Lectures 4 and 5
Introduction to Matlab

Material from MATLAB for Engineers, Moore, Chapter 1-4
Additional material by Peter Kovesi.
Objectives

◊ Understand what MATLAB is and why it is widely used in engineering and science
◊ Formulate problems by using a structured problem-solving approach
◊ Introduction to MATLAB calculations
◊ Scripts
◊ Functions
Introduction to Matlab

◊ High level language
◊ Developed for linear algebra, but substantially extended since then
◊ Matlab (Matrix Laboratory)
◊ Excels at numerical calculations, especially involving matrices
◊ Good for graphics e.g. creating charts, image processing
◊ Many additional toolboxes extend capabilities
  • Optimisation, control etc
Why MATLAB?

◊ Easy to use
◊ Versatile
◊ Built in programming language
◊ Convenient numerical solvers
◊ Not a general purpose language like C++ or Java
Find out more

◇ Matlab website:


◇ Student version (~$100):


◇ Demo Data and Resources:
  • Videos and examples

  • Webinar: http://www.mathworks.com/cmspro/webconf14845.html
What’s it look like?
◊ Command line
◊ Interactive i.e type command, get output, type new command
◊ Can also be used with script files known as m-files (more on these later)
The Command Window in MATLAB shows:

- **List of files** (e.g., m-files)
- **Past commands with output**
- **User defined variables with values**
- **History of commands**

The Command prompt is located at the bottom left, ready for input.
Student Edition of MATLAB

◊ MATLAB comes in both a student and professional edition

◊ Student editions are available for
  • Windows Operating Systems
  • Mac OS
  • Linux

◊ The student edition typically lags the professional edition by one release

◊ New releases are issued twice a year.
The student edition of release 2011a includes

◊ Full featured MATLAB 7.12
◊ Simulink 7.7
◊ Symbolic toolbox based on MuPad
◊ Limited number of other commonly used toolboxes
The command prompt is the biggest difference you'll notice

>> is the command prompt for the professional version

EDU>> is the command prompt for the student version
Free Matlab-like software

◊ GNU Octave: http://www.gnu.org/software/octave/
◊ Freemat: http://freemat.sourceforge.net/
◊ Scilab: http://www.scilab.org/

◊ Discussions on pros and cons of each
  • http://amca01.wordpress.com/2011/08/31/the-best-matlab-alternative/
  • http://userpages.umbc.edu/~gobbert/papers/SharmaGobbertTR2010.pdf

◊ Programming differences between Matlab and Octave:
  Differences_between_Octave_and_MATLAB
How is MATLAB used in Industry?

◊ Widespread, especially in the signal processing field
◊ Tool of choice in Academia for most engineering fields
◊ Some examples….
Biomedical Engineering

These images were created from MRI scan data using MATLAB. The actual data set is included with the standard MATLAB installation, allowing you experiment with manipulating the data yourself.
Image Analysis

Cell volume and spatial localisation of intracellular proteins from 2D confocal microscope image stacks

Tan et al. PloS One 7(2), 2012
Modelling bone cell communication

Pivonka et al., Bone, 2008
Results from a finite element analysis code were post processed using MATLAB to create this image.
Matlab as a calculator...

- MATLAB is a high level language that allows you to perform calculations on numbers, or arrays of numbers, in a very simple way.
- For example at the prompt within the MATLAB command window you can type
  \[
  \text{\texttt{3 + 4}}
  \]
  MATLAB will evaluate this for you and report the answer
  \[
  \text{\texttt{ans = 7}}
  \]
- Since MATLAB is \textit{interpreted} anything you can do in a program you can do from the command line, and vice versa
- → quick and easy prototyping
Variables

◊ You can also work with variables

\[
\begin{align*}
\text{\texttt{a}} &= 4 \\
\text{\texttt{b}} &= 6 \\
\text{\texttt{c}} &= \text{\texttt{a}} \times \text{\texttt{b}}
\end{align*}
\]

◊ These instructions set a variable called ‘a’ to the value of 4, a variable called ‘b’ to 6, and then multiplies ‘a’ and ‘b’ and stores the answer in a variable called ‘c’. (Computers use ‘*’ to indicate multiplication).

◊ The variables ‘a’, ‘b’ and ‘c’ remain available for use in other calculations.

◊ Note that if the value of ‘a’ is changed later in the program the value of ‘c’ remains unchanged (unlike a spreadsheet program where all values would be recalculated when you change one value)
# Arithmetic Operations between Two Scalars

<table>
<thead>
<tr>
<th>Operation</th>
<th>Algebraic Form</th>
<th>Matlab Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>$a + b$</td>
<td>a+b</td>
</tr>
<tr>
<td>Subtraction</td>
<td>$a - b$</td>
<td>a-b</td>
</tr>
<tr>
<td>Multiplication</td>
<td>$a \times b$</td>
<td>a*b</td>
</tr>
<tr>
<td>Division</td>
<td>$a ÷ b$</td>
<td>a/b</td>
</tr>
<tr>
<td>Exponentiation</td>
<td>$a^b$</td>
<td>a^b</td>
</tr>
</tbody>
</table>
# Logical Operations between Two Scalars

<table>
<thead>
<tr>
<th>Operation</th>
<th>Algebraic Form</th>
<th>Matlab Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equals</td>
<td>$a = b$</td>
<td>a==b</td>
</tr>
<tr>
<td>Not Equals</td>
<td>$a \neq b$</td>
<td>a~=b</td>
</tr>
<tr>
<td>Greater Than</td>
<td>$a &gt; b$</td>
<td>a&gt;b</td>
</tr>
<tr>
<td>Less Than</td>
<td>$a &lt; b$</td>
<td>a&lt;b</td>
</tr>
<tr>
<td>Greater Than/Equals</td>
<td>$a \geq b$</td>
<td>A&gt;=b</td>
</tr>
</tbody>
</table>
Comments

◊ An important part of programming is documenting your work. A program should convey both a message to the computer and a message to the human programmers who read the program.

◊ You need to document your programs so that your assumptions and solution techniques can be readily examined for their correctness by you and others.

◊ Matlab uses the the percent symbol (%) to indicate the start of a comment statement.

◊ Anything on a line after a % symbol is ignored by Matlab.

```matlab
>> 3 * 4 % + 7 (I would never do this)
ans =
    12
```
Suppressing output

◊ By default, Matlab will display the answer to every assignment statement you enter. For example:

```matlab
>> halfPi = pi / 2
halfPi =
     1.5708
```

◊ You can suppress the outputting of the answer by adding a semicolon (;) to the end of a statement. For example, the command:

```matlab
>> halfPi = pi / 2;
```

will still set the value of the variable `halfPi` to 1.5708, but will not print the answer on the screen.

(Important when you come to dealing with big matrices!)
Simple Expressions

◊ All the commands listed above form *expressions*. An expression is a valid statement that MATLAB can interpret or 'understand'.

◊ A statement is valid if it satisfies the *syntax* (the 'grammar') of the language.

◊ As the course progresses we will cover more and more of MATLAB's syntax, but at this stage you will find that any simple mathematical expression, expressed correctly, will have valid syntax.

◊ For example:

\[
b = 4 \times (a + 3); \quad \% \text{ is valid}
\]

\[
c = a + \quad \% \text{ invalid - incomplete expression.}
\]

\[
d = (b + 3)) / 6;\% \text{ invalid - unmatched brackets.}
\]
Operators

◊ The operators used for writing basic mathematical expressions are:

◊ + addition
◊ − subtraction
◊ * multiplication
◊ / division
◊ ^ exponentiation

◊ A common mistake can be to omit the multiplication operator when writing expressions, for example writing

\[ b = 4 (a+3); \]  

rather than

\[ b = 4* (a+3); \]
Operator Precedence

◊ Operators have *precedence*, that is, in an expression involving several operators some will be applied before others. This ensures uniqueness in interpreting an expression.

◊ Expressions are evaluated from left to right with exponentiation having highest precedence, followed by multiplication and division with equal precedence, and addition and subtraction at an equal (lower) precedence.

\[ a = b+c*d^e/f-g; \]

will be evaluated as

\[ a = b+(c*(d^e)/f)-g; \]

not (for example) \[ a = (((b+c)*d)^e)/f-g \]

◊ Where appropriate use brackets (even if they are not strictly necessary) to make expressions easier to read and interpret.
Identifiers/Variables

◊ An *identifier* or a *variable* in a programming language gives a name to a specific memory location. When a command such as

\[ x = 5.2 \]

is evaluated the system chooses a memory location, associates it with the identifier \( x \), and stores the value \( 5.2 \) in that location. Thus the name ‘\( x \)’ serves as a place-name for the location in memory where our value is stored.

◊ We never have to worry what the actual memory location is.
Variable Names

- MATLAB has some rules regarding the names of identifiers/variables (more commonly we use the term 'variable').

1. Variable names must **start with a letter**, this can then be **followed by any number of letters, numerals or underscores**. Punctuation characters are not allowed as many of these have a special meaning to MATLAB.

2. Variable names are case sensitive. **Items**, **items** and **iTeMs** are all separate variables.

- (Most computer languages have these rules)

- MATLAB maintains some special variables, some of these are:
  - **ans** - the default variable name used for results of calculations.
  - **pi** - ratio of circle circumference to its diameter.
  - **i** and **j** - the square root of -1.
Problem Solving in Engineering and Science

1. State the Problem
2. Describe the input and output
3. Develop an algorithm
4. Solve the problem
5. Test the solution
State the Problem

◊ If you don’t have a clear understanding of the problem, it’s unlikely that you’ll be able to solve it
◊ Drawing a picture often helps you understand the system better
Describe the Input and Output

◊ Be careful to include units
◊ Identify constants
◊ Label your sketch
◊ Group information into tables
Develop an Algorithm

◊ Identify any equations relating the ‘knowns’ and ‘unknowns’
◊ Work through a simplified version of the problem by hand or with a calculator
◊ Developing a flow chart is often useful for complicated problems. Flow charts are particularly useful for planning computer programs to solve a problem
Solve the problem

◊ Create a MATLAB solution
◊ Be generous with comments, so that others can follow your work
Test the Solution

◊ Compare to the hand solution
◊ Do your answers make sense physically?
  ◇ ...try a number of cases
  ◇ ...guess (estimate) the solution before solving and see if the two predictions match.
◊ Is your answer really what was asked for? Are you addressing the original problem?
◊ Graphs are often useful ways to check your calculations for reasonableness

If you use a consistent problem solving strategy you increase the chance that your result is correct. Here’s an example....
Example 1.1

- Albert Einstein
- $E=mc^2$
- The sun is fueled by the conversion of matter to energy
- How much matter does the sun consume every day?
State the Problem

◊ Find the amount of matter necessary to produce the amount of energy radiated by the sun everyday
Describe the Input and Output

◊ Input

• E=mc²

• Rate of energy radiation
  – E = 385*10^{24} Joules/second

• Speed of light
  – c = 3.0*10^8 meters/second

◊ Output

• Mass (m) in kilograms
Develop an Algorithm – Hand Example

◊ The energy radiated in one day is:

$$385 \times 10^{24} \frac{J}{sec} \times 3600 \frac{sec}{hour} \times 24 \frac{hours}{day} \times 1 \text{ day} = 3.33 \times 10^{31} J$$

◊ Rearrange $E=mc^2$ and solve for $m$

- $m=E/c^2$

$$m = \frac{3.33 \times 10^{31} J}{(3.0 \times 10^8 m/\text{sec})^2} = 3.7 \times 10^{14} \frac{J}{m^2/\text{sec}^2}$$
But the units are wrong!!

\[ 3.7 \times 10^{14} \frac{J}{m^2/\text{sec}^2} \]

\[ 1 \, J = 1 \, kg \, m^2/\text{sec}^2 \]

\[ = 3.7 \times 10^{14} \frac{kg \, m^2/\text{sec}^2}{m^2/\text{sec}^2} = 3.7 \times 10^{14} \, kg \]
The University of Western Australia

Once you hit the enter key, the program repeats your input. Notice the use of scientific notation in the result.

Now enter the equation to change the energy rate from kJ/day to kJ/s. Notice that the value of E is updated based on your calculation.

Enter the value for c, the speed of light. Once again, the result is repeated back to you in the command window.

Now enter the equation to calculate the mass.
Test your Solution

◊ Matches the hand solution
◊ Is it reasonable?
◊ Consider…
  • Mass of the sun = $2 \times 10^{30}$ kg
  • How long would it take to consume all that mass?
time = (mass of the sun)/(rate of consumption)

\[
time = \frac{2 \times 10^{30} \text{ kg}}{3.7 \times 10^{14} \text{ kg/day}} \times \frac{\text{ year}}{365 \text{ days}} = 1.5 \times 10^{13} \text{ years}
\]

That’s 15 trillion years!!
Yes – this is a reasonable result
Types

◊ Most computer languages have the concept of a *type* of a variable. For example, a variable may be an *integer*, a *character*, or a *floating point number*. The type specifies how the value of that variable is stored in the computer.

◊ For example, some of the types available in the Java programming language include:

<table>
<thead>
<tr>
<th>Type</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>Stores a character using 16 bits (unsigned).</td>
</tr>
<tr>
<td>int</td>
<td>Stores an integer using 32 bits (two’s complement).</td>
</tr>
<tr>
<td>float</td>
<td>Stores a floating point value (a non-integer) using 32 bits (4 bytes). These 32 bits are used differently from the storage of an integer.</td>
</tr>
<tr>
<td>double</td>
<td>Stores a floating point value using 64 bits (double precision)</td>
</tr>
</tbody>
</table>
Type Casting

◊ When two variables of different types are combined together (e.g. added together), some form of conversion, or *casting*, has to be performed to match the types.

◊ Without casting, strange results will occur because in binary terms, we are adding “apples and oranges” - combining things that are represented in completely different ways.

◊ In the case of adding an integer to a float, typically the integer would be *promoted* to become a float. The result would be a float.

◊ Fortunately, Matlab has very few types. The types we are likely to encounter are:

<table>
<thead>
<tr>
<th>Type</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>Characters</td>
</tr>
<tr>
<td>double</td>
<td>Double precision numbers</td>
</tr>
<tr>
<td>complex</td>
<td>Complex numbers represented using two doubles</td>
</tr>
</tbody>
</table>
Weak Typing

◊ In almost all cases, Matlab automatically determines the type that should be used for a variable and casts as necessary.

◊ Matlab also automatically handles any conversions between types that need to be made to evaluate an expression.

◊ $\rightarrow$ weakly typed

◊ Unlike many programming languages, Matlab does not require a programmer to make any distinction between integers, doubles, or complex numbers. (This is great, but could there be disadvantages? How could strong typing be an advantage?)

◊ For example, we could type:

\begin{verbatim}
>> a = 'B'; % Characters enclosed in quotes
>> b = 1;
>> c = 1.5;
>> d = 2 + 3*i; % Can also write d = 2 + 3i
>> e = c*d; % Matlab automatically handles the complex multiplication
\end{verbatim}
Getting to know the type in Matlab

◊ The `whos` command prints out information about the currently active variables. For example:

```matlab
>> whos
Name      Size  Bytes  Class      
  a   1x1    2   char array   
  b   1x1    8   double array  
  c   1x1    8   double array  
  d   1x1   16   double array (complex)
  e   1x1   16   double array (complex)
Grand total is 5 elements using 50 bytes
```
Release Memory

◊ The `clear variableName` command will remove a variable from memory:

```
>> clear a

>> whos

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Bytes</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>1x1</td>
<td>8</td>
<td>double array</td>
</tr>
<tr>
<td>c</td>
<td>1x1</td>
<td>8</td>
<td>double array</td>
</tr>
<tr>
<td>d</td>
<td>1x1</td>
<td>16</td>
<td>double array (complex)</td>
</tr>
<tr>
<td>e</td>
<td>1x1</td>
<td>16</td>
<td>double array (complex)</td>
</tr>
</tbody>
</table>

Grand total is 4 elements using 48 bytes
The Distinction between Assignment and Equality

◊ When we write:

```matlab
>> a = 2;
>> b = a;
>> a = a*3;
```

the `=` sign in this code does not mean equality in the mathematical sense (clearly it cannot). The `=` symbol really means the operation of *assignment*, sometimes read “becomes equal to”.

◊ We can read the expression `a = a*3` as:

1. Read the value stored at `a`.
2. Multiply the value by 3.
3. Assign this new value to the memory location referred to by `a`. 
A close look at assignment

◊ The value of 2 stored at the memory location referred to by b remains unchanged.

```
>> a = 2;
>> b = a;
>> a = a*3;
```

◊ Pseudo-code often uses the symbol ← (or :=) to denote assignment rather than the = symbol, so that the distinction between the two is made clear.

eg.  a ← a*3

◊ Testing for equality between two values is done with a different operator (more later).
Many expressions on one line

◊ You can put several expressions on one line with the semi-colon (;) operator (though I wouldn’t normally advise it).

◊ For example:

```plaintext
>> a = 'B'; b = 1; c = 1.5;
```

◊ The ; acts as a separator between expressions.

◊ If you want the result of each expression in the line to be printed to the screen, commas (,) are used as statement separators. For example:

```plaintext
>> a = 'B', b = 1, c = 1.5
```
One Expression on Many lines

◊ You can break an expression over more than one line using ... to indicate a continuation.

◊ For example:

```plaintext
>> a = 1 ...
   + 2 ...
   + 3;
```

◊ In general, seek to format your expressions to aid readability and understanding.
Script files

- Matlab allows commands to be entered interactively at the command prompt, but this is not really appropriate for extended sequences of commands. Instead, we can create a script file - a file that contains a sequence of commands (a script) for Matlab to follow.

- A script file is simply a text file that contains the sequence of Matlab commands/expressions to follow. Script files can be created using any text editor (e.g. Emacs) or by using the Matlab internal editor.

- By convention, script files should be saved with a .m ending.

- Script files are executed by typing the name of the script without the .m ending.

- The commands in the script file are executed as if they had been typed in at the command prompt.
The University of Western Australia

MATLAB

File  Edit  Debug  Parallel  Desktop  Window  Help

New
- Open...
- Close Command Window
- Import Data...
- Save Workspace As...
- Set Path...
- Page Setup...
- Print...
- Print Selection...

Script
- Function
- Class
- Enumeration
- Figure
- Variable
- Model
- GUI
- Deployment Project...

1 /Users/... models/CA_2d.m
2 /Users/...lab models/CA.m

Getting Started
Script output

Script name. Note ends with “.m”

Run Script

Note script can also be run by typing script name at prompt

Script commands

Current Folder: /Users/bruceg/Documents/MATLAB

New to MATLAB? Watch this Video, see Demos, or read Getting Started.
m-files must be saved in a “file path” that MATLAB knows to look in
Matlab Functions

◊ A function is similar to a script in that it is a set of Matlab commands stored in a file.
◊ The name of the file is the same as the name of the function with a .m ending.
◊ A function differs from a script in that it operates within its own private workspace.
◊ A function is also different in that it can accept arguments and return values, something a script cannot do.
◊ The typical layout of a .m function file...

First line must be:

```
function [output1, output2,...] = fName(inputArg1,inputArg2,...)
```
Using the Help Feature

◊ There are functions for almost anything you want to do
◊ Use the help feature to find out what they are and how to use them
  • From the command window
  • From the help selection on the menu bar
From the Command Window

>> help sqrt

SQRT    Square root.

    SQRT(X) is the square root of the elements of X. Complex
    results are produced if X is not positive.

See also sqrtm.

Overloaded functions or methods (ones with the same name in other directories)

    help sym/sqrt.m

Reference page in Help browser

    doc sqrt
From the Help Menu
Mathematics

Arrays and Matrices

Linear Algebra

Elementary Math

Basic array operators and operations, creation of elementary and specialized arrays and matrices

Matrix analysis, linear equations, eigenvalues, singular values, logarithms, exponentials, factorization

Trigonometry, exponentials and logarithms, complex values, rounding, remainders, discrete math

Multiplication, division, evaluation, roots, derivatives, integration, eigenvalue problem, curve fitting, partial fraction expansion

Interpolation and Computational Geometry

Conversions between Cartesian and polar or spherical coordinates

Nonlinear Numerical Methods

Differential equations, optimization, integration

Specialized Math

Airy, Bessel, Jacobi, Legendre, beta, elliptic, error, exponential integral, gamma functions

Sparse Matrices

Elementary sparse matrices, operations, reordering algorithms, linear algebra, iterative methods, tree operations
sqrt
Square root of fi object

Syntax

\[
c = \text{sqrt}(a)
\]
\[
c = \text{sqrt}(a, T)
\]
\[
c = \text{sqrt}(a, F)
\]
\[
c = \text{sqrt}(a, T, F)
\]

Description

This function computes the square root of a fi object using a bisection algorithm.

\(c = \text{sqrt}(a)\) returns the square root of fi object \(a\) with the same \(\text{fi}math\) object as \(a\). Intermediate quantities are also calculated using the \(\text{fi}math\) object of \(a\). The \text{numerictype} object of \(c\) is determined automatically for you using an internal rule.

\(c = \text{sqrt}(a, T)\) returns the square root of fi object \(a\) with \text{numerictype} object \(T\) and the same \(\text{fi}math\) object as \(a\). Intermediate quantities are calculated using the \(\text{fi}math\) object of \(a\). See Data Type Propagation Rules.

\(c = \text{sqrt}(a, F)\) returns the square root of fi object \(a\) with \(\text{fi}math\) object \(F\). Intermediate quantities are also calculated using \(\text{fi}math\) object \(F\). The \text{numerictype} object of \(c\) is determined automatically for you using an internal rule. When \(a\) is a built-in double or single data type, this syntax is equivalent to \(c = \text{sqrt}(a)\) and the \(\text{fi}math\) object \(F\) is ignored.

\(c = \text{sqrt}(a, T, F)\) returns the square root fi object \(a\) with \text{numerictype} object \(T\) and \(\text{fi}math\) object \(F\). Intermediate quantities are also calculated using \(\text{fi}math\) object \(F\). See Data Type Propagation Rules.

\text{sqrt} does not support complex, negative-valued, or [Slope Bias] inputs.

Internal Rule

For syntaxes where the \text{numerictype} object of the output is not specified as an input to the \text{sqrt} function, it is automatically calculated according to the following internal rule:

\[
sign_c = \text{sign}_a
\]
\[
WL_c = \text{ceil}\left(\frac{WL_a}{2}\right)
\]
MATLAB - Matrices
Objectives

◊ Manipulate matrices
◊ Extract data from matrices
◊ Solve problems with two variables
◊ Explore some of the special matrices built into MATLAB
Arrays and vectors

- An array is a collection of data objects of the same type, typically stored sequentially in memory.
- An array is the most elementary data structure.
- Almost all programming languages provide support for arrays.
- Matlab is a language that has been particularly specialised to support arrays (and subsequently matrices).
Arrays and vectors

◊ An array is the obvious way to represent a vector.

Equation:
\[ A_1 x_1 + A_2 x_2 + A_3 x_3 = 0 \]

Equation expressed using two 1x3 arrays

\[ [A_1, A_2, A_3] [ \begin{array}{c} x_1 \\ x_2 \\ x_3 \end{array} ] = 0 \]
In Matlab, we can construct an array and associate it with an identifier very easily. For example:

```matlab
>> a = [1, 2.5, 5]
```

```
a =
    1.0000   2.5000   5.0000
```

The commas are optional and can be omitted:

```matlab
>> a = [1 2.5 5]
```

```
a =
    1.0000   2.5000   5.0000
```

The square brackets indicate to Matlab that the contents represent an array:

```matlab
>> whos
```

```
Name      Size         Bytes  Class
a         1x3             24  double array
```

Grand total is 3 elements using 24 bytes
Array arithmetic

◊ In most programming languages you would have to declare an array and assign the values one at a time. In Matlab this is automatic.

◊ You can then perform arithmetic on arrays as simply as you can with scalars. For example:

```matlab
>> b = a*2
b =
    2     5    10
```
Array multiplication

◊ You can then perform multiplication between arrays (remembering your vector maths)

```matlab
>> a = [1, 2, 5]
a =
  1   2   5

>> b = [2; 3; 2]
b =
  2
  3
  2

>> a*b = 18
```
Working on Array Elements

◊ You can extract individual values from an array by specifying the index within the array using round brackets. For example:

```matlab
>>b = [2 5 10];
>>c = b(1) % c is set to the value of the first element of b
```

```
c =
  2
```

◊ You can also assign new values to individual elements of an array. For example:

```matlab
>> b(3) = 6 % Set the value of the 3rd element of b to 6
```

```
b =
 2 5 6
```

◊ In Matlab, the index of the first element of an array is always 1.

◊ Note: this differs from languages such as C or Java where the index of the first element is always 0.
Matrices in Matlab

- The data objects of the array can themselves be arrays.
- A *matrix* is typically represented by an array of arrays, or a 2D array. Matlab supports matrices in the same way that it supports vectors.
- Matlab uses the semi-colon (;) operator to distinguish between the different rows of a matrix. For example:

```matlab
>> a = [1 2 3; 4 5 6] % The ; separates the individual 1D arrays.

a =
    1     2     3
      4     5     6
```
Everything in Matlab is a Matrix

◊ Matlab also allows rows to be entered on different lines.

◊ Once an array is started by a square bracket ([), Matlab assumes that a new line means a new row of the matrix. For example:

```matlab
>> a = [1 2 3 % A matrix consisting of two rows
        4 5 6 ];
```

◊ As far as Matlab is concerned, *everything is a matrix*!

◊ A vector is a 1xN (or Nx1) matrix; a scalar is a 1x1 matrix.
We can define a matrix using the following syntax

◊ \(A = [3.5]\)
◊ \(B = [1.5, 3.1]\) or
◊ \(B = [1.5\ 3.1]\)
◊ \(C = [-1, 0, 0; 1, 1, 0; 0, 0, 2]\)

2-D Matrices can also be entered by listing each row on a separate line

\[
C = \begin{bmatrix}
-1 & 0 & 0 \\
1 & 1 & 0 \\
1 & -1 & 0 \\
0 & 0 & 2
\end{bmatrix}
\]
Scalar

```
>> A = 3.5;

Scalar
```

```
clc
x = [1 2 3; 4 5]
mean(x)
std(x)
x(:)
mean(x(:))
std(x(:))
clear, clc
A = 3.5;
```
Vector – the commas are optional
These semicolons are optional.

2-D matrix
You can define a matrix using other matrices as components

```matlab
>> B = [1.5, 3.1];
>> S=[3.0,B]
S =
    3.0000    1.5000    3.1000
>>
```
Or...

```matlab
>> B = [1.5, 3.1];
>> S=[3.0,B]
S =
     3.0000    1.5000    3.1000
>> T=[1,2,3;S]
T =
     1.0000    2.0000    3.0000
     3.0000    1.5000    3.1000
```
Indexing Into an Array allows you to change a value

```
>> s
s =

    3.0000    1.5000    3.1000
```
Adding Elements

```
>> S
S =
  3.0000  1.5000  3.1000
>> S(2)=1.0
S =
  3.0000  1.0000  3.1000
>> S(4)=5.5
S =
  3.0000  1.0000  3.1000  5.5000
```

If you add an element outside the range of the original array, intermediate elements are added with a value of zero.
Colon Operator

◊ Used to define new matrices
◊ Modify existing matrices
◊ Extract data from existing matrices

◊ zeros  Creates a matrix of all zeros
◊ ones   Creates a matrix of all ones
◊ diag   Extracts a diagonal or creates an identity matrix
◊ magic  Creates a “magic” matrix
◊ eye    Create an identity matrix
Evenly spaced vector

```
>> H=1:8
H =
    Columns 1 through 7
       1     2     3     4     5     6     7
    Column 8
       8
>>
```

User specified spacing

```
>> time=0.0:0.5:2.0
time =
    Columns 1 through 4
       0    0.5000    1.0000    1.5000
    Column 5
       2.0000
>>
```
The colon can be used to represent an entire row or column.

```matlab
>> M = [1 2 3 4 5
       2 3 4 5 6
       3 4 5 6 7];
>> x = M(:, 1)
x =
    1
    2
    3

>> y = M(:, 4)
y =
    4
    5
    6

>> z = M(1, :)
z =
    1
    2
    3
    4
    5
```

All the rows in column 1:

1
2
3

All the rows in column 4:

4
5
6

All the columns in row 1:

1
2
3
4
5
You don’t need to extract an entire row or column
Indexing techniques

- To identify an element in a 2-D matrix use the row and column number.
- For example element M(2,3)

Element M(2,3) is in row 2, column 3.

Or use single value indexing:

M(8) is the same element as M(2,3).

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>7</td>
<td>10</td>
<td>13</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>8</td>
<td>11</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>9</td>
<td>12</td>
<td>15</td>
</tr>
</tbody>
</table>
Problems with Two Variables

- All of our calculations thus far have only included one variable
- Most physical phenomena can vary with many different factors
- We need a strategy for determining the array of answers that results with a range of values for multiple variables
Two scalars give a scalar result

A scalar and a vector give a vector result
A scalar and a vector give a vector result

```
>> x=1:5
x =
    1     2     3     4     5
>> y=5;
>> A=x*y
A =
    5    10    15    20    25
```
When you multiply two vectors together, they must have the same number of elements.

Array multiplication gives a result the same size as the input arrays.
The meshgrid function maps two vectors onto a 2-D grid

Now the arrays are the same size, and can be multiplied