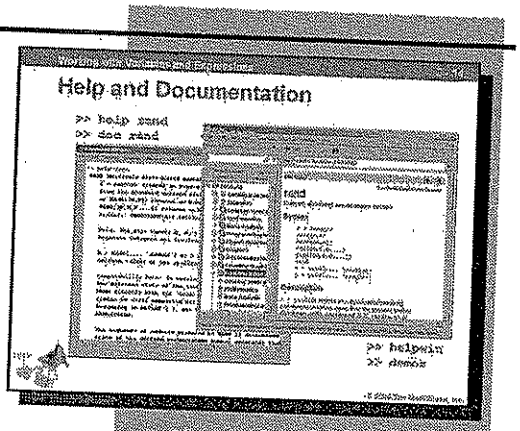
Playing dice

```
>> dice =
randi(6,10,5000);
>> d1 = dice(1,:);
>> d2 = ...
sum(dice(1:2,:));
>> d10 = sum(dice);
>> hist(d1,1:6)
>> hist(d2,2:12)
>> hist(d10,10:60)
```

Signal plus noise

```
>> fs = 8e3;
>> t = 0:1/fs:2;
>> f1 = 300;
>> f2 = 400;
>> f3 = 500;
>> x = ...
sin(2*pi*f1*t) + ...
sin(2*pi*f2*t) + ...
sin(2*pi*f3*t);
>> n = ...
0.5*randn(size(x));
>> sound(x+n,fs)
```

Help and Documentation



MATLAB help and documentation can show you

- The various ways to call a function
- The algorithm implemented by a function
- Examples of how to use a function
- Links to related functions
- Tutorials and background information

To see the help on a particular function, such as `rand`, type

```
>> help rand
```

Basic help for the function is displayed to the Command Window.

For complete documentation on the function, type

```
>> doc rand
```

The MATLAB Help browser opens to the appropriate documentation.

If you want to search the documentation by topic, rather than by function name, type

```
>> doc
```

to open the Help browser to its title page. You can then use the Help Navigator on the left to browse by topic or search by keywords.

If you prefer to browse the available functions by category (such as "elementary math functions"), entering

```
>> helpwin
```

opens the Help browser to a list of function categories.

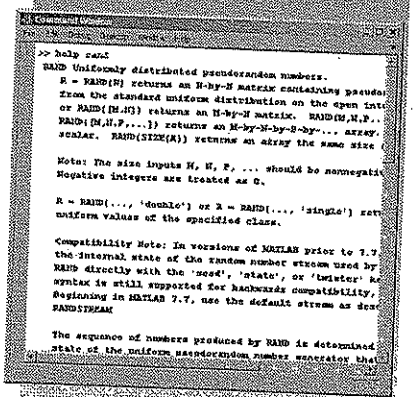
The **Demos** tab of the Help Navigator is a valuable resource for MATLAB users at any level. Rather than focusing on a particular function, demos put together MATLAB functions to show you how to approach common problems in many applications.

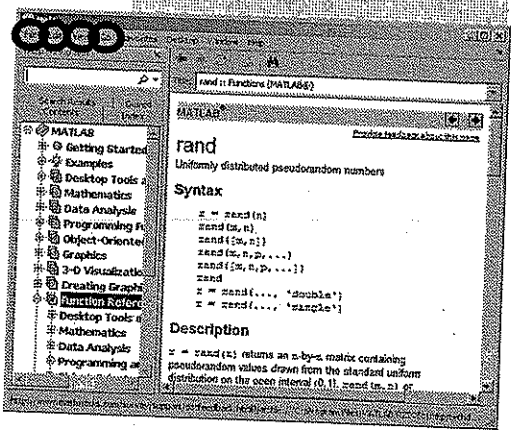
Try

```
>> help rand
>> doc rand
>> demos
```

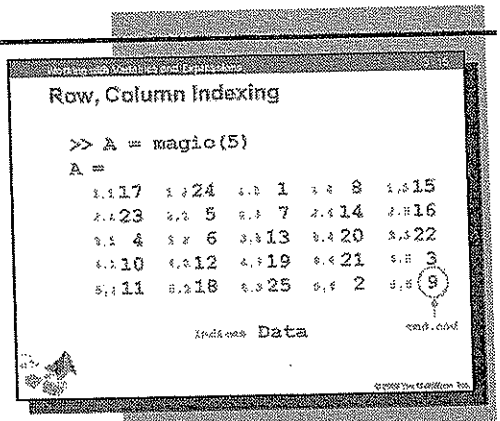
Setting the state of `rand`

```
>> doc RandStream
```





Row, Column Indexing



Data in MATLAB variables is accessed using one of several methods of *indexing*. The most common method of indexing is *row, column indexing*, in which every element of an array is designated by its unique (row number, column number) pair.

The syntax

```
>> A(m,n)
```

points MATLAB to the `m`th row and `n`th column of the variable `A`.

When combined with an assignment, indexing is used to access or alter the data in a variable.

```
>> x = A(m,n);
```

is called a *subscripted reference*, and it assigns the `(m,n)` element of `A` to the variable `x`.

```
>> A(m,n) = x;
```

is called a *subscripted assignment*, and it assigns the value `x` to the `(m,n)` element of `A`.

You can indicate a range of rows or columns in a variable using the colon (`:`) operator. For example, if `A` is a 5-by-5 array,

```
>> x = A(1:5,5);
```

assigns the entire 5th column of `A` to `x`. A shorthand for all rows or columns is the colon operator without beginning and end, so that

```
>> x = A(:,5);
```

also assigns the entire 5th column ("all rows, 5th column") of `A` to `x`.

Row and column indices in MATLAB variables always begin with 1. Where they end, of course, depends on the dimensions of the variable. You can use the MATLAB keyword `end` in either a row or column index to write code that will adapt to variables of changing dimension.

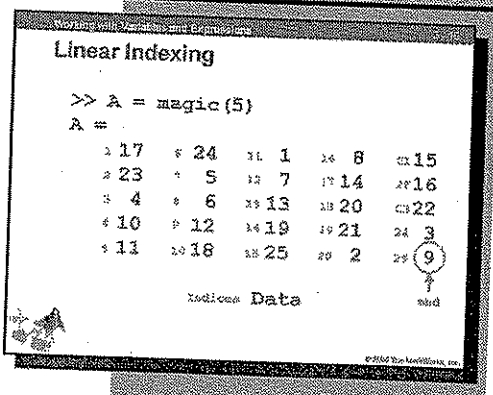
Try

```
>> A = magic(5)
>> a = A(2,3)
>> A(2,3) = 0
>> x = A(1:3,5)
>> y = A(:,5)
>> B = A(...
end-1:end,end-1:end)
```

Adding and removing rows and columns

```
>> A = magic(5)
>> A(:,6) = 1:5
>> A(5,:) = []
```

Linear Indexing



Every MATLAB array has an implicit *linear indexing*, down the columns, then left to right. MATLAB uses this order to store the data in a variable in memory. Row, column indexing is far more common in MATLAB code, because it is more intuitive and easier to read. It helps to know about the linear indices of MATLAB variables, however, and their occasional application.

The syntax

```
>> A(k)
```

points MATLAB to the `k`th element in the linear order of `A`.

One use of linear indexing is to use the colon operator (`:`) to indicate all elements in an array, in linear order. Typing

```
>> a = A(:);
```

puts all of the data in the array `A` into a column vector `a`.

Try

```
>> A = magic(5)
>> a = A(12)
>> A(12) = 0
>> x = A(21:23)
>> y = A(end-4:end)
>> y = y'
```

Row, column indexing

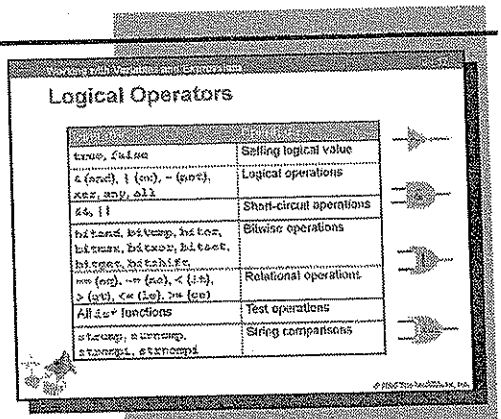
```
>> A = magic(5)
A =
 1,1 17  1,2 24  1,3  1  1,4  8  1,5 15
 2,1 23  2,2  5  2,3  7  2,4 14  2,5 16
 3,1  4  3,2  6  3,3 13  3,4 20  3,5 22
 4,1 10  4,2 12  4,3 19  4,4 21  4,5  3
 5,1 11  5,2 18  5,3 25  5,4  2  5,5  9
                                    ↑
                                 end,end
```

Linear indexing

```
>> A = magic(5)
A =
 1 17   6 24   11  1   16  8   21 15
 2 23   7  5   12  7   17 14   22 16
 3  4   8  6   13 13   18 20   23 22
 4 10   9 12   14 19   19 21   24  3
 5 11  10 18   15 25   20  2   25  9
                                   ↑
                                  end
```

Indices **Data**

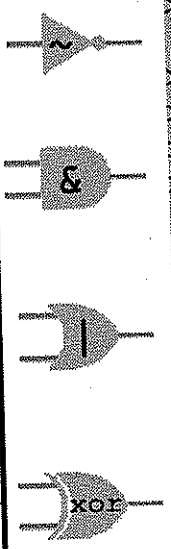
Logical Operators



MATLAB uses logical `1` and logical `0` (or the system variables `true` and `false`) to denote *logical values*. Variables of logical values are distinguished by a *logical* data type.

Logical expressions involving MATLAB variables are built up using the *logical operators* listed in the table below. When MATLAB evaluates logical expressions, it returns logical values. The size of the logical variable containing the logical values depends on the size of the variables in the logical expression.

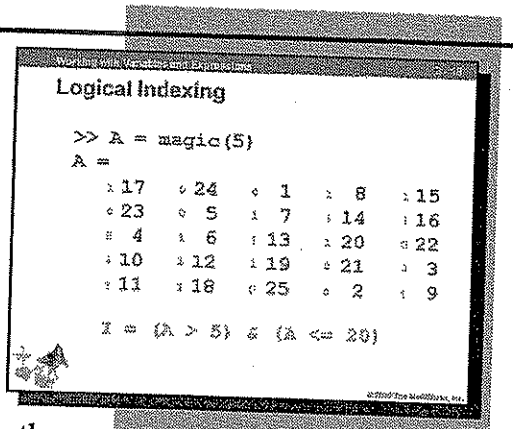
Operator	Operation
`true, false`	Setting logical value
`&` (and), `|` (or), `~` (not), `xor, any, all`	Logical operations
`&&, ||`	Short-circuit operations
`bitand, bitcmp, bitor, bitmax, bitxor, bitset, bitget, bitshift`	Bitwise operations
`==` (eq), `~=` (ne), `<` (lt), `>` (gt), `<=` (le), `>=` (ge)	Relational operations
All `is*` functions	Test operations
`strcmp, strncmp, strcmpi, strncmpi`	String comparisons



Try

```
>> help relop
>> help ops

>> 0 < 1
>> 0 > 1

>> dec2bin(4)
>> dec2bin(5)
>> and(4,5)
>> bitand(4,5)

>> x = 1;
>> isnumeric(x)
>> islogical(x)
>> (x <= 2) & (x | 0)

>> zeros(2) == eye(2)
>> magic(5) >= 10

>> strcmp(...
'foo','fob')
>> strncmp(...
'foo','fob',2)
```

Logical Indexing



Logical operators are central to any programming language. Among other things, logical operators introduce another method for accessing the data in MATLAB variables.

Imagine wanting to access the values in a MATLAB variable that are greater than 0. More generally, imagine wanting to access the values that satisfy *any* logical condition. Unless the locations where the condition evaluates to `true` fall neatly into a contiguous, known range of rows and columns, or a contiguous, known range of linear indices, a method of *logical indexing* is required to access the values.

Logical indexing uses *logical indices*, which are logical variables the same size as the variable being accessed. To create a logical index, you evaluate a logical expression involving the variable being accessed.

```
>> I = (A > 0);
```

returns a logical array `I` the same size as `A`, with logical `1`s in the locations where the elements of `A` are positive, and logical `0`s everywhere else. The syntax

```
>> A(I)
```

points MATLAB to the positive elements of `A`. Note that `A` is a *numeric* array and `I` is a *logical* array. Also note that `I` could be the result of evaluating *any* logical condition on `A`.

Like any indexing method, you must combine logical indexing with an assignment equality in a subscripted reference or a subscripted assignment to extract or alter the elements of `A`, respectively.

Logical indexing

```
>> A = magic(5)
A =
    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
```

```
I = (A > 5) & (A <= 20)
```

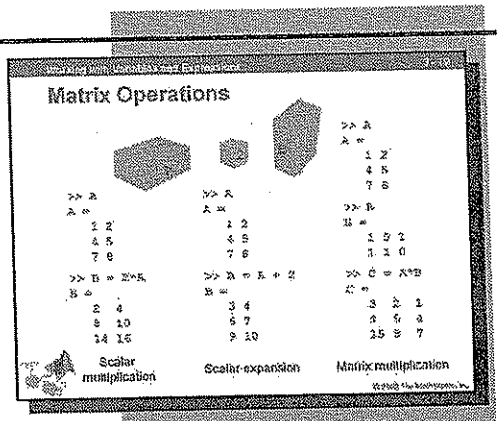
Try

```
>> A = magic(5)
>> I = ...
(A > 5) & (A <= 20)
>> x = A(I)
>> A(~I) = 0
>> J = eye(5)
>> A(J)
>> J = logical(J)
>> A(J)
```

Finding indices

```
>> edit findindx
>> findindx
```

Matrix Operations



When evaluating simple arithmetic operations such as addition, subtraction, and multiplication, MATLAB assumes, as usual, that operands are matrices, and it performs matrix operations.

Thus, to evaluate

```
>> C = A + B;
```

MATLAB requires that `A` and `B` have the same dimensions, and it performs the addition or subtraction element by element to produce an array `C` the same size as `A` and `B`.

Operations with scalars are treated as special cases, for convenience. Scalar multiplication, of the form

```
>> C = k*A;
```

where `k` is a scalar and `A` is an array of arbitrary size, multiplies each element of `A` by the factor `k`. Additions and subtractions of the form

```
>> C = A + k;
```

are interpreted as

```
>> C = A + k*ones(size(A));
```

This *scalar expansion* of `k` to the same size as `A` is a notational shorthand that can be used to simplify your code.

Matrix multiplication of the form

```
>> C = A*B
```

requires that the *inner dimensions*–the number of columns in `A` and the number of rows in `B`–agree. If `A` is `m`-by-`n` and `B` is `n`-by-`p`, then `C` will be `m`-by-`p`. MATLAB performs the usual matrix multiplication, succinctly expressing linear combinations of the elements in `A` and `B`.

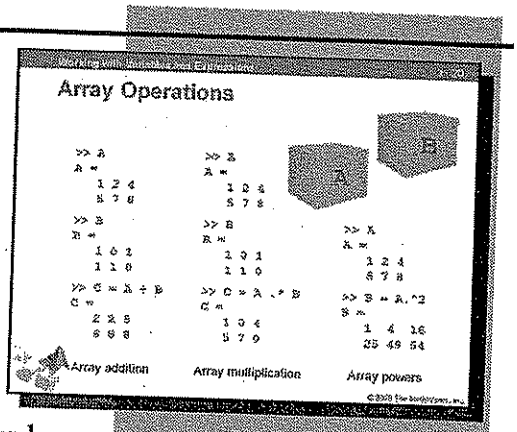
Try

```
>> A = reshape(...
randperm(9),3,3)
>> B = reshape(...
randperm(9),3,3)
>> C = A + B
>> D = A - B
>> E = 2*A
>> F = A + 2
>> G = A*B
>> H = B*A
```

Leslie population model

```
>> a = [0 2 0];
>> b = [0.6 0.8];
>> L = gallery(...
'leslie',a,b)
>> x0 = [10;10;10]
>> P0 = sum(x0)
>> x1 = L*x0
>> P1 = sum(x1)
>> x2 = L*x1
>> P2 = sum(x2)
>> x10 = (L^10)*x0
>> P10 = sum(x10)
```

Array Operations



Addition and subtraction are examples of *array operations* in MATLAB–they require operands of the same shape and size and they operate element by element. Other operations, such as multiplication and powers, do not have these requirements.

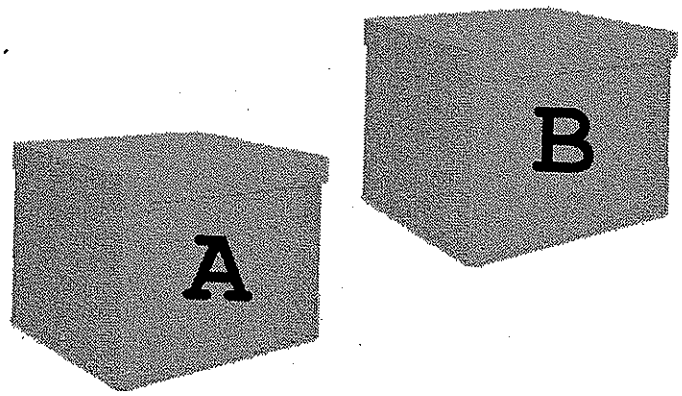
MATLAB allows you to perform array multiplications and powers by over-riding the default matrix operations. To do this, simply prefix the operator with a dot (.).

```
>> C = A.*B;
```

multiplies arrays A and B of the same size element by element. Similarly,

```
>> C = A.^k
```

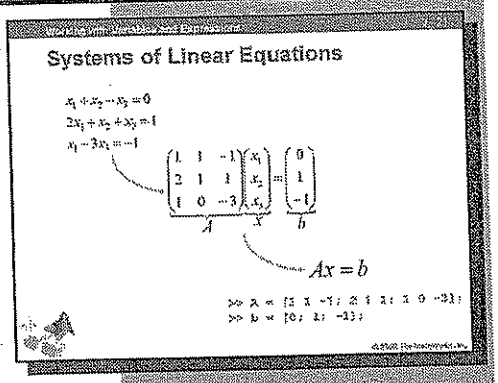
raises each element of A to the k th power, rather than multiplying A by itself k times.



Try

```
>> help arith

>> x = 1:5
>> y = x.^2 + 2

>> volume = rand(5)
>> density = rand(5)
>> mass = ...
volume.*density
```

Systems of Linear Equations



You can express systems of linear equations succinctly in MATLAB by creating variables for the coefficient matrix and the vector of constants on the right hand side of the equations.

For example,

$x_1 + x_2 - x_3 = 0$

$2x_1 + x_2 + x_3 = 1$

$x_1 - 3x_3 = -1$

can be written as

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

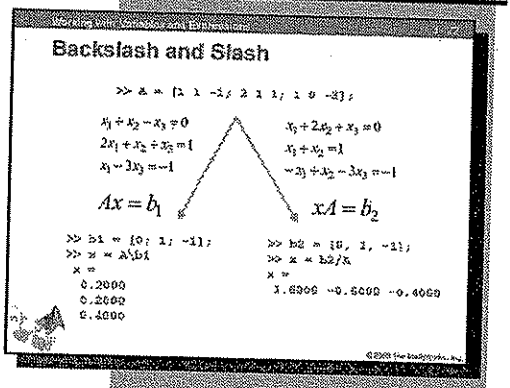
or as

$Ax = b.$

In MATLAB, the system is expressed by:

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

Backslash and Slash



Because division is not a standard matrix operation, MATLAB reserves the division operators for special purposes. MATLAB uses the backslash operator (\) to solve systems of linear equations of the form $Ax = b$. The slash operator (/) is used to solve systems of linear equations of the form $xA = b$.

For example, if

```
>> A = [1 1 -1; 2 1 1; 1 0 -3];
>> b1 = [0; 1; -1];
```

represents the system $Ax = b_1$, it is solved by typing

```
>> x = A\b1
 x =
  0.2000
  0.2000
  0.4000
```

Likewise, if

```
>> A = [1 1 -1; 2 1 1; 1 0 -3];
>> b2 = [0, 1, -1];
```

represents the system $xA = b_2$, it is solved by typing

```
>> x = b2/A
 x =
  1.6000 -0.6000 -0.4000
```

The notation for these operators can be remembered by comparing \ to left multiplication by A^{-1} ($A^{-1}Ax = A^{-1}b \rightarrow x = A^{-1}b \rightarrow x = A\backslash b$) and / to right multiplication by A^{-1} ($xAA^{-1} = bA^{-1} \rightarrow x = bA^{-1} \rightarrow x = b/A$). Under no circumstances, however, do \ and / actually compute the inverse of A. The underlying algorithm is much more sophisticated, and applies to nonsquare and singular systems where the inverse of the coefficient matrix does not exist.

Try

```
>> A = [1 1 -1
2 1 1
1 0 -3];
>> b1 = [0;1;-1];
>> b2 = [0,1,-1];
```

Systems of the form $Ax = b$

```
>> x = A\b1
```

Systems of the form $xA = b$

```
>> x = b2/A
```

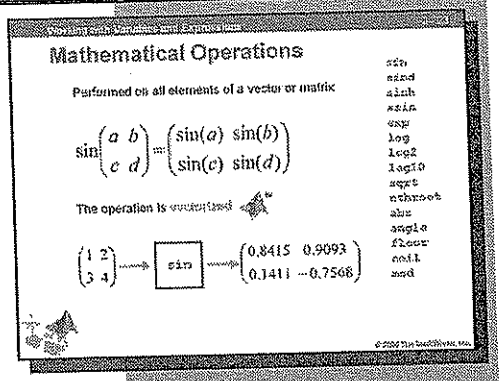
Solving multiple systems

```
>> B = [0 1;1 2;-1 3];
>> x = A\B
```

Comparing `inv(A)*b` , `A\b`

```
>> N = 1:50:1e3;
>> slashspeed(N)
```

Mathematical Operations



One of the principal advantages of working in the M language of MATLAB is that commands are *vectorized*, or able to perform mathematical operations on an entire vector or array of values in a single command. Many mathematical operations in MATLAB are vectorized.

For example, to compute the sine of a sequence of values, first create a vector containing the sequence of values, and then pass the entire vector to the `sin` command:

```
>> x = 0:pi/2:2*pi;
>> y = sin(x)
 y =
   0   1.0000   0.0000  -1.0000   0.0000
```

Similarly, to create an array with random integers between 0 and 9, type

```
>> A = 10*rand(10)
>> R = floor(A)
```

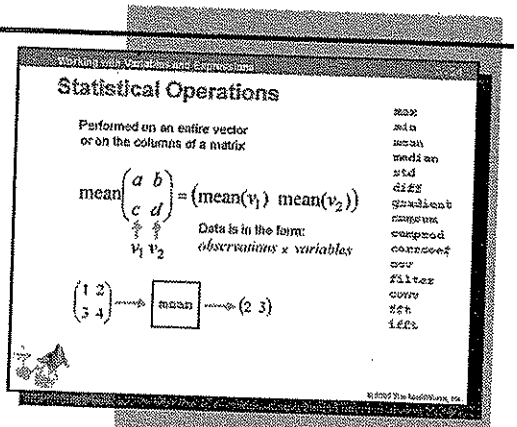
MATLAB performs the 100 round-offs in a single command.

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \longrightarrow \boxed{\texttt{sin}} \longrightarrow \begin{pmatrix} 0.8415 & 0.9093 \\ 0.1411 & -0.7568 \end{pmatrix}$$

Try

```
>> help elfun

>> x = 0:pi/2:2*pi
>> y = sin(x)

>> A = 10*rand(10)
>> R = floor(A)
```

Statistical Operations



Many statistical operations in MATLAB expect the data in the input argument to be organized in the form

observations by *variables*

That is, the rows of the input correspond to distinct observations of each of the variables in the columns.

For example,

```
>> x1 = rand(1,1e6);          % Row vector
>> mu1 = mean(x1)
```

and

```
>> x2 = rand(1e6,1);          % Column vector
>> mu2 = mean(x2)
```

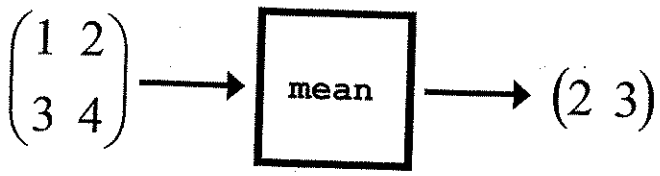
both find the average value of a million random numbers.

However,

```
>> X = rand(1e3);
>> MU = mean(X);
```

finds the means of 1000 variables of 1000 random observations.

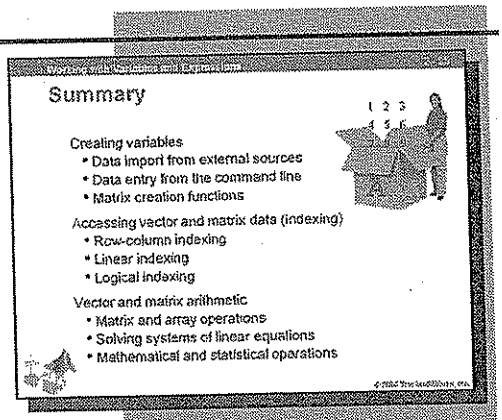
This characteristic of the statistical functions in MATLAB is of great assistance in automating data analysis tasks.



Try

```
>> help datafun

>> x1 = rand(1,1e6);
>> mu1 = mean(x1)
>> x2 = rand(1e6,1);
>> mu2 = mean(x2)
>> X = rand(1e3);
>> MU = mean(X);
>> hist(MU)

>> A = rand(3)
>> std(A)
>> std(A')
>> std(A(:))
```

Summary



Creating variables

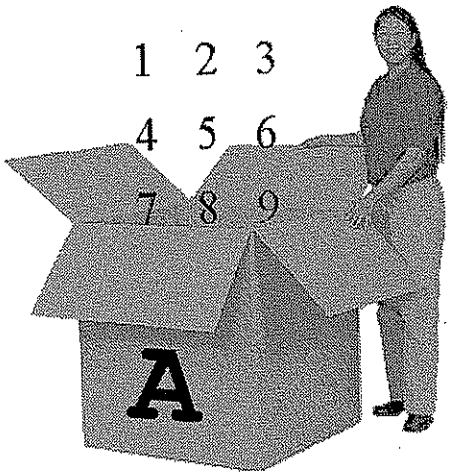
- Data import from external sources
- Data entry from the command line
- Matrix creation functions

Accessing vector and matrix data (indexing)

- Row-column indexing
- Linear indexing
- Logical indexing

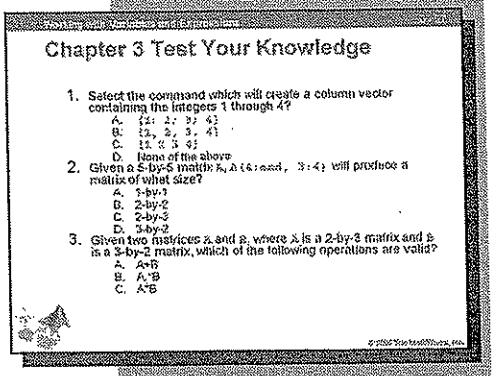
Vector and matrix arithmetic

- Matrix and array operations
- Solving systems of linear equations
- Mathematical and statistical operations



This page intentionally left blank

Chapter 3 Test Your Knowledge



Name:_____

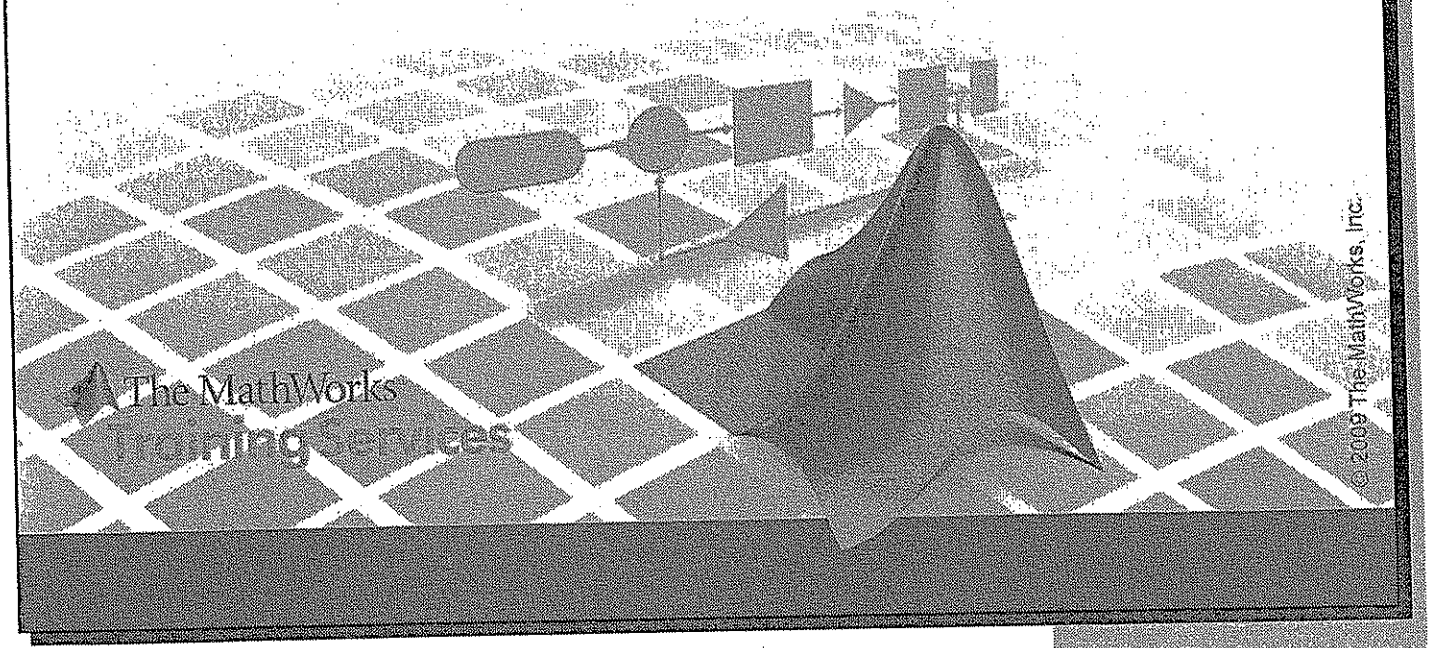
1. Which of the following will create a column vector containing the integers 1 through 4?
   A. `[1; 2; 3; 4]`
   B. `[1, 2, 3, 4]`
   C. `[1 2 3 4]`
   D. None of the above

2. Given a 5-by-5 matrix `A`, `A(4:end,3:4)` will produce a matrix of what size?
   A. 1-by-1
   B. 2-by-2
   C. 2-by-3
   D. 3-by-2

3. Given two matrices `A` and `B`, where `A` is a 2-by-3 matrix and `B` is a 3-by-2 matrix, which of the following operations are valid?
   A. A+B
   B. A.*B
   C. A*B

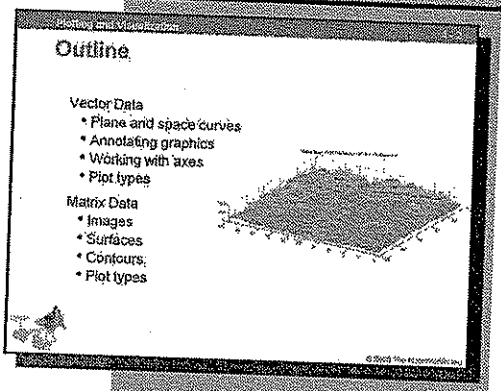
MATLAB® Fundamentals

Plotting and Visualization



The MathWorks Training Services

Outline



Vector Data

- Plane and space curves
- Annotating graphics
- Working with axes
- Plot types

Matrix Data

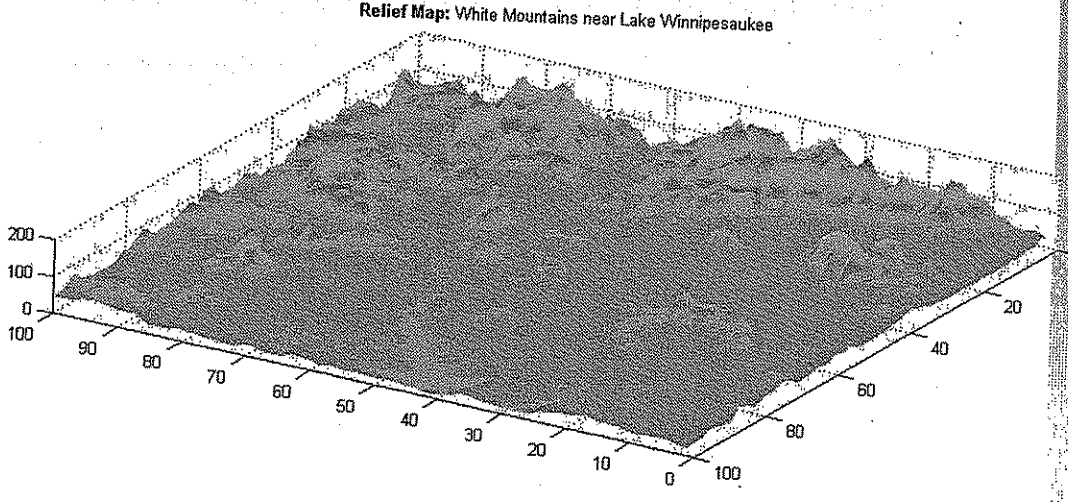
- Images
- Surfaces
- Contours
- Plot types

Try

```
>> graphicsdemo
```

Press any key to see next graph

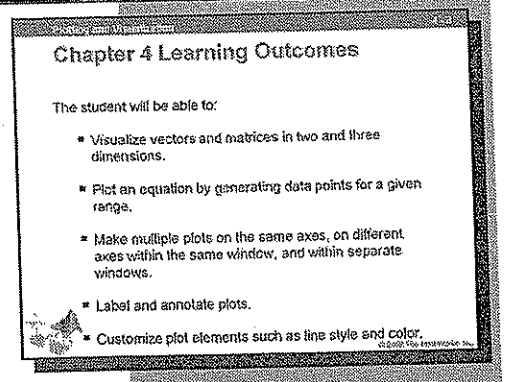
This chapter introduces the visual side of MATLAB® by showing how to create plots of both vector and matrix data. Visualizations complement the numerical capabilities of MATLAB and should play an equal role in any thorough data analysis.



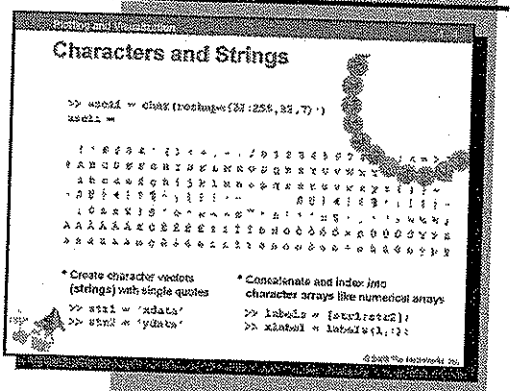
Chapter 4 Learning Outcomes

The student will be able to:

- Visualize vectors and matrices in two and three dimensions.
- Plot an equation by generating data points for a given range.
- Make multiple plots on the same axes, on different axes within the same window, and within separate windows.
- Label and annotate plots.
- Customize plot elements such as line style and color.



Characters and Strings



Strings of characters, delimited by single quotes (`' '`), play a major role in MATLAB plotting. Strings are used to specify property name/property value pairs of graphics objects and also to add labels and annotations.

You concatenate character strings to form rectangular *character arrays* with the same MATLAB concatenation operator (`[ ]`) used for numerical data. You access the character data in character arrays with the same indexing methods used for numerical arrays.

MATLAB has many dedicated functions for working with character data. For example, the `char` command converts numerical values into characters, and the `double` command converts them back. The standard, printable ASCII characters fall in the range `32:127`. Display them in a 3-by-32 array by typing

```
>> char(reshape(32:127,32,3)')
```

Characters in the range `0:31` are nonprinting characters, many of them (such as bells and form feeds) from the days of the teletype machine. Characters in the range `128:255` form an extended ASCII character set, including symbols and international characters. There are several variations of this extended set in use.

Another useful string-handling function in MATLAB is `strvcat`.

```
A = strvcat(s1,s2,s3,...)
```

forms the character array `A` with the strings `s1`, `s2`, `s3`, ... as rows. If the strings are of different lengths, the function appends spaces to the shorter strings to form a valid, rectangular MATLAB array.

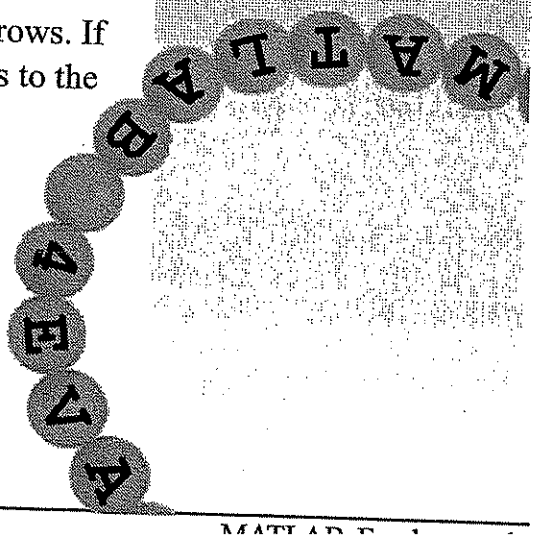
Try

Extended ASCII character set

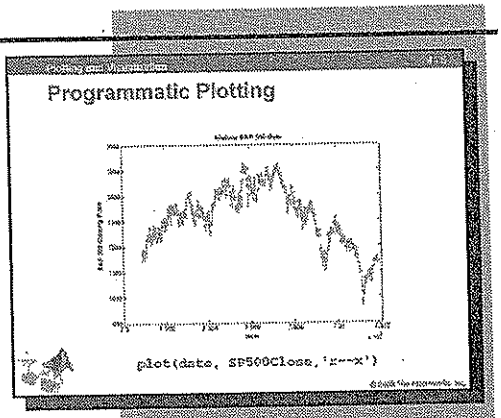
```
>> ascii = ...
char(...
reshape(0:255,32,8)')
```

Creating and indexing into character arrays

```
>> str1 = 'xdata';
>> str2 = 'ydata';
>> axeslabels = ...
[str1; str2];
>> title = 'My Data';
>> figurelabels = ...
strvcat(...
axeslabels,title);
>> xaxislabel = ...
figurelabels(1,:);
>> yaxislabel = ...
figurelabels(2,:);
```



Programmatic Plotting



So far we have used the MATLAB environment to create plots interactively. Now we will see how to perform the same tasks using MATLAB commands. Our example is S&P500 data.

1. Load and process data to plot.

```
>> load SP500;
>> date = SP500Data(:,1);
>> SP500Close = SP500Data(:,2);
```

2. Calculate volatility.

```
>> returns = log(SP500Close(2:end)./...
SP500Close(1:end-1));
>> percentVolatility = ...
std(returns)*sqrt(252)*100;
```

3. Plot a date vs. SP500Close.

```
>> plot(date, SP500Close);
```

4. Plot SP500 data with a red dashed line, using x for markers.

```
>> plot(date, SP500Close,'linestyle','--'...
,'marker','x','color','r');
```

or

```
>> plot(date, SP500Close,'r--x');
```

We have varied various properties to give the plot the look we desire. Issuing `plot` commands is necessary when writing M-files as we'll see later.

5. Add labels and title to the axes.

```
>> xlabel('date','fontweight','bold');
>> ylabel('S&P 500 Closing
Price','fontweight','bold');
>> title('Historic S&P 500
Data','fontweight','bold');
```

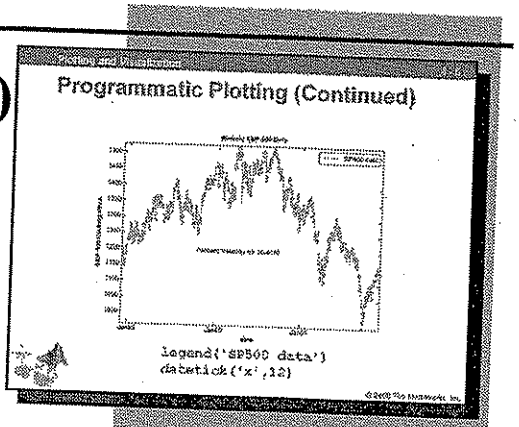
Try

```
>> doc plot
```

Resulting figure

```
>> open(...
'SP500_1.fig');
```

Programmatic Plotting (Continued)



6. Add a legend.

```
>> legend('SP500 data');
```

The annotation commands accept ASCII character strings as annotations. The string interpreter in MATLAB will also accept, and parse, strings in the TEX technical typesetting language.

You do not have to know much TEX to add technical annotations to your MATLAB plots. All TEX characters begin with a backslash (\). For example, `\lambda` is interpreted and displayed as λ, `\Lambda` as Λ, and `\int` as ∫. A list of characters appears in the documentation.

7. Add text indicating the volatility.

```
>> part1=texlabel('Percent Volatility sigma
= ');
>> part2=num2str(percentVolatility);
>> gtext([part1,part2]);
```

or

```
>> gtext(['Percent Volatility \sigma = ...
',num2str(percentVolatility)]);
```

8. Turn serial date number into mm/yy format.

```
>> datetick('x',12);
```

9. Tighten up the range covered by the axis.

```
>> axis tight;
```

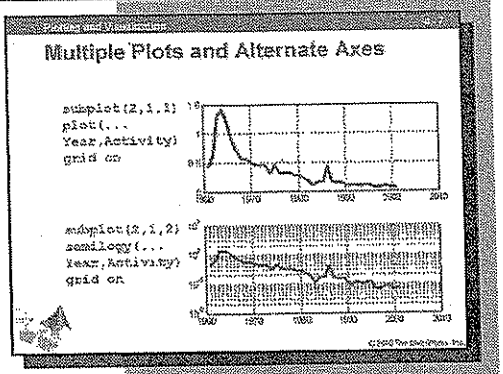
Try

```
>> doc annotation
>> doc datetick
```

Resulting figure

```
>> open(...
'SP500_2.fig');
```

Multiple Plots and Alternate Axes



There are many occasions where you will want to view multiple plots simultaneously. There are three basic ways to do this:

- Open a new figure window with `figure`.
- Plot repeatedly to the same set of axes using `hold`.
  (In Plot Tools, select the axes in the Plot Browser and use **Add Data**.)
- Create multiple axes in the same figure with `subplot`.
  (In Plot Tools, select **New Subplots** in the Figure Palette.)

For comparing data on very different scales, the `plotyy` command creates separate *y*-scales to the left and the right of an axes box, then scales and color-codes different plots to one specified axis or the other.

The `semilogx`, `semilogy`, and `loglog` commands allow you to scale the *x* and *y* axes to produce semilog or log-log plots.

By default, MATLAB will scale the axes of a plot to fit the data. To control the scaling and appearance of axes, use the `axis` command.

```
>> axis([xmin xmax ymin ymax])
```

sets the minimum and maximum values of the *x* and *y* axes.

```
>> axis equal
```

sets the aspect ratio so that the units are the same on all axes.

```
>> axis square
```

makes the axes box square (or a cube when three-dimensional).

Try

Multiple plots
```
>> edit multiplot
>> multiplot
```

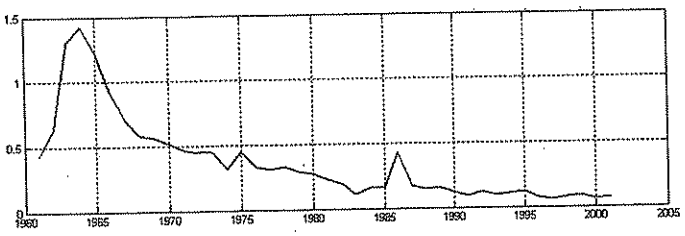
Data on different scales
```
>> edit yyplot
>> yyplot
```

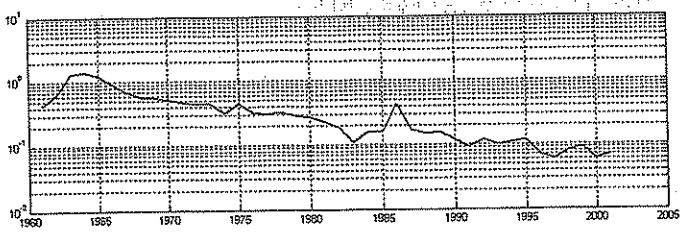
Log scale
```
>> edit milklogplot
>> milklogplot
```

Exponential vs. power
```
>> edit reexp
>> reexp
```

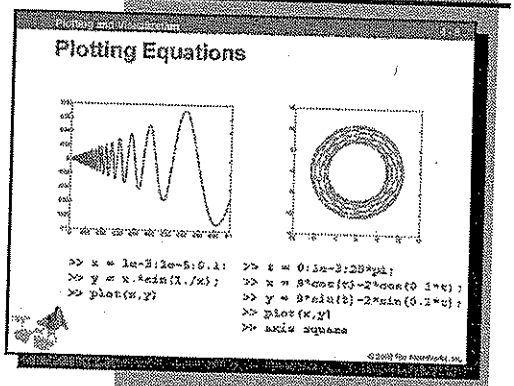
Linked axes
```
>> edit plotlink
>> plotlink
```

Piecewise function
```
>> edit pieceplot
>> pieceplot
```





Plotting Equations



The plotting methods discussed so far in this chapter have been concerned primarily with visualizing data. You may also want to visualize equations that can be used to model your data.

The method in MATLAB is to convert your equations to data. For example, suppose you want to visualize $y = x\sin(1/x)$ on the interval (0, 0.1]. You first create a vector of `x` values spanning the interval, with a mesh size appropriate for the resolution you require:

```
>> x = 1e-3:1e-5:0.1;
```

You then use your equation, and the ability of MATLAB to perform vectorized operations, to compute the corresponding `y` values:

```
>> y = x.*sin(1./x);
```

The problem has now been reduced to plotting the data in the vector `x` against the data in the vector of equal length, `y`.

You can handle plots of parametric equations similarly. To visualize $x = x(t), y = y(t)$, first create a vector `t` for the range of the independent variable, then use the equations to compute vectors for `x` and `y`. You have the option of plotting `x` versus `t`, `y` versus `t`, or `y` versus `x`.

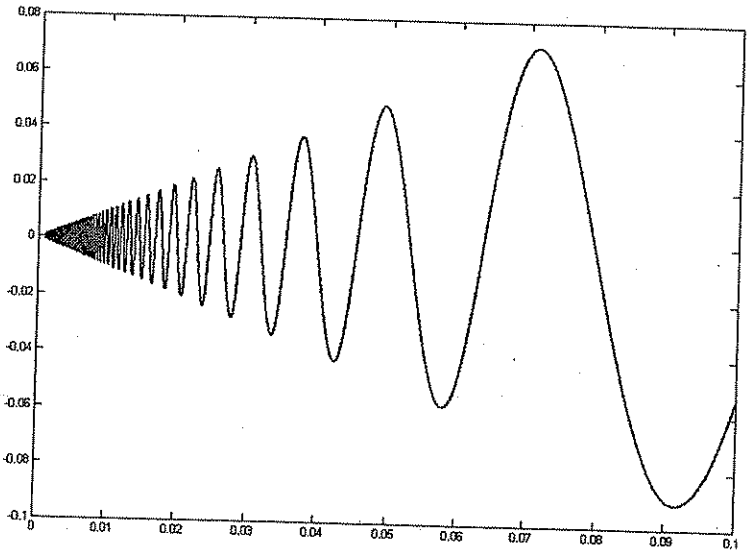
Try

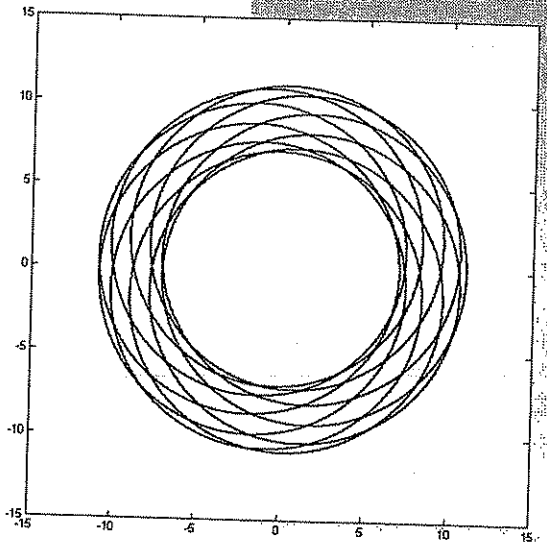
Equations of the form $y = f(x)$

```
>> x = 1e-3:1e-5:0.1;
>> y = x.*sin(1./x);
>> plot(x,y)
```

Parametric equations

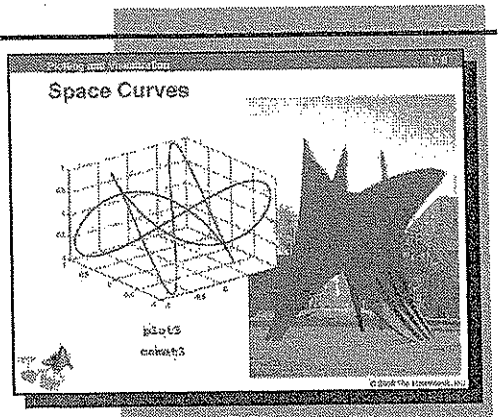
```
>> t = 0:1e-3:20*pi;
>> x = 9*cos(t) - ...
2*cos(0.1*t);
>> y = 9*sin(t) - ...
2*sin(0.1*t);
>> plot(x,y)
>> axis square
```



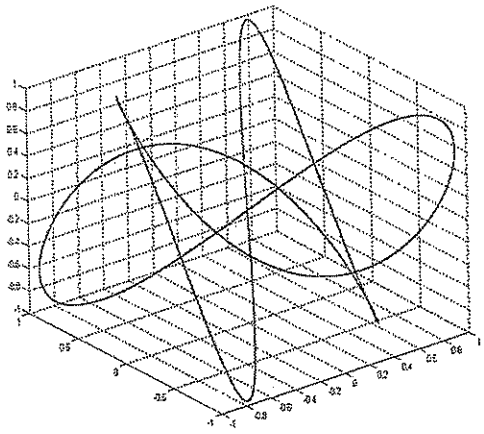


Space Curves

Vector data can be embedded in higher dimensions, where the continuous plot of an equation is usually called a *space curve*. Plotting methods are the same as for equations in 2-D, but the plotting commands are 3-D implementations, such as `plot3` and `comet3`, of 2-D commands like `plot` and `comet`.




```
t = linspace(...
    0,2*pi,1e4);
x = sin(2*pi*2*t);
y = sin(2*pi*3*t);
z = sin(2*pi*4*t);
plot3(x,y,z)
```



Try

Hawk soaring on a thermal

```
>> t = 0:0.01:100;
>> x = 10*t.*cos(t);
>> y = 10*t.*sin(t);
>> z = t;
>> plot3(x,y,z)
>> axis square
>> comet3(x,y,z)
```

Lissajous curve

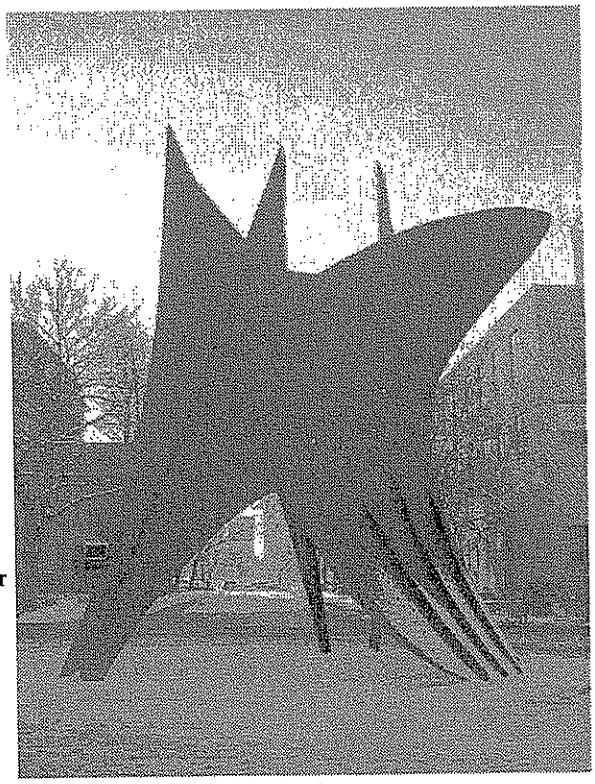
```
>> edit lissajous3
>> [x,y,z] = ...
lissajous3(2,3,4,...
2*pi);
>> grid on
>> slowcomet3(...
x,y,z,1e2)
```

Space-filling curve

```
>> edit hilbertcube
>> [I,J,K,H] = ...
hilbertcube(4);
>> slowcomet3(...
I,J,K,1e2)
```

Space curve with vector field

```
>> open('HL20.fig')
>> edit flightpath
>> flightpath
```

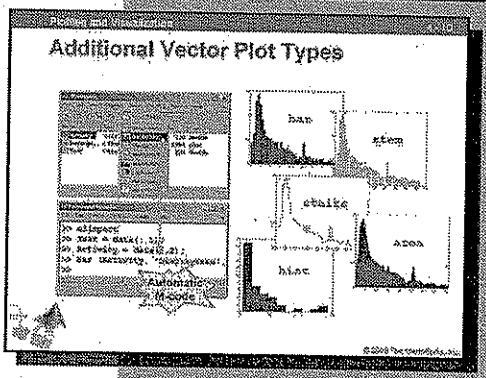


La Grande Voile
(The Big Sail)
Alexander Calder

Massachusetts
Institute of
Technology

Cambridge, MA

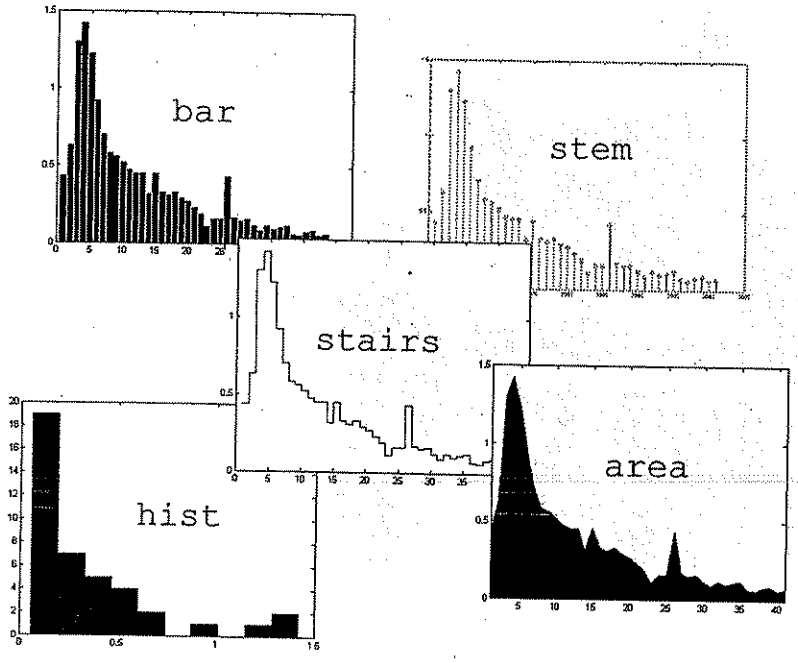
Additional Vector Plot Types



MATLAB has many specialized plotting commands to allow you to visualize and present your data with exactly the right emphasis on the data features you find important. Some of the additional plot types for vector data include

`bar`	Bar graph (vertical and horizontal)
`stem`	Discrete sequence (signal) plot
`stairs`	Stairstep graph
`area`	Filled area plot
`pie`	Pie chart
`hist`	Histogram

Create plots by using these commands from the command line, from within M-files, or interactively by choosing the plot type from the Plot Selection menu ▾ at the top of the Workspace Browser. When the menu is used, MATLAB automatically generates the corresponding command in M-code and displays it in the Command Window.



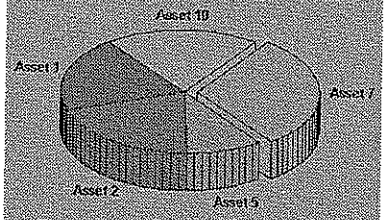
Try

Additional plot types

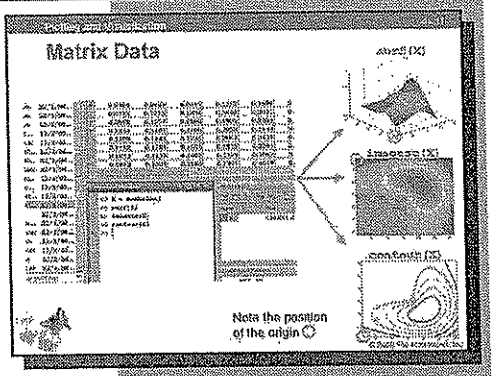
```
>> edit milkplots
>> milkplots
```

Pie charts

```
>> edit portfolio
>> portfolio
```



Matrix Data



Matrix data is organized along more than one linear dimension. It arises when multiple parameters determine the location of a measurement or observation.

Altitudes measured at different latitudes and longitudes will produce 2-D matrix data. Temperatures measured at different points within a test chamber will produce 3-D matrix data. Readings from an instrument with five dials will produce 5-D matrix data.

Matrix data comes with its own visualization challenges, and its own set of MATLAB plotting commands. These commands are vectorized in MATLAB, so that an entire matrix data set can be processed at once.

It is important to distinguish matrix data from mere collections of vector data organized into the columns of a matrix. Measurements of insect characteristics observed over a sample may be organized into a matrix, in the form of insect x characteristic, but the data would not be regarded, or visualized, as matrix data. It would be seen as different vector variables (characteristics) organized along a single dimension (insect). This organization is what MATLAB expects for vector data in statistical analysis. The statistical functions in MATLAB, rather than being vectorized across an entire input matrix, are vectorized only along the columns, which the functions treat independently.

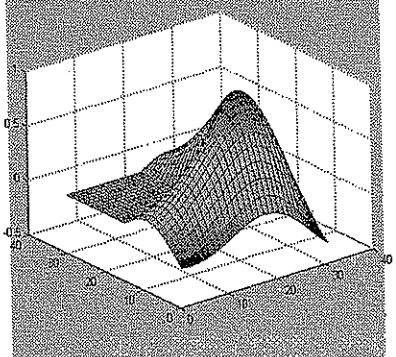
2-D matrix data occurs frequently in applications. Three basic plotting commands for 2-D matrix data are `surf`, `imagesc`, and `contour`. Each command presents a different perspective on the data.

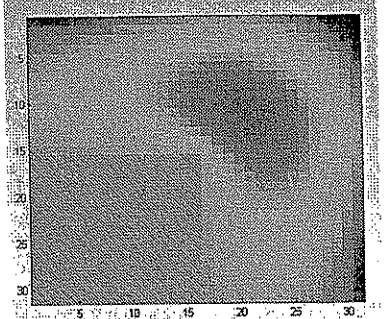
Try

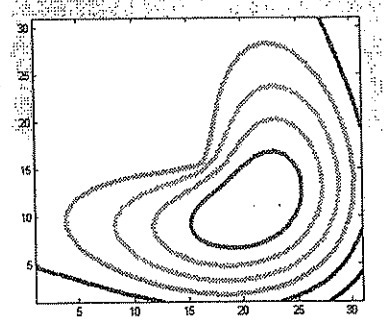
Basic matrix data plots

```
>> Z = membrane;
>> surf(Z)
>> imagesc(Z)
>> contour(Z)
```

Note the changing position of the origin in the data.



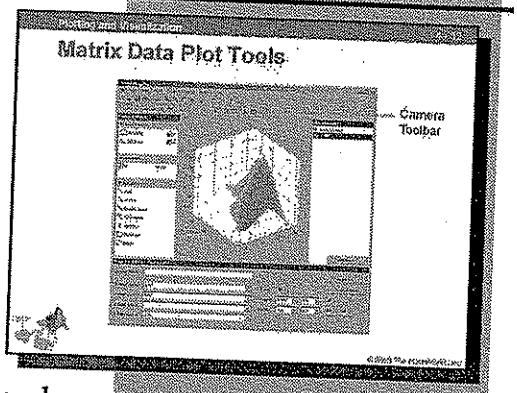




Array Editor

	1	2	3	4	5
1	-0.16692	-0.17288	-0.17476	-0.17231	-0.16543
2	-0.12236	-0.12171	-0.11674	-0.10725	-0.093198
3	-0.076557	-0.069667	-0.058256	-0.042178	-0.021437
4	-0.031716	-0.019156	-0.0019392	0.020042	0.046747
5	0.010073	0.027526	0.049705	0.076691	0.10842
6	0.046925	0.068317	0.094433	0.12533	0.16094
7	0.077246	0.10148	0.13036	0.16392	0.20211
8	0.099804	0.12569	0.15604	0.19091	0.23024
9	0.11379	0.14008	0.17056	0.2053	0.24426

Matrix Data Plot Tools



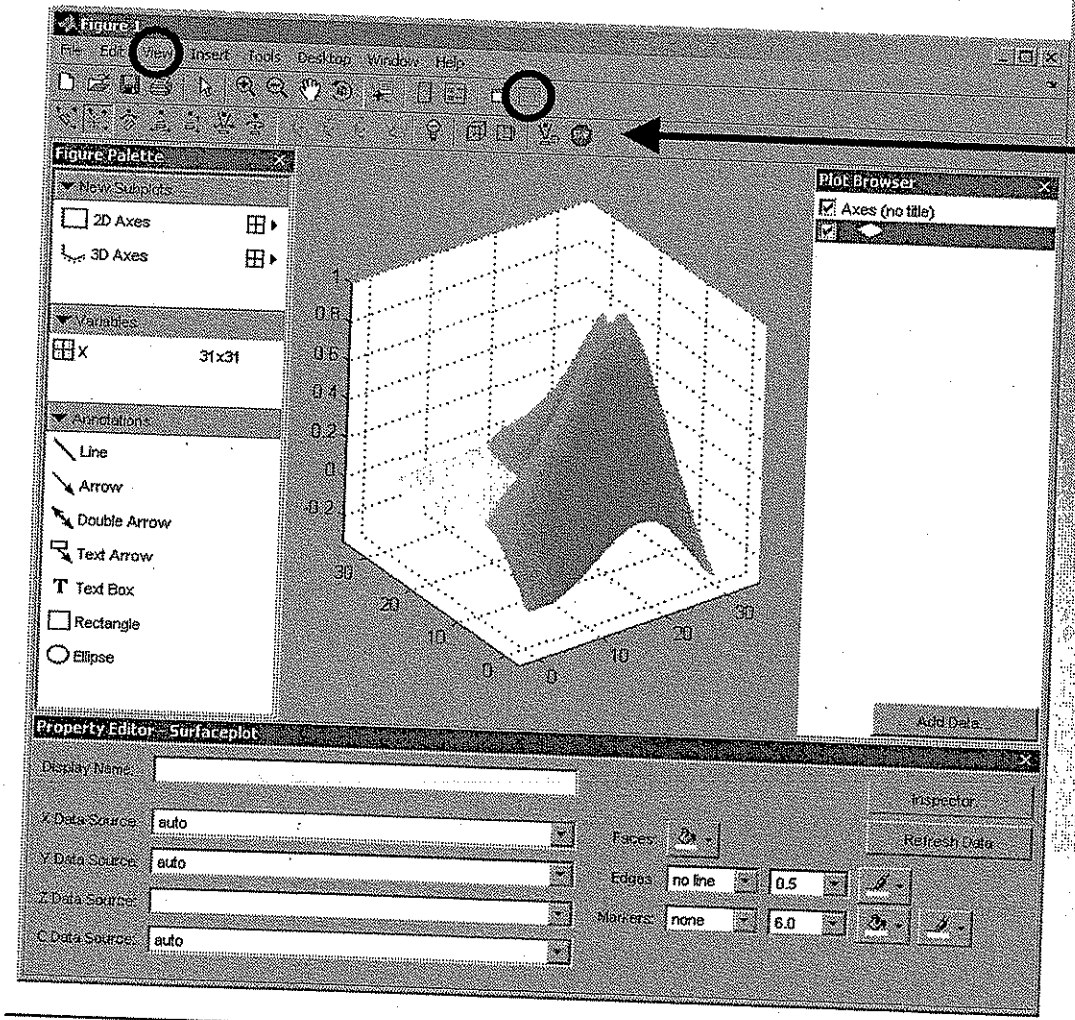
You access plot tools for matrix data plots in the same way as for vector data plots. Click the Show Plot Tools button at the top of the figure window to open up the Figure Palette, the Plot Browser, and the Property Editor.

For 3-D visualizations of 2-D matrix data, an additional toolbar can be added to the top of the figure that allows you to interactively manipulate the MATLAB "camera" that views the data. Choose **View → Camera Toolbar** from the menus at the top of the figure. The Camera Toolbar controls point of view, lighting, zooming, etc. Its features are implementations of OpenGL graphics routines, and the use of OpenGL hardware will improve rendering performance.

Try

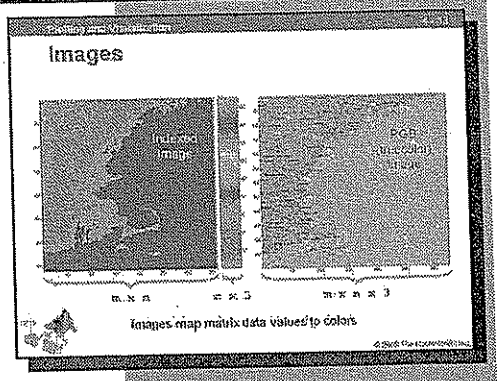
```
>> Z = membrane;
>> surf(Z)
```

Add the Camera Toolbar to the figure and experiment with its features.



Camera Toolbar

Images



Images map matrix data values to colors. Similar values in the data are mapped to similar colors, so that patterns in the data become apparent. Two common image formats in MATLAB are *indexed images* and *RGB* (or *truecolor*) *images*.

Indexed images consist of two arrays in MATLAB. The first is the `m`-by-`n` matrix of data itself. The second is a `c`-by-3 *colormap* that determines how values in the data will be mapped to particular colors. The number of rows, `c`, in the colormap determines the number of colors used in the plot. All colormaps have three columns. Each row in a colormap has the form `[r g b]`, where $0 \le r,g,b \le 1$. These three values give the relative intensity (0 for none, 1 for fully saturated) of the red, green, and blue components, respectively, of a single color in the colormap. To display the data matrix `A` using colormap `map`, use

```
>> image(A)
>> colormap(map)
```

Low values in the data are assigned to low row numbers in the colormap; high values in the data are assigned to high row numbers. If a colormap is not specified, MATLAB applies the default map `jet`.

The command

```
>> imagesc(A)
```

scales the data in `A` to the full range of the colormap. Scaling is useful when the matrix data varies within only a small range of values.

RGB images consist of a single, *m*-by-*n*-by-3 array in MATLAB. There is no colormap. Rather, the three planes of the array give the red, green, and blue intensities, respectively, of each of the *m*-by-*n* image elements. In a sense, the colormap is built into the array. However, the preimage of the colormap – the data itself – is gone. As a result, RGB images are used primarily for photographic images, not data visualization. The `image` command displays RGB images using the colors specified.

Try

Indexed image

```
>> A = magic(25);
>> map = ...
[1 0 0;0 1 0;0 0 1];
>> image(A)
>> colorbar
>> colormap(map)
>> colorbar
>> imagesc(A)
>> colorbar
>> colormap(map)
>> colorbar
```

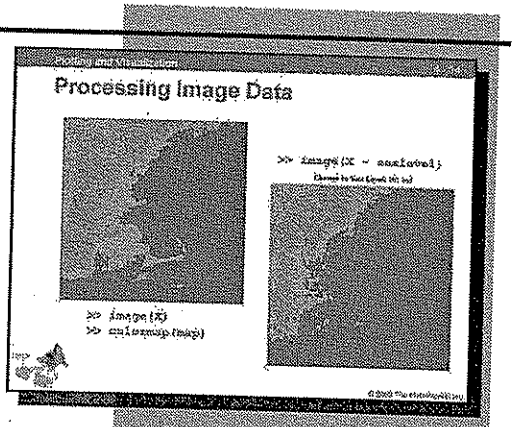
RGB image

```
>> load eli
>> image(E)
```

Satellite elevation data

```
>> edit mapcape
>> mapcape
```

Processing Image Data

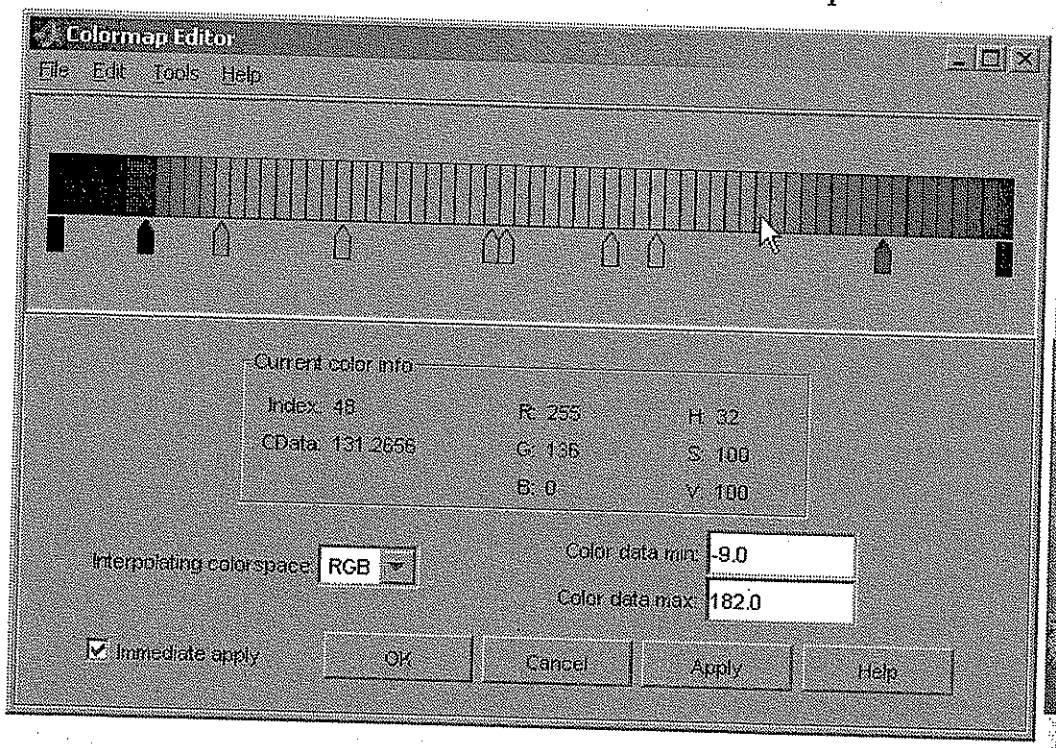


In MATLAB, *image processing* reduces to two operations: altering the data in an image, or altering the colors used to display the data. For an indexed image, MATLAB performs computations with either the data array or the colormap array. These computations are intended to produce a new image that enhances specific features in the data.

To alter the colormap interactively, type

```
>> colormapeditor
```

or choose **Edit → Colormap** from the menus at the top of a figure. The Colormap Editor opens and displays the current colormap.



Move your mouse along the colorbar to see information about each row (index) in the colormap. The nodes ⌂ below the colorbar indicate places where the rate of change of R, G, or B changes. You can move nodes with your mouse, add new nodes by clicking below the colorbar, or delete a node by clicking on it and pressing the **Delete** key. Double-clicking on a node allows you to choose a new color for that index.

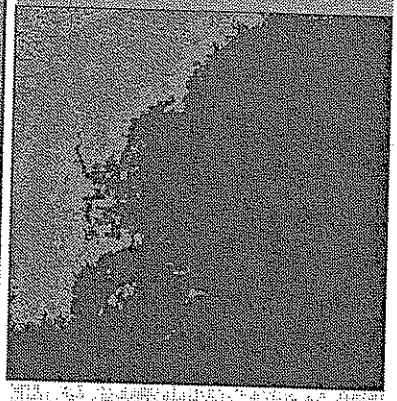
Try

Altering the colormap

```
>> load cape
>> image(X)
>> colormap(map)
>> colormapeditor
```

Altering the data

```
>> load cape
>> image(X)
>> colormap(map)
>> image(X-2)
>> image(X-10)
>> edit risingsea
>> risingsea
```



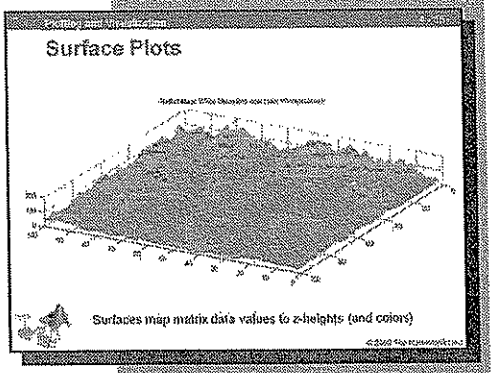
Logical indexing

```
>> edit lowcape
>> lowcape
```

Edge detection

```
>> edit edgedet
>> edgedet
```

Surface Plots



Surfaces map matrix data values to *z-heights* (and colors). This form of visualization is associated with equations of the form $z = f(x, y)$ and parametric equations of the form $x = x(t)$, $y = y(t)$, and $z = z(t)$. It is generally useful, however, whenever you want to highlight *differences* in 2-D matrix data.

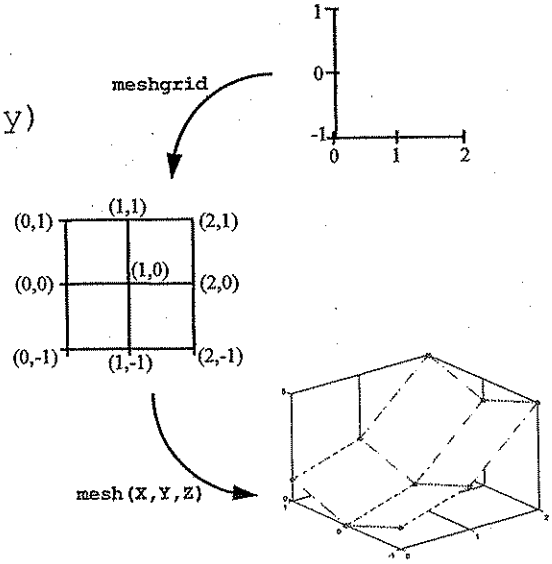
The basic surface plotting command in MATLAB is `surf`. If `A` is a 2-D matrix of data, then

```
>> surf(A)
```

displays the data in 3-D axes with the `z` height at any point (`x`,`y`) given by the value in the data at the `x`th row and `y`th column of `A`. MATLAB also colors the plot using the default colormap, just as in an image display of the data. Using the `colormap` command you can apply your own colormaps to highlight distinct levels in the data.

When plotting equations of the form $z = f(x, y)$, it is common to start with a vector of values `x` and a vector of values `y` that specify the extent of the plot. The MATLAB command `meshgrid` converts these vectors into 2-D arrays `X` and `Y` which can be passed to the equation to compute a 2-D array `Z` of values for `surf`. For example,

```
>> x = 0:2;
>> y = -1:1;
>> [X,Y] = meshgrid(x,y)
  X =
     0     1     2
     0     1     2
     0     1     2
  Y =
    -1    -1    -1
     0     0     0
     1     1     1
>> Z = X.^2 + Y.^2;
```



`surf(Z)` displays the data in `Z` vs. the row and column numbers.
`surf(X,Y,Z)` displays the data in `Z` vs. the coordinates in `x` and `y`.

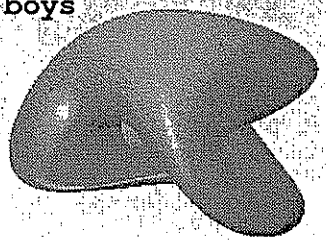
Try

Basic surface plot
```
>> x = 0:2;
>> y = -1:1;
>> [X,Y] = ...
meshgrid(x,y);
>> Z = X.^2 + Y.^2;
>> surf(X,Y,Z)
```
Repeat the above using
```
>> x = -1:0.1:1;
>> y = x;
```

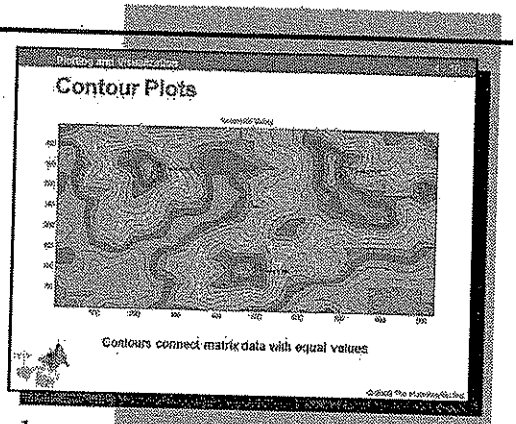
Satellite elevation data
```
>> edit surfcape
>> surfcape
```

Surface defined by an equation
```
>> edit folium
>> folium
```

Parametric surfaces
```
>> edit mobius
>> mobius
>> edit boys
>> boys
```



Contour Plots

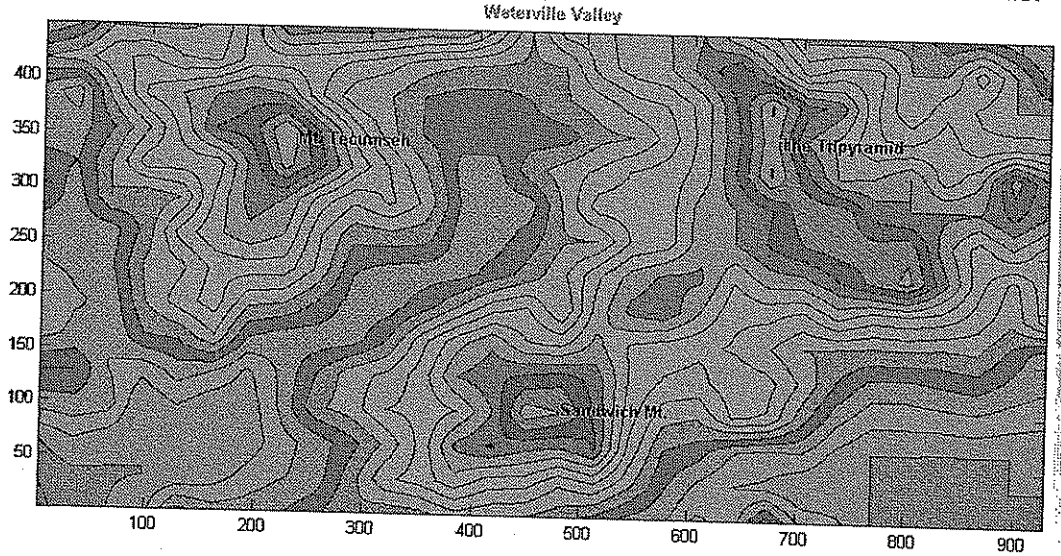


Contours connect matrix data with equal values. Contour plots are useful for visualizing rates of change in a data set – close contours indicate high rates of change, while widely spaced contours indicate low rates of change. Contour plots also have the advantage of reducing the dimension of a data visualization.

MATLAB provides a number of functions for producing contour plots:

`contour`	Basic contour plot of a matrix
`contourf`	Filled contour plot
`contour3`	3-D contour plot (contours at their *z*-heights)
`surfc`	Surface plot with the contours below

The syntax for each of these functions is similar to that for the `surf` command, with the basic input a 2-D matrix of data. The function `contourc` is a low-level function that returns the data in the contours.

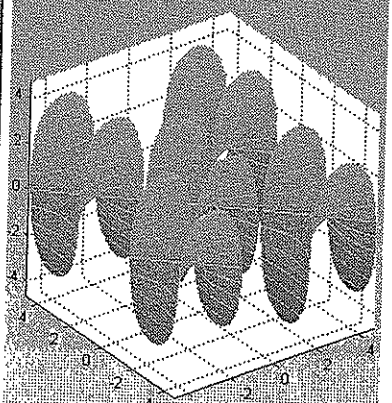


Try

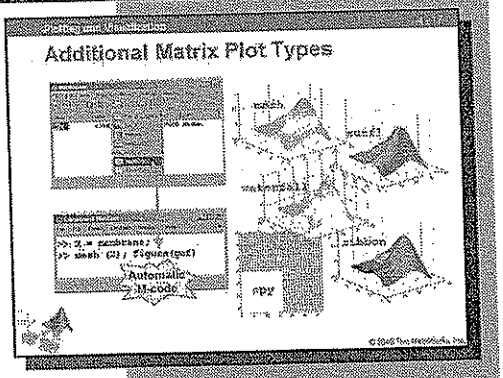
Contours in data
```
>> edit contourcape
>> contourcape
```

Contours of a surface
```
>> edit foliumc
>> foliumc
```

3-D contours
```
>> edit scherk
>> scherk
```



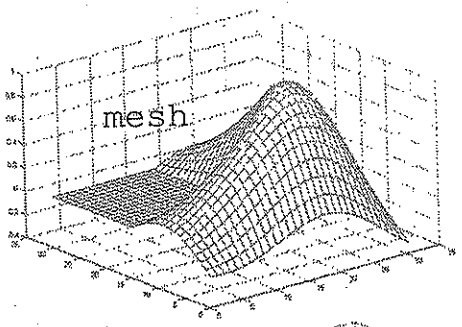
Additional Matrix Plot Types

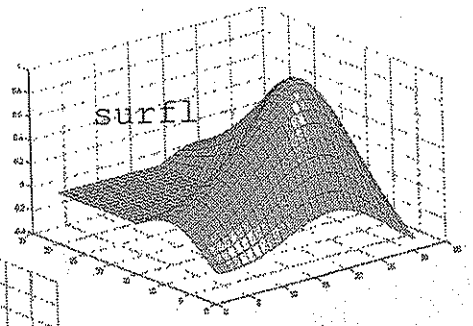


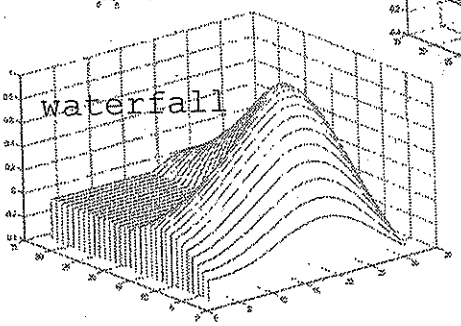
Some of the additional plot types for matrix data include

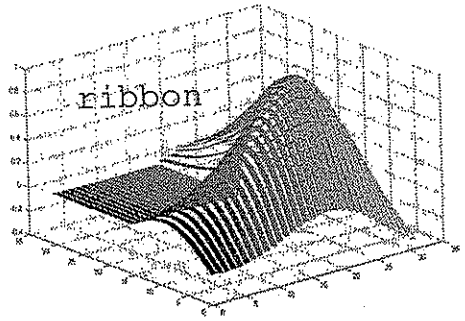
`mesh`	Wireframe mesh surface plot
`waterfall`	Mesh plot without column lines
`ribbon`	Columns as separate 3-D ribbons
`surfl`	Surface plot with lighting effects
`spy`	Zero/nonzero values (sparsity pattern)

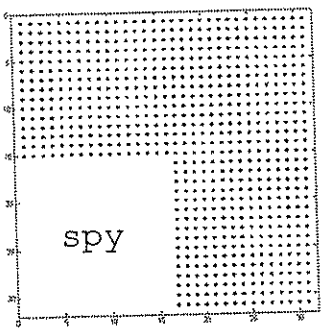
Plots are created by using these commands from the command line, from within M-files, or interactively by choosing the plot type from the Plot Selection menu at the top of the Workspace browser. When the menu is used, MATLAB automatically generates the corresponding command in M-code and displays it in the Command Window.











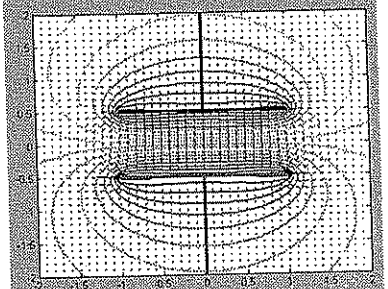
Try

Additional plot types

```
>> edit ...
membraneplots
>> membraneplots
```

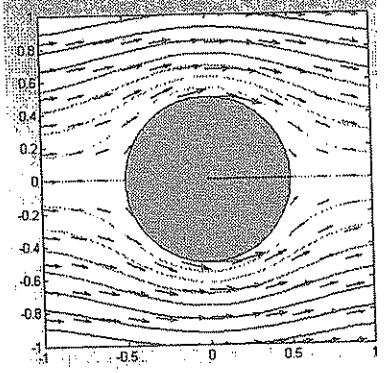
Electric field of a capacitor

```
>> edit capacitor
>> capacitor
```

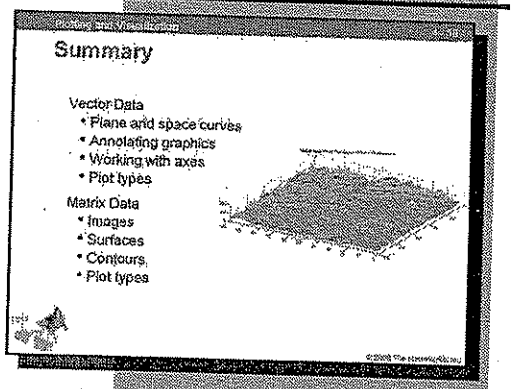


Fluid flow around a cylinder

```
>> edit fluidflow
>> fluidflow
```



Summary

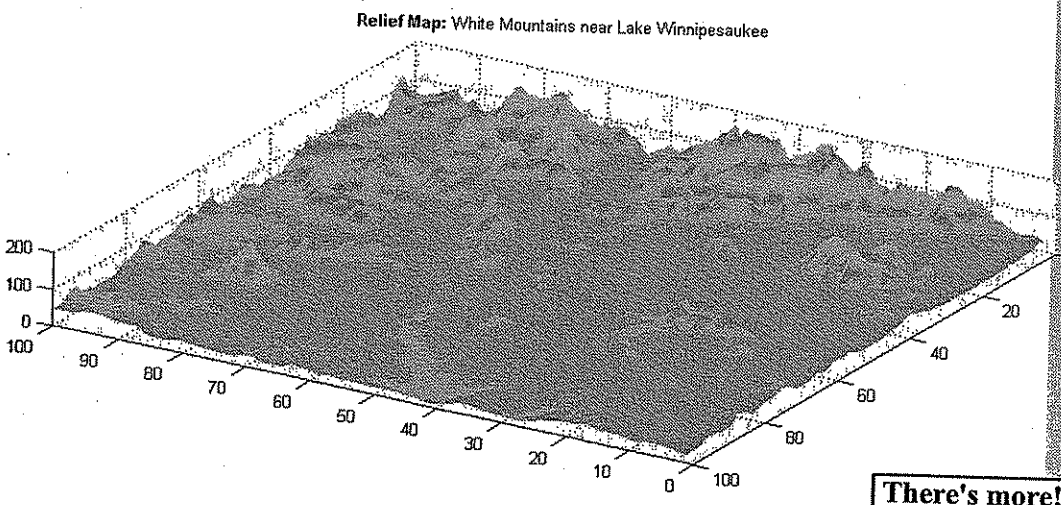


Vector Data

- Plane and space curves
- Annotating graphics
- Working with axes
- Plot types

Matrix Data

- Images
- Surfaces
- Contours
- Plot types



There's more!

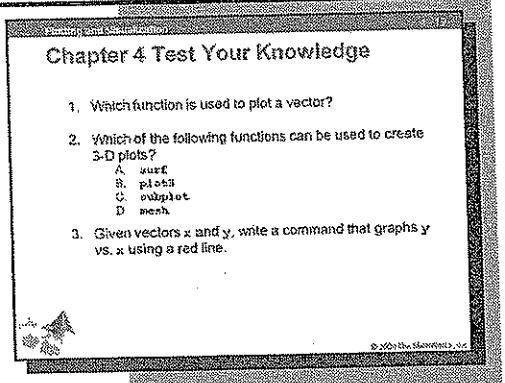
MATLAB also provides tools for visualizing volume data.

See Appendix D-2

MATLAB plots can be customized to produce publication-quality graphics. You can also create and export animations.

Course Offering: ***MATLAB for Data Processing and Visualization***

Chapter 4 Test Your Knowledge



Name:_____

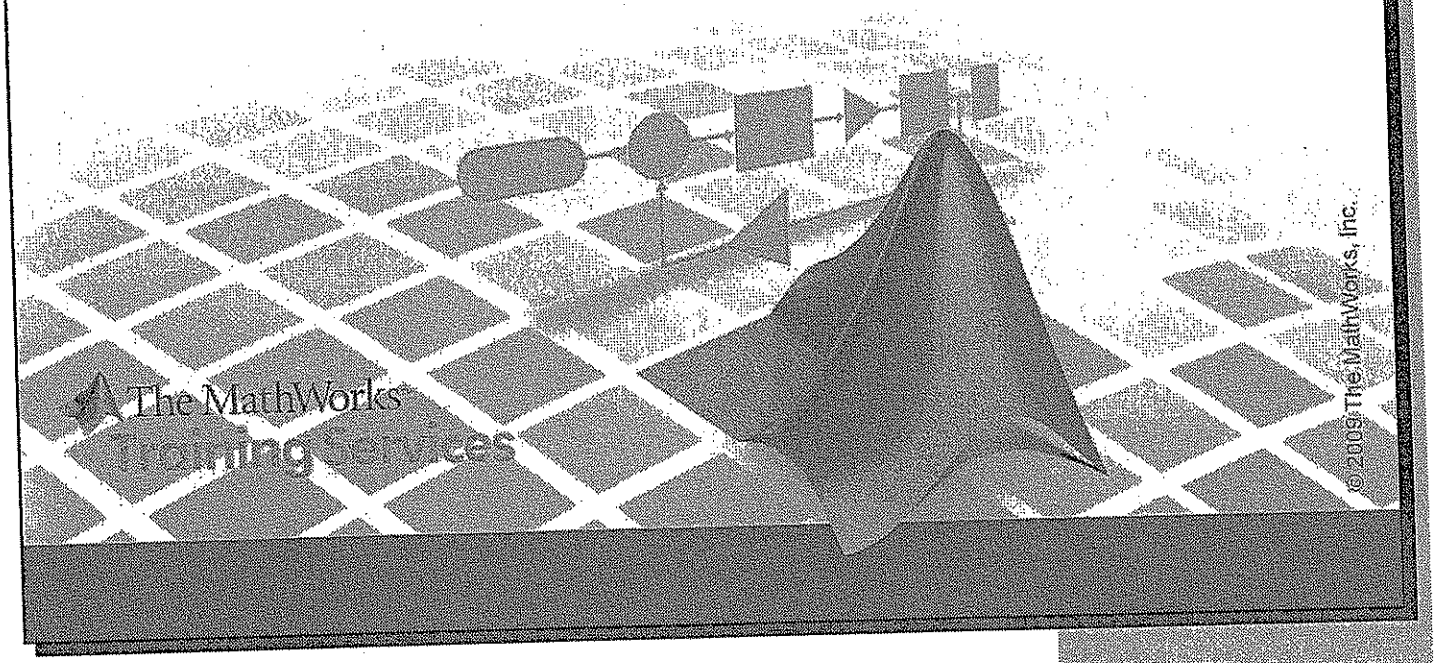
1. Which function is used to plot a vector?

2. Which of the following functions can be used to create 3-D plots?
   A. `surf`
   B. `plot3`
   C. `subplot`
   D. `mesh`

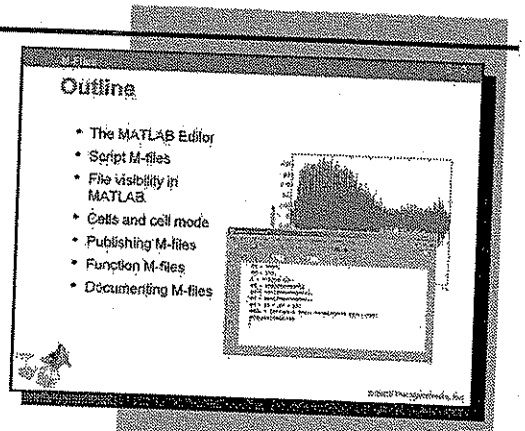
3. Given vectors `x` and `y`, write a command that graphs `y` vs. `x` using a red line.

MATLAB® Fundamentals

M-Files

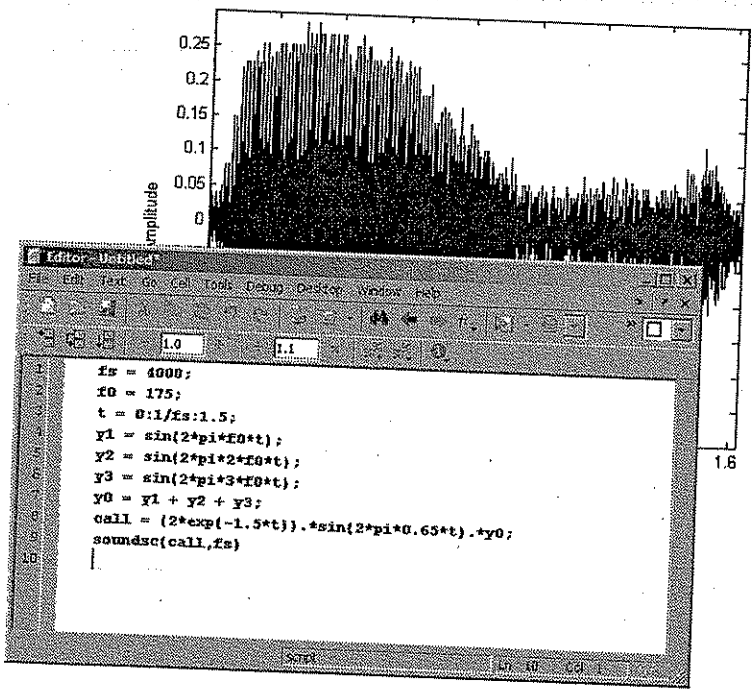


Outline



- The MATLAB® Editor
- Script M-files
- File visibility in MATLAB
- Cells and cell mode
- Publishing M-files
- Function M-files
- Documenting M-files

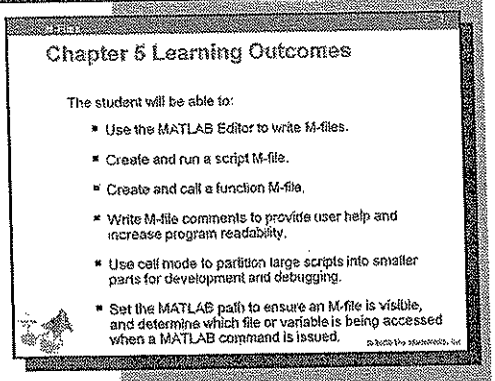
M-files are the setting for MATLAB programming. This chapter gives an overview of how to write, edit, run, document, and publish M-files. The distinction between script and function M-files is highlighted.



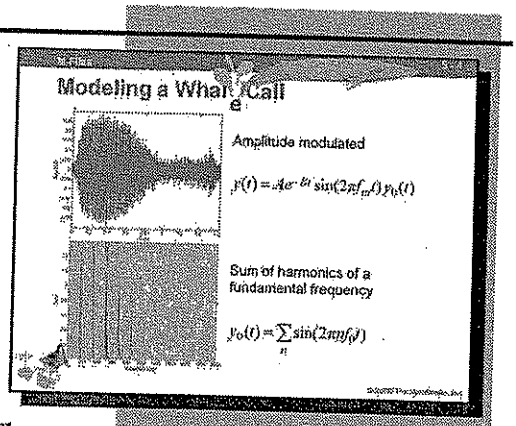
Chapter 5 Learning Outcomes

The student will be able to:

- Use the MATLAB Editor to write M-files.
- Create and run a script M-file.
- Create and call a function M-file.
- Write M-file comments to provide user help and increase program readability.
- Use cell mode to partition large scripts into smaller parts for development and debugging.
- Set the MATLAB path to ensure an M-file is visible, and determine which file or variable is being accessed when a MATLAB command is issued.



Modeling a Whale Call



The subject of the M-file programming in this chapter is a simple model of the call of a blue whale.

Befitting the largest mammal on earth, the call of an adult blue whale is loud and low. The sound carries farther than any other animal sound, and can be detected over a thousand miles away. The call has a distinct pattern. An A "trill" usually precedes a series of B "moans." There is little variation in the call among individual whales.

The B call is simpler and easier to analyze. It consists of a fundamental frequency around 16-17 hertz and a series of harmonics (multiples) of the fundamental frequency. The amplitude of the call is modulated to produce a load moan followed by a weaker moan.

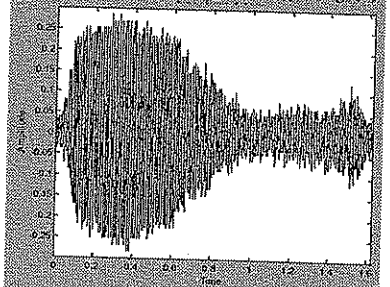
The call model in this chapter is the result of several modeling choices.

- The envelope of the signal is treated as a decaying sine wave of the form $Ae^{-Bt}\sin(2\pi f_m t)$, where the amplitude A, the decay rate B, and the modulating frequency f_m are all adjustable parameters.
- The signal itself is treated as a sum of harmonics $\sin(2\pi n f_0 t)$, for $n = 0, 1, 2, 3, \ldots$. The harmonics are assumed to be present for the entire duration of the call, even though the higher harmonics appear and disappear in the middle of the call. (Simply chopping off the higher harmonics introduces unwanted frequencies.)
- Because blue whale calls are so low, they are barely audible to humans. The time scale in the data being modeled is compressed by a factor of 10 to raise the pitch and make it more clearly audible. The model works with the same "x10" audible frequencies.

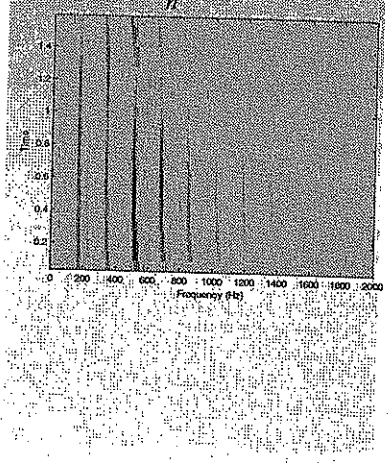
Try

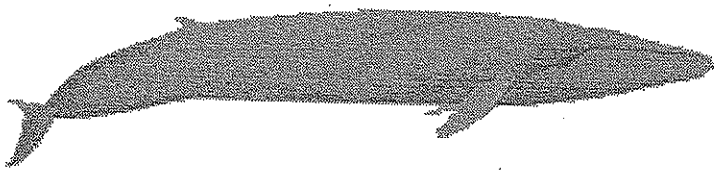
```
>> edit whalecall
>> whalecall
```

$$y(t) = Ae^{-Bt}\sin(2\pi f_m t)y_0(t)$$

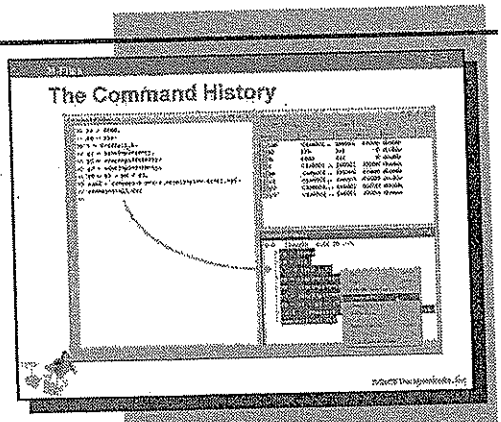


$$y_0(t) = \sum_n \sin(2\pi n f_0 t)$$





The Command History



MATLAB programming can be carried out in the Command Window by typing commands one after another at the prompt. Typing commands works well for computations and displays that can be expressed in a few commands, but it becomes tedious for longer sequences of commands, especially if they must be repeated.

For example, to implement the model of a whale call, you might type

```
>> fs = 4000;
>> f0 = 175;
>> t = 0:1/fs:1.5;
>> y1 = sin(2*pi*f0*t);
>> y2 = sin(2*pi*2*f0*t);
>> y3 = sin(2*pi*3*f0*t);
>> y0 = y1 + y2 + y3;
>> call = (2*exp(-1.5*t)).*sin(2*pi*0.65*t).*y0;
>> soundsc(call,fs)
```

It is difficult to work with the model (change frequencies, etc.) using this mode of entry. At best, the sequence of commands must be copied and pasted, and then altered, with every iteration of the model.

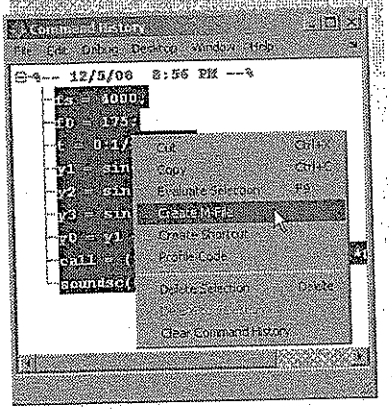
Fortunately, MATLAB keeps a history of the commands you type, and displays them in the Command History window on the MATLAB desktop. These commands remain from session to session until you choose to delete them, and they can be referenced from the command prompt using the up arrow key ↑ or by dragging (click and holding) them from the Command History and dropping them at the prompt. You can select a range of commands in the history by holding down the **Shift** key, or separated commands by holding down the **Ctrl** key.

Right-clicking in the Command History displays a context menu. Select the commands you typed to model the whale call, right-click, and choose **Create M-file** from the menu.

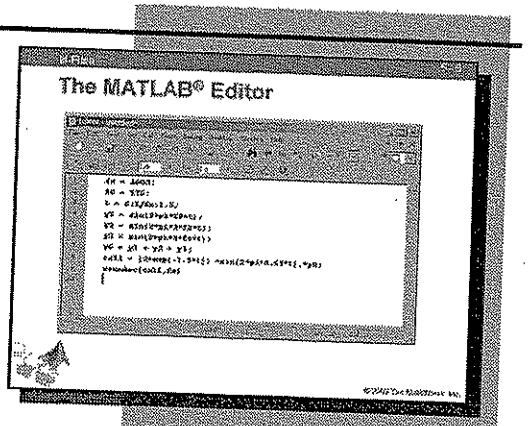
Try

```
>> fs = 4000;
>> f0 = 175;
>> t = 0:1/fs:1.5;
>> y1 = ...
sin(2*pi*f0*t);
>> y2 = ...
sin(2*pi*2*f0*t);
>> y3 = ...
sin(2*pi*3*f0*t);
>> y0 = y1 + y2 + y3;
>> call = ...
(2*exp(-1.5*t)).* ...
sin(2*pi*0.65*t).*y0;
>> soundsc(call,fs)
```

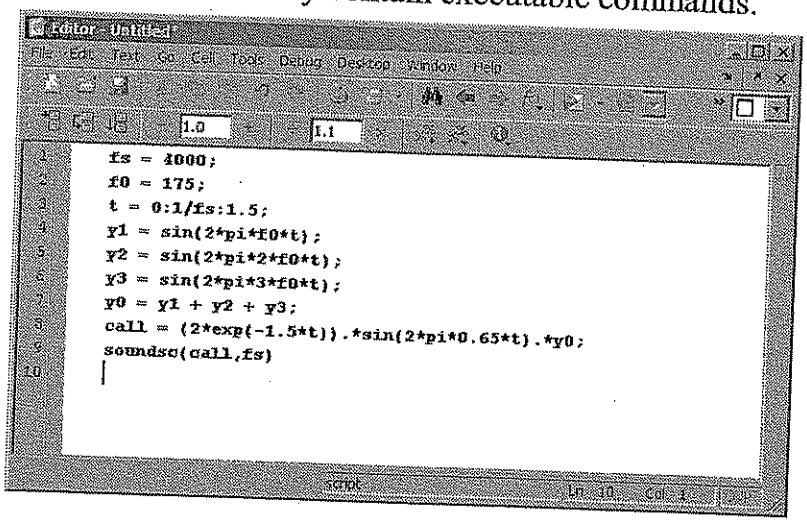
Select the above sequence of commands in the Command History, right-click, and choose **Create M-file** from the menu.



The MATLAB® Editor



When you choose **Create M-file** from the Command History, the selected commands are displayed in the MATLAB Editor. The MATLAB Editor is used to write, edit, run, debug, and publish *M-files*. M-files are simply sequences of MATLAB commands, saved to a file. M-files always have a `.m` extension, so MATLAB knows they contain executable commands.



Use the Editor to view and edit the thousands of M-files that ship with MATLAB or the M-files you write yourself to extend MATLAB.

To open the Editor and begin a new M-file, type

```
>> edit
```

You may also select **File → New → M-file** from the menus at the top of the MATLAB desktop, or select the New M-file icon in the upper-left corner of the MATLAB desktop.

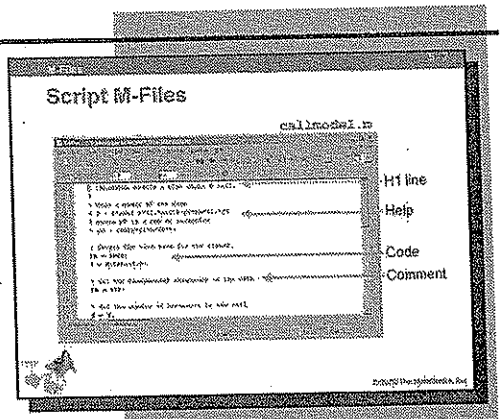
To open an existing M-file in the Editor, type

```
>> edit <filename>
```

where `<filename>` is the filename. (The `.m` is unnecessary.)

As a default preference, the Editor regularly creates *autosave* (`.asv`) duplicates of files open in the Editor. To adjust or disable autosave, choose **File → Preferences → Editor/Debugger**.

Script M-Files



The sequence of commands you have just opened in the Editor is a *script*. Scripts do not have inputs and outputs. When you run a script, MATLAB executes the commands as a batch, in order. Scripts are used to automate useful blocks of commands.

Although scripts do not take input arguments, they can operate on data in the workspace, and they can create new data on which to operate. Similarly, although scripts do not return output arguments, variables they create remain in the workspace and can be used in subsequent computations. Scripts can also produce graphical output by calling appropriate plotting functions.

M-files should contain *comments* (nonexecutable code) to improve their usability. Text following a percent sign (`%`) on a given line in an M-file is comment text. Comments can appear on lines by themselves, or they can be appended to the end of a line.

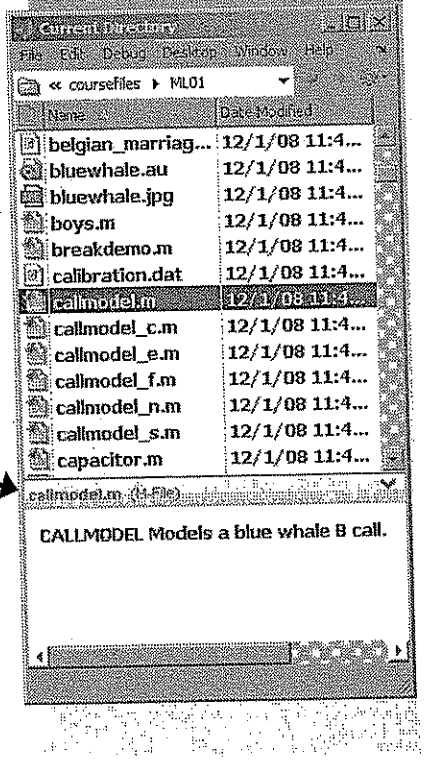
The first contiguous block of comments in an M-file is reserved for help information. Help is displayed when you type

```
>> help <filename>
```

The first comment line in the help is called the *H1 line*. It should contain the filename and a brief description of the purpose of the file. The H1 line is displayed in the Details Panel at the bottom of the Current Directory Browser.

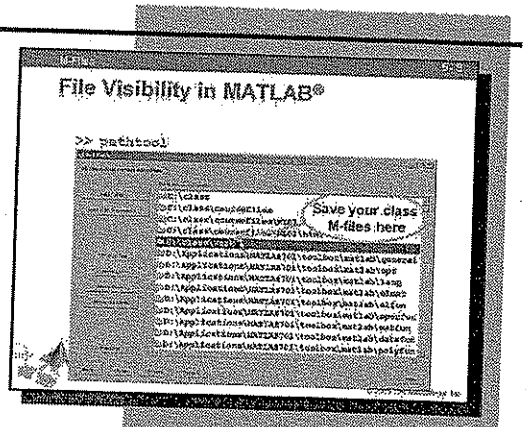
Try

Add an H1 line, help, and comments to your code.



Note MATLAB help uses UPPERCASE characters for M-files and variable names to make them stand out in text. However, when typing filenames (names of commands), use lowercase characters.

File Visibility in MATLAB®

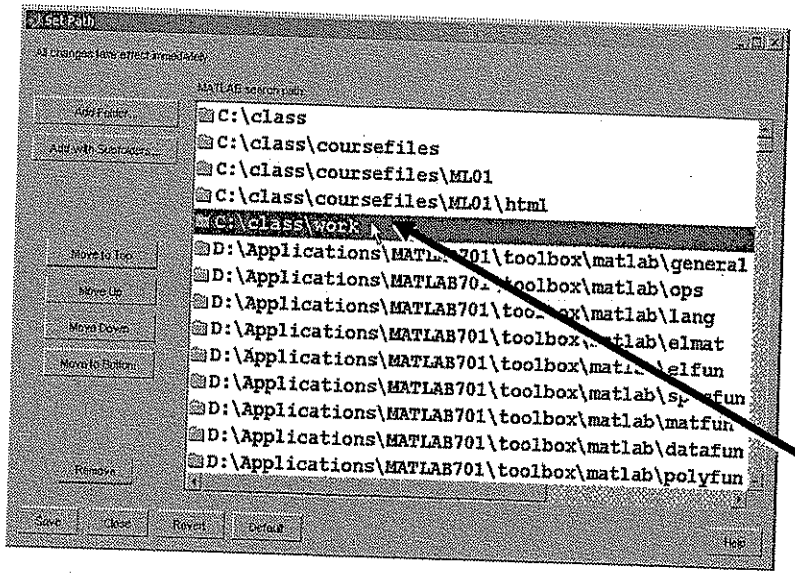


To make use of M-files (to run them from the MATLAB command prompt or from within other M-files), they must be on the MATLAB *search path* or in the current directory.

During installation, all files supplied with MATLAB and MathWorks toolboxes are placed on the search path. If you create new MATLAB related files, you must ensure that the directories containing the files are placed on the search path. Subdirectories must be explicitly added, even if parent directories are on the path.

To set and edit the search path, open the **Set Path** GUI by typing

```
>> pathtool
```



Save your class M-files here

Try

```
>> path
>> pathtool
```

Save your code with the name `myfile.m` to the directory `C:\class\work` and add the directory to the search path with the **Set Path** GUI.

```
>> which myfile -all
```

Give your file precedence over any other files with the same name by editing the path.

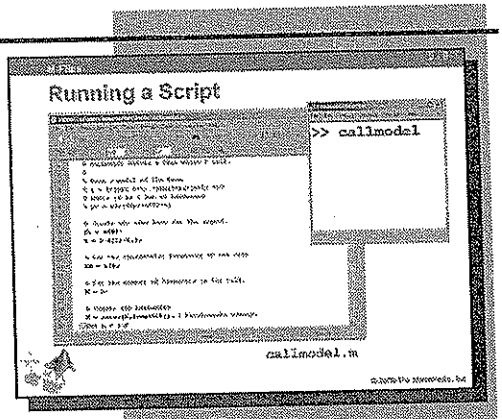
```
>> which whalecall
>> which nosuchfile
>> matlabroot
>> which pathtool
```

When MATLAB parses a command, like `whale`, it follows a *precedence order* to determine its meaning:

1. Looks for `whale` as a variable in the workspace
2. Looks in the current directory for a file named `whale`
3. Searches the path, in order, for a file named `whale`

The order of the directories on the search path is important if you have more than one file with the same name (independent of the extension). When MATLAB looks for a file, only the one with highest precedence is found. Other files with the same name are said to be *shadowed*.

Running a Script



After a script has been saved to the current directory or one of the directories on the MATLAB search path, it can be run by typing the filename (without the `.m` extension) at the prompt:

```
>> <filename>
```

For example, the file `callmodel.m` contains an implementation of the whale call model, including displays comparable to those used in `whalecall.m` to analyze the original signal. The file is located in the `C:\class\coursefiles\ML01` directory, which was added to the search path when the course files were installed. Open it in the editor by typing

```
>> edit callmodel
```

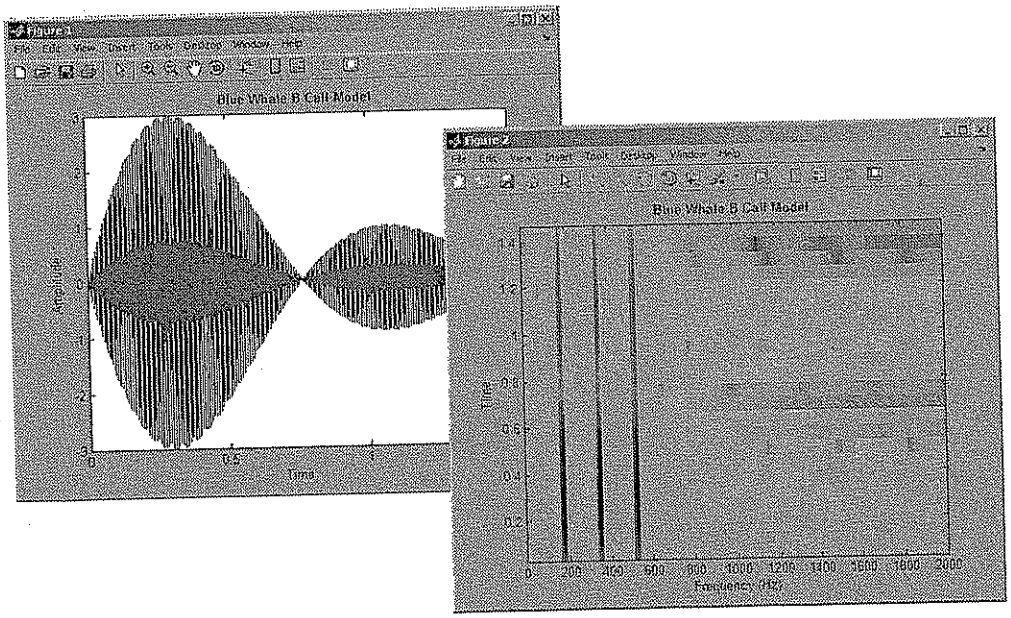
Run it by typing

```
>> callmodel
```

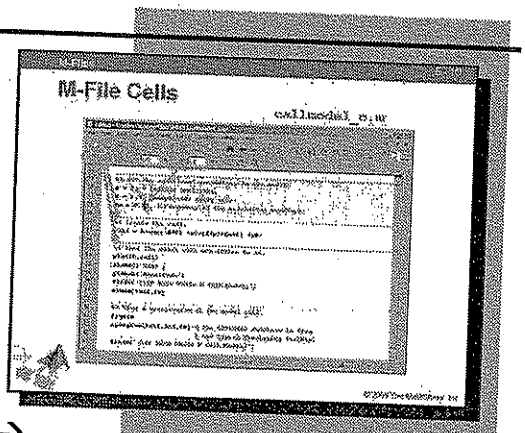
If a script is displayed in the Editor, you can run it by clicking on the Run button at the top of the Editor. You can run just a portion of a script by highlighting the desired lines of code and pressing **F9**.

Try

```
>> myfile

>> edit callmodel
>> callmodel
```



M-File Cells



The overall structure of most M-file scripts naturally divides into sections. Especially for larger files, you may want to focus on one section at a time. To facilitate this, The MATLAB Editor allows you to create M-file *cells*.

In the MATLAB Editor, you enable cell mode by selecting **Cell → Enable Cell Mode**. This mode makes all items in the **Cell** menu selectable and displays the **Cell** toolbar.

A cell consists of a line starting with `%%` and the lines that follow, up to the start of the next cell, which is identified by another `%%`. You define a cell by typing `%%` at the start of the line where you want it to begin. After the `%%`, use rest of the line for a *cell title*.

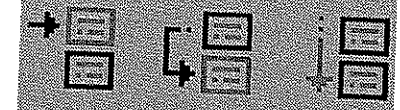
After you define cells, you can use the options in the **Cell** menu or the **Cell** toolbar to navigate from cell to cell, evaluate the code in a cell, and examine results one cell at a time. The cell evaluation options run the code currently shown in the Editor, even if the file contains unsaved changes. The file does not have to be on the search path.

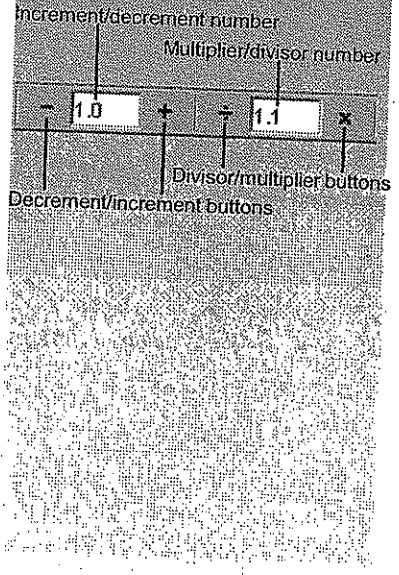
Cell mode allows you to modify values in cells and then reevaluate them to see how they impact the output. To modify a cell value, select it (or place the cursor near it) and use the Value Modification Tool in the **Cell** toolbar. Specify a modification number, and click the math operator to add (increment), subtract (decrement), multiply or divide the number. The cell will reevaluate automatically.

Try

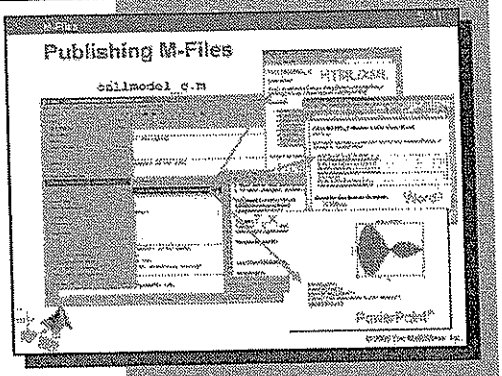
```
>> edit callmodel_c
```

Enable cell mode, step through the cells, and experiment with modifying model parameters.





Publishing M-Files



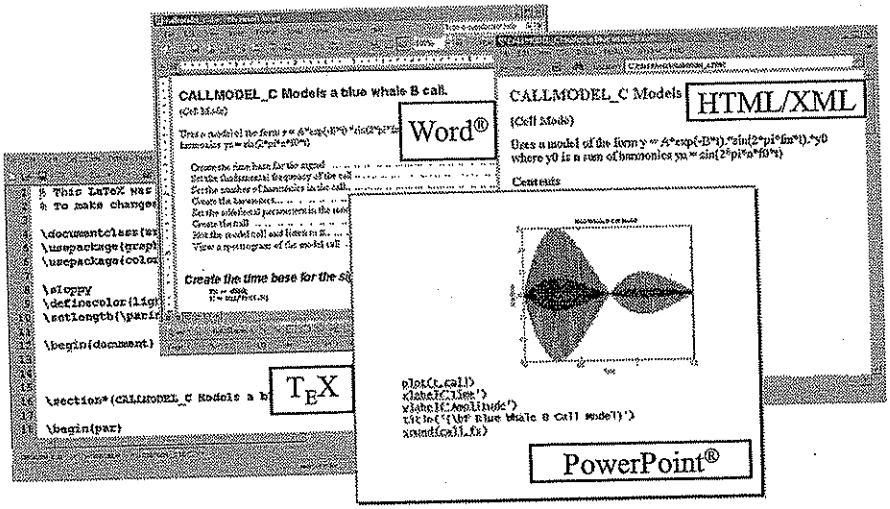
When you have completed writing an M-file script, you can use the cell features in the MATLAB Editor to publish the M-file and its results in any of several presentation formats: HTML, XML, or, when the applications are installed, Microsoft® Word® or PowerPoint®. Using the cell features allows you to share your work with others, presenting not only the code, but also comments on the code and results from running the file, including graphics.

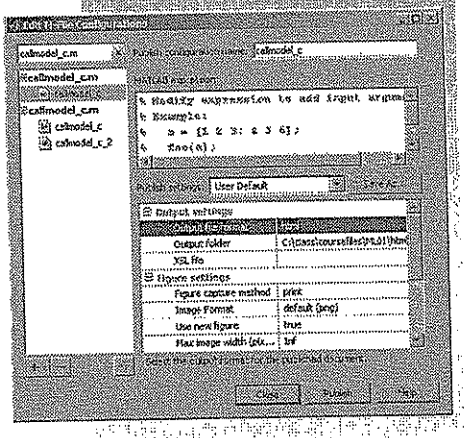
Try

Publishing `callmodel_c` in various formats.

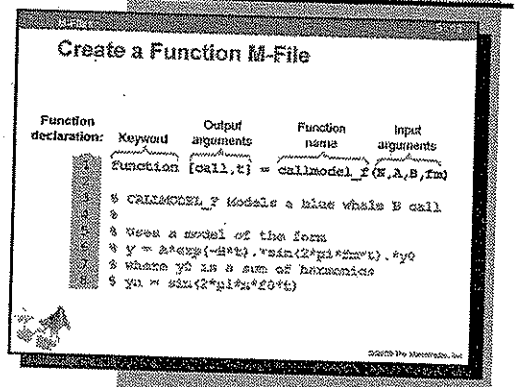
To publish an M-file using cell features in the Editor:

1. Enable cell mode and define cells as described on the previous page. For formatted publishing, all comments must appear at the start of a cell, before the code. Comments appearing after code in a cell appear as unformatted comments in code in the output.
2. (Optional) Select **Cell → Insert Text Markup** to insert markup symbols in the M-file comments to stylize the text for the presentation format, including TEX formatting of equations.
3. Select **File → Publish Configurations for [filename]** to select the format in which you want to publish the M-file: HTML, XML, TEX, Word, or PowerPoint. You can also set the preferences to adjust the output. For example, you can choose to include or exclude the code from the output.
4. Select **File → Publish [filename]**.





Create a Function M-File



Functions are programs that accept inputs and return outputs. They operate within their own *function workspace*, separate from the *base workspace* accessed at the command prompt or from within scripts. If a function calls another function, there is a *calling workspace* and a *called workspace*.

The separate workspaces of functions are convenient, but also a possible source of error.

The convenience comes with variable naming. You could, for example, have all of your functions accept input `x` and return output `y`. There would never be any confusion because the values of the variables are *local* to the individual function workspaces. There is no need to worry about variable name conflicts among your function M-files.

A common source of error in M-file programming is to refer to a variable in one function workspace (or the base workspace) from another function workspace. Because the workspaces are separated in memory, their variables are hidden from one another. MATLAB will tell you that the variable is not found.

MATLAB does maintain a *global workspace*, and you can use it to make variables visible across function workspaces. The preferred practice, however, is to share variables among function workspaces by explicitly passing the variables as function inputs.

Another difference between functions and scripts is syntactic. Function M-files always begin with a *function declaration*:

```
function [out1,out2,...] = function_name(in1,in2,...)
```

Values are assigned to input variables when the function is called with specific inputs. Code following the function declaration describes how output variables are computed from input variables. Each declared output must be assigned a value somewhere in the script. For functions with a single output, the square brackets are unnecessary (and slightly less efficient). Some functions (e.g., plotting functions) have no outputs.

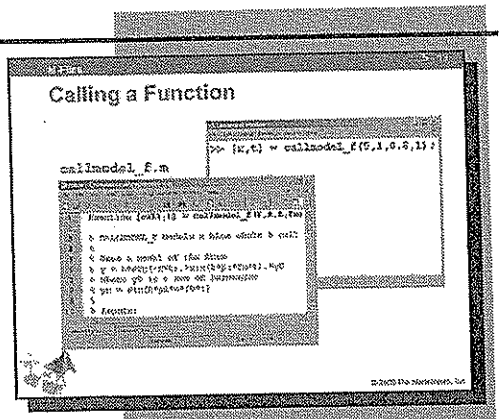
Try

Edit `callmodel.m` to turn it into a function that accepts the model parameters as inputs and returns the model call and time base as outputs.

Solution

```
>> edit callmodel_f
```

Calling a Function



You call a function from the command prompt using the syntax specified in the function declaration on line 1 of the M-file. However, you must assign input arguments specific values.

For example, to run the whale call model with new parameters using the function version of the M-file, `callmodel_f.m`, type

```
>> N = 5;
>> A = 1;
>> B = 0.8;
>> fm = 1;
>> [x,t] = callmodel_f(N,A,B,fm);
```

When the function is called, a separate workspace for `callmodel_f` is created in memory. The variables `N`, `A`, `B`, and `fm` are assigned local values passed from the base workspace, and these values are used to compute the outputs. The outputs are returned to the base workspace and assigned to the variables `x` and `t`. The function workspace is then cleared from memory, deleting all local variables.

If a function is written to return multiple outputs, it is not necessary to return all of the outputs to the calling workspace. For example, if only the output `x` is needed, call `callmodel_f` using

```
>> x = callmodel_f(N,A,B,fm);
```

Try

Previous parameters

```
>> N = 3;
>> A = 2;
>> B = 1.5;
>> fm = 0.65
>> [x,t] = ...
callmodel_f(...
N,A,B,fm);
```

New parameters

```
>> N = 5;
>> A = 1;
>> B = 0.8;
>> fm = 1;
>> [x,t] = ...
callmodel_f(...
N,A,B,fm);
```

There's more!

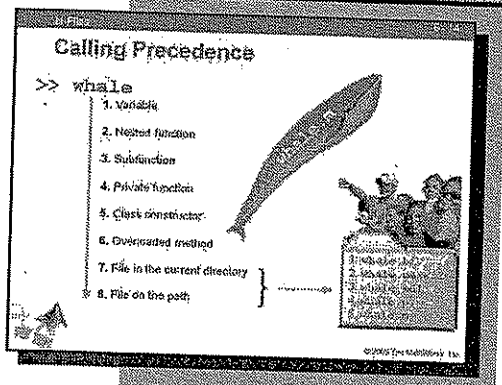
A function M-file can contain another function, called a *subfunction*, that the primary function can call, but that cannot be called from outside that primary function.

See Appendix D-3

There are other ways to organize functions and m-files, each with particular benefits and weaknesses.

Course offering: *MATLAB Programming Techniques.*

Calling Precedence



The precedence order used by the MATLAB interpreter, given in the earlier discussion of the MATLAB search path, is complicated when files other than scripts are involved. A fuller description of the precedence order is given here, for reference.

If you enter `whale` at the command prompt, MATLAB performs the following actions, in order, until it finds the first instance of `whale` that it can use to interpret the command:

1. Looks for `whale` as a variable in the local workspace.
2. Checks for `whale` as a nested function in the same M-file.
3. Checks for `whale` as a subfunction in the same M-file.
4. Checks for `whale` as a private function of the calling function.
5. Looks in the current directory for a file named `whale`.
6. Searches the path, in order, for a file named `whale`.

Private functions are stored in a `private` subdirectory off the path, and are accessible only from the parent directory.

If there is more than one file named `whale` in the directory found in step 6, MATLAB gives precedence according to extension:

1. Built-in files (`.bi` extension)
2. MEX-files (`.mexw32`, `.mexmac`, `.mexglx`, etc. extension)
3. MDL-files (`.mdl` extension)
4. P-files (`.p` extension)
5. M-files (`.m` extension)

MEX-files are MATLAB executables – external C or Fortran routines that run within the MATLAB environment. MDL-files are Simulink models used to describe dynamic systems. P-files contain MATLAB code preparsed by the interpreter.

Built-in files are executables for certain MATLAB functions that are used frequently or take more time to run. Unlike M-files, you cannot see the source code of a built-in function. Most built-in functions do have an M-file associated with them, but this file is there mainly to supply the help information for the function.

Try

Precedence problem: using file names as variable names

1)
```
>> which pi -all
>> pi = 2;
>> which pi -all
>> R = 3;
>> pi*R^2
>> clear pi
>> which pi -all
>> pi*R^2
```

2)
```
>> sin = pi;
>> sin(1)
```

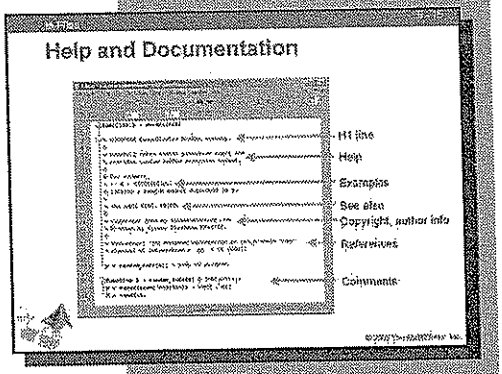
3)
```
>> i = 1:5;
>> z = 1 + 2*i;
>> abs(z)
>> clear i
>> z = 1 + 2*i;
>> abs(z)
```

Comparing nested function and subfunction precedence

```
>> edit nestvssub
>> nestvssub(0)
```

Help and Documentation

M-file documentation is often as important as the M-file code itself, for both reference and communication. Some common components of help and comments are shown in the following figure.

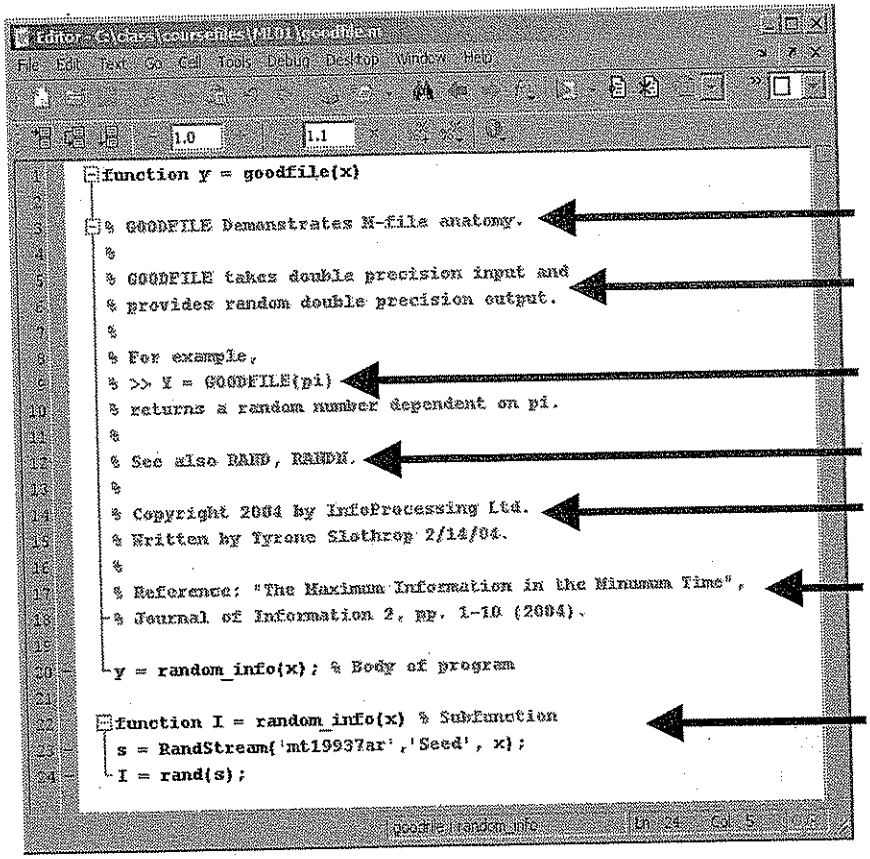


Try

```
>> edit goodfile
```

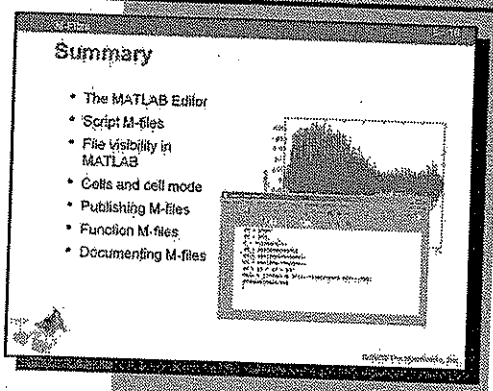
Change directory to the `html` subdirectory of the `coursefiles` directory

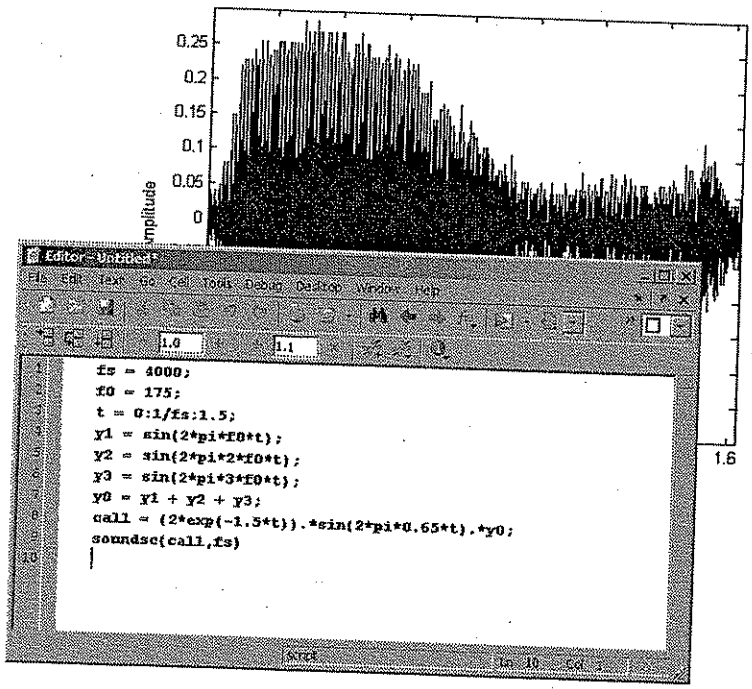
```
>> open(...
'goodfile.html')
```



Summary

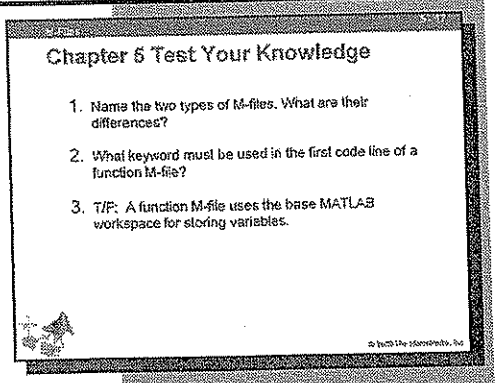
- The MATLAB Editor
- Script M-files
- File visibility in MATLAB
- Cells and cell mode
- Publishing M-files
- Function M-files
- Documenting M-files





Chapter 5 Test Your Knowledge

Name:_____



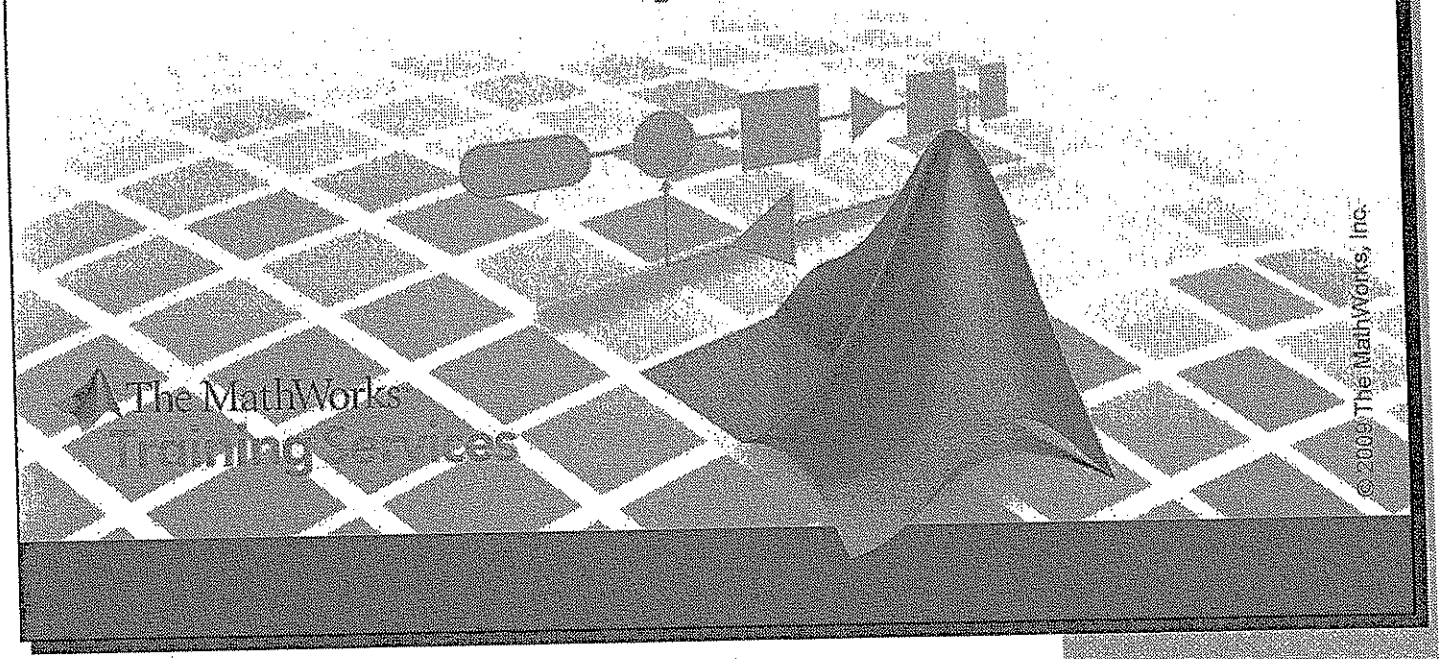
1. Name the two types of M-files. What are their differences?

2. What keyword must be used in the first code line of a function M-file?

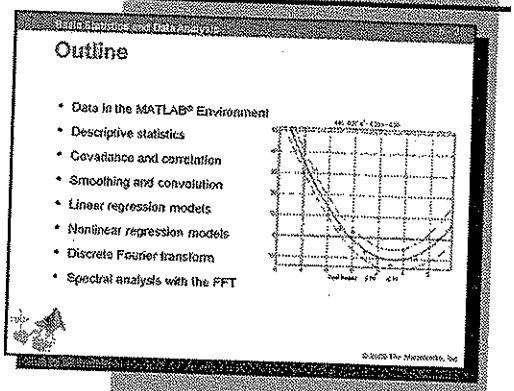
3. T/F: A function M-file uses the base MATLAB workspace for storing variables.

MATLAB® Fundamentals

Basic Statistics and Data Analysis

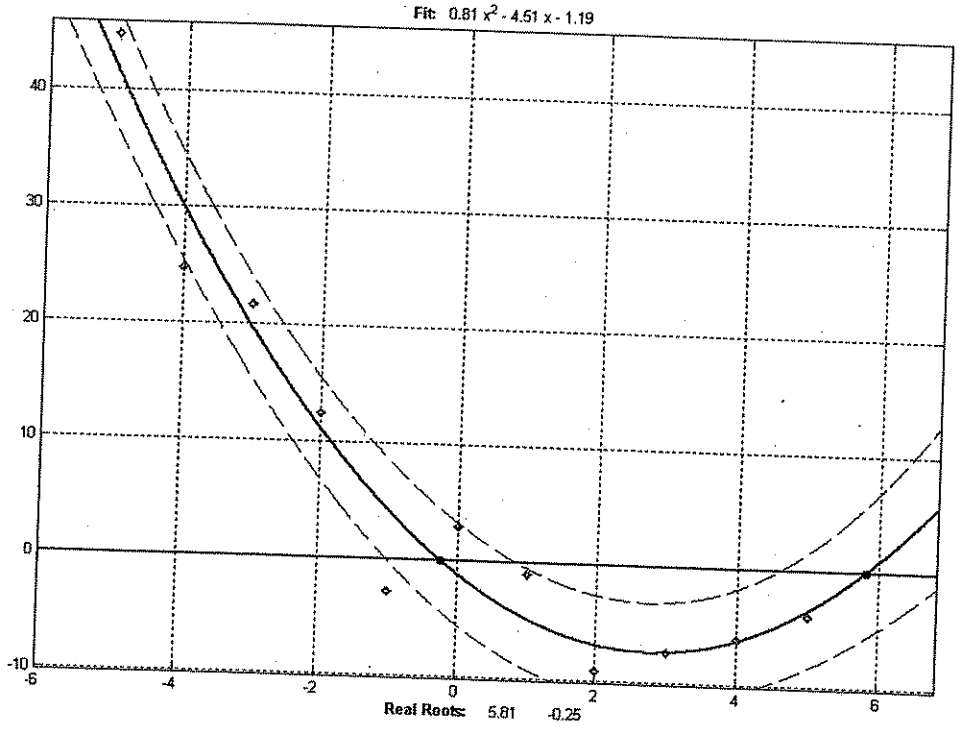


Outline

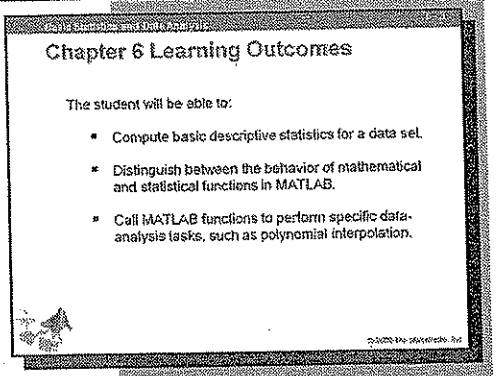


- Data in the MATLAB® Environment
- Descriptive statistics
- Covariance and correlation
- Smoothing and convolution
- Linear regression models
- Nonlinear regression models
- Discrete Fourier transform
- Spectral analysis with the fast Fourier transform (FFT)

This chapter highlights the data processing capabilities of MATLAB by looking at a few of the most common tools used in statistical analysis. MATLAB and the Statistics Toolbox™ have an extensive library of statistical functions and visualization methods that go well beyond the topics covered in this chapter. The goal of this chapter is to become familiar with the basic set up for carrying out common statistical tasks.



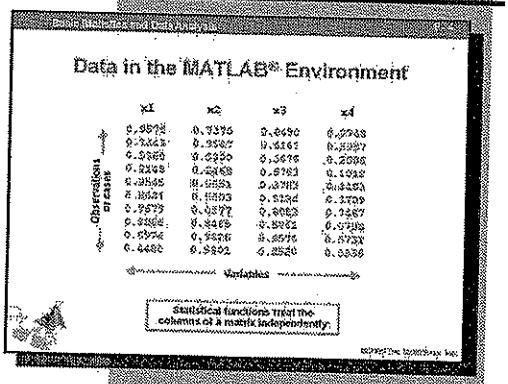
Chapter 6 Learning Outcomes



The student will be able to:

- Compute basic descriptive statistics for a data set.
- Distinguish between the behavior of mathematical and statistical functions in MATLAB.
- Call MATLAB functions to perform specific data-analysis tasks, such as polynomial interpolation.

Data in the MATLAB® Environment



In MATLAB, data is placed into "data containers" (variables) in the form of arrays. For statistical purposes, the rows of an array represent different observations or cases and the columns represent different variables.

	x1	x2	x3	x4
Observations or cases ↑	0.9501	0.6154	0.0579	0.0153
	0.2311	0.7919	0.3529	0.7468
	0.6068	0.9218	0.8132	0.4451
	0.4860	0.7382	0.0099	0.9318
	0.8913	0.1763	0.1389	0.4660
	0.7621	0.4057	0.2028	0.4186
	0.4565	0.9355	0.1987	0.8462
	0.0185	0.9169	0.6038	0.5252
	0.8214	0.4103	0.2722	0.2026
↓	0.4447	0.8936	0.1988	0.6721

← Variables →

One of the advantages of working in MATLAB is that functions operate on an entire array of data, not just one scalar value at a time. This capability allows both efficient expression and efficient computation.

When statistical functions in MATLAB operate on a vector of data (either a row vector or a column vector) they return a single computed statistic. When those functions operate on an array of data, they treat the columns independently, as separate variables, and return a vector of statistics–one for each column.

Try

Currency exchange data

```
>> load cx
>> edit cxviews
>> cxviews
```

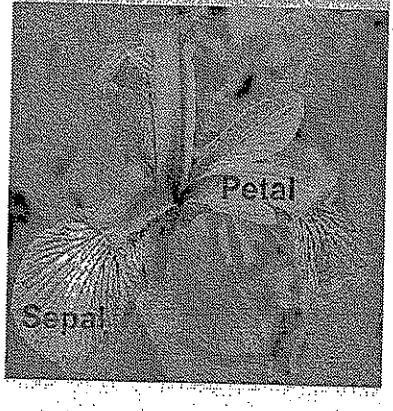
Fisher's iris data (1936)

- 50 specimens from each of 3 species: setosa, versicolor, and virginica.
- Measured variables, in order: sepal length, sepal width, petal length, and petal width

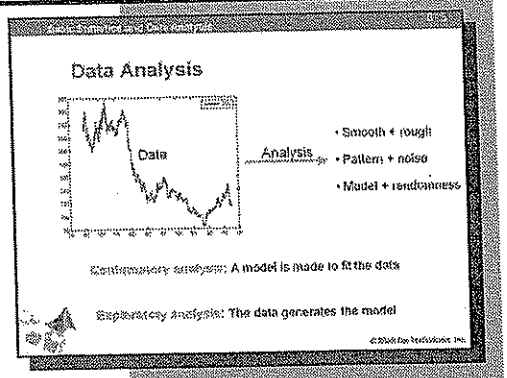
```
>> open('iris.fig')
>> load fisheriris
>> setosa = ...
meas(1:50,:);
>> mean(setosa)
```

Compare with:

```
>> mean(setosa(:))
```



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 decompositions describe the belief that patterns exist in the system being observed and that those patterns can be found in experimental data. The *rough* and the *noise* represent limits of measurement techniques, which are usually fixed by equipment and methodology.

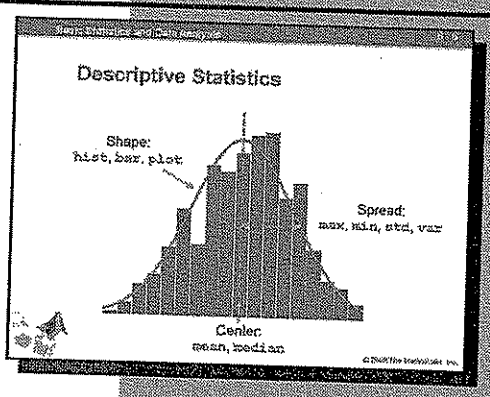
When building models, randomness is included to represent random processes in either the system under study or the data acquisition process. By comparing model outputs to measured behaviors of the system, inaccuracy can often be reduced through systematic revisions.

One approach to data analysis is to begin in the confirmatory mode, where a predetermined model is adjusted to fit data. This approach is used in engineering analyses, when a model has fixed specifications. Small, randomly distributed rough indicates a good fit, but does not mean that any real understanding of the system has been achieved.

In science, an exploratory mode is usually advocated. Beginning with a simple model, the rough is repeatedly mined for more smooth. The goal is to let the data itself generate the model.

Data visualization is an extremely important part of any analysis.

Descriptive Statistics



The distribution of a random variable over many observations can be described by its

- Center
- Spread
- Shape

The *center* (or *location*) of a distribution is its overall position 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* (or *scale*) of a distribution refers to its dispersion around the center. It is related to the volatility of the underlying parameter being measured and the limitations of the sampling process.

The *shape* of a distribution is more subtle, referring to its type (often categorized by the type of underlying process that leads to such a distribution), whether it is symmetric or skewed, whether it is single-peaked or multi-peaked, and whether it has outliers or gaps.

Summary statistics provide measures of a distribution's center, spread, and shape. To be useful for a wide variety of distributions, summary statistics should have some degree of resistance with respect to changes in a small proportion of the sampled values or to departures from the standard normal distribution.

In exploratory data analysis, summary statistics are combined with *visual summaries*, using the many plot types in MATLAB, for a more thorough understanding of data patterns.

Try

Currency exchange data

```
>> load cx
>> DEM = Data(:,7);
>> CHF = Data(:,9);
>> Rd = [DEM CHF];
>> Rs = ['DEM';'CHF'];
```

Shape

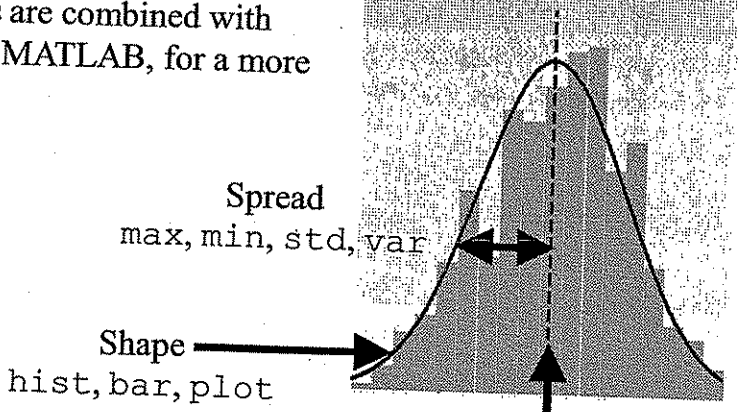
```
>> hist(Rd,100)
>> legend(Rs)
>> [n,x] = ...
hist(Rd,100);
>> bar(x,n,'stack')
>> legend(Rs)
```

Center

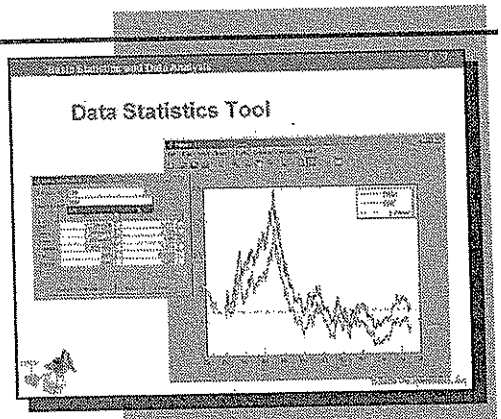
```
>> mean(Rd)
>> median(Rd)
```

Spread

```
>> max(Rd) - min(Rd)
>> var(Rd)
>> std(Rd)
```



Data Statistics Tool



The Data Statistics Tool is a simple, but convenient, way to combine summary statistics with visual summaries of your data. This combined numerical/graphical approach is at the heart of all good statistical practice.

To use the Data Statistics Tool, create any convenient 2-D plot of your data. Choose **Tools → Data Statistics** from the menus at the top of the figure window for the plot. The Data Statistics Tool opens in a separate dialog box and displays some basic summary statistics on the *x* and *y* data. Check the box next to a statistic to see it displayed with the plot.

If you have multiple plots on the same axes for comparison, you can also compare statistics. Choose a plot from the **Statistics for** pop-up menu at the top of the tool and check the desired statistics for plotting.

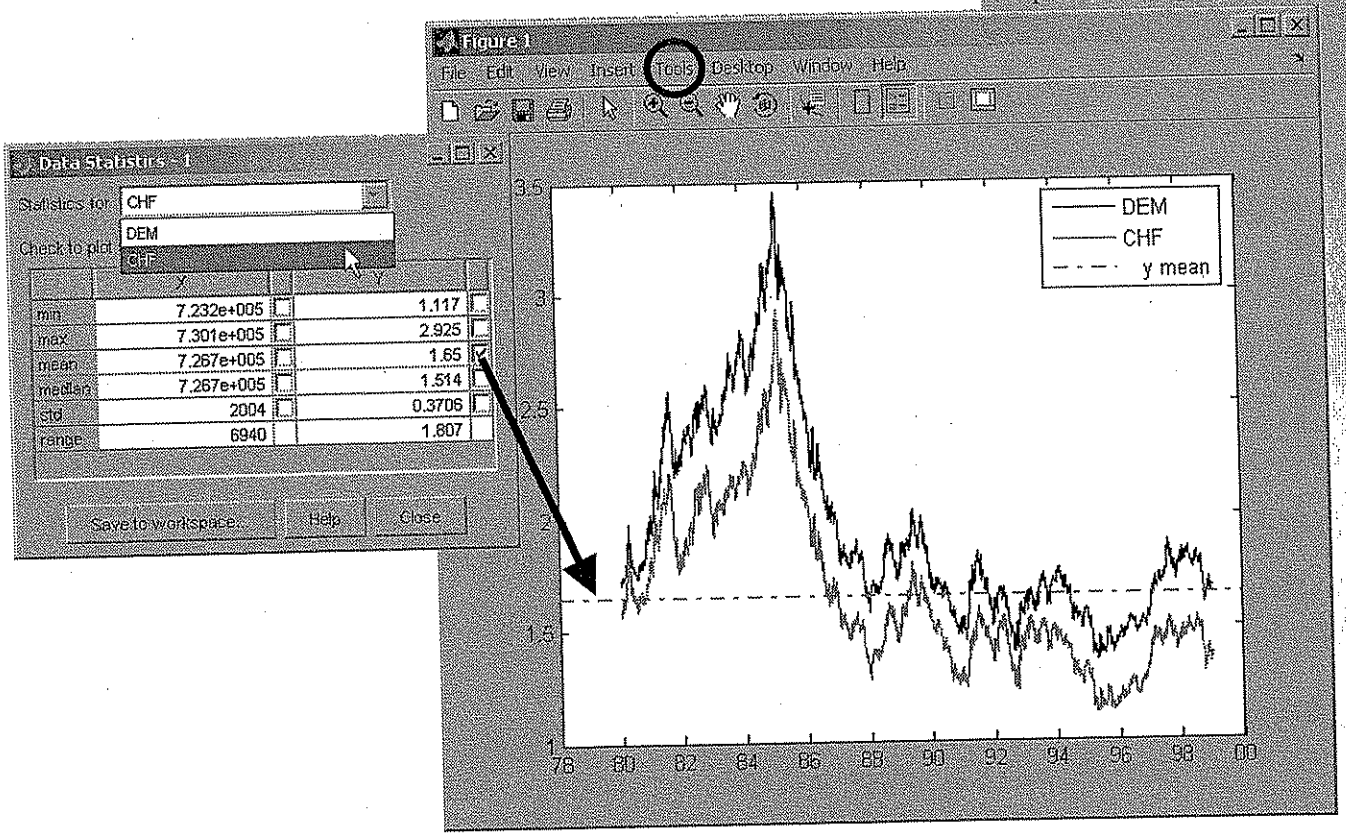
Click **Save to workspace** to create variables from the statistics.

Try

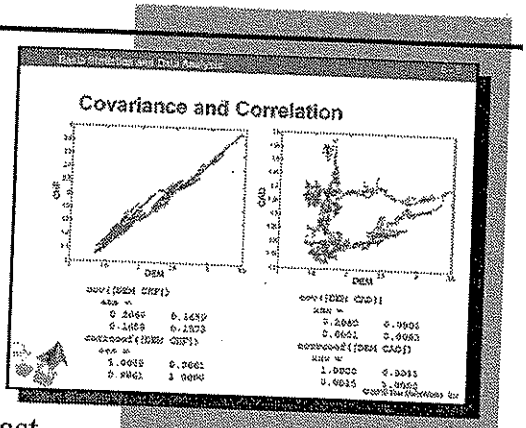
Currency exchange data

```
>> load cx
>> T = Data(:,1);
>> DEM = Data(:,7);
>> CHF = Data(:,9);
>> Rd = [DEM CHF];
>> Rs = ['DEM';'CHF'];
>> plot(T,Rd)
>> datetick('x','yy')
>> legend(Rs)
```

Use the Data Statistics Tool to compute and display statistics.



Covariance and Correlation



In statistics, a distinction is made between *predictor* variables (horizontal axes) and a *response* variable (vertical axis).

The strength of the *linear* relationship between two random variables is measured numerically by their *covariance*. Visually, the covariance determines how tightly the data will lie along a least squares line through a scatter plot of one variable vs. the other.

The covariance of vectors `X` and `Y` of length `m` is computed using

```
((X-mean(X))'* (Y-mean(Y))) / (m - 1)
```

When `X` and `Y` move together (occur jointly above or below their means), the products are positive. When `X` and `Y` move in opposite directions, the products are negative. The covariance averages these positive and negative contributions across all observations.

The covariance of a variable with itself is simply its variance.

Covariances can be normalized by dividing by the standard deviations of each variable. The *correlation coefficient*, r, that results has a value between +1 and –1. Normalization has the advantage of removing dependency on the units used to measure the variables.

The statistic r^2 is called the *coefficient of determination*. It gives the proportion of variation in a response that is eliminated or explained by the least squares line:

r^2 = (variance about the least squares line) / (variance about the mean).

r^2 is the fundamental measure of how well a least squares line predicts actual response data.

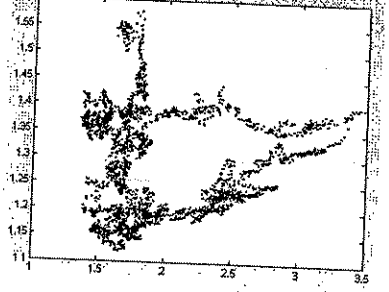
The MATLAB functions `cov` and `corrcoef` compute covariance and correlation, respectively, from a data sample. Results are displayed in a square matrix, with the entry in the ith row and jth column giving the covariance or correlation of the ith and jth variables in the data. The matrices are necessarily symmetric about the main diagonal.

Try

Currency exchange data

```
>> load cx
>> DEM = Data(:,7);
>> CHF = Data(:,9);
>> CAD = Data(:,4);
>> plot(...
DEM,CHF,'o',...
'MarkerSize',2)
>> cov([DEM CHF])
>> corrcoef([DEM CHF])
```

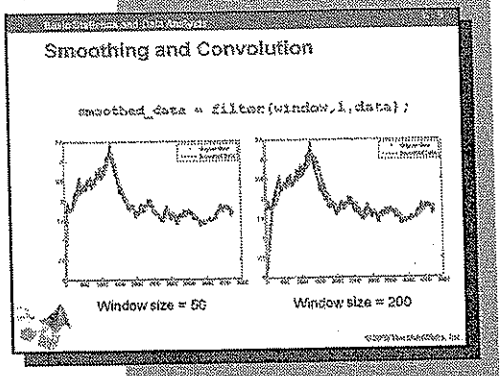
```
>> plot(...
DEM,CAD,'o',...
'MarkerSize',2)
>> cov([DEM CAD])
>> corrcoef([DEM CAD])
```



Grain prices and marriage rates

```
>> edit grainmarriage
>> grainmarriage
```

Smoothing and Convolution

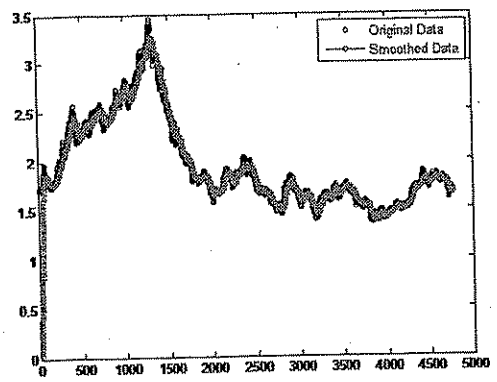


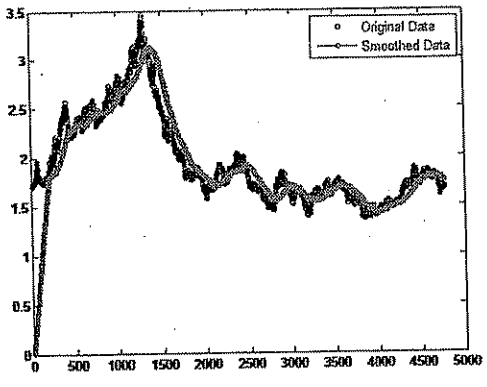
If your data is noisy, you might want to apply a smoothing algorithm to reveal its main features before building a model. Two basic assumptions underlie smoothing:

- The relationship between predictor and response is smooth.
- The smoothing process results in a smoothed value that is a better estimate of the original value because the noise has been reduced.

Response estimation is based on a specified number, called the *span*, of neighboring response values. The span defines a window of neighboring points to include in the smoothing calculation for each data point. This window moves across the data as the smoothed response value is calculated for each predictor. Large spans increase the smoothness but decrease the resolution of the smoothed data, while small spans decrease the smoothness but increase the resolution of the smoothed data. The optimal span depends on the data and the method, and usually requires some experimentation to find. Smoothing methods are generally less reliable on the edges of a data set.

Moving average smoothing is easily accomplished with the MATLAB convolution function `convn`. Convolution is a sum of products of data values and values in a *convolution kernel* (the window). Averages are just a special case. For example, to average the values `a`, `b`, and `c`, you form the sum of products `(1/3)*a + (1/3)*b + (1/3)*c`. The convolution kernel `[1/3 1/3 1/3]`, when convolved with a data vector, computes a moving average with span `3`.





Try

Convolution and filtering

```
>> x = [1 1 2 0 -1]
>> k = [1/3 1/3 1/3]
>> s1 = convn(x,k)
>> s2 = ...
convn(x,k,'same')
>> s3 = ...
filter(k,1,x)
```

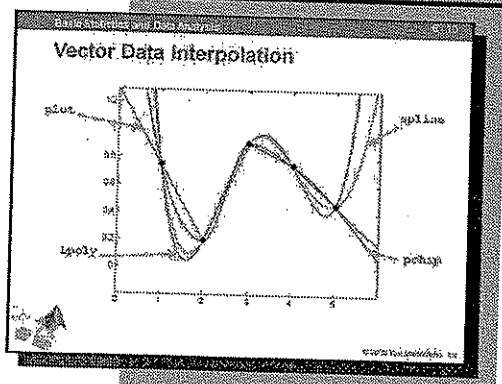
Currency exchange data

```
>> load cx
>> DEM = Data(:,7);
>> edit masmooth
>> masmooth(DEM,50);
>> masmooth(DEM,200);
```

2-D moving average

```
>> x = 1:0.25:2*pi;
>> y = 1:0.25:2*pi;
>> [X,Y] = ...
meshgrid(x,y);
>> Z = ...
sin(X).*sin(Y) + ...
rand(size(X));
>> edit masmooth2
>> ZS = ...
masmooth2(Z,5);
```

Vector Data Interpolation



Interpolation is the process of finding curves that pass exactly through a set of data. These curves are often used to model specifications or extremely reliable measurements and then to predict values between or beyond the data.

MATLAB and your course files provide a number of interpolation functions that let you balance the smoothness of the fit with speed of execution and memory usage. These functions include

`plot`	Linear interpolation
`Lpoly` (course file)	Lagrange polynomial interpolation
`spline`	Cubic spline interpolation
`pchip`	Piecewise cubic Hermite interpolation

The MATLAB function `interp1` incorporates several interpolation methods that can be chosen through a `method` argument.

```
>> yi = interp1(x,y,xi,method)
```

finds the values `yi`, at the points in the vector `xi`, of the interpolant that passes through the data in the vectors `x` and `y`. The `method` can be `'nearest'` (nearest neighbor interpolation), `'linear'`, `'spline'`, or `'pchip'`. (Lagrange polynomial interpolation is not supported by `interp1`; a separate course file, `Lpoly`, is provided for this purpose.)

Interpolation is the sometimes called *table lookup*. The "table" has columns for the data in `x` and `y`. You "look up" the values of `xi` in the table and, if they are not in the table lookup, their values are interpolated by `method`. Interpolation is the opposite of *decimation* (or *downsampling*), which reduces the number of values in the data for more efficient processing.

Try

Comparison of methods

```
>> x = 1:5;
>> y = rand(1,5);
>> interpgui(x,y)
```

Move the data with the mouse.

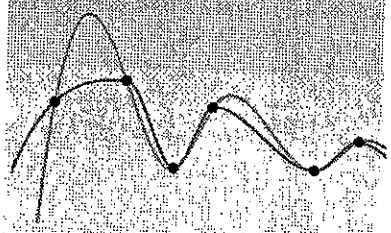
Lagrange polynomial

```
>> edit Lpoly
>> edit Linterp
>> Linterp
```

Run repeatedly.

Envelopes

```
>> edit envelope
>> edit envelopedemo
>> envelopedemo
```

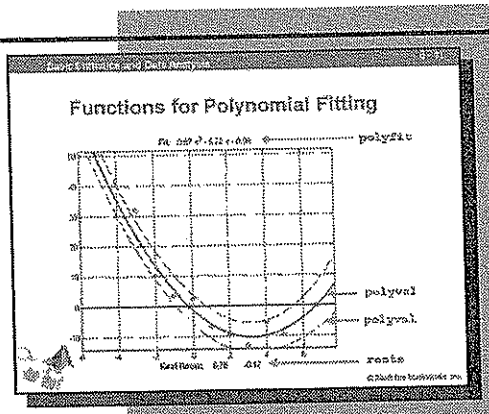


There's more!

MATLAB can also perform matrix data interpolation.

See Appendix D-4

Functions for Polynomial Fitting



To specifically fit polynomials to data, MATLAB offers a number of dedicated functions.

The MATLAB `polyfit` function computes the coefficients of a least squares polynomial fit of arbitrary degree. For example,

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

Polynomial coefficients are listed from highest to lowest degree, so $p(x) \approx -0.13x^3 + 0.69x^2 - 0.18x + 1.67$.

When the coefficients of a polynomial are expressed in a vector `p`, the MATLAB `polyval` function can be used to evaluate the polynomial at arbitrary inputs. For example,

```
>> xplot = -1:0.01:6;
>> yplot = polyval(p,xplot);
>> plot(xplot,yplot)
```

plots the fit between $x = -1$ and $x = 6$.

The MATLAB `roots` function can be used to find the roots of `p`:

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

The MATLAB function `poly` finds a polynomial with specified roots. It is the inverse of `roots` up to ordering, scaling, and roundoff error.

An optional output from `polyfit` can be passed to `polyval` to compute confidence intervals for the fit, assuming the errors in the y data are independent and normally distributed with constant variance.

Try

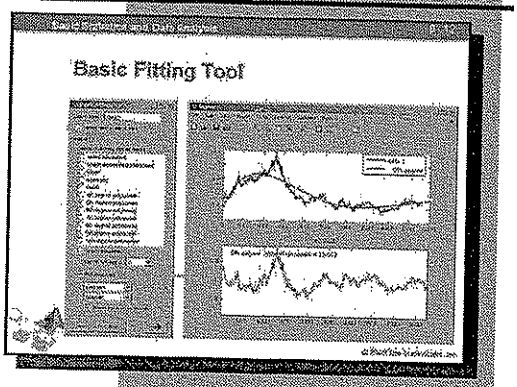
Currency exchange data
```
>> load cx
>> DEM = Data(:,7);
>> t = 1:length(DEM);
>> pbad = ...
polyfit(t',DEM,5);
```
Center and scale to avoid *bad conditioning* (that is, sensitivity of the coefficients to small changes in the data)
```
>> t = ...
(t-mean(t))/std(t);
>> pok = ...
polyfit(t',DEM,5);
>> plot(t,DEM,'b',...
t,polyval(pok,t),'r')
```

Convert coefficients to a string.
```
>> edit polystr
>> polystr([2 0 -1])
```

Polynomial fitting functions
```
>> edit polydemo
>> x = -5:5;
>> y = ...
x.^2 - 5*x - 3 + ...
5*randn(size(x));
>> polydemo(x,y,2)
```

Annual average marriages per month in Belgium, 1971–1990, above or below 20-year mean
```
>> load belgavg
>> polydemo(T,M,3)
>> polydemo(T,M,4)
>> polydemo(T,M,5)
```

Basic Fitting Tool

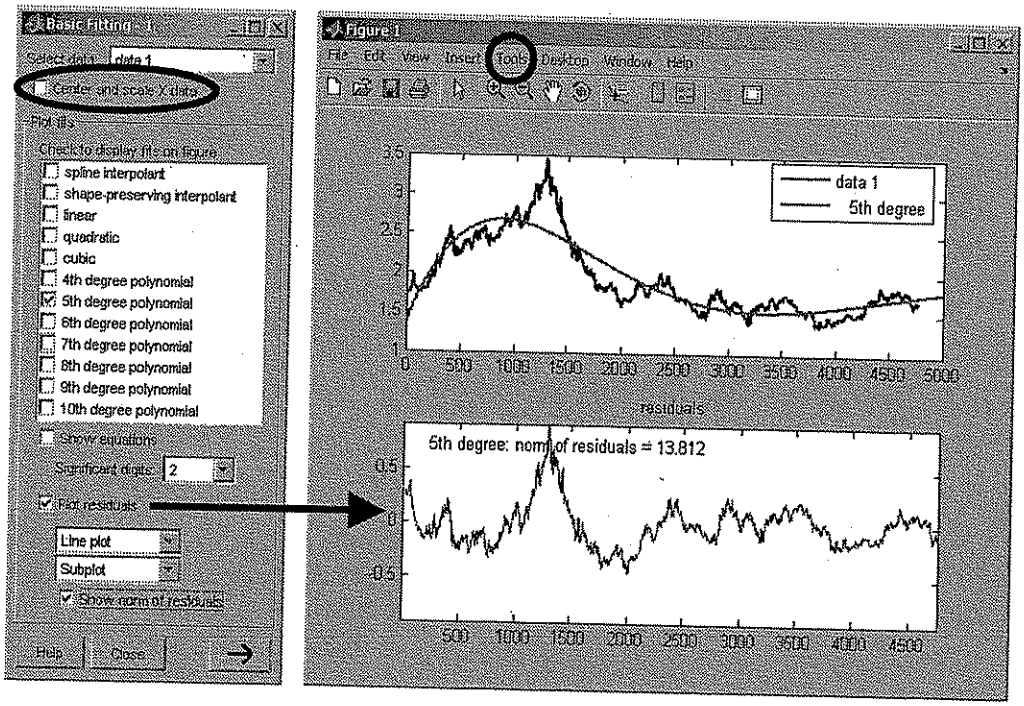


You can interactively apply MATLAB functions for polynomial fitting and vector data interpolation to your data using a graphical user interface (GUI) called the Basic Fitting Tool.

To use the tool, first create workspace variables with your data.

Next, plot your data using, e.g., the `plot` command. From the menu at the top of the figure window, choose **Tools → Basic Fitting**. This opens the first panel of the Basic Fitting Tool. Additional panels appear when you click the → button in the lower-right corner.

The Basic Fitting Tool allows you to fit various interpolants and polynomials of degree ≤ 10. It will plot residuals and compute their norm, and evaluate the fit over arbitrary input vectors `x`. Coefficients of polynomial fits (not interpolants), and values interpolated or extrapolated from the fit, can be saved to the workspace.



Try

Currency exchange data

```
>> load cx
>> DEM = Data(:,7);
>> plot(DEM)
```

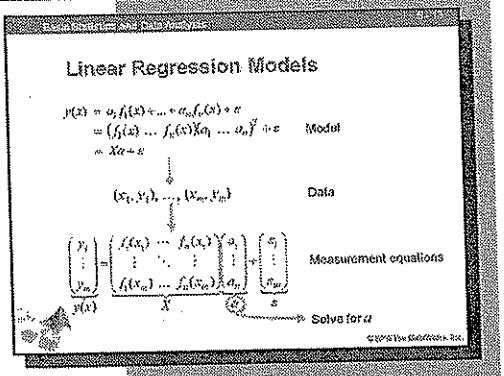
Use the Basic Fitting Tool to experiment with different fits. Select the **Center and scale X data** check box.

Annual average marriages per month in Belgium, 1971–1990, above or below 20-year mean.

```
>> load belgavg
```

Use the Basic Fitting Tool to experiment with different fits.

Linear Regression Models



Linear regression models represent the relationship between a continuous response variable y and a predictor variable x (continuous or categorical) in the form

$$\begin{aligned} y(x) &= a_1 f_1(x) + \ldots + a_n f_n(x) + \varepsilon \\ &= \begin{pmatrix} f_1(x) & \ldots & f_n(x) \end{pmatrix} \begin{pmatrix} a_1 & \ldots & a_n \end{pmatrix}^T + \varepsilon \\ &= Xa + \varepsilon \end{aligned}$$

The response is modeled as a linear combination of n (not necessarily linear) functions of the predictor, plus a random *residual* ε. The residual is usually assumed to be normally distributed with mean 0.

The parameters $a = (a_1 \ldots a_n)$ must be estimated to fit the model to a set of data. Given m joint observations $(x_1, y_1), \ldots, (x_m, y_m)$ of the predictor and the response, the equation above becomes an m x n system of *measurement equations*

$$\underbrace{\begin{pmatrix} y_1 \\ \vdots \\ y_m \end{pmatrix}}_{y(x)} = \underbrace{\begin{pmatrix} f_1(x_1) & \cdots & f_n(x_1) \\ \vdots & \ddots & \vdots \\ f_1(x_m) & \ldots & f_n(x_m) \end{pmatrix}}_{X} \underbrace{\begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix}}_{a} + \underbrace{\begin{pmatrix} \varepsilon_1 \\ \vdots \\ \varepsilon_m \end{pmatrix}}_{\varepsilon}$$

X is called the *design matrix*. If the columns are linearly independent, the functions f_j are called *basis functions*. To fit the model to the data, this system of equations must be solved for the n parameter values in a.

Examples of linear regression models include

- **Linear equations**. Basis functions are $f_1(x) = 1$ and $f_2(x) = x$.
- **Polynomials**. Basis functions are $f_1(x) = 1, f_2(x) = x, \ldots, f_n(x) = x^{n-1}$. The design matrix is formed from the *Vandermonde matrix*. The `polyfit` function sets up the Vandermonde matrix internally.
- **Chebyshev orthogonal polynomials**. Basis functions are $f_1(x) = 1$, $f_2(x) = x, \ldots, f_n(x) = 2x f_{n-1}(x) - f_{n-2}(x)$.
- **Trigonometric polynomials (Fourier series)**. The basis functions are $f_1(x) = 1/2$ and sines and cosines of different frequencies.

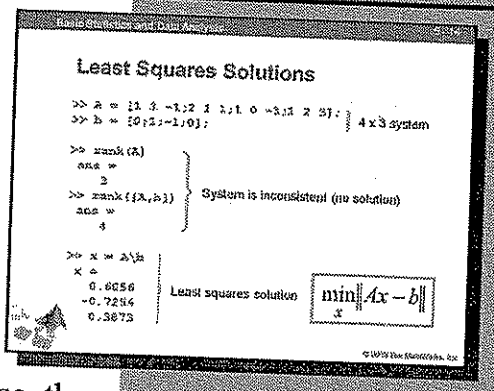
Try

Design matrices

```
>> doc vander
>> doc gallery
```

(see entry for chebvand)

Least Squares Solutions



If a system of linear equations has more equations (constraints) than variables (degrees of freedom), it may be *overdetermined*. Overdetermined systems are inconsistent (have no solution).

It is possible that some equations in a system express redundant information, as linear combinations of other equations. In this case, the number of equations is not a good measure of the number of distinct constraints on the variables. A more accurate measure is given by the *rank* of the coefficient matrix, computed by the MATLAB `rank` function. An *m*-by-*n* system $Ax = b$ is consistent (has a solution) if and only if `rank(A)` is equal to `rank([A,b])`.

When a system is overdetermined, the MATLAB backslash operator still returns values:

```
>> A = [1 1 -1;2 1 1;1 0 -3;1 2 3];
>> b = [0;1;-1;0];
>> rank(A)
 ans =
     3
>> rank([A,b])
 ans =
     4
>> x = A\b
 x =
   0.6056
  -0.7254
   0.3873
```

System is inconsistent (no solution)

For overdetermined systems, the backslash operator finds an x that minimizes the residual distance $\|Ax-b\|$. That is,

$$A \backslash b = \min_x \|Ax - b\|$$

This x is called a *least squares solution* to the system. For consistent systems, the norm is 0 and the solution is exact. For inconsistent systems, least squares solutions can approximate exact solutions that may be obscured by measurement errors in the coefficients.

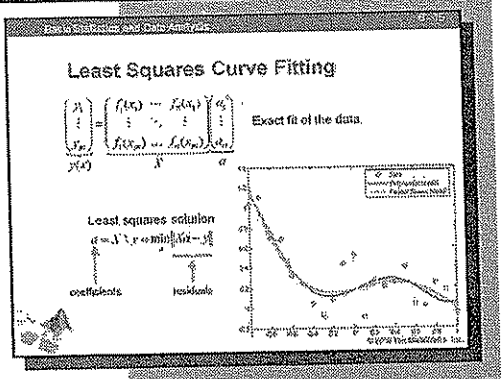
Try

```
>> 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
>> norm(A*x-b)
```

Compare:

```
>> y = rand(3,1);
>> norm(A*y-b)
```

Least Squares Curve Fitting



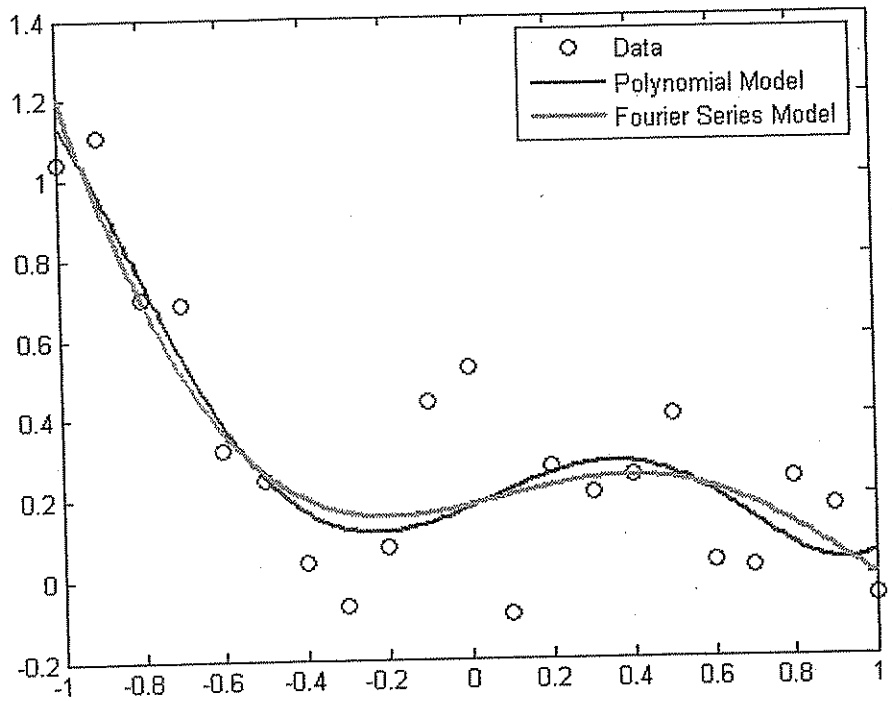
The least squares solution computed by the backslash operator allows you to fit linear regression models to data.

The measurement equations, without the residuals, describe an exact fit of the n parameter model $y = a_1 f_1(x) + \ldots + a_n f_n(x)$ to the m data points $(x_1, y_1), \ldots, (x_m, y_m)$

$$\underbrace{\begin{pmatrix} y_1 \\ \vdots \\ y_m \end{pmatrix}}_{y(x)} = \underbrace{\begin{pmatrix} f_1(x_1) & \cdots & f_n(x_1) \\ \vdots & \ddots & \vdots \\ f_1(x_m) & \ldots & f_n(x_m) \end{pmatrix}}_{X} \underbrace{\begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix}}_{a}$$

Typically, the number m of data points is larger than the number n of parameters in the model, and the m x n system is overdetermined. Backslash returns parameters a that minimize the sum of the squares of the distances between the model values in Xa and the data in y.

$$X \setminus y = \min_a \|Xa - y\| = \min_{a_1,\ldots,a_n} \sum_{i=1}^{m} \left[\left(\sum_{j=1}^{n} a_j f_j(x_i) \right) - y_i \right]^2$$

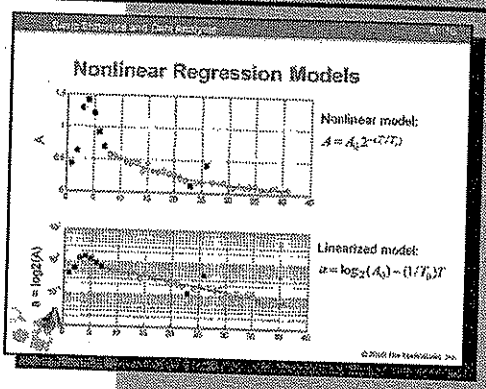


Try

```
>> edit simple_fit
>> simple_fit

>> edit lsfit
>> lsfit
```

Run repeatedly.

Nonlinear Regression Models



Not all regression models are linear. Coefficients may appear inside, rather than outside, the basis functions, and an exact fit of the data cannot be expressed by a system of linear equations. The backslash operator does not apply. Nonlinear least squares problems require a different approach to minimizing the residual distances between the data and the model. General problems of this type are handled by functions in the Optimization Toolbox™.

Some nonlinear least squares problems can be *reexpressed* as linear problems. In general, *reexpression* is the use of a scale of measurement other than the one in which variables were originally recorded. When it is used to convert a nonlinear problem to a linear problem, the process is often called *linearization*.

For example, exponential functions $y = a^x$ and power functions $y = x^a$ are both possible models of data displaying a concave upward trend. For exponentials, $\ln y = x \ln a$, and the data will show a simple linear trend when reexpressed using the variables $Y = \ln y$ and x. Likewise, for power functions, $\ln y = a \ln x$, and the data will show a simple linear trend when reexpressed using the variables $Y = \ln y$ and $X = \ln x$.

There are pitfalls to linearization. The problem is with transforms like the logarithm. Logarithms are nonlinear, and symmetric errors on the original scale may become asymmetric on a log scale. The transform disproportionately stretches out small values. When a linear fit is made on the log scale, it may be affected by "phantom outliers."

If data is modeled with multiplicative noise instead of additive noise, then errors in the original data may not be symmetric, and linear least squares on the original scale would not be appropriate. A log transform would make the errors symmetric on a log scale, and a linear least squares fit on that scale *would* be appropriate. The right method depends on the assumptions that you are willing to make about your data. In practice, when the noise is small relative to the trend (often not the case), the logarithm is locally linear in the sense that y values near the same x value will not be stretched out too asymmetrically. In that case, the two methods lead to essentially the same fit.

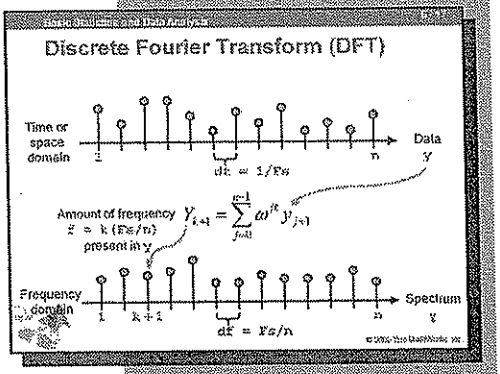
Try

Power vs. exponential models
```
>> edit reexp
>> reexp
```

Radioactivity data
```
>> edit milkfit
>> milkfit
```

Pitfalls of linearization
```
>> edit logpit
>> logpit
```

Discrete Fourier Transform (DFT)



It is often useful to decompose data into component frequencies. *Spectral analysis* gives an alternative view of time or space-based data in the *frequency domain*. The computational basis of spectral analysis is the *discrete Fourier transform* (DFT).

The DFT of a vector y of length n is another vector Y of length n

$$Y_{k+1} = \sum_{j=0}^{n-1} \omega^{jk} y_{j+1}$$

where ω is a complex n^{th} root of unity

$$\omega = e^{-2\pi i/n}$$

This notation uses i for the complex unit, and j and k for indices that run from 0 to n-1. The subscripts $j + 1$ and $k + 1$ run from 1 to n, corresponding to the range usually associated with MATLAB vectors.

Data in the vector y are assumed to be separated by a constant interval in time or space $dt = 1/Fs$. Fs is called the *sampling frequency* of y. The coefficient Y_{k+1} measures the amount of the frequency $f = k(Fs/n)$ that is present in the data in y. The vector Y is called the *spectrum* of y.

The midpoint of Y (or the point just to the right of the midpoint if n is even), corresponding to the frequency $f = Fs/2$, is called the *Nyquist point*. The real part of the DFT is symmetric about the Nyquist point and the imaginary part is antisymmetric about the Nyquist point.

The graphical user interface `fftgui` allows you to explore properties of the DFT. If `y` is a vector,

```
>> fftgui(y)
```

plots `real(y)`, `imag(y)`, `real(fft(y))`, and `imag(fft(y))`. You can use the mouse to move any of the points in any of the plots, and the points in the other plots respond.

Try

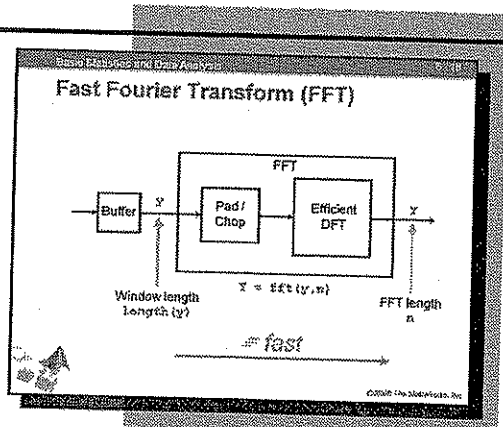
Roots of unity

```
>> edit zlroots
>> zlroots(3);
>> zlroots(7);
```

Explore the DFT

```
>> delta1 = ...
[1 zeros(1,11)];
>> fftgui(delta1)
>> delta2 = ...
[0 1 zeros(1,10)];
>> fftgui(delta2)
>> deltaNyq = ...
[zeros(1,6), ...
1,zeros(1,5)];
>> fftgui(deltaNyq)
>> square = ...
[zeros(1,4), ...
ones(1,4), ...
zeros(1,4)];
>> fftgui(square)
>> t = linspace(...
0,1,50);
>> periodic = ...
sin(2*pi*t);
>> fftgui(periodic)
```

Fast Fourier Transform (FFT)



The MATLAB function `fft`, called by `fftgui`, uses a *fast Fourier transform* (FFT) algorithm to compute the DFT.

DFTs with a million points are common in applications. For modern signal and image processing applications, and many other applications of the DFT, the key is the ability to do such computations rapidly. Direct application of the definition of the DFT requires n multiplications and n additions for each of the n coefficients –a total of $2n^2$ floating-point operations. This multiplication does not include the generation of the powers of ω. To do a million-point DFT, a computer capable of doing one multiplication and addition every microsecond would require a million seconds, or about 11.5 days.

Modern FFT algorithms have computational complexity $O(n\log_2 n)$ instead of $O(n^2)$. If n is a power of 2, a one-dimensional FFT of length n requires less than $3n\log_2 n$ floating-point operations. For $n = 2^{20}$, that is a factor of almost 35,000 faster than $2n^2$.

When using the FFT, a distinction is often made between a *window length* and an *FFT length*. The window length is the length of the input. It might be determined by, say, the size of an external buffer. The FFT length is the length of the output, the computed DFT.

```
>> Y = fft(y)
```

returns the DFT `Y` of `y`. The window length `length(y)` and the FFT length `length(Y)` are the same.

```
>> Y = fft(y,n)
```

returns a DFT `Y` with FFT length `n`. If the length of `y` is less than `n`, `y` is padded with trailing zeros to length `n`. If the length of `y` is greater than `n`, the sequence `y` is truncated. The FFT length is then the same as the padded/truncated version of the input `y`.

Try

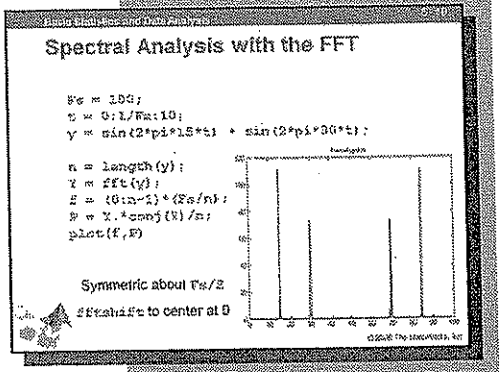
Vector data interpolation and the origins of the FFT

```
>> edit fftinterp
>> fftinterp
```

John Tukey (1915-2000)

Several people discovered fast DFT algorithms independently, and many people have since joined in their development, but it was a 1965 paper by John Tukey of Princeton University and John Cooley of IBM® Research that is generally credited as the starting point for the modern usage of the FFT. The MATLAB `fft` function is based on FFTW, "The Fastest Fourier Transform in the West," developed by MIT graduate students Matteo Frigo and Steven G. Johnson. (`http://www.fftw.org`)

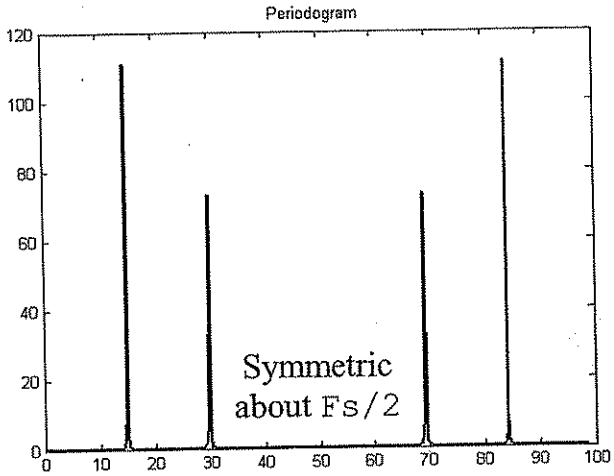
Spectral Analysis with the FFT



The FFT allows you to efficiently estimate the component frequencies in data from a discrete set of values sampled at a fixed rate. The following list shows the basic relationships among the various quantities involved in any spectral analysis. You can replace references to time by references to space.

`y`	Sampled data
`n = length(y)`	Number of samples
`Fs`	Samples/unit time
`dt = 1/Fs`	Time increment
`t = (0:n-1)/Fs`	Time range
`Y = fft(y)`	Discrete Fourier transform (DFT)
`abs(Y)`	Amplitude of the DFT
`(abs(Y).^2)/n`	Power of the DFT
`Fs/n`	Frequency increment
`f = (0:n-1)*(Fs/n)`	Frequency range
`Fs/2`	Nyquist frequency

A plot of the power spectrum is called a *periodogram*. The first half of the principal frequency range (from 0 to the Nyquist frequency `Fs/2`) is sufficient, because the second half is a reflection of the first half.



Spectra are sometimes plotted with a principal frequency range from `-Fs/2` to `Fs/2`. MATLAB provides the function `fftshift` to rearrange the outputs of `fft` and convert to a 0-centered spectrum.

Try

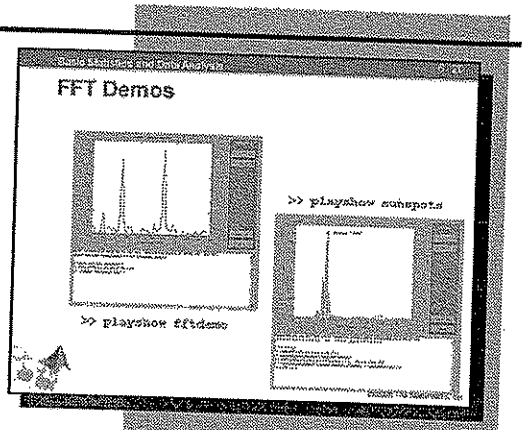
Periodograms
```
>> edit pgrams
>> pgrams
```

Whale call
```
>> edit whalefft
>> whalefft
```

Belgian marriage data
```
>> edit belgfft
>> belgfft
```

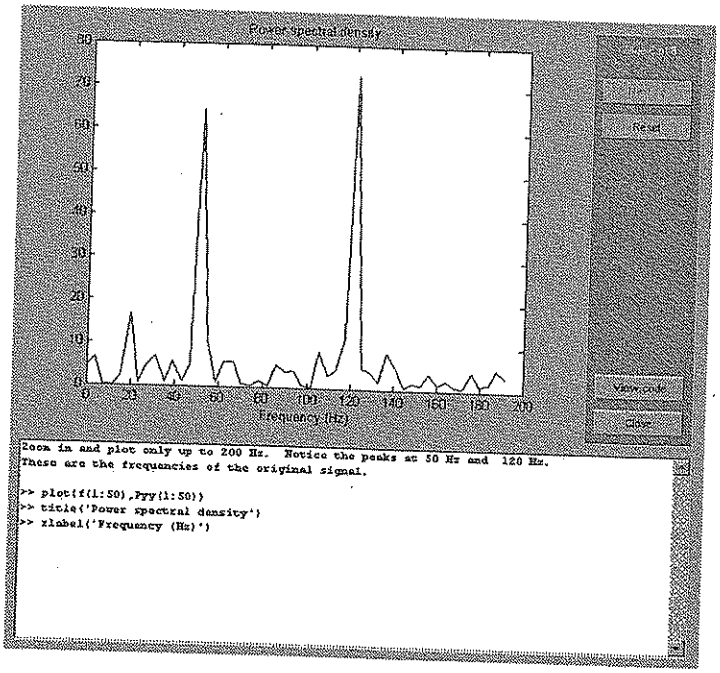
Currency exchange data
```
>> edit cxfft
>> cxfft
```

FFT Demos



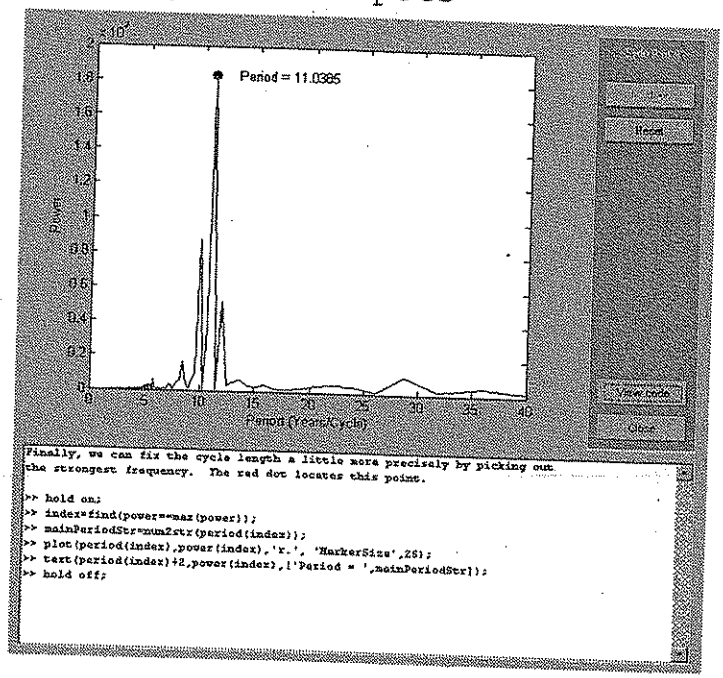
Several demos in the MATLAB documentation are related to using the FFT for spectral analysis.

```
>> playshow fftdemo
```



Step-by-step instructions for spectral analysis with the FFT

```
>> playshow sunspots
```



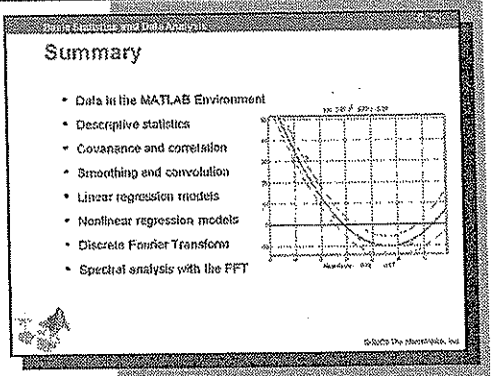
Analysis of variations in sunspot activity over 300 years

Try

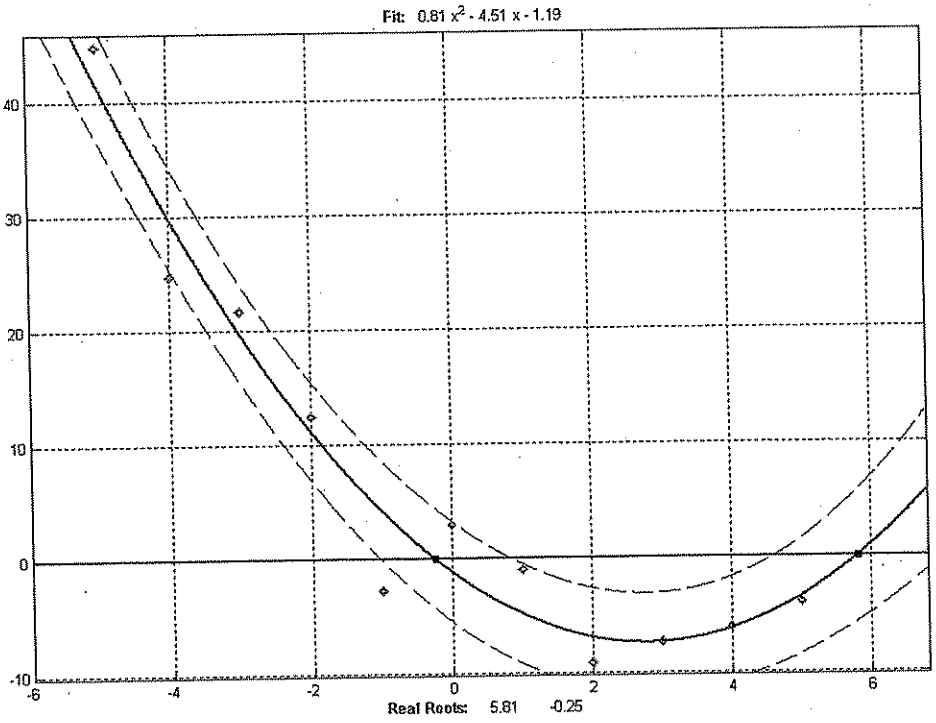
```
>> playshow fftdemo
```

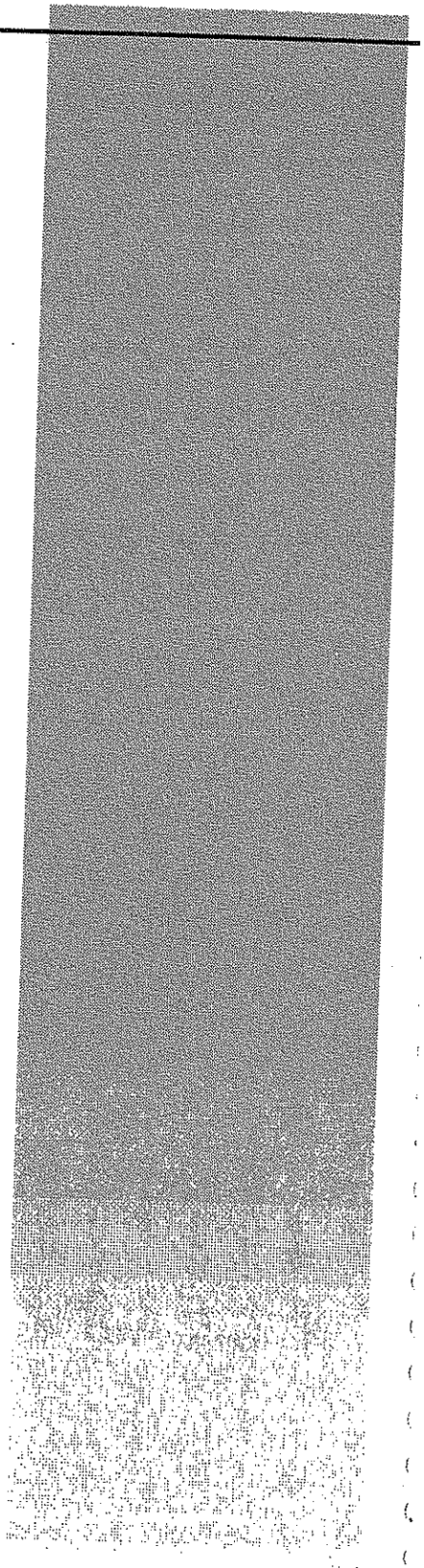
```
>> playshow sunspots
```

Summary



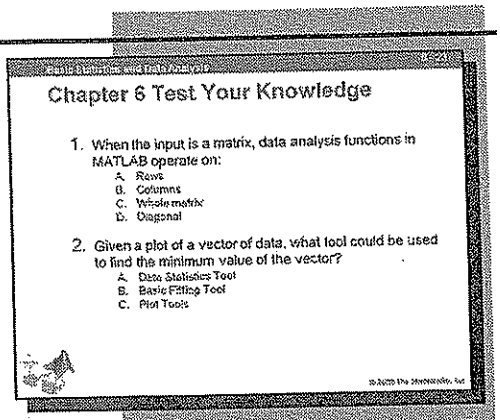
- Data in the MATLAB Environment
- Descriptive statistics
- Covariance and correlation
- Convolution and smoothing
- Linear regression models
- Nonlinear regression models
- Discrete Fourier transform
- Spectral analysis with the fast Fourier transform (FFT)





This page intentionally left blank

Chapter 6 Test Your Knowledge



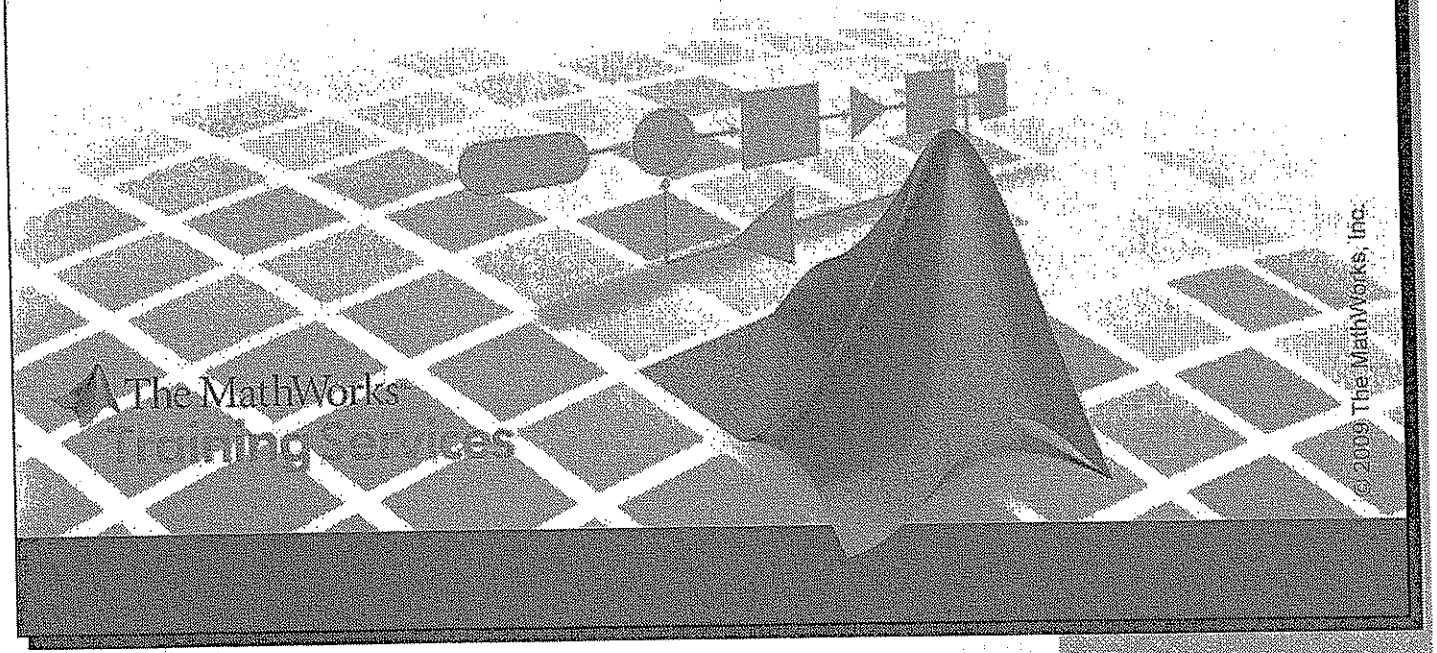
Name:_____

1. When the input is a matrix, data analysis functions in MATLAB operate on:
   A. Rows
   B. Columns
   C. Whole matrix
   D. Diagonal

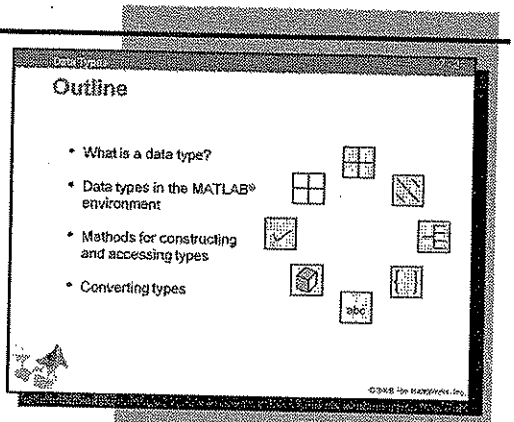
2. Given a plot of a vector of data, what tool could be used to find the minimum value of the vector?
   A. Data Statistics Tool
   B. Basic Fitting Tool
   C. Plot Tools

MATLAB® Fundamentals

Data Types

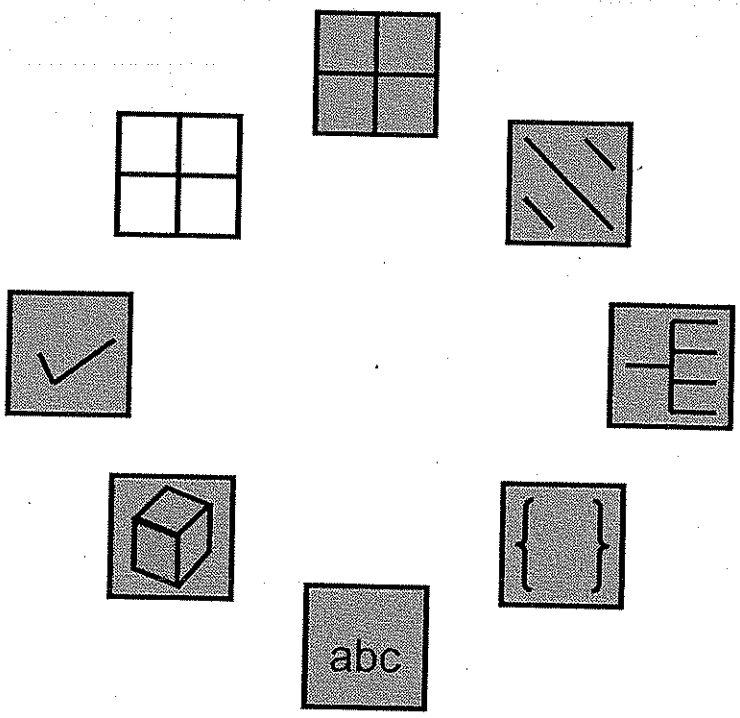


Outline



- What is a data type?
- Data types in the MATLAB® environment
- Methods for constructing and accessing types
- Converting types

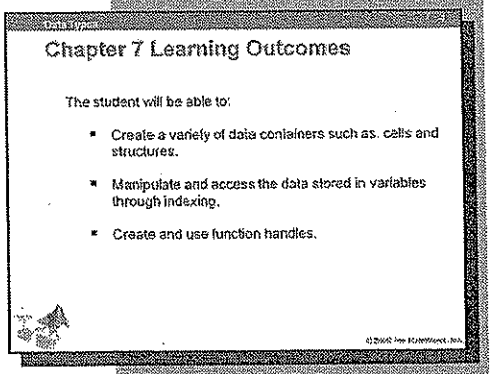
This chapter provides an overview of the different *types* of variables (data containers) you can create in MATLAB. Data types differ from one another in the kind of data they may contain and the way the data is organized. The chapter focuses on two basic operations associated with any data type: how to construct a new variable of that type and, after it is constructed, how to access and use the data it contains. The chapter also discusses methods for converting among data types.



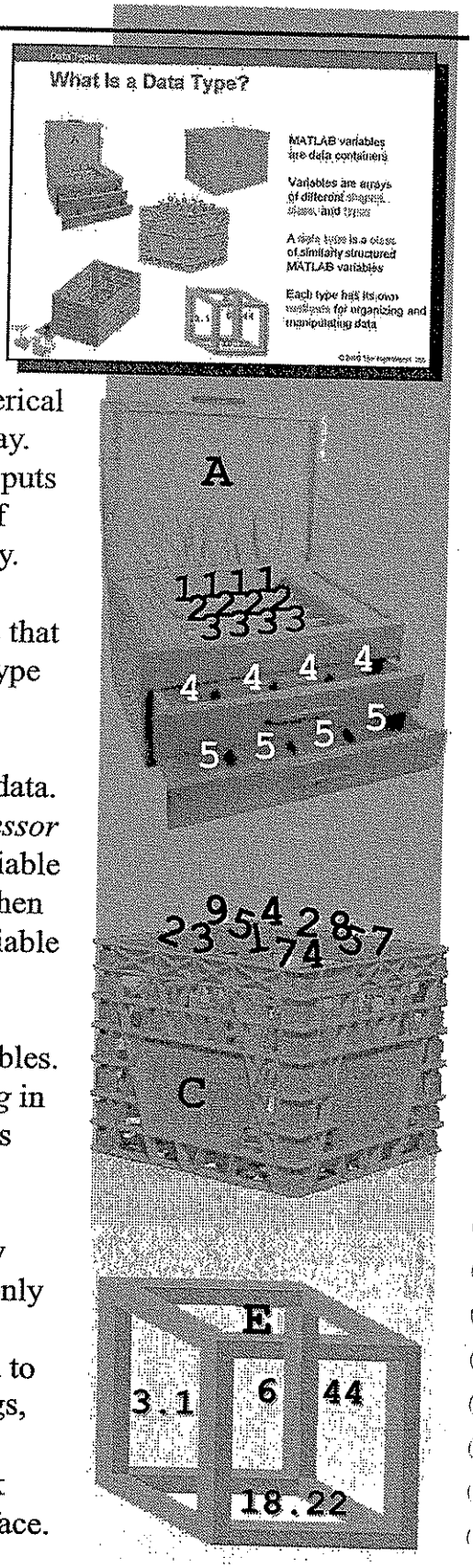
Chapter 7 Learning Outcomes

The student will be able to:

- Create a variety of data containers such as: cells and structures.
- Manipulate and access the data stored in variables through indexing.
- Create and use function handles.



What Is a Data Type?



MATLAB variables are data containers, constructed in sizes to accommodate your data. More fundamental than the size of a variable, however, is its *type*. Type determines the kind of data a variable may contain and the way the data is organized.

Often, MATLAB will chooses a type for you. If you import numerical data into the workspace, MATLAB typically creates a double array. If you type a character string at the command prompt, MATLAB puts it in a character array. If you ask MATLAB to import a number of character strings of different lengths, they are stored in a cell array.

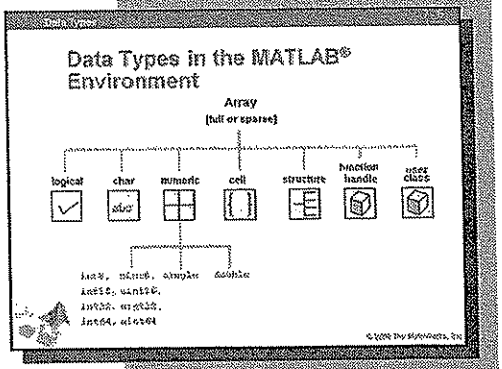
When you understand the characteristics of each of the data types that MATLAB supports, you can choose for yourself. Choose a data type that suits your data and the way you plan to work with it.

Every type has its own *methods* for organizing and manipulating data. The most basic methods are the *constructor methods* and the *accessor methods*. Constructor methods allow you to put your data in a variable of a specified type. Accessor methods allow you to extract data when you need it. *Converter methods*, which allow you to cast your variable in a new type, also play an important role in data organization.

Generally speaking, types are *classes* of similarly structured variables. Each variable is an *object* in a class. *Object-oriented programming* in MATLAB (not discussed here) allows you to create new data types with exactly the characteristics you want.

All supported types in MATLAB are some form of *array*. At a low level–the level at which MATLAB itself is programmed–there is only *one* type: the *MATLAB array*. All MATLAB variables are stored internally as MATLAB arrays. In C, MATLAB arrays are declared to be of type `mxArray`. This C structure contains, among other things, the MATLAB variable type, its dimensions, and its data. For most purposes, you do not need to know about these internals. You work with *different*, convenient types at the level of the MATLAB interface.

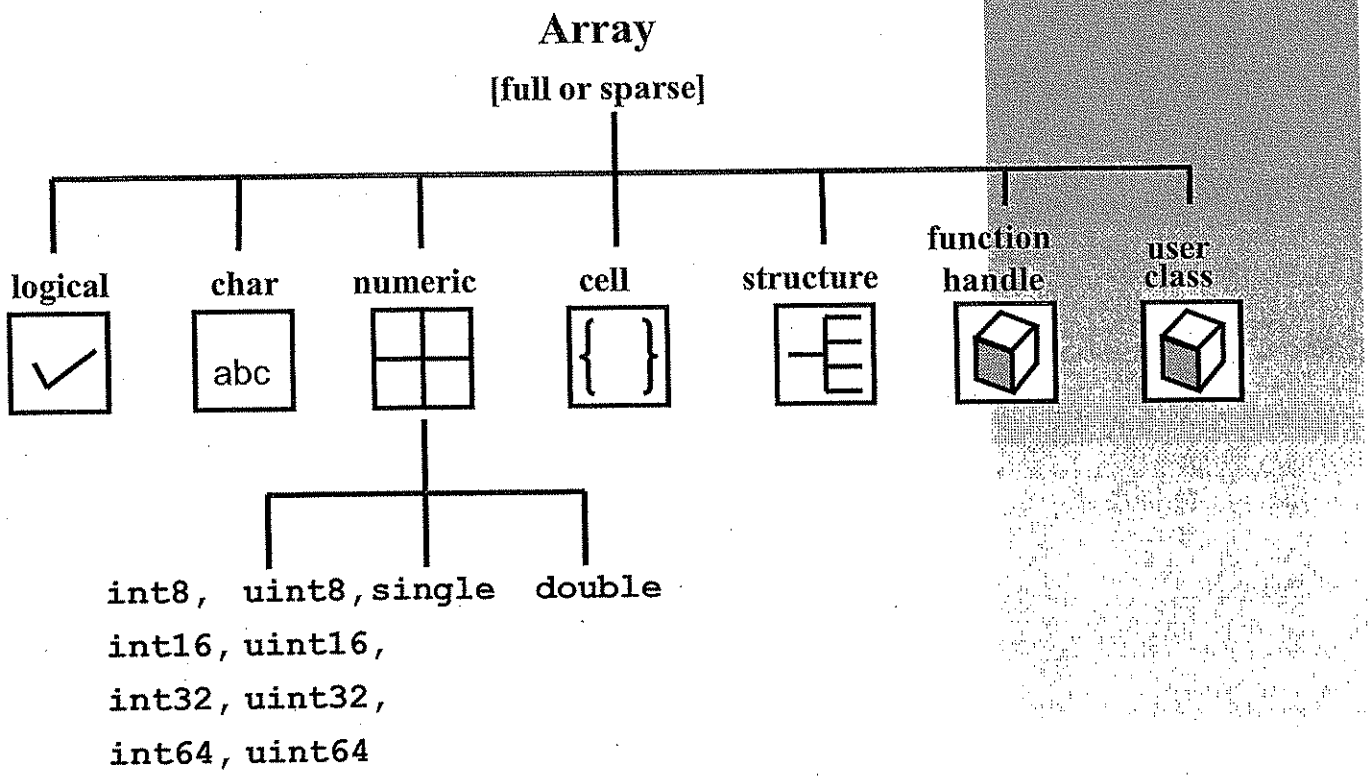
Data Types in the MATLAB® Environment



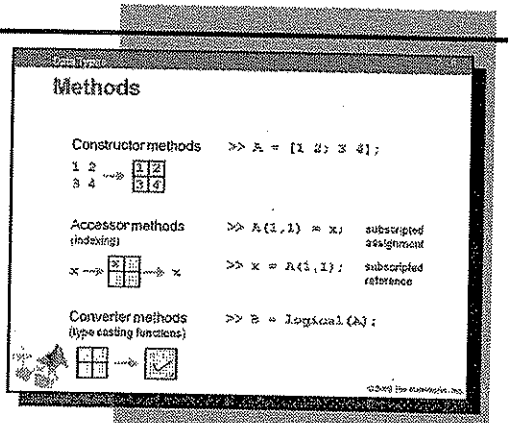
There are 15 fundamental data types in MATLAB. A variable of any type is an array with a characteristic organization of data.

Array sizes range from 0-by-0 (empty) to the maximum size your computer memory allows. MATLAB puts no limits on size.

All of the fundamental data types are shown following diagram. You can create additional data types as object-oriented classes.



Methods



Methods are operations designed specifically for a data type. You use them to construct new variables, access data contained in a variable, and convert variables from one type to another.

Constructor methods are the characteristic operations used to construct a variable of a given type. For example,

```
>> A = [1 2; 3 4];
```

assembles four numbers into a MATLAB double array, while

```
>> str = 'data';
```

assembles four characters into a MATLAB character array.

Accessor methods allow you to extract or alter the data in variables. These are the *indexing* methods in MATLAB. For example,

```
>> A(1,1) = x;      % Subscripted assignment
```

assigns the value `x` to the 1st row and 1st column of `A`, while

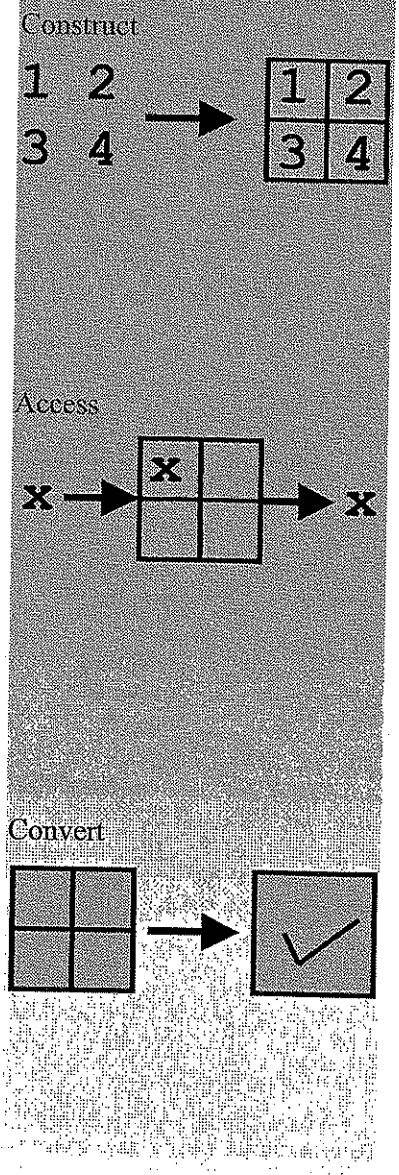
```
>> x = A(1,1);      % Subscripted reference
```

assigns the entry in the 1st row and 1st column of `A` to the variable `x`. The former use of indexing is called a *subscripted assignment*, while the latter is called a *subscripted reference*.

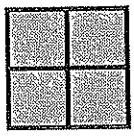
Converter methods (also known as *type casting functions*) take the data in a variable and reorganize it using the conventions of a different type. For example,

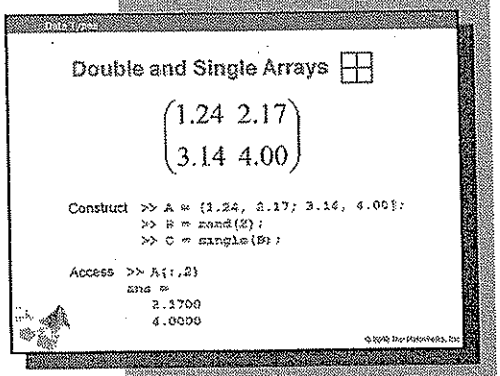
```
>> B = logical(A)
```

casts the nonzero elements of `A` as logical `1`s (`true`) and the zero elements of `A` as logical `0`s (`false`).



Double and Single Arrays





MATLAB stores floating-point numbers to double (64-bit) or single (32-bit) precision. Single-precision variables require less memory, but they have less accuracy and a smaller range.

The default type for numerical data in MATLAB is `double`.

```
>> B = single(A)
```

converts `A` to single precision, returning rounded (not truncated) data in `B`. Single precision is inefficient when creating large single arrays, because data must first be stored to double precision. It is more efficient to preallocate memory for single arrays using `zeros`, `ones`, or `eye`.

```
>> B = zeros(m,n,'single');
```

creates a placeholder single array of size `m` x `n` with elements all equal to 0. Data written to the variable is then stored as type `single`.

Complex variables are stored using separate vectors for the real and imaginary parts of the data and use twice the memory as real variables.

Constructor Methods

- Use matrix creation functions such as `rand`, `randn`, etc.
- Use the concatenation operator `[ ]`.
- Import numerical data using default settings.
- Convert from double to single-precision using `single`.
- Preallocate for single arrays with `zeros`, `ones`, or `eye`.

Accessor Methods

- Use parentheses `( )` following the variable name.
- Use a single index for linear indexing.
- Use two indices for row-column indexing.
- Use a logical array of the same size for logical indexing.

Try

Constructor methods

```
>> A = rand(2)
>> B = [1 2; 3 4]
>> C = ...
importdata(...
'numbers.dat')
>> D = single(C)
>> E = zeros(...
2,'single');
>> E(:) = ...
importdata(...
'numbers.dat')
>> whos
```

Accessor methods

```
>> A = rand(2)
>> x = A(3)
>> y = A(1,2)
>> z = A(A < 0.5)
>> A(2) = 0
>> A(3:4) = [5 6]
>> A(:,1) = []
```

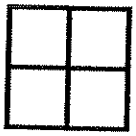
Real and complex storage

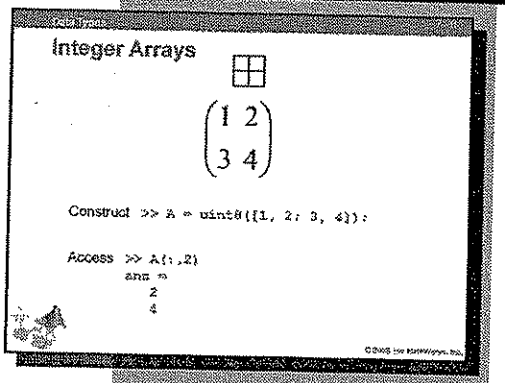
```
>> x = 1;
>> y = 1 + i;
>> z = 1 - j;
>> whos
```

Range of values

```
>> realmin('double')
>> realmax('double')
>> realmin('single')
>> realmax('single')
```

Integer Arrays





MATLAB has four signed (`int8`, `int16`, `int32`, `int64`) and four unsigned (`uint8`, `uint16`, `uint32`, `uint64`) integer data types. Signed types let you store negative as well as positive integers, but do not represent as wide a range of values as unsigned types (one bit flags a + or – sign). Unsigned types offer a wider range of values, but they must be nonnegative.

The four signed and unsigned types offer 8-bit, 16-bit, 32-bit, and 64-bit storage for integer data. Using the smallest integer type that accommodates your data saves both memory and execution time for your programs. For example, `uint8` is sufficient for nonnegative integer data in the range from 0 to 255 ($2^8 - 1$), while `int8` is sufficient for integer data in the range from –128 (-2^7) to 127 ($2^7 - 1$).

```
>> B = int8(A)
```

converts numeric type `A` to `int8` precision, returning rounded (not truncated) data in `B`. Conversion functions for other integer types are given by type name. As with single arrays, memory for integer types can be preallocated using `zeros`, `ones`, or `eye`. For example,

```
>> B = zeros(m,n,'int8');
```

creates a placeholder `int8` variable of size `m` x `n` with elements all equal to 0. Data written to the variable is then stored as `int8`.

Constructor Methods

- Convert from other numeric types using `int8`, `uint8`, etc.
- Preallocate for integer variables with `zeros`, `ones`, or `eye`.

Accessor Methods

- Use the same methods as for double arrays.

There's more!

Calculations involving single and integer arrays follow specific rules.

See Appendix D-5

Try

Constructor methods

```
>> A = 10*rand(2)
>> B = uint8(A)
>> C = zeros(...
2,'uint8');
>> C(:) = ...
importdata(...
'numbers.dat')
>> whos
```

Accessor methods

```
>> A = uint8(...
10*rand(2))
>> x = A(3)
>> y = A(1,2)
>> z = A(A < 5)
>> A(2) = 0
>> A(3:4) = [5 6]
>> A(:,1) = []
```

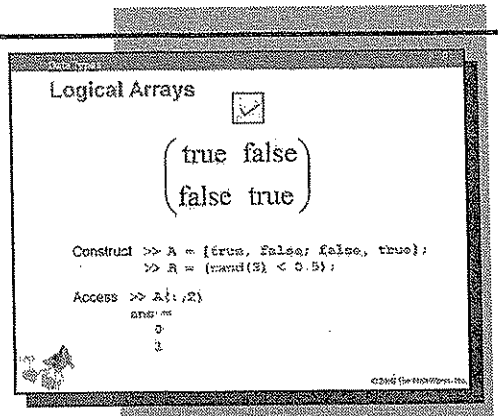
Range of values

```
>> intmin('uint8')
>> intmax('uint8')
>> intmin('int8')
>> intmax('int8')
```

Digital communication model

```
>> edit integerphone
>> integerphone
```

Logical Arrays



Logical arrays store logical (or *Boolean* or *truth*) values that result from evaluating logical expressions in MATLAB. Logical values are either `true` (logical `1`) or `false` (logical `0`).

For example,

```
>> x = 2;
>> Ix = (x <= 10) & (mod(x,2) == 0);
```

constructs the `1`-by-`1` logical array `Ix` and assigns a logical `1` to it.

Evaluation of logical expressions is vectorized in MATLAB, so logical values can be assigned to an entire array with a single expression.

```
>> A = magic(3);
>> IA = (A > 6) | (A == 2)
IA =
     1     0     0
     0     0     1
     0     1     1
```

constructs a logical array `IA` the same size as `A`.

Logical arrays are used to introduce branching in programs and in *logical indexing*, an accessor method that allows you to index (perhaps noncontiguous) elements in an array that satisfy a logical condition.

Constructor Methods

- Evaluation of logical expressions
- Convert from numeric types using `logical`.
- Use the concatenation operator `[ ]` with `true` and `false` elements.

Accessor Methods

- Use the same methods as for double arrays.
- Logical arrays are often used to access *other* arrays.

Try

Constructor methods

```
>> x = 2;
>> Ix = ...
(x <= 10) & ...
(mod(x,2) == 0)
>> A = magic(3)
>> IA = ...
(A > 6) | (A == 2)
>> B = pascal(3) - 1
>> J = logical(B)
>> K = [true;false]
```

Accessor methods

```
>> I = logical(...
[1 0;2 0;3 4;0 5])
>> x = I(6)
>> y = I(:,1)
>> J = sum(I(:) == 1)
```

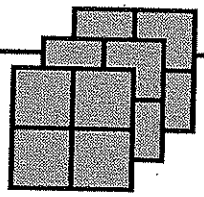
Accessing other arrays (logical indexing)

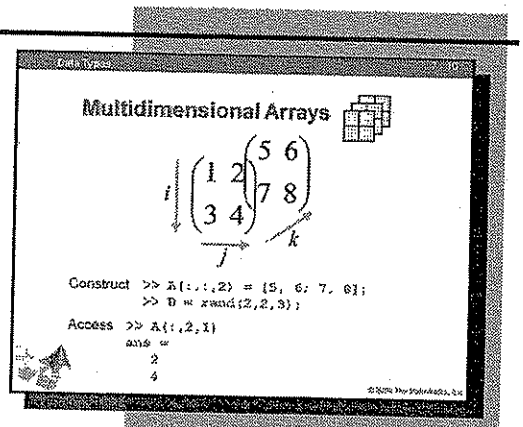
```
>> A = magic(3)
>> IA = ...
(A > 6) | (A == 2)
>> x = A(IA)
>> A(~IA) = 0
>> A(eye(3))
>> A(logical(eye(3)))
```

Programming

```
>> x = 0; y = 0;
>> while x < 3
x = x + 1; y = [y,x]
end
```

Multidimensional Arrays





Multidimensional arrays are not a separate type in MATLAB, they simply arrange data along more than two linear dimensions. You create and index into multidimensional arrays exactly as you do for 2-D arrays–you just use more indices (or, for logical indexing, logical arrays of commensurate dimension).

MATLAB puts no limit on the number of dimensions you can use to organize your data. If your data are observations of a response to n discrete predictor variables, it makes sense to organize your data into an n-D array. This way, accessing the data becomes much more natural.

For example, if

```
>> R = rand(100,100);
>> G = rand(100,100);
>> B = rand(100,100);
```

represents the 2-D (in this case random) planes in an RGB image, then

```
>> I = cat(3,R,G,B);
```

generalizes the MATLAB concatenation operator `[ ]` to higher dimensions, and assembles the planes into a 3-D array. To access the blue value in the first row and second column of `I`, you use

```
>> B12 = I(1,2,3);
```

Constructor Methods

- Preallocate (with `0` values) by assigning a value to the corner of the array opposite the origin.
- Preallocate with `zeros` or `ones`.
- Use matrix creation functions such as `rand`, `randn`, etc.
- Use the concatenation operator `cat` to assemble parts of an array.

Accessor Methods

- Use the same methods as for double arrays, with more indices.

Try

Constructor methods

```
>> A(2,2) = 1
>> B(:,:,2) = A
>> C(:,:,:,2) = B
>> D = zeros(2,2,2)
>> E = rand(2,2,2)
>> R = rand(100,100);
>> G = rand(100,100);
>> B = rand(100,100);
>> I = cat(3,R,G,B);
>> image(I)
```

Accessor methods

```
>> I = uint8(...
256*rand(100,100,3));
>> B12 = I(1,2,3)
>> B = I(:,:,3);
```

Locate dark pixels

```
>> [R,C] = find(...
(I(:,:,1) < 25) & ...
(I(:,:,2) < 25) & ...
(I(:,:,3) < 25));
>> image(I)
```

Zoom in on the image to see the pixels at `[R,C]`.

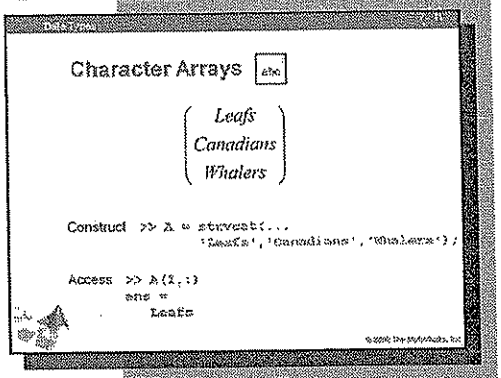
RGB image color planes

```
>> edit import2
>> import2
```

Space-filling curve

```
>> edit hilbertcube
>> [I,J,K,H] =
hilbertcube(3);
```

Character Arrays



Character arrays store nonnumeric characters and strings.

Each character has a unique numerical identifier. The `double` command converts characters into numerical values, and the `char` command converts them back. The standard, printable ASCII characters fall in the range `32:127`. Display them by typing

```
>> char(reshape(32:127,32,3)')
```

Characters in the range `0:31` are nonprinting characters, many of them (such as bells and form feeds) from the days of the teletype machine. Characters in the range `128:255` form an extended ASCII character set, including symbols and international characters. There are several variations of this extended set in use.

Character strings are delimited by single quotes (`' '`). Concatenate character strings to form rectangular character arrays with the same concatenation operator (`[]`) used for numerical data. Access character data with the same indexing methods used for `double` arrays.

A useful MATLAB string-handling function is `strvcat`.

```
A = strvcat(s1,s2,s3,...)
```

forms the character array `A` with the strings `s1`, `s2`, `s3`, ... as rows. If the strings are of different lengths, the function appends spaces to the shorter strings to form a valid, rectangular MATLAB array.

Constructor Methods

- Use `char` to convert positive integers from 0 to 65535 to characters.
- Use string delimiters `' '` to concatenate characters.
- Use the operator `[ ]` to concatenate characters and strings.
- Use string functions such as `strvcat`.

Accessor Methods

- Use the same methods as for double arrays.

Try

Constructor methods

```
>> s1 = 'foo'
>> s2 = 'bar'
>> s3 = 'bark'
>> A = [s1 s2]
>> B = [s1;s2]
>> C = [s1;s3]
>> C = strvcat(s1,s3)
```

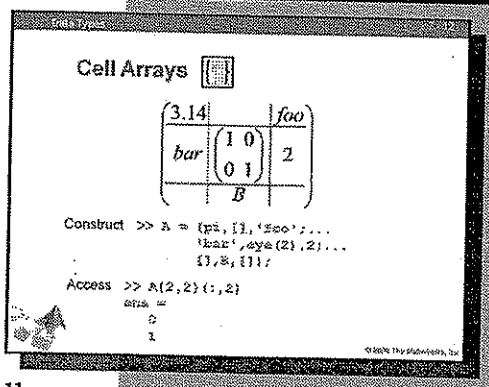
Accessor methods

```
>> A = char(...
reshape(...
32:127,32,3)')
>> A(2,2:27)
>> A((A > 64) & ...
(A < 91))
```

ASCII character codes

```
>> nums = (32:127)';
>> NUMS = ...
num2str(nums);
>> SEP = repmat(...
' ',length(nums),1);
>> CHARS = ...
char(nums);
>> dictionary = ...
[NUMS SEP CHARS]
>> char(63)
>> double('?')
```

Cell Arrays



Cell arrays are an easy way to assemble data of dissimilar types and sizes into a single "container of containers." Technically, the cells in a cell array are pointers to locations in memory where each cell element is stored separately as a particular type. The number of elements in a cell array is the number of cells *plus* the number of elements in the variables referenced by the cells.

You construct and access cell arrays much like double arrays, but you use braces ({ }) instead of brackets ([ ]) to concatenate elements, and braces ({ }) instead of parentheses (( )) to index and access elements. Use empty brackets ([ ]) to indicate an empty cell.

Constructor Methods

- Use the `cell` function to preallocate an array of empty cells.
- Use braces { } to concatenate arbitrary MATLAB variables.
- Convert from numeric types using `num2cell`.

Accessor Methods

- Use braces { } following the variable name.
- Use a single index for linear indexing.
- Use two indices for row-column indexing.
- Use a logical array of the same size for logical indexing.

$$\left(\begin{array}{c|c|c} 3.14 & & foo \\ \hline bar & \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} & 2 \\ \hline & B & \end{array}\right)$$

Try

Constructor methods

```
>> A = cell(2,3)
>> A{1,1} = pi
>> A{1,3} = 'foo'
>> A{2,2} = rand(2)
>> whos
>> B = {0,[];'A',A}
>> whos
>> C = num2cell(...
magic(3))
>> whos
```

Note the number of elements

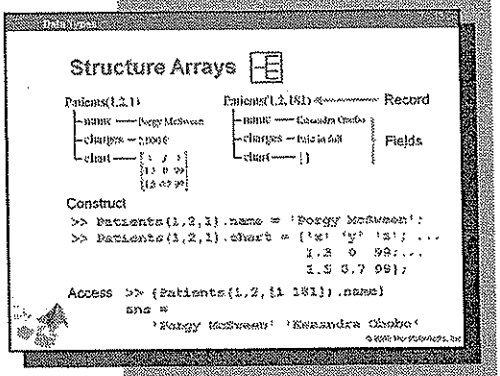
Accessor methods

```
>> A = {eye(2);'moo'}
>> I = A{1}
>> ghost = ...
['b',A{2,1}(2:end)]
```

Importing strings

```
>> edit teams.dat
>> A = importdata(...
'teams.dat')
>> B = strvcat(...
A{1},A{2},A{3})
```

Structure Arrays



Structure arrays, like cell arrays, are arrays of pointers used to assemble data of dissimilar types and sizes into one variable. Structure arrays, however, have a distinct organization and syntax with origins in database programming. Think of structure arrays as miniature databases.

Data in a structure array is organized into (perhaps multidimensional) *records*. Each record consists of various *fields* of information. The data stored in a particular field can be a MATLAB variable of any type.

For example, two hospital records might be organized as follows:

Patients(1,2,1)

- name — Porgy McSween
- charges — 2,900 €
- chart —

$$\begin{bmatrix} x & y & z \\ 1.3 & 0 & 99 \\ 1.5 & 0.7 & 99 \end{bmatrix}$$

Patients(1,2,181)

- name — Kesandra Obobo
- charges — Paid in full
- chart — [ ]

The mulidimensional *record number* (1,2,1) might indicate the first patient in the second hospital in the first city. The *field names* "name," "charges," etc. might indicate relevant (or perhaps empty) data.

The following syntax is used for both assignments and references:

```
<variable name>(<record number>).<field name>
```

Record numbers are not required for arrays with a single record.

Constructor Methods

- Use the `struct` function to preallocate for a structure array.
- Use the syntax above.

Accessor Methods

- Use the syntax above.

Try

Constructor methods

```
>> A.name = ...
'McSween'
>> A.chart = ...
{'x' 'y'; 1 2}
>> A(2).name = ...
'Obobo'
>> A(2).chart = {}
```

Accessor methods

```
>> load gradebook
```

Examine G in the Array Editor

```
>> G(1,1).grades
>> G(1,:).grades
>> {G(1,:).grades}
>> cat(...
2,G(1,:).grades)
```

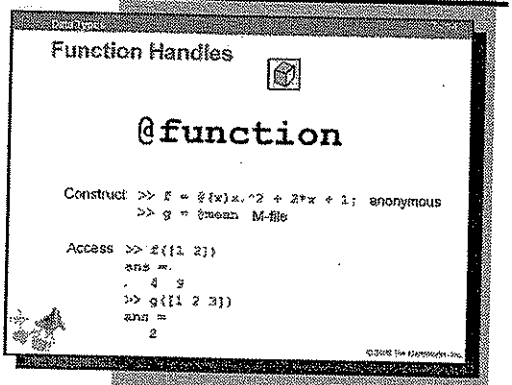
Dynamic field naming

```
>> edit getG
>> getG(1,2,'name')
>> getG(1,2,'grades')
>> getG(1,2,'info')
```

Encapsulating diverse data

```
>> load cx
>> CX.data = Data;
>> CX.info = ...
Description;
>> CX.date = date;
>> CX
```

Function Handles



Function handles store data on MATLAB functions. For M-file functions, function handles store the absolute path to the file. For *anonymous functions*, defined at the command prompt, function handles store the functional rule. Function handles are especially useful for passing functional arguments to MATLAB "function functions" (operators) such as integrators, ODE solvers, and optimizers.

To create a function handle to an M-file function, use the syntax

```
>> h = @<filename>;
```

To create a function handle to an anonymous (no M-file) function defined at the command prompt, use syntax like the following:

```
>> h = @(x1,x2)(2*x1.^2 - 3*x2.^2);
```

There are a number of advantages to working with function handles rather than directly with function M-files. One advantage is efficiency of function calls, due to the elimination of path look-up (because the absolute path is stored in the handle). Another advantage is robustness of code with respect to path changes (because a handle remembers where a file is even if it is removed from the MATLAB path).

Constructor Methods

- Use the `@` operator before the name of a function M-file on the path.
- Use the `@` operator anonymously, followed by a comma-separated variable list in parentheses, and then a functional rule.

Accessor Methods

- Evaluate the function handle `h` using functional notation like `h(x)`.
- Use function handles as arguments in MATLAB function functions and the functional rule is referenced as needed.

Try

Constructor methods

```
>> edit humps
>> h1 = @humps;
>> h2 = @(x)(x.^2);
>> h3 = @(x1,x2)...
(2*x1.^2 - 3*x2.^2);
```

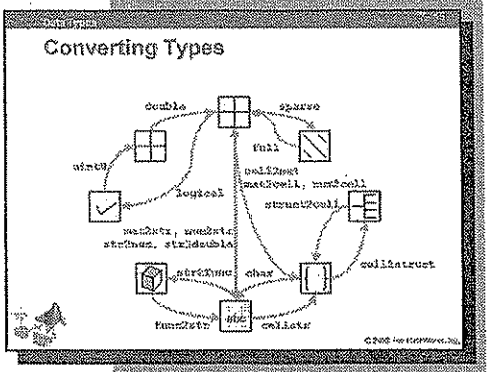
Accessor methods

```
>> h1 = @humps;
>> h1([.3 .9])
>> fplot(h1,[0 1])
>> h2 = @(x)(x.^2);
>> h2(1:5)
>> h3 = @(x1,x2)...
(2*x1.^2 - 3*x2.^2);
>> h3(2,2)
```

Functions with parameters

```
>> a = 5; b = 5;
>> h = @(x1,x2)...
a*x1.^2 - b*x2.^2;
>>[X,Y] = ...
meshgrid(...
-1:0.1:1,-1:0.1:1);
>> Z = h(X,Y);
>> surf(X,Y,Z)
>> a = 5; b = -5;
>> surf(X,Y,Z)
>> h = @(x1,x2)...
a*x1.^2 - b*x2.^2;
>> Z = h(X,Y);
>> surf(X,Y,Z)
```

Converting Types



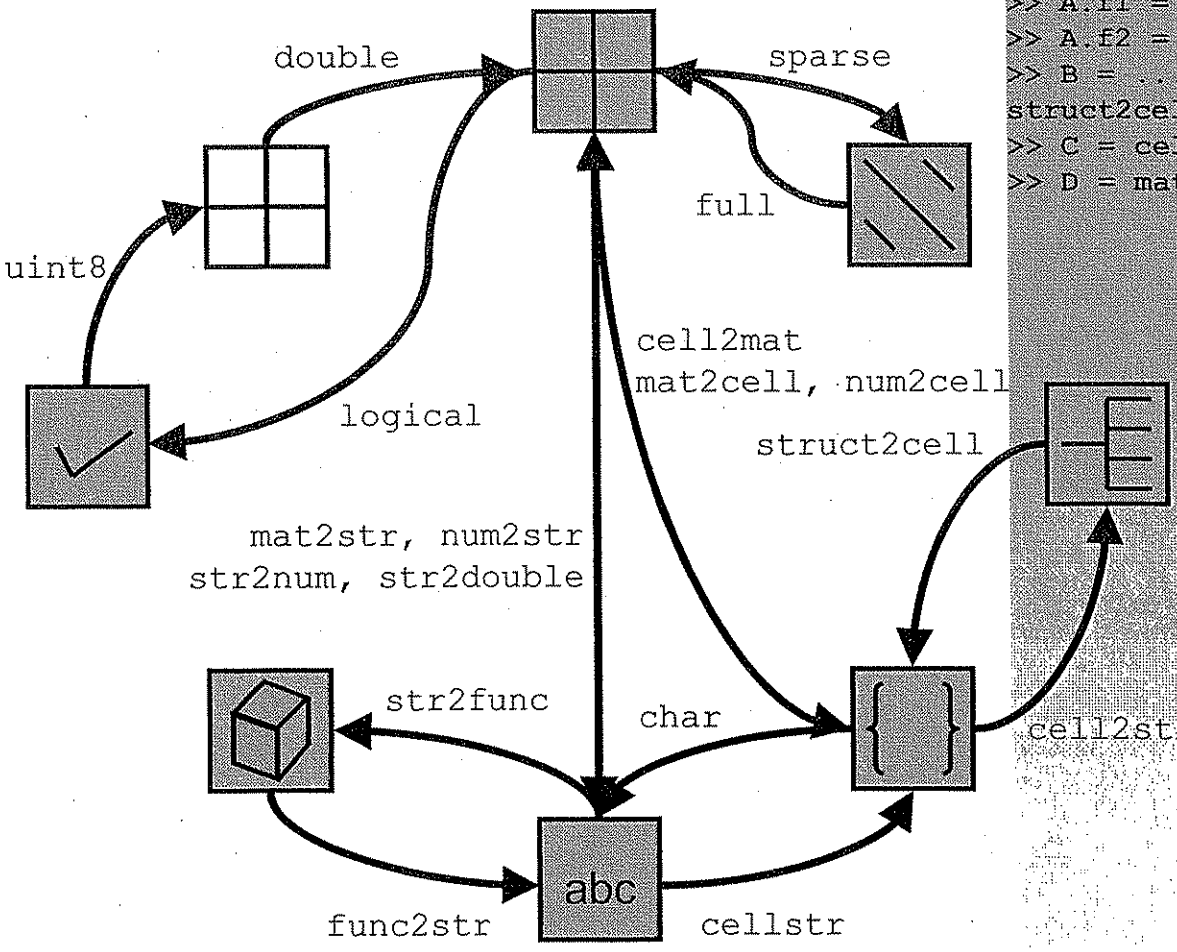
The following diagram indicates some of the converter methods (type casting functions) for MATLAB data types. For a full list, with links to relevant documentation, type

```
>> doc
```

and use the Help Navigator to navigate to **MATLAB → Functions – Categorical List → Programming and Data Types → Data Types → Data Type Conversion**.

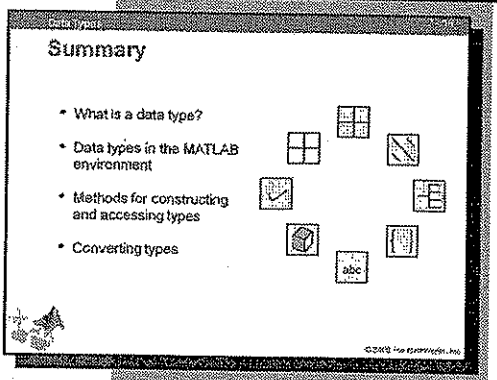
Try

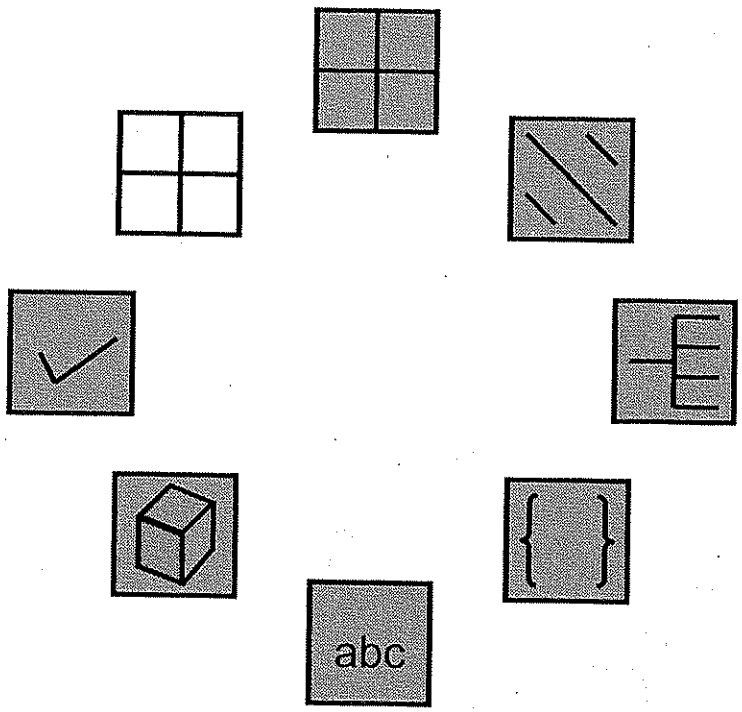
```
>> help datatypes

>> A.f1 = [1 2; 3 4]
>> A.f2 = rand(2)
>> B = ...
struct2cell(A)
>> C = cell2mat(B)
>> D = mat2str(C)
```



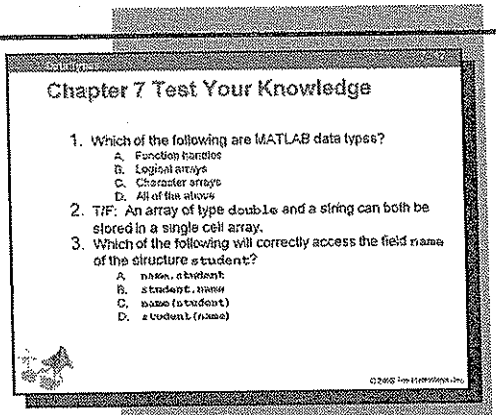
Summary

- What is a data type?
- Data types in the MATLAB environment
- Methods for constructing and accessing types
- Converting types





Chapter 7 Test Your Knowledge



Name:_____

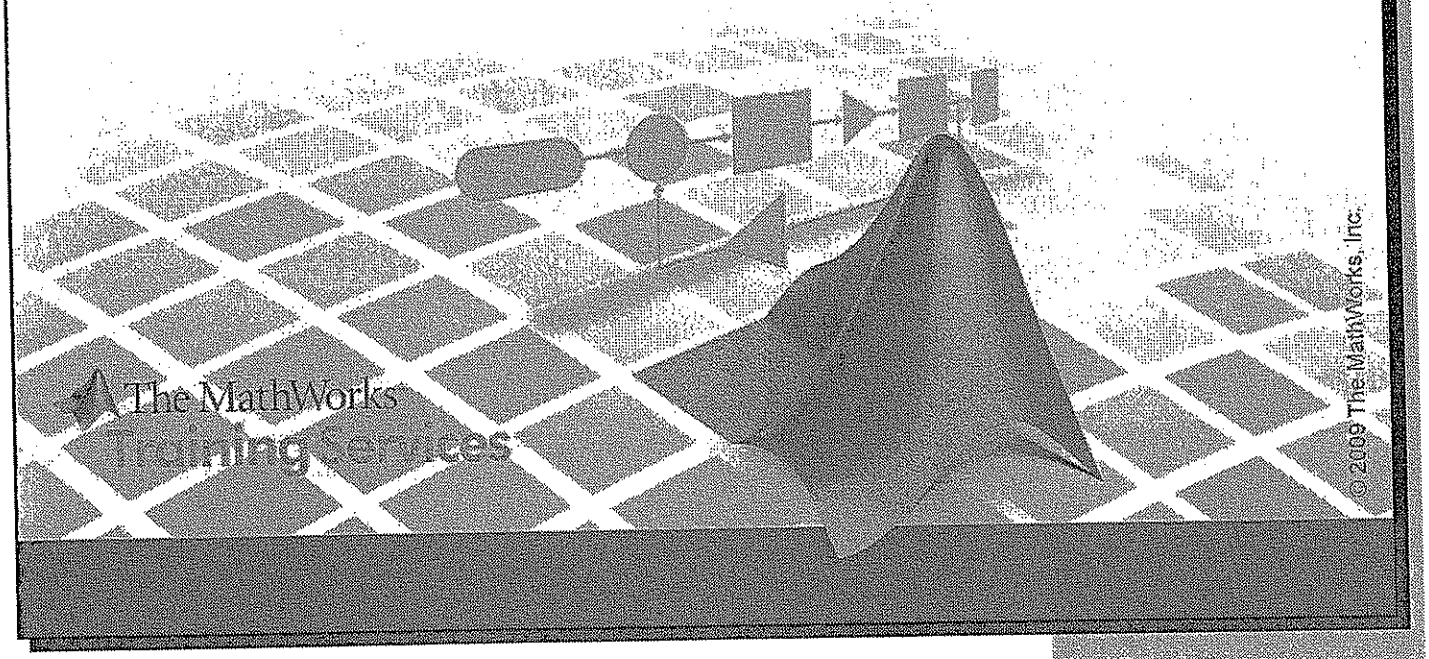
1. Which of the following are MATLAB data types?
   A. Function handles
   B. Logical arrays
   C. Character arrays
   D. All of the above

2. T/F: An array of type `double` and a string can both be stored in a single cell array.

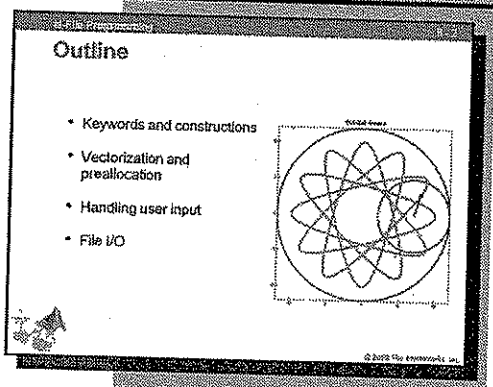
3. Which of the following will correctly access the field `name` of the structure `student`?
   A. `name.student`
   B. `student.name`
   C. `name(student)`
   D. `student(name)`

MATLAB® Fundamentals

M-File Programming

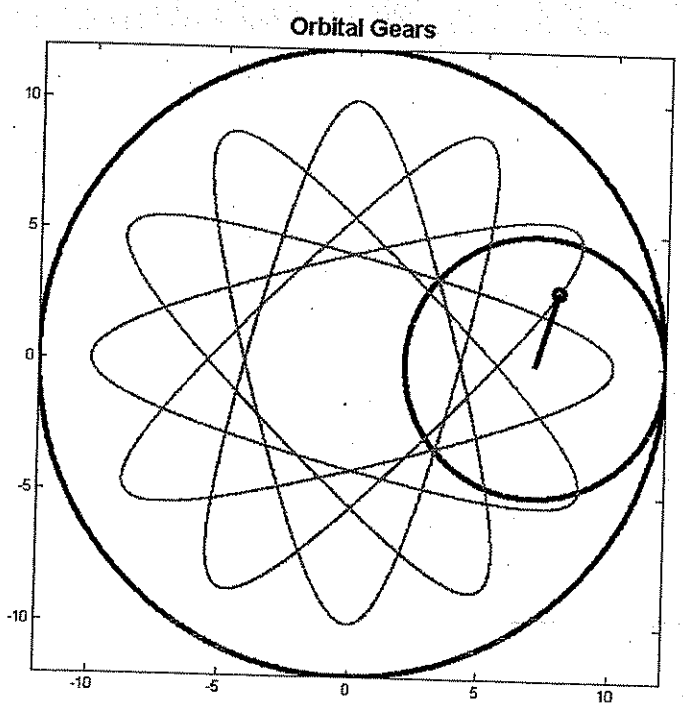


Outline



- Keywords and constructions
- Vectorization and preallocation
- Handling user input
- File I/O

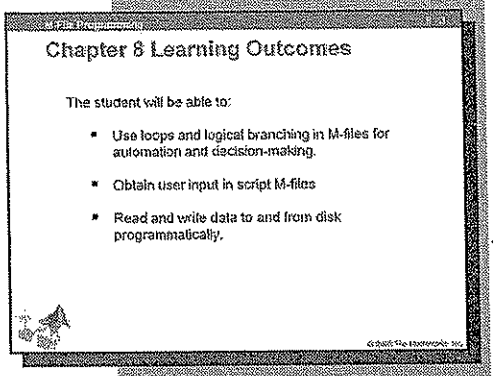
MATLAB® is a language. You speak the language through programs. Whether you type in a single line of code at the command prompt or assemble multiple M-files into a sophisticated application, you are programming in the M language. This chapter highlights programming constructs that allow loops, conditional branching, user interactivity, and data import/export.



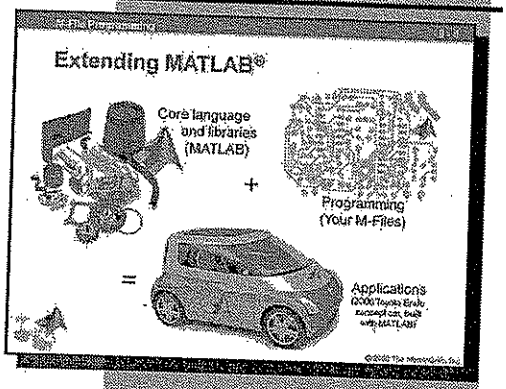
Chapter 8 Learning Outcomes

The student will be able to:

- Use loops and logical branching in M-files for automation and decision-making.
- Obtain user input in script M-files.
- Read and write data to and from disk programmatically.



Extending MATLAB®



At a basic level, MATLAB consists of a core programming language and an extensive set of function libraries. These core components are presented to you through an interface that integrates the tools and documentation that you need to assemble sophisticated applications.

The MATLAB libraries contain thousands of high-level functions that carry out common numerical and graphical tasks, accurately and efficiently. You do not have to write and test these functions yourself – they are ready to use.

MATLAB library functions, however, are meant only as building blocks for larger applications. They do not attempt to anticipate every user preference. They are designed to solve general problems. As such, it is rare that any one function will solve your specific problem.

To make full use of MATLAB, you have to *program*. MATLAB gives you the tools, and then respects your ability to build what you need.

When you build your own MATLAB functions, you save them as M-files and then call them by name:

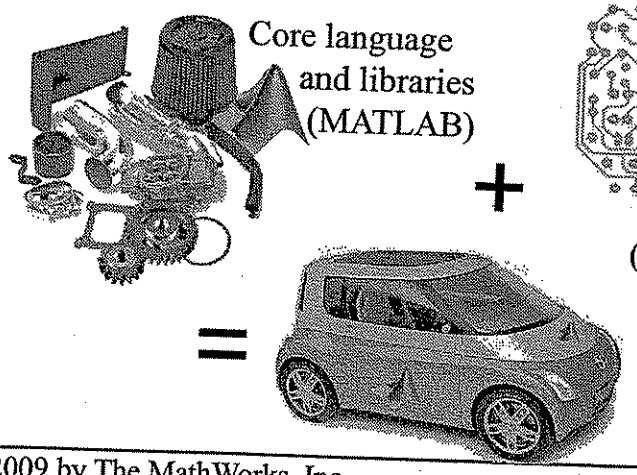
```
>> myfunction(a,b,c)
```

This is the same way you call MATLAB library functions. Think of your M-files as *new* commands–*extensions* of MATLAB that add the features you require to solve your problems and build your applications.

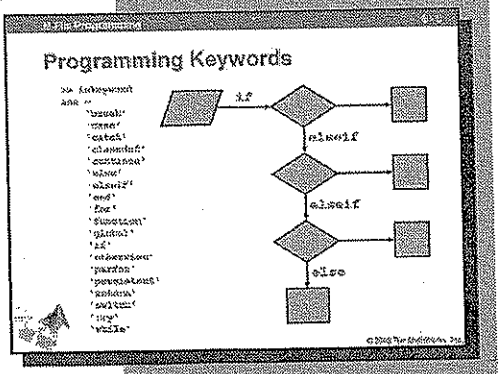
Try

New functions for solving systems of linear equations

```
>> which ...
mldivide -all
>> edit mldivide
>> edit mydivide

>> A = rand(2,2);
>> b = rand(2,1);
>> A\b
>> mydivide

>> A = rand(3,2);
>> b = rand(3,1);
>> A\b
>> mydivide

>> A = rand(2,3);
>> b = rand(2,1);
>> A\b
>> mydivide
```



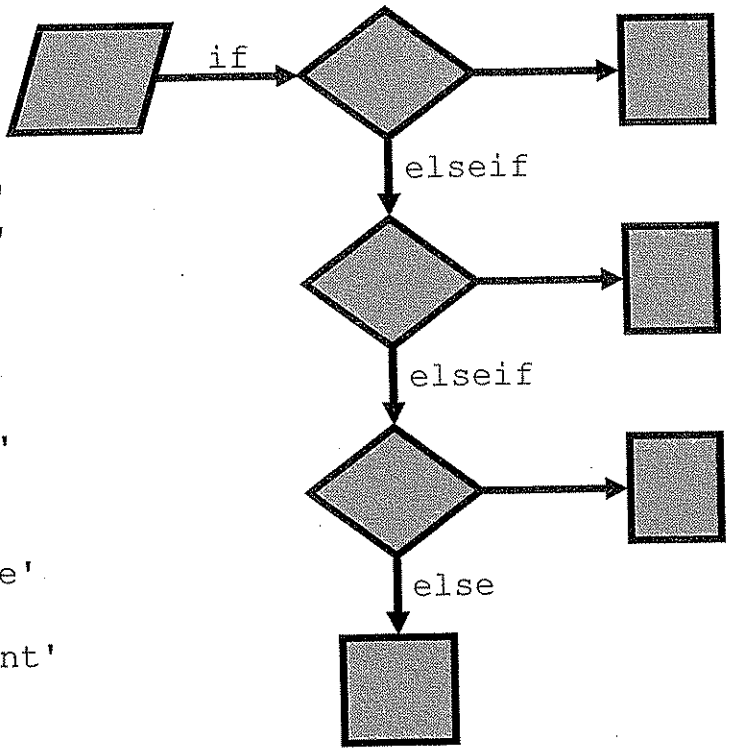
Programming Keywords



Keywords are the basic programming components of the MATLAB language. They are reserved words with special meanings. The MATLAB Editor displays them as blue, and they are recognized as directions for control flow in your programs.

To see a list of MATLAB programming keywords, type

```
>> iskeyword
ans =
    'break'
    'case'
    'catch'
    'classdef'
    'continue'
    'else'
    'elseif'
    'end'
    'for'
    'function'
    'global'
    'if'
    'otherwise'
    'parfor'
    'persistent'
    'return'
    'switch'
    'try'
    'while'
```



To read about the use of a particular keyword, and see examples, type

```
>> doc <keyword>
```

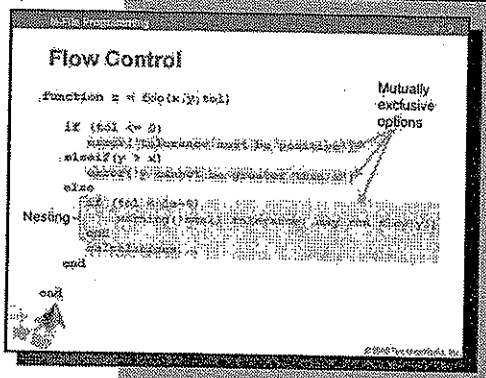
Try

List of keywords
```
>> iskeyword
```

Keyword example:
break
```
>> doc break
>> edit mean
>> edit breakdemo
>> breakdemo
```

Keyword example:
continue
```
>> doc continue
>> edit magic
>> edit continuedemo
>> continuedemo
```

Flow Control



MATLAB keywords and functions are assembled into code blocks called *programming constructions*. Two common, useful types are *loops* and *flow control* constructions. Flow control constructions allow you to introduce conditional branching into your programs, so that execution follows one path or another, depending on certain checks that are made along the way.

Mutually exclusive branching based on the values of arbitrary logical conditions is achieved using the `if-elseif-else` construction:

```
if (logical test 1)
    statements 1
elseif (logical test 2)
    statements 2
else
    statements 3
end
```

Only one block of code (`statements n`) is executed . The logical tests are performed in order and only as necessary. Hence, if `logical test 1` evaluates as true, `statements 1` is executed and `logical test 2` is never performed. There can be any number of `elseif` statements, including none. There can be at most one `else` statement (and it must be the last condition).

If there is a finite set of discrete possible values for a variable (e.g., a set of menu options or the number of dimensions of an array), you can use the `switch-case` construction in place of an `if-elseif-else`:

```
switch variable
    case value 1        % executes if
        statements 1    % variable == value 1
    case value 2
        statements 2
    otherwise           % "else"
        statements 3
end
```

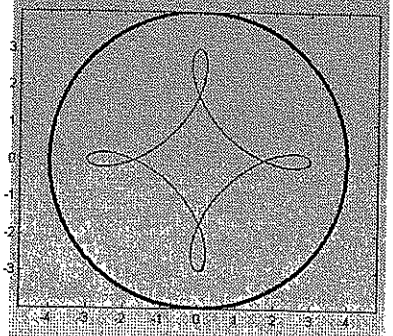
Try

Basic examples

```
>> edit ifelse
>> ifelse

>> edit switchcase
>> switchcase
```

Orbital gears (`switch-case`)

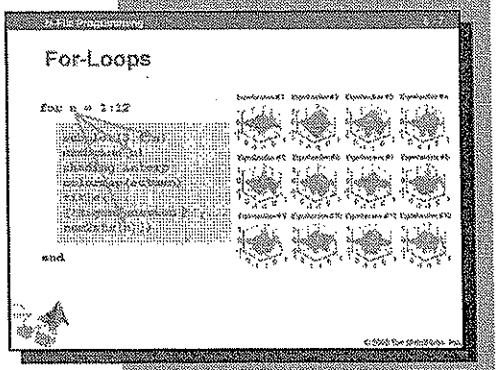
```
>> edit roulette
>> roulette(...
4,3,2,3,'hyp')
```



Systems of linear equations (nested `if-else`)

```
>> edit mydivide
>> A = rand(2,2);
>> b = rand(2,1);
>> mydivide
```

For-Loops



Loops execute a block of code repeatedly within a larger program. A `for`-loop executes code for a fixed sequence of values of a parameter within the loop, one time for each value, in order. The parameter is known at the *loop index* and is a regular MATLAB variable.

```
for index = first:increment:last
    statements
end
```

Note the use of the colon syntax to define the values that the index will take. Although this notation creates a vector, *`index`* will be a scalar, taking each value of the vector in turn. A vector variable in memory can be used in place of the range declaration:

```
v = [1,2,3,5,8,13];
for k = v
    statements
end
```

As a MATLAB variable, the index can be used in calculations within the loop; it also persists in memory (with the value of *`last`*) after the loop terminates. It is even possible to redefine *`index`* within the loop, although this is strongly discouraged, as it leads to confusing code.

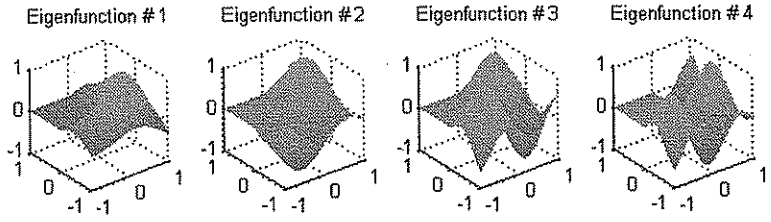
As a vectorized language, many MATLAB commands contain implicit loops. Judicious use of vectorized MATLAB functions can enable you to write clean code that uses a single line of code whether other languages would require a `for`-loop.

Try

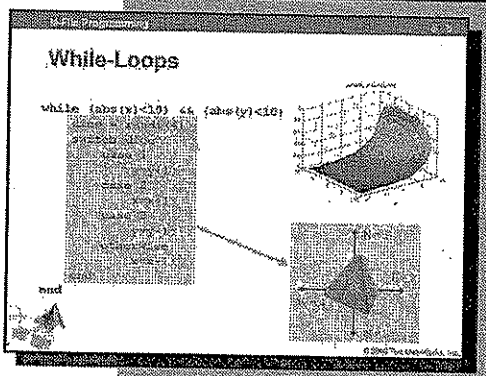
```
>> edit forloop
>> forloop
```

Implied loops

```
>> edit loops_noloops
>> loops_noloops
```



While-Loops



To use a `for`-loop, it is necessary to know in advance how many iterations of the loop are required. In many cases, however, you may want to execute a block of code repeatedly until a result is achieved or a condition is violated. For this, you can use a `while`-loop:

```
while condition
    statements
end
```

The code contained in *`statements`* will be evaluated as long as the logical *`condition`* evaluates as true. Note that *`condition`* is evaluated before the *`statements`* block runs (i.e., at the beginning of the loop), which may require some initialization prior to the `while` statement. Furthermore, once the *`statements`* block starts to execute, the entire block will be executed even if *`condition`* becomes false; the loop will terminate only when *`condition`* is reevaluated at the beginning of the loop. (Once *`condition`* evaluates as false at the beginning of the `while`-loop, the *`statements`* block will not execute and execution will jump to the next statement after `end`.)

A common problem when writing code with `while`-loops is the creation of infinite loops. This is certain to happen if the variables involved in *`condition`* are not updated in the *`statements`* block. However, even if they are updated, it is possible that *`condition`* will never evaluate as true:

```
x = 3;
while (x>2)
    x = 2*x;
end
```

If this happens, execution can be interrupted at the command line by typing **CTRL-C**. A simple way to prevent infinite loops is to include a counter variable and a corresponding test condition:

```
x = 3; n = 1;
while ((x>2) && (n<1000))
    x = 2*x;
    n = n+1;
end
```

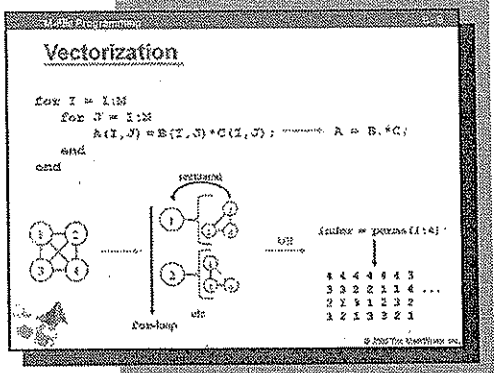
Try

```
>> edit whileloop
>> whileloop
```

Random walk simulation

```
>> edit randwalk
>> [x,y] = ...
randwalk(0,0)

>> edit rwalkimage
>> rwalkimage(100);
```

Vectorization



From its inception MATLAB was intended to be an array-based language: all of the vector and matrix operations in MATLAB are optimized for efficient performance. Consequently, there are many operations that, in other languages, would be written with scalar operations within `for`-loops but that can be written in MATLAB with a small number of vector and matrix operations. This process, called *vectorization*, can lead to clearer, more concise code.

Some opportunities for vectorization are obvious, such as any basic mathematical operation on arrays:

```
for k=1:n
    y(k) = sin(x(k));
end
```

}

```
y = sin(x);
```

Others, however, are more subtle, and there are, of course, many instances in which it is impossible to avoid use of a `for`-loop (e.g., Monte Carlo simulation where each experiment is a complex sequence of calculations). Nonetheless, consider each `for`-loop as a potential opportunity for vectorization. Some useful techniques include:

- Logical indexing.
- Using multiple dimensions (many statistical functions in MATLAB include an optional dimension argument).
- Searching the documentation – often MATLAB has a built-in function that either performs the desired action or can be adapted to your purpose.

Try

The traveling salesman

Getting the permutations:

```
>> type perms
>> perms(1:4)'
```

Comparing permutations:

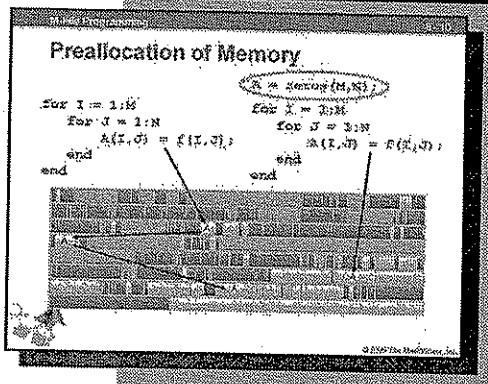
Using loops:

```
>> travelingsales1
```

Vectorized:

```
>> travelingsales2
```

Preallocation of Memory



Unlike many languages, MATLAB does not require variables to be declared before use. This means that you can change the dimensions of your variables dynamically. Although this is convenient (and, in some cases, very hard to avoid), it can lead to poor performance. When creating large arrays it is preferable to *preallocate* memory for the variable.

When creating variables in memory, MATLAB ensures that, when completed, all of the elements are located together in a contiguous block of addresses. This collocation improves the performance of subsequent calculations: the read/write head never has to travel very far when accessing the elements of the variable.

The process of *creating* the variable, however, can be slowed down if MATLAB does not know in advance how large a variable it is creating. This is the case in the function `makeA1.m`, where each time through the inner loop MATLAB is told to augment the existing `A` with an additional element. MATLAB does its best, attempting to place the new element together with the existing elements in memory. If MATLAB runs out of space in a particular location, however, it must search memory for more room, write all of the elements to new addresses, and continue. This process takes time, and can happen repeatedly.

A simple solution is to begin by creating a dummy version of the variable you want to create, of appropriate size, populated entirely by zeros. If you type

```
>> A = zeros(m,n)
```

MATLAB understands immediately that you require room for `m*n` elements. It allocates a block of memory of that size and with zeros as placeholders. Your code can then overwrite the zeros.

This change is implemented in `makeA2.m`. For large `A`, there is a noticeable performance increase. Preallocation also prevents memory fragmentation, leaving room for further large variables.

Try

Matrix creation function with and without preallocation

```
>> edit makeA1
>> edit makeA2
>> A = makeA1(3,2)
>> A = makeA2(3,2)
```

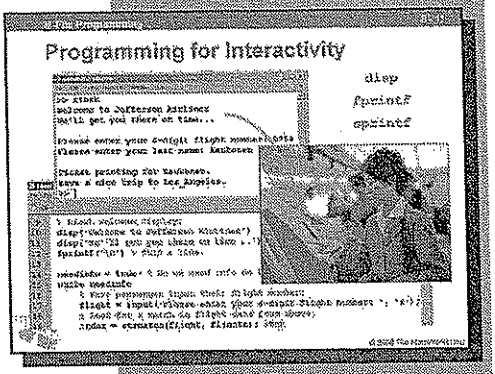
Compare speeds

```
>> clear A;
>> tic; A = ...
makeA1(500,500); ...
toc;
>> clear A;
>> tic; A = ...
makeA2(500,500); ...
toc;
```

Preallocate for integer type

```
>> edit numbers.dat
>> data = zeros(...
2,'int8');
>> fid = fopen(...
'numbers.dat','rt');
>> data(:) = ...
fscanf(fid,'%d');
>> data = data'
```

Programming for Interactivity



When you write a program for your own use, you have a good idea about how to provide appropriate inputs, how the program will behave, and how to interpret the outputs. If you forget the details of a program, however, or it becomes so complex that you lose track of its overall structure, then it is more likely that you will provide inappropriate inputs, the program will behave unexpectedly, and the outputs will become indecipherable. When a program is shared with a wider audience, the user/program interaction becomes even more unpredictable.

Usability means providing a friendly, intuitive, and consistent interface to the user. From a programming perspective, this means anticipating user behavior–desired or not–and handling each case judiciously.

Usability can be increased simply by providing clear interaction with the user. To display information while the program is running, the MATLAB commands `disp`, `fprintf`, and `sprintf` are useful. `disp` displays variables back to the console; `fprintf` and `sprintf` are low-level commands for writing formatted data to files or strings, respectively. The file in `fprintf` can be the console (give no file id); the string returned by `sprintf` can be displayed with `disp`.

It can also be helpful to provide errors and warnings to the user, in the case of undesired behaviors. You can create your own error and warning messages in MATLAB with the `error` and `warning` functions, respectively. In their simplest use, they work very much like `disp`:

```
if (x < 0)
   error('Negative square root?!  I quit!')
else
    y = sqrt(x);
end
```

However, unlike simple display statements, these functions create actual MATLAB errors or warnings. This means that errors will stop program execution, and provide a link to the error; warnings can be optionally disabled.

Try

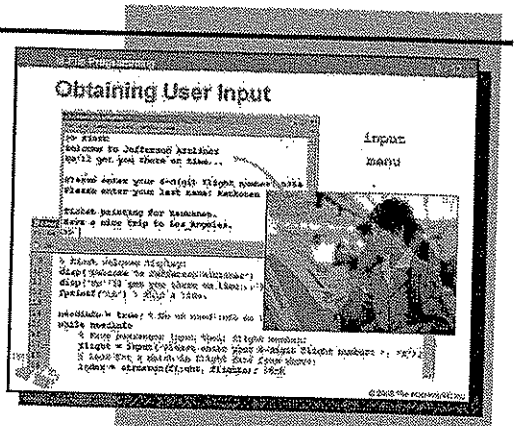
```
>> disp('Ciao')
>> fprintf(...
'%s\n','Ciao')
>> disp(sprintf(...
'%s\n','Ciao'))
```

There's more!

When writing functions, two important usability considerations are: allowing for optional inputs or permitting different behaviors based on the calling syntax; and error-checking of inputs.

See Appendix D-7, D-8

Obtaining User Input



One way to control user-program interaction is to find out how the user responds to the program. MATLAB provides a number of functions that allow you to communicate with the user through the console during execution, and give the user the opportunity to provide feedback.

The `input` command pauses execution and displays a message (or *prompt*) to the user. It waits for the user to enter a valid MATLAB expression and press **Enter**. The expression is then assigned to a variable for use by the program, and execution continues:

```
uage = input('Please enter your age:  ');
```

When expecting text input, it is useful to include the `'s'` option to the `input` function, which forces MATLAB to interpret the user input as a string:

```
uname = input('Please enter your name:  ','s');
```

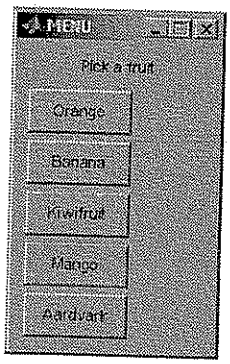
Without this option, any text entered by the user will be interpreted as a MATLAB variable or expression unless it is explicitly enclosed in single quotes.

The menu function is a simple way to create a graphical menu of a finite number of discrete choices, without requiring any GUI programming:

```
fruit = menu('Pick a fruit','Orange',...
    'Banana','Kiwifruit','Mango','Aardvark')
```

The return variable `fruit` will take an integer value corresponding to the number of the option selected by the user ("Orange" = 1, "Banana" = 2, etc.):

```
if (fruit==5)
    error('Aardvarks are mammals, not fruit!')
end
```



Try

Guess the number

```
>> edit numguess
>> numguess
```

Formatted display

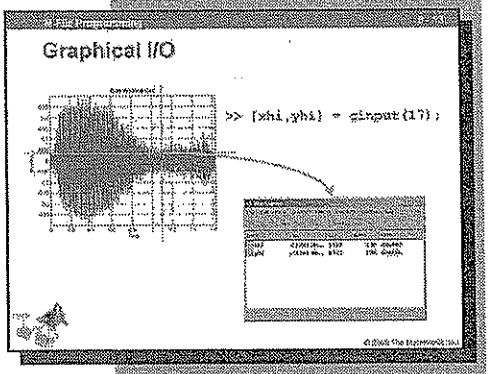
```
>> edit welcome
>> welcome
```

Airport e-ticket kiosk

```
>> edit kiosk
>> edit flights.txt
>> edit ...
warninglist.txt
>> kiosk
```

Try invalid and valid flight numbers, and names not on and on the warning list.

Graphical I/O



When you create a MATLAB plot, certain features in your data may become more noticeable. You may want to locate the exact position of these features, and label them on the plot. MATLAB provides features and functions that allow you to interactively collect graphical input and position graphical labels.

The function `ginput` allows you to collect data from a 2-D MATLAB plot and return the data to the workspace for processing. If you type

```
>> [x,y] = ginput(n)
```

when a 2-D plot is open, the plot comes forward. When you move your mouse into the axes containing the plot, a large positioning crosshair (╬) is displayed. Each click of the mouse on a point in the plot collects the coordinates of that point and returns them to the workspace. After `n` clicks, the vectors `x` and `y` each contain `n` coordinates, and control returns to the command prompt.

The function `gtext` allows you to place text labels in a plot at positions determined by your mouse clicks.

```
>> gtext('string')
```

waits for you to press a mouse button or key while the pointer is within a figure window. Pressing a mouse button or key places `'string'` on the plot at the selected location.

```
>> gtext({'string1','string2','string3',...})
```

places all strings with one click, each on a separate line.

```
>> gtext({'string1';'string2';'string3';...})
```

places one string per click, in the sequence specified.

Try

```
>> edit whalecall
>> whalecall
```

Close Figures 1 and 3 and maximize Figure 2.

```
>> grid on
```

Collect 17 approximately equally spaced data points on the upper envelope of the plot.

```
>> [xhi,yhi] = ...
ginput(17);
```

Collect 17 approximately equally spaced data points on the lower envelope of the plot.

```
>> [xlo,ylo] = ...
ginput(17);
```

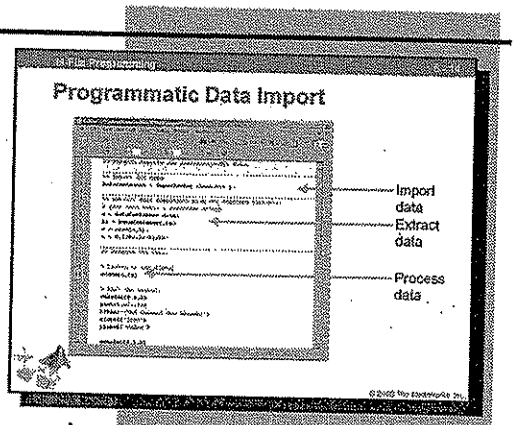
Add the envelopes to the plot.

```
>> hold on
>> plot(xhi,yhi,'ro-')
>> plot(xlo,ylo,'ro-')
```

Label the envelopes.

```
>> gtext(...
{'Upper Envelope';...
'Lower Envelope'})
```

Programmatic Data Import



The Import Wizard is useful for quick, *interactive*, one-time import of data in supported file formats. When automating this process with scripts, however, the import must be made *programmatic*.

The MATLAB function `importdata` is convenient for programmatic import from supported file formats. It is essentially the Import Wizard without the interface. Whatever breakdown into MATLAB variables the Import Wizard suggests for data from a particular file, the `importdata` function accepts. You do not have the same opportunity to revise the breakdown of the data as you do in the Import Wizard. The syntax for the `importdata` function is correspondingly simple:

```
>> A = importdata('filename');
```

`A` is a structure where the fields of `A` and their data types depend on the file format. You need to test the import and observe the characteristics of `A` before writing any additional code to process the data that `A` will contain.

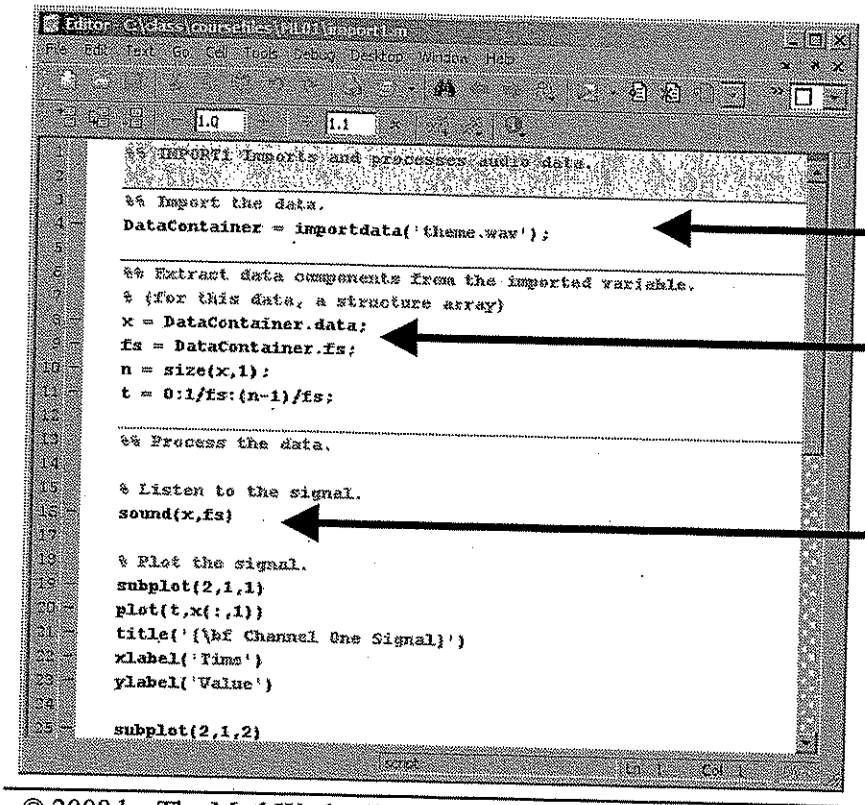
Try

Import and process audio data.

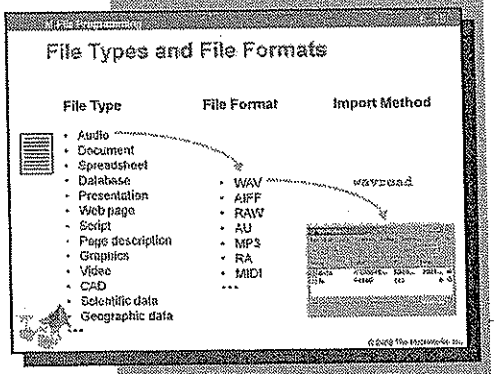
```
>> edit import1
>> import1
```

Import and process image data.

```
>> edit import2
>> import2
```



File Types and File Formats



MATLAB provides many functions that offer greater control over programmatic data importing than `importdata`. Your choice of which method to use depends on the *file type* and *file format* of the data.

File type refers very generally to the origins of the data, the way it will be used, and the kinds of devices that may be required to see, hear, and manipulate the data. Common file types include

- Audio
- Computer Aided Design (CAD)
- Document
- Spreadsheet
- Database
- Presentation
- Webpage
- Script
- Page description
- Graphics
- Video
- Scientific data
- Geographic information

File format refers to the particular way information is encoded in a file. There are different formats for different file types, and there are often different (sometimes competing) formats for any one file type. Because a computer can store only bits, it must have some way of converting the formatted information in a file to 0s and 1s when importing data, and vice versa when exporting data. When functions exist for making the conversions for a file format, the format is said to be *supported*.

Supported file formats in MATLAB can be listed by typing

```
>> help fileformats
```

Corresponding MATLAB I/O functions can be listed by typing

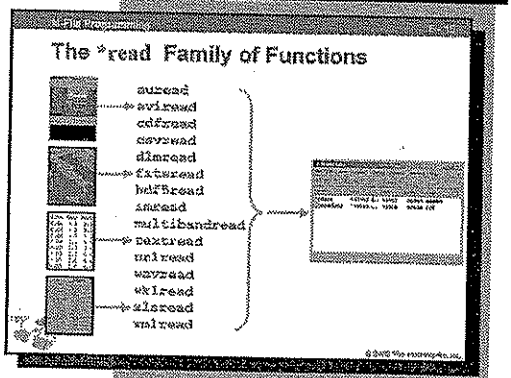
```
>> help iofun
```

Try

Supported file formats

```
>> help fileformats
>> help iofun
```

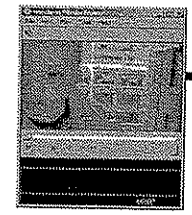
The *read Family of Functions

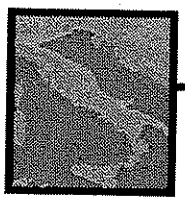


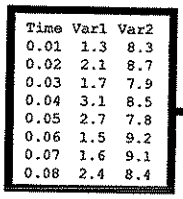
For data import, MATLAB provides a family of functions that each work with a particular supported file format. Functions in this family all have names of the form `*read`, where `*` is short for the name of the format (often the extension of the format).

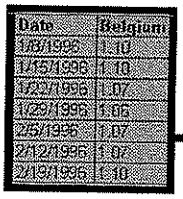
The `*read` functions typically allow lower-level control of data import than do either the Import Wizard or the `importdata` function, but that lower level control usually comes with some requirement that you understand the encoding used by a format. Inputs are the filename and, typically, formatting information about how to break down the data into MATLAB variables. If you do not understand the format, you may have difficulty supplying the formatting information, and the Import Wizard or the `importdata` function may be a better choice.

The `*read` functions are listed below. All family members accept and return similar arguments, but you should review the documentation for idiosyncrasies of individual family members.

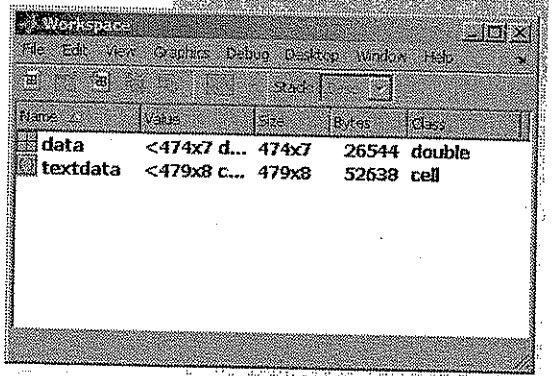








auread
aviread
cdfread
csvread
dlmread
fitsread
hdf5read
imread
multibandread
textread
urlread
wavread
wk1read
xlsread
xmlread



Try

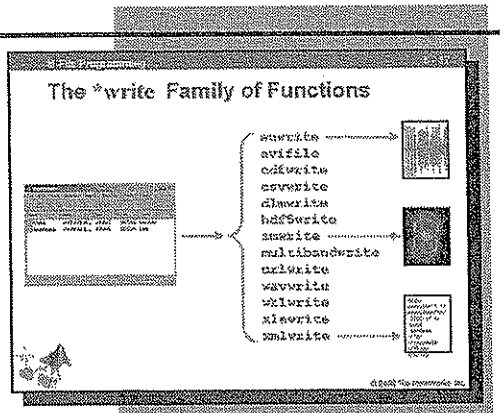
Text files
```
>> edit interpdata2
>> interpdata2
>> edit flightpath
>> flightpath
```

Microsoft® Excel® files
```
>> edit getgas
>> getgas
```

Audio files
```
>> edit whalecall
>> whalecall
```

Image files
```
>> edit deconstruct
>> deconstruct
```

The *write Family of Functions



For data export, MATLAB provides a family of functions that complement the family of `*read` functions. Functions in this family all have names of the form `*write`.

As with the `*read` functions, the `*write` functions require that you have some understanding of the encoding used by a format.

Both the `*read` functions and the `*write` functions allow you to specify absolute path information as part of the filename, so that files can be imported from or exported to any disc or network directory.

The `*write` functions are listed below. Again, all family members accept and return similar arguments, but you should review the documentation for idiosyncrasies of individual family members.

Try

Excel® files
```
>> edit stockwrite
>> stockwrite
```

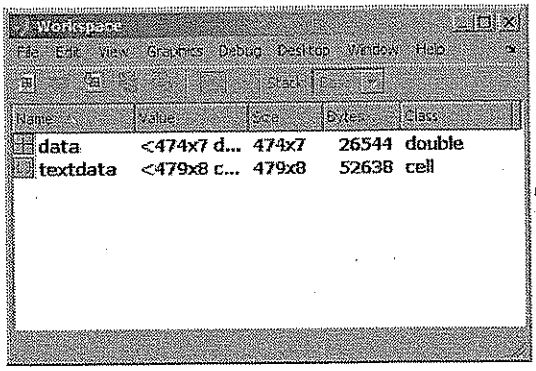
Audio files
```
>> edit whalewrite
>> whalewrite
```

Image files
```
>> edit deconstruct
>> deconstruct
```

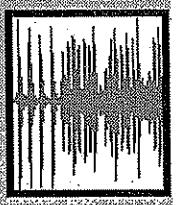
There's more!

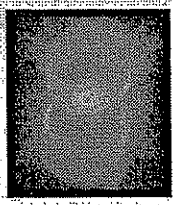
MATLAB includes functions for specialized file types, and low-level functions for reading and writing text and binary files with arbitrary formatting. Toolboxes are available that allow real-time I/O from a variety of sources.

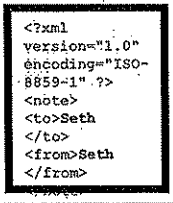
See Appendix D-9 – D-12



auwrite
avifile
cdfwrite
csvwrite
dlmwrite
hdf5write
imwrite
multibandwrite
urlwrite
wavwrite
wklwrite
xlswrite
xmlwrite

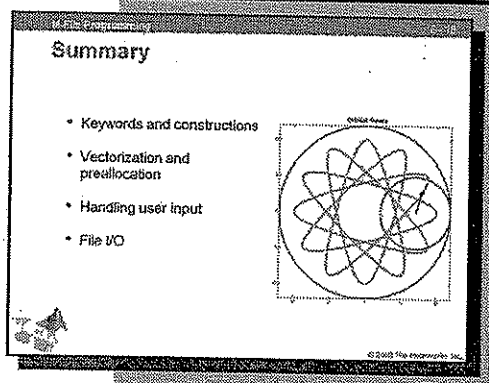


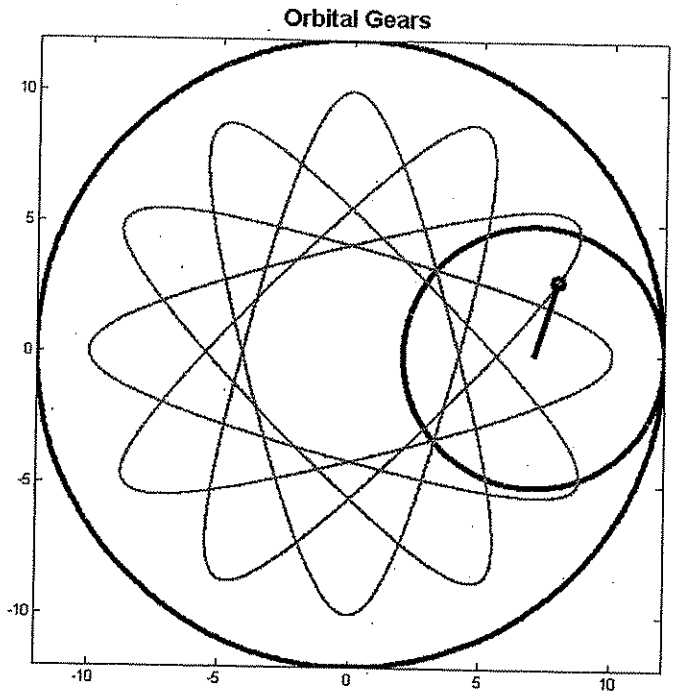




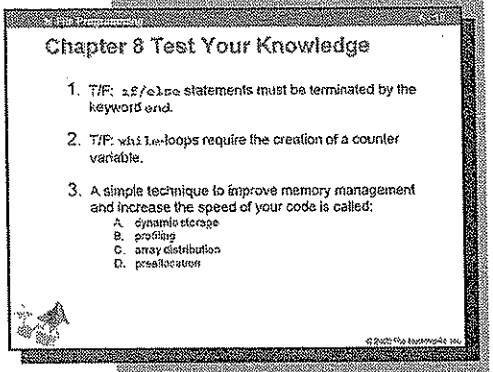
Summary

- Keywords and constructions
- Vectorization and preallocation
- Handling user input
- File I/O





Chapter 8 Test Your Knowledge



Name:_____

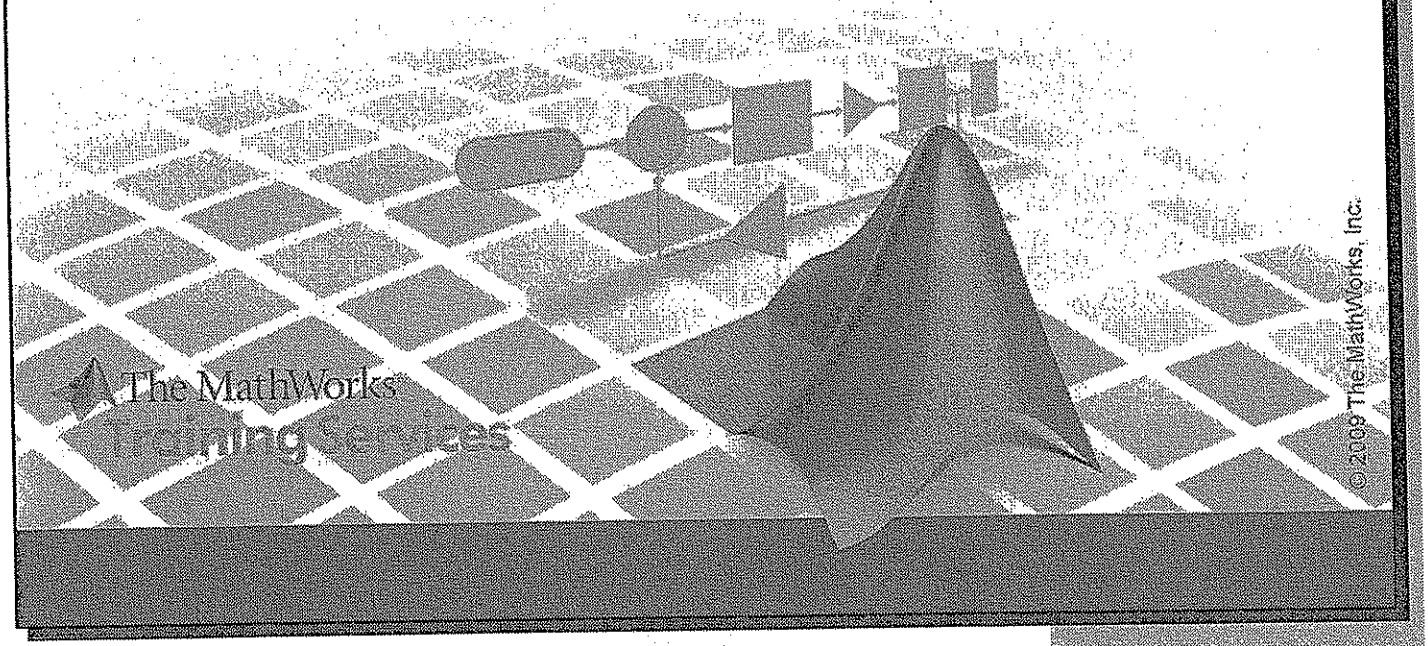
1. T/F: `if/else` statements must be terminated by the keyword `end`.

2. T/F: `while`-loops require the creation of a counter variable.

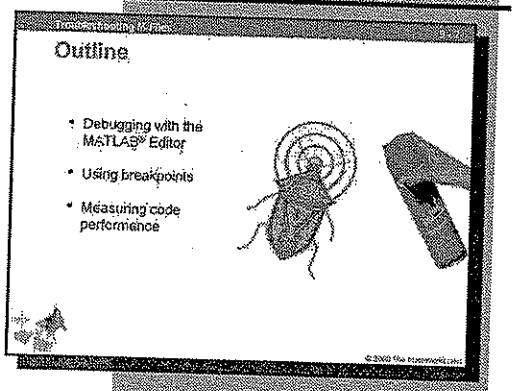
3. A simple technique to improve memory management and increase the speed of your code is called:
   A. Dynamic storage
   B. Profiling
   C. Array distribution
   D. Preallocation

MATLAB® Fundamentals

Troubleshooting M-Files

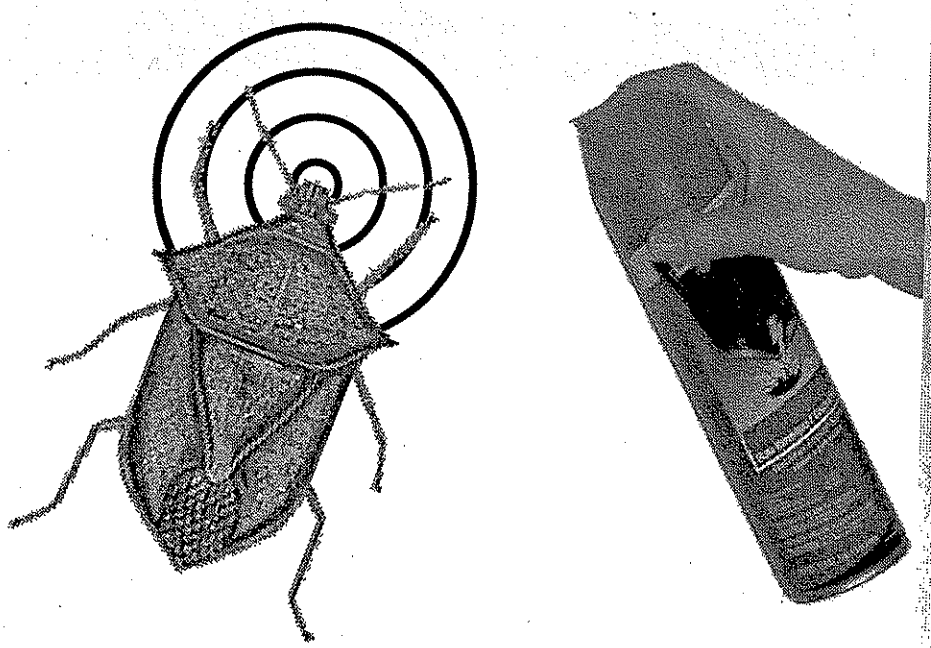


Outline



- Debugging with the MATLAB® Editor
- Using breakpoints
- Measuring code performance

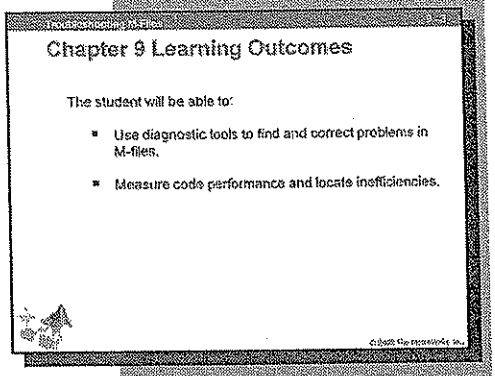
Very few programs work perfectly at the first attempt. Tracking down all possible problems and unintended behaviors of a program takes time and effort. In this chapter we discuss the tools available in MATLAB to assist in the debugging process. We also highlight techniques for measuring code performance.



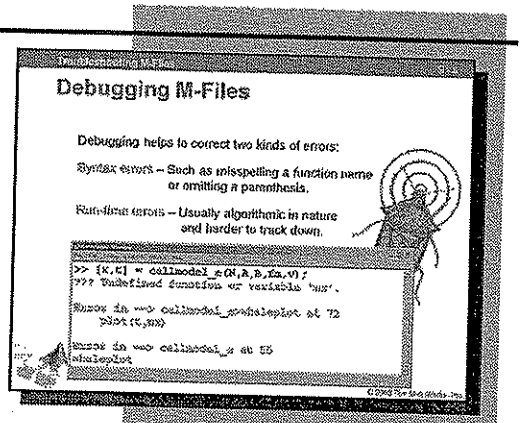
Chapter 9 Learning Outcomes

The student will be able to:

- Use diagnostic tools to find and correct problems in M-files.
- Measure code performance and locate inefficiencies.



Debugging M-Files



To introduce the debugging features of the MATLAB Editor, a nonworking version of the function `callmodel_n.m`, named `callmodel_e.m`, is available in your course files.

Debugging is the process by which you isolate and fix problems with your code. Debugging helps to correct two kinds of errors:

- Syntax errors – For example, misspelling a function name or omitting a parenthesis.
- Run-time errors – These errors are usually algorithmic in nature. For example, you might modify the wrong variable or code a calculation incorrectly. Run-time errors are usually apparent when an M-file produces unexpected results. Run-time errors can be difficult to track down because the local workspace of a function is lost when errors force a return to the MATLAB base workspace.

The MATLAB Editor can help you to discover many syntax errors, as you type. For example, strings missing a delimiter (') are colored red, while closed strings turn the color purple. The Editor also finds mismatched parentheses. To set your editing preferences, choose **File → Preferences** from within the Editor.

The call

```
>> [x,t] = callmodel_e(N,A,B,fm,v);
```

produces a run-time error with the following message:

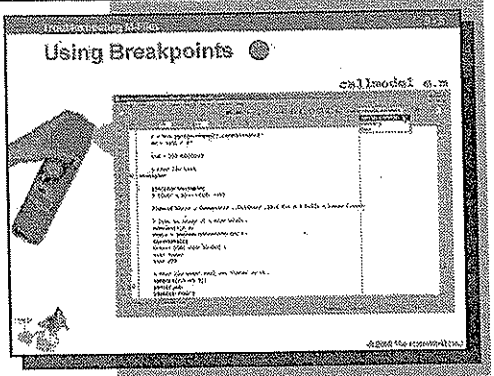
```
??? Undefined function or variable 'nx'.

Error in ==> callmodel_e>whaleplot at 72
    plot(t,nx)

Error in ==> callmodel_e at 55
whaleplot
```

Often, the error message that MATLAB displays in the Command Window is enough to locate and diagnose the source of a problem.

Try

```
>> N = 3;
>> A = 2;
>> B = 1.5;
>> fm = 0.65;
>> v = 0.05;
>> [x,t] = ...
callmodel_e(...
N,A,B,fm,v);
```

Using Breakpoints



If an error message is not enough to diagnose a problem, you can activate the MATLAB Debugger by setting a *breakpoint* in the Editor. Breakpoints are specific lines of code where you would like MATLAB to stop execution and "hold everything" – that is, keep all active function workspaces open and accessible.

There are three basic types of breakpoints:

- A *standard breakpoint*, which stops at a specified line.
- A *conditional breakpoint*, which stops at a specified line only under specified conditions.
- An *error breakpoint*, which stops when an M-file produces the specified type of warning, error, or `NaN` or infinite value.

To set a standard breakpoint using the Editor, click on a dash in the *breakpoint alley* next to the line number of an executable line of code. The dash turns into a red dot: ●. The breakpoint is disabled (gray) if there is a syntax error or if you have not saved to a directory on the path. To remove the breakpoint, click on the dot.

Conditional and error breakpoints can be set by choosing one of **Debug → Set/Modify Conditional Breakpoint** or **Debug → Stop if Errors/ Warnings**. Both choices open a dialog box to specify the particulars.

After setting a breakpoint, run the M-file. The prompt changes to `K>>`, indicating that MATLAB has entered *debug mode*. The program pauses at the first breakpoint, and that line will be executed when you continue to the next breakpoint (F5) or step through the program one line at a time (F10). The pause is indicated in the Editor by a green arrow ➪ just to the right of the breakpoint. A hollow arrow ⇨ indicates that MATLAB control is in a subfunction or nested function call.

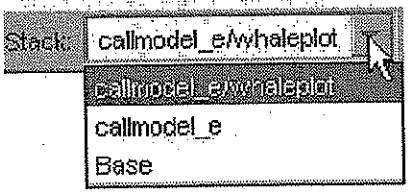
The function displayed in the **Stack** field on the toolbar at the top of the Editor changes to reflect the current function. The call stack includes subfunctions as well as called functions.

Try

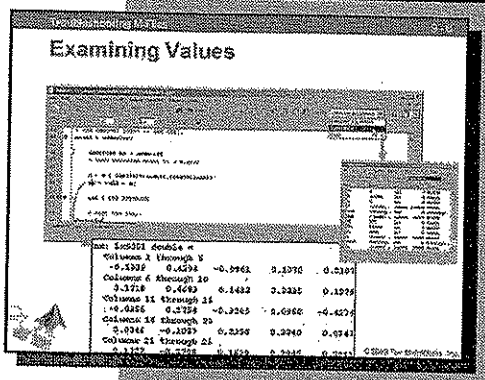
Set a breakpoint in `callmodel_e.m` near a line relevant to the error message. Run the file again by typing

```
>> [x,t] = ...
callmodel_e(...
N,A,B,fm,v);
```

In debug mode, use the buttons at the top of the Editor to step through the file near the error.



Examining Values



Examine values of variables when you want to see whether a line of code has produced the expected result or not. If the result is as expected, continue running or step to the next line. If not, then that line, or a previous line, contains an error.

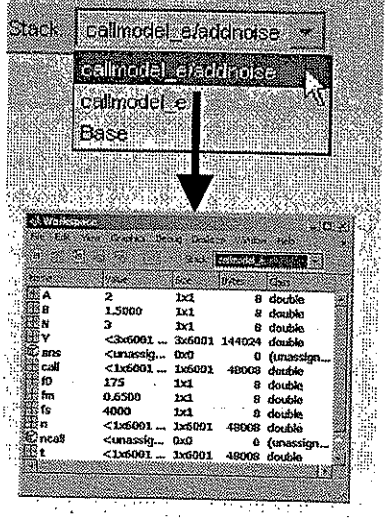
One way to examine values, outside of debug mode, is to remove semicolons from the ends of lines making assignments. When the file is run, the values are displayed in the Command Window.

In debug mode, if you position your cursor near a variable, its current value pops up as a *datatip*. The datatip stays in view until you move the cursor. There is no need to remove semicolons.

```
47
48      function nx = addnoise
        % Adds Gaussian noise to a signal.
49
50 -    n = 0 + sqrt(v)*randn(1,length(call));
51 -    nx = call + n;
52
53 ◆⇨   e
54
55      % Plo
```

```
nx: 1x6001 double =
  Columns 1 through 5
    0.0563   -0.1608    0.0620   -0.1926    0.2566
```

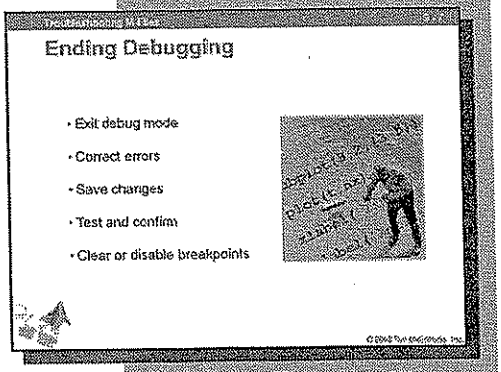
In debug mode, you can choose any active workspace from the **Stack** field on the toolbar at the top of the MATLAB Editor. The workspace variables then appear in the Workspace browser where they can be examined in the Array Editor.



Try

Examine various values in `callmodel_e.m` while in debug mode.

Ending Debugging



While debugging, you can change the value of a variable in the current workspace to see if a new value produces the expected results. While the program is paused, assign a new value to the variable in the Command Window, Workspace browser, or Array Editor, then continue running or stepping through the program. If the new value does not produce the expected results, the program has another problem.

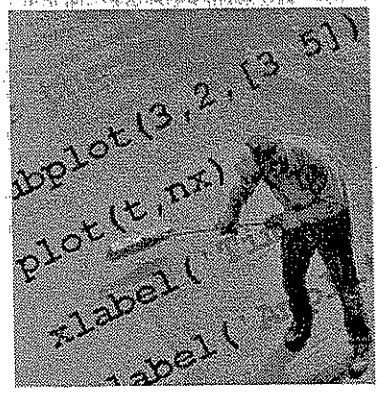
Try

Exit debug mode, correct and save `callmodel_e.m`, and clear all breakpoints.

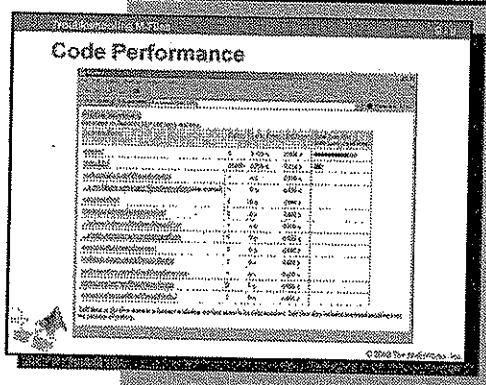
After identifying a problem, end the debugging session. You must end a debugging session if you want to change and save an M-file to correct a problem, or if you want to run other functions in MATLAB. If you edit an M-file while in debug mode, you can get unexpected results when you run the file.

To end debugging, click the Exit Debug Mode button or select **Exit Debug Mode** from the **Debug** menu. After quitting debugging, the pause arrows in the Editor display no longer appear, and the normal prompt >> reappears in the Command Window. You can no longer access the call stack.

After you think you have identified and corrected a problem, you can disable breakpoints, so that the program ignores them (this is especially useful for conditional breakpoints), or you can remove them entirely. Clicking a breakpoint dot clears it. Right-clicking a breakpoint dot shows a context menu that will allow you to disable the breakpoint (an X will appear through the dot), clear it, or, if it is conditional, change the condition. Conditional and error breakpoints can also be cleared or modified in the **Debug** menu.



Code Performance



To improve the efficiency of your M-files, you must first locate the source of inefficiencies.

Some inefficiencies may be hardware related. The command `bench` runs a sequence of moderately demanding numerical calculations and graphical manipulations on your computer. It then displays a report comparing execution speed with several other computers. Fluctuations of 5–10% on repeated runs are common. Run the tests *n* times by typing

```
>> bench(n)
```

To time sections of your code, or an entire M-file, use the commands `tic` and `toc`. Typing

```
>> tic; <code>; toc;
```

displays the elapsed time for running `<code>` on your computer. You can use `tic` and `toc` in the MATLAB Editor around a selection of code. If you highlight the selection, then right-click on it, you can choose **Evaluate Selection** from the context menu to time it.

The MATLAB Profiler allows you to get a more global view of where your M-files, and the files they call, are spending their time. To open the Profiler, select **Desktop → Profiler** or type

```
>> profile viewer
```

Type the code you would like to run in the **Run this code** field, and then click **Start Profiling**. A Profile Summary report is generated.

The Profile Summary report provides time and calling frequency for the files that ran while the Profiler was on. From this summary, identify the functions whose speed you want to improve, either because of the time or the number of calls. Click a name in the **Filename** list, and a more detailed report appears for that function.

Try

Benchmarking your computer

```
>> bench
>> T = bench(10);
```

Basic timing functions

```
>> A = rand(1e3);
>> tic; ...
[L,U] = lu(A);
toc;
```

Matrix creation function

```
>> edit makeA1
>> A = makeA1(2,3)
>> A = makeA1(3,2)
>> clear A; ...
tic; ...
A = ...
makeA1(500,500); ...
toc;
```

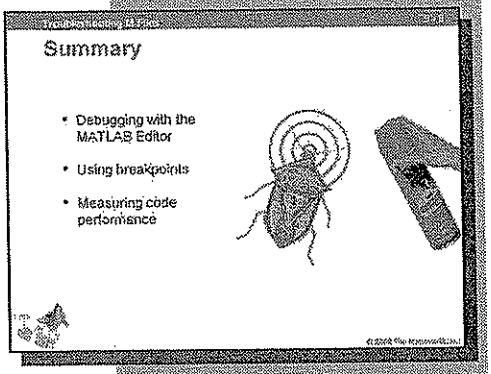
Then run

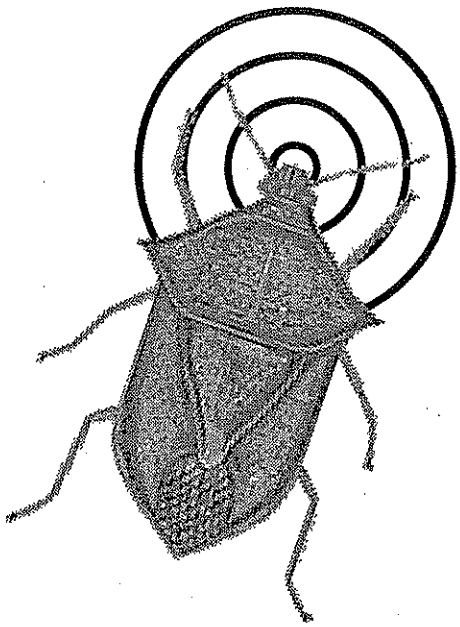
```
A = makeA1(500,500);
```

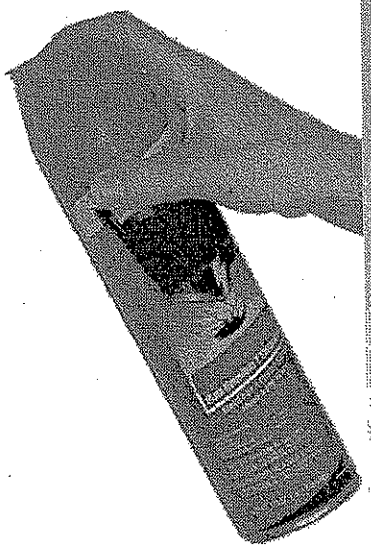
in the Profiler.

Summary

- Debugging with the MATLAB Editor
- Using breakpoints
- Measuring code performance



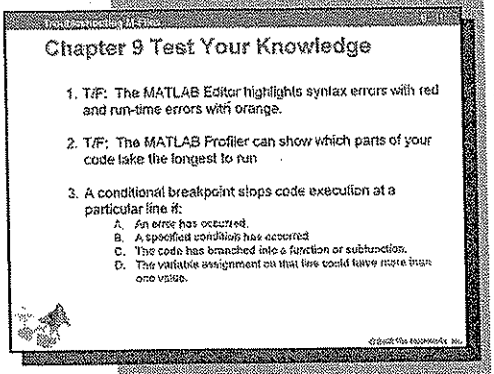




This page intentionally left blank

Chapter 9 Test Your Knowledge

Name:_____



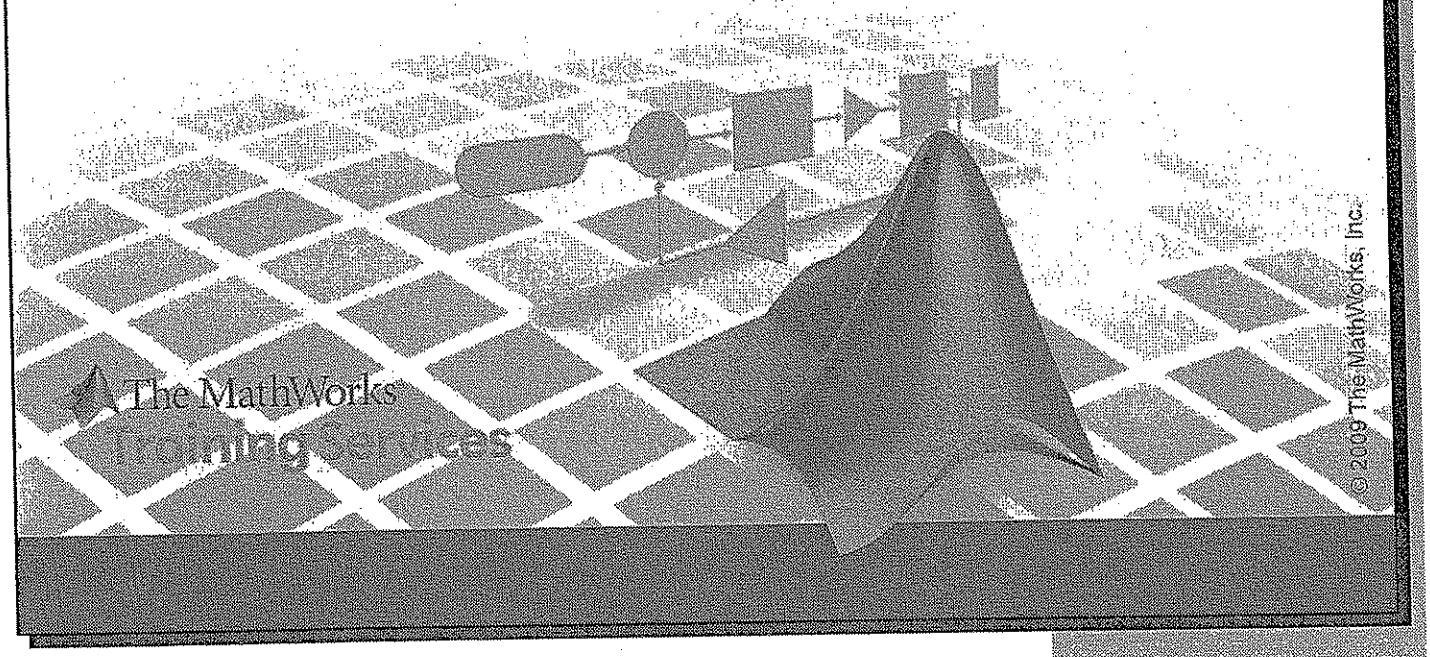
1. T/F: The MATLAB Editor highlights syntax errors with red and run-time errors with orange.

2. T/F: The MATLAB Profiler can show which parts of your code take the longest to run.

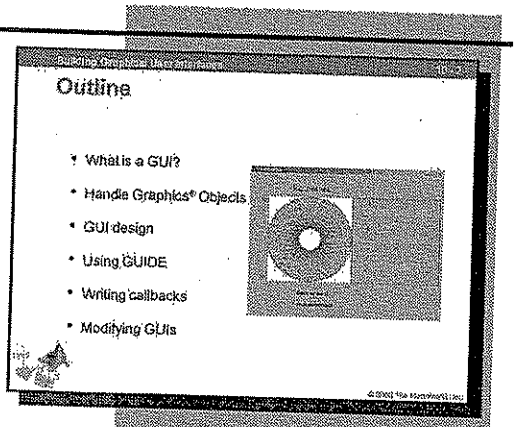
3. A conditional breakpoint stops code execution at a particular line if:
   A. An error has occurred.
   B. A specified condition has occurred.
   C. The code has branched into a function or subfunction.
   D. The variable assignment on that line could have more than one value.

MATLAB® Fundamentals

Building Graphical User Interfaces

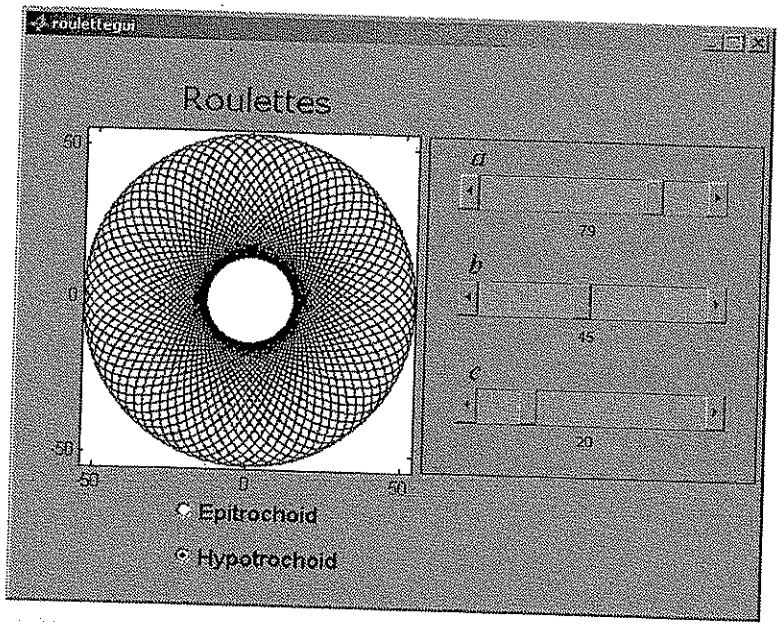


Outline



- What is a GUI?
- Handle Graphics® Objects
- GUI design
- Using GUIDE
- Writing callbacks
- Modifying GUIs

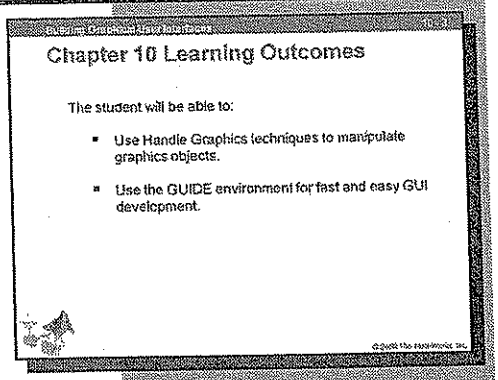
This chapter shows you how to make user-friendly programs in MATLAB® with the use of a graphical user interface (GUI). GUIs allow users to interact with your programs without having to understand, or even see, the code that does the work in the background. GUIs also allow you to focus user attention on specific input/output behaviors of a program, while deemphasizing the intermediate mechanisms. GUIs offer many usability advantages over simple M-file programs.



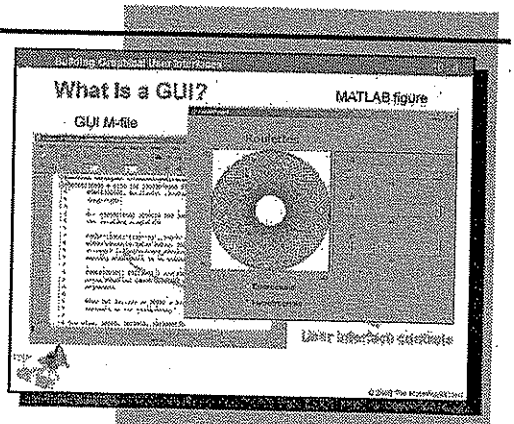
Chapter 10 Learning Outcomes

The student will be able to:

- Use Handle Graphics techniques to manipulate graphics objects.
- Use the GUIDE environment for fast and easy GUI development.



What Is a GUI?



A GUI in MATLAB typically consists of two components:

- A MATLAB figure file displaying the user interface
- A MATLAB M-file that runs the GUI in the background

The figure file and the M-file are associated with each other by a common name, such as `mygui.fig` and `mygui.m`. When a user interacts with the controls on the interface, MATLAB looks to the associated M-file for instructions on how to update the interface. It is possible, using additional graphics commands, to store the GUI entirely in the M-file, and have the M-file create the figure when it is run.

GUIs have their own terminology. Controls on the interface, such as sliders, push buttons, text boxes, etc. are *user interface controls*. They determine how a user can, and cannot, interact with the program in the M-file. A change in the state of a user interface control is an *event*. The specific piece of code in the M-file that is responsible for updating the interface when the state of a control changes is a *callback* for that control. The M-file assembles all of the callbacks for the GUI and adds initialization code, called a *switchyard*, that directs the execution to the appropriate callback each time the M-file is called.

Try

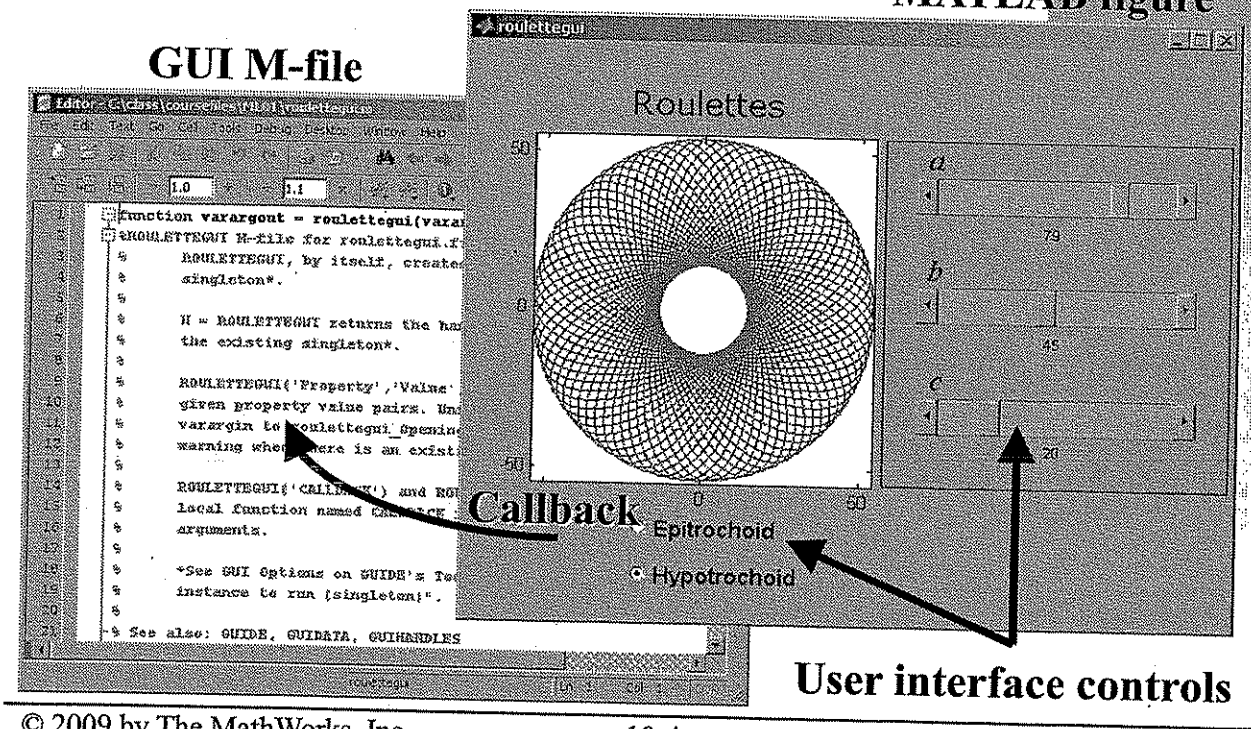
Orbital gears

Run the GUI.

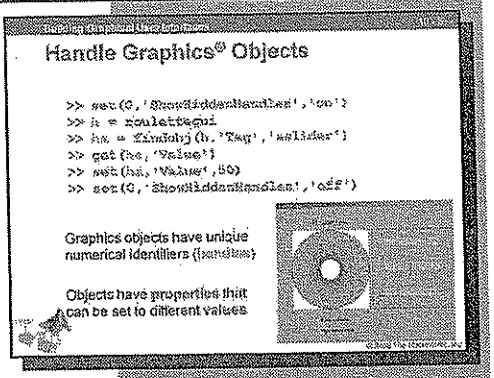
```
>> roulettegui
```

View the GUI M-file.

```
>> edit roulettegui
```



Handle Graphics® Objects



MATLAB figures consist of various graphics *objects*, such as a figure window, axes, text, and, for GUIs, user interface controls. Each object has a fixed set of *properties*. You modify the *values* of these properties to control the appearance of the figure. This is how callbacks update a GUI interface.

The MATLAB `get` function lets you get the value of any property, and the companion `set` function lets you set new values. Before you can use these functions, however, you have to point MATLAB to the graphics object you want to either query (`get`) or modify (`set`). For this purpose, whenever MATLAB creates a graphics object, it assigns it a unique numerical identifier, called a *handle*.

You can obtain the handle of an object being plotted (a line, a surface, etc.), or the handle of a GUI figure window, simply by asking for output when the figure is created (or opened). For example,

```
>> h = plot(x,y)
```

returns the handle to the plotted object and assigns it to `h`, and

```
>> h = mygui
```

returns the handle to the GUI figure and assigns it to `h`. Then

```
>> get(h,'Property')
```

returns the current value of `'Property'`, and

```
>> set(h,'Property','Value')
```

modifies `'Property'` to a new `'Value'`.

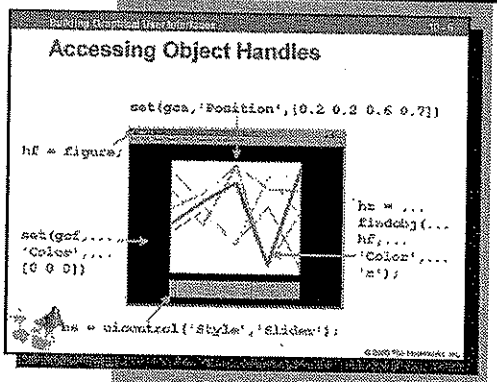
After you obtain the handle of a graphics object, you can point `get` and `set` (and various other MATLAB graphics commands) directly at that object, query it, and then change the values of its properties to modify its appearance.

Try

Basic graphics commands

```
>> h = ...
plot(rand(5),'o-')
>> C = ...
get(h(1),'Color')
>> set(h(1),...
'MarkerFaceColor',C)
>> set(h(1),...
'LineWidth',3)
```

Accessing Object Handles



There are three methods for obtaining the handle to an object:

- Assign the handle to a variable at the time the graphics object is created (or opened).

```
>> h = plot(x,y)
>> h = mygui
```

- Use MATLAB system variables containing common object handles.

```
0       % Root object handle
gcf     % Current figure handle
gca     % Current axes handle
gco     % Current object handle
```

The current object is the last object created or the last object clicked.

- Use the `findobj` or `findall` command to locate graphics objects with specific properties.

```
h = findobj(h_parent,'Property','Value',...)
h = findall(h_parent,'Property','Value',...)
```

The input `h_parent` is the handle of a *parent object*, which limits the scope of the search. A parent object is at a level above the object of interest in the MATLAB graphics hierarchy. For example, the root object (the screen, with handle `0`) is a parent of all graphics objects, a figure is a parent of the axes and user interface controls it contains, an axes is a parent of the plotted objects it contains, etc.

Following the input for the parent object is a sequence of property, value pairs that describe the object. If multiple objects meet the description, multiple handles are returned.

The two commands differ in that `findall` returns *hidden handles*, where `findobj` does not. Hidden handles prevent figures or axes from becoming the target of graphics output, and are invisible to functions that return or reference handles, such as `gcf`, `gca`, and `findobj`. Handle visibility is set with the `HandleVisibility` property of any object. Its values are `on`, `off`, and `callback`. The latter makes the handle invisible to functions executing on the command line, but not from within event-triggered callbacks.

Try

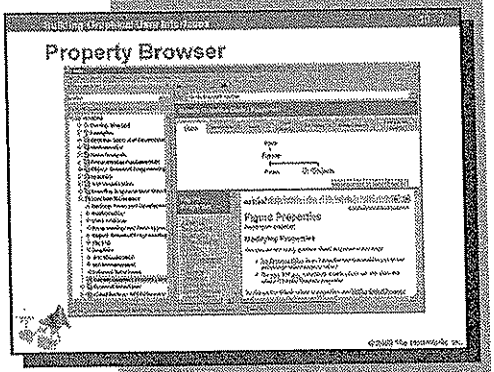
Basic GUI (no figure file)

```
>> edit hg1
>> hg1

>> edit hg2
>> hg2
```

findobj vs findall

```
>> h = rouletteGui
>> ha = findobj(...
h,'Tag','aslider')
>> ha = findall(...
h,'Tag','aslider')
>> get(ha,...
'HandleVisibility')
>> set(0,...
'ShowHiddenHandles',
...
'on')
>> ha = findobj(...
h,'Tag','aslider')
>> get(ha,'Value')
>> set(ha,...
'Value',50)
>> set(0,...
'ShowHiddenHandles',
...
'off')
```

Property Browser



To query properties of a graphics object with the `get` command and then change their value with the `set` command, you need to know the names of the property name/property value pairs that pertain to that object. You also how changing a property to a new value affects the display and operation of your GUI.

The MATLAB documentation contains a special section just for this purpose. It is called the Handle Graphics Property browser, and you access it by opening the Help browser

```
>> doc
```

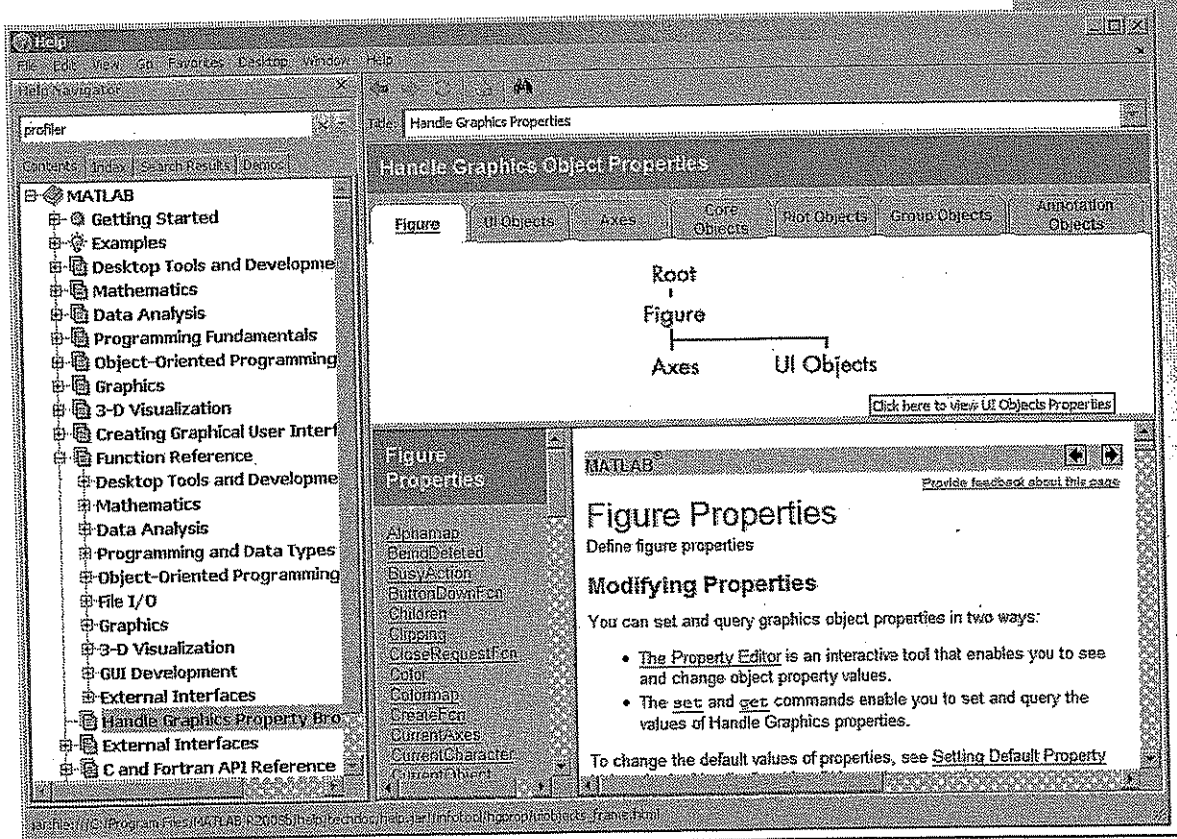
and then navigating under the **Contents** tab of the Help Navigator to **MATLAB → Handle Graphics Property Browser**.

A schematic of the MATLAB graphics hierarchy appears, and you can click any type of graphics object to see a complete list of its property-value pairs, together with full explanations of their meanings.

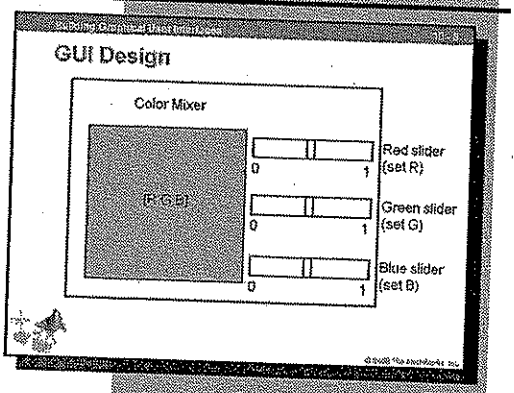
Try

```
>> doc
```

Navigate to the Handle Graphics Property browser.

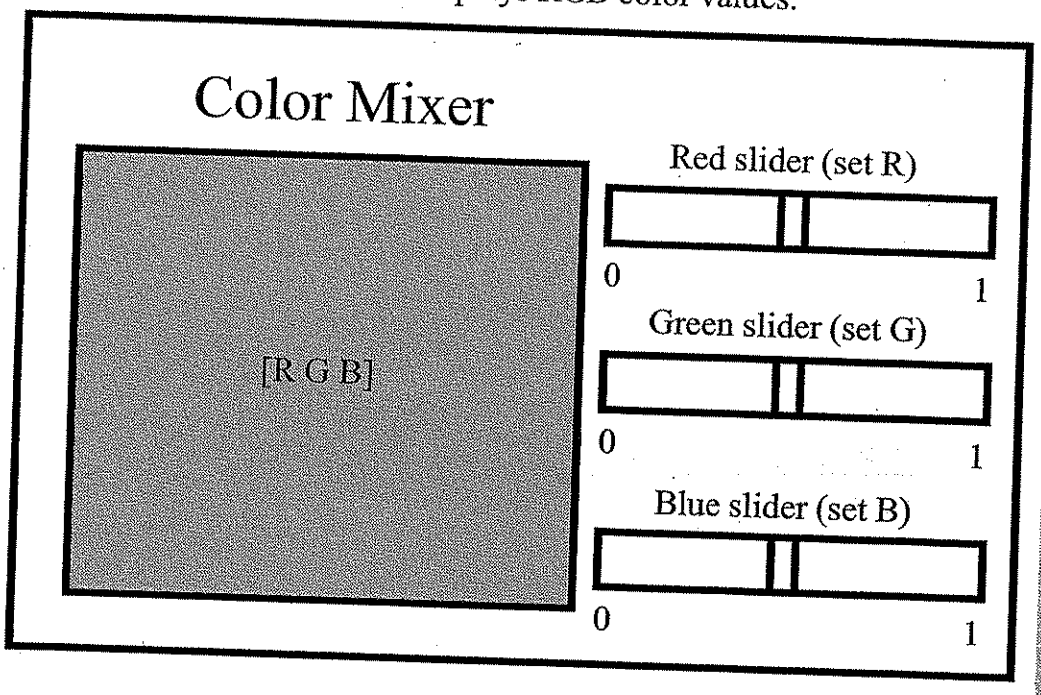


GUI Design



Before you build a GUI, you should design it. Designing is sketching out the layout of the interface, including all user interface controls, and making notes about how a change to any control should affect the interface. The sketch is used to create the layout of your GUI, and the notes are used to write the callbacks.

For example, the sketch below might be the first step in the design of a simple GUI that mixes and displays RGB color values.

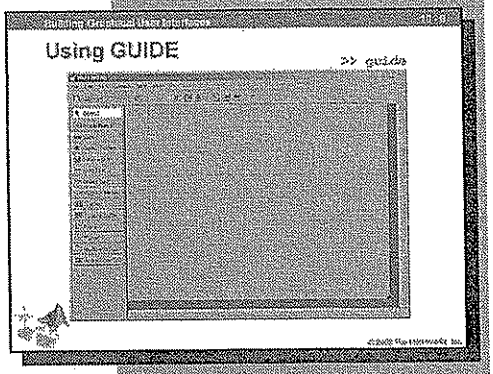


The three sliders each move between a minimum value of 0 and a maximum value of 1. They are used to set the red (R), green (G), and blue (B) components, respectively, of the color being mixed. Each slider has an associated callback that updates the color of the figure axes to the RGB value set by the sliders.

Try

Design principles
`www.math.duke.edu/education/ccp/resources/write/design`

Using GUIDE



When you have a design for your GUI, building it becomes a two-step process:

1. Lay out the interface (create the figure file).

2. Write the callbacks (create the M-file).

MATLAB provides a tool called the Graphical User Interface Design Environment (GUIDE) that automates much of this process for you.

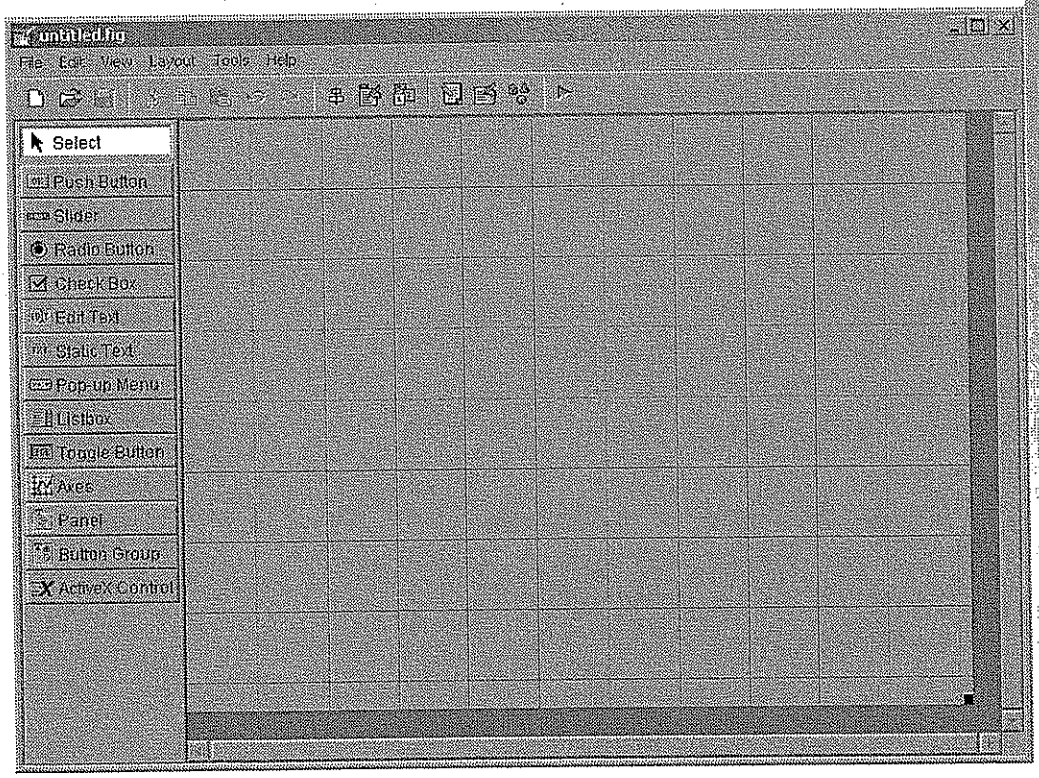
To open GUIDE, type

```
>> guide
```

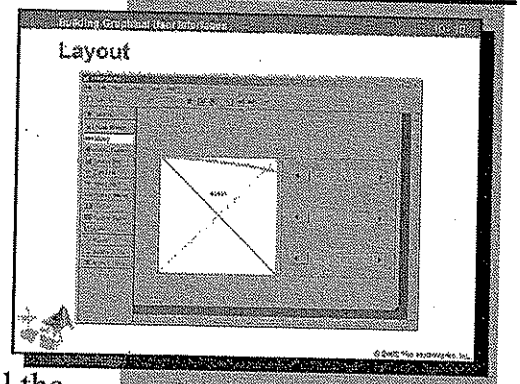
If you choose to begin from a blank GUI (default), a blank layout grid appears. Buttons along the left-hand side indicate various graphics objects, and buttons along the top indicate various formatting options.

Try

```
>> guide
```



Layout



Select and position graphics objects in GUIDE as follows:

1. Click a button in the left-hand pane to toggle it down.
2. Move the mouse into the layout grid. It becomes a crosshair.
3. Click and hold the left mouse button and drag the mouse until the rubber band box that appears encloses the approximate extent of the graphics object you would like to position.
4. Release the mouse button.

The graphics object appears in the layout with the extent specified.

Position an object by clicking it to select it, holding down the left mouse button, and moving the object to a new position. To resize an object, select it, click and hold a corner, and then move the mouse. Deselect an object by clicking away from it in the layout grid.

Create a copy of an object by right-clicking it, holding down the mouse button, and dragging the copy to a new position.

Align individual objects with grid lines by selecting **Layout → Snap to Grid** from the menus at the top of GUIDE. Objects then jump as they approach a grid line when they are moved. The size of the grid can be set by selecting **Tools → Grid and Rulers** from the menu.

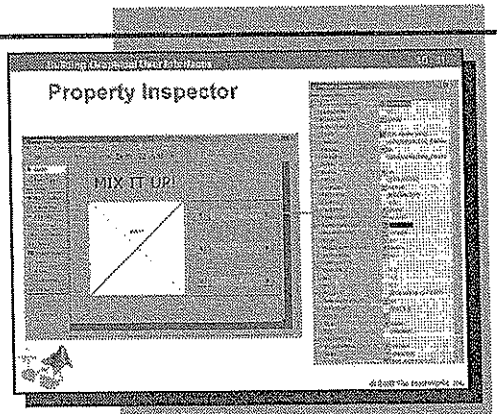
Align multiple objects with each other by holding down the **Ctrl** key and selecting them, and then choosing **Tools → Align Objects** (or the button in the top toolbar). The **Align Objects** dialog opens and allows you to set precise inter-object spacing and alignment.

Choosing **View → Object Browser** (or using the button in the top button palette) displays the hierarchy of graphics objects in your GUI.

Try

Use GUIDE to lay out the color mixer GUI using the design specifications.

Property Inspector



After you have created a basic layout for your GUI, you set property-value pairs for each of the graphics objects so that they look and behave the way you would like in the interface.

GUIDE provides an interactive version of the MATLAB `get` and `set` commands that makes this task especially easy. To set property, value pairs for a graphics object in your layout, double-click the object in GUIDE. A Property Inspector for that object opens in a separate window. Along the left of the Property Inspector is a list of properties of the object selected (the equivalent of a `get` command). Along the right of the Property Inspector is a list of current values. You can change a value by clicking it and then editing it or selecting a new value from a pop-up menu (the equivalent of a `set` command). Closing the Property Inspector sets the values for the object.

One property that is important to set for every graphics object is the `Tag`. This property has no effect on the appearance or operation of the object in the interface, but it is used by GUIDE to identify the object. In particular, when the GUI is activated, the handle of the object is automatically stored in a structure array called `handles`, with a field name given by `Tag`, and a callback "stub" (subfunction with no code) is created in the GUI M-file, with a name that is also given by `Tag`.

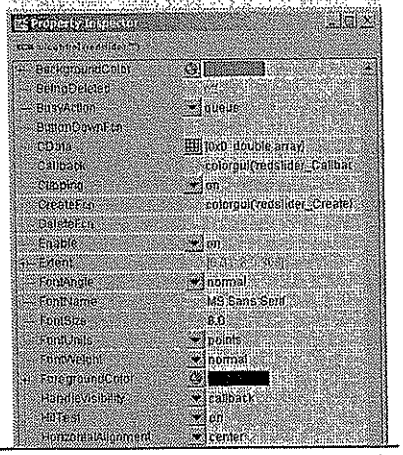
It is also important to set the `HandleVisibility` property of your graphics objects. Setting this property to a value of `off` ensures that users will not be able to access the handle from the command prompt using the `findobj` command, or make an axes current for plotting from outside the GUI. (Handles can still be accessed with the `findall` command.) For axes and user interface controls, a better choice for most purposes is to set `HandleVisibility` to a value of `callback`. This setting keeps the handle invisible to graphics commands issued from outside of the GUI, but allows the handle to be accessed from the callbacks that control the operation of the GUI.

Try

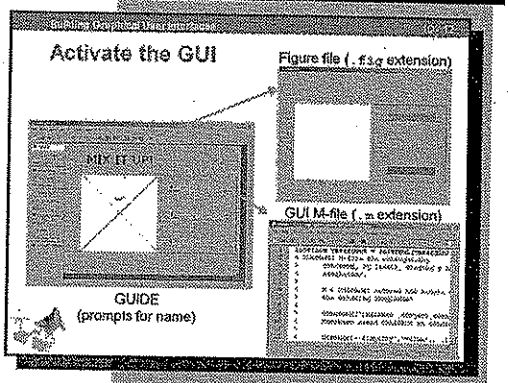
Set properties of the objects in the color mixer GUI using the Property Inspector.

Set:

1. The `String` of the static text box to `'MIX IT UP!'`
2. The `FontSize` of the static text box to 24.
3. The `Tag` of each object to a short, descriptive string.
4. The `HandleVisibility` of the axes and the sliders to `callback`.
5. The `BackgroundColor` of the sliders to red, green, and blue, respectively.
6. The `Max` and `Min` of the sliders to `1` and `0`, respectively.



Activate the GUI



When you have completed your layout, and set the values of properties of your graphics objects with the Property Inspector, you can proceed to activate the GUI. Choose **Tools → Run** from the menus at the top of GUIDE (or use the ▶ button in the top toolbar).

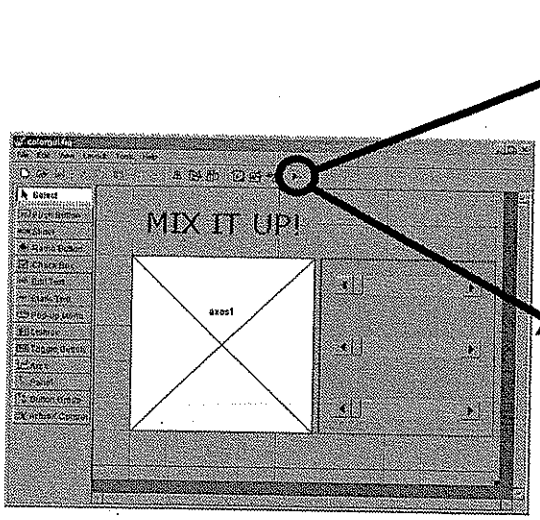
GUIDE prompts you for a name, and then automatically creates both a MATLAB figure file (with a `.fig` extension) and a GUI M-file (with a `.m` extension) using the name you provide. The figure file and the M-file are linked by the common name.

The figure file implements the layout you created in GUIDE, and serves as the user interface for the GUI. User interface controls are now operational in the sense that you can push buttons, slide sliders, select check boxes, etc. Of course, nothing changes in the GUI interface when you adjust the controls, because you have not yet written the callbacks that tell MATLAB how to respond to such events.

The GUI M-file opens in the MATLAB Editor. It contains a supervisory primary function for the GUI, and a sequence of subfunction callback stubs. To make the GUI operational, you have to fill in the callback stubs with the code that implements your design.

Try

Activate the layout of the color mixer GUI with **Tools → Run**.



GUIDE
(prompts for name)

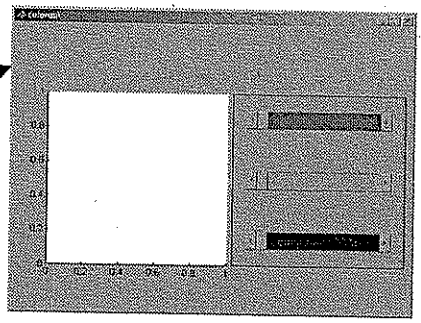
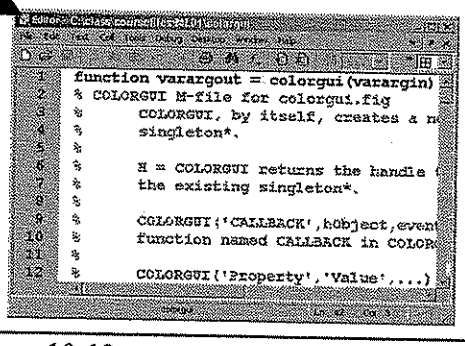
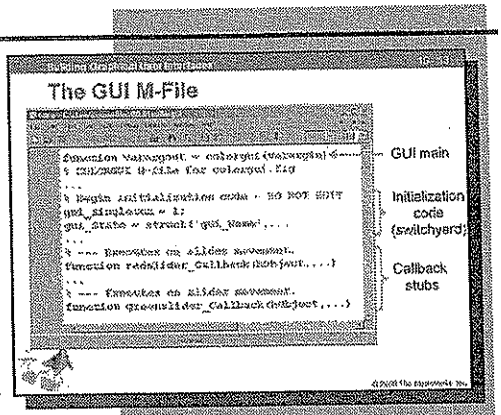


Figure file (`.fig` extension)



GUI M-file (`.m` extension)

The GUI M-File



Scroll through the M-file created by GUIDE in the MATLAB Editor. It has a characteristic structure used in all GUI M-files.

The M-file is supervised by the *primary* or *main* function at the top of the file, with the name you chose when you activated your GUI. The code for the primary function is automatically generated by GUIDE. When the GUI is opened, and every time there is an event in the interface, MATLAB calls this code with an input that identifies the event. The code then serves as a *switchyard*, directing execution to the appropriate callback in the rest of the M-file. *Do not edit this code.*

Callbacks are implemented as subfunctions, called from the primary. As you scroll through the M-file, you see placeholder callback stubs for each of the user interface controls that you added to your layout in GUIDE. The names of the callbacks are `Tag_Callback`, where `Tag` is the string assigned by the Property Inspector in GUIDE.

GUIDE also creates callback stubs for events other than changes to the user interface controls. The `filename_OpeningFcn` is called when the GUI opens, and can be used set initial values of controls, create initial plots, etc. The `filename_OutputFcn` is called if the primary is called with an output assignment; it determines the value assigned. User interface control callbacks are preceded by a `Tag_CreateFcn` callback, called each time the control appears on the interface, either when the GUI opens or if the control is made to appear by other events.

Every callback is passed an input argument named `handles`, created in the initialization code. `handles` is a structure array with field names given by the `Tag` of every graphics object in the GUI. It is the most convenient way to access the handles of objects in callback code.

You can add other utility subfunctions to the GUI M-file. This is especially useful if changes to various user interface controls result in the same sort of update to the interface. The update code can be collected in its own subfunction, and then called from individual callbacks.

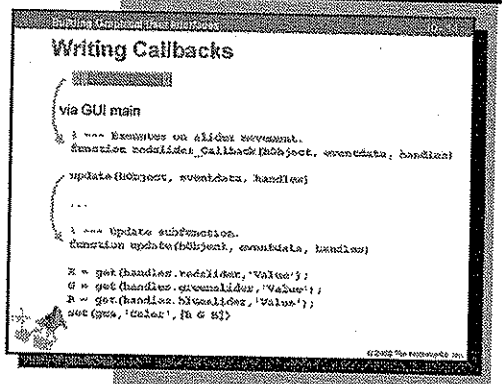
Try

Scroll through the M-file for the color mixer GUI generated by GUIDE.

```
>> edit colorgui_0
```

```
function <primary>

   initialization
       code
   (switchyard)

function <callback>

function <callback>

function <callback>

function <utility>
```

Writing Callbacks



The final step in creating your GUI is to write the callbacks that tell MATLAB how to update the interface each time a user interface control is changed. Write your callbacks in the space provided by the callback stubs generated by GUIDE.

Callbacks often have a similar structure:

1. Gather information on the current state of the interface with a series of `get` commands.
2. Compute something based on the current information.
3. Update the interface with a series of `set` and plotting commands.

For the color mixer GUI, the callbacks are especially simple. Each of the three sliders behaves identically. If the position of one of the sliders is changed, its callback should get its new `Value`, get the `Value` of each of the other two sliders, and then set the `Color` of the axes object accordingly. No intermediate computations are required.

In code,

```
R = get(handles.redslider,'Value');
G = get(handles.greenslider,'Value');
B = get(handles.blueslider,'Value');
set(gca,'Color',[R G B])
```

Because each slider has identical behavior, it is convenient to put the code in a utility subfunction and call it from the individual callbacks.

via GUI main

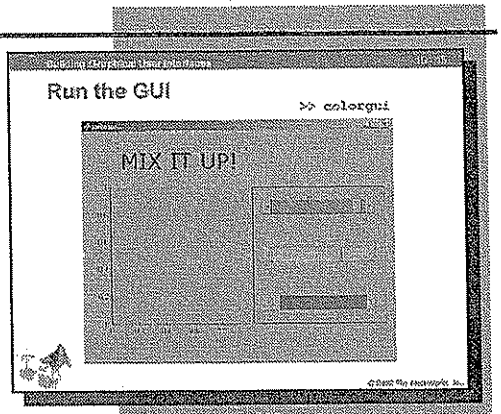
```
% --- Executes on slider movement.
function redslider_Callback(hObject, eventdata, handles)

update(hObject, eventdata, handles)

...

% --- Update subfunction.
function update(hObject, eventdata, handles)

R = get(handles.redslider,'Value');
G = get(handles.greenslider,'Value');
B = get(handles.blueslider,'Value');
set(gca,'Color',[R G B])
```

Try

Add callback code to the M-file for the color mixer GUI.

```
>> edit colorgui
```

Run the GUI

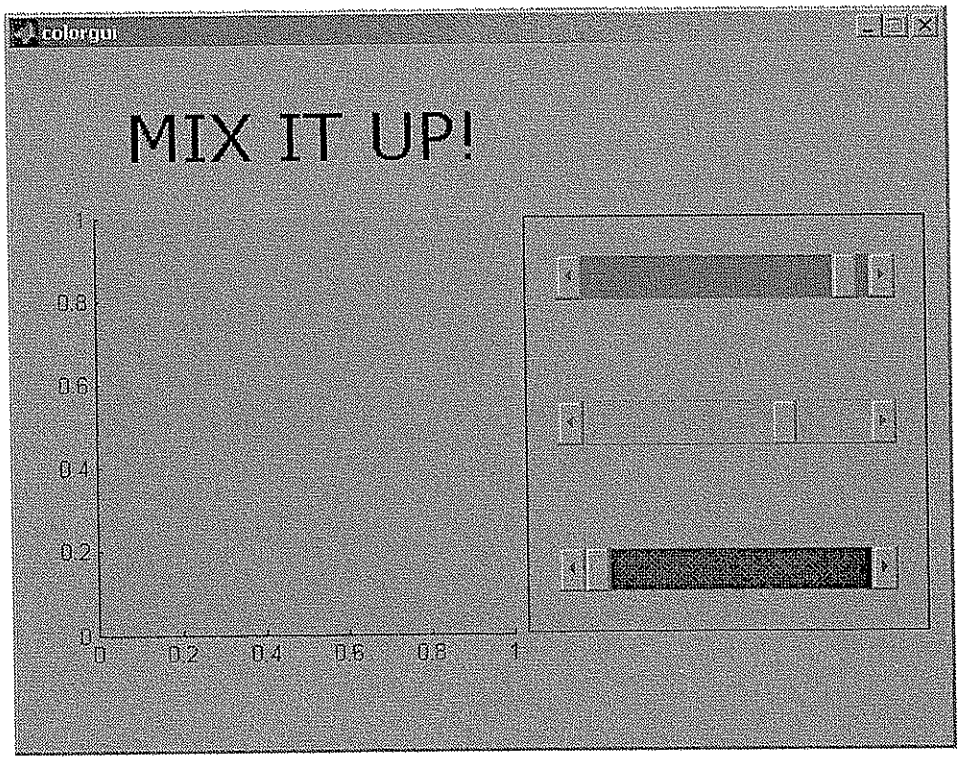


To run the GUI, you do not need to have the layout open in GUIDE or the M-file open in the MATLAB Editor. You only need to make sure that the GUI M-file and the GUI figure file are somewhere on the MATLAB search path.

Open the GUI by typing the name of the M-file (without the `.m` extension) at the command prompt. For example,

```
>> colorgui
```

opens the color mixer GUI interface. This is how a user would access the GUI. The M-file and its code are hidden from view.

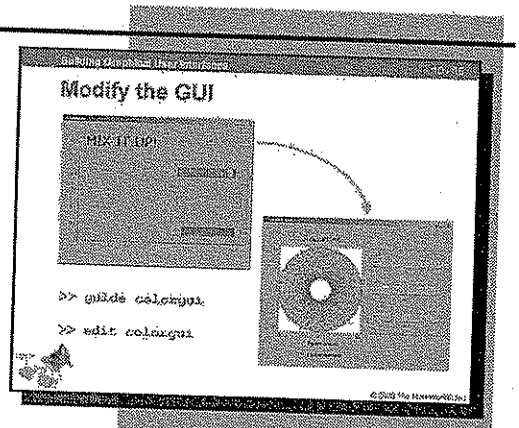


Try

Run the color mixer GUI:

```
>> colorgui
```

Modify the GUI



You can easily modify both the layout and operation of your GUI after you have created it.

For example, to modify the layout of the color mixer GUI, type

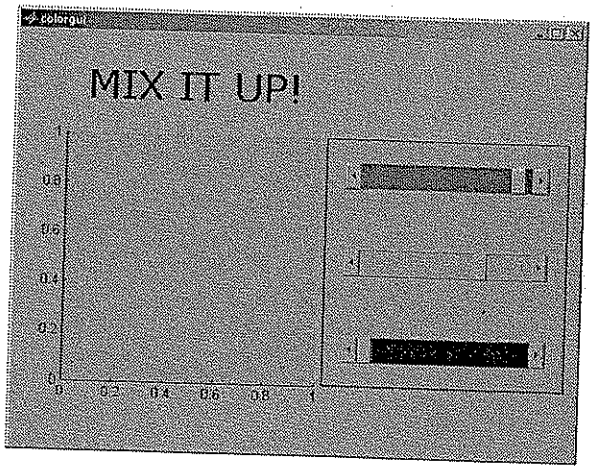
```
>> guide colorgui
```

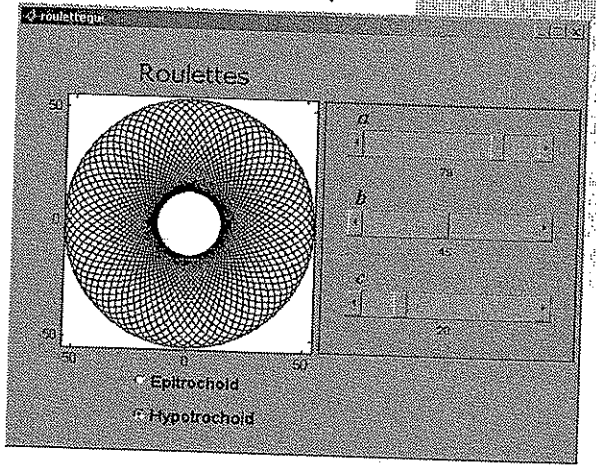
The layout opens in GUIDE. Make the modifications you would like, and then save the figure file by choosing **File → Save** from the menus at the top of GUIDE (or pressing the button in the toolbar). If you add new user interface controls, reactivate the GUI by choosing **Tools → Run** from the menus at the top of GUIDE (or use the button in the toolbar). This reactivation adds a new callback stub to the GUI M-file. If you remove a user interface control, you have to delete the existing callback by opening the GUI M-file in the MATLAB Editor.

To open the M-file for the color mixer GUI in the Editor, type

```
>> edit colorgui
```

Edit the callbacks to modify the operation of the GUI.





Try

Modify the color mixer GUI to look and behave like the orbital gears GUI introduced at the beginning of this chapter.

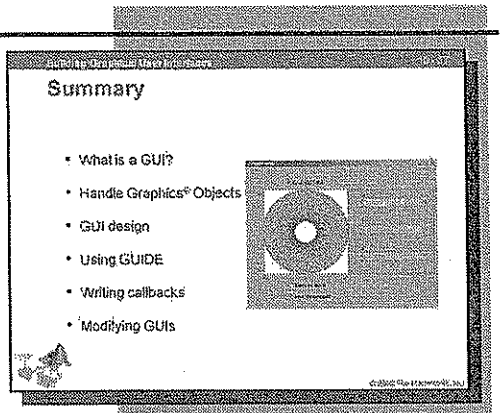
```
>> guide colorgui
>> edit colorgui
```

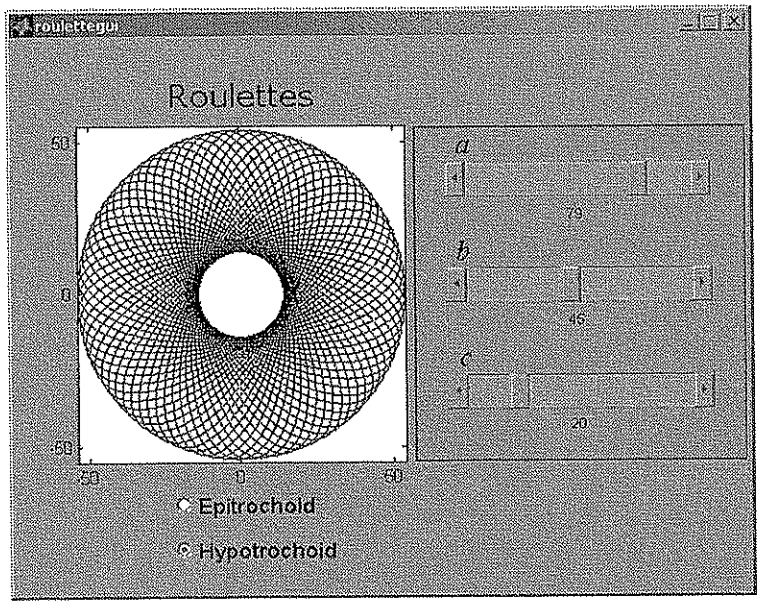
Parametric equations for the trochoid curves traced by orbital gears can be found at

```
www.math.duke.edu
/education/ccp
/materials/mvcalc
/spirograph/
```

Summary

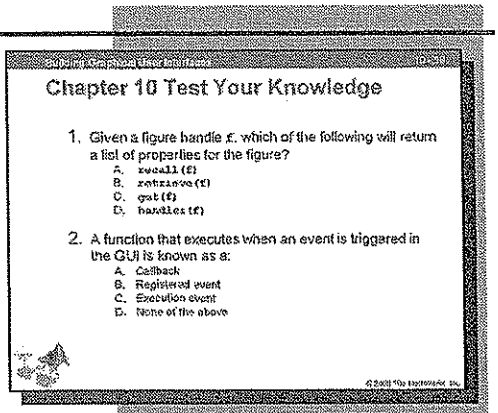
- What is a GUI?
- Handle Graphics® Objects
- GUI design
- Using GUIDE
- Writing callbacks
- Modifying GUIs





This page intentionally left blank

Chapter 10 Test Your Knowledge



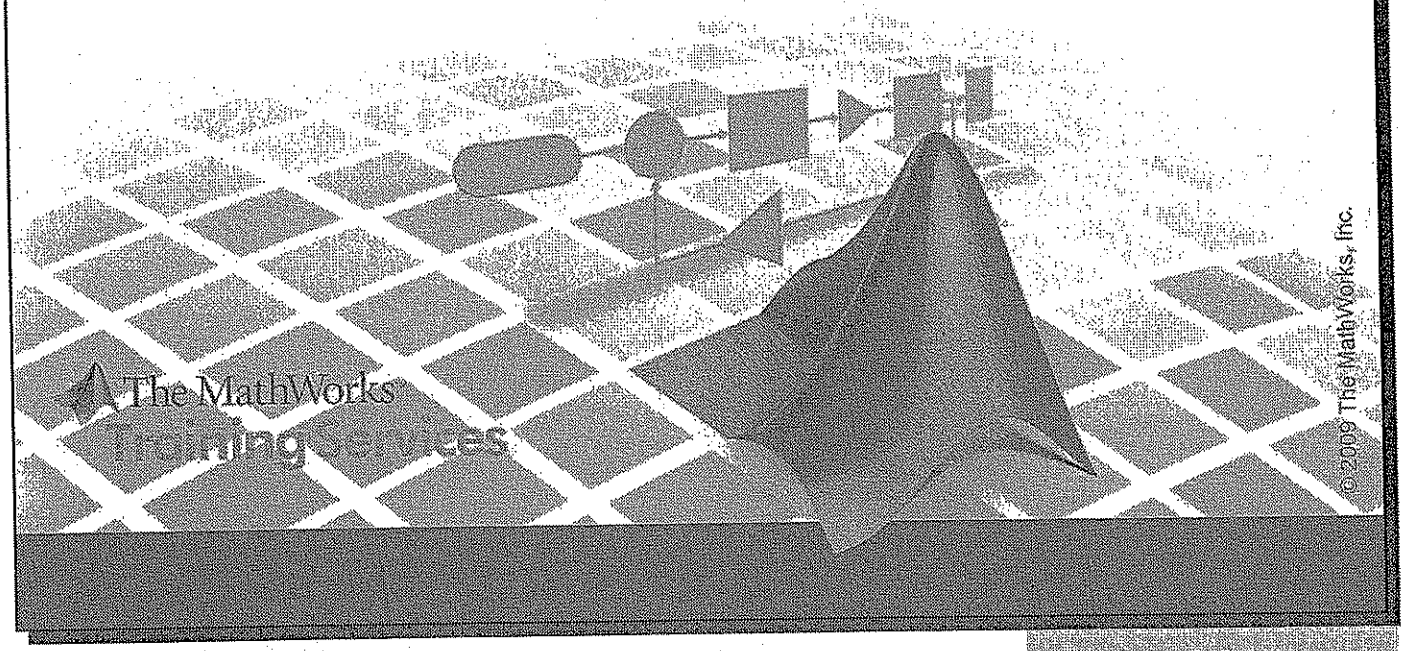
Name: _____

1. Given a figure handle `f`, which of the following will return a list of properties for the figure?
   A. `recall(f)`
   B. `retrieve(f)`
   C. `get(f)`
   D. `handles(f)`

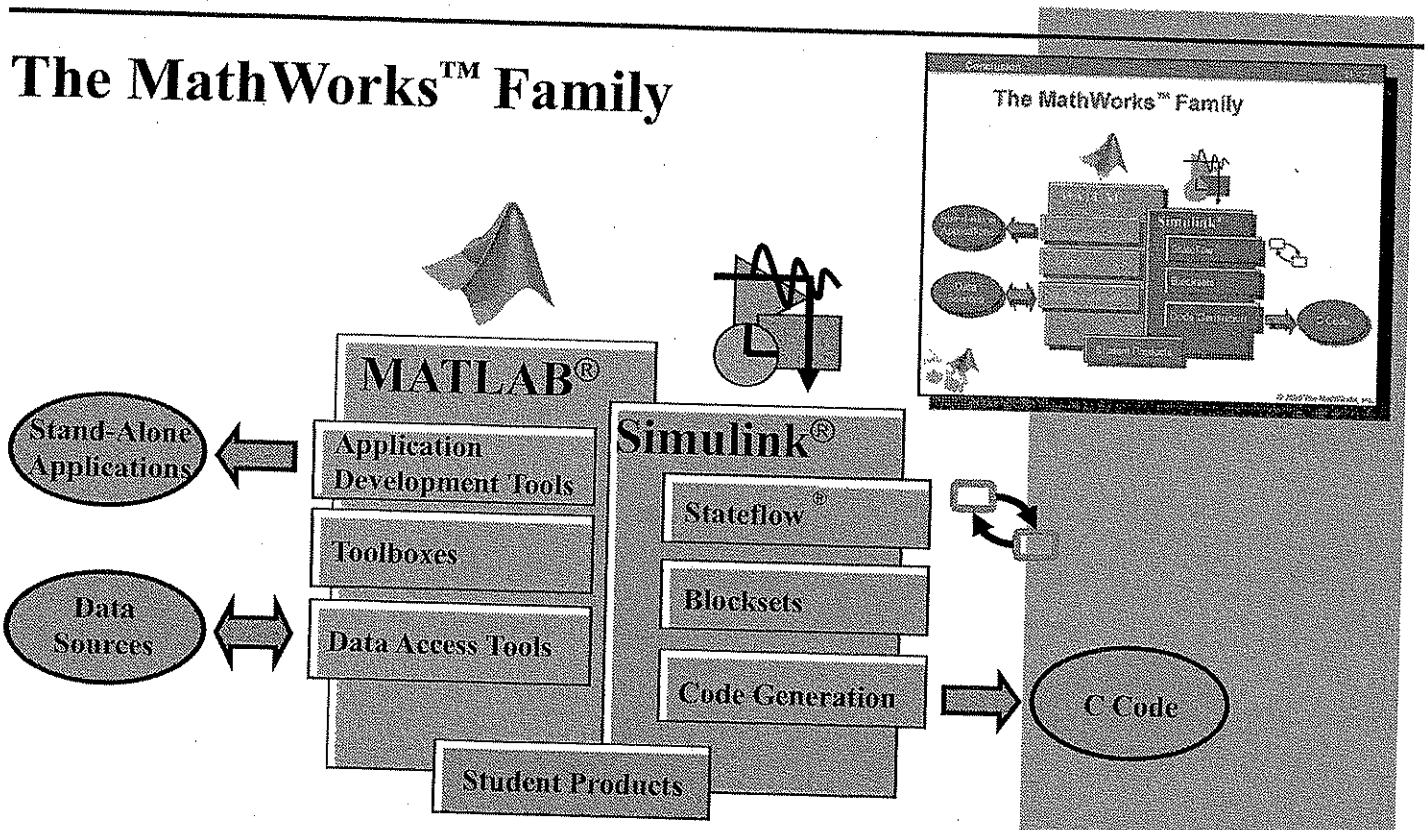
2. A function that executes when an event is triggered in the GUI is known as a:
   A. Callback
   B. Registered event
   C. Execution event
   D. None of the above

MATLAB® Fundamentals

Conclusion



The MathWorks™ Family



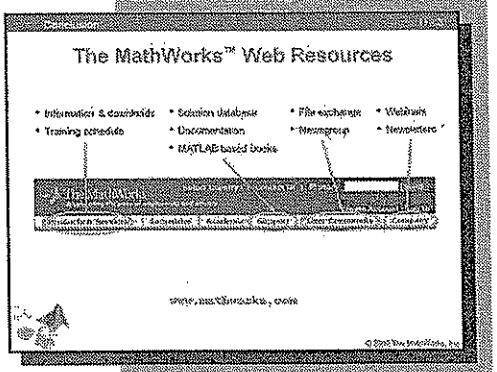
Key Characteristics of the MATLAB® Language

- 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



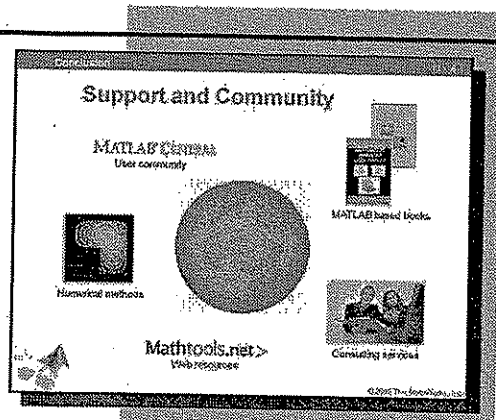
The MathWorks Web site at `www.mathworks.com` contains a wealth of resources beyond the materials provided for this course.*

- 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

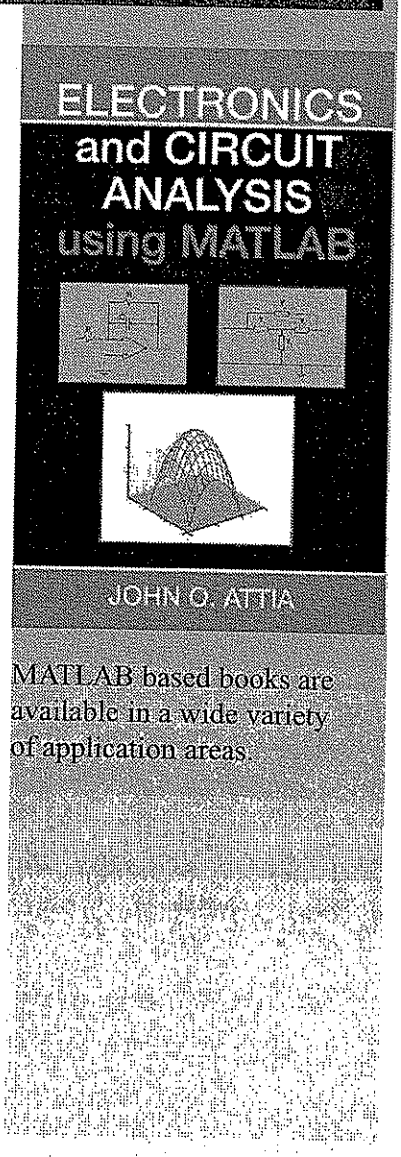


You are connected to a world of creative MATLAB users, both inside and outside of The MathWorks, for information, support, and community.

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.



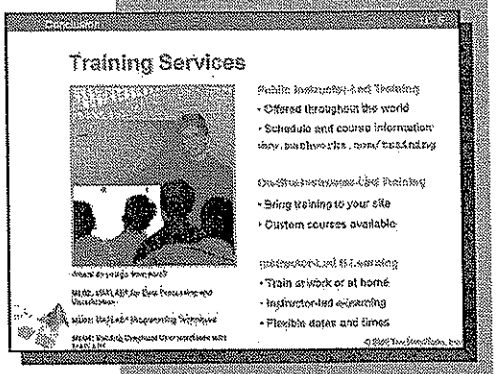
MATLAB based books are available in a wide variety of application areas.

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?

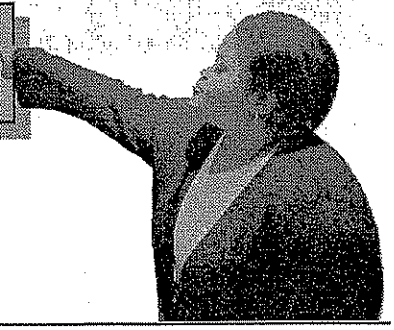
Related courses

ML02: MATLAB® for Data Processing and Visualization

ML03: MATLAB® Programming Techniques

ML04: Building Graphical User Interfaces with MATLAB®

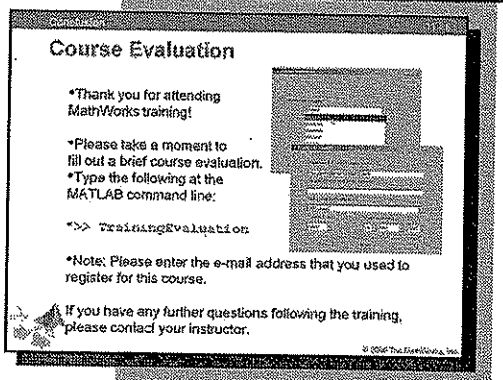
Want more training?



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



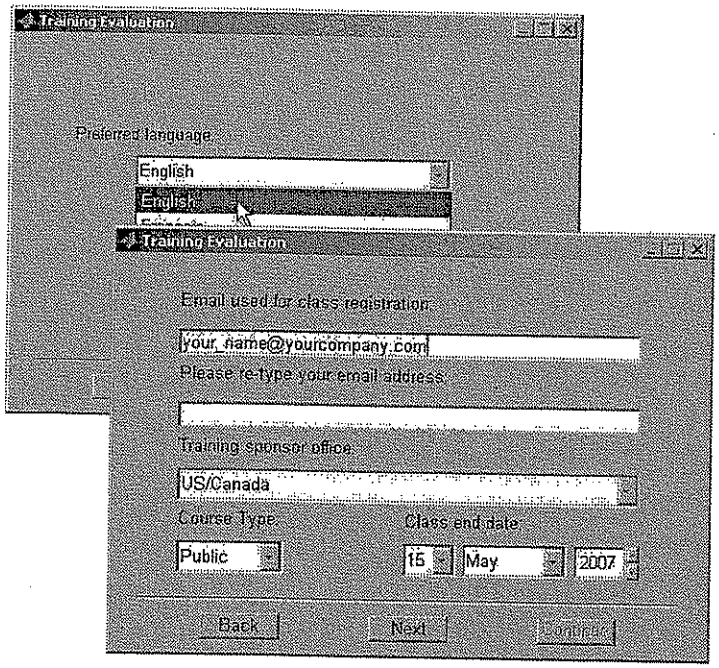
Thank you for attending MathWorks training!

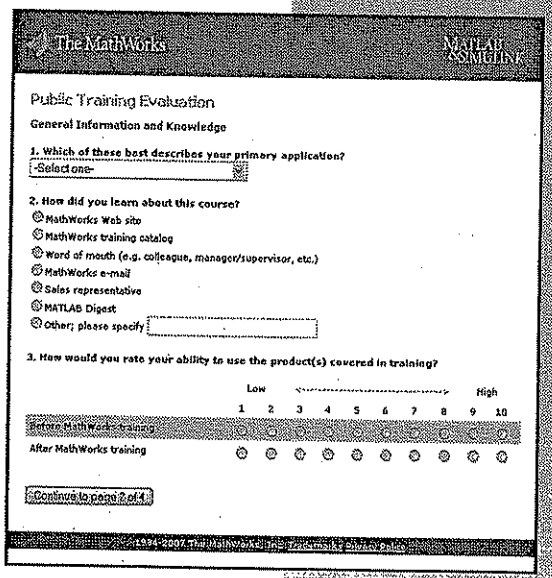
Please take a moment to fill out a brief course evaluation.

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.

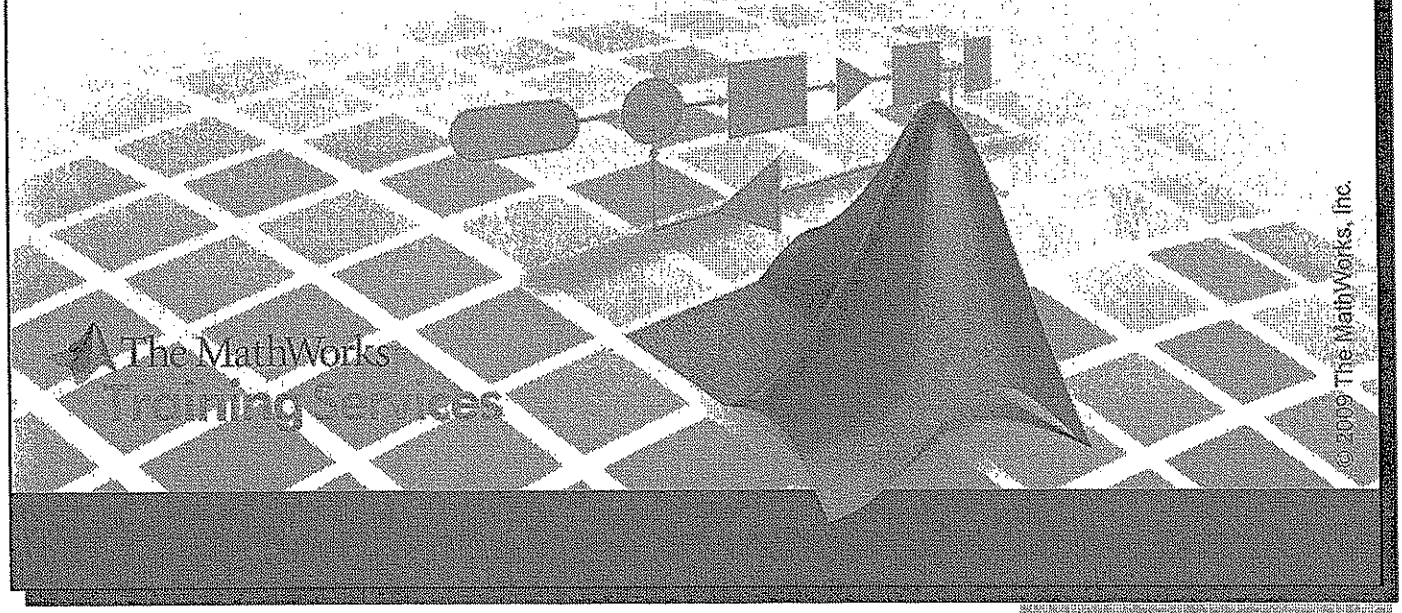
If needed, an alternate method of launching the evaluation is through the following URL:

```
http://www.mathworks.com/services/training/eval
```

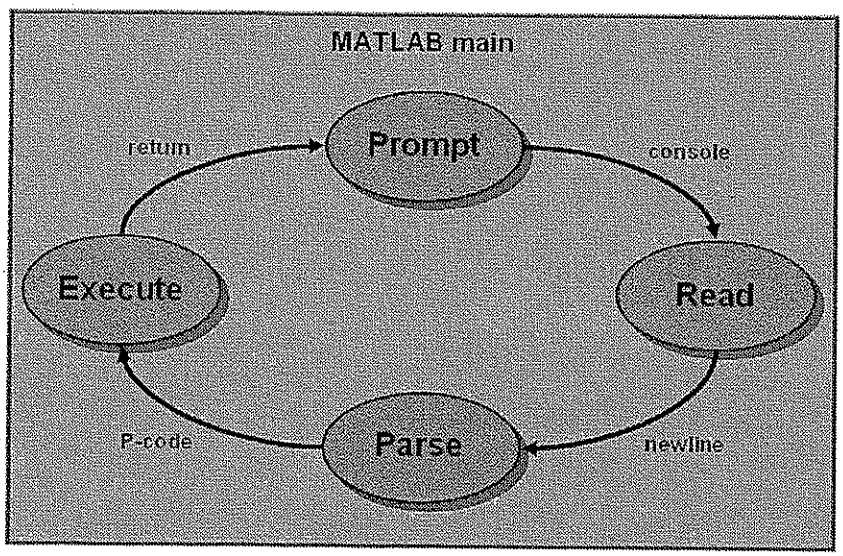
The following is the information you will need to fill in the form at the above URL:
Training Type: Public, or On-Site
Preferred Language: EN (English), FR (French), DE (German), IT (Italy), KO (Korean), ES (Spanish), ZH (Simplified Chinese) Training Office: AU (Australia), BNL (Benelux), CH (Switzerland), 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
E-mail Address: [e-mail address you used to register for the course]

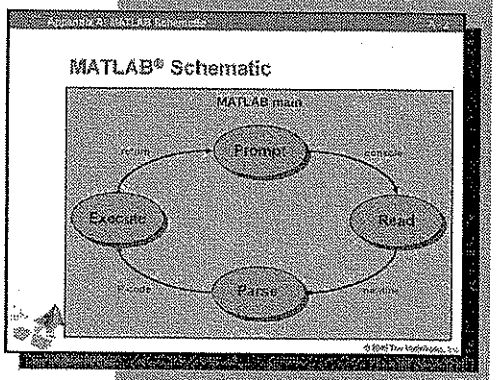
MATLAB® Fundamentals

Appendix A: MATLAB® Schematic



MATLAB® Schematic






```
>> command1
>> command2
>> command3
```

MATLAB® executes commands by cycling through a sequence of tasks and calling on a variety of specialized subroutines. The above schematic is a high-level, much-simplified view of the process.

The flow from MATLAB command to machine execution depends on whether the command is:
- At the command prompt
- In an M-file script
- In an M-file function

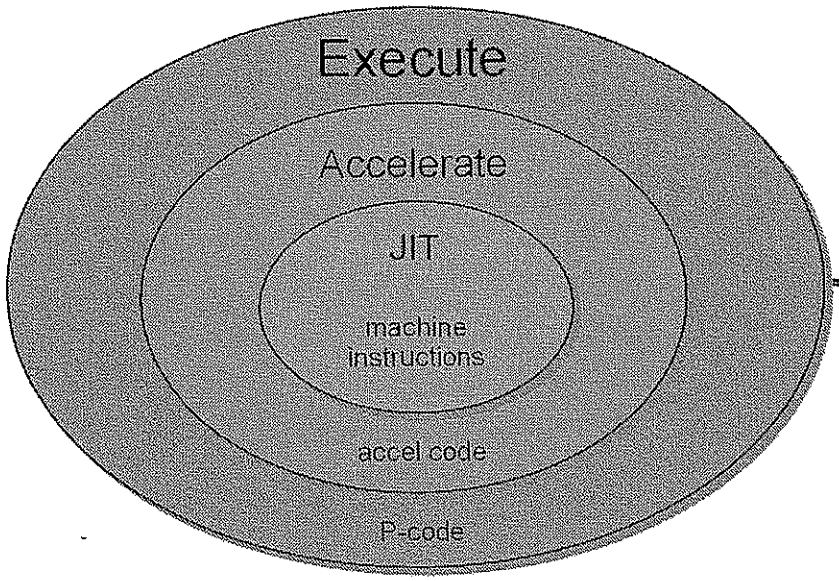
Each situation is handled differently. For M-files, MATLAB considers overall file structure as well as individual commands.

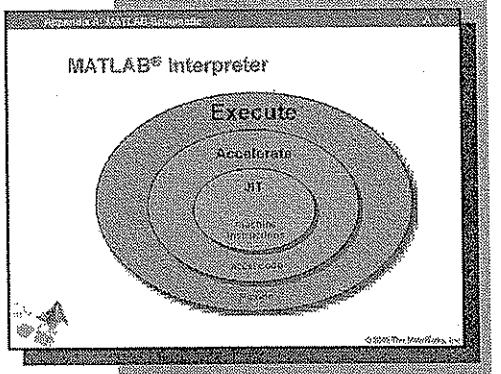
Prompt and Read: The MATLAB Command Line Processor supervises tasks at the console such as line editing, syntax coloration, tab completion, recognition of multiple lines, etc.

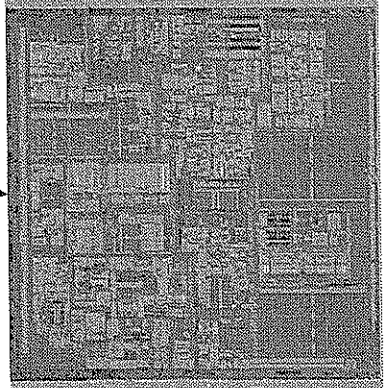
Parse: The MATLAB Parser determines if commands are well-formed MATLAB expressions, parses their syntactic structure, optimizes the parse tree, and expresses the instructions in P-code.

Execute: P-code is platform-independent pseudocode executed on a virtual MATLAB machine. The virtual machine is given a platform-specific implementation within the MATLAB Interpreter.

MATLAB® Interpreter







The MATLAB Interpreter is responsible for executing P-code.

MATLAB stores P-code for M-file functions in cache memory. Storing in cache avoids reparsing with each call. P-code remains in memory until it is cleared using the `clear` command, or until MATLAB quits. For repeated calls to large files, this can improve speed.

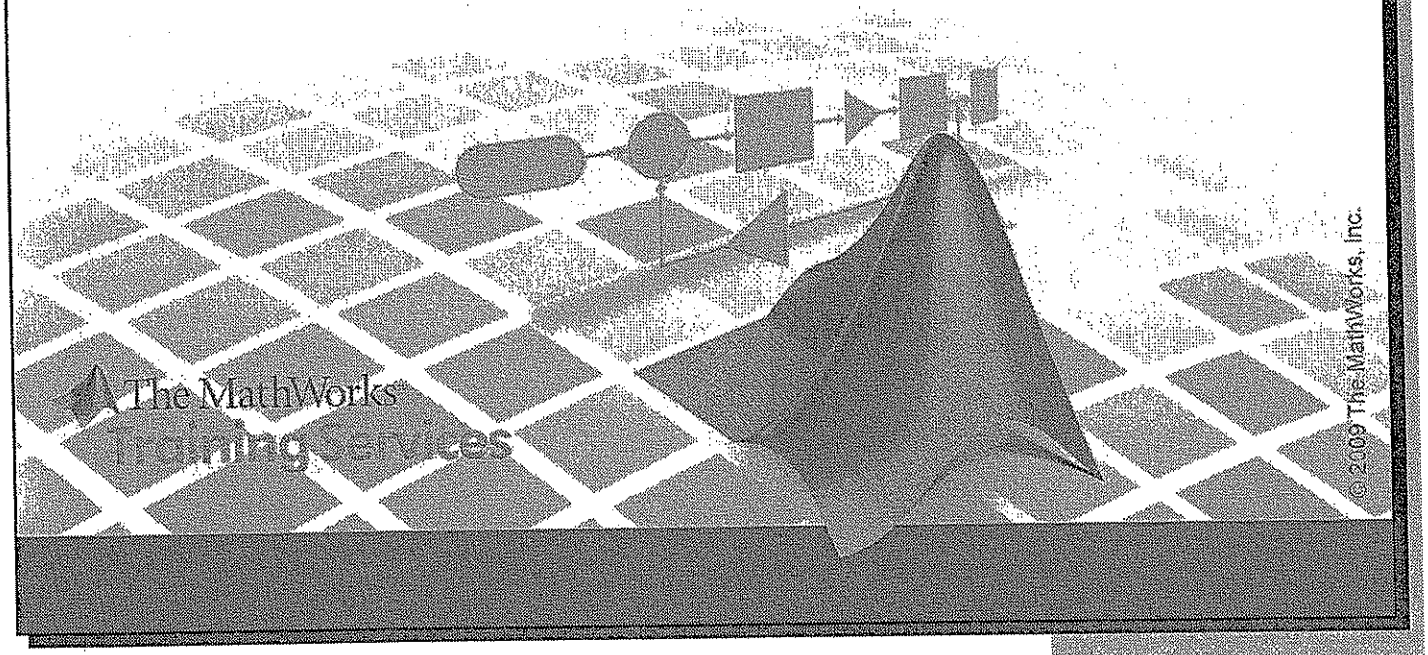
The MATLAB Dispatcher calls functions at the service of the Interpreter, both of which apply MATLAB precedence rules.

An important benefit of the MATLAB language is its dynamic typing and sizing of variables, without declarations. This analysis can be time-consuming, however, when done repeatedly (in, e.g., loops). The MATLAB Accelerator analyzes variables once, stores the information in accel code, and subsequently looks only for changes. If types change, code is regenerated, but typically the Interpreter's overhead is significantly reduced. The JIT Compiler assists the Accelerator by examining the incoming P-code and converting certain operations directly into machine instructions.

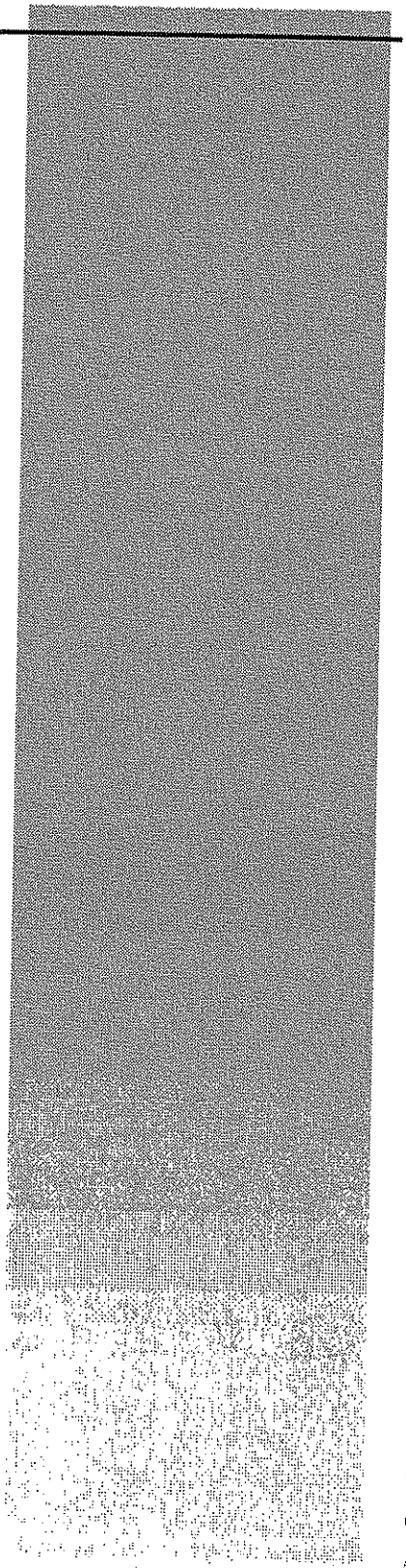
The proportion of P-code optimized by the Accelerator and the JIT Compiler grows with each MATLAB release.

MATLAB® Fundamentals

Appendix B: MATLAB® Reference



Development Environment



Command Window

`clc`	Clear Command Window
`format`	Set numeric format
`;`	Suppress output display
`...`	Continue to next line without execution
↑	Scroll through previously typed commands
`s`↑	Scroll through commands beginning with `s`
Esc	Clear command line
Home, **End**	Move to beginning, end of expression
Tab	Complete partial expressions
!	Invoke operating system

Files and directories

`pathtool`	Set search path
`which`	Locate files on search path
`matlabroot`	Locate MATLAB® installation directory
`pwd`	Identify current directory
`cd`	Change current directory
`dir`, `ls` (UNIX™)	List current directory
`what`	List current directory by file type
`mkdir`	Make new directory
`rmdir`	Remove directory
`copyfile`	Copy file from source to destination
`movefile`	Move file from source to destination
`delete`	Delete file
`type`	Display text file
`echo`	Display M-file during execution
`more`	Display paged output
Ctrl+C	Return control to the command line

Help

`help`	Display help information
`doc`	Open documentation
`demo`	Application-specific demos
`lookfor`	Search help by keyword
`support`	Open tech support site
`whatsnew`	Product release notes
`ver`	Version information

Variables

General

`clear`	Remove variables from workspace
`who`	Display workspace variables
`whos`	Display workspace variables (verbose)
`size`	Variable dimensions
`length`	Longest dimension
`linspace, :`	Create vectors
`meshgrid`	Create grid from vectors
`.'`	Transpose
`'`	Conjugate transpose
`horzcat, [ , ]`	Horizontal concatenation
`vertcat, [ ; ]`	Vertical concatenation
`cat`	Concatenate along specified dimension
`reshape`	Change dimensions
`repmat`	Replicate and tile
`=`	Assignment
`(),{},.`	Subscripted assignment and reference
`global`	Declare global variables

Workspace allocation

`zeros`	All elements `0` (any numerical type)
`ones`	All elements `1` (any numerical type)
`cell`	Preallocate for cell array
`struct`	Preallocate for structure array
`pack`	Consolidate workspace memory

Random numbers

`rand`	Uniformly distributed random numbers
`randn`	Normally distributed random numbers
`randperm`	Random permutation of integers

Operators and Constants

Arithmetic operators

`plus`	Plus	+
`uplus`	Unary plus	+
`minus`	Minus	-
`uminus`	Unary minus	-
`mtimes`	Matrix multiplication	*
`times`	Array multiplication	.*
`mpower`	Matrix power	^
`power`	Array power	.^
`mldivide`	Backslash or left matrix divide	\
`mrdivide`	Slash or right matrix divide	/
`ldivide`	Left array divide	.\
`rdivide`	Right array divide	./

Relational operators

`eq`	Equal	==
`ne`	Not equal	~=
`lt`	Less than	<
`gt`	Greater than	>
`le`	Less than or equal	<=
`ge`	Greater than or equal	>=

Logical operators

`and`	Element-wise AND	&
`and`	Short-circuit AND	&&
`or`	Element-wise OR	|
`or`	Short-circuit OR	||
`xor`	Exclusive OR	
`not`	Negation	~
`true`	Logical 1	
`false`	Logical 0	
`any`	Test for any nonzero elements	
`all`	Test for all nonzero elements	
`bitand`	Bitwise AND	
`bitcmp`	Complement bits	
`bitor`	Bitwise OR	
`bitxor`	Bitwise XOR	
`bitshift`	Bitwise shift	

Set operators

`union`	Set union
`intersect`	Set intersection
`setdiff`	Set difference
`setxor`	Set exclusive OR
`ismember`	Test for membership
`unique`	Find nonrepeated values

Constants

`i,j`	Imaginary unit
`pi`	Trigonometric unit
`exp(1)`	Natural logarithmic unit (e)
`Inf`	IEEE® Infinity
`NaN, nan`	IEEE® Not-a-Number
`eps`	Floating-point relative accuracy
`realmax`	Largest positive floating-point number
`realmin`	Smallest positive floating-point number
`intmax`	Largest value of specified integer type
`intmin`	Smallest value of specified integer type

Mathematical Operations

Trigonometry

`sin`	Sine
`sind`	Sine (degrees)
`asin`	Inverse sine
`sinh`	Hyperbolic sine

} Similarly for other trigonometric functions

Exponentials and logarithms

`sqrt`	Square root
`nthroot`	Nth root
`exp`	Natural exponential
`pow2`	Power of 2
`log`	Natural logarithm
`log2`	Base 2 logarithm
`log10`	Base 10 logarithm

Complex numbers

`real`	Real part
`imag`	Imaginary part
`abs`	Modulus
`angle`	Phase angle
`conj`	Conjugate

Rounding

`fix`	Round toward zero
`floor`	Round toward `-Inf`
`ceil`	Round toward `Inf`
`round`	Round toward nearest integer
`mod`	Modulus after division
`rem`	Remainder after division
`rat, rats`	Rational fraction approximation

Discrete

`perms`	All permutations
`nchoosek`	All combinations (binomial coefficients)
`factorial`	Factorial
`lcm, gcd`	LCM, GCD
`factor`	Prime factors
`primes, isprime`	List primes, test for primality

Statistical Operations

Descriptive statistics

`sort`	Sort data
`max`	Maximum value
`min`	Minimum value
`mean`	Average value
`std`	Standard deviation
`var`	Variance
`median`	Median
`cumprod`	Cumulative product
`cumsum`	Cumulative sum

Interpolation and curve fitting

`spline`	Cubic spline interpolation
`pchip`	Piecewise cubic Hermite interpolation
`interp1`	General vector data interpolation
`interp2`	General matrix data interpolation
`griddata`	Interpolate irregularly spaced data
`polyfit`	Polynomial curve fit
`polyval`	Polynomial interpolation
`roots`	Polynomial roots
`\`	Arbitrary linear model fit

Covariance and correlation

`cov`	Covariance
`corrcoef`	Correlation coefficient

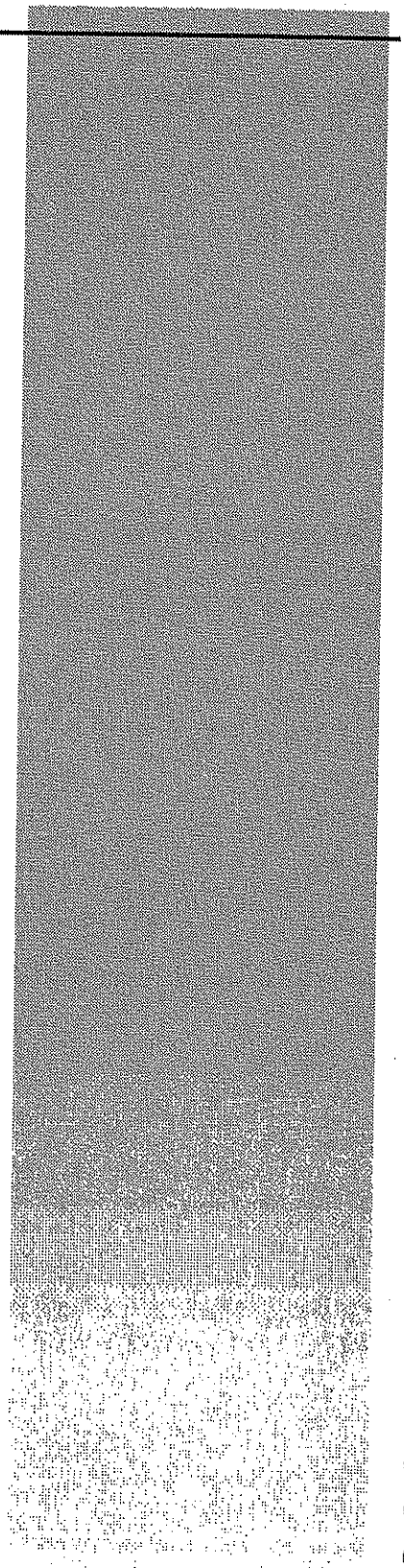
Convolution

`conv`	Convolution
`conv2`	Two-dimensional convolution
`convn`	N-dimensional convolution

Fourier transforms

`fft`	Discrete Fourier transform
`fftshift`	Rearrange `fft` for 0-centered spectrum
`ifft`	Inverse discrete Fourier transform

Plotting



Vector data

`plot, plot3`	Line plots
`plotyy`	Line plots with two vertical scales
`comet, comet3`	Animated line plots
`scatter, scatter3`	Scatter plots
`bar, bar3, bar3h`	Bar plots
`hist`	Histogram
`rose`	Angle histogram
`polar`	Polar coordinate plot
`stem, stem3`	Stem plots
`stairs`	Stairstep plot
`area`	Filled area plot
`pie, pie3`	Pie charts
`compass`	Radial vector plot
`feather`	Horizontal vector plot
`quiver, quiver3`	Vector field plots
`triplot`	Triangle plot
`fill`	Filled polygon plot
`voronoi`	Voronoi diagram
`convhull`	Convex hull

Matrix data

`surf, surfc, surfl`	Surface plots
`surfnorm`	Surface normals
`cylinder`	Cylinder plot
`sphere, ellipsoid`	Sphere, ellipsoid plot
`mesh, meshc, meshz`	Wire-frame surface plots
`waterfall`	Row mesh plot
`ribbon`	Column ribbon plot
`trimesh, tetramesh`	Triangle, tetrahedron mesh plots
`contour, contourf, contour3, contourc`	Contour plots
`image, imagesc`	Images
`bar, bar3, bar3h`	Matrix bar plots
`plotmatrix`	Matrix scatter plots
`fill3`	3D Filled polygon plot
`patch`	Patch object
`spy`	Sparsity pattern

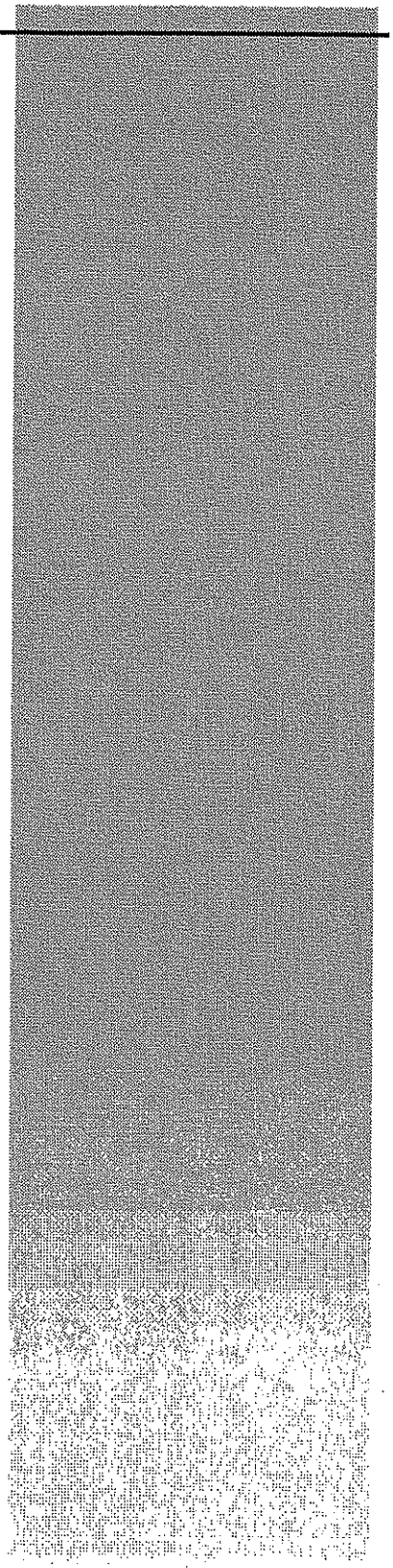
Volume data

`slice`	Slice plot
`contourslice`	Slice plot with contours
`isosurface`	Isosurface plot
`quiver3`	Vector field plot
`coneplot`	Vector field plot with cone markers
`streamline`	Vector field streamline plot
`streamribbon`	Vector field streamline and curl plot

Animation

`for`	For loop
`while`	While loop
`pause`	Pause for framerate
`getframe`	Capture movie frame
`movie`	Play recorded movie
`movie2avi`	Convert movie to AVI file
`frame2im`	Convert movie frame to image
`im2frame`	Convert image to movie frame

Plot Formatting



Figure

`figure`	Create or bring forward figure
`refresh`	Refresh figure
`clf`	Clear figure
`close`	Close figure

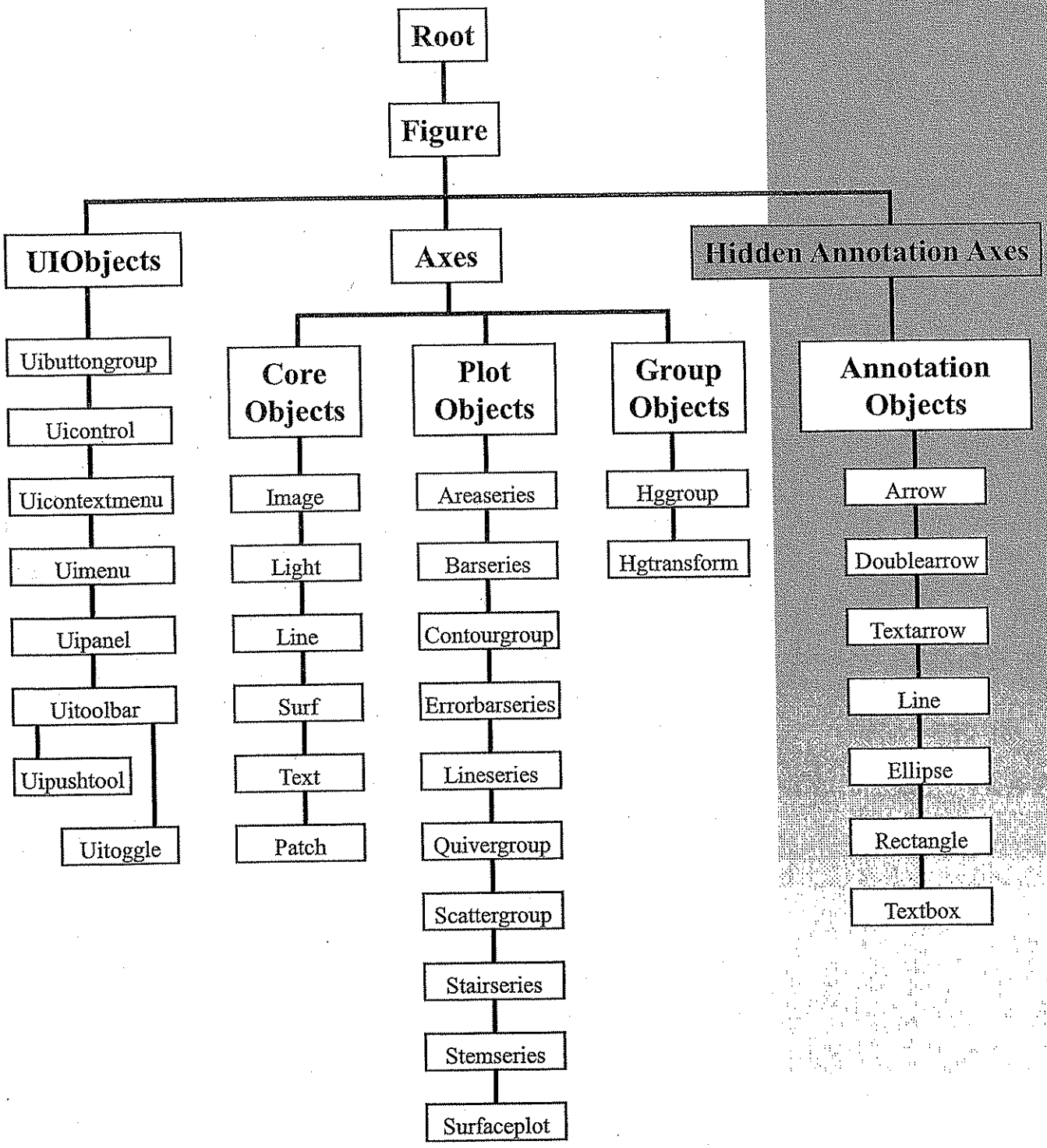
Axes and scales

`hold`	Retain/reset current axes properties
`axis`	Modify axes properties
`subplot`	Create subplots
`linkaxes`	Synchronize axes limits
`pan`	Turn panning on or off
`loglog`	Log scales
`semilogx, semilogy`	Semilog scales
`grid`	Add grid lines
`zoom`	Turn zooming on or off
`plottools`	Start plotting tools

Annotation

`xlabel, ylabel, zlabel`	Axes labels
`title`	Figure title
`legend`	Multiple plot legend
`text`	Text annotation
`gtext`	Place text annotation with mouse
`annotation`	Annotation objects
`texlabel`	T_EX format from character string

Graphics Hierarchy



Characters and Strings

Characters

`char`	Convert numbers to ASCII characters
`double`	Convert ASCII characters to numbers
`native2unicode`	Convert numbers to Unicode® characters
`unicode2native`	Convert Unicode® characters to numbers

Strings

`' '`	Create character string
`strvcat`	Create string matrix (without empty strings)
`str2mat`	Create string matrix (with empty strings)
`strcat, [ , ]`	Concatenate strings horizontally
`num2str, mat2str`	Convert numbers to strings
`str2num`	Convert strings to numbers
`cellstr`	Create cell array of strings
`deblank`	Strip trailing blanks
`strtrim`	Remove leading and trailing whitespace
`lower`	Convert string to lowercase
`upper`	Convert string to uppercase
`strjust`	Justify character array
`date`	Current date string
`datestr`	Convert date number to string
`datenum`	Convert date string to number
`strcmp`	Compare strings
`strcmpi`	Compare strings, ignoring case
`strncmp`	Compare `n` string characters
`strncmpi`	Compare `n` string characters, ignoring case
`strfind`	Find string within another
`findstr`	Find string within another, longer string
`strrep`	Find and replace substring
`strtok`	Return selected parts of string
`regexp`	Match regular expression
`regexpi`	Match regular expression, ignoring case
`regexprep`	Replace string using regular expression
`sscanf`	Read string under format control
`sprintf`	Write formatted data to string
`ischar, isletter`	Test for character arrays, alphabetic letters
`disp`	Display string
`eval`	Execute MATLAB expression string

T_EX Characters

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\alpha	α	\upsilon	υ	\sim	~
\beta	β	\phi	φ	\leq	≤
\gamma	γ	\chi	χ	\infty	∞
\delta	δ	\psi	ψ	\clubsuit	♣
\epsilon	ε	\omega	ω	\diamondsuit	♦
\zeta	ζ	\Gamma	Γ	\heartsuit	♥
\eta	η	\Delta	Δ	\spadesuit	♠
\theta	θ	\Theta	Θ	\leftrightarrow	↔
\vartheta	ϑ	\Lambda	Λ	\leftarrow	←
\iota	ι	\Xi	Ξ	\uparrow	↑
\kappa	κ	\Pi	Π	\rightarrow	→
\lambda	λ	\Sigma	Σ	\downarrow	↓
\mu	μ	\Upsilon	Υ	\circ	°
\nu	ν	\Phi	Φ	\pm	±
\xi	ξ	\Psi	Ψ	\geq	≥
\pi	π	\Omega	Ω	\propto	∝
\rho	ρ	\forall	∀	\partial	∂
\sigma	σ	\exists	∃	\bullet	•
\varsigma	ς	\ni	∋	\div	÷
\tau	τ	\cong	≅	\neq	≠
\equiv	≡	\approx	≈	\aleph	ℵ
\Im	ℑ	\Re	ℜ	\wp	℘
\otimes	⊗	\oplus	⊕	\oslash	⊘
\cap	∩	\cup	∪	\supseteq	⊇
\supset	⊃	\subseteq	⊆	\subset	⊂
\int	∫	\in	∈	\o	º
\rfloor	⌋	\lceil	⌈	\nabla	∇
\lfloor	⌊	\cdot	·	\ldots	...
\perp	⊥	\neg	¬	\prime	'
\wedge	∧	\times	x	\0	∅
\rceil	⌉	\surd	√	\mid	|
\vee	∨	\varpi	ϖ	\copyright	©
\langle	⟨	\rangle	⟩		

T_EX Quick Reference

```
>> winopen(...
'TeXQR.pdf')
```

M-Files and Programming

M-files

`script`	Script M-file description
`run`	Run a script not on the path
`function`	Function declaration
`@`	Create function handle
`( , )`	Functional input arguments
`[ , ]`	Functional output arguments
`end`	Terminate nested function
`%`	Comment code
`...`	Continue expression on next line
`depfun`	List file dependencies
`depdir`	List dependent directories
`mfilename`	Name of M-file currently executing
`inputname`	Name of function input
`varargin`	Variable number of input arguments
`varargout`	Variable number of output arguments
`nargin`	Number of input arguments
`nargout`	Number of output arguments
`nargchk`	Check number of input arguments
`nargoutchk`	Check number of output arguments
`persistent, global`	Declare persistent, global variable
`echo`	Echo code to console during execution
`input`	Request user input
`pause`	Halt execution temporarily
`pcode`	Create preparsed pseudocode file (P-file)
`rehash`	Refresh file system path caches

Error Handling

`try`	Attempt block of code, catch errors
`catch`	Specify how to respond to an error in `try`
`error`	Display error message
`ferror`	Query for errors during file input or output
`lasterr`	Return last error message
`lasterror`	Last error message and related information
`rethrow`	Reissue error
`warning`	Display warning message
`intwarning`	Control state of integer warnings
`lastwarn`	Return last warning message

Keywords

`for`	Execute code a specified number of times
`while`	Execute code until a condition fails
`break`	Terminate execution of `for` or `while` loop
`continue`	Go to next iteration of `for` or `while` loop
`return`	Return to invoking function
`if`	Conditionally execute code
`elseif`	Conditionally execute other code
`else`	Execute remaining case (`if-else`)
`switch`	Declare switch variable
`case`	Execute code conditional on `switch`
`otherwise`	Execute remaining case (`switch-case`)
`end`	Terminate conditional block

Graphics programming

`get`	Query for property value
`set`	Modify property value
`findobj`	Obtain visible handles of described objects
`findall`	Obtain hidden handles of described objects
`gcf`	Handle of current figure window
`gca`	Handle of current axis
`gco`	Handle of current object
`0`	Handle of root object

Timing

`bench`	Time and compare hardware
`tic`	Start stopwatch timer
`toc`	Stop stopwatch timer
`profile`	Time and profile M-files
`timer`	Create timer object
`start`	Start timer object
`wait`	Wait for timer object to complete
`stop`	Stop timer object
`set`	Display or set timer object properties
`get`	Retrieve timer object properties
`delete`	Delete timer object from memory

Precedence

Interpreter precedence

1. Variable
2. Nested function
3. Subfunction
4. Private function
5. Class constructor
6. Overloaded method
7. File in the current directory
8. File on the path

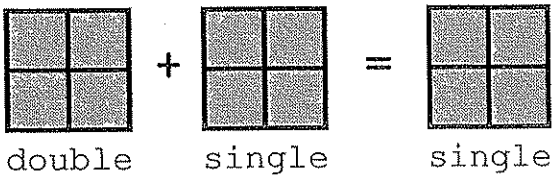
File precedence

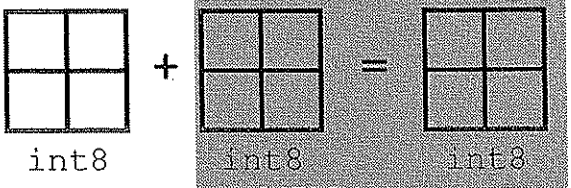
1. MEX-file
2. MDL-file (Simulink® model)
3. P-code file
4. Built-in file
5. M-file

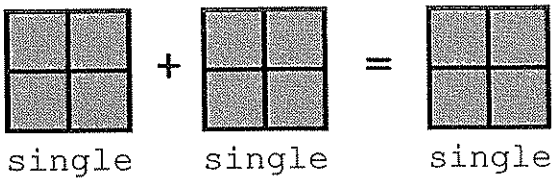
Operator precedence

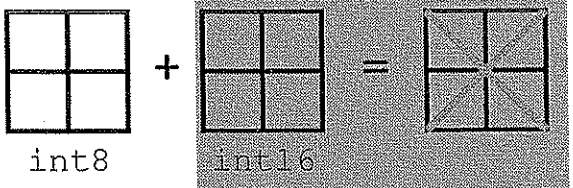
1. Parentheses ( )
2. Transpose (. '), power (. ^), complex conjugate transpose ('), matrix power (^)
3. Unary plus (+), unary minus (−), logical negation (~)
4. Multiplication (. *), right division (. /), left division(. \ ), matrix multiplication (*), matrix right division (/), matrix left division (\)
5. Addition (+), subtraction (−)
6. Colon operator (:)
7. Less than (<), less than or equal to (<=), greater than (>), greater than or equal to (>=), equal to (==), not equal to (~=)
8. Element-wise AND (&)
9. Element-wise OR (|)
10. Short-circuit AND (&&)
11. Short-circuit OR (||)

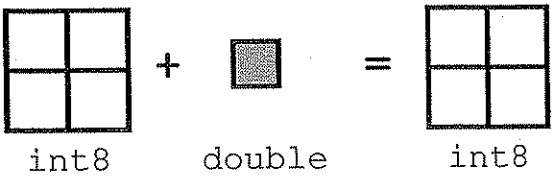
Nondouble Arithmetic

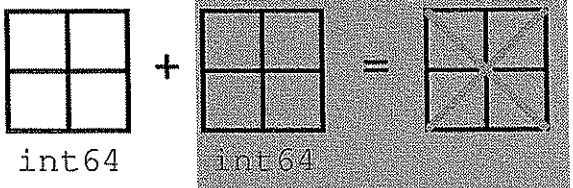


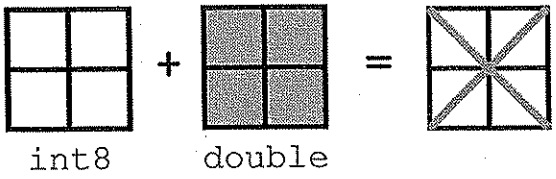


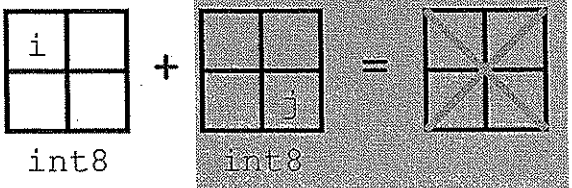




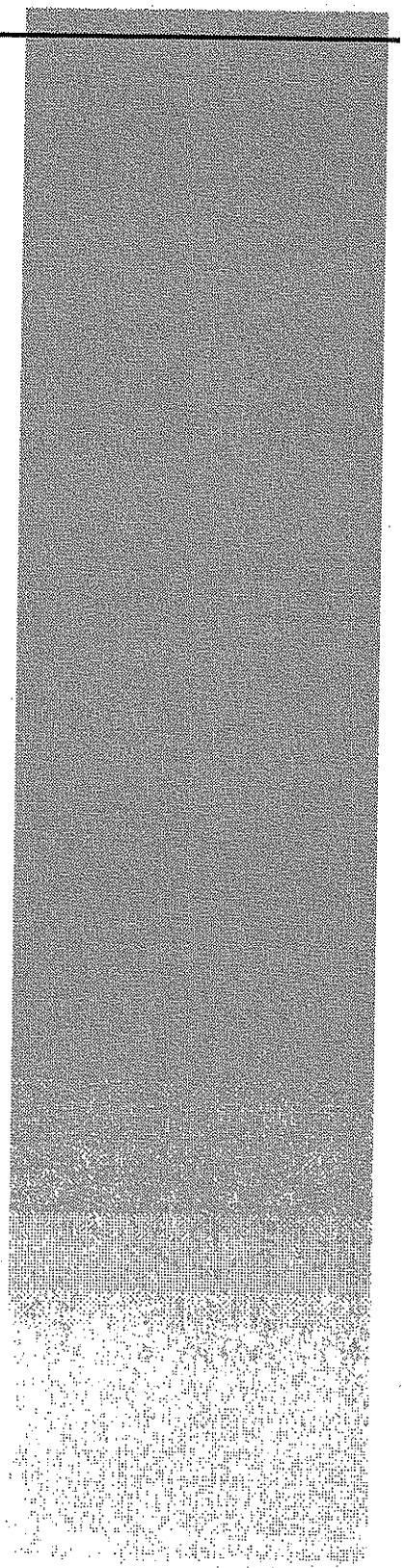








Data Types



Storage formats

`full`	Convert sparse matrix to full
`sparse`	Convert full matrix to sparse

Numeric arrays

`zeros, ones`	Preallocate for any numeric type
`double`	Convert to double precision
`single`	Convert to single precision
`uint8, uint16, uint32, uint64`	Convert to unsigned integer precision
`int8, int16, int32, int64`	Convert to signed integer precision
`isnumeric`	Test for numeric arrays
`isfloat`	Test for double and single arrays
`isinteger`	Test for integer arrays

Logical arrays

`logical`	Convert to logical
`islogical`	Test for logical arrays

Cell arrays

`cell`	Construct cell array
`cellfun`	Apply function to cell array elements
`celldisp`	Display cell array contents
`cellplot`	Graphical display of cell array structure
`cell2mat`	Convert cell array of matrices to matrix
`cell2struct`	Convert cell array to structure array
`num2cell`	Convert matrix to cell array
`mat2cell`	Convert matrix to cell array of submatrices
`deal`	Assign elements of cell array to variables
`iscell`	Test for cell arrays

Structure arrays

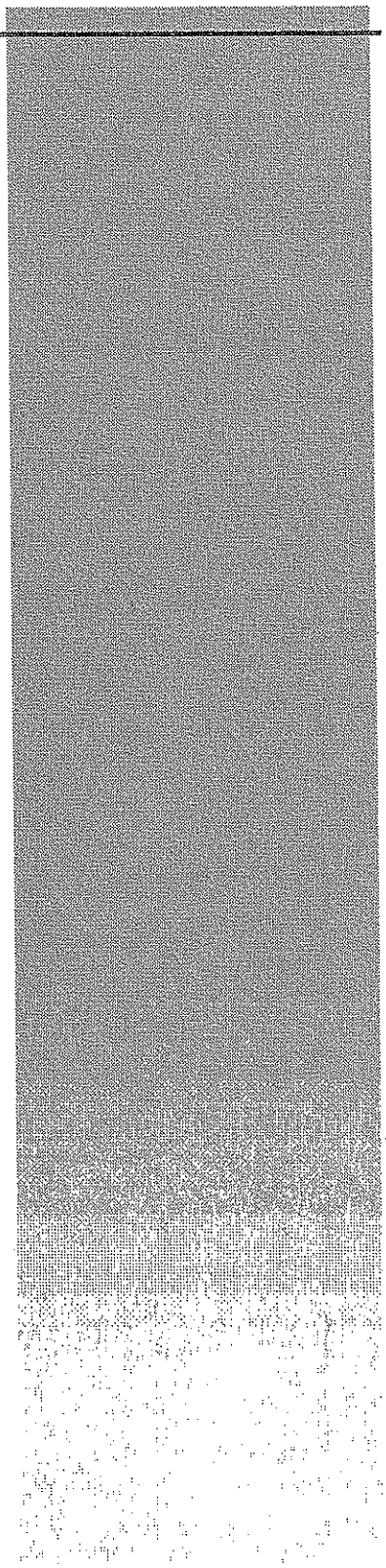
`struct`	Construct structure array
`fieldnames`	Get field names
`rmfield`	Remove field
`orderfields`	Order fields
`deal`	Assign fields to variables
`struct2cell`	Convert structure array to cell array
`isstruct`	Test for structure arrays
`isfield`	Test if field exists

Function handles

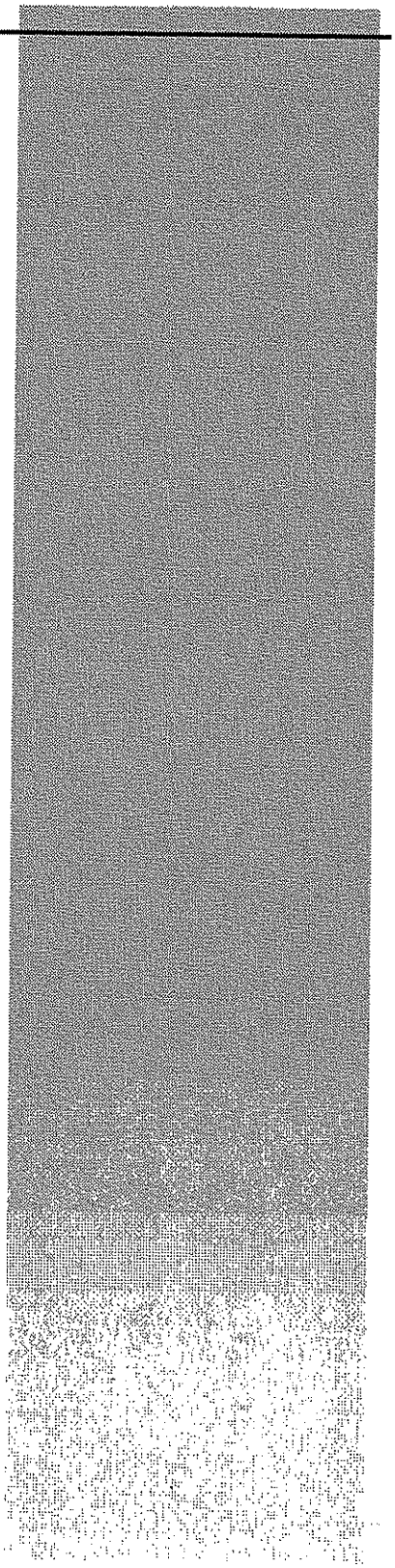
`@`	Construct function handle
`functions`	Information on function handles
`feval`	Evaluate function handle or M-file function
`func2str`	Convert function handle to name string
`str2func`	Convert function name string to handle
`isa`	Test for type (including function handles)
`isequal`	Test for equality (handles and other arrays)

Java™ classes

`javaArray`	Construct a Java™ array
`javaObject`	Construct a Java™ object
`javaMethod`	Invoke a Java™ method
`methods`	Display class methods
`javaaddpath`	Add entries to dynamic class path
`javarmpath`	Remove entries from dynamic class path
`isjava`	Test for Java™ object



File I/O



Workspace variables

`save`	Save workspace to MAT-file
`load`	Load workspace from MAT-file

***read functions**

`auread`	NeXT/SUN® sound file
`aviread`	Audio/Video Interleaved file
`cdfread`	Common Data Format file
`csvread`	Comma-separated value file
`dlmread`	ASCII-delimited file of numeric data
`fitsread`	Flexible Image Transport System file
`hdf5read`	Hierarchical data format file
`imread`	Image or graphics file
`multibandread`	Binary band interleaved data file
`textread`	Text file
`urlread`	Web content
`wavread`	Microsoft® WAVE sound file
`wk1read`	Lotus®1-2-3® spreadsheet file
`xlsread`	Microsoft® Excel® spreadsheet file
`xmlread`	XML document

***write functions**

`auwrite`	NeXT/SUN® sound file
`avifile`	Audio/Video Interleaved file
`cdfwrite`	Common Data Format file
`csvwrite`	Comma-separated value file
`dlmwrite`	ASCII-delimited file of numeric data
`hdf5write`	Hierarchical data format file
`imwrite`	Image or graphics file
`multibandwrite`	Binary band interleaved data file
`urlwrite`	Web content
`wavwrite`	Microsoft® WAVE sound file
`wk1write`	Lotus®1-2-3® spreadsheet file
`xlswrite`	Microsoft® Excel® spreadsheet file
`xmlwrite`	XML document

Opening and closing

`fopen`	Open file
`fclose`	Close file

Binary data

`fread`	Read binary data from file
`fwrite`	Write binary data to file

Formatted text

`fscanf`	Read formatted data from file
`fprintf`	Write formatted data to file
`fgetl`	Read line (discard newline character)
`fgets`	Read line (keep newline character)

Conversion characters

`%d`, `%i`	Integer
`%u`	Unsigned integer
`%o`	Octal integer
`%x`	Hexadecimal integer
`%f`	Floating-point number (6 decimal places)
`%e`	Floating-point number (exponential format)
`%g`	Floating-point number (`%f` or `%e` format)
`%c`	Single character
`%s`	Null-terminated character string

String conversion

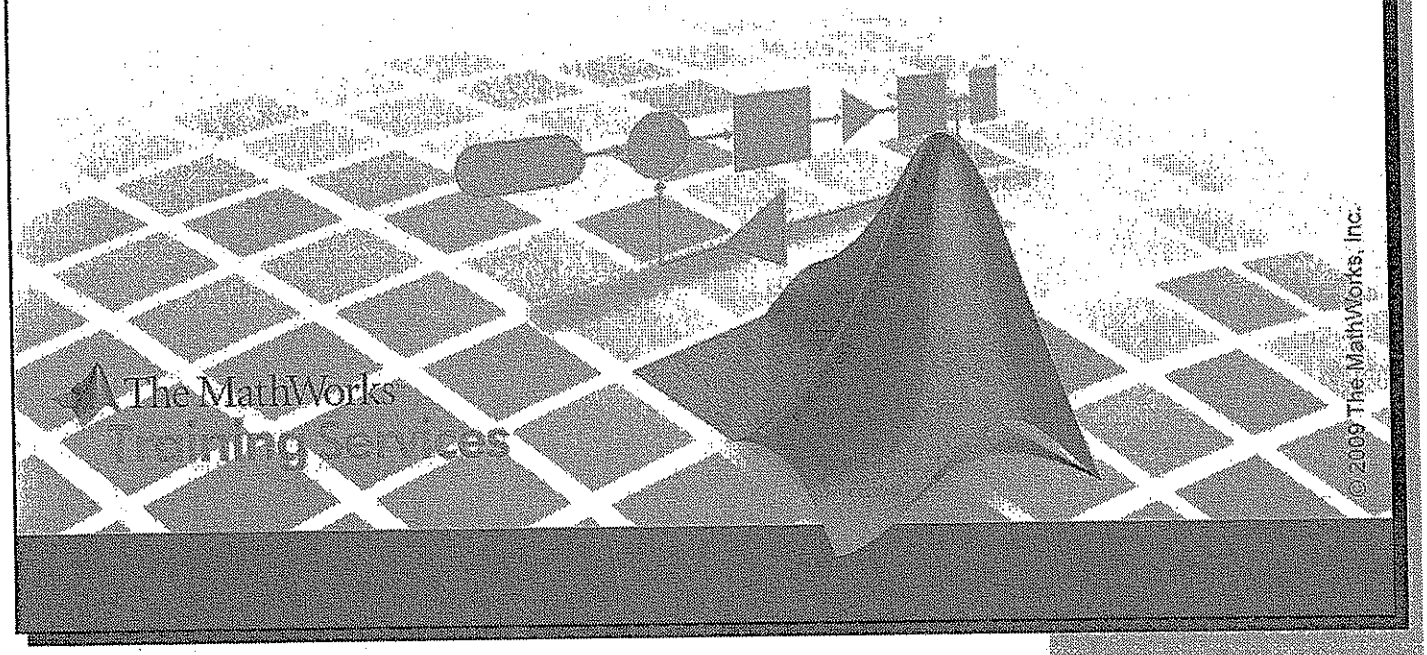
`sprintf`	Write formatted data to string
`sscanf`	Read string under format control

Positioning

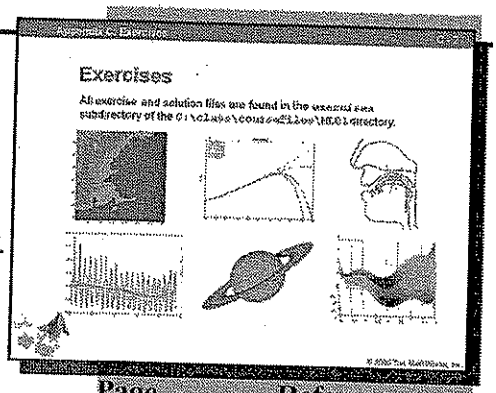
`fseek`	Set file position indicator
`ftell`	Get file position indicator
`frewind`	Rewind file
`feof`	Test for end-of-file
`ferror`	Inquire file error status

MATLAB® Fundamentals

Appendix C: Exercises

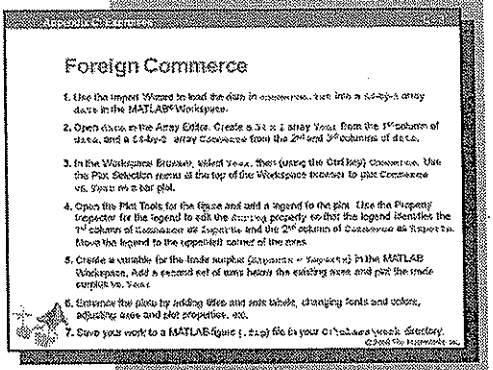


Exercises



All exercise and solution files are found in the `exercises` subdirectory of the `C:\class\coursefiles\ML01` directory.

Foreign Commerce


Foreign Commerce

1. Use the Import Wizard to load the data in commerce.txt into a 54-by-3 array data in the MATLAB® Workspace.
2. Open data in the Array Editor. Create a 54 x 1 array Year from the 1st column of data, and a 54-by-2 array Commerce from the 2nd and 3rd columns of data.
3. In the Workspace Browser, select Year, then (using the Ctrl key) Commerce. Use the Plot Selection menu at the top of the Workspace browser to plot Commerce vs. Year as a bar plot.
4. Open the Plot Tools for the figure and add a legend to the plot. Use the Property Inspector for the legend to edit the String property so that the legend identifies the 1st column of Commerce as Imports and the 2nd column of Commerce as Exports. Move the legend to the opposite corner of the axes.
5. Create a variable for the trade surplus (Exports – Imports) in the MATLAB Workspace. Add a second set of axes below the existing axes and plot the trade surplus vs. Year.
6. Enhance the plots by adding titles and axis labels, changing fonts and colors, adjusting axes and plot properties, etc.
7. Save your work to a MATLAB figure (.fig) file in your C:\class\work directory.

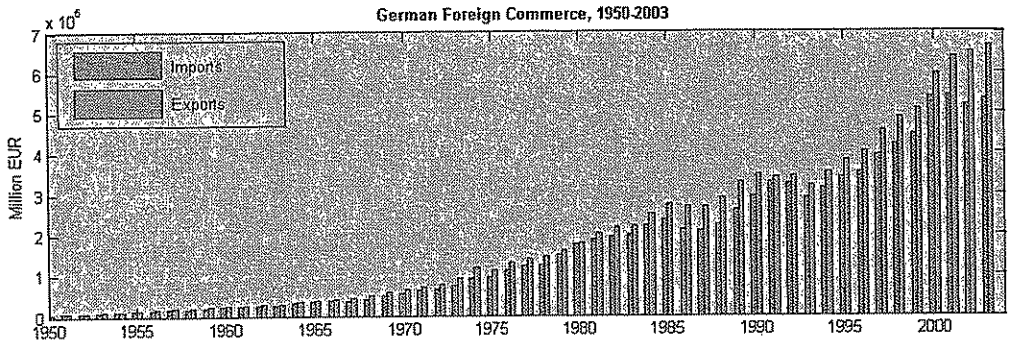
Reference: Section 2
Topics: MATLAB® user interface, file I/O, plotting

Exercise

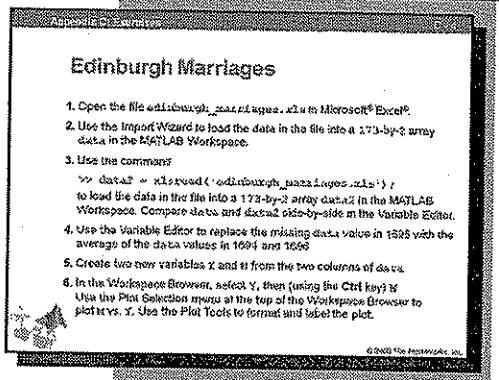
1. Use the Import Wizard to load the data in `commerce.txt` into a `54`-by-`3` array `data` in the MATLAB workspace.
2. Open `data` in the Variable Editor. Create a `54`-by-`1` array `Year` from the 1st column of `data`, and a `54`-by-`2` array `Commerce` from the 2nd and 3rd columns of `data`.
3. In the Workspace browser, select `Year`, then (using the **Ctrl** key) `Commerce`. Use the Plot Selection menu at the top of the Workspace browser to plot `Commerce` vs. `Year` as a bar plot.
4. Open the Plot Tools for the figure and add a legend to the plot. Use the Property Inspector for the legend to edit the `String` property so that the legend identifies the 1st column of `Commerce` as `Imports` and the 2nd column of `Commerce` as `Exports`. Move the legend to the upper-left corner of the axes.
5. Create a variable for the trade surplus (`Exports – Imports`) in the MATLAB workspace. Add a second set of axes below the existing axes and plot the trade surplus vs. `Year`.
6. Enhance the plots by adding titles and axis labels, changing fonts and colors, adjusting axes and plot properties, etc.
7. Save your work to a MATLAB figure (`.fig`) file in your `C:\class\work` directory.

SOLUTION

```
>> open(...
'commerce.fig')
```



Edinburgh Marriages



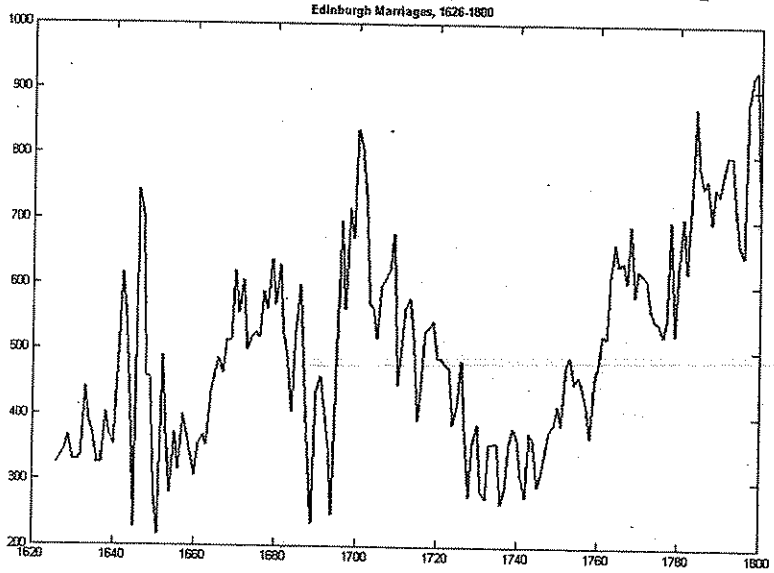
Reference: Section 2
Topics: MATLAB user interface, file I/O, plotting

Exercise

1. Open the file `edinburgh_marriages.xls` in Microsoft® Excel®.
2. Use the Import Wizard to load the data in the file into a `173`-by-`2` array `data` in the MATLAB workspace.
3. Use the command

   ```
   >> data2 = xlsread('edinburgh_marriages.xls');
   ```

   to load the data in the file into a `173`-by-`2` array `data2` in the MATLAB workspace. Compare `data` and `data2` side by side in the Variable Editor.
4. Use the Variable Editor to replace the missing `data` value in 1695 with the average of the `data` values in 1694 and 1696.
5. Create two new variables `Y` and `M` from the two columns of `data`.
6. In the Workspace browser, select `Y`, then (using the **Ctrl** key) `M`. Use the Plot Selection menu at the top of the Workspace browser to plot `M` vs. `Y`. Use the Plot Tools to format and label the plot.

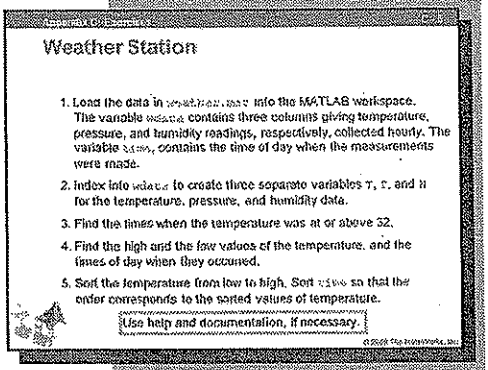


SOLUTION

```
>> edit emsoln
>> emsoln
```

Weather Station

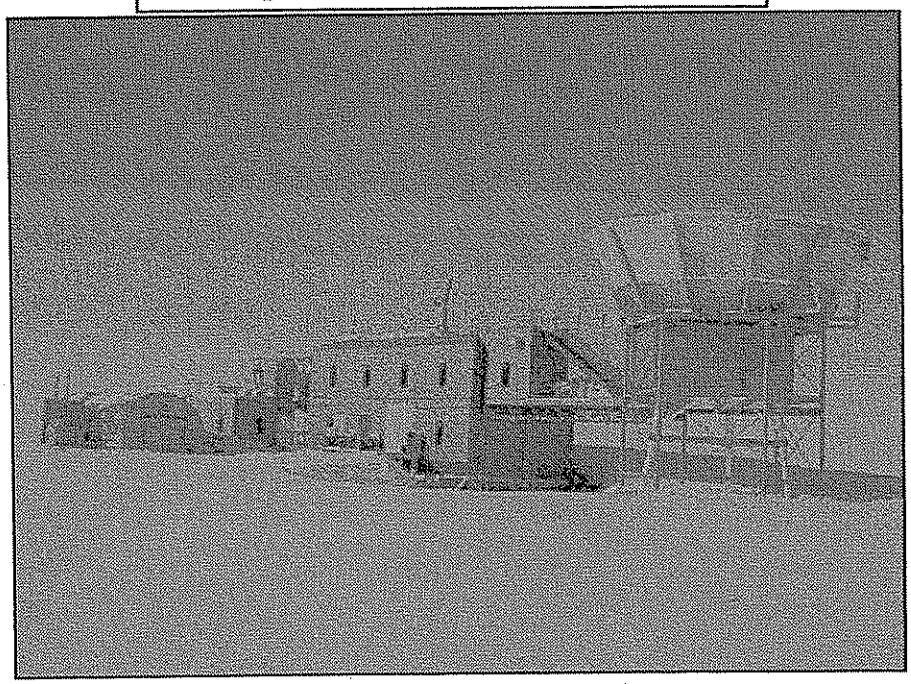
Reference: Section 3
Topics: MATLAB variables, indexing, help and documentation



Exercise

1. Load the data in `weather.mat` into the MATLAB workspace. The variable `wdata` contains three columns giving temperature, pressure, and humidity readings, respectively, collected hourly. The variable `time`, contains the time of day when the measurements were made.
2. Index into `wdata` to create three separate variables `T`, `P`, and `H` for the temperature, pressure, and humidity data.
3. Find the times when the temperature was at or above 32.
4. Find the high and the low values of the temperature, and the times of day when they occurred.
5. Sort the temperature from low to high. Sort `time` so that the order corresponds to the sorted values of temperature.

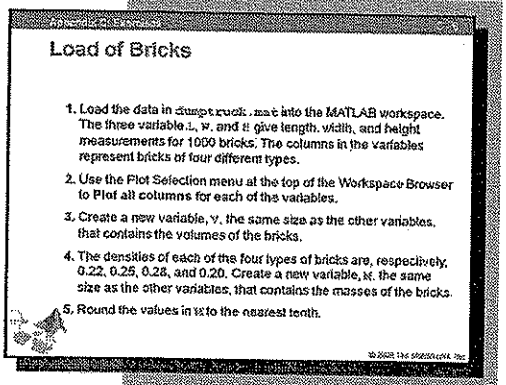
Use help and documentation, if necessary.



SOLUTION

```
>> edit weather
>> weather
```

Load of Bricks



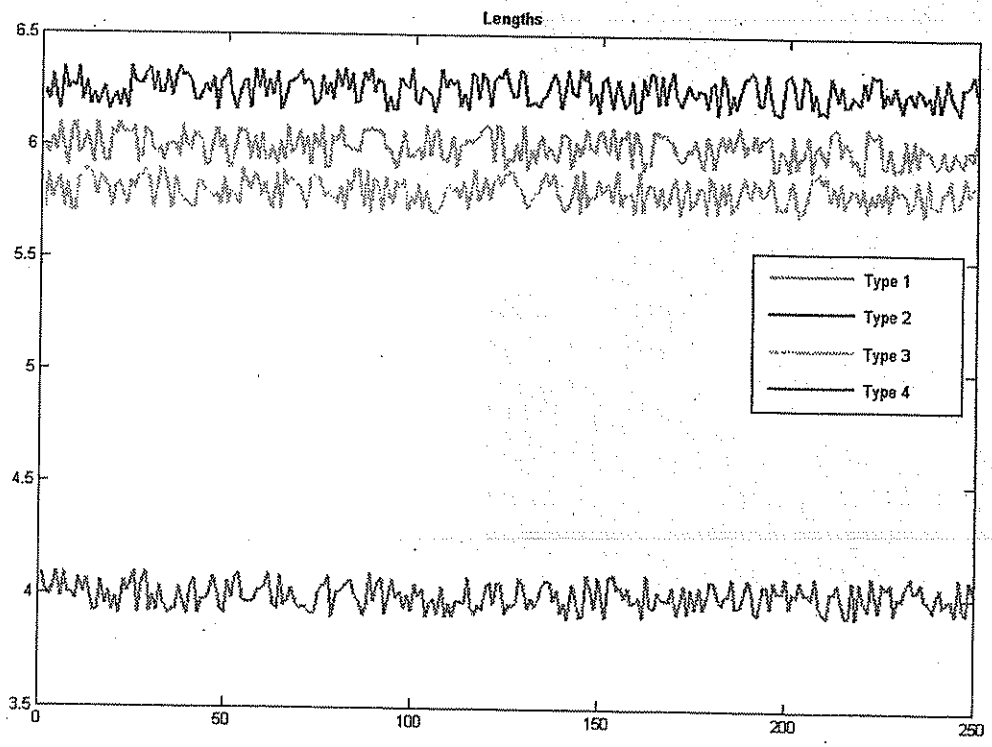
Reference: Section 3
Topics: MATLAB variables, matrix and array operations

Exercise

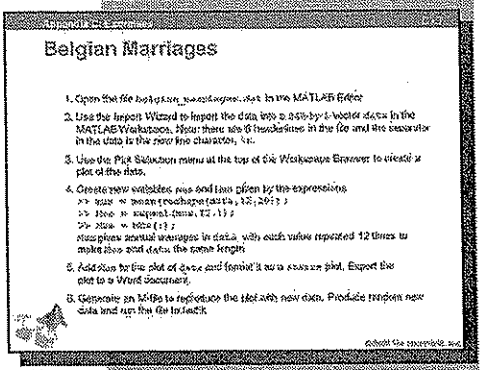
1. Load the data in `dumptruck.mat` into the MATLAB workspace. The three variables `L`, `W`, and `H` give length, width, and height measurements for 1000 bricks. The columns in the variables represent bricks of four different types.
2. Use the Plot Selection menu at the top of the Workspace browser to **Plot all columns** for each of the variables.
3. Create a new variable, `V`, the same size as the other variables, that contains the volumes of the bricks.
4. The densities of each of the four types of bricks are, respectively, 0.22, 0.25, 0.28, and 0.20. Create a new variable, `M`, that is the same size as the other variables, and contains the masses of the bricks.
5. Round the values in `M` to the nearest tenth.

SOLUTION

```
>> edit bricks
>> bricks
```



Belgian Marriages



Reference: Section 3
Topics: MATLAB user interface, file I/O, plotting, working with variables, exporting figures, generating M-files, random numbers.

Exercise

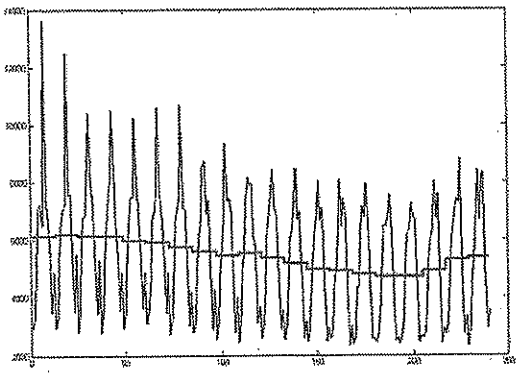
1. Open the file `belgian_marriages.dat` in the MATLAB Editor.
2. Use the Import Wizard to import the data into a `240`-by-`1` vector `data` in the MATLAB workspace. There are six header lines in the file and the separator in the data is the new line character, `\n`.
3. Use the Plot Selection menu at the top of the Workspace browser to create a plot of the data.
4. Create new variables `mus` and `Mus` given by the expressions

```
>> mus =mean(reshape(data,12,20));
>> Mus =repmat(mus,12,1);
>> Mus = Mus(:);
```

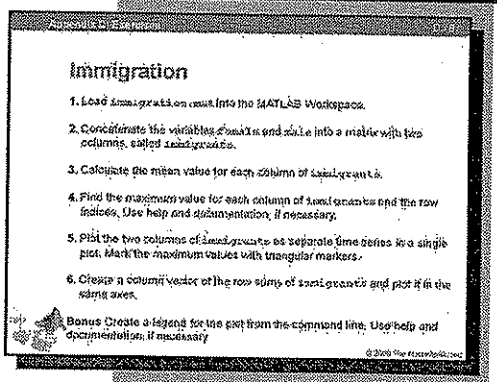
   `Mus` gives annual averages in `data`, with each value repeated 12 times to make `Mus` and `data` the same length.
5. Add `Mus` to the plot of `data` and format it as a `stairs` plot. Export the plot to a Word document.
6. Generate an M-file to reproduce the plot with new data. Produce random new data and run the file to test it.

SOLUTION

```
>> edit belgiansoln
>> belgiansoln
```



Immigration



Reference: Section 3
Topics: MATLAB variables, plotting, help and documentation

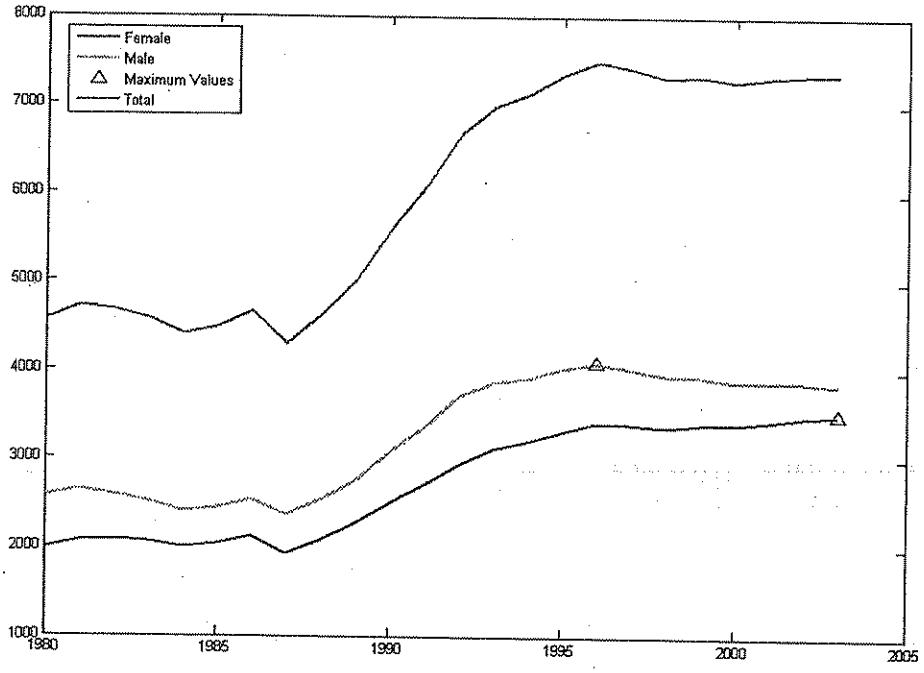
Exercise

1. Load `immigration.mat` into the MATLAB workspace.
2. Concatenate the variables `female` and `male` into a matrix with two columns, called `immigrants`.
3. Calculate the mean value for each column of `immigrants`.
4. Find the maximum value for each column of `immigrants` and the row indices. Use help and documentation, if necessary.
5. Plot the two columns of `immigrants` as separate time series in a single plot. Mark the maximum values with triangular markers.
6. Create a column vector of the row sums of `immigrants` and plot it in the same axes.

Bonus Create a legend for the plot from the command line. Use help and documentation, if necessary.

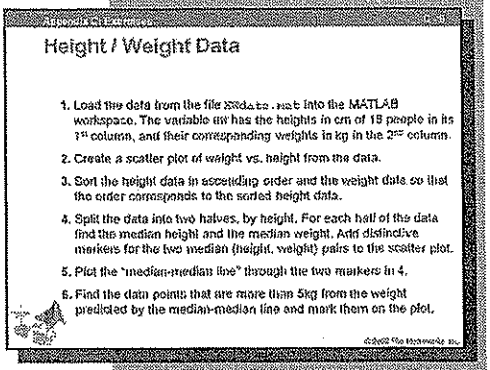
SOLUTION

```
>> edit immigration
>> immigration
```



Height / Weight Data

Reference: Section 4
Topics: MATLAB variables, plotting , help and documentation

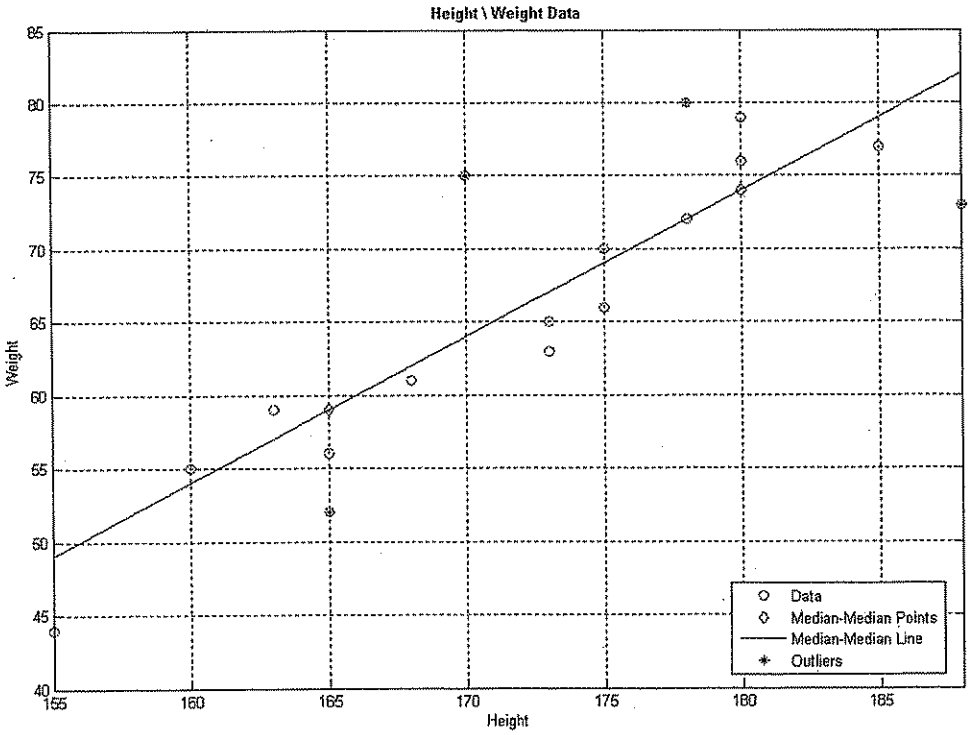


Exercise

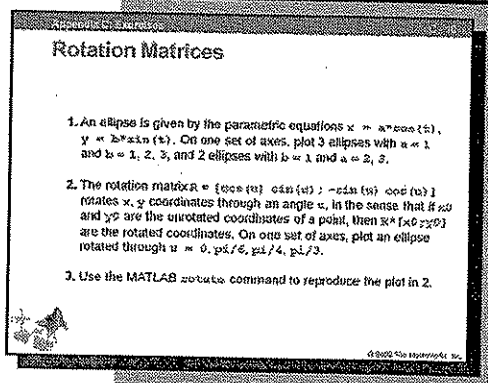
1. Load the data from the file `HWdata.mat` into the MATLAB workspace. The variable `HW` has the heights in cm of 18 people in its 1st column, and their corresponding weights in kg in the 2nd column.
2. Create a scatter plot of weight vs. height from the data.
3. Sort the height data in ascending order and the weight data so that the order corresponds to the sorted height data.
4. Split the data into two halves, by height. For each half of the data find the median height and the median weight. Add distinctive markers for the two median (height, weight) pairs to the scatter plot.
5. Plot the median-median line through the two markers in 4.
6. Find the data points that are more than 5 kg from the weight predicted by the median-median line and mark them on the plot.

SOLUTION

```
>> edit HWsoln
>> HWsoln
```



Rotation Matrices


Rotation Matrices

1. An ellipse is given by the parametric equations x = a*cos(t), y = b*sin(t). On one set of axes, plot 3 ellipses with a = 1 and b = 1, 2, 3, and 2 ellipses with b = 1 and a = 2, 3.

2. The rotation matrix R = [cos(u) sin(u); -sin(u) cos(u)] rotates x, y coordinates through an angle u, in the sense that if x0 and y0 are the unrotated coordinates of a point, then R*[x0;y0] are the rotated coordinates. On one set of axes, plot an ellipse rotated through u = 0, pi/6, pi/4, pi/3.

3. Use the MATLAB rotate command to reproduce the plot in 2.

Reference: Section 4
Topics: Matrix arithmetic, parametric equations, plotting, help and documentation

Exercise

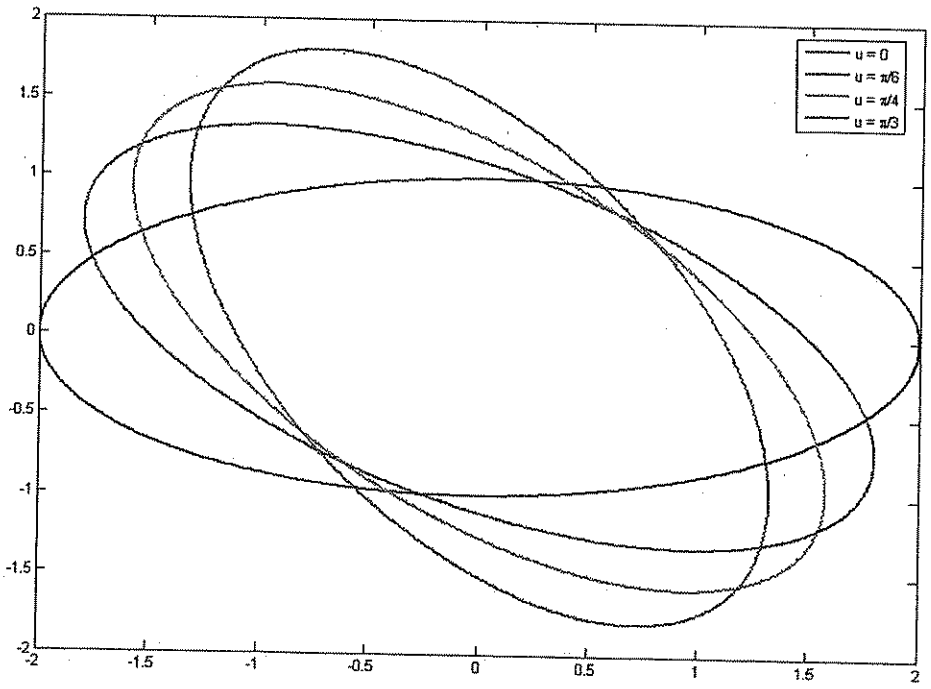
1. An ellipse is given by the parametric equations `x = a*cos(t)`, `y = b*sin(t)`. On one set of axes, plot 3 ellipses with `a = 1` and `b = 1, 2, 3`, and 2 ellipses with `b = 1` and `a = 2, 3`.

2. The rotation matrix `R = [cos(u) sin(u); -sin(u) cos(u)]` rotates `x`, `y` coordinates through an angle `u`, in the sense that if `x0` and `y0` are the unrotated coordinates of a point, then `R*[x0;y0]` are the rotated coordinates. On one set of axes, plot an ellipse rotated through `u = 0, pi/6, pi/4, pi/3`.

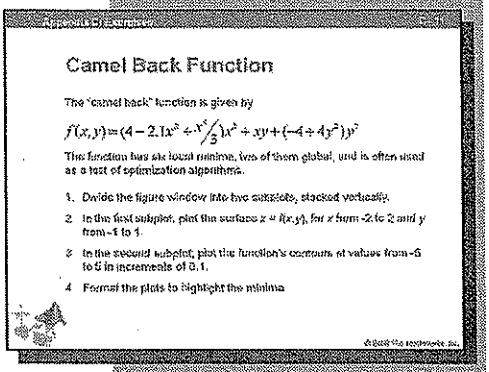
3. Use the MATLAB `rotate` command to reproduce the plot in 2.

SOLUTION

```
>> edit rots
>> rots
```



Camel Back Function



Reference: Section 4
Topics: Matrix data, surface and contour plotting, subplotting

Exercise

The "camel back" function is given by

$$f(x,y)=(4-2.1x^2+\frac{x^4}{3})x^2+xy+(-4+4y^2)y^2$$

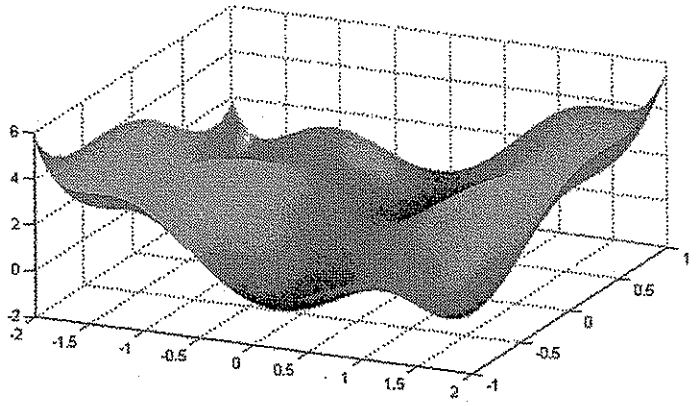
The function has six local minima, two of them global, and is often used as a test of optimization algorithms.

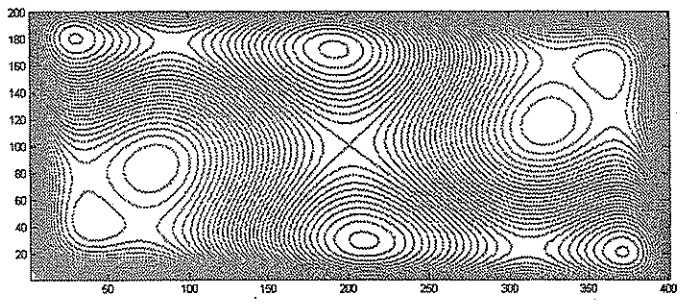
1. Divide the figure window into two subplots, stacked vertically.
2. In the first subplot, plot the surface $z = f(x,y)$, for x from -2 to 2 and y from -1 to 1.
3. In the second subplot, plot the function's contours at values from -5 to 5 in increments of 0.1.
4. Format the plots to highlight the minima.

SOLUTION

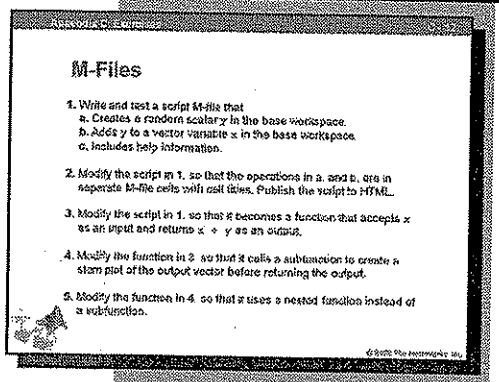
```
>> edit cbplot
>> xrange = [-2 2];
>> yrange = [-1 1];
>> levels = -5:0.1:5;
>> cbplot(...
xrange,yrange,levels)
```

Use the Camera Toolbar to rotate the plot and add lighting.





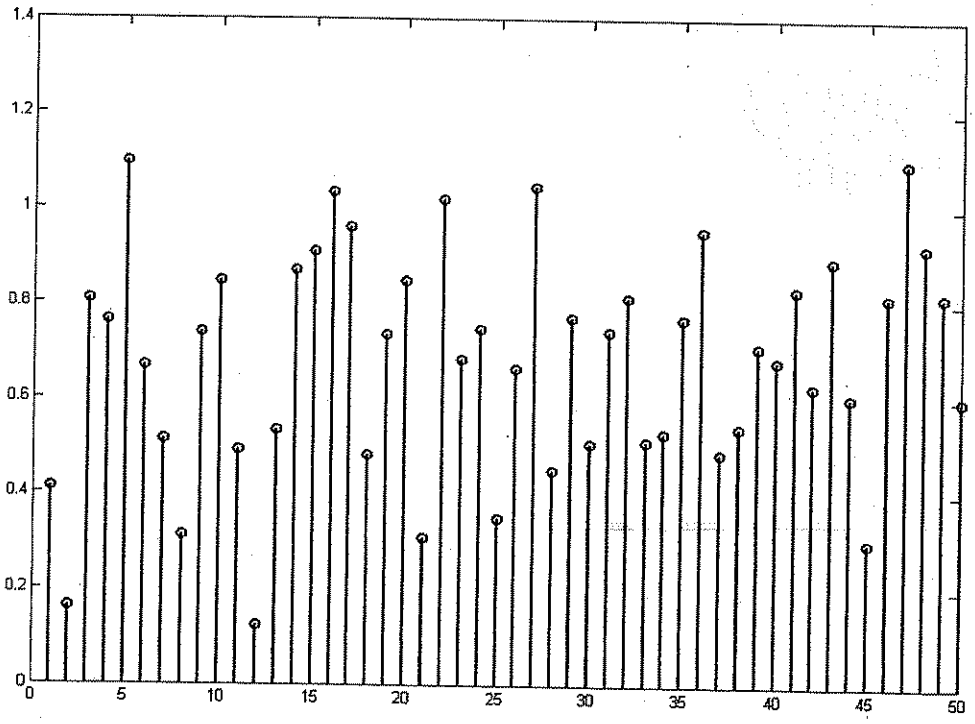
M-Files



Reference: Section 5
Topics: M-file scripts and functions

Exercise

1. Write and test a script M-file that
   a. Creates a random scalar `y` in the base workspace.
   b. Adds `y` to a vector variable `x` in the base workspace.
   c. Includes help information.
2. Modify the script in 1 so that the operations in a and b are in separate M-file cells with cell titles. Publish the script to HTML.
3. Modify the script in 1 so that it becomes a function that accepts `x` as an input and returns `x + y` as an output.
4. Modify the function in 3 so that it calls a subfunction to create a stem plot of the output vector before returning the output.
5. Modify the function in 4 so that it uses a nested function instead of a subfunction.



SOLUTION

1.
```
>> edit M1
>> x = rand(1,50);
>> M1
```

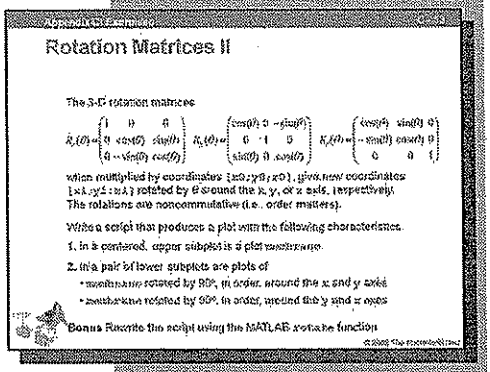
2.
```
>> edit M2
>> x = rand(1,50);
>> M2
```

3.
```
>> edit M3
>> x = rand(1,50);
>> z = M3(x)
```

4.
```
>> edit M4
>> x = rand(1,50);
>> z = M4(x)
```

5.
```
>> edit M5
>> x = rand(1,50);
>> z = M5(x)
```

Rotation Matrices II



Reference: Section 5
Topics: M-file scripts, matrix arithmetic, plotting

Exercise

The 3-D rotation matrices

$$R_x(\theta)=\begin{pmatrix}1 & 0 & 0\\ 0 & \cos(\theta) & \sin(\theta)\\ 0 & -\sin(\theta) & \cos(\theta)\end{pmatrix} \quad R_y(\theta)=\begin{pmatrix}\cos(\theta) & 0 & -\sin(\theta)\\ 0 & 1 & 0\\ \sin(\theta) & 0 & \cos(\theta)\end{pmatrix} \quad R_z(\theta)=\begin{pmatrix}\cos(\theta) & \sin(\theta) & 0\\ -\sin(\theta) & \cos(\theta) & 0\\ 0 & 0 & 1\end{pmatrix}$$

when multiplied by coordinates `[x0;y0;z0]`, give new coordinates `[x1;y1;z1]` rotated by θ around the `x`, `y`, or `z` axis, respectively. The rotations are noncommutative (i.e., order matters).

Write a script that produces a plot with the following characteristics:

1. In a centered, upper subplot is a plot of `membrane`.
2. In a pair of lower subplots are plots of
   - `membrane` rotated by 90°, in order, around the `x` and `y` axes
   - `membrane` rotated by 90°, in order, around the `y` and `x` axes

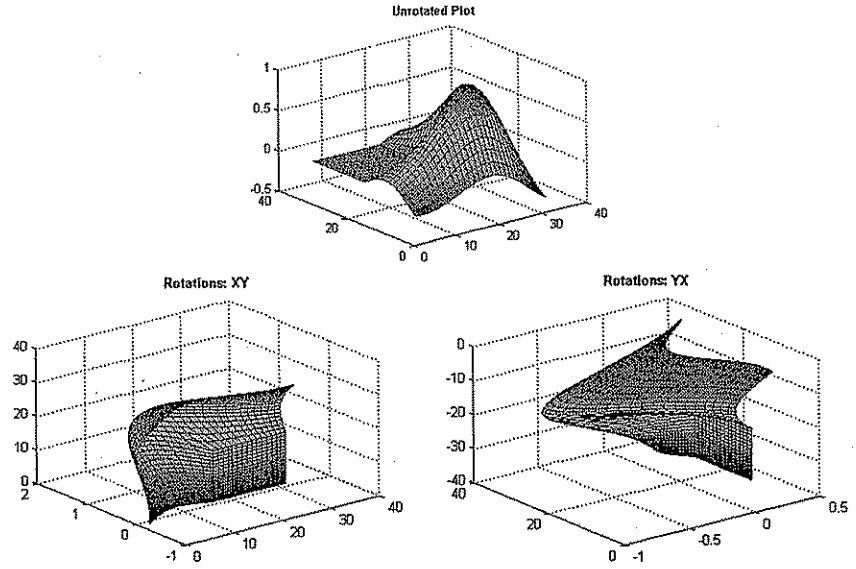
Bonus Rewrite the script using the MATLAB `rotate` function.

SOLUTION

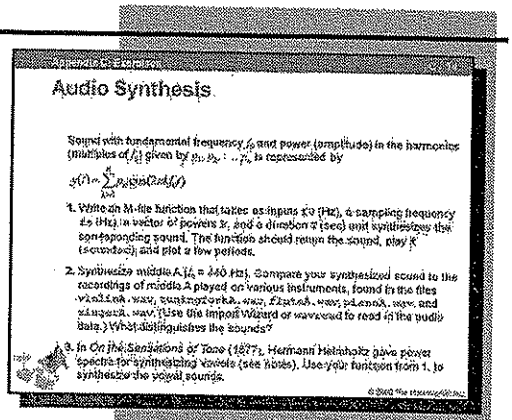
```
>> edit rot3d
>> rot3d
```

Bonus

```
>> edit rot3db
>> rot3db
```



Audio Synthesis


Audio Synthesis

Sound with fundamental frequency f_0 and power (amplitude) in the harmonics (multiples of f_0) given by $p_1, p_2, \ldots, p_n$ is represented by

$$y(t) = \sum_{k=1}^{n} p_k \cos(2\pi k f_0 t)$$

1. Write an M-file function that takes as inputs f0 (Hz), a sampling frequency fs (Hz), a vector of powers P, and a duration T (sec) and synthesizes the corresponding sound. The function should return the sound, play it (soundsc), and plot a few periods.
2. Synthesize middle A (f_0 = 440 Hz). Compare your synthesized sound to the recordings of middle A played on various instruments, found in the files violinA.wav, tuningforkA.wav, fluteA.wav, pianoA.wav, and singerA.wav. (Use the Import Wizard or wavread to read in the audio data.) What distinguishes the sounds?
3. In *On the Sensations of Tone* (1877), Hermann Helmholtz gave power spectra for synthesizing vowels (see table). Use your function from 1. to synthesize the vowel sounds.

Reference: Section 5
Topics: M-file functions, matrix arithmetic, plotting

Exercise

Sound with fundamental frequency f_0 and power (amplitude) in the harmonics (multiples of f_0) given by $p_1, p_2, \ldots, p_n$ is modeled by

$$y(t) = \sum_{k=1}^{n} p_n \cos(2\pi k f_0 t)$$

1. Write an M-file function that takes as inputs `f0` (Hz), a sampling frequency `fs` (Hz), a vector of powers `P`, and a duration `T` (sec) and synthesizes the corresponding sound. The function should return the sound, play it (`soundsc`), and plot a few periods.
2. Synthesize middle A (f_0=440 Hz). Compare your synthesized sound to the recordings of middle A played on various instruments, found in the files `violinA.wav`, `tuningforkA.wav`, `fluteA.wav`, `pianoA.wav`, and `singerA.wav`. (Use the Import Wizard or `wavread` to read in the audio data.) What distinguishes the sounds?
3. In *On the Sensations of Tone* (1877), Hermann Helmholtz gave the following power spectra for synthesizing vowels.

	p1	p2	p3	p4	p5	p6	p8	p16
U oo as in boot	*ff*	*mf*	*pp*					
O oh as in no	*mf*	*f*	*mf*	*p*				
A ah as in caught	*p*	*p*	*p*	*mf*	*mf*	*p*	*p*	
E eh as in bed	*mf*		*mf*			*ff*		
I ee as in see	*mf*	*p*				*p*		*mf*

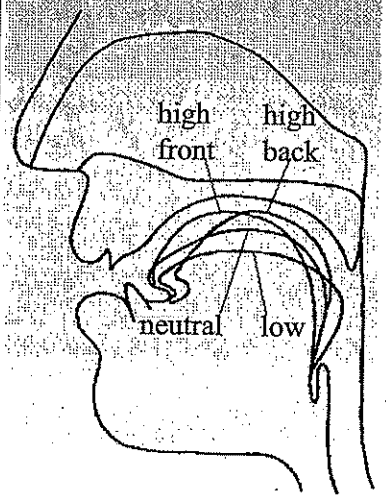
where the abbreviations are *ff* for *fortissimo* (very loud), *f* for *forte* (loud), *mf* for *mezzo forte* (moderately loud), *p* for *piano* (soft) and *pp* for *pianissimo* (very soft). Use your function from 1 to synthesize the vowel sounds.

SOLUTION

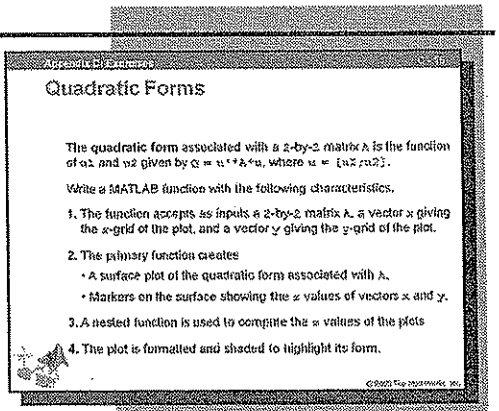
1.
```
>> edit synthesize
```

2.
```
>> edit middleA
>> middleA
```

3.
```
>> edit vowels
>> vowels
```



Quadratic Forms


Quadratic Forms

The quadratic form associated with a 2-by-2 matrix A is the function of u1 and u2 given by Q = u'*A*u, where u = [u1;u2].

Write a MATLAB function with the following characteristics.

1. The function accepts as inputs a 2-by-2 matrix A, a vector x giving the x-grid of the plot, and a vector y giving the y-grid of the plot.
2. The primary function creates
   - A surface plot of the quadratic form associated with A.
   - Markers on the surface showing the z values of vectors x and y.
3. A nested function is used to compute the z values of the plots
4. The plot is formatted and shaded to highlight its form.

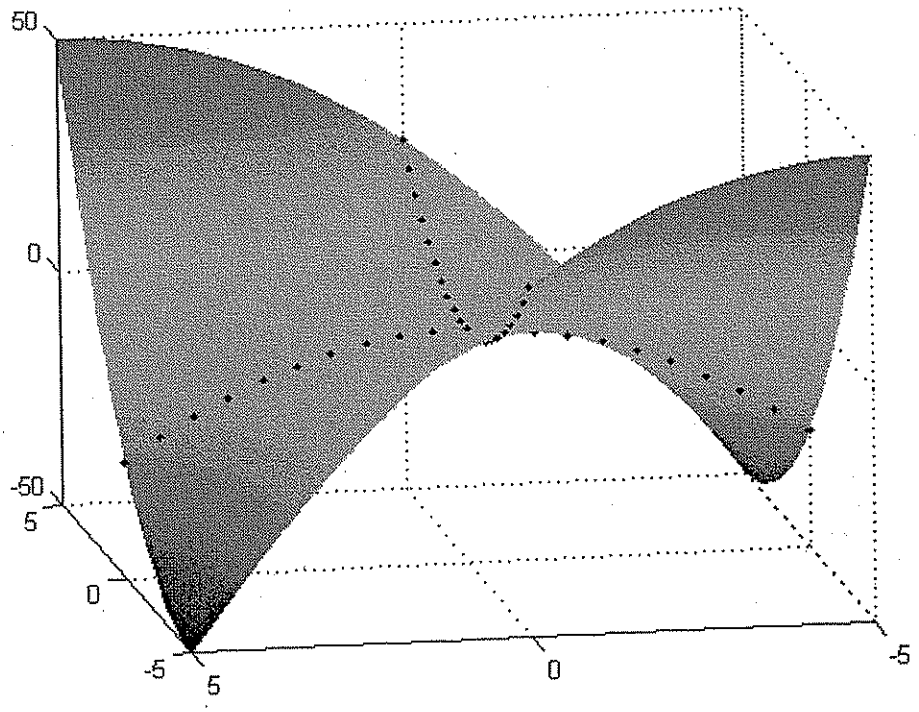
Reference: Section 5
Topics: Matrix arithmetic, plotting matrix data, nested functions

Exercise

The *quadratic form* associated with a 2-by-2 matrix `A` is the function of `u1` and `u2` given by `Q = u'*A*u`, where `u = [u1;u2]`.

Write a MATLAB function with the following characteristics.

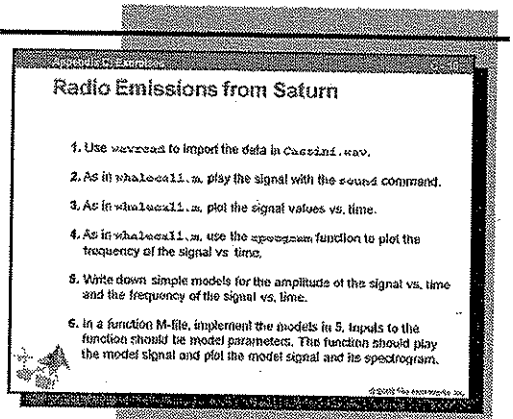
1. The function accepts as inputs a 2-by-2 matrix `A`, a vector `x` giving the `x`-grid of the plot, and a vector `y` giving the `y`-grid of the plot.
2. The primary function creates
   - A surface plot of the quadratic form associated with `A`.
   - Markers on the surface showing the `z` values of vectors `x` and `y`.
3. A nested function is used to compute the `z` values of the plots.
4. The plot is formatted and shaded to highlight its form.



SOLUTION

```
>> edit Qform
>> x = -5:0.5:5;
>> y = x;
>> A = eye(2)
>> Qform(A,x,y)
>> A(4) = -1
>> Qform(A,x,y)
>> A([2 3]) = 1
>> Qform(A,x,y)
```

Radio Emissions from Saturn

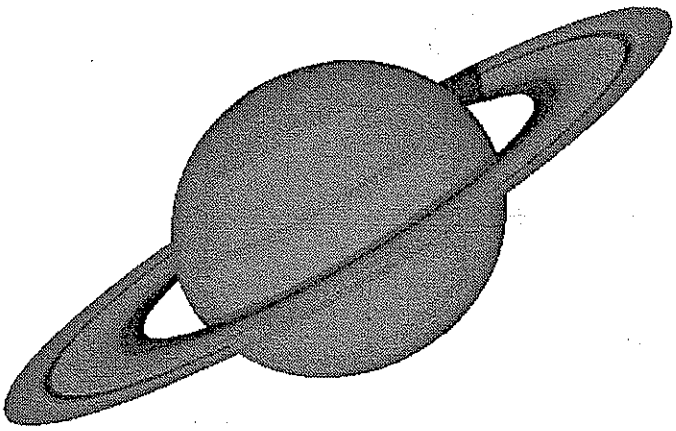


Reference: Section 5
Topics: M-file functions, modeling

Exercise

Saturn is a source of intense radio emissions, which have been monitored by the Cassini spacecraft. The emissions show an amazing array of variations in frequency and time. `Cassini.wav` is a recording of a 27-second emission, compressed to 13 seconds and shifted downward by a factor of 260 to make it audible. Three tones seem to rise from a slowly varying, lower frequency narrowband emission. If so, the signal represents a very complicated interaction between waves in Saturn's radio source region.

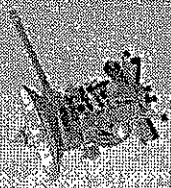
1. Use `wavread` to import the data in `Cassini.wav`.
2. As in `whalecall.m`, play the signal with the `sound` command.
3. As in `whalecall.m`, plot the signal values vs. time.
4. As in `whalecall.m`, use the `specgram` function to plot the frequency of the signal vs. time.
5. Write down simple models for the amplitude of the signal vs. time and the frequency of the signal vs. time.
6. In a function M-file, implement the models in 5. Inputs to the function should be model parameters. The function should play the model signal and plot the model signal and its spectrogram.



SOLUTION

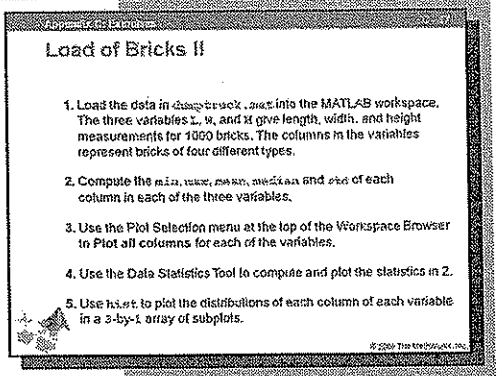
Reference

```
>> edit whalecall

>> edit radio
>> radio
>> edit radiomodel
>> Amp = ...
[0.15,0.10,0.15];
>> BaseF = ...
[200,0.13];
>> RisingF = [65,3];
>> sigma = 0.05;
>> radiomodel(...
Amp,BaseF,...
RisingF,sigma);
```



The Cassini-Huygens mission is a cooperative project of NASA, the European Space Agency, and the Italian Space Agency. The Jet Propulsion Laboratory (JPL), a division of the California Institute of Technology in Pasadena, manages the mission for NASA's Science Mission Directorate, Washington, D.C. The Cassini orbiter was designed, developed, and assembled at JPL. The radio and plasma wave science team is based at the University of Iowa, Iowa City.

Load of Bricks II



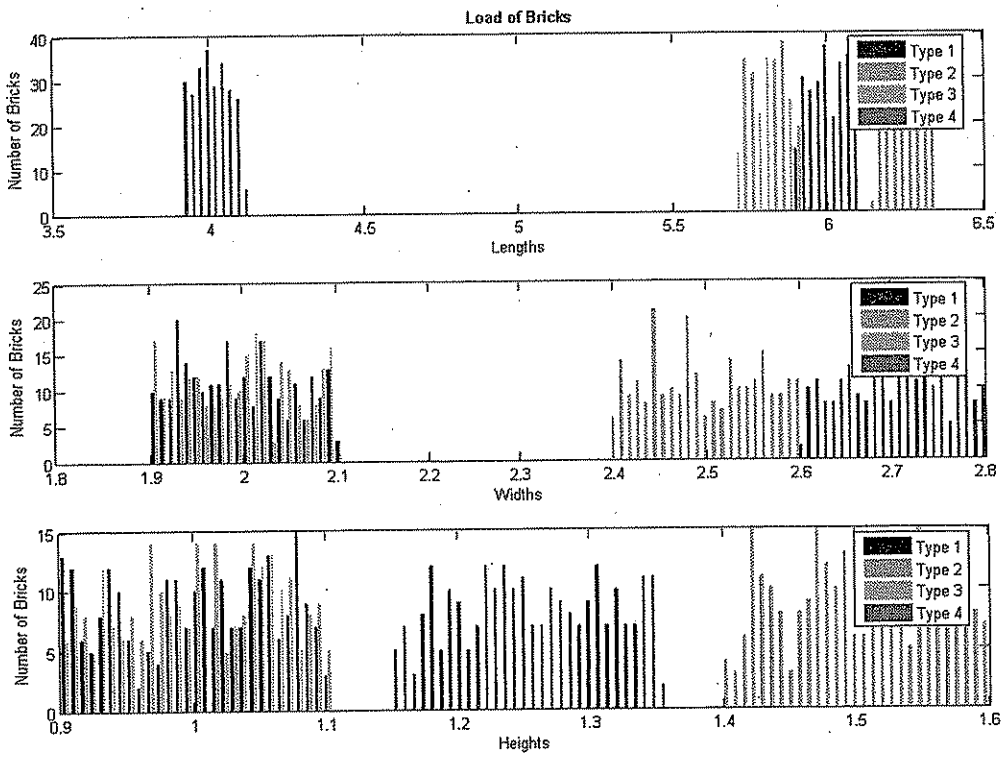
Reference: Section 6
Topics: Matrix data, basic statistics, Data Statistics Tool

Exercise

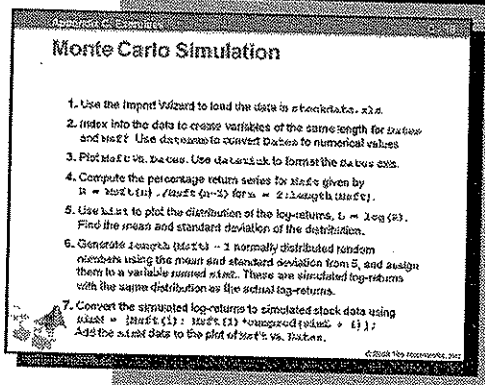
1. Load the data in `dumptruck.mat` into the MATLAB workspace. The three variables `L`, `W`, and `H` give length, width, and height measurements for 1000 bricks. The columns in the variables represent bricks of four different types.
2. Compute the `min`, `max`, `mean`, `median`, and `std` of each column in each of the three variables.
3. Use the Plot Selection menu at the top of the Workspace Browser to **Plot all columns** for each of the variables.
4. Use the Data Statistics Tool to compute and plot the statistics in 2.
5. Use `hist` to plot the distributions of each column of each variable in a `3-by-1` array of subplots.

SOLUTION

```
>> edit bricks2
>> bricks2
```



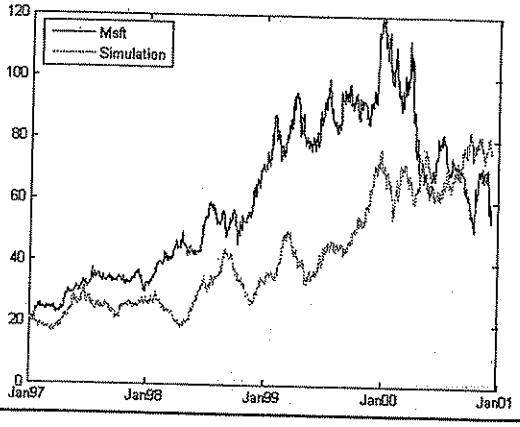
Monte Carlo Simulation



Reference: Section 6
Topics: Distributions and random numbers, plotting

Exercise

1. Use the Import Wizard to load the data in `stockdata.xls`.
2. Index into the data to create variables of the same length for `Dates` and `Msft`. Use `datenum` to convert `Dates` to numerical values.
3. Plot `Msft` vs. `Dates`. Use `datetick` to format the `Dates` axis.
4. Compute the *percentage return series* for `Msft` given by `R = Msft(n)./Msft(n-1)` for `n = 2:length(Msft)`.
5. Use `hist` to plot the distribution of the *log-returns*, `L = log(R)`. Find the mean and standard deviation of the distribution.
6. Generate `length(Msft) - 1` normally distributed random numbers using the mean and standard deviation from 5, and assign them to a variable named `simL`. These are *simulated* log-returns with the same distribution as the actual log returns.
7. Convert the simulated log returns to simulated stock data using[1] `simM = [Msft(1); Msft(1)*cumprod(simL + 1)];` Add the `simM` data to the plot of `Msft` vs. `Dates`.



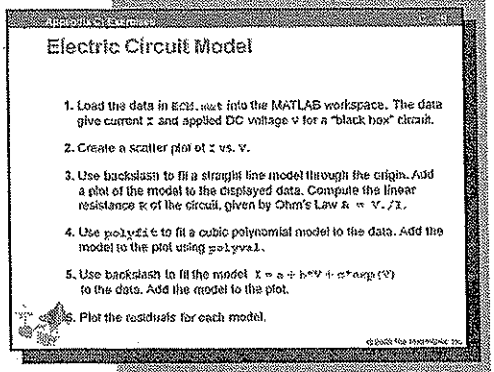
SOLUTION

```
>> edit simMsft
>> simMsft
```

1.
$L = \log(R) = \log(\text{Msft}(n) / \text{Msft}(n-1))$
$= \log([\text{Msft}(n) / \text{Msft}(n-1)] - 1 + 1)$
$\approx \text{Msft}(n) / \text{Msft}(n-1) - 1$ (Taylor approximation: $\log(x+1) \approx x$)

Solving: $\text{Msft}(n) = \text{Msft}(n-1) * (L + 1)$

Electric Circuit Model



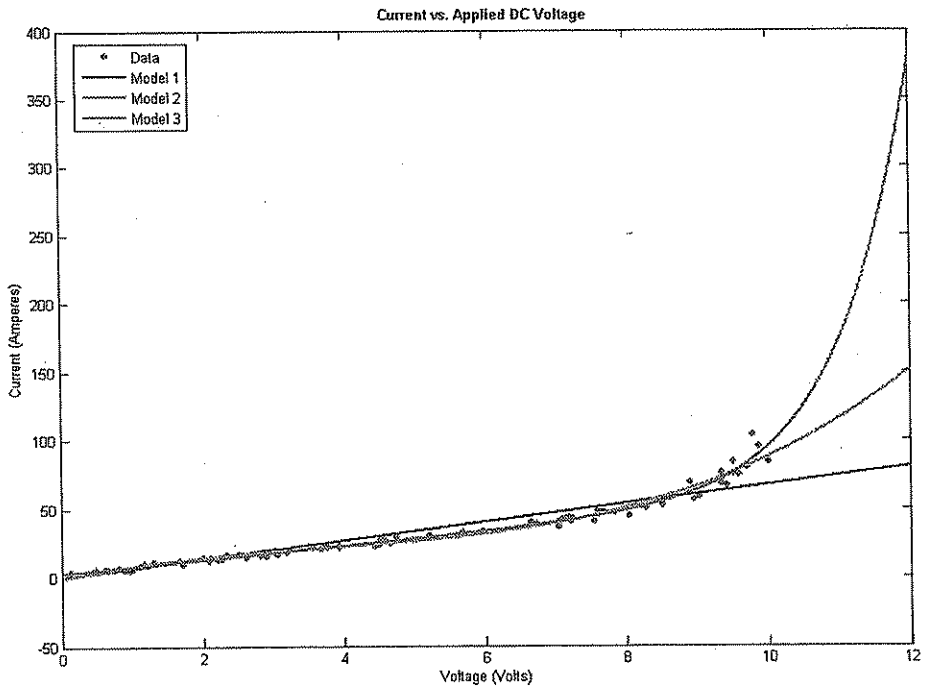
Reference: Section 6
Topics: Regression and curve fitting

Exercise

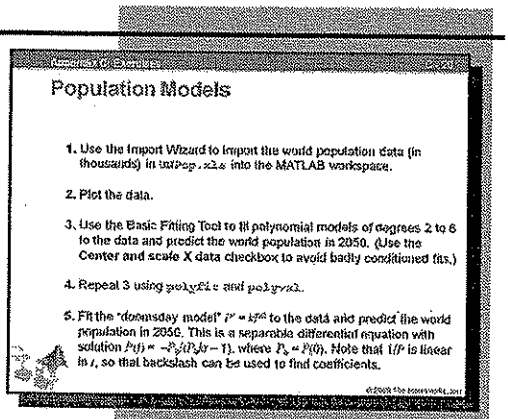
1. Load the data in `ECM.mat` into the MATLAB workspace. The data give current `I` and applied DC voltage `V` for a black box circuit.
2. Create a scatter plot of `I` vs. `V`.
3. Use backslash to fit a straight line model through the origin. Add a plot of the model to the displayed data. Compute the linear resistance `R` of the circuit, given by Ohm's Law `R = V./I`.
4. Use `polyfit` to fit a cubic polynomial model to the data. Add the model to the plot using `polyval`.
5. Use backslash to fit the model `I = a + b*V + c*exp(V)` to the data. Add the model to the plot.
6. Plot the residuals for each model.

SOLUTION

```
>> edit ECM
>> ECM
```



Population Models



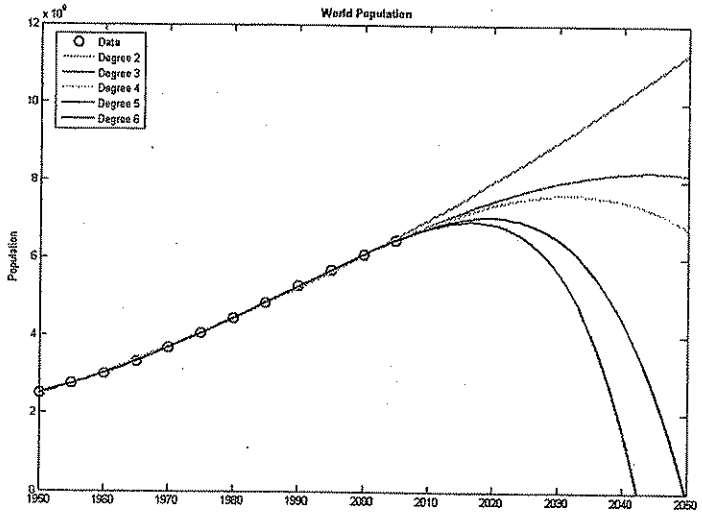
Reference: Section 6
Topics: Curve fitting, polynomial models, nonlinear models

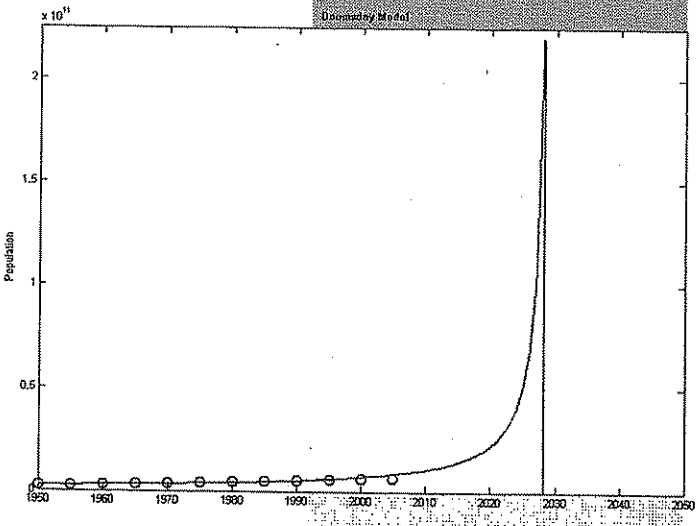
Exercise

1. Use the Import Wizard to import the world population data (in thousands) in `UNPop.xls` into the MATLAB workspace.
2. Plot the data.
3. Use the Basic Fitting Tool to fit polynomial models of degrees 2 to 6 to the data and predict the world population in 2050. (Use the **Center and scale X data** check box to avoid badly conditioned fits.)
4. Repeat 3. using `polyfit` and `polyval`.
5. Fit the "doomsday model" $P' = kP^2$ to the data and predict the world population in 2050. This equation is a separable differential with the solution $P(t) = -P_0/(P_0kt - 1)$, where $P_0 = P(0)$. Note that $1/P$ is linear in t, so that backslash can be used to find coefficients.

SOLUTION

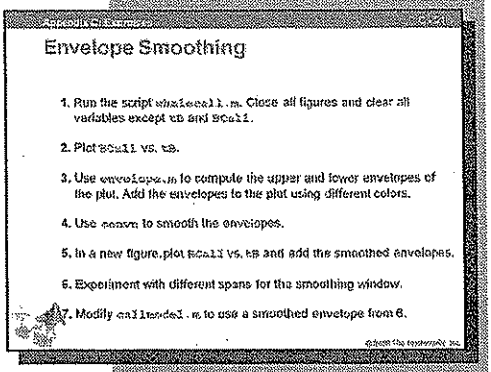
```
>> edit worldpop
>> worldpop
```





Envelope Smoothing

Reference: Section 6
Topics: 1-D convolution and smoothing

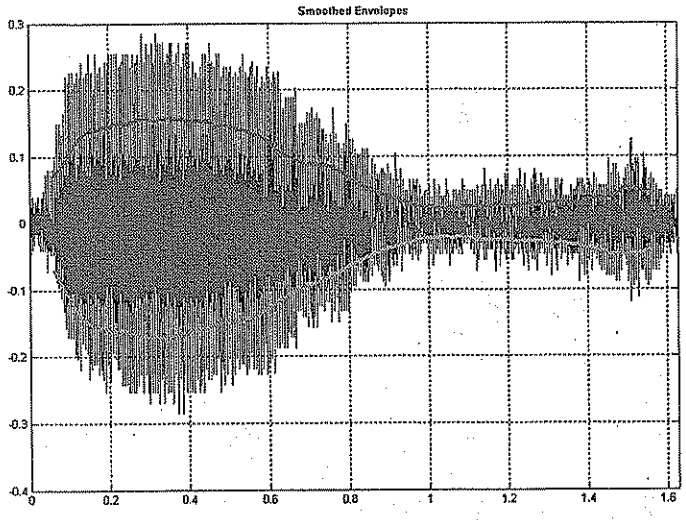


Exercise

1. Run the script `whalecall.m`. Close all figures and clear all variables except `tB` and `BCall`.
2. Plot `BCall` vs. `tB`.
3. Use `envelope.m` to compute the upper and lower envelopes of the plot. Add the envelopes to the plot using different colors.
4. Use `convn` to smooth the envelopes.
5. In a new figure, plot `BCall` vs. `tB` and add the smoothed envelopes.
6. Experiment with different spans for the smoothing window.
7. Modify `callmodel.m` to use a smoothed envelope from 6.

SOLUTION

```
>> edit whalesmooth
>> whalesmooth

>> edit ...
callmodel_smooth
>> callmodel_smooth
>> sound(call,fs)
>> callsnip = ...
call(~isnan(call));
>> sound(...
callsnip,fs)
```



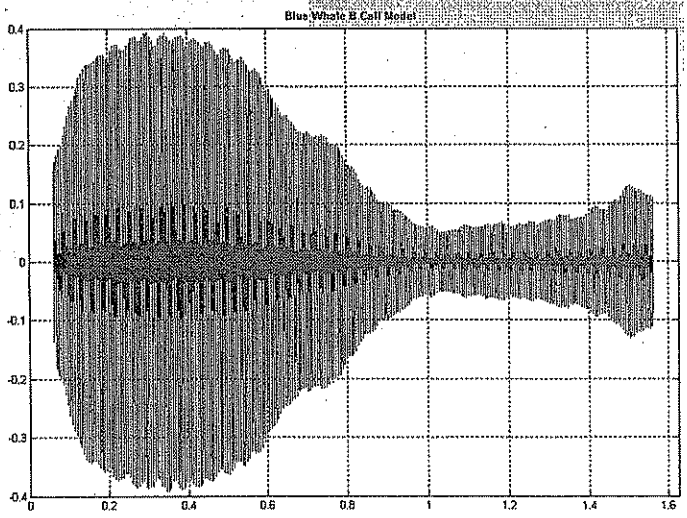
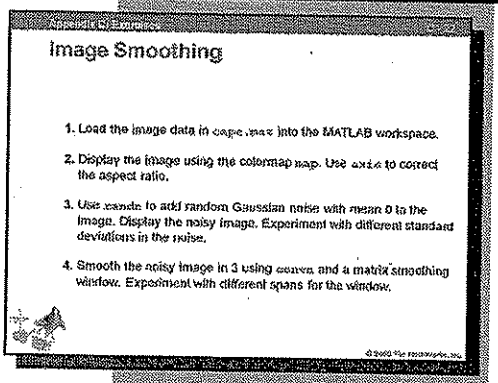


Image Smoothing



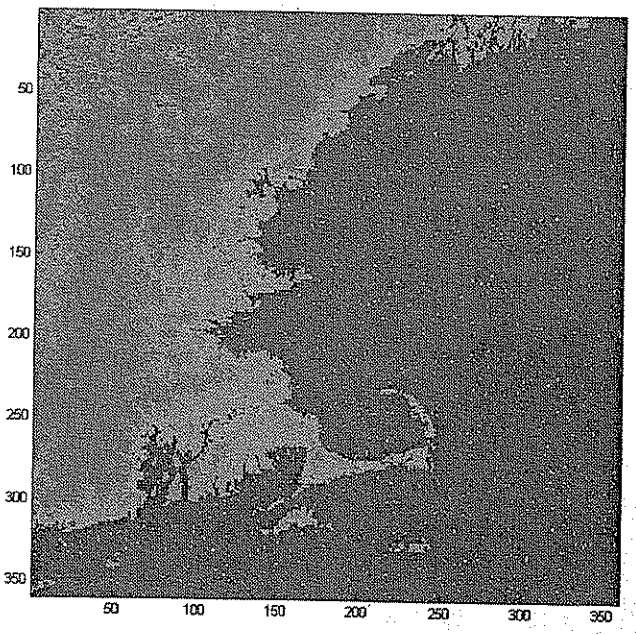
Reference: Section 6
Topics: 2-D convolution and smoothing

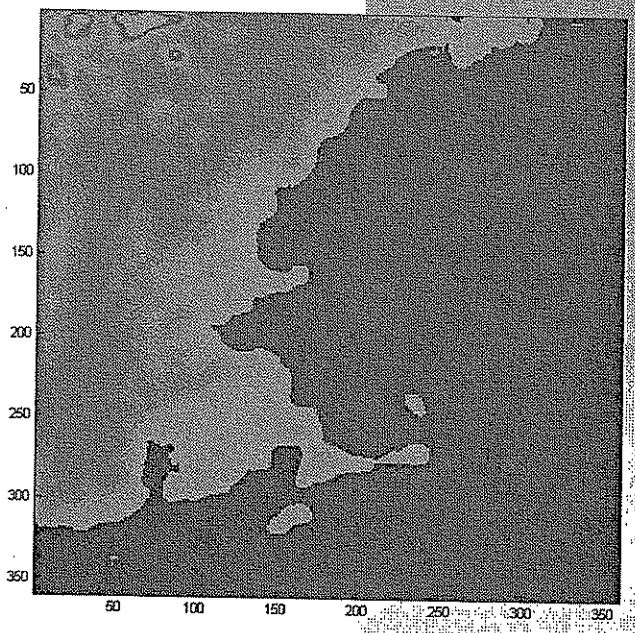
Exercise

1. Load the image data in `cape.mat` into the MATLAB workspace.
2. Display the image using the colormap `map`. Use `axis` to correct the aspect ratio.
3. Use `randn` to add random Gaussian noise with mean 0 to the image. Display the noisy image. Experiment with different standard deviations in the noise.
4. Smooth the noisy image in 3. using `convn` and a matrix-smoothing window. Experiment with different spans for the window.

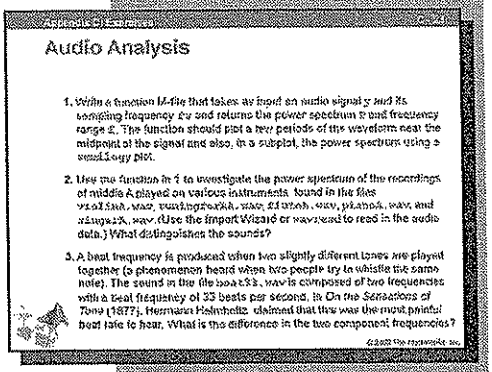
SOLUTION

```
>> edit capesmooth
>> capesmooth
```





Audio Analysis


Audio Analysis

1. Write a function M-file that takes as input an audio signal y and its sampling frequency fs and returns the power spectrum P and frequency range f. The function should plot a few periods of the waveform near the midpoint of the signal and also, in a subplot, the power spectrum using a semilogy plot.
2. Use the function in 1 to investigate the power spectrum of the recordings of middle A played on various instruments found in the files violinA.wav, tuningforkA.wav, fluteA.wav, pianoA.wav, and singerA.wav. (Use the Import Wizard or wavread to read in the audio data.) What distinguishes the sounds?
3. A beat frequency is produced when two slightly different tones are played together (a phenomenon heard when two people try to whistle the same note). The sound in the file beat33.wav is composed of two frequencies with a beat frequency of 33 beats per second. In *On the Sensations of Tone* (1877), Hermann Helmholtz claimed that this was the most painful beat rate to hear. What is the difference in the two component frequencies?

Reference: Section 6
Topics: FFT, alternative axes, indexing

Exercise

1. Write a function M-file that takes as input an audio signal `y` and its sampling frequency `fs`, and returns the power spectrum `P` and frequency range `f`. The function should plot a few periods of the waveform near the midpoint of the signal and also, in a subplot, the power spectrum using a `semilogy` plot.

2. Use the function in 1 to investigate the power spectrum of the recordings of middle A played on various instruments, found in the files `violinA.wav`, `tuningforkA.wav`, `fluteA.wav`, `pianoA.wav`, and `singerA.wav`. (Use the Import Wizard or `wavread` to read in the audio data.) What distinguishes the sounds?

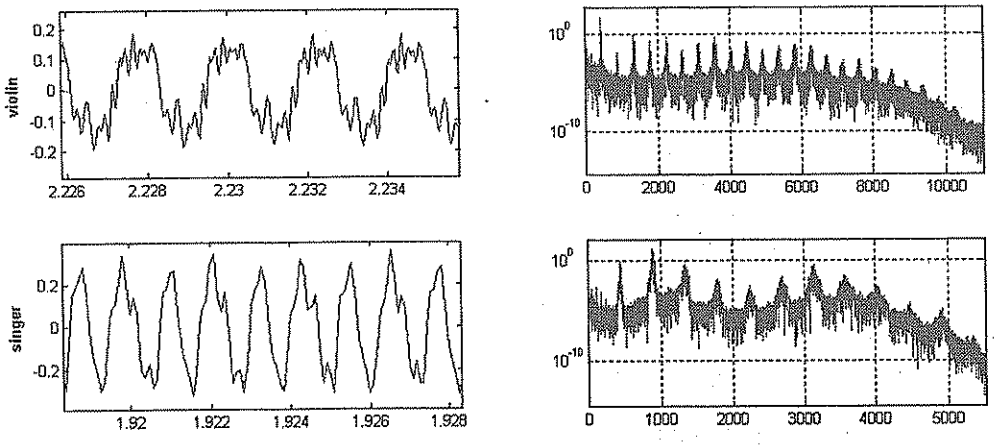
3. A *beat frequency* is produced when two slightly different tones are played together (a phenomenon heard when two people try to whistle the same note). The sound in the file `beat33.wav` is composed of two frequencies with a beat frequency of 33 beats per second. In *On the Sensations of Tone* (1877), Hermann Helmholtz claimed that this was the most painful beat rate to hear. What is the difference in the two component frequencies?

SOLUTION

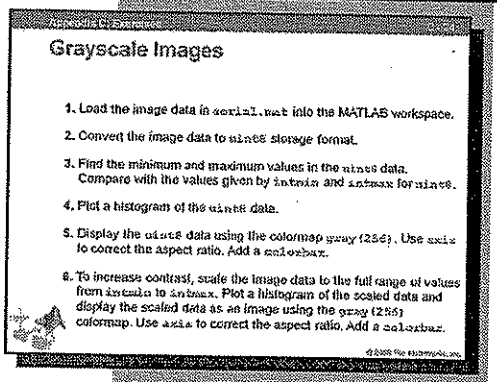
1.
```
>> edit audiospec
```

2.
```
>> edit midApower
>> midApower
```

3.
```
>> edit eardrum
>> eardrum
```
For hearty ears only
```
>> soundsc(y,fs)
```
Is the sound "painful" by modern standards?



Grayscale Images



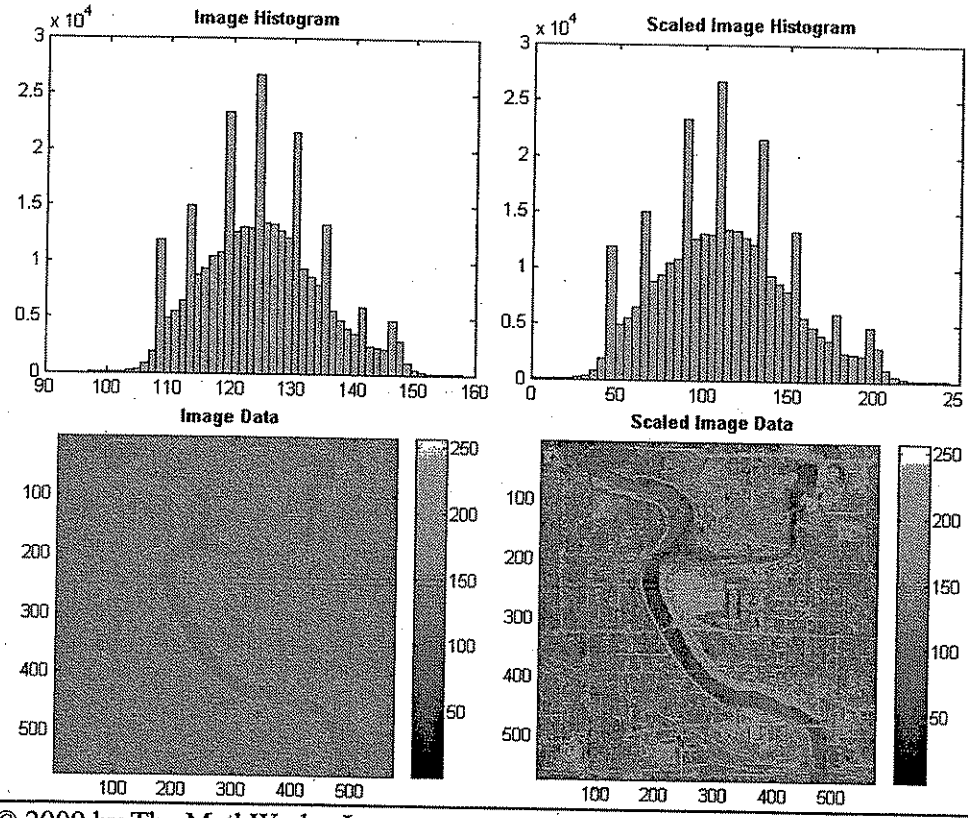
Reference: Section 7
Topics: Integer data types

Exercise

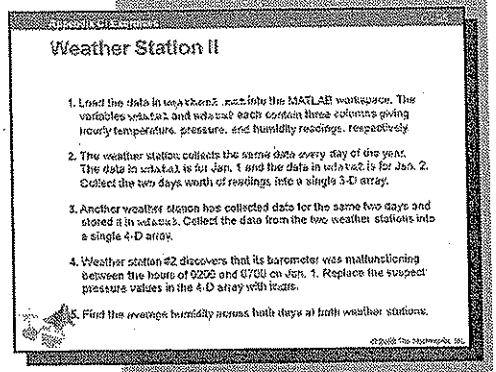
1. Load the image data in `aerial.mat` into the MATLAB workspace.
2. Convert the image data to `uint8` storage format.
3. Find the minimum and maximum values in the `uint8` data. Compare with the values given by `intmin` and `intmax` for `uint8`.
4. Plot a histogram of the `uint8` data.
5. Display the `uint8` data using the colormap `gray(256)`. Use `axis` to correct the aspect ratio. Add a colorbar.
6. To increase contrast, scale the image data to the full range of values from `intmin` to `intmax`. Plot a histogram of the scaled data and display the scaled data as an image using the `gray(256)` colormap. Use `axis` to correct the aspect ratio. Add a colorbar.

SOLUTION

```
>> edit grayscale
>> grayscale
```



Weather Station II


Weather Station II

1. Load the data in weather2.mat into the MATLAB workspace. The variables wdata1 and wdata2 each contain three columns giving hourly temperature, pressure, and humidity readings, respectively.
2. The weather station collects the same data every day of the year. The data in wdata1 is for Jan. 1 and the data in wdata2 is for Jan. 2. Collect the two days worth of readings into a single 3-D array.
3. Another weather station has collected data for the same two days and stored it in wdata3. Collect the data from the two weather stations into a single 4-D array.
4. Weather station #2 discovers that its barometer was malfunctioning between the hours of 0200 and 0700 on Jan. 1. Replace the suspect pressure values in the 4-D array with NaNs.
5. Find the average humidity across both days at both weather stations.

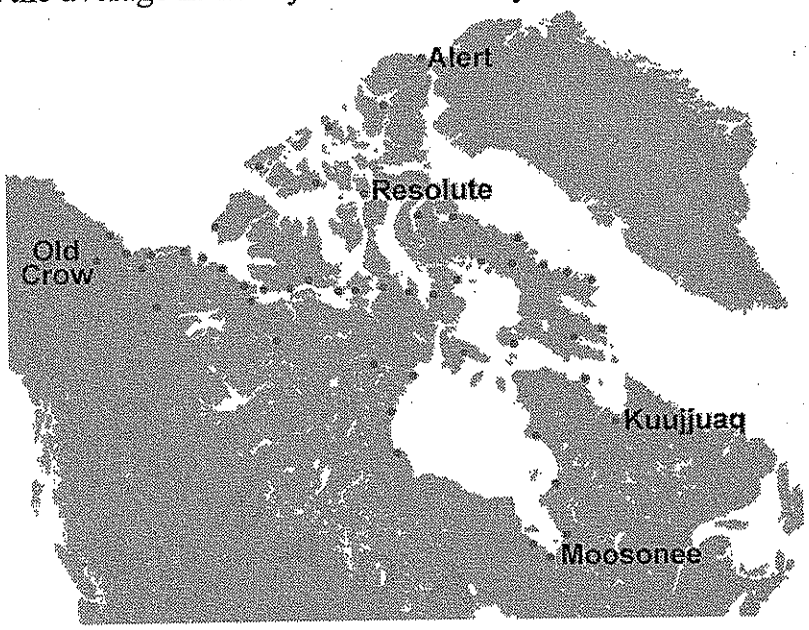
Reference: Section 7
Topics: Multidimensional arrays

Exercise

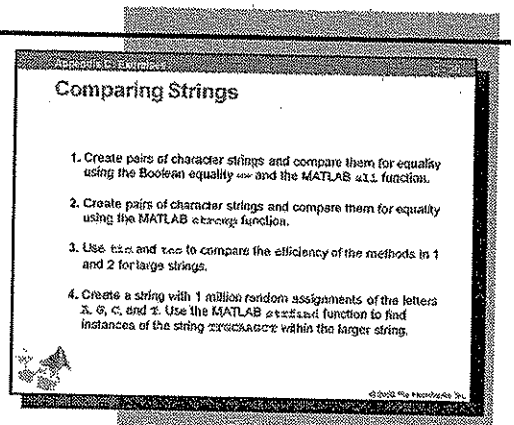
1. Load the data in `weather2.mat` into the MATLAB workspace. The variables `wdata1` and `wdata2` each contain three columns giving temperature, pressure, and humidity readings, respectively, collected hourly.
2. The weather station collects the same data every day of the year. The data in `wdata1` is for Jan. 1, and the data in `wdata2` is for Jan. 2. Collect the two days worth of readings into a single 3-D array.
3. Another weather station has collected data for the same two days and stored it in `wdata3`. Collect the data from the two weather stations into a single 4-D array.
4. Weather station 2 discovers that its barometer was malfunctioning between the hours of `0200` and `0700` on Jan. 1. Replace the suspect pressure values in the 4-D array with `NaNs`.
5. Find the average humidity across both days at both weather stations.

SOLUTION

```
>> edit weather2
>> weather2
```



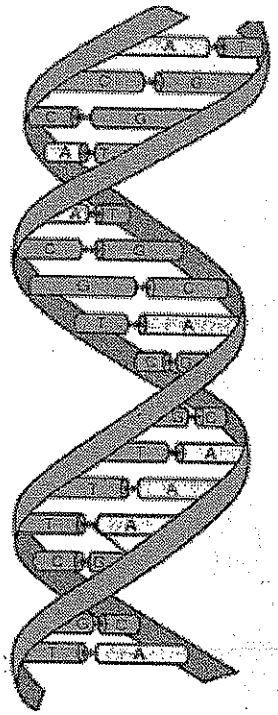
Comparing Strings



Reference: Section 7
Topics: String functions, basic programming

Exercise

1. Create pairs of character strings and compare them for equality using the Boolean equality == and the MATLAB `all` function.
2. Create pairs of character strings and compare them for equality using the MATLAB `strcmp` function.
3. Use `tic` and `toc` to compare the efficiency of the methods in 1 and 2 for large strings.
4. Create a string with 1 million random assignments of the letters `A`, `G`, `C`, and `T`. Use the MATLAB `strfind` function to find instances of the string `TTGCAAGCT` within the larger string.

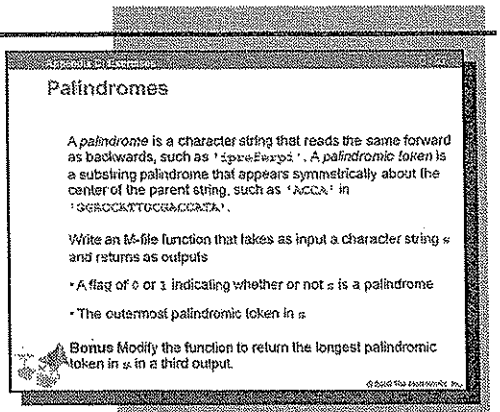


SOLUTION

```
>> edit strfcns
>> strfcns
```

Palindromes

Reference: Section 7
Topics: Strings, logical indexing, basic programming



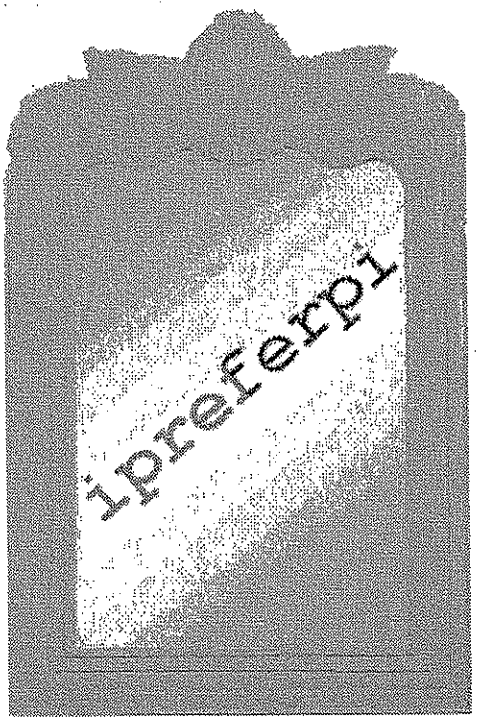
Exercise

A *palindrome* is a character string that reads the same forward as backwards, such as `'ipreferpi'`. A *palindromic token* is a substring that appears symmetrically about the center of the parent string, such as `'ATCA'` in `'GGATCATTGCGACTATA'`.

Write an M-file function that takes as input a character string `s` and returns as outputs

- A flag of `0` or `1` indicating whether or not `s` is a palindrome
- The outermost palindromic token in `s`

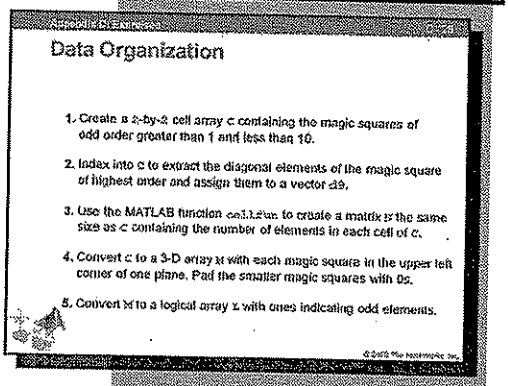
Bonus Modify the function to return the longest palindromic token in `s` in a third output.



SOLUTION

```
>> edit palindrome
>> [F,T] = ...
palindrome(...
'ipreferpi')
>> [F,T] = ...
palindrome(...
'GGATCATTGCGACTATA')
```

Data Organization



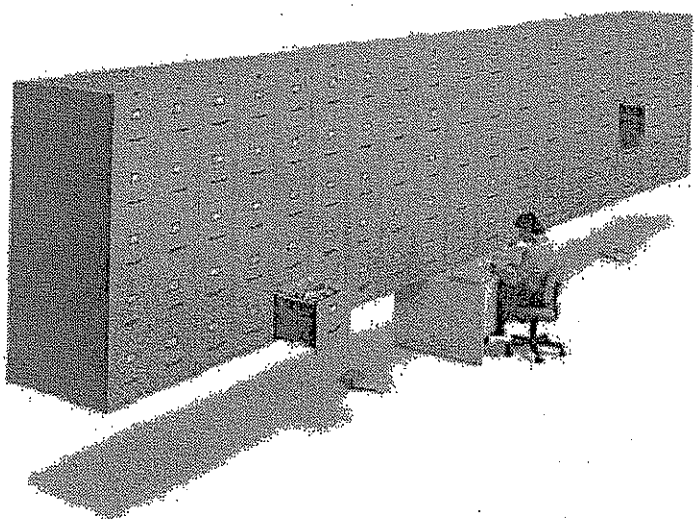
Reference: Section 7
Topics: Cell arrays, multidimensional arrays, logical arrays

Exercise

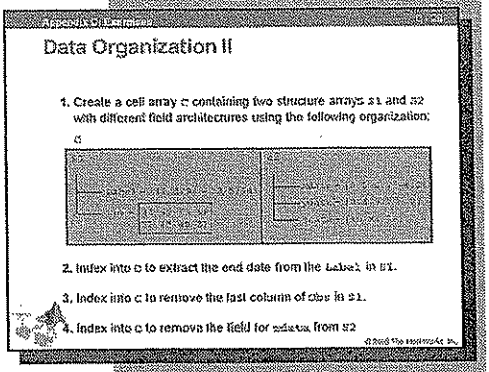
1. Create a 2-by-2 cell array `C` containing the magic squares of odd order greater than 1 and less than 10.
2. Index into `C` to extract the diagonal elements of the magic square of highest order and assign them to a vector `d9`.
3. Use the MATLAB function `cellfun` to create a matrix `N` the same size as `C` containing the number of elements in each cell of `C`.
4. Convert `C` to a 3-D array `M` with each magic square in the upper-left corner of one plane. Pad the smaller magic squares with 0s.
5. Convert `M` to a logical array `L` with ones indicating odd elements.

SOLUTION

```
>> edit dataorg
>> dataorg
```



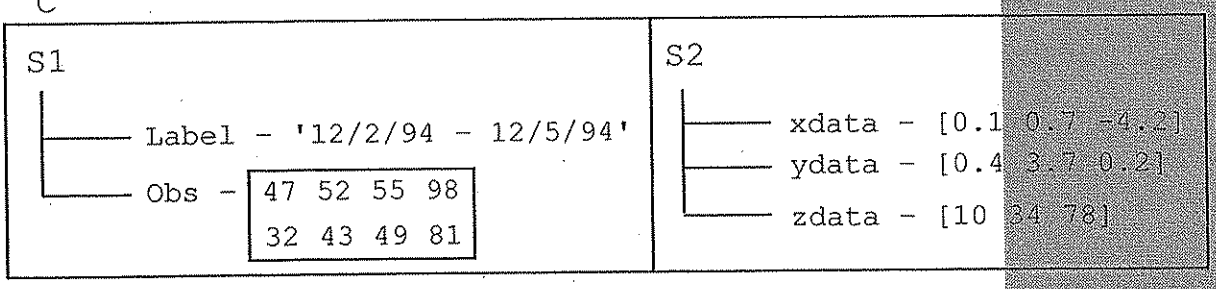
Data Organization II



Reference: Section 7
Topics: Cell arrays, structure arrays

Exercise

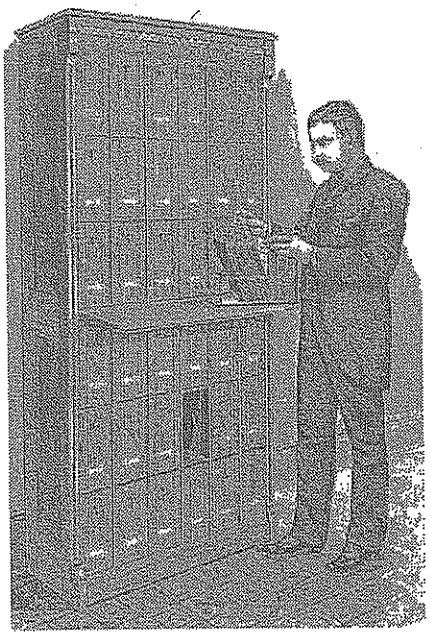
1. Create a cell array `C` containing two structure arrays `S1` and `S2` with different field architectures using the following organization:



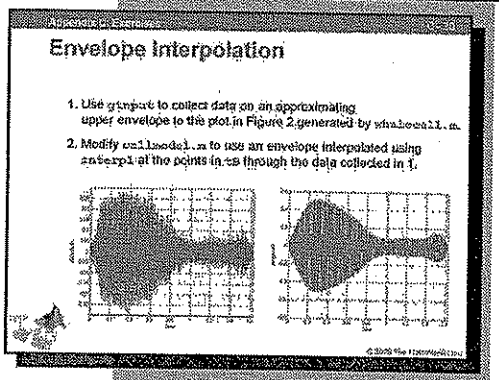
2. Index into `C` to extract the end date from the `Label` in `S1`.
3. Index into `C` to remove the last column of `Obs` in `S1`.
4. Index into `C` to remove the field for `zdata` from `S2`.

SOLUTION

```
>> edit dataorg2
>> dataorg2
```



Envelope Interpolation



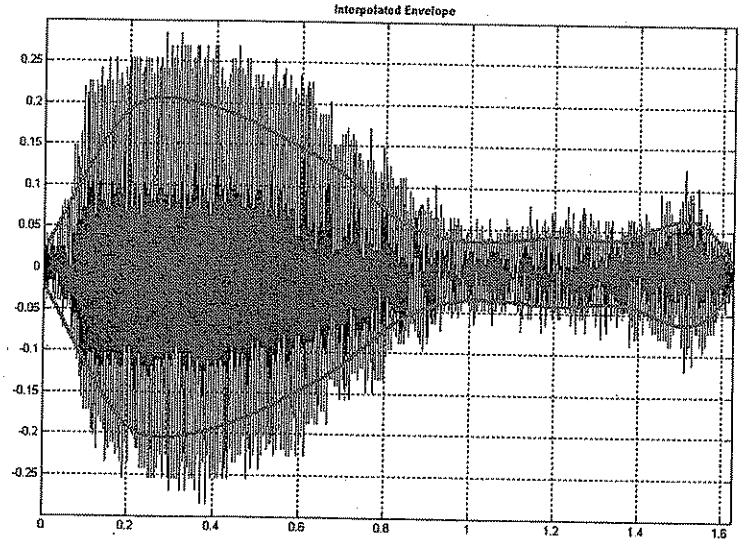
Reference: Section 8
Topics: Graphical input, data interpolation, modeling

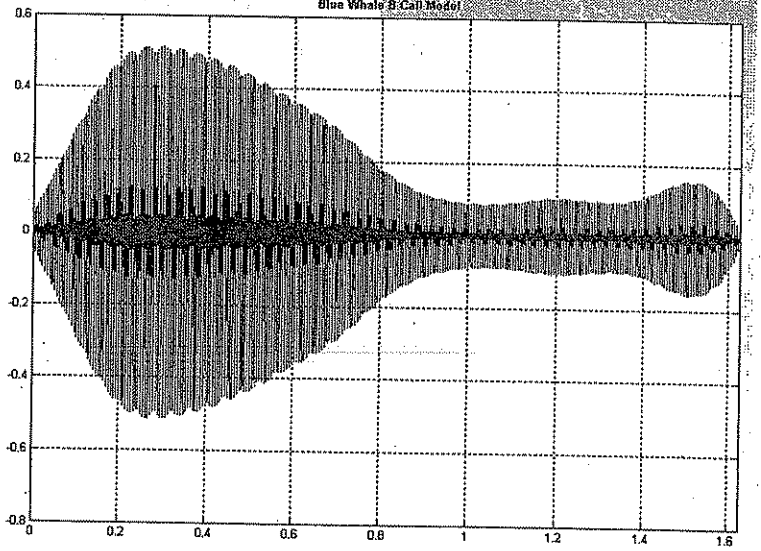
Exercise

1. Use `ginput` to collect data on an approximating upper envelope to the plot in Figure 2 generated by `whalecall.m`.
2. Modify `callmodel.m` to use an envelope interpolated using `interp1` at the points in `tB` through the data collected in 1.

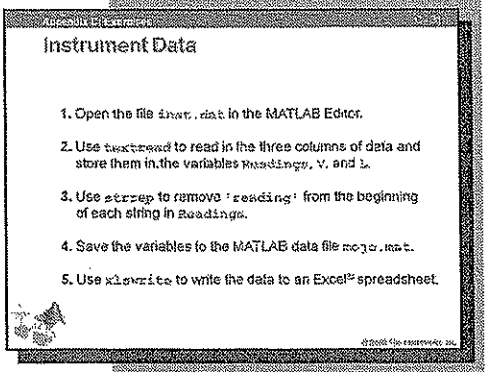
SOLUTION

```
>> edit ...
callmodel_interp
>> callmodel_interp
>> sound(call,fs)
```





Instrument Data



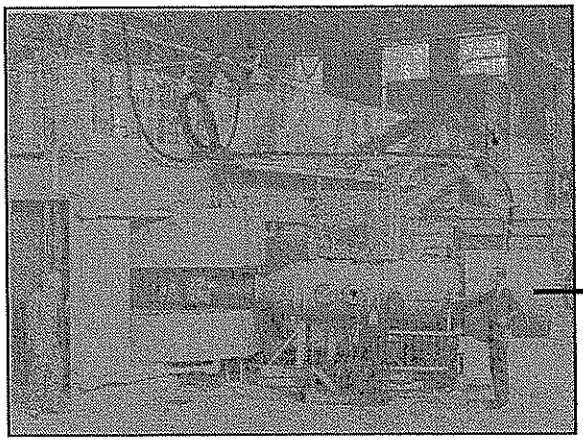
Reference: Section 8
Topics: File I/O, `*read` and `*write` functions, string functions

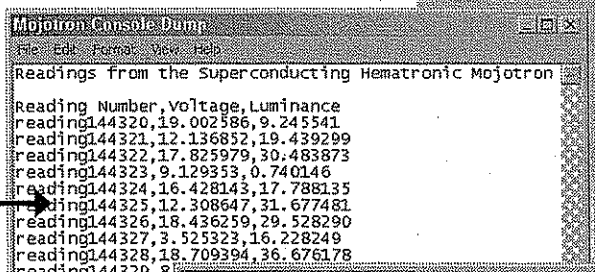
Exercise

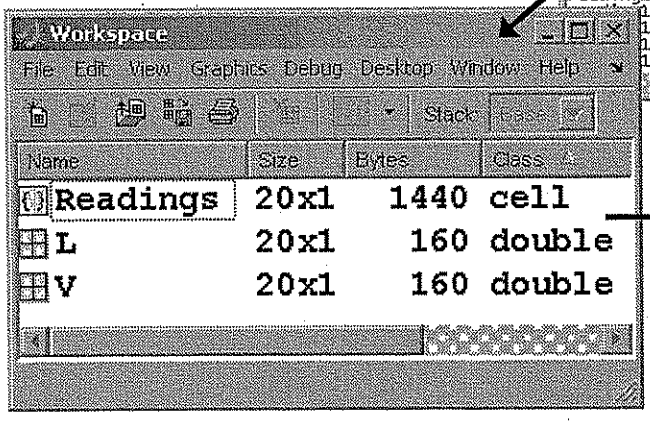
1. Open the file `inst.dat` in the MATLAB Editor.
2. Use `textread` to read in the three columns of data and store them in the variables `Readings`, `V`, and `L`.
3. Use `strrep` to remove `'reading'` from the beginning of each string in `Readings`.
4. Save the variables to the MATLAB data file `mojo.mat`.
5. Use `xlswrite` to write the data to an Excel® spreadsheet.

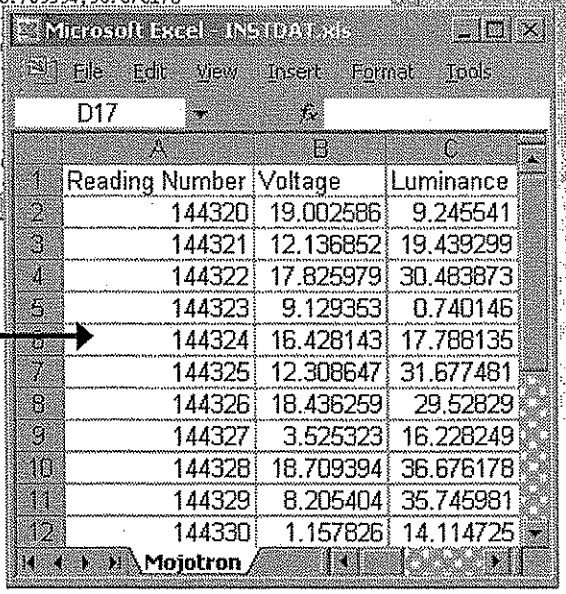
SOLUTION

```
>> edit instdat
>> instdat
```

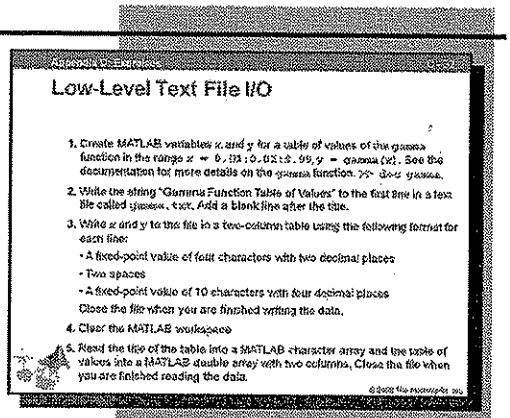








Low-Level Text File I/O



Reference: Section 8
Topics: Low-level file I/O functions

Exercise

1. Create MATLAB variables `x` and `y` for a table of values of the `gamma` function in the range `x = 0.01:0.01:9.99`, `y = gamma(x)`. See the documentation for more details on the gamma function, `>> doc gamma`.

2. Write the string "Gamma Function Table of Values" to the first line in a text file called `gamma.txt`. Add a blank line after the title.

3. Write `x` and `y` to the file in a two-column table using the following format for each line:

   • A fixed-point value of four characters with two decimal places

   • Two spaces

   • A fixed-point value of 10 characters with four decimal places

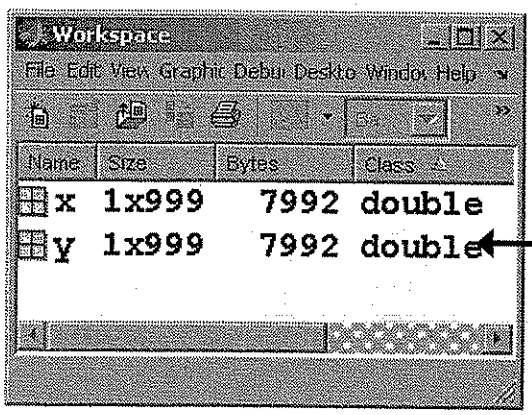
   Close the file when you are finished writing the data.

4. Clear the MATLAB workspace.

5. Read the title of the table into a MATLAB character array and the table of values into a MATLAB double array with two columns. Close the file when you are finished reading the data.

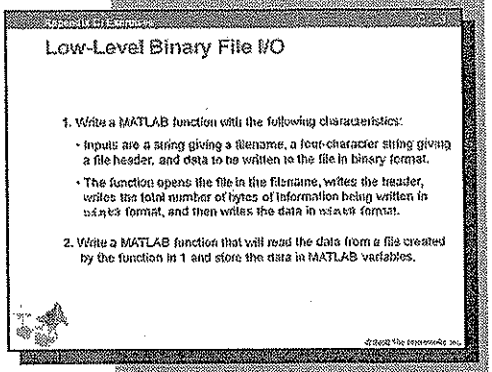
SOLUTION

```
>> edit textio
>> textio
```




```
Gamma Function Table of Values

0.01      99.4326
0.02      49.4422
0.03      32.7850
0.04      24.4610
0.05      19.4701
0.06      16.1457
0.07      13.7736
0.08      11.9966
0.09      10.6162
0.10       9.5135
0.11       8.6127
0.12       7.8633
```

Low-Level Binary File I/O



Reference: Section 8
Topics: Low-level file I/O functions

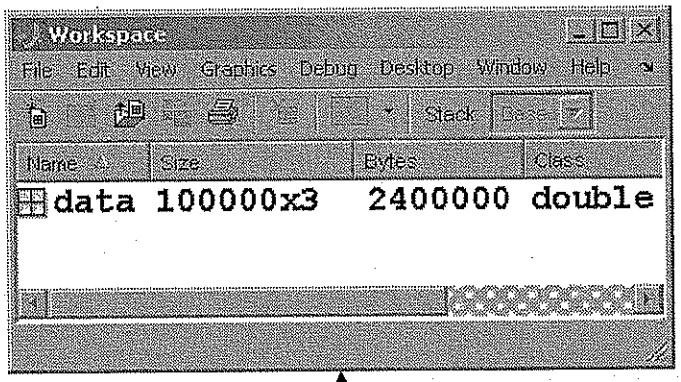
Exercise

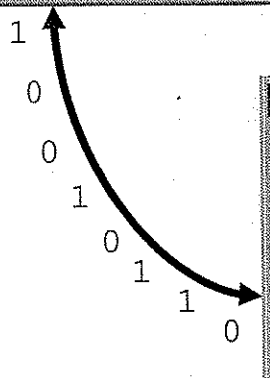
1. Write a MATLAB function with the following characteristics:
   - Inputs are a string giving a filename, a four-character string giving a file header, and data to be written to the file in binary format.
   - The function opens the file in the filename, writes the header, writes the total number of bytes of information being written in `uint8` format, and then writes the data in `uint8` format.

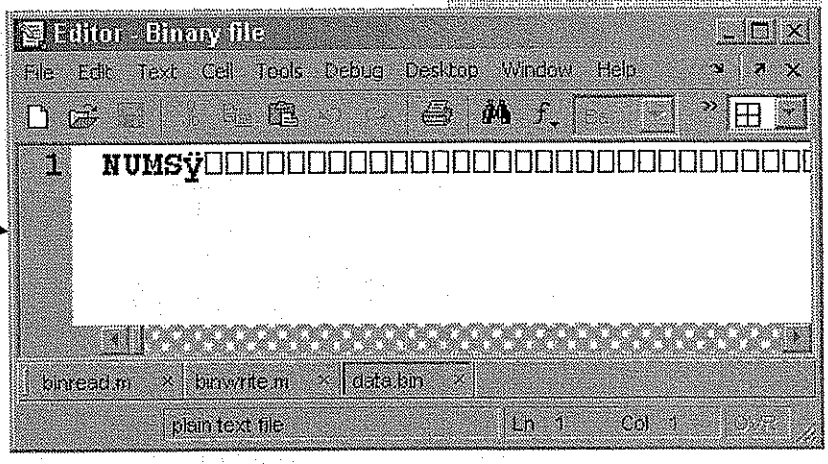
2. Write a MATLAB function that reads the data from a file created by the function in 1 and stores the data in MATLAB variables.

SOLUTION

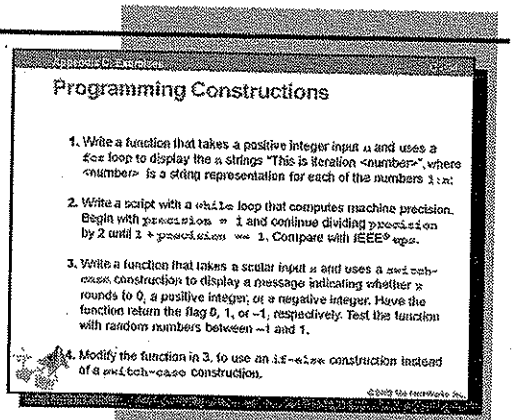
```
>> edit binwrite
>> binwrite(...
'test1.bin',...
'NUMS',magic(3))
>> binwrite(...
'test2.bin',...
'CHRS','Matlab')

>> edit binread
>> [data,length,...
head] = binread(...
'test1.bin')
>> reshape(...
data,3,3)'
>> num = length/8
>> char(head)
>> [data,length,...
head] = binread(...
'test2.bin')
>> char(data)
>> num = length/8
>> char(head)
```







Programming Constructions


Programming Constructions

1. Write a function that takes a positive integer input n and uses a for loop to display the n strings "This is iteration <number>", where <number> is a string representation for each of the numbers 1:n.

2. Write a script with a while loop that computes machine precision. Begin with precision = 1 and continue dividing precision by 2 until 1 + precision == 1. Compare with IEEE® eps.

3. Write a function that takes a scalar input x and uses a switch-case construction to display a message indicating whether x rounds to 0, a positive integer, or a negative integer. Have the function return the flag 0, 1, or –1, respectively. Test the function with random numbers between –1 and 1.

4. Modify the function in 3. to use an if-else construction instead of a switch-case construction.

Reference: Section 8
Topics: Basic programming constructions

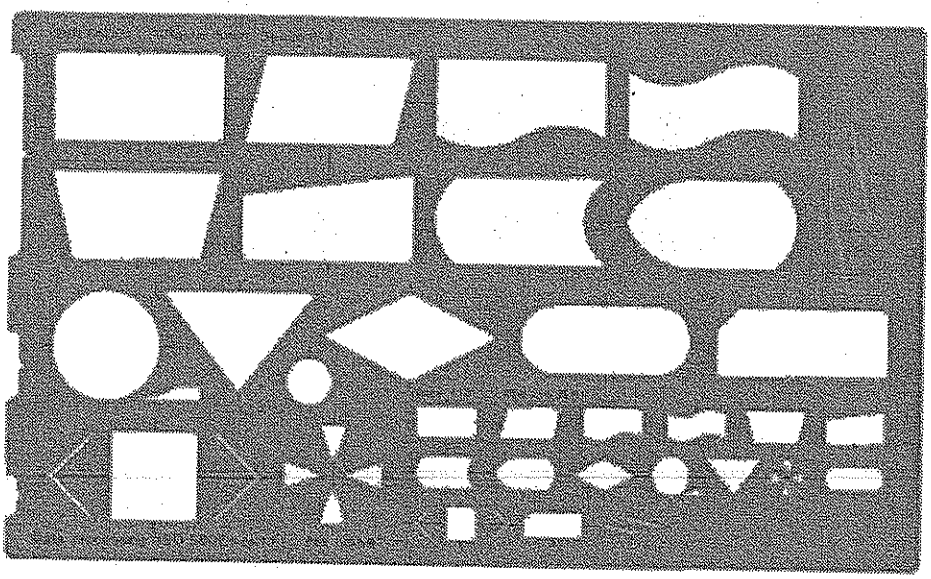
Exercise

1. Write a function that takes a positive integer input `n` and uses a `for`-loop to display the `n` strings "This is iteration <number>", where <number> is a string representation for each of the numbers `1:n`.

2. Write a script with a `while`-loop that computes machine precision. Begin with `precision = 1` and continue dividing `precision` by 2 until `1 + precision == 1`. Compare with IEEE® `eps`.

3. Write a function that takes a scalar input `x` and uses a `switch-case` construction to display a message indicating whether `x` rounds to 0, a positive integer, or a negative integer. Have the function return the flag 0, 1, or –1, respectively. Test the function with random numbers between –1 and 1.

4. Modify the function in 3. to use an `if-else` construction instead of a `switch-case` construction.



SOLUTION

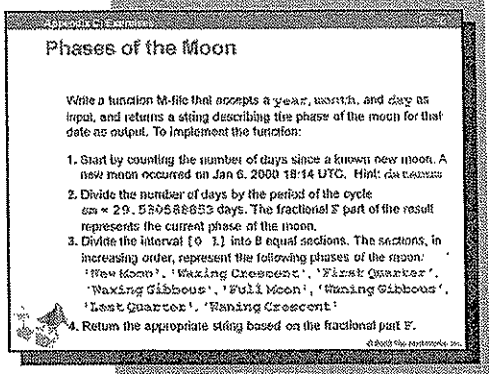
1.
```
>> edit basicFor
>> basicFor(5)
```

2.
```
>> edit basicWhile
>> basicWhile
```

3.
```
>> edit basicSwitch
>> edit switchtest
>> switchtest
```

4.
```
>> edit basicIf
>> flag = ...
basicIf(1.3)
>> flag = ...
basicIf(0.3)
>> flag = ...
basicIf(-1.3)
```

Phases of the Moon


Phases of the Moon

Write a function M-file that accepts a year, month, and day as input, and returns a string describing the phase of the moon for that date as output. To implement the function:

1. Start by counting the number of days since a known new moon. A new moon occurred on Jan 6, 2000 18:14 UTC. Hint: datenum
2. Divide the number of days by the period of the cycle sm = 29.530588853 days. The fractional F part of the result represents the current phase of the moon.
3. Divide the interval [0 1] into 8 equal sections. The sections, in increasing order, represent the following phases of the moon: 'New Moon', 'Waxing Crescent', 'First Quarter', 'Waxing Gibbous', 'Full Moon', 'Waning Gibbous', 'Last Quarter', 'Waning Crescent'
4. Return the appropriate string based on the fractional part F.

Reference: Section 5, 8
Topics: Function M-Files, Programming

Exercise

Write a function M-file that accepts a year, month, and day as input, and returns a string describing the phase of the moon for that date as output. To implement the function:

1. Start by counting the number of days since a known new moon. A new moon occurred on Jan 6, 2000 18:14 UTC. Hint: `datenum`

2. Divide the number of days by the period of the cycle sm = 29.530588853 days. The fractional F part of the result represents the current phase of the moon.

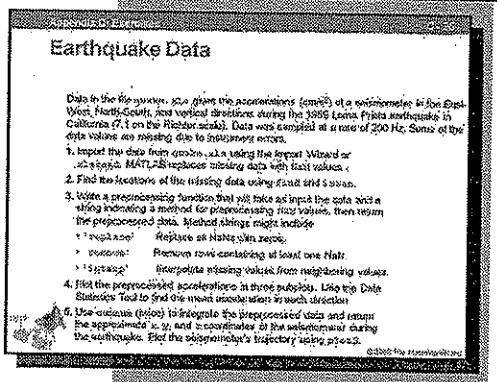
3. Divide the interval [0 1] into 8 equal sections. The sections, in increasing order, represent the following phases of the moon: `'New Moon', 'Waxing Crescent', 'First Quarter', 'Waxing Gibbous', 'Full Moon', 'Waning Gibbous', 'Last Quarter', 'Waning Crescent'`

4. Return the appropriate string based on the fractional part F.

SOLUTION

```
>> year = 2006;
>> month = 6;
>> day = 30;
>> str = moon(...
year,month,day)
```

Earthquake Data


Earthquake Data

Data in the file quake.xls gives the accelerations (cm/s²) of a seismometer in the East-West, North-South, and vertical directions during the 1989 Loma Prieta earthquake in California (7.1 on the Richter scale). Data was sampled at a rate of 200 Hz. Some of the data values are missing due to instrument errors.

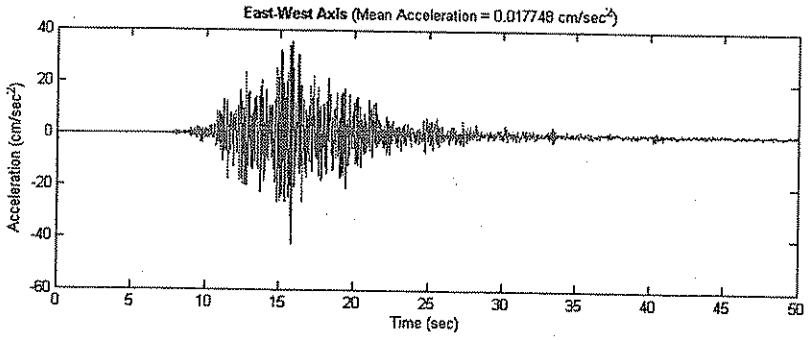
1. Import the data from quake.xls using the Import Wizard or xlsread. MATLAB replaces missing data with NaN values.
2. Find the locations of the missing data using find and isnan.
3. Write a preprocessing function that takes as input the data and a string indicating a method for preprocessing NaN values, then returns the preprocessed data. Method strings might include
4. Plot the preprocessed accelerations in three subplots. Use the Data Statistics Tool to find the mean acceleration in each direction.
5. Use cumsum (twice) to integrate the preprocessed data and return the approximate x, y, and z coordinates of the seismometer during the earthquake. Plot the seismometer's trajectory using plot3.

Reference: Section 8
Topics: Data preprocessing, missing values, interpolation, switch-case constructions, logical indexing

Exercise

Data in the file `quake.xls` gives the accelerations (cm/s^2) of a seismometer in the East-West, North-South, and vertical directions during the 1989 Loma Prieta earthquake in California (7.1 on the Richter scale). Data was sampled at a rate of 200 Hz. Some of the data values are missing due to instrument errors.

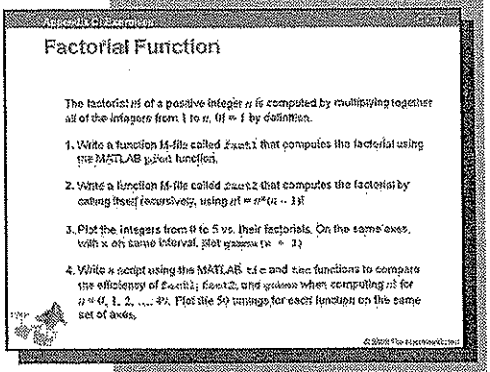
1. Import the data from `quake.xls` using the Import Wizard or `xlsread`. MATLAB replaces missing data with `NaN` values.
2. Find the locations of the missing data using `find` and `isnan`.
3. Write a preprocessing function that takes as input the data and a string indicating a method for preprocessing `NaN` values, then returns the preprocessed data. Method strings might include
   - `'replace'` Replace all `NaN`s with zeros.
   - `'remove'` Remove rows containing at least one `NaN`.
   - `'interp'` Interpolate missing values from neighboring values.
4. Plot the preprocessed accelerations in three subplots. Use the Data Statistics Tool to find the mean acceleration in each direction.
5. Use `cumsum` (twice) to integrate the preprocessed data and return the approximate `x`, `y`, and `z` coordinates of the seismometer during the earthquake. Plot the seismometer's trajectory using `plot3`.



SOLUTION

```
>> edit shake
>> shake
```

Factorial Function


Factorial Function

The factorial $n!$ of a positive integer n is computed by multiplying together all of the integers from 1 to n. $0! = 1$ by definition.

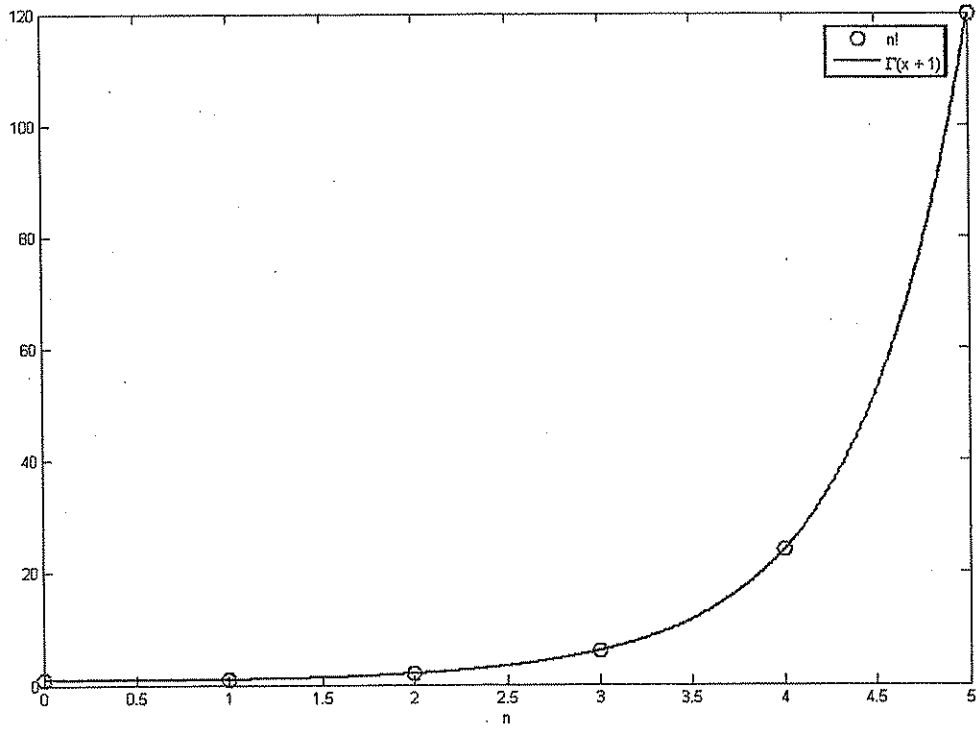
1. Write a function M-file called `fact1` that computes the factorial using the MATLAB `prod` function.
2. Write a function M-file called `fact2` that computes the factorial by calling itself recursively, using $n! = n*(n-1)!$
3. Plot the integers from 0 to 5 vs. their factorials. On the same axes, with `x` on same interval, plot `gamma(x + 1)`.
4. Write a script using the MATLAB `tic` and `toc` functions to compare the efficiency of `fact1`, `fact2`, and `gamma` when computing $n!$ for $n = 0, 1, 2, \ldots, 49$. Plot the 50 timings for each function on the same set of axes.

Reference: Section 8
Topics: Function M-files, plotting, code performance

Exercise

The factorial $n!$ of a positive integer n is computed by multiplying together all of the integers from 1 to n. $0! = 1$ by definition.

1. Write a function M-file called `fact1` that computes the factorial using the MATLAB `prod` function.
2. Write a function M-file called `fact2` that computes the factorial by calling itself recursively, using $n! = n(n-1)!$
3. Plot the integers from 0 to 5 vs. their factorials. On the same axes, with `x` on same interval, plot `gamma(x + 1)`.
4. Write a script using the MATLAB `tic` and `toc` functions to compare the efficiency of `fact1`, `fact2`, and `gamma` when computing $n!$ for $n = 0, 1, 2, \ldots, 49$. Plot the 50 timings for each function on the same set of axes.



SOLUTION

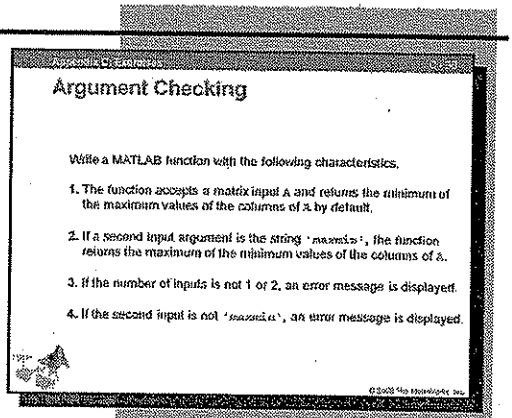
1.
```
>> edit fact1
>> fact1(5)
```

2.
```
>> edit fact2
>> fact2(5)
```

3.
```
>> edit factplot
>> factplot
```

4.
```
>> edit facttest
>> facttest
```

Argument Checking



Reference: Section 8 (Advanced topic)
Topics: Argument checking in function M-files

Exercise

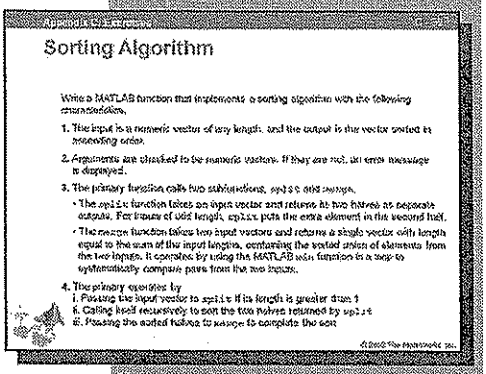
Write a MATLAB function with the following characteristics:

1. The function accepts a matrix input `A` and returns the minimum of the maximum values of the columns of `A` by default.
2. If a second input argument is the string `'maxmin'`, the function returns the maximum of the minimum values of the columns of `A`.
3. If the number of inputs is not 1 or 2, an error message is displayed.
4. If the second input is not `'maxmin'`, an error message is displayed.

SOLUTION

```
>> edit ArgChk
>> A = magic(5)
>> mM = ArgChk(A)
>> Mm = ArgChk(...
A,'maxmin')
>> test1 = ArgChk
>> test2 = ArgChk(...
A,2)
>> test3 = ArgChk(...
A,'maxmin',2)
```

Sorting Algorithm


Sorting Algorithm

Write a MATLAB function that implements a sorting algorithm with the following characteristics.

1. The input is a numeric vector of any length, and the output is the vector sorted in ascending order.
2. Arguments are checked to be numeric vectors. If they are not, an error message is displayed.
3. The primary function calls two subfunctions, `split` and `merge`.
   - The `split` function takes an input vector and returns its two halves as separate outputs. For inputs of odd length, `split` puts the extra element in the second half.
   - The `merge` function takes two input vectors and returns a single vector with length equal to the sum of the input lengths, containing the sorted union of elements from the two inputs. It operates by using the MATLAB `min` function in a loop to systematically compare pairs from the two inputs.
4. The primary operates by
   i. Passing the input vector to `split` if its length is greater than 1
   ii. Calling itself recursively to sort the two halves returned by `split`
   iii. Passing the sorted halves to `merge` to complete the sort

Reference: Section 8
Topics: MATLAB programming

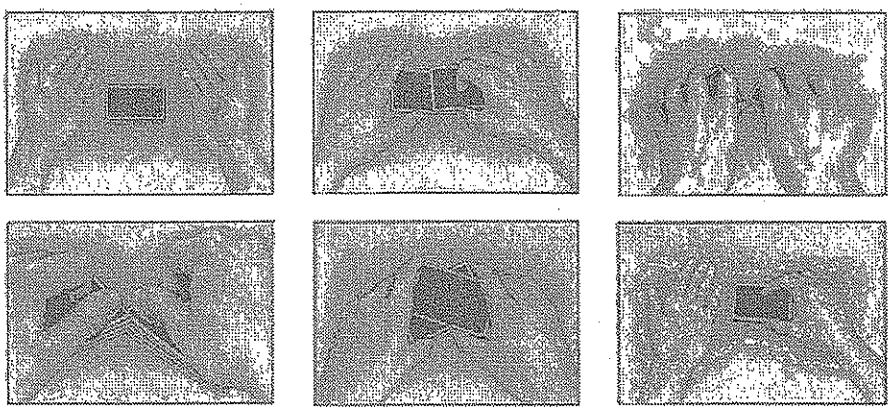
Exercise

Write a MATLAB function that implements a sorting algorithm with the following characteristics:

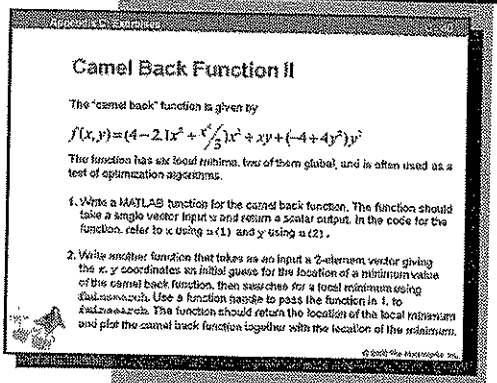
1. The input is a numeric vector of any length, and the output is the vector sorted in ascending order.
2. Arguments are checked to be numeric vectors. If they are not, an error message is displayed.
3. The primary function calls two subfunctions, `split` and `merge`.
   a. The `split` function takes an input vector and returns its two halves as separate outputs. For inputs of odd length, `split` puts the extra element in the second half.
   b. The `merge` function takes two input vectors and returns a single vector with length equal to the sum of the input lengths, containing the sorted union of elements from the two inputs. It operates by using the MATLAB `min` function in a loop to systematically compare pairs from the two inputs.
4. The primary operates by
   a. Passing the input vector to `split` if its length is greater than 1
   b. Calling itself recursively to sort the two halves returned by `split`
   c. Passing the sorted halves to merge to complete the sort

SOLUTION

```
>> edit mergeSort
>> s = ...
mergeSort(...
randperm(50));
>> issorted(s)
```



Camel Back Function II


Camel Back Function II

The "camel back" function is given by

$f(x,y)=(4-2.1x^2+\frac{x^4}{3})x^2+xy+(-4+4y^2)y^2$

The function has six local minima, two of them global, and is often used as a test of optimization algorithms.

1. Write a MATLAB function for the camel back function. The function should take a single vector input u and return a scalar output. In the code for the function, refer to x using u(1) and y using u(2).

2. Write another function that takes as an input a 2-element vector giving the x, y coordinates an initial guess for the location of a minimum value of the camel back function, then searches for a local minimum using fminsearch. Use a function handle to pass the function in 1. to fminsearch. The function should return the location of the local minimum and plot the camel back function together with the location of the minimum.

Reference: Section 7,8
Topics: Function handles, optimization

Exercise

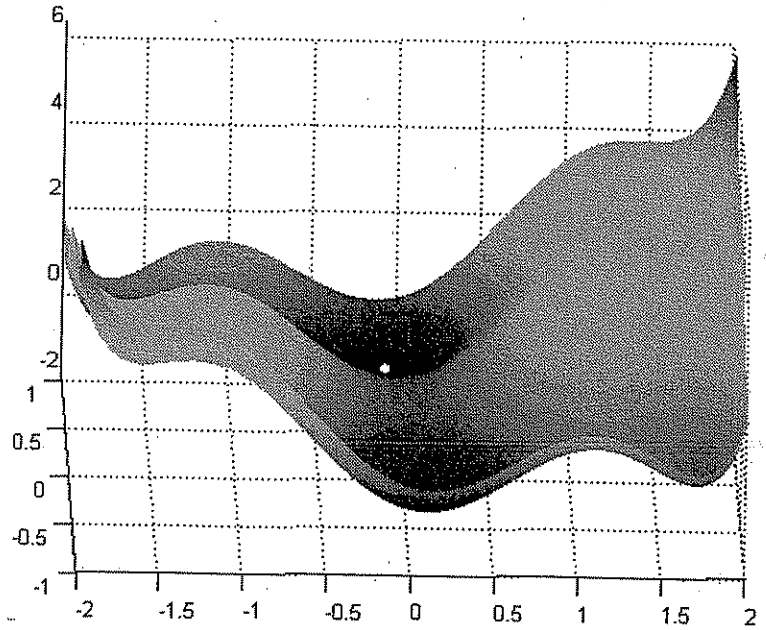
The "camel back" function is given by

$$f(x,y)=(4-2.1x^2+\frac{x^4}{3})x^2+xy+(-4+4y^2)y^2$$

The function has six local minima, two of them global, and is often used as a test of optimization algorithms.

1. Write a MATLAB function for the camel back function. The function should take a single vector input `u` and return a scalar output. In the code for the function, refer to `x` using `u(1)` and `y` using `u(2)`.

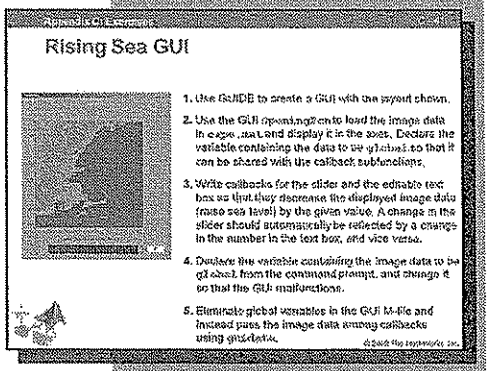
2. Write another function that takes as an input a 2-element vector giving the `x, y` coordinates of an initial guess for the location of a minimum value of the camel back function, then searches for a local minimum using `fminsearch`. Use a function handle to pass the function in 1. to `fminsearch`. The function should return the location of the local minimum and plot the camel back function together with the location of the minimum.



SOLUTION

```
>> edit cbfun
>> edit cbsearch
>> cbsearch([-2 -1])
>> close
>> cbsearch([-2 1])
>> close
>> cbsearch([0 -1])
>> close
>> cbsearch([0 1])
>> close
>> cbsearch([2 -1])
>> close
>> cbsearch([2 1])
```

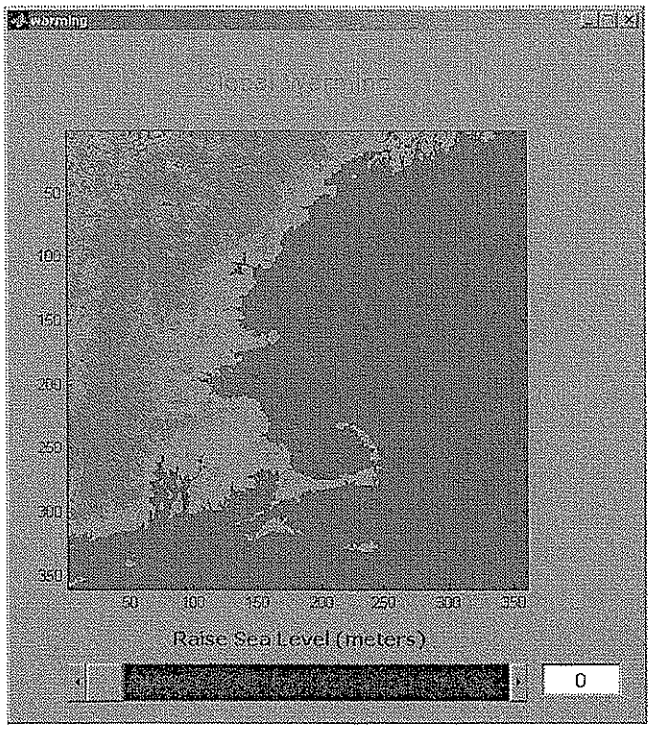
Rising Sea GUI



Reference: Section 10
Topics: GUIDE, callback programming, global variables, GUI data

Exercise

1. Use GUIDE to create a GUI with the following layout:



2. Use the GUI `OpeningFcn` to load the image data in `cape.mat` and display it in the axes. Declare the variable containing the data to be `global` so that it can be shared with the callback subfunctions.

3. Write callbacks for the slider and the editable text box so that they decrease the displayed image data (raise sea level) by the given value. A change in the slider should automatically be reflected by a change in the number in the text box and vice versa.

4. Declare the variable containing the image data to be `global` from the command prompt, and change it so that the GUI malfunctions.

5. Eliminate global variables in the GUI M-file and instead pass the image data among callbacks using `guidata`.

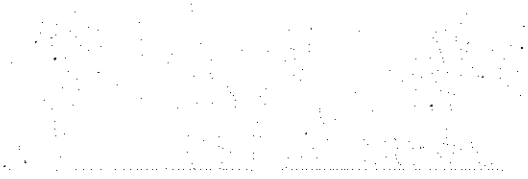
SOLUTION

1.
```
>> guide warming
```

2., 3.
```
>> edit warming
>> warming
```

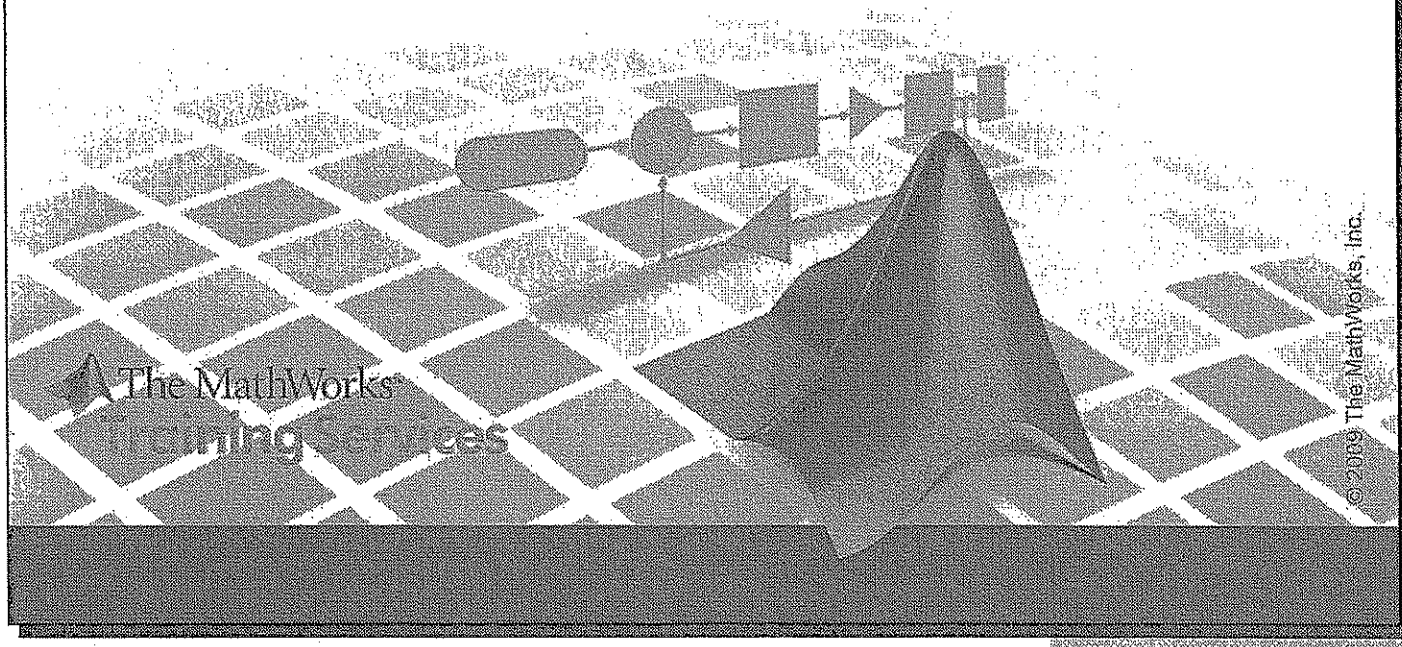
4.
```
>> global I
>> I = I - 500;
```
Move the slider

5.
```
>> edit warming2
>> warming2
```

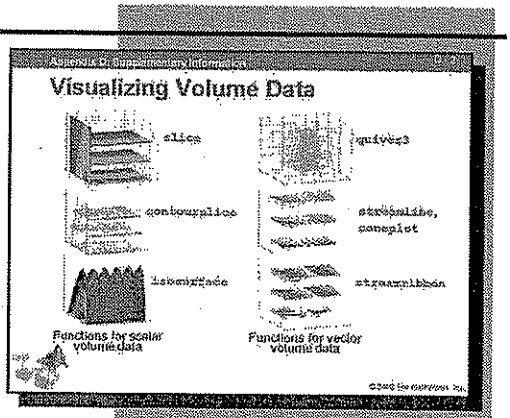


MATLAB® Fundamentals

Supplementary Information



Visualizing Volume Data



Volume visualization is the creation of graphical representations of data sets that are defined on 3-D grids. Volume data sets are characterized by multidimensional arrays of scalar or vector data. Data is typically defined on lattice structures representing values sampled in 3-D space.

There are two basic types of volume data:

- *Scalar volume data* contain single values for each point.
- *Vector volume data* contain two or three values for each point, defining the components of a vector.

The techniques you select to visualize volume data depend on what type of data you have and what you want to learn. In general, scalar data is best viewed with *slice planes*, *contour slices* and *isosurfaces*. Vector data represents both a magnitude and direction at each point, which is best displayed by *stream lines* (*particles*, *ribbons*, and *tubes*), *cone plots*, and *arrow plots*. Most visualizations, however, employ a combination of techniques to best reveal the content of the data.

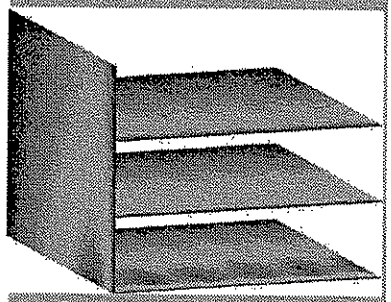
To create an effective volume visualization:

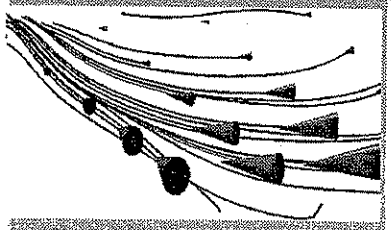
1. Determine the characteristics of your data. Graphing volume data requires knowing the range of the coordinates and the data values.
2. Select an appropriate plotting routine. See "Volume Visualization Techniques" in the MATLAB documentation.
3. Define the view. The information conveyed by a complex 3-D graph can be enhanced through careful composition of the scene. Use camera position, aspect ratio and project type, and zooming.
4. Add lighting and specify coloring. Lighting enhances the visibility of surface shape and provides a 3-D perspective to volume graphs. Color can convey data values, both constant and varying.

Try

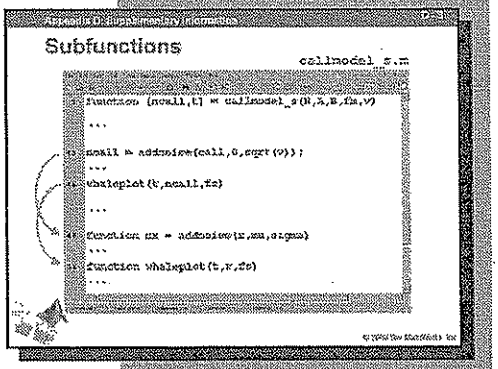
Simple hurricane model

```
>> edit hurricane
>> hurricane
```





Subfunctions



M-files can contain code for more than one function. Additional functions within the file are called *subfunctions*. Subfunctions are only visible to the *primary function* or to other subfunctions in the same file (*not* from the command line).

Each subfunction begins with its own function declaration. Subfunctions can be in any order, but the primary function, called from outside the M-file, must appear first:

```
function [A,B] = primary(x,y)

A = subfunction1(x);
B = subfunction2(y);

function v = subfunction1(u)
v = rand(u,1);

function v = subfunction2(u)
v = randn(u,1);
```

Functions within the same M-file cannot access each other's variables unless they are declared `global` or passed as arguments. The help facility can access only the primary function.

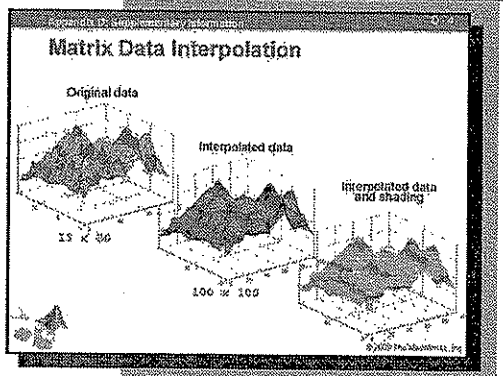
When you call another function from within an M-file, MATLAB checks to see if the called function is a subfunction *before* it looks for an M-file on the search path. As a result, subfunctions take precedence over M-files functions with the same name.

Subfunctions are a useful way to modularize your code, contain the components of a computation in a single file, and reduce path lookup.

Try

```
>> edit callmodel_s
>> N = 3;
>> A = 2;
>> B = 1.5;
>> fm = 0.65;
>> v = 0.05;
>> [x,t] = ...
callmodel_s(...
N,A,B,fm,v);
```

Matrix Data Interpolation



As with vector data, interpolation is the process of finding a model (in the case of 2-D matrix data, a *surface*) that passes exactly through data. You can use the model to approximate values between or beyond the data and create smoother plots.

The MATLAB function `interp2` is the 2-D generalization of the function `interp1` used with vector data.

```
>> ZI = interp2(X,Y,Z,XI,YI,method)
```

returns matrix `ZI` with values corresponding to the elements of `XI` and `YI`, interpolated from the data in `X`, `Y`, and `Z`. The data in `X` and `Y` must be monotonic, with the same "plaid" format produced by `meshgrid`. The `method` can be `'nearest'` (nearest neighbor interpolation), `'linear'`, `'spline'`, or `'pchip'`.

The interpolation performed by `interp2` is useful for data that can be organized naturally into a grid, such as image data. Often, however, vector data `x` and `y` does not increase or decrease monotonically, and `meshgrid` cannot put it into a neat plaid pattern. The MATLAB function `griddata` takes arbitrary vector data `x`, `y`, and `z` and interpolates it to a grid using Delaunay triangulation.

```
>> ZI = griddata(x,y,z,XI,YI,method)
```

returns matrix `ZI` with values corresponding to the elements of `XI` and `YI`, interpolated from the vector data in `x`, `y`, and `z`. The method can be less efficient than `interp2`, because of the work to create the grid.

Interpolation can be used to improve the display of a plot by smoothly approximating values between the data. The display can also be improved through the use of *interpolated shading*. The command

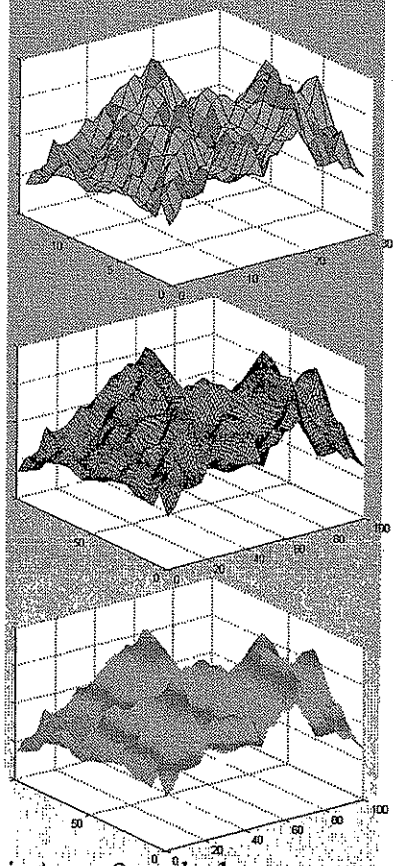
```
>> shading interp
```

varies the color in each line segment and face of a plot by interpolating the colormap index or true color value across the line or face. This display technique should not be confused with interpolation of the data.

Try

Satellite elevation data

```
>> edit interpcape
>> interpcape
```



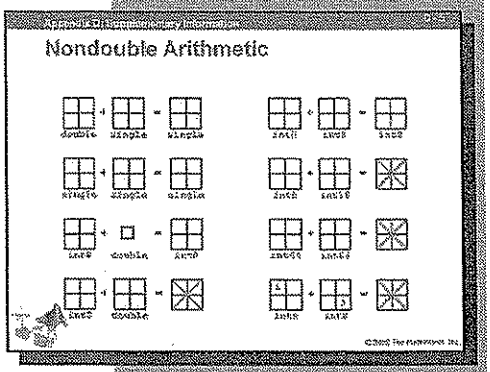
`interp2` methods

```
>> edit interpdata1
>> interpdata1
```

Nonmonotonic data

```
>> edit ...
calibration.dat
>> edit interpdata2
>> interpdata2
```

Nondouble Arithmetic



MATLAB defines all arithmetic operators with floating-point operands of type `double` and `single`. When one of the operands is of type `single`, the result is also `single`.

Most MATLAB arithmetic operators (notable exceptions are `*` and `\`) are defined with real integer operands of type `int8`, `uint8`, `int16`, `uint16`, `int32`, and `uint32`. Operands must be of the *same* integer type. Results are the same type as the operands.

Arithmetic operators are *not* defined on `int64` and `uint64` types.

Arithmetic operators (excepting the unary operators `+A` and `A.'`) are *not* defined on complex arrays of *any* integer data type.

For operations defined on integer data types, scalar or array integer operands can be combined with scalar, but *not* array, operands of type `double`. Results are the same type as the integer operand. Arrays of integer type can *not* be combined with scalars or arrays of a different integer type, or with scalars or arrays of type `single`.

For operations in which one operand is an array of integer type and the other is a scalar `double`, or for the array operations `A./B` and `A.\B` where `A` and `B` are of integer type, MATLAB computes the operation using elementwise double-precision arithmetic . It then converts the result back to the original integer type. For example, when computing

```
>> int8([1 2 3 4 5])*0.8
 ans =
     1     2     2     3     4
```

the product `[1 2 3 4 5]*0.8` is found to double precision and then converted to `int8`. Note that the 2nd and 3rd entries of product (1.6 and 2.4) are both rounded to the nearest integer, which is 2.

Try

Single and double types

```
>> A = rand(2)
>> B = single(A)
>> X = 2* B
>> Y = A + B
```

Integer types

```
>> A = uint8(...
10*rand(2))
>> B = uint8(...
10*rand(2))
>> C = int8(...
10*rand(2))
>> W = A + B
>> X = A*B
>> Y = A*uint8(2)
>> Z = A - C
```

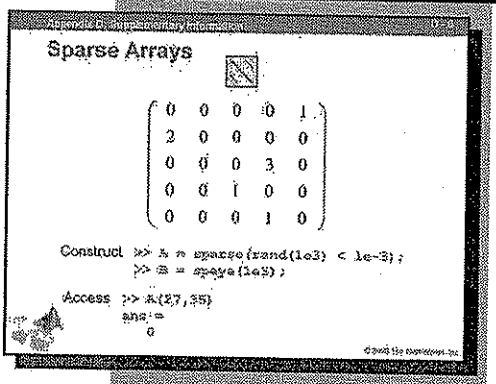
Mixed integer/double types

```
>> A = 10*rand(2)
>> B = uint8(...
10*rand(2))
>> C = uint8(...
10*rand(2))
>> B + 2
>> B + A
>> B./C
```

Casting strategies

```
>> edit sumcast
>> sumcast
```

Sparse Arrays



You can create 2-D double and logical arrays in one of two storage formats: *full* or *sparse*. For matrices with mostly zero-valued elements, a sparse matrix requires a fraction of the storage space and access time of an equivalent full matrix. Sparse matrices also have methods designed to work efficiently with sparse data by eliminating unnecessary operations on zeros.

MATLAB never creates sparse matrices automatically. Instead, you must determine if a matrix contains a large enough percentage of zeros to benefit from sparse formatting. The *density* of a matrix is the number of nonzero elements divided by the total number of matrix elements. Large matrices with a density of 2/3 or less are often good candidates for the sparse format.

You can convert a full matrix to sparse storage using the MATLAB `sparse` function. Likewise, you can convert a sparse matrix to full storage using the `full` function (provided the matrix is not too large to be stored in this format).

Converting a full matrix to sparse storage is not the typical way that sparse matrices are constructed. If a matrix is small enough that full storage is possible, then conversion to sparse storage may not offer significant advantages. MATLAB provides a number of functions that create sparse matrices directly. To see a list of available functions, type

```
>> help sparfun
```

Constructor Methods

- Use the `sparse` function to convert full matrices.
- Use the functions in `sparfun` to construct sparse matrices directly.

Accessor Methods

- Use the same methods as for double arrays.

Try

```
>> help sparfun
```

Constructor methods

```
>> A = eye(1e3);
>> B = sparse(A);
>> C = speye(1e3);
>> D = full(C);
>> E = sparse(...
rand(1e3) < 1e-3);
>> whos
```

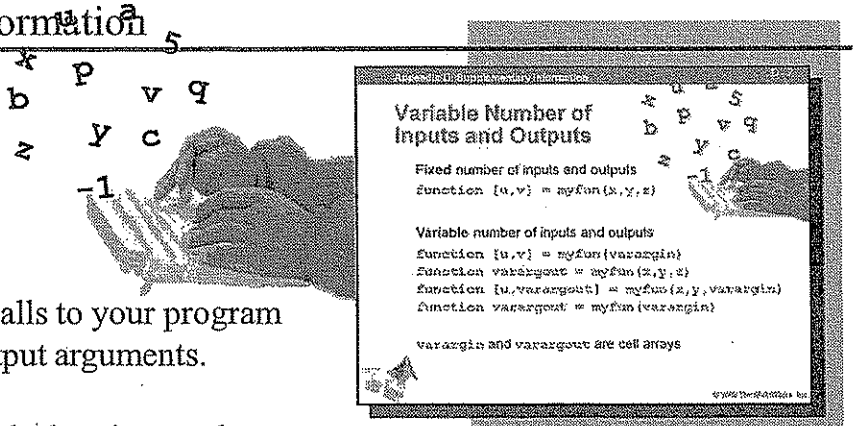
Accessor methods

```
>> E = sparse(...
rand(1e3) < 1e-3);
>> E(1:1e2,1:1e2)
>> sum(E(:))
```

Computational efficiency

```
>> A = eye(1e3);
>> C = speye(1e3);
>> R = rand(1e3);
>> tic; A*R; toc;
>> tic; C*R; toc;
```

Variable Number of Inputs and Outputs



One aspect of usability is anticipating calls to your program with a variable number of input and output arguments.

Function M-files begin with a function declaration, such as

```
function [u,v] = myfun(x,y,z)
```

This function expects three input arguments and returns up to two output arguments. Any number of input arguments other than three results in an error. The function can be called with zero, one, or two output arguments, but asking for any more results in an error.

It is possible to create functions with greater flexibility by using the MATLAB system variables `varargin` and `varargout`. These variables may appear in function declarations in a variety of forms:

```
function [u,v] = myfun(varargin)
function varargout = myfun(x,y,z)
function [u,varargout] = myfun(x,y,varargin)
function varargout = myfun(varargin)
```

These functions allow for any number of additional input or output arguments beyond those specified explicitly.

The variables `varargin` and `varargout` are cell arrays. The elements of `varargin` are assigned when the function is called. For example, calling a function with a declaration of the form

```
function varargout = myfun(varargin)
```

by typing

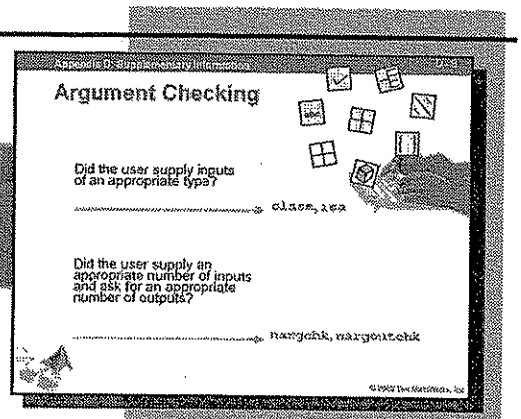
```
>> [p,q,r] = myfun(1,2,3,4,5)
```

implicitly makes the assignment `varargin = {1,2,3,4,5}`. For the same call to execute without error, the M-file code must assign values to `varargout{1}`, `varargout{2}` and `varargout{3}`. The elements of `varargout` may then be returned as `p`, `q`, and `r`.

Try

```
>> ones(3)
>> ones(3,2)

>> X = magic(3);
>> V = eig(X)
>> [V,D] = eig(X)
>> help eig

>> edit surf
>> edit textread
```

Argument Checking



You probably want to limit the number and type of inputs that users are able to provide, so that your code can handle all cases in a predictable manner.

If you use `varargin` or `varargout` in a function declaration, your M-file code has to perform three tasks:

1. Extract the input arguments by indexing into `varargin`.
2. Check the number and type of input and output arguments.
3. Assign output arguments by indexing into `varargout`.

It is during the second step that warning and error messages are issued. To assist in this step, MATLAB creates two additional system variables in the function workspace: `nargin` and `nargout`. These variables are assigned, respectively, the number of input arguments and the number of output arguments in the function call.

MATLAB provides several useful argument checking functions.

```
msg = nargchk(low,high,n)
msg = nargoutchk(low,high,n)
```

return appropriate error messages if `n` is less than `low` or greater than `high`. If `n` is between `low` and `high` (inclusive), `msg` is empty. The messages can be displayed with either the `error` (stops execution) or `warning` (continues execution) commands.

```
str = class(obj)
```

returns a string giving the class (MATLAB data type) of `obj`.

```
K = isa(obj,'class_name')
```

returns logical true (`1`) if `obj` is of class `class_name`, and logical false (`0`) otherwise.

Try

Checking the number of inputs and outputs

```
>> edit varargdemo
>> varargdemo
>> varargdemo(1,2,3)
>> [out1,out2] = ...
varargdemo(1)
>> out1 = ...
varargdemo(1,2)
>> out1 = ...
varargdemo(1)
>> [out1,out2] = ...
varargdemo(1,2)
```

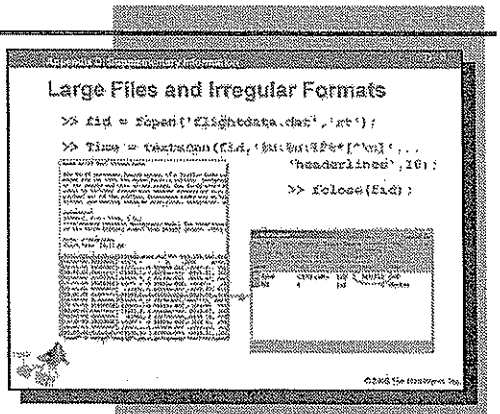
Error and warning messages

```
>> edit errordemo
>> errordemo(1,2,3)
>> [out1,out2] = ...
errordemo(1)
>> out1 = ...
errordemo(1,2)
>> out1 = ...
errordemo(1)
>> [out1,out2] = ...
errordemo(1,2)
```

Checking the type of inputs

```
>> edit typechkdemo
>> typechkdemo(...
pi,'foo',{1,2})
```

Large Files and Irregular Formats



The `textread` function is useful for reading ASCII text data with a uniform format from one row of the data to the next. This uniform format is specified in the formatting string in the second input argument, which is used to determine how each row of the data is assigned to a possible multiplicity of output arguments. If the data in the file breaks from the specified format, however, `textread` stops reading.

MATLAB offers the lower level `textscan` function to complement the `textread` function and address some of its limitations. In particular, `textscan` can adjust its formatting and conversion specifications as it reads through an irregularly formatted file.

The `textscan` and `textread` functions differ in the following ways:

- `textscan` is generally more efficient than `textread`, making it a better choice when reading large files.
- After a file is open, you can ask `textscan` to start reading from any point in a file. `textread` starts reading from the beginning of a file.
- Subsequent calls to `textscan` start reading at the point where the last `textscan` left off. The `textread` function always begins at the start of a file, regardless of prior calls to `textread`.
- `textscan` returns a single cell array regardless of how many fields are specified in the format. `textscan` does not need the number of output arguments to equal the number of fields, as with `textread`.
- When an integer conversion specifier is given, `textscan` reads in integer data and stores it directly as an integer type, whereas `textread` initially stores the data as a double type (taking more memory) and then converts it to an integer type. `textscan` also has more choices for converting data to MATLAB types.
- `textscan` has more low-level options than `textread`.

Try

Large file

```
>> edit flightdata.dat
>> open('HL20.fig')
>> fid = fopen(...
'flightdata.dat',...
'rt');
>> Time = ...
textscan(fid,...
'%u:%u:%f%*[^\n]',...
'headerlines',18);
>> fclose(fid);
```

Irregular CSV lists

```
>> edit IDD.dat
>> edit readIDD
>> readIDD
```

Irregular column formatting

```
>> edit FC.dat
>> edit readFC
>> readFC
```

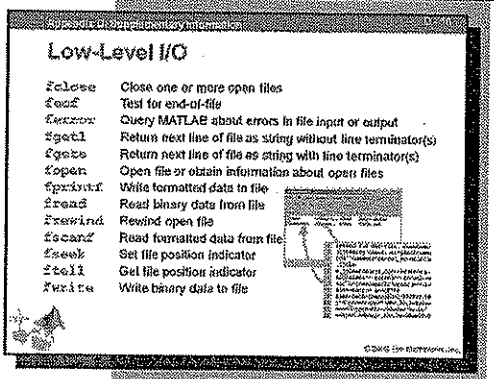
Irregular block formatting

```
>> edit CCX.dat
>> edit readCCX
>> readCCX
```

Direct read of integer types

```
>> fid = ...
fopen('numbers.dat');
>> A = ...
textscan(fid,'%u8')
>> fclose(fid)
```

Low-Level I/O



MATLAB includes a set of low-level file I/O functions that are based on the I/O functions of the ANSI Standard C Library. If you know C, you are probably familiar with these functions. The MATLAB and C commands have some differences, particularly where MATLAB commands are vectorized to work with arrays of data. These differences are highlighted in the documentation.

Low-level I/O functions are especially useful when working with files in unsupported formats, where the Import Wizard, the `importdata` command, and the `*read` and `*write` functions are inappropriate. The low-level I/O functions also give the most detailed programmatic control over the way you import and export your data.

To read or write data at a low level, follow this basic outline:

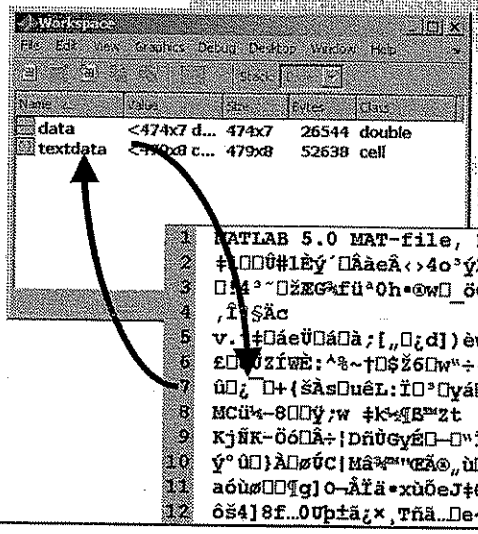
1. Open the file using `fopen`, which returns a file identifier that is used with all the other low-level file I/O routines.
2. Operate on the file.
   a. Read binary data using `fread`.
   b. Write binary data using `fwrite`.
   c. Read text strings line-by-line using `fgets` or `fgetl`.
   d. Read formatted ASCII data using `fscanf`.
   e. Write formatted ASCII data using `fprintf`.
3. Close the file using `fclose`.

Other functions, such as `frewind` and `fseek`, allow you to position yourself exactly where you want to be in a file, and functions like `ftell` and `feof` return information on that position.

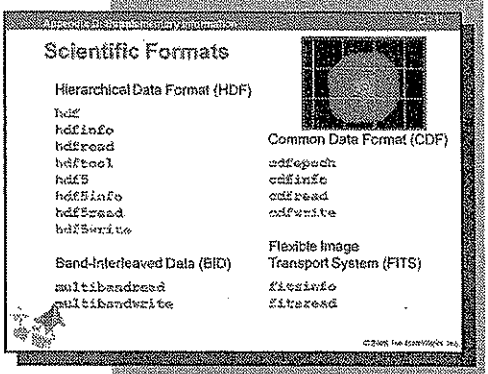
Try

Write/read an ASCII text file
```
>> edit mymatrix
>> mymatrix
```

Write/read a binary file
```
>> edit mymatrix_b
>> mymatrix_b
```



Scientific Formats



MATLAB provides specialized functions for reading and writing data in common scientific formats. These formats include

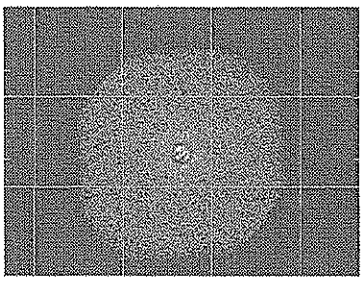
- Hierarchical Data Format (HDF)
- Common Data Format (CDF)
- Flexible Image Transport System (FITS)
- Band-Interleaved Data (BID)

HDF4 and its extension, HDF5, are machine-independent, general-purpose standards for storing scientific data in files, developed by the National Center for Supercomputing Applications (NCSA). HDF-EOS is an extension of HDF4 developed by the National Aeronautics and Space Administration (NASA) for storage of data returned from the Earth Observing System (EOS). MATLAB supports all forms of HDF.

CDF was created by the National Space Science Data Center (NSSDC) to provide a self-describing data storage and manipulation format that matches the structure of scientific data and applications (i.e., statistical and numerical methods, visualization, and management).

FITS is the standard data format used in astronomy, endorsed by both NASA and the International Astronomical Union (IAU). FITS is designed to store scientific data sets consisting of multidimensional arrays (1-D spectra, 2-D images, or 3-D data cubes) and 2-D tables containing rows and columns of data.

BID format (sometimes called RAW format) divides data into multiple bands, commonly found in signal and image processing applications. MATLAB represents data from BID files as multidimensional arrays.



Try

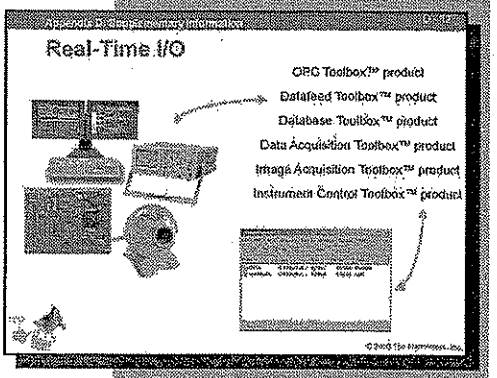
HDF information
```
hdf.ncsa.uiuc.edu
```

CDF information
```
nssdc.gsfc.nasa.gov/cdf/
```

FITS information
```
fits.gsfc.nasa.gov/
```

Available functions
```
hdf
hdfinfo
hdfread
hdftool
hdf5
hdf5info
hdf5read
hdf5write
cdfepoch
cdfinfo
cdfread
cdfwrite
fitsinfo
fitsread
multibandread
multibandwrite
```

Real-Time I/O



While the I/O capabilities of core MATLAB focus on reading and writing data from external files, it is also possible to connect MATLAB to real-time data sources using functions and GUIs in a number of specialized, add-on MATLAB toolboxes.

The OPC Toolbox™ product allows MATLAB to read, write, and log OPC data in devices that support the OPC Foundation Data Access standard, such as distributed control systems, supervisory control and data acquisition systems, and programmable logic controllers.

The Database Toolbox™ product allows MATLAB to read and write data in any ODBC/JDBC-compliant database. A Visual Query Builder tool lets you query stored data without needing to know or learn SQL.

The Datafeed Toolbox™ product allows MATLAB to make multiple data source connections, monitor the status and history of each connection, and request real-time, time series, and historical data from a connection. The toolbox supports connections to Bloomberg®, FactSet®, IDC, HyperFeed®, and Yahoo!® Finance.

The Data Acquisition Toolbox™ product allows MATLAB to control and communicate with a variety of PC-compatible data acquisition hardware. The toolbox lets you configure your external hardware devices and send and receive data from within MATLAB.

The Image Acquisition Toolbox™ product allows MATLAB to acquire video and images from PC-compatible frame-grabber cards and video devices. You can connect to and configure your hardware, preview video, and stream images directly into MATLAB for analysis.

The Instrument Control Toolbox™ product allows MATLAB to communicate with instruments such as oscilloscopes, function generators, and analytic instruments. You can communicate with instruments via instrument drivers, such as IVI and VXI Plug & Play, and commonly used communication protocols, such as GPIB, VISA, TCP/IP, and UDP. Data generated in MATLAB can be sent to an instrument, and data can be read into MATLAB for analysis and visualization.

Try

Real-time audio capture

Turn on your microphone, and turn off your speakers.

```
>> Fs = 11025;
>> y = wavrecord(...
5*Fs,Fs,'int16');
```

Speak for 5 seconds

Turn off your microphone.

Turn on your speakers.

```
>> wavplay(y,Fs);
```