

# **MATLAB Tutorial**

## **1.An Introduction to MATLAB**

- Starting and stopping
- Orientation: Windows and such (Not Finished)
  - Command window
  - Figure window(s)
  - Editor window(s)
  - Help window(s)
- Using MATLAB as a calculator
  - Assigning values to variables
  - Using MATLAB's built-in functions
- Using on-line help
- Suppressing output with the semicolon
- Setting the MATLAB path

## **2.MATLAB Variables**

- scalars
  - Creating
  - Assigning values
  - Scalar operations
- vectors
  - Creating
  - Assigning values
  - Addressing individual elements
  - Colon notation
  - Vector operations
- matrices (Not Finished)
  - creating
  - assigning values
  - addressing individual elements
  - colon notation
  - matrix operations
- Strings (Not Finished)
  - creating
  - assigning values
  - addressing individual elements
  - colon notation
  - string operations

## **3.Plotting**

- Line plots
  - The basic plot command
  - A simple line plot
  - Logarithmic axis scaling
- Symbol plots

- Choosing symbol and line types
- A simple symbol plot
- Multiple curves on the same plot
  - Plotting the data
  - Labelling it with the legend command
- Loading Data into MATLAB for Plotting plots
  - The load command
  - More flexible data import with the readColData command
- Finishing Touches (Not Finished)
  - Controlling the axis limits
  - Multiple plots in a figure window
  - Saving plots to a file
- Contour Plots (Not Finished)
- Surface Plots (Not Finished)
- Vector Plots (Not Finished)

#### **4.Programming MATLAB**

- Scripts
  - Scripts versus Functions
  - Creating a script m-file
  - Using (and abusing) scripts
  - Side effects on the MATLAB workspace
- MATLAB Functions -- The Basics
  - Functions versus Scripts
  - Anatomy of a MATLAB function
  - Calling MATLAB Functions
  - Local Variables
  - Flow Control
- MATLAB Functions -- Intermediate Features (Not Finished)
  - Variable numbers of input and output parameters
  - Global variables
- Debugging MATLAB Functions (Not Finished)
  - Built-in debugging commands
  - Debugging Strategies
- Improving the Speed of MATLAB Calculations
  - Using vector operations instead of loops
  - Pre-allocating memory for vectors and matrices

## **An Introduction to MATLAB**

This section is an introduction to MATLAB by way of example. First some screen shots of MATLAB are shown to help you get oriented to using MATLAB on your computer. Then some simple MATLAB commands are presented that you are encouraged to try. The syntax and general use of these commands are described in detail later.

Starting and stopping

Windows and such

Command window

Figure window(s)

Editor window(s)

Using MATLAB as a calculator

Assigning values to variables

Using MATLAB's built-in functions

On-line Help(!)

Suppressing output with the semicolon

### Starting and stopping

If you are using a DOS or Unix computer, start MATLAB by typing ```matlab"` at the command prompt. If you are using a Macintosh, or a computer running Windows, double click on the MATLAB icon. Refer to the MATLAB User's guide for additional information to help you get started on your computer.

To stop MATLAB type ```quit"` at the MATLAB prompt, i.e.

```
>> quit
```

### Windows and such

MATLAB evolved as an interface to a powerful library of linear algebra routines programmed in FORTRAN. Though it is a very modern computer package, MATLAB's roots are still evident in the way that you interact with it. The most basic way to get MATLAB to do work is to type a command in the Command Window.

### Using MATLAB as a calculator

MATLAB can be used as an expression evaluator. To do this you simply type a mathematical expression into the command window. The command window

prompt is `>>`. To enter an expression, type it after the prompt (correct any mistakes by backspacing) and press return. MATLAB prints the result back to the command window. For example, to "evaluate" pi type

```
>> pi
```

MATLAB responds with

```
ans =  
3.1416
```

pi is a built-in MATLAB variable. The preceding command is actually a request for MATLAB to print the value of this variable.

Try the following examples

```
>> sin(pi/4)  
>> 2^(log2(4))  
>> sqrt(9)
```

### Assigning Values to Variables

MATLAB allows you create variables on the fly. To create a variable just use it on the left hand side of an equal sign. The following examples show how to assign values to three variables, x, y and z. It also shows the MATLAB response to the assignment statements.

```
>> x = 5  
x = 5
```

```
>> y = pi/4  
y = 0.7854
```

```
>> z = y + x^0.25  
z = 2.2807
```

Note that z exists only as a numerical value, not as the formula with which it was created.

Use the who command to list the currently active variables. For the preceding session this results in

```
>> who  
Your variables are:  
ans    x      y      z  
leaving 7433616 bytes of memory free.
```

To print the value of a variable that is already defined just type its name at the prompt.

```
>> z
z = 2.2807
```

The variables section describes how to work with MATLAB variables

## Built-in Functions and Variables

MATLAB has many very powerful built-in functions. As the preceding example demonstrated these functions include the familiar set found on many hand-held calculators. Here is just a tiny sample of other MATLAB functions

<b>Function Name</b>	<b>Meaning</b>
bessel	Bessel function of the first kind
erf	error function
gamma	gamma function
chol	Cholesky factorization of a matrix
det	determinant of a matrix
lu	LU decomposition of a matrix
svd	singular value decomposition of a matrix
plot	plot $y = f(x)$ data
surf	3-D shaded surface plot
contour	contour plot of $z = f(x,y)$ data
movie	play recorded movie frames

**A sample of built-in MATLAB functions**

You can also create your own functions. To do this you type MATLAB commands into a plain text file. The file must have the extension ".m", and because of this MATLAB functions are often referred to as "m-files". Creating and using functions is described in the Programming Matlab section.

MATLAB also has some special variables and functions. You have already encountered the ans variable which is automatically created whenever a mathematical expression is not assigned to another variable.

The built-in eps variable is used as a tolerance for internal calculations. Initially eps is equal to machine epsilon. You can reset the value of eps, but this is not recommended unless you are very sure you know what you are doing.

The realmax, realmin, Inf and NaN built-in variables are used to handle floating point exceptions. All numerical variables in MATLAB are stored as 32 bit floating point numbers. This corresponds to "double precision" on most computers (supercomputers and some high-end workstations being the exception). The Inf and NaN values appear if a floating point exception occurred during the calculations.

In general you should not try to (re)assign the values of any built-in variables listed in the table below.

## On-line Help(!)

MATLAB provides on-line help for all built-in functions and commands. To browse a list of help topics type

```
>> help
```

To get help on a specific topic, for example the cosine function, type

```
>> help cos
```

Using help is one way to learn about the variety of built-in functions. The help command is most useful when you know the name of the function, but are unsure how to use it. The MATLAB manuals provide cross-referenced and indexed descriptions of all aspects of using MATLAB.

## Suppressing Output with Semicolon

In the preceding examples it was useful to have MATLAB print the results of the calculations. This is not always the case. MATLAB will print the result of every assignment operation unless the expression on the right hand side is terminated with a semicolon. Unlike C the semicolon is not required by MATLAB syntax. Rather, it is a convenience that allows multiple statements to be executed without printing the intermediate results. For example, most lines in m-files end with semicolons because only the final results of the calculations are of interest.

To see how the semicolon works enter the following statements exactly as shown, ending each line with a carriage return. Do not enter the prompt character, >>.

```
>> x = pi/3;  
>> y = sin(x)/cos(x);  
>> z = y - tan(x);  
>> z  
>> y
```

The last two lines do not end in semicolons so MATLAB prints the results of evaluating z and y. The the values of z and y were assigned in the y = and z = statements. The last two lines merely printed the values of the expressions z and y.

## Setting the MATLAB path

Setting the internal MATLAB path is such a universal concern that it has its very own page. If you are just beginning to learn MATLAB the process of setting the internal path may be confusing. Don't worry about it right away, but don't forget about it either. There are links throughout these pages that point you back to the path-setting information.

## ***MATLAB Variables***

Overview

Scalars

- Creating

- Assigning values

- Scalar operations

Vectors

- Creating

- Assigning values

- Addressing individual elements

- Colon notation

- Vector operations

Matrices

- creating

- assigning values

- addressing individual elements

- colon notation

- matrix operations

Strings

- creating

- assigning values

- addressing individual elements

- colon notation

- string operations

### Overview of MATLAB Variables

This section describes the fundamental operations involved in the creation and use of MATLAB variables. Detailed discussions of scalars, vectors, matrices, and strings are provided in separate sections.

MATLAB variables are created when they appear on the left of an equal sign. The generic statement

```
>> variable = expression
```

creates the "variable" and assigns to it the value of the expression on the right hand side. You do not need to define or declare a variable before it is used. Matrices do not need to be explicitly dimensioned, and MATLAB allows you to increase the size of a matrix as you work. "Holy FORTRAN, Batman!"

Advanced MATLAB users will point out that the speed of MATLAB functions can be increased by pre-allocating memory for matrices and vectors. This and other performance considerations are discussed elsewhere.

Variable names must begin with an alphanumeric letter. Following that, any number of letters, digits and underscores can be added, but only the first 19 characters are retained. MATLAB variable names are case sensitive so x and X are different variables.

All numerical variables in MATLAB are matrices, a mathematical data type corresponding to a two-dimensional array of numbers. Before performing any calculations with a numeric variable, MATLAB prods and pokes into its contents. MATLAB is very careful to follow the rigorous rules of linear algebra, and it only allows legal operations between matrices, vectors and scalars.

But what about the statement that "all numerical variables in MATLAB are matrices"?

MATLAB keeps track of the dimension of each variable. Thus, the statement

```
>> x = 2
```

creates a scalar variable x. In MATLAB, a scalar is a variable with one row and one column.

A vector is a matrix with only one row or only one column. The distinction between row and column vectors is crucial. When working with MATLAB you will need to understand how to properly perform linear algebra using scalars, vectors and matrices. MATLAB enforces rules on the use of each of these variables

MATLAB variables may also be strings, which are matrices of individual characters. There is no typographical difference in appearance between numerical variables and string variables. For example, the following statements assign a numerical value to x and a string value to y.

```
>> x = 5.2  
>> y = 'Bob'
```

The type of variable (numerical or string) is determined when the variable is created.

### Complex Variables

Numerical matrix elements may be either real or complex. The type of element is initially determined at the time the variable is created. However, matrix elements may change from real to complex if, during an assignment, the right hand side expression evaluates to a complex number. In this sense, matrix elements with real values are simply complex numbers with zero imaginary parts.



## Scalar Variables

- Overview
- Creating
- Scalar operations

### Scalars

In MATLAB a scalar is a variable with one row and one column. Scalars are the simple variables that we use and manipulate in simple algebraic equations.

### Creating scalars

To create a scalar you simply introduce it on the left hand side of an equal sign.

```
>> x = 1;  
>> y = 2;  
>> z = x + y;
```

### Scalar operations

MATLAB supports the standard scalar operations using an obvious notation. The following statements demonstrate scalar addition, subtraction, multiplication and division.

```
>> u = 5;  
>> v = 3;  
>> w = u + v  
>> x = u - v  
>> y = u * v  
>> z = u/v
```

## Vectors

In MATLAB a vector is a matrix with either one row or one column. The distinction between row vectors and column vectors is essential. Many programming errors are caused by using a row vector where a column vector is required, and vice versa.

MATLAB vectors are used in many situations, e.g., creating x-y plots, that do not fall under the rubric of linear algebra. In these contexts a vector is just a convenient data structure. MATLAB still enforces the rules of linear algebra so paying attention to the details of vector creation and manipulation is always important.

- Creating Vectors
- Addressing individual elements
- Increasing the size of a vector
- Colon notation

## Vector operations

### Creating vectors

To create a vector you simply introduce it on the left hand side of an equal sign. Of course this means that the expression on the right side of the equal sign must evaluate to a vector. There are numerous ways to actually create a vector, each one having advantages in particular situations.

- 1.using the the built-in functions ones, zeros, linspace, and logspace
- 2.assingning a mathematical expressions involving vectors
- 3.appending elements to a scalar
- 4.using colon notation

Each method is useful and you will probably develop your own preferences. The first two methods will be described in this section. The last two methods are described in later sections.

### Creating vectors with ones, zeros, linspace, and logspace

The ones, zeros linspace, and logspace functions allow for explicit creations of vectors of a specific size and with a prescribed spacing between the elements. These functions will be demonstrated by example without providing an exhaustive reference. Refer to the MATLAB manual (or help pages) for details.

To create a vector with one of these functions you must (atleast initially) decide how long do you want the vector to be. You must also decide whether the vector is a row or column vector.

The ones and zeros functions have two arguments. The first is the number of rows in the matrix you wish to create. The second is the number of columns. To create a row or a column vector set the appropriate argument of ones and zeros to one.

To create a row vector of length 5, filled with ones use

```
>> x = ones(1,5)
```

To create a column vector of length 5, filled with zeros use

```
>> y = zeros(5,1)
```

The linspace and logspace functions create vectors with linearly spaced or logarithmically spaced elements, respectively. Here are examples including the MATLAB output.

```
>> x = linspace(1,5,5)
```

```
x =
```

```

          1      2      3      4      5
>> y = logspace(1, 4, 4)
y =
          10      100      1000      10000

```

The third argument of both `linspace` and `logspace` is optional. The third argument is the number of elements to use between the the range specified with the first and second arguments.

Assigning vector expressions to a vector

Once a vector has been created, it may be assigned to another vector. If the vector on the left of the equal sign does not exist it is created to fit the expression on the right hand side of the equal sign.

```

>> x = zeros(1, 5);
>> y = x;

```

Addressing vector elements

Individual elements of a vector can be addressed with a Fortran like subscript. For example

```

>> x = linspace(11, 15, 5);
>> x(2)

ans =

    12

```

MATLAB automatically interprets the index as the appropriate row or column

Increasing the size of a vector (or scalar)

MATLAB allocates memory for all variables on the fly. This allows you to increase the size of a vector simply by assigning a value to an element that has not been previously used.

```

>> x = linspace(21, 25, 5)
x =
    21    22    23    24    25

```

```
>> x(7) = -9
x =
    21    22    23    24    25     0    -9
```

This augmentation should be avoided for vectors involved in computations where speed is critical. Refer to Pre-allocating memory for vectors and matrices for a discussion of the performance issues.

Automatic augmentation of vectors does not allow you to refer to elements that have not yet been allocated.

```
>> y = linspace(21,25,5)
y =
    21    22    23    24    25

>> y(7)
??? Index exceeds matrix dimensions.
```

### Colon notation

MATLAB colon notation is a compact way to refer to ranges of matrix elements. It is often used in copy operations and in the creation of vectors and matrices.

Colon notation can be used to create a vector as follows

```
>> x = xbegin:dx:xend
```

or

```
>> x2 = xbegin:xend
```

where `xbegin` and `xend` are the range of values covered by elements of the `x` vector, and `dx` is the (optional) increment. If `dx` is omitted a value of 1 (unit increment) is used. The numbers `xbegin`, `dx`, and `xend` need not be integers.

The preceding statements create row vectors. For example

```
>> x = 1:5
x =
    1     2     3     4     5
```

To create a column vector, append the transpose operator to the end of the vector-creating expression

```
>> y = (1:5)'
```

```
y =
```

```
1  
2  
3  
4  
5
```

Note that the colon expression needs to be enclosed in parentheses. Otherwise the transpose operator is applied to the value, 5, before the vector is created. (see Vector Transpose, below)

Using colon notation to create a vector requires you to specify the increment, whereas using the linspace command requires you to specify the total number of elements. The following commands show how to create the same vector with both approaches.

```
>> xbegin=1; xend=10; nx=5; dx=(xend-  
xbegin)/(nx-1);  
>> x1 = linspace(xbegin,xend,nx);  
>> x2 = xbegin:dx:xend;
```

The vectors are the same only if the increment dx corresponds to an integer number of elements. Prove this by repeating the preceding statements with nx=6;

When a colon expression appears in place of a vector (or matrix) index, the expression is a kind of implied do loop. The expression

```
istart:istop
```

refers to the range of numbers between istart and istop, inclusive. For example, the following statements create a row vector, x, and then copies the third through seventh elements of x into y.

```
>> x = linspace(31,40,10);  
>> y = x(3:7)
```

```
y =
```

```
33    34    35    36    37
```

```
>> y(3)
```

```
ans =
```

```
35
```

The expression,  $y = x(3:7)$ , copies the third through seventh elements of  $x$  into the first through fourth elements of  $y$ . If  $y$  did not already exist it is created by the assignment.

## Vector operations

Mathematical operations involving vectors follow the rules of linear algebra. Addition and subtraction of vectors is performed with the  $+$  and  $-$  operators.

- Addition and subtraction
- Operations with scalars
- Vector transpose
- Vector Multiplication
- Element-wise vector operations
- Passing vectors to built-in functions

## Matrices

Matrices are the fundamental data type in MATLAB.

- Creating
- Assigning values
- Addressing individual elements
- Colon notation
- Matrix operations

### Creating matrices

To create a matrix you simply introduce it on the left hand side of an equal sign. Of course this means that the expression on the right side of the equal sign must evaluate to a matrix.

### Addressing matrix elements

### Colon notation

### Matrix operations

- Addition and subtraction
- Operations with scalars
- Operations with vectors
- Matrix Multiplication
- Element-wise matrix operations
- Passing matrices to built-in functions

## Strings

Strings variables are primarily used for labeling output and passing the names of m-files as arguments to other functions. While strings are not usually thought of as fundamental to numerical analysis, strings can add a degree of flexibility and polish to your MATLAB code.

- Creating
- Assigning values
- Addressing individual elements
- Colon notation
- Matrix operations

## Creating strings

Strings are matrices with character elements. The normal rules of assignment and variable creation apply. The simplest way to create a string is to use it on the left side of an equal sign where the right side of the equal sign is an expression that evaluates to a string.

String constants (literals) are enclosed in single quotes. The following example shows how to create string variables.

```
>> first = 'John';  
>> last  = 'Coltrane';  
>> name  = [first, ' ', last]
```

The first two lines define two string variables. The last line creates a new string variable from three other strings. This assignment is the most straightforward way to concatenate two strings.

## Addressing string elements

### Colon notation

## String operations

- Comparison
- Concatenation
- Reading strings from files
- Element-wise string operations
- Passing strings to built-in functions

## ***Plotting with MATLAB***

MATLAB is very useful for making scientific and engineering plots. You can create plots of known, analytical functions, you can plot data from other sources such as experimental measurements, you can analyze data, perhaps by fitting it to a curve, and then plot a comparison. MATLAB also has powerful built-in routines for drawing contour and three-dimensional surface plots  $z = f(x,y)$  data.

MATLAB plotting commands are best illustrated by example. I have tried to make the examples compact, while avoiding the trivial. The examples build on each other so it is advisable to at least skim through the presentation in order. Jumping from link to link will probably be more effective after you get a feel for the basic plotting commands and the organization of these web pages.

To keep the network response relatively quick, I organized the plotting information into several separate web pages. This page (the one you're reading now) serves as an introduction and as an index. Whenever you click the "Section Outline" button at the bottom of one subsidiary plotting pages you will be brought back to this page.

I've created one example to demonstrate each of the major plotting features. Both the plotting commands and the MATLAB output are shown. The examples are available as script files that are linked to these pages. Using these links you can easily download the example scripts to your computer. If you are not familiar with Web browsers you may want to refer to a brief set of instructions for help in downloading the scripts.

Loading your own data into MATLAB can sometimes be frustrating. I've presented a detailed discussion how to do this in two ways. One way is to use the MATLAB load command, which requires that your data be organized into columns with no labeling text. Another way involves a custom MATLAB function (available for downloading) that can strip heading text and read column labels from a plain text file.

MATLAB plot commands take vectors and matrices as input so you should be familiar with manipulating these variable types. Refer to the vectors and matrices sections for review. You should also know how to use colon notation to refer to parts of matrices and vectors. And finally, since script files are used to automate the process of creating plots you may want to skim ahead to the section on scripts.

Few things are more frustrating (to me at least) than having a plot of valuable data that has no labels. When I grade homework or reports I ignore all plots that are not properly labelled. I figure that if the author of the report did not bother to label the plot, the data must not be very important. It's not my job to try to read the mind of the person writing the report.

The following examples show how easy it is to not only create plots, but label them as well.

Here's the Outline of the Plotting Section



## Line plots

- The basic plot command

- A simple line plot

- Logarithmic axis scaling

## Symbol plots

- Choosing symbol and line types

- A simple symbol plot

## Multiple curves on the same plot

## Plotting data stored in a separate file

- The load command

- More flexible data import with the readColData command

## Finishing Touches

- Controlling the axis limits

- Multiple plots in a figure window

- Saving plots to a file

## Contour plots

## Surface plots

## Vector plots

## Creating Line Plots with MATLAB

This page describes how to plot  $y = f(x)$  by connecting the data points with a line.

- The basic plot command

- A simple line plot

- Logarithmic axis scaling

## The basic plot command

Two-dimensional line and symbol plots are created with the plot command. In its simplest form plot takes two arguments

```
>> plot(xdata,ydata)
```

where `xdata` and `ydata` are vectors containing the data. Note that `xdata` and `ydata` must be the same length and both must be the same type, i.e., both must be either row or column vectors. Additional arguments to the `plot` command provide other options including the ability to plot multiple data sets, and a choice of colors, symbols and line types for the data.

### A simple line plot

Here are the MATLAB commands to create a simple plot of  $y = \sin(3\pi x)$  from 0 to  $2\pi$ . For your convenience these commands are also stored in the `sin3xPlot.m` script file. (Refer to "help with downloading scripts" for instructions on downloading this script to your computer.)

```
>> x = 0:pi/30:2*pi;    % x vector, 0 <= x <= 2*pi, increments of pi/30
>> y = sin(3*x);       % vector of y values
>> plot(x,y)           % create the plot
>> xlabel('x (radians)'); % label the x-axis
>> ylabel('sine function'); % label the y-axis
>> title('sin(3*x)');   % put a title on the plot
```

The effect of the labeling commands, `xlabel`, `ylabel`, and `title` are indicated by the text and red arrows in the figure. Don't skimp on the labels!

### Logarithmic axis scaling

Log-log and semi-log plots are created with commands that act just like the `plot` command. These are summarized in the table below

Commands for creating log-log and semilog plots	
Command Name	Plot type
<code>loglog</code>	$\log(y)$ versus $\log(x)$
<code>semilogx</code>	$y$ versus $\log(x)$
<code>semilogy</code>	$\log(y)$ versus $x$

This page describes how to plot  $y = f(x)$  by putting a symbol at each data point.

### Changing symbol or line types

#### A simple symbol plot

### Changing symbol or line types

The symbol or line type for the data can be changed by passing an optional third argument to the plot command. For example

```
>> plot(x,y,'o');
```

plots data in the x and y vectors using circles drawn in the default color (yellow), and

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

plots data in the x and y vectors by connecting each pair of points with a red dashed line.

The third argument of the plot command is a one, two or three character string of the form 'cs', where 'c' is a single character indicating the color and 's' is a one or two character string indicating the type of symbol or line. The color selection is optional. Allowable color and symbols types are summarized in the following tables. Refer to ``help plot" for further information.

#### A simple symbol plot

This example shows you how to plot data with symbols. This type of plot is appropriate, for example, when connecting data points with straight lines would give the misleading impression that the function being plotted is continuous between the sampled data points.

Here are the MATLAB commands to create a symbol plot with the data generated by adding noise to a known function. These commands are also in the script file `noisyData.m` for you to download.

```
>> x = 0:0.01:2;           % generate the x-vector
>> noise = 0.02*randn(size(x)); % and noise
>> y = 5*x.*exp(-3*x) + noise; % Add noise to known function
>> plot(x,y,'o');         % and plot with symbols
>>
>> xlabel('x (arbitrary units)'); % add axis labels and plot title
>> ylabel('y (arbitrary units)');
>> title('Plot of y = 5*x*exp(-3*x) + noise');
```

Note that the ``.`.*`` operator is necessary when multiplying the vector `x` by the vector `exp(-3*x)`. The preceding statements create the following plot.

### Putting Multiple Curves on the Same Plot

This page describes how to plot  $y_1 = f_1(x)$ ,  $y_2 = f_2(x)$ , ...  $y_n = f_n(x)$  on the same plot, and how to label these curves with the legend command.

- Passing multiple data pairs to the plot command
- The legend command
- An example of putting multiple curves on a plot

Passing multiple data pairs to the plot command

The plot command will put multiple curves on the same plot with the following syntax

```
plot(x1,y1,s1,x2,y2,s2,x3,y3,s3,...)
```

where the first data set  $(x_1, y_1)$  is plotted with symbol definition `s1`, the second data set  $(x_2, y_2)$  is plotted with symbol definition `s2`, etc. The separate curves are labeled with the legend command as demonstrated in the following example.

Note that `x1`, `x2`, ... etc need not be the same vector.

Each pair of vectors,  $(x_1, y_1)$ ,  $(x_2, y_2)$ , etc. must have the same length. In other words the length of `x1` and `y1` must be the same. The length of `x2` and `y2` must be the same, but can be different from the length of `x1` and `y1`.

The legend command

The legend is used to label the different data curves when more than one curve is drawn on the same plot. The syntax is

```
legend('label1','label2',...)
```

Where `label1`, etc are strings used to create the legend. This is best demonstrated by an example, which follows.

An example of putting multiple curves on a plot

Here are the MATLAB commands to create a symbol plot with the data generated by adding noise to a known function. The original function is drawn with a solid line and the function plus noise is plotted with open circles. These commands are also in the script file `noisyDataLine.m` for you to download.

```

>> x = 0:0.01:2;           % generate the x-vector
>> y = 5*x.*exp(-3*x);    % and the "true" function, y
>> yn = y + 0.02*randn(size(x)); % Create a noisy version of y
>> plot(x,y,'-',x,yn,'ro'); % Plot the true and the noisy

>> xlabel('x (arbitrary units)'); % add axis labels and plot title
>> ylabel('y (arbitrary units)');
>> title('Plot of y = 5*x*exp(-3*x) + noise');
>> legend('true y','noisy y');

```

Note that the ``.`.*` operator is necessary when multiplying the vector `x` by the vector `exp(-3*x)`. The preceding statements create the following plot.

### Loading Data into MATLAB for Plotting

In addition to plotting values created with its own commands, MATLAB is very useful for plotting data from other sources, e.g., experimental measurements. Typically this data is available as a plain text file organized into columns. MATLAB can easily handle tab or space-delimited text, but it cannot directly import files stored in the native (binary) format of other applications such as spreadsheets.

The simplest way to import your data into MATLAB is with the load command. Unfortunately, the load command requires that your data file contain no text headings or column labels. To get around this restriction you must use more advanced file I/O commands. Below I demonstrate both approaches with examples. I've included an m-file to handle the more complex case of a file with an arbitrary number of lines of text header, in addition to text labels for each column of data. Though hardly a cure-all, this function is much more flexible than the load command because it allows you to provide documentation inside your data file.

Here's an outline of this section

- The MATLAB load Command
- A simple plot of data from a file
- Plotting data from files with column headings

#### The MATLAB load Command

There is more than one way to read data into MATLAB from a file. The simplest, though least flexible, procedure is to use the load command to read the entire contents of the file in a single step. The load command requires that the data in the file be organized into a rectangular array. No column titles are permitted. One useful form of the load command is

```
load name.ext
```

where ``name.ext" is the name of the file containing the data. The result of this operation is that the data in ``name.ext" is stored in the MATLAB matrix variable called name. The ``ext" string is any three character extension, typically ``dat". Any extension except ``mat" indicates to MATLAB that the data is stored as plain ASCII text. A ``mat" extension is reserved for MATLAB matrix files (see ``help load" for more information).

Suppose you had a simple ASCII file named my\_xy.dat that contained two columns of numbers. The following MATLAB statements will load this data into the matrix ``my\_xy", and then copy it into two vectors, x and y.

```
>> load my_xy.dat;    % read data into the my_xy matrix
>> x = my_xy(:,1);   % copy first column of my_xy into x
>> y = my_xy(:,2);   % and second column into y
```

You don't need to copy the data into x and y, of course. Whenever the ``x" data is needed you could refer to it as my\_xy(:,1). Copying the data into ``x" and ``y" makes the code easier to read, and is more aesthetically appealing. The duplication of the data will not tax MATLAB's memory for most modest data sets.

The load command is demonstrated in the following example.

If the data you wish to load into MATLAB has heading information, e.g., text labels for the columns, you have the following options to deal with the heading text.

Delete the heading information with a text editor and use the load command :-)

Use the fegetl command to read the heading information one line at a time. You can then parse the column labels with the strtok command. This technique requires MATLAB version 4.2c or later.

Use the fscanf command to read the heading information.

Of these options, using fegetl and strtok is probably the most robust and convenient. If you read the heading text into MATLAB, i.e., if you don't use the load command, then you will have to also read the plot data with fscanf. The example, Plotting data from files with column headings shows how this is done.

### A simple plot of data from a file

This example shows you how to load a simple data set and plot it.

The PDXprecip.dat file contains two columns of numbers. The first is the number of the month, and the second is the mean precipitation recorded at the Portland International Airport between 1961 and 1990. (For an abundance of weather data like this check out the Oregon Climate Center)

Here are the MATLAB commands to create a symbol plot with the data from PDXprecip.dat. These commands are also in the script file precipPlot.m for you to download.

```
>> load PDXprecip.dat;    % read data into PDXprecip matrix
>> month = PDXprecip(:,1); % copy first column of PDXprecip into month
>> precip = PDXprecip(:,2); % and second column into precip

>> plot(month,precip,'o'); % plot precip vs. month with circles

>> xlabel('month of the year'); % add axis labels and plot title
>> ylabel('mean precipitation (inches)');
>> title('Mean monthly precipitation at Portland International Airport');
```

Although the data in the month vector is trivial, it is used here anyway for the purpose of exposition. The preceding statements create the following plot.

### Finishing Touches for Plots

- Controlling the axis limits
- Multiple plots in a figure window
- Saving plots to a file

## ***Programming MATLAB***

Soon after you become accustomed to using MATLAB interactively you will probably grow tired of typing the same sequences of commands in order to perform routine tasks. When this happens you will be ready to start programming in MATLAB.

MATLAB "programs" are sequences of MATLAB commands saved in plain text files. MATLAB requires that these files have the extension ".m". As a result MATLAB programs are often referred to as m-files.

There are two types of MATLAB programs. These are scripts and functions.

### Scripts

- Scripts versus Functions
- Creating a script m-file
- Recording a script with the diary command
- Using (and abusing) scripts
- Side effects on the MATLAB workspace

### MATLAB Functions -- Intermediate Features

- Functions versus Scripts
- Creating a function m-file
- Structure of a MATLAB function
  - Input and Output parameters
  - Comment statements
  - Function prologue -- providing help
- Calling MATLAB Functions
- Local Variables
- Flow Control
  - if constructs
  - for constructs
  - while constructs

### MATLAB Functions -- Intermediate Features

- Variable numbers of input and output parameters
- Global variables

### Improving the Speed of MATLAB Calculations

- Using vector operations instead loops
- Pre-allocating memory for vectors and matrices



## MATLAB Scripts

Scripts are collections of MATLAB commands stored in plain text files. When you type the name of the script file at the MATLAB prompt the commands in the script file are executed as if you had typed them in from the keyboard. Script files must end with the extension ``.m'` (for example `myScript.m`), and often these files are referred to as m-files.

Apart from syntax errors, perhaps the most common problem with using script files is that MATLAB cannot locate the script file you wish to execute. This error can be very frustrating to inexperienced MATLAB users because MATLAB gives the error message

```
??? Undefined function or variable ...
```

and it is not immediately clear why the function is undefined when its m-file is clearly visible to the user. The solution to this is to make sure that MATLAB's internal path variable is set to the direction containing the script. Refer to the `Setting the MATLAB Path`" of this reference for help on setting the path.

Here's an outline of this section

- Scripts versus Functions
- Creating a script m-file
- Using (and abusing) scripts
- Side effects on the MATLAB workspace

### Scripts versus Functions

Scripts are m-files containing MATLAB statements. MATLAB `functions`" are another type of m-file. The biggest difference between scripts and functions is that functions have input and output parameters. Script files can only operate on the variables that are hard-coded into their m-file. As you can see, functions much more flexible. They are therefore more suitable for general purpose tasks that will be applied to different data. Scripts are useful for tasks that don't change. They are also a way to document a specific sequence of actions, say a function call with special parameter values, that may be hard to remember.

There are more subtle differences between scripts and functions. A script can be thought of as a keyboard macro: when you type the name of the script, all of the commands contained in it are executed just as if you had typed these commands into the command window. Thus, all variables created in the script are added to the workspace for the current session. Furthermore, if any of the variables in the script file have the same name as the ones in your current workspace, the values of those variables in the workspace are changed by the actions in the script. This can be used to your advantage. It can also cause unwanted side effects.

In contrast, function variables are local to the function. (The exception is that it's possible to declare and use global variables, but that requires an explicit action by

the user.) The local scope of function variables gives you greater security and flexibility. The only way (besides explicitly declared global variables) to get information into and out of a function is through through the variables in the parameter lists.

### Creating a script m-file

Script files are usually created with a plain text editor. You can also use the MATLAB diary command to record commands as you type

### Creating scripts with a (plain) text editor

MATLAB m-files must be plain text files, i.e. files with none of the special formatting characters included by default in files created by a word-processors. Most word-processors provide the option of saving the file as plain text, (look for a "Save As..." option in the file menu). A word-processor is overkill for creating m-files, however, and it is usually more convenient to use a simple text editor, or a "programmer's editor" to create m-files. Most computer systems have several text editors, though if you are running MATLAB on a personal computer only one of these may be installed.

Most of the time when you are writing m-files you want to have the text editor and MATLAB open at the same time. Since modern word-processors require lots of system RAM it may not even be possible for you to use one for m-file development. In this case a plain text editor will be your only option for m-file development.

### Recording a script with the diary command

The MATLAB diary command is used to record an interactive session. To start recording type

```
>> diary filename
```

where "filename" is the name of the file to which the diary command copies all of the commands entered and all of the resulting output. To stop recording type

```
>> diary off
```

In other words a typical session would be

```
>> diary myDiary
```

```
>> (some MATLAB commands)
```

```
(some MATLAB output)
```

```
>> diary off
```

The result is a plain text file called ``myDiary" in the current working directory.

A diary file can be an easy way to create a script (or function) m-file. After you've recorded the diary you delete the MATLAB output, including any error messages and incorrect commands, and save the result as another file.

Make sure MATLAB knows the path to your script

MATLAB cannot execute a script unless it knows where to find its m-file. This requires that the script be in the internal MATLAB path. Refer to ``Setting the MATLAB Path" for more information.

Using and abusing scripts

I suppose it's only a matter of personal preference, but I rarely use script files for complex numerical calculations.

Scripts are useful for setting global behavior of a MATLAB session. This includes any terminal settings for a remote serial line, or setting the parameters of the Figure window for a certain size plot.

Sometimes a script is a useful starting point in developing a MATLAB function. When I'm starting to write a new function, but I'm uncertain about the command syntax or the actual sequence of commands I want to use, I'll use the diary command to record my tests. After I get the correct command sequence I close the diary file and open it with a text editor. This gives me a jump start on the function development because I don't have to re-enter the commands. I just delete the incorrect lines, add the function definition, and insert variables.

I've witnessed inexperienced MATLAB users who have become dependent on scripts when functions would ultimately be easier to use. This typically occurs when someone gets stuck on the function definition syntax, especially on the use of input and output parameters. Rather than figure out how to properly pass parameters to the function they repeatedly edit their script files to simulate the effect of variable arguments. I hope this hypertext reference can help you avoid that fate. The Programming Basics section contains several example function files that you can use as starting points for your own functions (or scripts).

There is nothing wrong with using scripts, of course. Scripts and functions are two tools for working with MATLAB programming. Using the appropriate tool for the job will help you achieve your analysis goals more easily.

Side effects on the MATLAB workspace

Scripts can have unwanted side effects on the MATLAB workspace used in an interactive session.

Excessive reliance on scripts can result in a confusing mix of variables in the MATLAB workspace. This workspace is the collection of variables in the current interactive session. Because script variables are added to the workspace it is quite easy, in a long sequence of calculations, to have a variable changed by the script. Sometimes this side effect is immediately obvious, for example a vector that you had been using is reassigned as a scalar, leading to a syntax error. Other times only a numerical value is changed. This latter situation is obviously dangerous and can lead to subtle errors in your calculation.

## MATLAB Functions -- Basic Features

- Functions versus Scripts

  - Anatomy of a MATLAB function

    - Creating function m-files with a plain text editor

    - Function Definition

    - Input and Output parameters

    - Comment statements

  - Some simple examples

  - Calling MATLAB Functions

  - Local Variables

  - Flow Control

    - if constructs

    - Logical comparisons

    - while constructs

    - for constructs

Functions versus Scripts

See Scripts versus Functions.

Anatomy of a MATLAB function

MATLAB functions are similar to C functions or Fortran subroutines.

MATLAB programs are stored as plain text in files having names that end with the extension ``.m'`. These files are called, not surprisingly, m-files. Each m-file contains exactly one MATLAB function. Thus, a collection of MATLAB functions can lead to a large number of relatively small files.

One nifty difference between MATLAB and traditional high level languages is that MATLAB functions can be used interactively. In addition to providing the obvious support for interactive calculation, it also is a very convenient way to debug functions that are part of a bigger project.

MATLAB functions have two parameter lists, one for input and one for output. This supports one of the cardinal rules of MATLAB programming: don't change the input parameters of a function. Like all cardinal rules, this one is broken at times. My free advice, however, is to stick to the rule. This will require you to

make some slight adjustments in the way you program. In the end this shift will help you write better MATLAB code.

### Creating function m-files with a plain text editor

MATLAB m-files must be plain text files, i.e. files with none of the special formatting characters included by default in files created by word-processors. Most word-processors provide the option of saving the file as plain text, (look for a "Save As..." option in the file menu). A word-processor is overkill for creating m-files, however, and it is usually more convenient to use a simple text editor, or a "programmer's editor". For most types of computers there are several text editors (often as freeware or shareware). Usually one plain text editor is included with the operating system.

When you are writing m-files you will usually want to have the text editor and MATLAB open at the same time. Since modern word-processors require lots of system RAM it may not even be possible or practical (if you are working on a stand-alone personal computer) for you to use a word-processor for m-file development. In this case a simple, text editor will be your only option.

### Function Definition

The first line of a function m-file must be of the following form.

```
function [output_parameter_list] = function_name(input_parameter_list)
```

The first word must always be "function". Following that, the (optional) output parameters are enclosed in square brackets [ ]. If the function has no output\_parameter\_list the square brackets and the equal sign are also omitted. The function\_name is a character string that will be used to call the function. The function\_name must also be the same as the file name (without the ".m") in which the function is stored. In other words the MATLAB function, "foo", must be stored in the file, "foo.m". Following the file name is the (optional) input\_parameter\_list.

There can exactly be one MATLAB function per m-file.

### Input and Output parameters

The input\_parameter\_list and output\_parameter\_list are comma-separated lists of MATLAB variables.

Unlike other languages, the variables in the input\_parameter\_list should never be altered by the statements inside the function. Expert MATLAB programmers have ways and reasons for violating that principle, but it is good practice to consider the input variables to be constants that cannot be changed. The separation of input and output variables helps to reinforce this principle.

The input and output variables can be scalars, vectors, matrices, and strings. In fact, MATLAB does not really distinguish between variables types until some calculation or operation involving the variables is performed. It is perfectly acceptable that the input to a function is a scalar during one call and a vector during another call.

To make the preceding point more concrete, consider the following statement

```
>> y = sin(x)
```

which is a call to the built-in sine function. If  $x$  is a scalar (i.e. a matrix with one row and one column) then  $y$  will be a scalar. If  $x$  is a row vector, then  $y$  will be a row vector. If  $x$  is a matrix then  $y$  is a matrix. (You should verify these statements with some simple MATLAB calculations.)

This situation-dependence of input and output variables is a very powerful and potentially very confusing feature of MATLAB. Refer to the ``addtwo.m" function below for an example.

### Comment statements

MATLAB comment statements begin with the percent character, `%`. All characters from the `%` to the end of the line are treated as a comment. The `%` character does not need to be in column 1.

### Some simple examples

Here is a trivial function, `addtwo.m`

```
function addtwo(x,y)
% addtwo(x,y) Adds two numbers, vectors, whatever, and
%           print the result = x + y
x+y
```

The `addtwo` function has no output parameters so the square brackets and the equal sign are omitted. There is only one MATLAB statement, `x+y`, in the function. Since this line does not end with a semicolon the results of the calculation are printed to the command window.

The first two lines after the function definition are comment statements. Not only do these statements describe the statements in the file, their position in the file supports the on-line help facility in MATLAB. If the first line of a MATLAB function definition is immediately followed by non-blank comment statements, then those comment statements are printed to the command window when you type ``help function\_name". Try it with the `addtwo` function. MATLAB will print up until a blank line or an executable statement, whichever comes first. Refer to the Function prologues -- providing help section for more information.

To test your understanding of input and output variables, pass the following definitions of `x` and `y` to the `addtwo` function. (To save space the `x` and `y` variables are defined on the same line. You can enter these variables on the same line, as shown, or use separate lines.)

```
>> x = 2;    y = 3;
>> x = [2 3];  y = [4; 5];
>> x = eye(3,3); y = -2*eye(3,3);
>> x = 'Sue';  y = 'Bob'
>> x = 'Jane'; y = 'Bob'
```

Here is another simple function, `traparea.m`, with three input parameters and one output parameter. Since there is only one output parameter the square brackets may be omitted.

```
function area = traparea(a,b,h)
% traparea(a,b,h) Computes the area of a trapezoid given
%           the dimensions a, b and h, where a and b
%           are the lengths of the parallel sides and
%           h is the distance between these sides

% Compute the area, but suppress printing of the result
area = 0.5*(a+b)*h;
```

Note that there is a blank line between the comment statements that describe the function and the single comment statement that describes the calculation of area. The comment statement beginning ``Compute the area...'' will not be printed if you type ``help traparea''. Also note that there is no ``return'' statement needed. (There is a MATLAB return statement, but it is not needed here.) The output variable, `area`, is defined in the first line of the file. The value assigned to `area` is returned to the calling function or the command window.

Finally, here is another simple function, `cart2plr.m`, with two input parameters and two output parameters.

```
function [r,theta] = cart2plr(x,y)
% cart2plr Convert Cartesian coordinates to polar coordinates
%
% [r,theta] = cart2plr(x,y) computes r and theta with
%
%   r = sqrt(x^2 + y^2);
%   theta = atan2(y,x);

r = sqrt(x^2 + y^2);
theta = atan2(y,x);
```

The comment statements have empty lines, but these will be printed if you type ``help cart2plr''.

Make sure MATLAB knows the path to your function

MATLAB cannot execute a function unless it knows where to find its m-file. This requires that the function be in the internal MATLAB path. Refer to "Setting the MATLAB Path" for more information.

### Calling MATLAB Functions

use or ignore return arguments (see also nargin/nargout)  
suppress output

### Local Variables

Unless explicitly declared to be global variables, all variables appearing in a MATLAB function are local to that function.

### Flow Control

MATLAB supports the basic flow control constructs found in most high level programming languages. The syntax is a hybrid of C and Fortran and I often create polyglot statements which lead to the joyless task of squashing trivial bugs.

if constructs

MATLAB supports these variants of the "if" construct

```
if ... end
if ... else ... end
if ... elseif ... else ... end
```

Here are some examples based on the familiar quadratic formula. (n.b. This is not an endorsement of the code logic, just an easy to follow example of the "if" constructs.)

A simple warning

```
>> d = b^2 - 4*a*c;
>> if d<0
>> disp('warning: discriminant is negative, roots are imaginary');
>> end
```



or a warning plus extra notification

```
>> d = b^2 - 4*a*c;  
>> if d<0  
>> disp('warning: discriminant is negative, roots are imaginary');  
>> else  
>> disp('OK: roots are real, but may be repeated');  
>> end
```

or, no secrets whatsoever

```
>> d = b^2 - 4*a*c;  
>> if d<0  
>> disp('warning: discriminant is negative, roots are imaginary');  
>> elseif d==0  
>> disp('discriminant is zero, roots are repeated');  
>> else  
>> disp('OK: roots are real and distinct');  
>> end
```

Careful inspection of the preceding statements reveals that

- no semicolon is needed to suppress output at the end of lines containing if, else, elseif or endif
- elseif has no space between "else" and "if"
- the end statement is required
- the "is equal to" operator has two equals signs (see Logical Comparisons below)

Indentation of "if" blocks is not required, but considered good style.

The compact if

If you read tightly coded m-files (e.g., many of the built-in MATLAB functions) you will discover a variant of the if ... end construct that is written on one line. Here's an example

```
if x<0, disp('imaginary'); end
```

Notice the comma between the  $x<0$  and the `disp(...)`. Apparently the comma tells the MATLAB interpreter that the conditional test has ended. To my knowledge this is only place where a statement (OK, part of a statement) ends with a comma. It's just one of those quirks that true believers come to use without hesitation.

Logical comparisons

The preceding table lists the logical comparison operators used in MATLAB. Note that these operators can be used in assignment statements, as illustrated in this example

```
enough = xnew-xold < delta
if ~enough
    stepSize = stepSize/2;
end
```

The first statement assigns the value of the logical comparison to the variable ``enough". If the comparison on the right hand side evaluates to be true, then enough is given the value 1. If the comparison evaluates to be false, then enough is given the value 0. In the subsequent if statement, enough = 1 is true, and enough = 0 is false.

### Binary and unary logical operators

Binary operators take two arguments (operands). Unary operators take one argument.

The && operator (read ``and" operator) takes two logical expressions and returns ``true" if both expressions are true, and ``false" otherwise.

The || operator (read ``or" operator) takes two logical expressions and returns ``true" if either of the expressions are true, and ``false" only if both expressions are false. The possible outcomes of && and || operations are summarized in the following truth table.

The ~ operator (read ``not") takes only one logical expression and returns the opposite (negation) of that expression. Consider the following code

```
a = 2; b = 3;
asmall = a < b; % asmall is ``true"
if ~asmall
    disp('a is greater than b')
end
```

Since a is less than b the if ~asmall block is not executed because ~asmall is false.

### while constructs

The syntax of the while construct is

```
while expression
    do something
end
```

where ``expression" is a logical expression. The ``do something" block of code is repeated until the expression in the while statement becomes false.

Here is an example

```
i = 2;
while i<8
    i = i + 2
end
```

Executing the preceding block of code results in the following output to the command window

```
i = 4
i = 6
i = 8
```

(The output has been adjusted to take up less space. MATLAB likes to sprawl its printout over several lines.) Note that no disp or fprintf statement was needed in the preceding block because the i = i + 2 statement did not end in a semicolon.

for constructs

The for construct is used to create a loop, usually over a fixed range of steps

```
sum = 0;
for i=1:length(x)
while i<10
    sum = sum + abs(x(i));
end
```

## MATLAB Functions -- Intermediate Features

It's highly subjective to say just what constitutes an ``intermediate" versus a ``basic" programming feature.

The preceding section, Programming Basics, covers what you need to get started with writing MATLAB functions. Soon after you get comfortable with the process of using and writing functions, I believe you will want to incorporate the features described below.

- Function prologues -- providing help
- Variable Number of Input and Output Arguments
- Global Variables
- Using eval to provide an error trap

## Function prologues -- providing help

If the first line of a MATLAB function definition is immediately followed by non-blank comment statements, then those comment statements are printed to the command window when you type ``help function\_name". Experiment with this feature by getting help for MATLAB built-in functions.

For all but the most trivial MATLAB functions, I usually write an extensive function prologue that will be returned by the help facility. The prologue contains a brief statement of the what the function does, a synopsis of how the function may be called, and complete descriptions of all input and output parameters.

As an example, here is the prologue to readColData.m function, which is described in the section Plotting data from files with column headings .

```
function [labels,x,y] = readColData(fname,ncols,nhead,nlrows)
% readColData reads data from a file containing data in columns
%           that have text titles, and possibly other header text
%
% Synopsis:
% [labels,x,y] = readColData(fname)
% [labels,x,y] = readColData(fname,ncols)
% [labels,x,y] = readColData(fname,ncols,nhead)
% [labels,x,y] = readColData(fname,ncols,nhead,nlrows)
%
% Input:
% fname = name of the file containing the data (required)
% ncols = number of columns in the data file. Default = 2. A value
%         of ncols is required only if nlrows is also specified.
% nhead = number of lines of header information at the very top of
%         the file. Header text is read and discarded. Default = 0.
%         A value of nhead is required only if nlrows is also specified.
% nlrows = number of rows of labels. Default = 1
%
% Output:
% labels = matrix of labels. Each row of labels is a different
%         label from the columns of data. The number of columns
%         in the labels matrix equals the length of the longest
%         column heading in the data file. More than one row of
%         labels is allowed. In this case the second row of column
%         headings begins in row ncol+1 of labels. The third row
%         column headings begins in row 2*ncol+1 of labels, etc.
%
% NOTE: Individual column headings must not contain blanks
%
% x = column vector of x values
% y = matrix of y values. y has length(x) rows and ncols columns
```

## Variable Number of Input and Output Arguments

Not Done

## Global Variables

Not Done

## Using eval to provide an error trap

In version 4.1 or later of MATLAB the eval command can be used to provide an error trap. This is one way to recover more gracefully from an error condition, especially if the error occurs in a built-in MATLAB function.

The syntax of the eval error trap is

```
eval('desired','fallBack')
```

where desired is the m-file you wish to execute and fallBack is the name of the m-file (or MATLAB command) that is executed if an error occurs in the desired function.

Normally when MATLAB encounters an error it prints a message and the name of the function in which the error occurred. It then stops. With the eval error trap, MATLAB will continue execution. Presumably some meaningful recovery operation is provided by the fallBack function or text. If all you want to do is provide an error message use, the MATLAB error function instead.

Refer to the version 4.1 (or version 4.2) release notes for a nice example of using eval and lasterr to build an error trap.

## Improving the Speed of MATLAB Calculations

Large scale numerical calculations can put heavy demands on your computer. Some software companies argue that they do not have to do a good job of performance tuning because computer hardware is advancing so rapidly. While it may be possible to foist slug-like word-processors onto the average computer user, engineers and scientists are usually less tolerant of poorly designed and implemented software.

In technical markets the gains in computing performance raise, not lower, the expectations of users. It is often the case that improved hardware and software allows simulations to be run with more detail, which often translates to more realistic results.

MATLAB is an interpreted language. At first this would seem to make it inappropriate for large scale scientific computing. The power of MATLAB is realized with its extensive set of libraries which are compiled or are carefully coded in MATLAB to utilize "vectorization". The concept of vectorization is central to understanding how to write efficient MATLAB code.

Vectorized code takes advantage, wherever possible, of operations involving data stored as vectors. This even applies to matrices since the columns (or rows) of a matrix are usually stored in contiguous locations in the computer's RAM. The speed of a numerical algorithm in MATLAB is very sensitive to whether or not vectorized operations are used.

This section presents some basic considerations to writing efficient MATLAB routines. It is possible, of course, to become obsessed with performance and waste man-hours of programming time in order to save a few seconds of execution time. The ideas presented below require little extra effort, yet can yield significant speed improvements. Once you understand how to use these procedures they will become natural parts of your MATLAB programming style.

For a more advanced discussion of vectorization techniques be sure to check the Mathworks Home Page and in particular their technical notes

The following topics are covered in this section

- Using vector operations instead of loops
- Pre-allocating memory for vectors and matrices

Using vector operations instead of loops

Consider the following loop, translated directly from Fortran or C

```
>> dx = pi/30;
>> nx = 1 + 2*pi/dx;
>> for i = 1:nx
>>   x(i) = (i-1)*dx;
>>   y(i) = sin(3*x(i));
>> end
```

The preceding statements are perfectly legal MATLAB statements, but they are an inefficient way to create the x and y vectors. Recall that MATLAB allocates memory for variables on the fly. On the first time through the loop (i=1), MATLAB creates two row vectors x and y, each of length one. On each subsequent pass through the loop MATLAB appends new elements to x and y. Not only does this incur extra overhead in the memory allocation calls, the elements of x and y will not be stored in contiguous locations in RAM. Thus, any subsequent operations with x and y, even though these operations may be vectorized, will take a performance hit because of memory stride overhead.

The preferred way to create the same two  $x$  and  $y$  vectors is with the following statements.

```
>> x = 0:pi/30:2*pi  
>> y = sin(3*x);
```

The first statement creates the row vector,  $x$ , with  $1 + \pi/15$  elements stored in contiguous locations in RAM. The second statement creates a new (matrix) variable,  $y$ , with the same number of rows and columns as  $x$ . Since  $x$  is a row vector, as determined by the preceding step,  $y$  is also a row vector. If  $x$  were, for example, a 5 by 3 matrix, then  $y = \sin(3*x)$  would create a 5 by 3 matrix,  $y$ .

MATLAB is designed to perform vector and matrix operations efficiently. To take maximum advantage of the computer hardware at your disposal, therefore, you should use vectorized operations as much as possible.

Pre-allocating memory for vectors and matrices

Though MATLAB will automatically adjust the size of a matrix (or vector) it is usually a good idea to pre-allocate the matrix. Pre-allocation incurs the cost of memory allocation just once, and it guarantees that matrix elements will be stored in contiguous locations in RAM (in row major order?).

Consider the following (admittedly artificial) sequence of statements.

```
>> dx = pi/30;  
>> nx = 1 + 2*pi/dx;  
>> nx2 = nx/2;  
  
>> for i = 1:nx2  
>>   x(i) = (i-1)*dx;  
>>   y(i) = sin(3*x(i));  
>> end  
  
>> for i = nx2+1:nx  
>>   x(i) = (i-1)*dx;  
>>   y(i) = sin(5*x(i));  
>> end
```

Since we know the size of  $x$  and  $y$ , a priori, we can pre-allocate memory for these vectors. Pre-allocation involves creating a matrix (or vector) with one vectorized statement before any of the matrix elements are referenced individually. The ones and zeros functions are typically used to pre-allocate memory.

Here is an improvement of the preceding statements with pre-allocation of memory for  $x$  and  $y$ .

```

>> dx = pi/30;
>> nx = 1 + 2*pi/dx;
>> nx2 = nx/2;

>> x = zeros(1,nx);    % pre-allocate row-vectors, x
>> y = zeros(1,nx);    % and y

>> for i = 1:nx2
>>   x(i) = (i-1)*dx;
>>   y(i) = sin(3*x(i));
>> end

>> for i = nx2+1:nx
>>   x(i) = (i-1)*dx;
>>   y(i) = sin(5*x(i));
>> end

```

The statements  $x(i) = \dots$ , and  $y(i) = \dots$  still do not take advantage of possibilities for vectorization, but at least the elements of  $x$  and  $y$  are stored contiguously in RAM. We will improve the efficiency of the loops shortly. First, however, note that we could have written the pre-allocation statements as

```

>> x = zeros(1,nx);    % pre-allocate row-vectors, x
>> y = x;              % and y

```

The statement  $y = x$  does not mean that  $y$  will stay equal to  $x$ . It simply creates another matrix  $y$  with the same "shape" as  $x$ . Understanding that pre-allocation is important for efficiency will help you understand these apparently confusing twists of MATLAB programming logic.

We can further improve our loops by pulling the assignment of  $x$  out of the loops.

```

>> x = 0:pi/30:2*pi;    % vectorized calculation of x
>> nx = length(x);
>> nx2 = nx/2;

>> y = x;              % pre-allocate memory for y

>> for i = 1:nx2
>>   y(i) = sin(3*x(i));
>> end

>> for i = nx2+1:nx
>>   y(i) = sin(5*x(i));
>> end

```

Finally, being obsessed with performance, we observe that the calculation of  $y$  can also be vectorized. To do this we use the colon notation to refer to segments of the  $x$  and  $y$  vectors.



```
>> x = 0:pi/30:2*pi;    % vectorized calculation of x
>> nx = length(x);
>> nx2 = nx/2;

>> y = x;                % pre-allocate memory for y

>> y(1:nx2) = sin(3*x(1:nx2));    % compute first part of y
>> y(nx2+1:nx) = sin(5*x(nx2+1:nx)); % and the second part
```

To those new to MATLAB programming, the preceding statements may appear unnecessarily obfuscated. The comment statements, of course, help, but the logic behind the logic comes from a true understanding of vectorization. Once you get the hang of MATLAB's colon notation you too will come to write code like this. Whenever the speed of MATLAB code is important, there is no substitute for vectorization.