A Brief Introduction to MATLAB*

Kwa Kiam Heong

November 10, 2019

These notes make no attempt to cover either the range of topics or the depth of detail one can find in a reference manual or in a standard text. Some such good references are

1. Essential MATLAB for Engineers and Scientists by Brian D. Hahn and Daniel T. Valentine [2];
2. MATLAB Programming with Applications for Engineers by Stephen J. Chapman [1];
3. Learning to program with MATLAB by Craig S. Lent [3].

There is also a free tutorial

4. A MATLAB Tutorial by Ed Oveman [4] available at

   `https://people.math.osu.edu/overman.2/`

Last but not least, MathWorks® has a complete documentation on MATLAB online at

`https://www.mathworks.com/help/matlab/`

It is definitely worthwhile to browse through the documentation on the MATLAB language fundamental at

`https://www.mathworks.com/help/matlab/language-fundamentals.html`

However, do not expect all materials in the documentation do make sense to you; they will make more sense as you move along.

These notes were designed initially in 2013 to be a concise reference for students learning MATLAB while taking the course SJEM1130 Introduction to

*MALTAB was initially a MATtix LABoratory.

Computing at Institute of Mathematical Sciences, University of Malaya. A long time ago indeed! Now they will be updated for the course SIN1001 Introduction to Computing, not radically though, as I attempt to improve them.

MATLAB is expensive. However, as a student of University of Malaya, you should be able to install a licensed copy on your machine by creating and signing in your MathWorks account at

`https://www.mathworks.com/login?uri=https%3A%2F%2Fwww.mathworks.com%2Fcloud.html`

On the other hand, an almost compatible but free system is provided by GNU Octave. For interested students, instructions for installation are avaiable at

`https://www.gnu.org/software/octave/`

together with its documentation.

As an alternative for Windows users, one can install Cygwin, a large collection of GNU and Open Source tools which provide functionality similar to a Linux distribution on Windows, available at

`http://cygwin.com/`

and install Octave as one of its packages.

One can also try GNU Octave online at

`http://octave-online.net/`

Contents

1 Introduction

MATLAB has a huge number of commands and functions. Usually we do not discuss the complete behaviors of these commands and functions – since it is unlikely that we are able to do so – but only their relevant behaviors. Entering

```
help <command>
```

in the Command Window gives you complete information about the command or function <command>.

Notation: Here <command> is the character vector – that is, text enclosed in single quotation marks – that names the command or function.

For instance, to get information about the sine function, we enter

```
help 'sin'
```

in the Command Window. This yields

```
sin    Sine of argument in radians.
    sin(X) is the sine of the elements of X.

    See also asin, sind, sinpi.

    Reference page for sin
    Other functions named sin
```

By the way, in view of the **command-function duality**, the above statement is equivalent to

```
help('sin')
```

More about command and function syntaxes can be found at

`https://www.mathworks.com/help/matlab/matlab_prog/command-vs-function-syntax.html`

To search for commands which might be of use, use the command `lookfor` instead. For instance, entering

```
lookfor sine
```

returns all MATLAB commands that have something to do with the sine function. An entire reference manual on-line can also be accessed by using the command `doc`'. To retrieve information about a specific command or function, the syntax is

```
doc <command>
```

MATLAB documentation by MathWorks is available at

`https://www.mathworks.com/help/matlab/index.html`

To keep track of everything done during a particular MATLAB session, we can use the command `diary`. The syntax is

```
diary <filename>
```

where `<filename>` is the character vector denoting the name of the file into which to save the current session. If the file name `<filename>` is omitted, then the file 'diary' is created in the current working directory. After this command is issued, a copy of all input and most output is echoed into the diary file `<filename>` until the command

```
diary off
```

is entered to suspend it. To turn it back on, we enter

```
diary on
```

The command `diary`, by itself, toggles the diary state.

MATLAB documentation by MathWorks on `diary` is available at

`https://www.mathworks.com/help/matlab/ref/diary.html`

Next, to clear the contents of the Command Window, we use the command `clc`.

MATLAB documentation by MathWorks on `clc` is available at

`https://www.mathworks.com/help/matlab/ref/clc.html`

Finally, a few common directory manipulation commands are:

- `pwd` displays the current working directory;
- `cd` changes the current working directory;
- `dir` and `ls` list the content of the current working directory.

.

MATLAB documentation by MathWorks on `pwd` is available at

`https://www.mathworks.com/help/matlab/ref/pwd.html`

MATLAB documentation by MathWorks on `cd` is available at

`https://www.mathworks.com/help/matlab/ref/cd.html`

MATLAB documentation by MathWorks on `dir` is available at

`https://www.mathworks.com/help/matlab/ref/dir.html`

MATLAB documentation by MathWorks on `ls` is available at

`https://www.mathworks.com/help/matlab/ref/ls.html`

1.1 Simple Arithmetic Operations

MATLAB offers the standard arithmetic operations for scalars listed in the following table.

Syntax	Operation
`a+b`	Addition.
`a-b`	Subtraction.
`a*b`	Multiplication.
`a/b`	Right division.
`a\b`	Left division.
`a^b`	Exponentiation.
`mod(a,b)`	Modulo operation.

Table 1.1.1: Standard scalar arithmetic operations. Here `a` and `b` are scalars.

Note: `a\b` is exactly the same to `b/a`.

MATLAB follows the standard order of operations used throughout mathematics, science, technology, and many computer programming languages. The order of operations can be overriden by using parentheses (`()`).

Example 1.1.1 *The sum* $3.17 \times 5.7 - 16/3$ *can be calculated by entering*

```
3.17*5.7-16/3
```

and MATLAB responds by displaying

```
ans =

   12.7357
```

Likewise, the sum $\sum_{i=1}^{10} i/(i+1)$ *can be calculated by entering*

```
1/2+2/3+3/4+4/5+5/6+6/7+7/8+8/9+9/10+10/11
```

and MATLAB echos by displaying

```
ans =

    7.9801
```

Example 1.1.2 *A number in scientific notation can be entered using the caret operator (^). For instance, the number* 2.4×10^{-5} *can be entered as*

```
2.4*10^-5
```

and MATLAB returns

```
ans =

  2.4000e-005
```

MATLAB, however, uses `e` *and* `E` *to represent "*`10^`*" so that the number* 2.4×10^{-5} *can be entered as*

```
2.4e-5
```

and MATLAB returns

```
ans =

  2.4000e-005
```

as before.

Warning: $10\,\hat{}\,-5$ cannot be input as `e-5`, but must be input as `1e-5` or `1E-5` or `1.e-5` or `1.E-5`, etc.

MATLAB documentation by MathWorks on arithmetic operators is available at

`https://www.mathworks.com/help/matlab/matlab_prog/matlab-operators-and-special-characters.html`

MATLAB can also handle complex numbers, where `i` or `j` represents $i = \sqrt{-1}$.

Example 1.1.3 *The complex number* $3 + 5i$ *can be entered as*

```
3+5i
```

or

```
3+5*i
```

The output is

```
ans =

   3.0000 + 5.0000i
```

The complex number $5 \times 10^{30}i$ *can be entered as*

```
5e30i
```

or

```
5e30*i
```

or

```
5*10^30*i
```

The output is

```
ans =

        0.0000e+00 + 5.0000e+30i
```

Warning: The complex number $5 \times 10^{30}i$ **cannot** be entered as

```
5*10^30i
```

whose output is

```
ans =

   4.9965 - 0.1874i
```

This is so because MATLAB considers `5*10^30i` as the number 5×10^{30i} instead.

MATLAB documentation by MathWorks on complex numbers and related functions is available at

`https://www.mathworks.com/help/matlab/ref/i.html`

1.2 Initializing and Removing Variables

A **variable** is a region of memory containing certain information that is known by a user-specified name. A valid variable name starts with a letter, followed by letters, digits, or underscores (_). The maximum length of a variable name is the value that the `namelengthmax` command returns.

Warning: MATLAB is case sensitive, so `A` and `a` are not the same variable.

Variables are automatically created when they are initialized, which is commonly done through one of the following.

1. Assign data to the variable in an assignment statement.
2. Input data into the variable from the keyboard.
3. Read data from a file.

An **assignment statement** has the general form

```
var = expr
```

Here `var` is the name of a variable and `expr` is an expression which can be a scalar constant, an array, or the combination of constants, other variables, and mathematical operations which result in a value. The value of the expression is calculated using the normal rules of mathematics, the result of which is then stored in the variable.

It is worth noting that = is the **assignment operator** which assigns the value of `expr` to `var`. It should be distinguished from the comparison opetator == which is used to determine equality in MATLAB.

Once variables are created, they continue to exist in the **workspace** of current MATLAB session. We can list all current variables using the commands `who` or `whos`, the latter of which usually provides more details.

We can remove variables using the command `clear`.

Warning: The command `clear` is **very dangerous!** Entering

```
clear
```

removes all variables from the workspace.

Example 1.2.1 *Suppose we have typed in*

```
x = 1, y = 2
```

which yields

```
x =

     1


y =

     2
```

Entering

```
who
```

yields

```
Your variables are:

x  y
```

while entering

```
whos
```

yields

```
  Name      Size            Bytes  Class     Attributes

  x         1x1                 8  double
  y         1x1                 8  double
```

Then the variables `x` *can be removed by entering*

```
clear x
```

Entering either `who` *or* `whos` *now yields only information about* `y`.

To remove more than one variables, we can enter

```
clear x y
```

Example 1.2.2 *Suppose we have typed in*

```
x = 3, y = 4
```

Then the variables `x` *and* `y` *can be removed by typing*

```
clear
```

In fact, **all** *variables will have been removed! Typing either* `who` *or* `whos` *now indicates that no variables are defined.*

MATLAB documentation by MathWorks on `who` is available at

`https://www.mathworks.com/help/matlab/ref/who.html`

MATLAB documentation by MathWorks on `whos` is available at

`https://www.mathworks.com/help/matlab/ref/whos.html`

MATLAB documentation by MathWorks on `clear` is available at

`https://www.mathworks.com/help/matlab/ref/clear.html`

Example 1.2.3 *Character vectors can be stored as follows.*

```
my_name = 'Kwa Kiam Heong'
```

Note that a **character vector** *is enclosed in* **single quotes** *(`'`). Now entering* `myname` *at the MATLAB prompt gives*

```
my_name =

Kwa Kiam Heong
```

On the other hand, a **string array** *is enclosed in* **double quotes** *(`"`) such as*

```
yr_name = "Clark Ken"
```

String arrays was introduced in MATLAB R2017a. If you are using an earlier version of MATLAB, this won't work. The command `whos` *returns*

```
  Name        Size            Bytes  Class    Attributes

  my_name      1x14              28  char
  yr_name       1x1             156  string
```

MATLAB documentation by MathWorks on texts and characters is available at

`https://www.mathworks.com/help/matlab/learn_matlab/character-strings.html`

Example 1.2.4 *Variables can be input from the keyboard using the command* `input`*. For instance, typing*

```
my_age = input('Enter a value for my age: ');
```

causes MATLAB to prompt the user with the message

```
Enter a value for my age:
```

MATLAB then expects the user to enter a 'non-character' input. Suppose the user type in `39`*:*

```
Enter a value for my age: 39
```

Then the variable `myage` *constains the value* `39`*, which we can check by entering*

```
my_age
```

which gives

```
my_age =

    39
```

Example 1.2.5 *Character vectors can also be input from keyboard using the* `input` *command. For instance, entering*

```
my_name = input('What is my name? ', 's');
```

cuases MATLAB to prompt the user for a character input. The character `'s'` *signifies that the command* `input` *expects a character vector.*

Example 1.2.6 *In fact, variables of the form of matrices can also be input using* `input`*. A sample run is as follows:*

```
>> my_matrix = input('Enter a matrix my matrix: ')
Enter a matrix my matrix: [1 2 3; 4 5 6; 7 8 9]

my_matrix =

     1     2     3
     4     5     6
     7     8     9
```

More is true. The command `input` *also accepts expressions as inputs:*

```
>> rand_matrix = input('Enter a random integer matrix: ')
Enter a random matrix: randi([-10, 10], 3, 4)

rand_matrix =

    10     7    -2    -1
   -10     8    -5     9
     6    -9     6    -7
```

MATLAB documentation by MathWorks on `input` is available at

`https://www.mathworks.com/help/matlab/ref/input.html`

1.3 Predefined Variables

Predefined variable	Value
`ans`	The default variable name when one has not been specified. When you run MATLAB code that returns an output without specifying an output argument, MATLAB creates the `ans` variable and stores the output there.
`eps`	Approximately the smallest positive real number on the computer such that `1+eps`$\neq$`1`.
`Inf` or `inf`	IEEE arithmetic representation for positive infinity (as in `1/0`). It is also a function.
`NaN` or `nan`	Not-a-number. It results from operations that return undefined numeric output, such as `0/0` or `0*Inf`. Between, it is also a function.
`pi`	π
`realmax`	The largest "usable" positive real number on the computer. This is "approximately" the largest positive real number which can be represented on the computer.
`realmin`	The smallest "usable" positive real number on the computer. This is "approximately" the smallest positive real number which can be represented on the computer.

Table 1.3.1: Some predefined variables.

MATLAB documentation by MathWorks on `Inf` is available at

`https://www.mathworks.com/help/matlab/ref/inf.html?searchHighlight=Inf&s_tid=doc_srchtitle`

MATLAB documentation by MathWorks on `NaN` is available at

`https://www.mathworks.com/help/matlab/ref/nan.html?searchHighlight=Nan&s_tid=doc_srchtitle`

Warning: MATLAB considers `1/0` and `Inf` as equivalent. However, it should be born in mind that division by zero such as 1/0 is undefined in standard mathematics.

1.4 Formatting Output Data

The `format` command changes how numerical results are displayed. It does not affect how MATLAB computations are done internally.

Option	Setting
`format short`	The default setting.
`format long`	Results are displayed to approximately the maximum number of digits of accuracy in MATLAB.
`format short e`	Results are displayed in scientific notation using five significant digits.
`format long e`	Results are displayed in scientific notation to approximately the maximum number of digits of accuracy in MALTAB.
`format short g`	Results are displayed in the best of either `format short` or `format short e`.
`format long g`	Results are displayed in the best of either `format long` or `format long e`.

Table 1.4.1: Options for `format`.

MATLAB documentation by MathWorks on `format` is available at

`https://www.mathworks.com/help/matlab/ref/format.html`

1.5 Script M-Files

Instead of entering MATLAB statements directly into the Command Window and having them be executed immediately, we can type the statements in a separate file named `<file name>.m`. Some of the advantages are the following:

1. The file can be edited to refine the code or to remove any errors.
2. The statements in the file can be executed repeatedly.

Such a file is called a **script m-file** or a **script** for short.

We can work on a new script m-file by clicking on the menu item `NewScript`. If the m-file already exists, we can click on `Open` to search for the file.

Warning: These instructions may differ for different releases of MATLAB.

MATLAB documentation by MathWorks on creating scripts is available at

`https://www.mathworks.com/help/matlab/matlab_prog/create-scripts.html`

Another option is to type

```
edit <filename>
```

where `<filename>` is the a character vector denoting the name of the file. If `<filename>.m` already exists in the search path, its content will be retrieved and shown. Otherwise, a new m-file will be created in the current directory.

Warning: Make sure our filename is not the same as one of MATLAB's commands or functions. If it is, our file might not execute – MATLAB's might! Or, a MATLAB function which has the same name as ours might be run! The m-file being executed depends on the order in which directories are searched for m-files – use the command `path` for more details. This we can check by entering

```
exist <filename>
```

or, more simply,

```
type <file name>
```

before saving the m-file. MATLAB will return an error message if the file does not exist. The command `exist` checks the existence of a variable, a script, a function, a folder, or a class with the specified name, while the command `type` displays the content of a file if available.

MATLAB documentation by MathWorks on `exist` is available at

`https://www.mathworks.com/help/matlab/ref/exist.html`

MATLAB documentation by MathWorks on `type` is available at

`https://www.mathworks.com/help/matlab/ref/type.html`

If the `<filename>.m` is in the search path, then it can be executed by entering

```
<filename>
```

Good practice: Always use descriptive and easy-to-remember variable names. This makes programs clearer and easier to understand.

Good practice: Create a data dictionary*for each program to ease program maintenance.

Example 1.5.1 *This example is taken from [1, Example 2.5, pp. 72–74]. Let Q_0 be the initial amount of a radioactive substance, say carbon-14 to be specific. Its decay is an exponential process in the sense that its amount at time t is given by*

$$Q(t) = Q_0 e^{-\lambda t}$$

for all $t > 0$, where λ is a numerically positive constant, called the decay constant. If we set

$$t_{decay} = \textit{the elapsed time since the decay started},$$

then the equation for $Q(t)$ implies that

$$t_{decay} = -\frac{1}{\lambda} \log_e \frac{Q(t_{decay})}{Q_0},$$

where $Q(t_{decay})$ is the amount of carbon-14 left at the current time. This way t_{deca} gives the age of the sample of carbon-14.

*A data dictionary is a list of definitions of variables used in a program.

This calcaluation can be facilitated using MATLAB. The following script reads the percentage of carbon-14 remaining in a sample, calculates the age of the sample from it, and displays the result with proper units.

```
%  Script file: c14_date.m
%
%  Purpose:
%    To calculate the age of an organic sample from the percentage
%    of the original carbon 14 remaining in the sample.
%
%  Record of revisions:
%      Date       Programmer          Description of change
%      ====       ==========          =====================
%    01/05/14    S. J. Chapman        Original code
%
% Define variables:
%   age        -- The age of the sample in years
%   lamda      -- The radioactive decay constant for carbon-14,
%                 in units of 1/years.
%   percent    -- The percentage of carbon 14 remaining at the time
%                 of the measurement
%   ratio      -- The ratio of the carbon 14 remaining at the time
%                 of the measurement to the original amount of
%                 carbon 14.

% Set decay constant for carbon-14
lamda = 0.00012097;

% Prompt the user for the percentage of C-14 remaining.
percent = input('Enter the percentage of carbon 14 remaining:\n');

% Perform calculations
ratio = percent / 100;              % Convert to fractional ratio
age = (-1.0 / lamda) * log(ratio); % Get age in years

% Tell the user about the age of the sample.
string = ['The age of the sample is ' num2str(age) ' years.'];
disp(string);
```

Suppose the m-file `c14_date.m` *is in the search path. To run it, we type*

```
c14_date
```

A sample run of the m-file is

```
>> c14_date
Enter the percentage of carbon 14 remaining:
```

```
50
The age of the sample is 5729.9097 years.
```

which tells us that if the percentage of the amount of the sample left is found to be 50%*, then the age of the sample is approximately* 5729.9097 *years.*

2 Arrays: Vectors and Matrices

2.1 Generating Matrices

Roughly speaking, an **array** – the fundamental unit of data in a MATLAB program – is a collection of data values organized into rows and columns (or more than two dimensions) and known by a single name. The **size** of an array refers to its numbers of elements in each dimension. **Scalars**, **vectors**, and **matrices** are special types of zero-, one- and two-dimensional arrays, respectively. The **size** of a matrix refers to its numbers of rows and number of columns.

Note: In MATLAB, a scalar is in fact a 1×1 matrix, i.e., a matrix with one row and one column, a row vector is a $1 \times n$ matrix, where n is its number of columns, and a column vector is a $m \times 1$ matrix, where m is its number of rows.

MATLAB documentation by MathWorks on matrices and arrays is available at

`https://www.mathworks.com/help/matlab/matrices-and-arrays.html`

However, an array may have more than two dimensions. So, a scalar, a vector, or a matrix is an array, but an array may not be a matrix!

MATLAB documentation by MathWorks on multidimensional arrays is available at

`https://www.mathworks.com/help/matlab/math/multidimensional-arrays.html`

Either spaces of commas are used to delineate the elements of each row of a matrix; semicolons are used to separate rows.

Warning: Rows can also be separated by beginning each on a separate line. However, this way there is no way to correct an element on a previous line.

Example 2.1.1 *The matrix* $A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$ *can be generated in MATLAB by typing*

```
A = [1 2 3; 4 5 6]
```

or

```
A = [1, 2, 3; 4, 5, 6]
```

or

```
A = [1, 2, 3; 4 5 6]
```

or

```
A = [1, 2, 3; 4 5, 6]
```

or

```
A = [1 2 3
4 5 6]
```

and MATLAB returns

```
A =

     1     2     3
     4     5     6
```

Why do we not prefer to separate rows by beginning each on a separate line?

The size of A can be obtained by entering

```
size(A)
```

and MATLAB returns

```
ans =

     2     3
```

The numbers of rows and columns can be obtained, respectively, by entering

```
size(A,1)
```

and

```
size(A,2)
```

MATLAB returns

```
ans =

     2
```

and

```
ans =

     3
```

respectively.

The total number of elements in a vector or matrix can be obtained by typing

```
numel(A)
```

and MATLAB returns

```
ans =

     6
```

For a vector `x`*, the total number of elements can also be obtained by typing*

```
length(x)
```

Warning: When an element of a matrix consists of more than one terms, the terms must be entered without spaces – unless everything is enclosed in parentheses. Why?

Example 2.1.2 *The inputs*

```
x = [1 pi+3]
```

and

```
x = [1 pi+ 3]
```

and

```
x = [1 (pi +3)]
```

invite the same response from MATLAB. However,

```
y = [1 pi +3]
```

yields a different response. Try them!

The **transpose** of a matrix A, denoted A^T, is obtained by reversing the rows and columns of A. For instance,

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \text{ if and only if } A^T = \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}.$$

Example 2.1.3 *In MATLAB, the transpose of a matrix* `A` *is given by* `A.'` *or* `transpose(A)`*. For instance, the input*

```
A = [1 2 3; 4 5 6]; A.' % This is equivalent to transpose(A).
```

yields the output

```
ans =

     1     4
     2     5
     3     6
```

The **complex conjugate** of a matrix B, denoted B^*, is obtained by taking the complex conjugates of all the elements. For instance,

$$B = \begin{pmatrix} 1+i & 4-3i \\ -i & 4 \end{pmatrix} \text{ if and only if } B^* = \begin{pmatrix} 1-i & 4+3i \\ i & 4 \end{pmatrix}.$$

Example 2.1.4 *In MATLAB, the complex conjugate of a matrix* `B` *is given by* `conj(B)`*. For instance, the input*

```
B = [1+i 4-3i; -i 4]; conj(B)
```

yields the output

```
ans =

   1.0000 - 1.0000i   4.0000 + 3.0000i
        0 + 1.0000i   4.0000
```

The **conjugate transpose** of a matrix B, denotes B^H, is the matrix $(B^*)^T = (B^T)^*$. In words, B^H is the transpose of the complex conjugate of B or the complex conjugate of the transpose of B.

Example 2.1.5 *In MATLAB, the conjugate transpose of a matrix* `B` *is given by* `B'` *or* `ctranspose(B)`*. For instance, the input*

```
B = [1+i 4-3i; -i 4]; B' % This is equivalent to ctranspose(B).
```

yields the output

```
ans =

   1.0000 - 1.0000i        0 + 1.0000i
   4.0000 + 3.0000i   4.0000
```

Warning: `B'` and `B.'` yield different outputs in MATLAB unless (the elements of) the matrix is real.

Example 2.1.6 *We can construct a matrix from smaller matrices so long as the latter matrices have compatible dimensions. For instance, say*

```
x = [1 2 3];
y = [5 6 7];
z = [4; 8];
```

Then observe the following outputs:

```
>> [x y]

ans =

     1     2     3     5     6     7

>> [x; y]

ans =

     1     2     3
     5     6     7

>> [[x; y] z]

ans =

     1     2     3     4
     5     6     7     8

>> [z [x; y]]

ans =

     4     1     2     3
     8     5     6     7
```

Example 2.1.7 *We can extend a matrix by specifying a value for an element beyond the currently defined dimensions. For instance, if* x *has not been defined, then*

```
x(4, 2) = 5
```

creates a matrix of dimensions 4×2 *whose* $(4, 2)$*-entry is* 5*. That is, we get the output*

```
>> x =

     0     0
     0     0
     0     0
     0     5
```

The other previously undefined entries are automatically set to `0`*. If we type*

```
x(4,3) = 2
```

then `x` *is extended to a* 4×3 *matrix:*

```
>> x =

     0     0     0
     0     0     0
     0     0     0
     0     5     2
```

Good practice: It is always safer not to let MATLAB initialize the entries of a matrix automatically unless one is really sure about the value that will be given to these entries.

There is also a handy function called `repmat` that builds a larger matrix from a given one: it repeats copies of an array.

MATLAB documentation by MathWorks on `repmat` is available at

`https://www.mathworks.com/help/matlab/ref/repmat.html`

2.2 Some Special Matrices

Some special matrices can be generated using MATLAB built-in functions.

Function	Generated matrix
`zeros(m,n)`	An $m \times n$ matrix with all elements being 0.
`zeros(size(A))`	A zero matrix with the same size as `A`.
`zeros(n)`	Same as `zeros(n,n)`.
`ones`	A matrix with all elements being 1. The usage and arguments are the same as for `zeros`.
`eye`	A matrix with 1's on the diagonal and 0's elsewhere. The usage and arguments are the same as for `zeros`. Note that `eye(n)` is the $n \times n$ identity matrix.
`diag`	Create diagonal matrix or get diagonal elements of matrix.
`rand`	A matrix whose elements are uniformly distributed (pseudo)random numbers in the interval $(0, 1)$. The usage and arguments are the same as for `zeros`.
`randi([imin,imax],m,n)`	An $m \times n$ matrix whose elements are uniformly distributed (pseudo)random integers drawn from `imin:imax`.
`randi([imin,imax],size(A))`	A matrix with the same size as `A` whose elements are uniformly distributed (pseudo)random integers drawn from `imin:imax`.
`randi([imin,imax],n)`	Same as `randi([imin,imax],n,n)`.

`randi(imax,m,n)`
`randi(imax,size(A))`
`randi(imax,n)`
	Same as `randi([1:imax],m,n)`,
	`randi([1:imax],size(A))`, and
	`randi([1:imax],n)`.

`randn`	A matrix whose elements are normally distributed (pseudo)random numbers with mean 0 and standard deviation 1. The usage and arguments are the same as for `zeros`.

Table 2.2.1: Some special matrices.

MATLAB documentation by MathWorks on `zeros` is available at

https://www.mathworks.com/help/matlab/ref/zeros.html

MATLAB documentation by MathWorks on `ones` is available at

https://www.mathworks.com/help/matlab/ref/ones.html

MATLAB documentation by MathWorks on `eye` is available at

https://www.mathworks.com/help/matlab/ref/eye.html

MATLAB documentation by MathWorks on `diag` is available at

`https://www.mathworks.com/help/matlab/ref/diag.html`

MATLAB documentation by MathWorks on `rand` is available at

`https://www.mathworks.com/help/matlab/ref/rand.html`

MATLAB documentation by MathWorks on `randi` is available at

`https://www.mathworks.com/help/matlab/ref/randi.html`

MATLAB documentation by MathWorks on `randn` is available at

`https://www.mathworks.com/help/matlab/ref/randn.html`

2.3 Displaying Output Data

The `disp` function accepts an array argument and displays the value of the array. It is often used in combination with the functions `num2str` and `int2str`[†].

Note: A character vector is an array of type `char`.

Example 2.3.1 *Typing*

```
disp(['The value of pi = ' num2str(pi)]);
```

at a MATLAB prompt yields

```
The value of pi = 3.1416
```

Example 2.3.2 *The following MATLAB session indicates the usage of the commands `input` and `disp`:*

[†]The function `num2str` converts a number to a character vector, while the function `int2str` converts an integer to a character vector.

```
>> my_name = input('What is your name? ', 's');
What is your name? Kwa Kiam Heong
>> my_age = input('How old are you? (in years) ');
How old are you? (in years) 45
>> disp(['You are ' my_name ' and you are ' num2str(my_age)
' years old.']);
You are Kwa Kiam Heong and you are 45 years old.
```

Example 2.3.3 *Let us request two* 3×3 *matrices using the command* `input`*:*

```
>> A = input('Enter a 3x3 matrix: ');
Enter a 3x3 matrix: -2*eye(3)
>> B = input('Enter another 3x3 matrix: ');
Enter another 3x3 matrix: 5*ones(3)
```

Then typing

```
disp('The sum of the matrices'); ...
disp(A); ...
disp('and'); ...
disp(B); ...
disp('are'); ...
disp(A+B);
```

yields

```
The sum of the matrices
    -2     0     0
     0    -2     0
     0     0    -2

and
     5     5     5
     5     5     5
     5     5     5
```

```
are
     3     5     5
     5     3     5
     5     5     3
```

Note: If a statement is too long to type in a single line, it can be continued on successive lines by typing ellipsis (...). For instance, instead of typing

```
A + B
```

we can type

```
A + ...
B
```

to achieve the same effect.

MATLAB documentation by MathWorks on `disp` is available at

`https://www.mathworks.com/help/matlab/ref/disp.html`

MATLAB documentation by MathWorks on the ellipsis (...) is available at

`https://www.mathworks.com/help/matlab/matlab_prog/matlab-operators-and-special-characters.html`

The `fprintf` function displays one or more values together with related text. Its general form of usage is

```
fprintf(format, data)
```

where `format` is a string describing the way `data` is to be displayed, while `data` is one or more scalars or arrays to be displayed.

Example 2.3.4 *The statement*

```
fprintf('The value of pi is %f.\n', pi);
```

displays

```
The value of pi is 3.141593.
```

The conversion characters `%f` *indicate a value in the data is to be displayed in floating point format at the location in the format string. The* escape characters `\n` *indicate that a line feed is to be issued so that the following text starts on a new line. Conversion characters and escape characters are collectively called* **special characters** *in* `fprintf` *format strings. See Table 2.3.1.*

It is possible to specify the width of the field in which a number is to be displayed. For instance,

```
fprintf('The value of pi is %6.2f.\n', pi);
```

displays

```
The value of pi is   3.14.
```

The conversion characters `%6.2f` *indicate that the first data item is to be displayed in floating point format in field of six (6) characters wide, including two (2) digits after the decimal point.*

Format string	Results
`%d`	Display value as an integer.
`%e`	Display value in exponential format.
`%f`	Display value in floating point format.
`%g`	Display value in either floating point or exponential format whichever is shorter.
`%s`	Display character vector or string array.
`\n`	Skip to a new line.

Table 2.3.1: Common special characters in `fprintf` format strings.

MATLAB documentation by MathWorks on `fprintf` is available at

`https://www.mathworks.com/help/matlab/ref/fprintf.html`

2.4 Common Built-in Functions

MATLAB contains a large number of mathematical functions. Tables 2.4.1 and 2.4.2 list some common numerical functions in terms of scalar inputs. These functions can be applied to arrays, whose outputs are then arrays with elements obtained from applying the functions elementwise to the input arrays.

Note: These functions usually have more functionalities than are listed here. Use the commands `help` and `doc` to obtain more details.

Function	Description
`abs(x)`	$\lvert x\rvert$.

`acos(x)`
`asin(x)`
`atan(x)`
	$\cos^{-1}(x)$, $\sin^{-1}(x)$, and $\tan^{-1}(x)$, respectively, in radians.

`cos(x)`
`sin(x)`
`tan(x)`
	$\cos(x)$, $\sin(x)$, and $\tan(x)$, respectively, where x is in radians.

`exp(x)`	e^x.
`log(x)`	$\log_e(x)$.
`mod(x,y)`	The "remainder" of division of x by y.
`sqrt(x)`	$\sqrt{x}$.

Table 2.4.1: Some common mathematical functions. Here x and y are scalars.

Function	Description
`ceil(x)` `floor(x)`	Rounds `x` to the nearest integer towards $+\infty$ and $-\infty$, respectively.
`fix(x)`	Rounds `x` to the nearest integer towards 0.
`round(x)`	Rounds `x` to the nearest integer.

Table 2.4.2: Some common rounding functions. Here x is a scalar.

Table 2.4.3 lists some common data manipulation commands.

Function	Description
`cumprod(x)` `cumsum(x)`	The cumulative product/sum of the elements of `x`.
`cumprod(A)` `cumsum(A)`	A matrix the same size as `A` containing the cumulative products/sums over each column.
`max(x)` `min(x)`	The maximum/minimum element of a real vector `x`.
`max(A)` `min(A)`	A row vector containing the maximum/minimum element from each column of a real matrix `A`.
`mean(x)` `std(x)`	The mean value/standard deviation of the elements in `x`.
`mean(A)` `std(A)`	A row vector containing the mean value/standard deviation of each column of `A`.
`prod(x)` `sum(x)`	The product/sum of the elements of `x`.
`prod(A)` `sum(A)`	A row vector with the product/sum over each column of `A`.

Table 2.4.3: Some data manipulation commands. Here `x` is a vector and `A` and `B` are matrices.

MATLAB documentation by MathWorks on various functions, which may or may not be mathematical, is available at

`https://www.mathworks.com/help/matlab/functionlist.html`

2.5 The Colon Operator and the Function `linspace`

For real numbers a and b, the MATLAB statement

```
[a:b]
```

or, simply,

```
a:b
```

generates the row vector $\begin{pmatrix} a & a+1 & \cdots & a+k \end{pmatrix}$, where k is the largest integer such that $a + k \leq b$. If c is also a real number, then

```
[a:c:b]
```

or

```
a:c:b
```

yields a row vector where the difference between successive elements is c.

Example 2.5.1 *The MATLAB statement*

```
5:15
```

returns

```
ans =

     5     6     7     8     9    10    11    12    13    14    15
```

The MATLAB statement

```
5:3:15
```

returns

```
ans =

     5     8    11    14
```

Also,

```
5:3.5:15
```

yields

```
ans =

    5.0000    8.5000   12.0000
```

The MATLAB statement

```
5:20:15
```

yields

```
ans =

    5
```

Example 2.5.2 *The MATLAB statement*

```
18:-2:11
```

yields

```
ans =

   18   16   14   12
```

The MATLAB statement

```
pi:-0.5:1
```

yields

```
ans =

    3.1416    2.6416    2.1416    1.6416    1.1416
```

An **empty matrix** is a matrix having at least one dimension equal to zero.

Example 2.5.3 *The MATLAB statement*

```
 10:1
```

returns

```
ans =

    1×0 empty double row vector
```

Checking the size

```
size(10:1)
```

yields

```
ans =

   1   0
```

In MATLAB, the statement `isempty(A)` *returns* `true (1)` *if* `A` *is an empty array. Otherwise, it returns* `false (0)`[‡]*. For instance, the input*

```
isempty(10:1)
```

returns

```
ans =

    1
```

while the input

```
isempty(1:10)
```

returns

```
ans =

    0
```

[‡] `true` and `false` are the two possible values of logical data type, the latter of which will be discussed in later section.

MATLAB documentation by MathWorks on the colon (`:`) operator is available at

`https://www.mathworks.com/help/matlab/ref/colon.html`

Example 2.5.4 *The* `linspace` *function generates linearly spaced vectors. It is similar to the colon operator (*`:`*), but gives direct control over the number of points. The general syntax is*

```
linspace(a, b, n)
```

which generates `n` *number of equally spaced points between* `a` *and* `b` *inclusively. If* `n` *is not specified, then it is assumed to be* `100`*. For instance,*

```
linspace(5, 15, 11)
```

yields

```
ans =

  Columns 1 through 10

     5     6     7     8     9    10    11    12    13    14

  Column 11

    15
```

Effectively, this is the same to `5:15`*.*

MATLAB documentation by MathWorks on `linspace` is available at

`https://www.mathworks.com/help/matlab/ref/linspace.html`

2.6 Array and Matrix Operations

An **array operation** is also referred to as an **elementwise operation**. Such an operation is performed between arrays on an *element-by-element basis*. That is, the operation is carried out on corresponding elements in the arrays. In contrast, a *matrix operation* follows the normal rule of linear algebra.

Some elementwise operations and matrix operations are identical, such as addition and subtraction. MATLAB uses a **period** (`.`) before the symbol to indicate an array operation.

MATLAB offers the array and matrix operations listed in Table 2.6.1.

Syntax	Operation
`A+B`	Matrix addition.
`A-B`	Matrix subtraction.
`A*B`	Matrix multiplication.
`A^p`	Matrix exponentiation.
`A\b`	Matrix left division. The solution `x` of `Ax=b` by Gaussian elimination when `A` is a nonsingular square matrix.
`A\B`	Matrix left division. The solution `X` of `AX=B` by Gaussian elimination.
`b/A`	Matrix right division. The solution `x` of `xA=b` where `x` and `b` are row vectors.
`B/A`	Matrix right division. The solution `X` of `XA=B` by Gaussian elimination.
`A.*B`	Elementwise multiplication. The product is knwon as the **Hadamard product** or the **Schur product** in mathematics, and is customarily denoted either by $A \odot B$ or by $A \circ B$.

`A.^p`
`p.^A`
`A.^B`
	Elementwise exponentiation.

`A./B`	Elementwise right division.
`B.\A`	Elementwise left division. Same as `A./B`

Table 2.6.1: Here `A` and `B` are matrices, `b` is a vector, and `p` is a scalar.

MATLAB documentation by MathWorks on basic matrix operations is available at

`https://www.mathworks.com/help/matlab/examples/basic-matrix-operations.html`

A comparison of matrix operations and array operations is documented by MathWorks at

`https://www.mathworks.com/help/matlab/matlab_prog/array-vs-matrix-operations.html`

MATLAB follows a series of rules, as in Table 2.6.2, governing the order of array and matrix operations.

Precedence	**Operation**
1	The contents of all parentheses (`(...)`) are evaluated, starting from the innermost parentheses and working outword.
2	All exponentials are evaluated, working from left to right.
3	All multiplications and divisions are evaluated, working from left to right.
4	All additions and subtractions are evaluated, working from left to right.

Table 2.6.2: Hierarchy of elementwise and matrix operations.

Example 2.6.1 *Suppose*

$$A = B = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \text{ and } C = \begin{pmatrix} 3 & 2 & 2 \\ 2 & 3 & 2 \\ 2 & 2 & 3 \end{pmatrix}.$$

which we can define in MATLAB by typing

```
A = [1:3; 4:6; 7:9]; B = A;
C = eye(3) + 2*ones(3);
% This is equivalent to C = eye(3) + repmat(2, 3, 3).
```

Then typing

```
A * B
```

yields the matrix product

```
ans =

    30    36    42
    66    81    96
   102   126   150
```

while typing

```
 A.*B
```

yields the elementwise product

```
ans =

     1     4     9
    16    25    36
    49    64    81
```

Also,

```
A/B
```

means multiply `A` *by the inverse of* `B`*, which yields*

```
ans =

     1     0     0
     0     1     0
     0     0     1
```

(with a warning), while

```
A./B
```

means elementwise division, which returns

```
ans =

     1     1     1
     1     1     1
     1     1     1
```

On the other hand,

```
A.^C
```

is the elementwise exponentiation, which gives

```
ans =

     1     4     9
    16   125    36
    49    64   729
```

It is also worthwhile to note the difference between

```
A^2
```

which yields the matrix exponentiation `A*A`*, i.e.,*

```
ans =

    30    36    42
    66    81    96
   102   126   150
```

and

```
A.^2
```

which is an elementwise exponentiation, which yields

```
ans =

     1     4     9
    16    25    36
    49    64    81
```

Compare this also to

```
2.^A
```

which returns

```
ans =

     2     4     8
    16    32    64
   128   256   512
```

Finally,

```
A+2
```

returns

```
ans =

     3     6     9
     4     7    10
     5     8    11
```

2.7 Manipulating Matrices

The matrix

$$E = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 & 17 & 18 \\ 19 & 20 & 21 & 22 & 23 & 24 \\ 25 & 26 & 27 & 28 & 29 & 30 \end{pmatrix}$$

can be generated in MALTAB by

```
E = [1:6; 7:12; 13:18; 19:24; 25:30]
```

or by

```
E = reshape(1:30, 5, 6).'
```

The (i, j)-entries of `E` can be referred to by typing `E(i,j)`.

Example 2.7.1 *Typing*

```
E = [1:6; 7:12; 13:18; 19:24; 25:30]; E(5,6)
```

yields

```
ans =

    30
```

which is the $(5,6)$*-entry of the matrix* `E`*. Typing*

```
E(4,3) = 0
```

assigns the $(4,3)$*-entry of* `E` *to* 0*. The output is*

```
E =

     1     2     3     4     5     6
     7     8     9    10    11    12
    13    14    15    16    17    18
    19    20     0    22    23    24
    25    26    27    28    29    30
```

The same entry of `E` *can be re-assigned to* 21 *by typing*

```
E(4,3) = 21
```

which yields

```
E =

     1     2     3     4     5     6
     7     8     9    10    11    12
    13    14    15    16    17    18
    19    20    21    22    23    24
    25    26    27    28    29    30
```

The submatrices of the matrix E can be referred to by using lists of indices.

Example 2.7.2 *Typing*

```
E = [1:6; 7:12; 13:18; 19:24; 25:30]; E([1 3 5],[2 4 6])
```

yields the submatrix

```
ans =

     2     4     6
    14    16    18
    26    28    30
```

of E*. This can also be done by typing*

```
E(1:2:5, 2:2:6)
```

A submatrix can be re-assigned as another matrix of the same size. Typing

```
E(1:2:5, 2:2:6) = ones(size(E(1:2:5, 2:2:6)))
```

yields

```
E =

     1     1     3     1     5     1
     7     8     9    10    11    12
    13     1    15     1    17     1
    19    20    21    22    23    24
    25     1    27     1    29     1
```

Typing

```
E([1 2], [1 2]) = -3 * ones(2)
```

yields

```
E =

    -3    -3     3     1     5     1
    -3    -3     9    10    11    12
    13     1    15     1    17     1
    19    20    21    22    23    24
    25     1    27     1    29     1
```

MATLAB supports **scalar expansion**; *that is, a scalar value on the right-hand side of an assignment statements always matches the shape specified on the left-hand side. For instance, typing*

```
E([3 4], [3 4]) = -4
```

is the same to typing `E([3 4], [3 4])= -4 * ones(2)`, *yielding*

```
E =

    -3    -3     3     1     5     1
    -3    -3     9    10    11    12
    13     1    -4    -4    17     1
    19    20    -4    -4    23    24
    25     1    27     1    29     1
```

More interestingly, typing

```
E = E + 3
```

gives

```
E =

     0     0     6     4     8     4
     0     0    12    13    14    15
    16     4    -1    -1    20     4
    22    23    -1    -1    26    27
    28     4    30     4    32     4
```

See this post by Loren Shure on scalar expansion:

`https://blogs.mathworks.com/loren/2006/02/22/scalar-expansion-and-more-take-2/`

Using a *colon (:)* by itself as an index, the colon represents an entire row or column.

Example 2.7.3 *Typing*

```
E = [1:6; 7:12; 13:18; 19:24; 25:30]; E(:,3)
```

yields the third column of `E`*, i.e.,*

```
ans =

     3
     9
    15
    21
    27
```

Typing

```
E(5,:)
```

yields the fifth row of `E`*, i.e.,*

```
ans =

    25    26    27    28    29    30
```

An empty matrix *of dimensions* 0×0 *can be generated in MATLAB as* `[]`*. Combining this with the colon* `(:)`*, this allows us to remove a row or a column from a matrix. For instance,*

```
E(2,:) = []
```

removes the second row of `E`*, yielding*

```
E =

     1     2     3     4     5     6
    13    14    15    16    17    18
    19    20    21    22    23    24
    25    26    27    28    29    30
```

Also,

```
E(:,3) = []
```

removes the third column of `E`*, yielding*

```
E =

     1     2     4     5     6
    13    14    16    17    18
    19    20    22    23    24
    25    26    28    29    30
```

We can also type

```
E(:,[2 4]) = []
```

to remove some selected (in this case, the second and the fourth) columns, yielding

```
E =

     1     4     6
    13    16    18
    19    22    24
    25    28    30
```

The same technique can be used to remove selected rows.

Example 2.7.4 *The* `end` function*, when being used as an index, returns the highest value taken on by that index. Typing*

```
E = [1:6; 7:12; 13:18; 19:24; 25:30]; E(:,5:end)
```

yields

```
ans =

     5     6
    11    12
    17    18
    23    24
    29    30
```

the last two columns of `E`*.*

Incidentally, the matrix `E` *can be generated also by typing* `E(:,:)` *or* `E(:,1:end)` *or* `E(1:end,:)` *or* `E(1:end,1:end)`*.*

Example 2.7.5 *Entries of a matrix is stored in memory by MATLAB in* column major order. *Suppose we have typed in*

```
F = [1:4; 5:8; 9:12].' % Equivalently, F = reshape(1:12, 4, 3).
```

yielding

```
F =

     1     5     9
     2     6    10
     3     7    11
     4     8    12
```

Then the $(3, 2)$*-entry of* `F`*, which is* `7`*, can be obtained by typing* `F(3,2)` *or, alternatively, by typing*

```
F(7)
```

because it is the seventh element of `F` *according to the column major order. The fourth to the ninth elements of* `F` *are given by*

```
F(4:9)
```

yielding

```
ans =

     4     5     6     7     8     9
```

This way of referring to the entries of `F` *is called* **linear indexing**.

To convert from subscripts to the corresponding linear indices and vice versa, use the functions `sub2ind` and `ind2sub`. For instance,

```
sub2ind(size(F),3,2)
```

yields

```
ans =

     7
```

while

```
[m, n]=ind2sub(size(F), 7)
```

yields

```
m =

     3


n =

     2
```

MATLAB documentation by MathWorks on `sub2ind` is available at

`https://www.mathworks.com/help/matlab/ref/sub2ind.html`

MATLAB documentation by MathWorks on `ind2sub` is available at

`https://www.mathworks.com/help/matlab/ref/ind2sub.html`

2.8 Logical Indexing

In **logical indexing**, we use a single logical array to extract the elements of a matrix corresponding to the nonzero values of the logical array.

Note: It can be helpful to take a glance at Section 5.1 at this point.

Example 2.8.1 *Suppose*

```
>> A = magic(4)

A =

    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

has already been executed. The function `magic(n)` *yields a magic square of order* n[§]*. Then typing*

```
A(A > 8)
```

extracts all elements of `A` *which are greater than* 8*. That is, MATLAB responses with*

```
ans =

    16
     9
    11
    14
    10
    15
    13
    12
```

[§]A magic square of order n is an $n \times n$ matrix whose elements are integers 1 through n^2 with equal row, column, and diagonal sums.

Note that

```
A > 8
```

yields a matrix of the same size as `A` *whose elements are* `0` *(*`false`*) and* `1` *(*`true`*), i.e.,*

```
ans =

     1     0     0     1
     0     1     1     0
     1     0     0     1
     0     1     1     0
```

The elements getting the value `1` *are precisely those corresponding to elements of* `A` *which are greater than* 8*.*

To obtain a matrix of the same size as `A`*, say* `B`*, whose elements are the same to those of* `A` *if the elements are less than or equal to* 8*, otherwise the negative of the corresponding elements, we type*

```
B = zeros(size(A));    % Initialize B to be the same size as A
B(A <= 8) = A(A <= 8); % Corresponding elements of A and B are equal
                       % if the elements <= 8
B(A > 8) = -A(A > 8)   % Corresponding elements of A and B are negative
                       % of each other if the elements > 8
```

The result of which is

```
B =

   -16     2     3   -13
     5   -11   -10     8
    -9     7     6   -12
     4   -14   -15     1
```

More compactly, we can type

```
B = A;                % Assign A to B
B(B > 8) = -B(B > 8) % Set elements of B > 8 to their negatives
```

to get the same result.

To obtain a matrix of the same size as `A`*, say* `C`*, whose elements are the same to those of* `A` *if the elements are less than or equal to* 8*, otherwise the elements take the value* 8*, we type*

```
C = 8 * ones(size(A)); % Initialize C to be a matrix with all
                       % elements being 8
C(A <= 8) = A(A <= 8)  % Assign elements of A <= 8 to corresponding
                       % elements of C
```

The result is

```
C =

     8     2     3     8
     5     8     8     8
     8     7     6     8
     4     8     8     1
```

Example 2.8.2 *Suppose*

$$f(x) = \begin{cases} x & \text{if } x < 0; \\ x^2 & \text{if } 0 \le x < 2; \\ 8 - x^2 & \text{if } x \ge 2. \end{cases}$$

To assign a vector `y` *so that it takes on the values of* $f(x)$ *for* $-4 \le x \le 4$*, we type*

```
x = -4:.01:4;
y = zeros(size(x));
y(x < 0) = x(x < 0);
y((x >= 0)&(x < 2)) = x((x >= 0)&(x < 2)).^2;
y(x >= 2) = 8 - x(x >= 2).^2;
plot(x, y);
```

The command `plot(x, y)` *plots the graph of* `x` *versus* `y`*. More on plotting will be discussed in Section 4.*

More compactly, we can type

```
x = -4:.01:4;
y = x.^2;
y(x < 0) = x(x < 0);
y(x >= 2) = 8 - x(x >= 2).^2;
plot(x, y);
```

to achieve the same result.

MATLAB documentation by MathWorks on array indexing is available at

`https://www.mathworks.com/help/matlab/math/array-indexing.html`

2.9 Solving Linear Systems of Equations using \, `linsolve`, and `rref`

One of the basic uses of MATLAB is to solve the linear system

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1, \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2, \\ &\vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &= b_m. \end{aligned}$$

In matrix notation, this reads

$$Ax = b,$$

where

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}, \quad x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}, \text{ and } b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}.$$

When $m = n$, the solution `x` can be obtained in MATLAB by typing

```
x = A\b     % This is the same to mldivide(A, B)
```

or

```
x = linsolve(A, b)
```

Use `help <command>` to retrieve more information.

MATLAB documentation by MathWorks on `linsolve` is available at

`https://www.mathworks.com/help/matlab/ref/linsolve.html`

To solve the system using the function `rref`, consider the augmented matrix of the system, i.e.,

$$(A \mid b) = \left(\begin{array}{cccc|c} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ a_{21} & a_{22} & \cdots & a_{2n} & b_2 \\ \vdots & \vdots & & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} & b_n \end{array}\right).$$

This matrix can be reduced to the so-called reduced row echelon form using the following **elementary row operations**:

- **Row switching.** Any two rows can be switched.
- **Row multiplication.** Each row can be multiplied by a nonzero constant.
- **Row addition.** Each row can be replaced by the sum of that row and a scalar multiple of another row.

Suppose the augmented matrix is entered as `E` in MATLAB. The elementary row operations can be done in MALTAB as follows.

- **Switching the ith and the jth rows.**

```
E([j i],:) = E([i j],:)
```

- **Multiplying the ith row by a nonzero scalar c.**

```
E(i,:) = c*E(i,:)
```

- **Adding c times the jth row to the ith row, where c is a scalar and $i \neq j$.**

```
E(i,:) = E(i,:)+c*E(j,:)
```

Recall that a matrix is said to be in *reduced row echelon form* if

1. The first nonzero entry in each row is a 1, and
2. the first nonzero entry in a particular row occurs later than in all the previous rows.

The function `rref` yields the reduced row echelon form of a given matrix.

Example 2.9.1 *Suppose we want to solve*

$$\begin{aligned} x_1 + 2x_2 + 3x_3 &= -1, \\ 4x_1 + 5x_2 + 6x_3 &= -1, \\ 7x_1 + 8x_2 + 10x_3 &= 0. \end{aligned}$$

We first input the coefficient matrix and the non-homogeneous term:

```
A = [1:3; 4:6; 7 8 10]; b = [-1 -1 0]';
```

Note that `A` *is nonsingular, which can be found out by computing the determinant*

```
det(A)
```

Then the solution of the system is obtained by typing

```
A\b
```

or

```
linsolve(A, b)
```

MATLAB responds with

```
ans =

    2.0000
   -3.0000
    1.0000
```

That is, $x_1 = 2$, $x_2 = -3$, *and* $x_3 = 1$.

To solve the system using `rref`, *set up the augmented matrix by typing*

```
E = [A b]
```

Then the reduced echelon form of `E` *is obtained by typing*

```
rref(E)
```

the result of which is

```
ans =

     1     0     0     2
     0     1     0    -3
     0     0     1     1
```

This implies that

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

or, simply,

$$x_1 = 2,\ x_2 = -3,\ \textit{and}\ x_3 = 1.$$

MATLAB documentation by MathWorks on `rref` is available at

`https://www.mathworks.com/help/matlab/ref/rref.html`

Example 2.9.2 *Suppose we want to solve*

$$\begin{aligned} x_1 + 2x_2 + 3x_3 &= -1, \\ 4x_1 + 5x_2 + 6x_3 &= -1, \\ 7x_1 + 8x_2 + 9x_3 &= -1. \end{aligned}$$

We first input the coefficient matrix and the non-homogeneous term:

```
A = [1:3; 4:6; 7:9]; b = [-1; -1; -1];
```

It is easy to check that `A` *is in fact singular. Calculating the determinant of* A *by hand yields* $\det(A) = 0$. *However, typing*

```
det(A)
```

at a MATLAB prompt yields

```
ans =

  6.6613e-016
```

Warning: *If not being used with caution, MATLAB does not always give correct results!*

Trying to "solve" the system by typing either `A\b` *or* `linsolve(A,b)` *yields*

```
Warning: Matrix is close to singular or badly
scaled. Results may be inaccurate. RCOND =
1.541976e-018.

ans =

    2.5000
   -4.0000
    1.5000
```

This is only one of the many solutions of the system!

Warning: *The command* `linsolve` *does not always give correct results. Use it with caution!*

To solve the system using `rref`*, set up the augmented matrix by typing*

```
E = [A b]
```

Then the reduced echelon form of `E` *is obtained by typing*

```
rref(E)
```

the result of which is

```
ans =

     1     0    -1     1
     0     1     2    -1
     0     0     0     0
```

This implies that

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

or, simply,

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

This indicates that there are infinitely many solutions of the form

$$x_1 = 1 + c,\ x_2 = -1 - 2c,\ \textit{and}\ x_3 = c,$$

where $c \in \mathbb{C}$.

3 User-Defined Functions

3.1 Anonymous Functions and Function Handles

If a mathematical function is particularly simple, say

$$f(\langle arg1\rangle, \langle arg2\rangle, \cdots) = \langle expression\rangle,$$

then we can define it in MATLAB by typing

```
f = @(<arg1>, <arg2>, ...) <expression>
```

where

```
@(<argument list>) <expression>
```

is the general syntax for defining an **anonymous function**. For instance, we can define the function $f(t) = t^2 e^{-2t}\sin(t^2)$ by typing

```
f = @(t) t.^2.*exp(-2*t).*sin(t.^2);
```

Then we can evaluate f and plot the graph as follows:

```
x = 0:.01:1;
plot(x, f(x));
```

More generally, we can define $g(x, y, z, a) = x^a y^3 \cos(z^4)$ by

```
g = @(x,y,z,a) x.^a.*y.^3.*cos(z.^4);
```

Then we can evaluate g as follows:

```
A = rand(3); X = rand(3); Y = rand(3); Z = rand(3);
g(X, Y, Z, A)
```

The at sign (`@`) is the MATLAB operator that constructs a function handle. A **handle** is used in MATLAB to denote a variable which refers to some "object" that has been created. A **function handle** creates an association between a variable and a function. In the above examples, `f` and `g` are function handles. Suppose `f` has already been defined in MATLAB as above. Then the assignment

```
h = f;
```

makes `h` a function handle referring to the same function as `f`. This way

```
f(1:10)
```

and

```
h(1:10)
```

give the same results.

We can create a function handle to a built-in function by using `@`. For instance, the assignment

```
s = @sin;
```

makes `s(0:.1:pi)` identical to `sin(0:.1:pi)`.

MATLAB documentation by MathWorks on anonymous functions is available at

`https://www.mathworks.com/help/matlab/matlab_prog/anonymous-functions.html`

MATLAB documentation by MathWorks on function handles is available at

`https://www.mathworks.com/help/matlab/function-handles.html`

Warning: When a function is created as an anonymous function such as

```
f = @(t) t.*t;
```

the identifier `f` is already a function handle. However, this is not so for functions created otherwise. For instance, the function handle to the built-in function `sin` is `@sin`, not `sin`. Likewise, if `g` is a function defined in a function m-file, then the function handle is `@g`, not `g`.

3.2 Function M-files

To define a more complicated function, we can use a function m-file. It may be good to begin with an example.

Example 3.2.1 *Consider the following m-file [1, pp. 270-271]. It must be named* `dist2.m` *for reasons which will be clarified.*

```
function distance = dist2(x1, y1, x2, y2)
% DIST2 Calculate the distance between two points.
% Function DIST2 calculates the distance between
% two points (x1,y1) and (x2,y2) in a Cartesian
% coordinate system.
%
% Calling sequence:
%    distance = dist2(x1, y1, x2, y2)

% Define variables:
%   x1       -- x-position of point 1
%   y1       -- y-position of point 1
%   x2       -- x-position of point 2
%   y2       -- y-position of point 2
%   distance -- Distance between points
```

```
%  Record of revisions:
%      Date       Programmer          Description of change
%      ====       ==========          =====================
%    02/01/10    S. J. Chapman        Original code

% Calculate distance.
distance = sqrt((x2-x1).^2 + (y2-y1).^2);

end
```

It is an example of a **function m-file**. *It begins with the* `function` *statement which specifies*

1. *the name of the function '*`dist2`*',*
2. *the input argument lists '*`x1, y1, x2, y2`*', and*
3. *the output argument lists '*`distance`*'.*

The **input argument list** *contains a list of variable names representing values which will be passed to the function from the caller. These are called* **dummy arguments** *because they are just placeholders for actual values that are passed from the caller when the function is invoked. On the other hand, the* **output argument list** *contains a list of dummy arguments which are placeholders for the values returned to the caller when the function finishes its execution.*

Assuming the m-file is in the search path, then we can use the function as any other MATLAB built-in functions. For instance, to calculate the distance from the point $(2, 3)$ *to the point* $(4, 5)$*, we type*

```
dist2(2, 3, 4, 5)
```

and MATLAB responses with

```
ans =

    2.8284
```

The **H1 comment line** *is the first comment line after the function statement. It should contains a one-line summary of the purpose of the function. It is searched and displayed by the command* `lookfor`*:*

```
>> lookfor distance
dist2        - Calculate the distance between two points.
turningdist  - Find the turning distance of two polyshapes
distances    - Compute shortest path distances between node pairs in a graph
distances    - Compute shortest path distances between node pairs in a digraph
distfcm      - Distance measure in fuzzy c-mean clustering.
mahal        - Mahalanobis distance.
...
```

Many commands and functions with the keyword "distance" in their H1 comment lines have been suppressed here. Also, the list may differ for different releases of MATLAB.

The remaining comment lines from the H1 line until the first blank line or the first executable statement are displayed by the command `help`*:*

```
>> help dist2
  dist2 Calculate the distance between two points
  Function dist2 calculates the distance between
  two points (x1,y1) and (x2,y2) in a Cartesian
  coordinate system.

  Calling sequence:
     distance = dist2(x1, y1, x2, y2)
```

MATLAB documentation by MathWorks on adding help and comments to a user-defined function is available at

`https://www.mathworks.com/help/matlab/matlab_prog/add-help-for-your-program.html`

Example 3.2.2 *A function m-file can be invoked within any script m-file or function m-file. The following script m-file does so for the function m-file* `dist2.m` *from the previous example [1, pp. 271].*

```
%  Script file: test_dist2.m
%
%  Purpose:
```

```
%    This program tests function dist2.
%
%  Record of revisions:
%      Date       Programmer          Description of change
%      ====       ==========          =====================
%    02/01/10    S. J. Chapman        Original code
%
% Define variables:
%   ax      -- x-position of point a
%   ay      -- y-position of point a
%   bx      -- x-position of point b
%   by      -- y-position of point b
%   result -- Distance between the points

% Get input data
disp('Calculate the distance between two points:');
ax = input('Enter x value of point a:   ');
ay = input('Enter y value of point a:   ');
bx = input('Enter x value of point b:   ');
by = input('Enter y value of point b:   ');

% Evaluate function
result = dist2 (ax, ay, bx, by);

% Write out result
fprintf('The distance between points a and b is %f\n',result);
```

A session of running the script m-file is shown below:

```
>> test_dist2
Calculate the distance between two points:
Enter x value of point a:   2
Enter y value of point a:   3
Enter x value of point b:   5
Enter y value of point b:   7
The distance between points a and b is 5.000000
```

Note that the variables `x1`, `y1`, `x2`, `y2`, *and* `distance` *are* **local** *to the function* `dist2` *in the sense that their values are visible only within the function's workspace. Trying to access them after a call to the function* `dist2` *is completed will either yield an error message provided they are not defined in MATLAB's workspace, i.e.,*

```
>> x1
Undefined function or variable 'x1'.
```

or their values as defined in the workspace, say

```
>> x1 = 10;
>> dist2(1,2,3,4)

ans =

    2.8284

>> x1

x1 =

    10
```

Example 3.2.3 *MATLAB and MATLAB m-files communicate with functions using a* **pass-by-value scheme**. *This means that when a function calls occurs, a copy of the actual arguments are passed to the function. This way even if the function modifies the input arguments, it will not affect the original data in the caller. For instance, say we have the function m-file*

```
function out = sample(a, b, c)
%
fprintf('In     sample: a = %f, b = %f %f\n',a,b);
a = b(1) + 2*a;
b = a .* b;
out = a + b(1);
fprintf('In     sample: a = %f, b = %f %f\n',a,b);

end
```

To test the values of the variables `a`, `b`, *and* `c` *within the function* `sample` *and within the MATLAB's workspace, we run the following script m-file:*

```
a = 2; b = [6 4];
fprintf('Before sample: a = %f, b = %f %f\n',a,b);
out = sample(a,b);
fprintf('After  sample: a = %f, b = %f %f\n',a,b);
fprintf('After  sample: out = %f\n',out);
```

A session of running the above script m-file yields

```
>> test_sample
Before sample: a = 2.000000, b = 6.000000 4.000000
In     sample: a = 2.000000, b = 6.000000 4.000000
In     sample: a = 10.000000, b = 60.000000 40.000000
After  sample: a = 2.000000, b = 6.000000 4.000000
After  sample: out = 70.000000
```

The variables `a` *and* `b` *are both changed inside the function* `sample`*, but those changes are not visible in the calling m-file.*

The following documents provide more info on the syntax to create functions with more than one input arguments and/or more than one output arguments.

MATLAB documentation by MathWorks on creating functions in files is available at

https://www.mathworks.com/help/matlab/matlab_prog/create-functions-in-files.html

MATLAB documentation by MathWorks on declaring functions is available at

https://www.mathworks.com/help/matlab/ref/function.html

3.3 Side Effects

In addition to performing operations on inputs and to produce outputs, a function can also have **side effects**. That is, a function can do something beyond assigning values to its outputs. For instance, the built-in function `disp` takes a string as an input and produces no output, but has the side-effect of making the string appear in Command Window.

Example 3.3.1 *The following is a function with no output, but has the side effect of greeting the person whose name is given by the input argument.*

```
function greet(n)
% Greet the person whose name is given by the input argument.
%
% Syntax
%   greet(n)
%
```

```
% Description
%   greet(n) displays a greeting message to the person whose name ...
    is n.

if not(isempty(n))
    fprintf('Hello, %s!\n', n);
else
    fprintf('You are?\n');
end

end
```

Some sample runs are as follows:

```
>> greet('Kwa')
Hello, Kwa!
>> greet('')
You are?
```

Example 3.3.2 *The following is also a function with no output, but has the side effect of displaying a specified number of asterisks in Command Window.*

```
function star(n)
% Display a number of asterisks (stars) in a single row in Command ...
    Window
%
% Syntax
%   star(n)
%
% Description
%   star(n) displays n asterisks (stars) in a single row in Command ...
    Window.

if n < 1
    fprintf('Incomprehensible number of stars.');
end

for i = 1:n
    fprintf('*');
end
fprintf('\n');

end
```

Some sample runs are as follows:

```
>> star(10)
```

```
**********
>> star(20)
********************
>> star(-1)
Incomprehensible number of stars.
```

4 Introduction to Plotting

MATLAB has has extensive device-independent plotting capabilities.

4.1 Simple xy-Plot

The command `plot` plots data points in Cartesian coordinates.

Example 4.1.1 *To plot the function $y = cos(x)$ for values of x between -2π to 2π, type*

```
x = -2*pi:0.1:2*pi;
y = cos(x);
plot(x,y);
```

Title and axis labels can be added by typing

```
title('Plot of y = cos(x)');
xlabel('x');
ylabel('y');
```

Grid lines are added by typing

```
grid on;
```

The resulting plot is shown in Figure 4.1.1.

We can clear the current figure by typing

```
clf
```

We can close the current figure window by typing

```
close
```

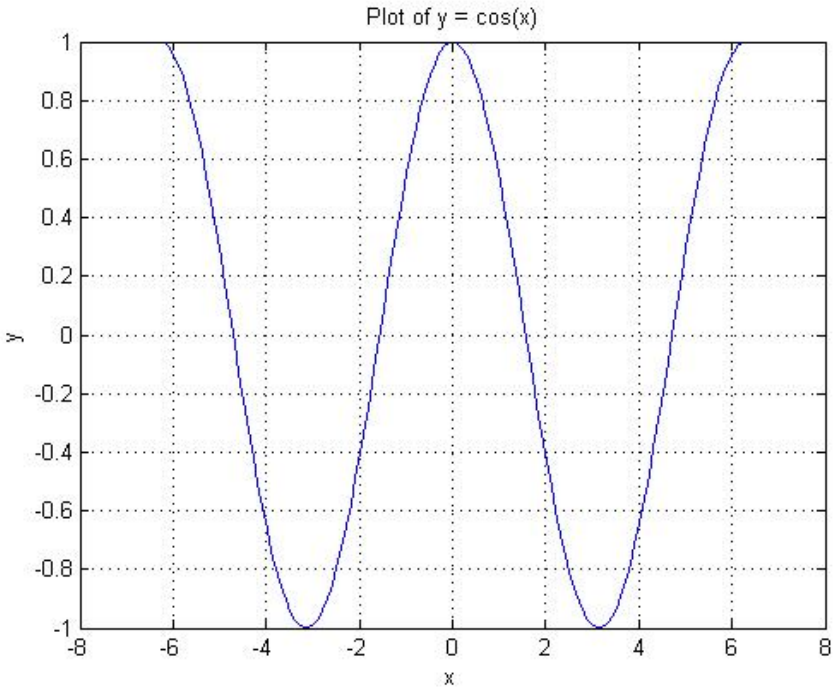


Figure 4.1.1: Plot of $y = \cos(x)$ generated using `plot`.

Example 4.1.2 *Multiple functions can be plotted on the same graph as follows.*

```
x = -2*pi:0.1:2*pi;
y1 = cos(x);
y2 = -sin(x);
plot(x,y1,x,y2); % Alternatively, use plot([x' x'],[y1' y2']);
                 % or, more simply, plot(x, [y1' y2']);
                 % Refer to the documentation at MathWorks.com
                 % for more variants.
title('Plot of y = cos(x) and y = -sin(x)');
xlabel('x');
ylabel('y');
grid on;
```

Then legends can be added by typing

```
legend('y = cos(x)', 'y = -sin(x)');
```

The result is shown in Figure 4.1.2.

Example 4.1.3 *This is the way to understand the statement*

```
plot([x' x'],[y1' y2']);
```

from last example. Note that `[x' x']` *and* `[y1' y2']` *are two matrices of the same size. In general, if* `A` *and* `B` *are two matrices of the same size, then* `plot(A, B);` *plots the columns of* `A` *versus the corresponding columns of* `B`*. For instance,*

```
A = repmat((0:0.01:1)', 1, 5); % This creates matrix of five
                               % columns each of which is the
                               % column vector (0:0.01:1)'.
B = randn(A);
plot(A, B);
```

yields the graphs of five randomly generated functions whose common domain is the interval $[0, 1]$*. In fact, since the columns of* `B` *are assumed to have the same domain, we can get the same result as follows:*

```
plot((0:0.01:1)', B);
```

More simply,

```
plot(0:0.01:1, B);
```

where MATLAB interprets the row vector `0:0.01:1` *as its transpose automatically.*

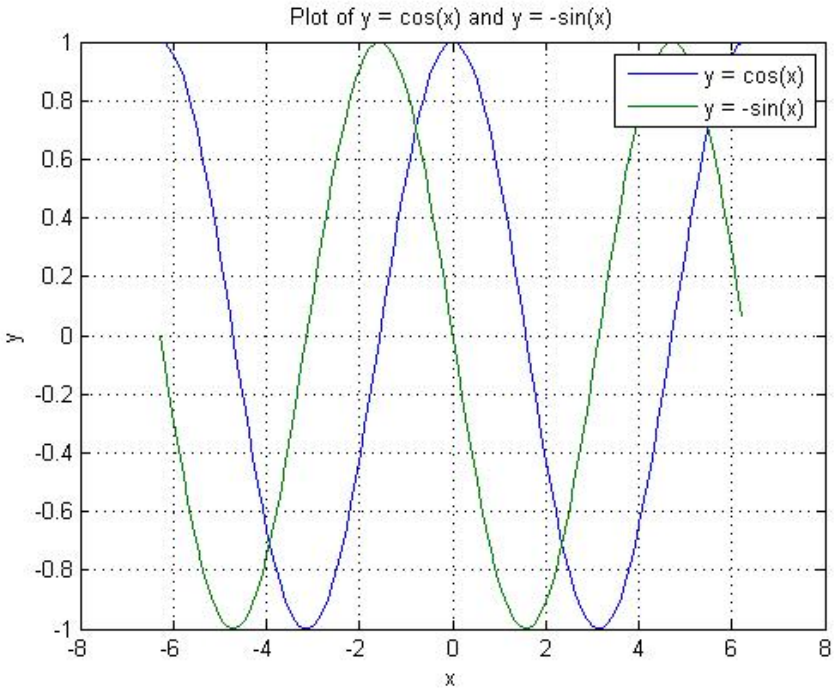


Figure 4.1.2: Plots of $y = \cos(x)$ and $y = -\sin(x)$ generated using `plot`.

Example 4.1.4 *There are options for the color, the marker type, and the line style of a plot. Consider*

```
x = -2*pi:0.1:2*pi;
y1 = cos(x);
y2 = -sin(x)
plot(x,y1,'r--', x, y2, 'bo');
title('Plot of y = cos(x) and y = -sin(x)');
xlabel('x');
ylabel('y');
grid on;
legend('y = cos(x)', 'y = -sin(x)');
```

The result is shown in Figure 4.1.3.

Here `'r--'`*, called* **attribute characters**, *indicate that the color of the*

plot is red and the line style is dashed (--). Likewise, 'bo' indicate that the color of the plot is blue and the marker type is circle (o). Both the marker type and the line style are options in the attribute characters. A complete list of colors, marker types, and line styles can be found by typing

```
help plot
```

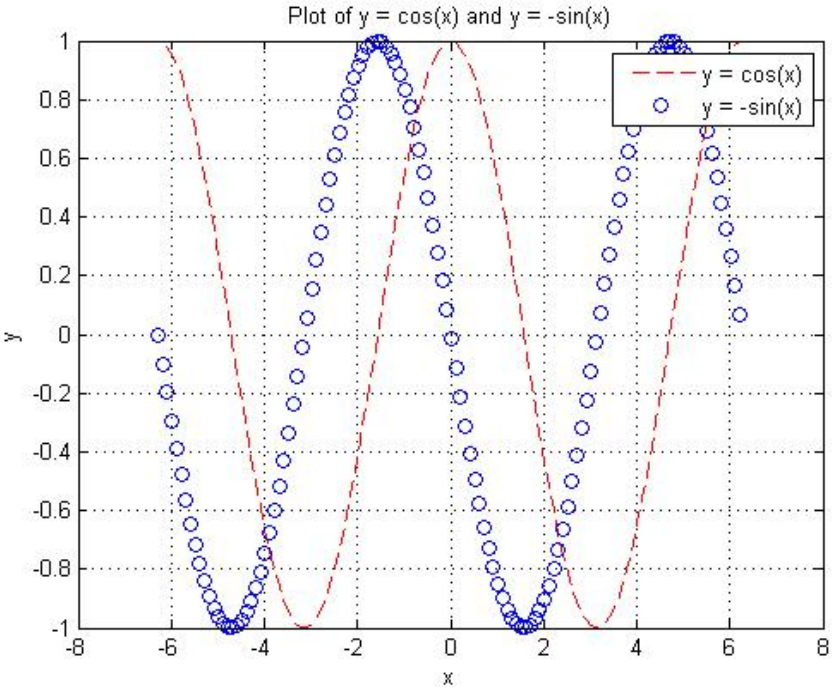


Figure 4.1.3: Plots of $y = \cos(x)$ and $y = -\sin(x)$ generated using `plot`. Attribute characters have been used to distinguish the plots.

MATLAB documentation by MathWorks on `plot` is available at

`https://www.mathworks.com/help/matlab/ref/plot.html`

Example 4.1.5 *We can also have multiple plots on the same graph using the command `hold`. For instance,*

```
x = -2*pi:0.1:2*pi;
y1 = cos(x);
plot(x, y1, 'b');
```

gives us the first plot. Then we can have the second plot on the same graph by typing

```
hold on;
y2 = -sin(x);
plot(x, y2, 'r');
```

The command `hold on` retains plots in the current axes so that subsequent plots added to the axes do not delete existing plots. The command `hold off` returns to the default mode whereby plot commands erase the previous plots and reset all axis properties before drawing new plots. The command `hold`, by itself, toggles the hold state.

MATLAB documentation by MathWorks on `hold` is available at

`https://www.mathworks.com/help/matlab/ref/hold.html`

Example 4.1.6 *Multiple functions can also be plotted within different Figure Windows using the command `figure`. For instance,*

```
x = -2*pi:0.1:2*pi;
y1 = cos(x);
figure(1);
plot(x, y1, 'b');
```

plots the first function in the first Figure Window, while

```
y2 = -sin(x);
figure(2); % This can be done simply using figure
           % which creates a new figure handle and opens a
```

```
          % a new Figure Window
plot(x, y2, 'r');
```

plots the second function in the second Figure Window. Note that `clf` *by itself clears the current Figure Window referred to by the* **currrent figure handle**. *This can be obtained by using the typing*

```
gcf
```

Suppose the output is

```
ans =

  Figure (1) with properties:

      Number: 1
        Name: ''
       Color: [0.9400 0.9400 0.9400]
    Position: [559 423 560 420]
       Units: 'pixels'

  Show all properties
```

Then typing `clf` *by itself clears Figure 1. To clear a particular one, say the Figure Window 2, type*

```
clf(2);
```

It is noted that the command `close` *exhibits the same behavior. In addition, to close all Figure Windows, we can type*

```
close all
```

We can set the current figure handle by using the command `figure`*. For example,*

```
figure(1);
```

sets the figure handle to refer to Figure Window 1. By itself, `figure` *creates a new Figure Window.*

MATLAB documentation by MathWorks on `figure` is available at

`https://www.mathworks.com/help/matlab/ref/figure.html`

4.2 The Command `subplot`

The command

```
subplot(m, n, p)
```

divides the current figure into $m \times n$ equal-sized rectangles, arranged in m rows and n columns, and selects the pth rectangle for current plot. The rectangles are numbered from left to right and from top to bottom.

Example 4.2.1 *The following sequence of MATLAB statements generates Figure 4.2.1:*

```
x = -2:0.1:2;
h1 = subplot(2, 3, 1);
plot(x, x);
xlabel('x');
ylabel('x');
h4 = subplot(2, 3, 4);
plot(x, -x);
xlabel('x');
ylabel('-x');
h3 = subplot(2, 3, 3);
plot(x, x.^3);
xlabel('x');
```

```
ylabel('x^3');
h5 = subplot(2, 3, 5);
plot(x, -x.^2);
xlabel('x');
ylabel('-x^2');
h2 = subplot(2, 3, 2);
plot(x, x.^2);
xlabel('x');
ylabel('x^2');
h6 = subplot(2, 3, 6);
plot(x, -x.^3);
xlabel('x');
ylabel('-x^3');
```

Each of the variables `h1`, `h2`, `h3`, `h4`, `h5`, *and* `h6` *is an* **axes object** *to the corresponding subplot. Each of them can be used alone to specify the axes for further plots or in conjunction with* `hold` *to set the hold state for the axes specified by the axes object.*

Try the following in the Command Window and observe the changes in the figure after the execution of each `plot` *statement:*

```
x = -pi:.1:pi;
plot(h6, x, sin(x));
hold(h1, 'on');
plot(h1, x, -x, 'r');
```

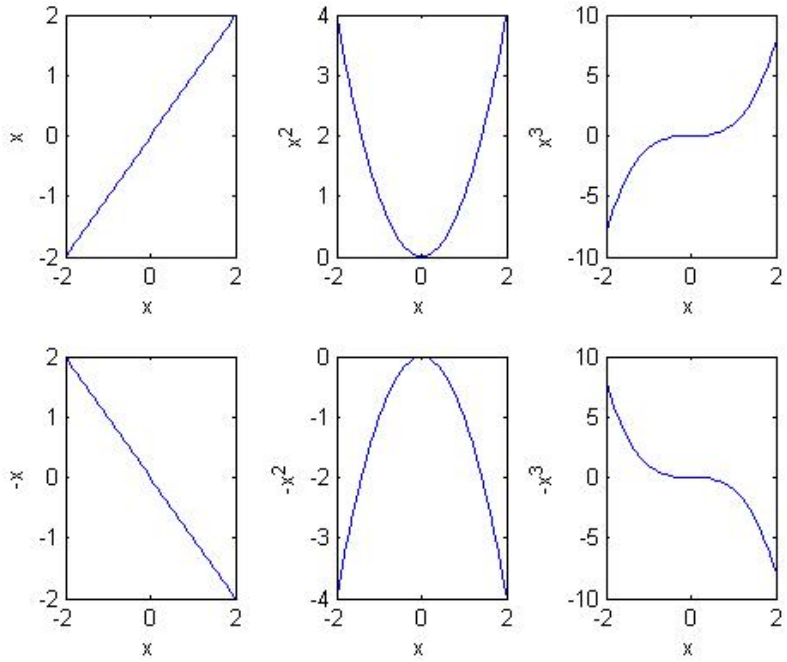


Figure 4.2.1: An instance of an application of `subplot`.

MATLAB documentation by MathWorks on `subplot` is available at

`https://www.mathworks.com/help/matlab/ref/subplot.html`

4.3 More Plotting Commands: `fplot` and `ezplot`

We can plot a function without first generating intermediate data arrays using the commands `fplot` and `ezplot`. To plot the sine function over the interval $[-\pi, \pi]$, we can do the following:

```
fplot(@sin, [-pi, pi]);
```

Note that the first input argument to `fplot` is a function handle. We can obtain the same result by entering

```
fplot(@(t) sin(t), [-pi, pi]);
```

More is true. We can plot the parametric curve $x(t) = \sin(2t)$, $y(t) = \cos(3t)$ parametrized by t on $[-\pi, \pi]$ by typing

```
x = @(t) sin(2*t);
y = @(t) cos(3*t);
fplot(x, y, [-pi, pi]);
```

Warning: MATLAB R2016b, more variants of input arguments are allowed; not so in the older versions of MATLAB.

MATLAB documentation by MathWorks on `fplot` is available at

`https://www.mathworks.com/help/matlab/ref/fplot.html`

Warning: `ezplot` is no longer recommended.

MATLAB documentation by MathWorks on `ezplot` is available at

`https://www.mathworks.com/help/matlab/ref/ezplot.html`

4.4 Simple 3D Plot – Curves in 3D

The command `plot3` plots curves in three-dimensions.

Example 4.4.1 *To generate a helix, the 3D parametric curve* $x(t) = \cos(t)$, $y(t) = \sin(t)$, $z(t) = t$ *parametrized by* t, *we enter*

```
t = linspace(0, 2*pi, 1001);
plot3(cos(t), sin(t), t);
```

As a remark, the statement `linspace(0, 2*pi, 1001)` *generates* 1001 *linearly equally spaced points between* 0 *and* 2π *inclusively*[¶]. *We can then label the axes by typing*

```
xlabel('x');
ylabel('y');
zlabel('z');
```

Likewise, we can generate a conical helix by entering

```
t = linspace(0, 20*pi, 2001);
plot3(t.*cos(t), t.*sin(t), t);
```

The command `plot3`*, as its parallel* `plot` *in two-dimensions, can be used concurrently with plotting commands such as* `subplot`, `title`, `grid`*, etc. Figure 4.4.1 is the result of such an instance.*

[¶]More generally, the statement `linspace(a, b, n)` generates a row vector of n linearly equally spaced points between the scalars `a` and `b`.

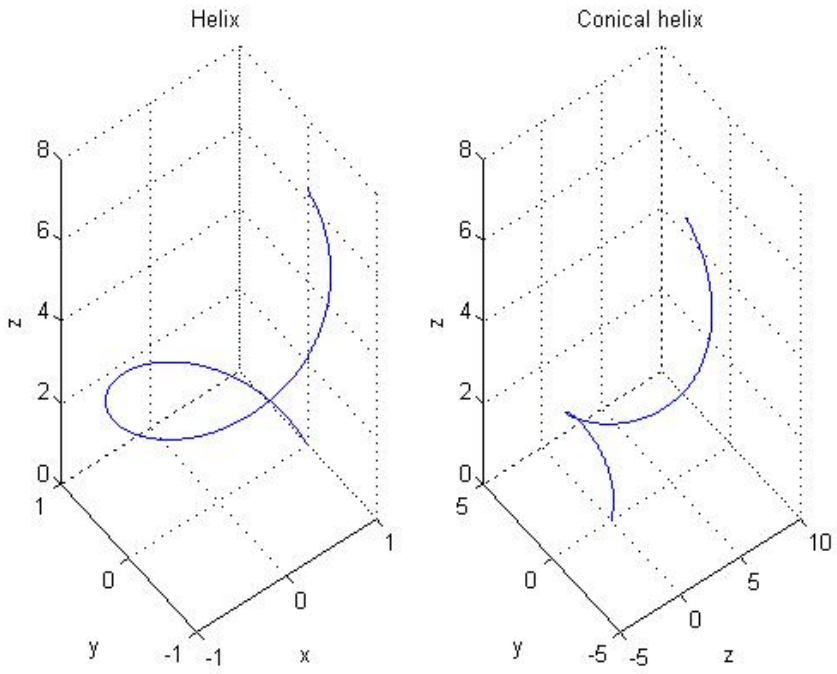


Figure 4.4.1: An instance of the application of `plot3`.

MATLAB documentation by MathWorks on `plot3` is available at

`https://www.mathworks.com/help/matlab/ref/plot3.html`

4.5 Simple 3D Plot – Mesh, Surface, and Contour Plots

Very often the underlying grids must be created before surfaces in the three-dimensional space can be plotted. One way to do so is to use the command `meshgrid` which combines a discretization of the x-axis, say

$$\{x_1, x_2, \ldots, x_m\},$$

and one of the y-axis, say

$$\{y_1, y_2, \ldots, y_n\},$$

into the rectangular mesh

$$\{(x_i, y_j) | i = 1, 2, \ldots, m,\ j = 1, 2, \ldots, n\}.$$

Example 4.5.1 *Suppose we have the following discretizations of the x- and the y-axes in MATLAB:*

```
x = 1:5

x =

     1     2     3     4     5

y = 7:9

y =

     7     8     9
```

The the statement

```
[X, Y] = meshgrid(x, y)
```

returns the following values for `X` *and* `Y`*, respectively:*

```
X =

     1     2     3     4     5
     1     2     3     4     5
     1     2     3     4     5


Y =

     7     7     7     7     7
     8     8     8     8     8
     9     9     9     9     9
```

For each element in `y`*, a copy of* `x` *is generated as a row in* `X`*. Likewise, for each element in* `x`*, a copy of* `y` *is generated as a column in* `Y`*. Note in particular that the size of each of* `X` *and* `Y` *is* `length(y)` *times* `length(x)`*. This way the discretization of the x-axis*

$$\{1, 2, 3, 4, 5\}$$

and that of the y-axis

$$\{7, 8, 9\}$$

are combined into the rectangular grid

$$\begin{array}{lllll} \{(1,7), & (2,7), & (3,7), & (4,7), & (5,7), \\ (1,8), & (2,8), & (3,8), & (4,8), & (5,8), \\ (1,9), & (2,9), & (3,9), & (4,9), & (5,9)\}. \end{array}$$

Why do we want to do that? Because we can then evaluate a function of two variables, say $z = z(x, y) = x^2 + y^2$, over the plane region containing the points in the rectangular grid.

```
z = X.^2+Y.^2

z =

    50    53    58    65    74
    65    68    73    80    89
    82    85    90    97   106
```

The sample run above indicates that $z(1, 7) = 50$, $z(2, 7) = 53$, etc.

MATLAB documentation by MathWorks on `meshgrid` is available at

`https://www.mathworks.com/help/matlab/ref/meshgrid.html`

MATLAB documentation by MathWorks on gridded data is available at

`https://www.mathworks.com/help/matlab/math/interpolating-gridded-data.html`

Example 4.5.2 *Consider the function*

$$z = e^{-0.5[x^2+0.5(x-y)^2]}.$$

We can create a **surface plot** *of its graph over the plane region defined by $-4 \leq x \leq 4$ and $-4 \leq y \leq 4$ by typing the following:*

```
[x y] = meshgrid(-4:0.2:4, -4:0.2:4);
% This can be compacted into [x, y] = meshgrid(-4:0.2:4)
z = exp(-0.5*(x.^2 + 0.5*(x-y).^2));
surf(x, y, z);
title('z=e^{-0.5[x^2+0.5(x-y)^2]}');
xlabel('x'); ylabel('y'); zlabel('z');
```

The output is Figure 4.5.1.

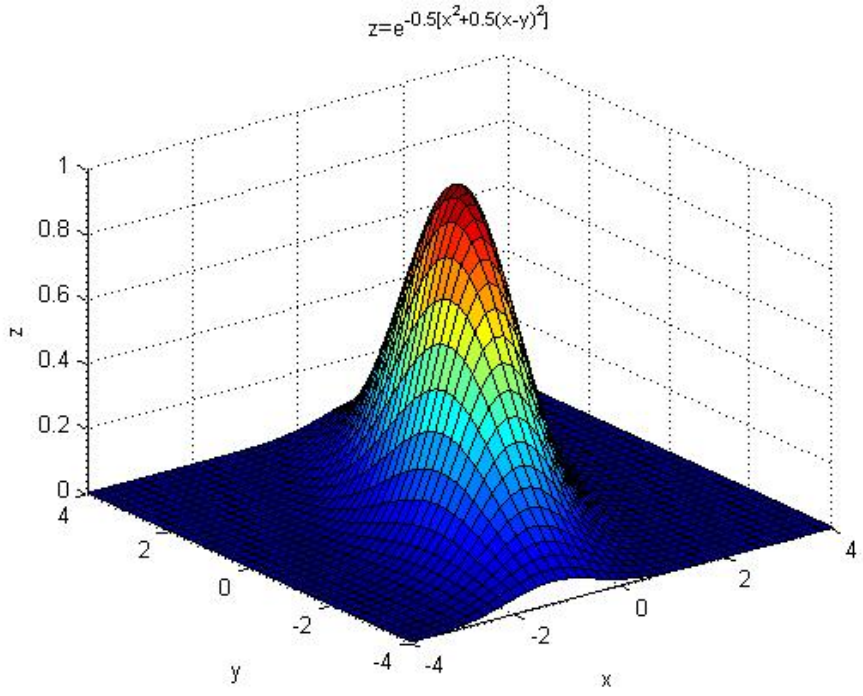


Figure 4.5.1: The output of an application of `surf`.

We can create a **mesh plot** *or* **wireframe plot** *of the same graph by typing*

```
mesh(x, y, z);
```

This gives Figure 4.5.2.

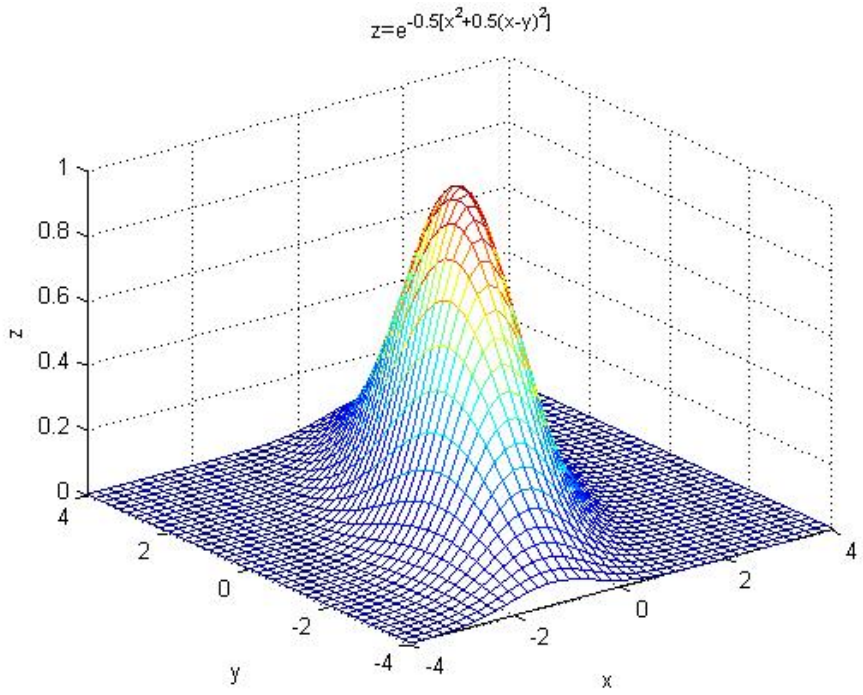


Figure 4.5.2: The output of an application of `mesh`.

Note the difference of `mesh` *and* `surf`*:* `mesh` *produces wireframe surfaces that color only the lines connecting the defining points, while* `surf` *displays both the connecting lines and the faces of the surface in color.*

Finally, we can create the **contour plot** *of the graph by entering*

```
contour(x, y, z);
```

See Figure 4.5.3.

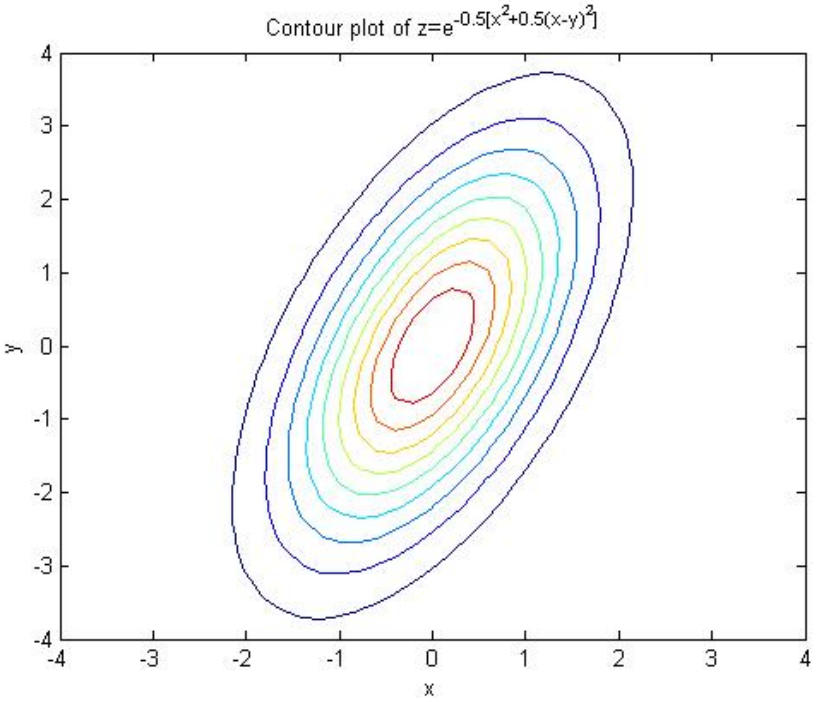


Figure 4.5.3: The output of an application of `contour`.

MATLAB documentation by MathWorks on `surf` is available at

`https://www.mathworks.com/help/matlab/ref/surf.html`

MATLAB documentation by MathWorks on `mesh` is available at

`https://www.mathworks.com/help/matlab/ref/mesh.html`

MATLAB documentation by MathWorks on `contour` is available at

`https://www.mathworks.com/help/matlab/ref/contour.html`

Example 4.5.3 *Consider the function*

$$z = \frac{\sin\left(\sqrt{x^2+y^2}\right)}{\sqrt{x^2+y^2}}.$$

Setting $r = \sqrt{x^2+y^2}$, *we usually write* $z = \dfrac{\sin r}{r}$ *and call it the sinc function. We can create the surface plot of the graph together with its contour beneath the surface by typing*

```
[x, y] = meshgrid(-8:.5:8);
r = sqrt(x.^2 + y.^2);
z = sin(r)./r;
surfc(x, y, z);
title('z=\sin(r)/r');
xlabel('x'); ylabel('y'); zlabel('z');
```

This gives Figure 4.5.4.

Likewise, we obtain a mesh plot with its contour beneath the plot by typing

```
meshc(x, y , z);
```

See Figure 4.5.5.

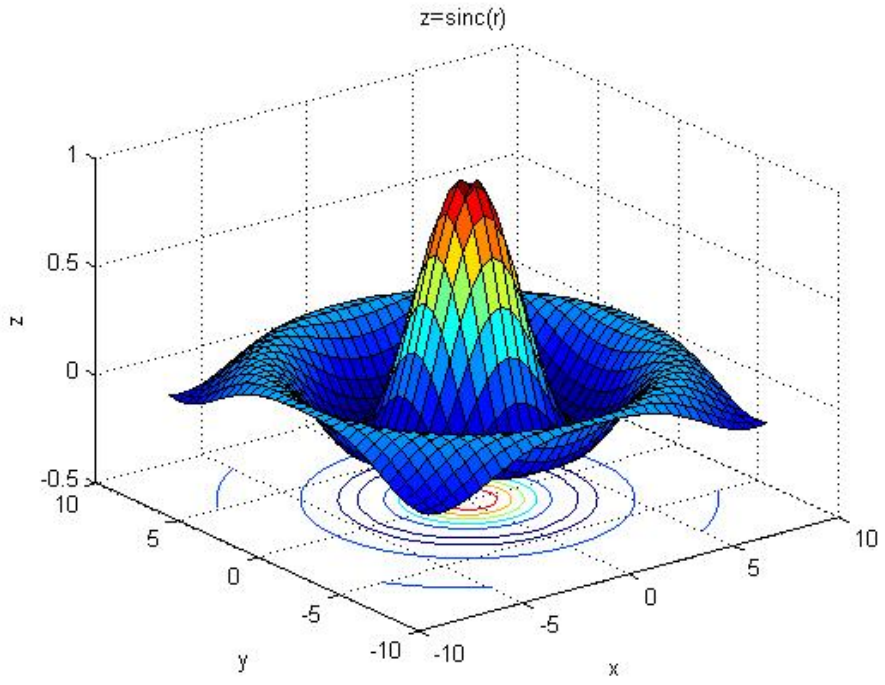


Figure 4.5.4: The output of an application of `surfc`.

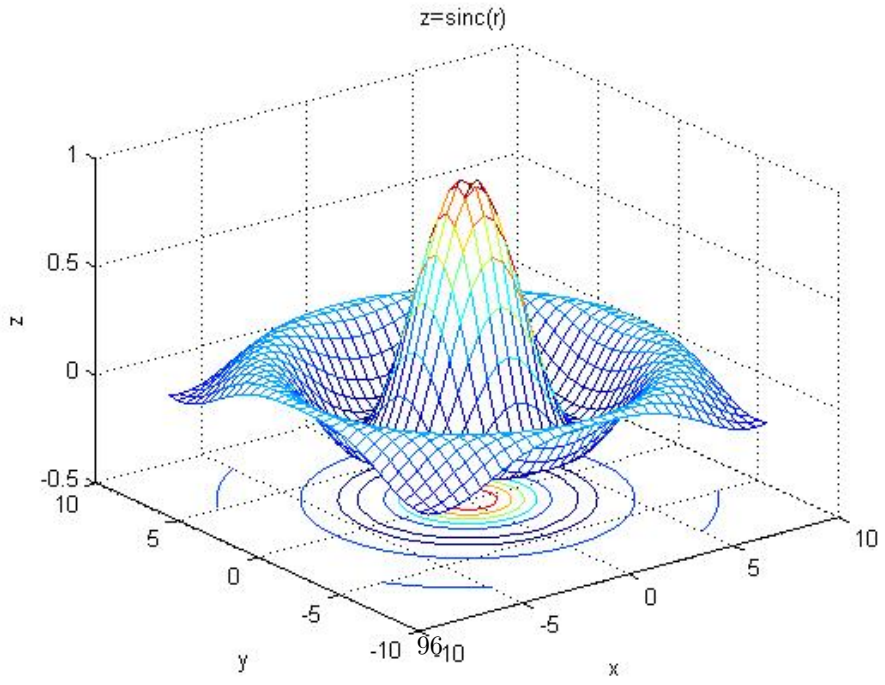


Figure 4.5.5: The output of an application of `meshc`.

MATLAB documentation by MathWorks on `surfc` is available at

`https://www.mathworks.com/help/matlab/ref/surfc.html`

MATLAB documentation by MathWorks on `meshc` is available at

`https://www.mathworks.com/help/matlab/ref/meshc.html`

5 Branching Statements

5.1 The Logical Data Type

The **logical data type** is a type of data taking one of only two possible values: `true` or `false`. These values are generated by

1. the functions `true`, `false`, and `logical`,
2. relational operators, and
3. logic operators.

If a real numerical value is used in place of a logical value, non-zero values are converted to `true` and a zero to a `false`. Conversely, if a logical value is used in placed of a numerical value, `true` values are converted to `1` and `false` values to `0`. Why do we use a logical value instead of a numerical value? Because *a logical value is stored in a single byte of memory, while a numerical value usually occupies* 8 *bytes.* Recall that the memory usage of a variable, say `A`, can be found by entering `whos('A')` in the Command Window. To list the info of all variables in the Workspace, enter `whos` instead.

MATLAB documentation by MathWorks on the function `true` is available at

https://www.mathworks.com/help/matlab/ref/true.html

MATLAB documentation by MathWorks on the function `false` is available at

https://www.mathworks.com/help/matlab/ref/false.html

Explicit conversions of logical values to numerical values and vice versa are achieved through the `logical` and `real` functions.

MATLAB documentation by MathWorks on `logical` is available at

https://www.mathworks.com/help/matlab/ref/logical.html

MATLAB documentation by MathWorks on `real` is available at

https://www.mathworks.com/help/matlab/ref/real.html

Warning: Neither complex nor imaginary numerical values can be used in place of logical values.

5.2 Relational Operators

A **relational operator** is an operator with two numerical or string operands which yields a result of logical data type. The general form of the usage of a relational operator `op` is

$$a_1 \texttt{ op } a_2$$

where a_1 and a_2 are compatible arithmetic expressions, variables, or strings and `op` is one of the operators shown below.

Operator	Operation
==	Equal to
~=	Not equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to

Generally, relational operators can be used to compare

1. a scalar with an array,
2. two arrays of the same dimensions, or
3. two strings of equal lengths.

Comparisons are made element-wise between the operands. Each comparison returns a `true` value (`1`) if the relationship between the corresponding elements expressed by the operator is true; otherwise, it returns a `false` value (`0`). See the following MATLAB session:

```
>> a = [1:3; 4:6; 7:9]; b = ones(3);
>> a == b

ans =

  3×3 logical array

   1   0   0
   0   0   0
```

```
   0   0   0

>> a > b

ans =

  3×3 logical array

   1   0   0
   0   0   0
   0   0   0

>> c = 'ABC'; d = 'AbD';
>> c == d

ans =

  1×3 logical array

   1   0   0

>> c < d

ans =

  1×3 logical array

     0     1     1
```

Warning: Do not confuse the equivalence relational operator (==) with the assignment operator (=).

As a remark, relational operators have the following equivalent functions.

Operator	Equivalent Functions
==	eq
~=	ne
>	gt
>=	ge
<	lt
<=	le

```
>> c = 'ABC'; d = 'AbD';
>> eq(c, d)

ans =

  1×3 logical array

>> lt(c, d)

ans =

  1×3 logical array
```

MATLAB documentation by MathWorks on array comparison with relational operators is available at

https://www.mathworks.com/help/matlab/matlab_prog/array-comparison-with-relational-operators.html

5.3 Logical Operators

A **logical operator** is an operator with one or two logical operands which yields a result of logical data type. The general form of the usage of a unary logic operator op is

$$\mathsf{op}\ l_1$$

while the one of a binary logic operator op is

$$l_1\ \mathsf{op}\ l_2$$

where l_1 and l_2 are expressions or variables and op is one of the operators shown below. Instead of logical data, real numerical data can also be used with logic operators.

Warning: Neither complex nor imaginary values can be used with logic operators.

Operator	Operation
&	Logical AND
&&	Logical AND with shortcut evaluation
|	Logical inclusive OR
||	Logical inclusive OR with shortcut evaluation
xor	Logical exclusive OR
∼	Logical NOT

The results of the operations are summarized in the following **truth table**.

Inputs		**AND**		**OR**		**XOR**	**NOT**
l_1	l_2	$l_1 \& l_2$	$l_1 \&\& l_2$	$l_1 \mid l_2$	$l_1 \mid\mid l_2$	$\mathsf{xor}(l_1, l_2)$	$\sim l_1$
0	0	0	0	0	0	0	1
0	1	0	0	1	1	1	1
1	0	0	0	1	1	1	0
1	1	1	1	1	1	0	0

As in the case of relational operators, operations are done elementwise between the operands. Also, there are equivalent functions for &, |, and ∼. See the following MATLAB session:

```
>> a = logical(randi([0,1],2,4))

a =

  2×4 logical array

   1   1   1   0
   1   1   0   1

>> b = logical(randi([0,1],2,4))

b =

  2×4 logical array

   0   0   1   1
   0   0   1   0

>> and(a, b) % This is the same to a&b.

ans =

  2×4 logical array
```

```
   0   0   1   0
   0   0   0   0

>> or(a, b) % This is the same to a|b.

ans =

  2×4 logical array

   1   1   1   1
   1   1   1   1

>> not(a) % This is the same to ~a.

ans =

  2×4 logical array

   0   0   0   1
   0   0   1   0
```

oindent

MATLAB documentation by MathWorks on logical operations is available at

`https://www.mathworks.com/help/matlab/logical-operations.html`

Why are there two AND operators? The differences between & and && are as follows.

1. && supports *short-circuit evaluations* (or partial evaluations), while & does not.
2. && only works between scalars, while & works with scalars or arrays of compatible sizes.

There is a similar set of differences between the OR operators | and ||.

Now we explain short-circuit evaluations of && and ||. For the statement

$$l_1 \,\&\&\, l_2$$

if l_1 is logical `0`, then l_2 is not evaluated because the value of l_1 is sufficient to tell that the above statement evaluates to a logical `0`. Likewise, for the statement

$$l_1 \,||\, l_2$$

if l_1 is logical `1`, then l_2 is not evaluated because the value of l_1 is sufficient to tell that the above statement evaluates to a logical `1`.

MATLAB documentation by MathWorks on short-circuit evaluations is available at

`https://www.mathworks.com/help/matlab/ref/logicaloperatorsshortcircuit.html`

5.4 Hierarchy of Operations

The following is the order in which the operators in an expression are evaluated.

1. All arithmetic operations are evaluated first in the order previously described.
2. All relational operators are evaluated from left to right.
3. All NOT operators are evaluated.
4. All AND operators are evaluated from left to right.
5. All OR and XOR operators are evaluated from left to right.

Nevertheless, parentheses can be added to change the default order of evaluation.

MATLAB documentation by MathWorks on operator precedence is available at

`https://www.mathworks.com/help/matlab/matlab_prog/operator-precedence.html`

5.5 More Logical Functions

In additional to `true`, `false`, `logical` and the logical operators mentioned above, some other logical functions are `strcmp`, `isequal`, `all`, `any`, `find`, `ischar`, `isnumeric`, `isempty`, `isinf`, and `isnan`. Readers are encouraged to browse through the documentation by MathWorks on these functions.

In order to compare two character vectors, one uses `strcmp`:

```
>> s1 = 'This is OK.'

s1 =

    'This is OK.'

>> s2 = 'This is NOT OK.'

s2 =

    'This is NOT OK.'

>> t = s1

t =

    'This is OK.'

>> strcmp(s1, s2)

ans =

  logical

   0

>> strcmp(s1, t)

ans =

  logical

   1
```

MATLAB documentation by MathWorks on `strcmp` is available at

`https://www.mathworks.com/help/matlab/ref/strcmp.html`

In order to compare two arrays, one must use `isequal`:

```
>> a = rand(2)

a =

    0.8323    0.9575
    0.5526    0.8928

>> b = rand(2)

b =

    0.3565    0.3467
    0.5464    0.6228

>> isequal(a, b)

ans =

  logical

   0

>> isequal(a, a)

ans =

  logical

   1
```

MATLAB documentation by MathWorks on `isequal` is available at

`https://www.mathworks.com/help/matlab/ref/isequal.html`

MATLAB documentation by MathWorks on `all` is available at

`https://www.mathworks.com/help/matlab/ref/all.html`

MATLAB documentation by MathWorks on `any` is available at

`https://www.mathworks.com/help/matlab/ref/any.html`

MATLAB documentation by MathWorks on `find` is available at

`https://www.mathworks.com/help/matlab/ref/find.html`

MATLAB documentation by MathWorks on `ischar` is available at

`https://www.mathworks.com/help/matlab/ref/ischar.html`

MATLAB documentation by MathWorks on `isnumeric` is available at

`https://www.mathworks.com/help/matlab/ref/isnumeric.html`

MATLAB documentation by MathWorks on `isempty` is available at

`https://www.mathworks.com/help/matlab/ref/isempty.html`

MATLAB documentation by MathWorks on `isinf` is available at

`https://www.mathworks.com/help/matlab/ref/isinf.html`

MATLAB documentation by MathWorks on `isnan` is available at

`https://www.mathworks.com/help/matlab/ref/isnan.html`

5.6 The `if` Construct

The `if` construct has the form

```
if control_expr_1
     block of statements #1
```

```
elseif control_expr_2
     block of statements #2
     ...
     ...
else
     block of statements #3
end
```

There can actually be no or more than one `elseif` clauses, but there can only be at most one `else` clause. Also, the `else` clause is optional.

The expressions `control_expr_1`, `control_expr_2`, etc. are **control expressions**, which are some combinations of relational or logic operators.

1. If a control expression is evaluated to be `false` (a logical `0`), the next control expression will be tested.
2. If `control_expr_j` is the first control expression evaluated to be `true` (a logical `1`), then
   - the block of statements $\#j$ is executed, and
   - program execution skips to the first statement following the end keyword.
3. If all control expressions are evaluated to be `false`, then the block of statements associated with the else clause is executed.

Good practice: Always indent the body of an `if` construct by two or more spaces to improve readability of the code.

The `if` constructs can be nested. Two `if` constructs are **nested** if one lies entirely within a single code block of the other.

Good practice: For branches in which there are many mutually exclusive options, use a single `if` construct with multiple `elseif` clauses instead of nested `if` constructs.

Example 5.6.1 *The following MATLAB script using `if` construct is adapted from [1, Example 4.3]. It returns the value of*

$$f(x,y) = \begin{cases} x+y & \text{if } x \geq 0 \text{ and } y \geq 0, \\ x+y^2 & \text{if } x \geq 0 \text{ and } y < 0, \\ x^2+y & \text{if } x < 0 \text{ and } y \geq 0, \\ x^2+y^2 & \text{if } x < 0 \text{ and } y < 0. \end{cases}$$

for any user-specified values of x and y.

```
%  Script file: funxy.m
%
%  Purpose:
%    This program solves the function f(x,y) for a
%    user-specified x and y, where f(x,y) is defined as:
%                  -
%                 |
%                 | x + y             x >= 0 and y >= 0
%                 | x + y^2           x >= 0 and y < 0
%       f(x,y) = | x^2 + y           x < 0  and y >= 0
%                 | x^2 + y^2         x < 0  and y < 0
%                 |-
%
%  Record of revisions:
%      Date       Programmer          Description of change
%      ====       ==========          =====================
%    01/03/10    S. J. Chapman        Original code
%
% Define variables:
%   x      -- First independent variable
%   y      -- Second independent variable
%   fun    -- Resulting function

% Prompt the user for the values x and y
x = input ('Enter the x coefficient: ');
y = input ('Enter the y coefficient: ');

% Calculate the function f(x,y) based upon
% the signs of x and y.
if x >= 0 && y >= 0
   fun = x + y;
elseif x >= 0 && y < 0
   fun = x + y^2;
elseif x < 0 && y >= 0
   fun = x^2 + y;
else
   fun = x^2 + y^2;
end

% Write the value of the function.
disp (['The value of the function is ' num2str(fun)]);
```

Example 5.6.2 *The following MATLAB script using* `if` *construct is adapted from [1, Example 4.2]. It solves for the roots of a quadratic equation of the form* $ax^2 + bx + c = 0$.

```
%  Script file: calc_roots.m
%
%  Purpose:
%    This program solves for the roots of a quadratic equation
%    of the form a*x**2 + b*x + c = 0.  It calculates the answers
%    regardless of the type of roots that the equation possesses.
```

```
%
%  Record of revisions:
%      Date       Programmer          Description of change
%      ====       ==========          =====================
%    01/02/10    S. J. Chapman        Original code
%
% Define variables:
%   a             -- Coefficient of x^2 term of equation
%   b             -- Coefficient of x term of equation
%   c             -- Constant term of equation
%   discriminant -- Discriminant of the equation
%   imag_part     -- Imag part of equation (for complex roots)
%   real_part     -- Real part of equation (for complex roots)
%   x1            -- First solution of equation (for real roots)
%   x2            -- Second solution of equation (for real roots)

% Prompt the user for the coefficients of the equation
disp ('This program solves for the roots of a quadratic ');
disp ('equation of the form A*X^2 + B*X + C = 0. ');
a = input ('Enter the coefficient A: ');
b = input ('Enter the coefficient B: ');
c = input ('Enter the coefficient C: ');

% Calculate discriminant
discriminant = b^2 - 4 * a * c;

% Solve for the roots, depending on the value of the discriminant
if discriminant > 0 % there are two real roots, so...

   x1 = ( -b + sqrt(discriminant) ) / ( 2 * a );
   x2 = ( -b - sqrt(discriminant) ) / ( 2 * a );
   disp ('This equation has two real roots:');
   fprintf ('x1 = %f\n', x1);
   fprintf ('x2 = %f\n', x2);

elseif discriminant == 0  % there is one repeated root, so...

   x1 = ( -b ) / ( 2 * a );
   disp ('This equation has two identical real roots:');
   fprintf ('x1 = x2 = %f\n', x1);

else % there are complex roots, so ...

   real_part = ( -b ) / ( 2 * a );
   imag_part = sqrt ( abs ( discriminant ) ) / ( 2 * a );
   disp ('This equation has complex roots:');
   fprintf('x1 = %f +i %f\n', real_part, imag_part );
   fprintf('x1 = %f -i %f\n', real_part, imag_part );

end
```

MATLAB documentation by MathWorks on the `if` construct is available at

https://www.mathworks.com/help/matlab/ref/if.html

5.7 The `switch` Construct

The `switch` construct has the form

```
switch (swith_expr)
     case case_expr_1
          block of statements #1
     case case_expr_2
          block of statements #2
          ...
          ...
     otherwise
          block of statements #3
end
```

There can actually be more than one `case` clauses, but there can only be at most one `otherwise` clause. Also, the `otherwise` clause is optional.

The execution of the `switch` is similar to the one of the `if` construct. Here the value of `switch_expr` is tested against the values of `case_expr_1`, `case_expr_2`, etc. to decide which block of statements is to be executed.

If multiple values of `switch_expr` cause the execution of the same block of codes, those values can be included in a single block by enclosing them in brackets as follows.

```
switch (swith_expr)
     case {case_expr_1, case_expr_2, case_expr_3}
          block of statements #1
          ...
          ...
     otherwise
          block of statements #2
end
```

Example 5.7.1 *The following example using* `switch` *construct is adapted from [1]. It determines whether an integer between* 1 *and* 10 *is even or odd, and print an appropriate message.*

```
switch(value)
     case {1,3,5,7,9}
          disp('The value is odd.');
     case {2,4,6,8,10}
          disp('The value is even.');
     otherwise
          disp('The value is out of range.');
end
```

The same effect can be achieved by using `if` *statement:*

```
if mod(value, 2) == 1
     disp('The value is odd.');
else
     disp('The value is even.');
end
```

MATLAB documentation by MathWorks on the `switch` construct is available at

`https://www.mathworks.com/help/matlab/ref/switch.html`

6 Loops

6.1 The `while` Loop

A **while loop** is a block of statements which is repeated indefinitely so long as some condition is satisfied. The general form of such a loop is

```
while control_expr
     loop body (block of codes)
end
```

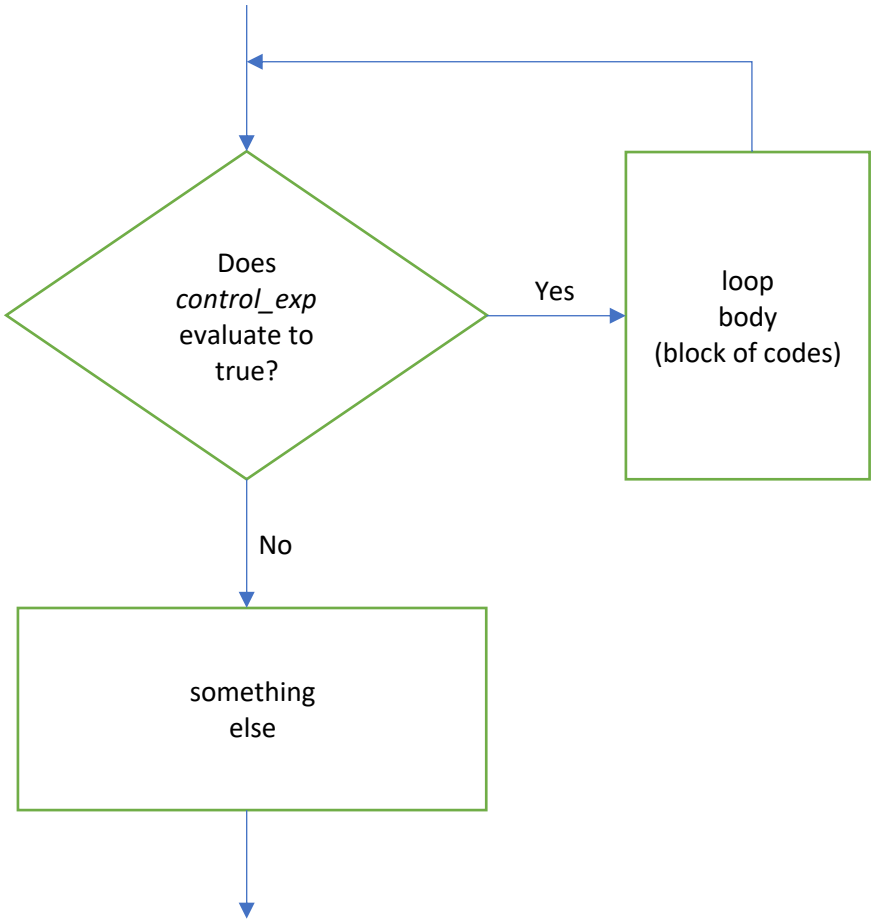
The control expression `control_expr` should produce a logical value.

1. If `control_expr` evaluates to true, then the block of codes within the loop, called the **loop body**, is executed. Upon the completion of the execution, the control is passed back to the while statement and the control expression `control_expr` is re-tested. The process is repeated until `control_expr` becomes false.
2. If `control_expr` evaluates false, then the program execution skips to the first statement following the end keyword.

Warning: If `control_expr` evaluates to a matrix, the loop body will only be executed if all elements of the matrix are true. To have the loop body executed if any element of the matrix is true, wrap the matrix in the function `any`.

Warning: If `control_expr` always evaluates to true within the loop body, then the loop will never terminate, resulting in an **infinite loop**, unless the flow of the program runs into a `break` statement.

Illustration of a **while** loop



Example 6.1.1 *The following script m-file, adapted from [1, Example 5.1] calculates the mean and the standard deviation of an input data set containing an arbitrary number of input values. It makes use of a* `while` *loop to allow the user to type in as many number of inputs as he/she wants until the user indicates to the program that there is no more data using a negative input.*

```
%  Script file: stats_2.m
%
%  Purpose:
%    To calculate mean and the standard deviation of
%    an input data set containing an arbitrary number
%    of input values.
%
%  Record of revisions:
%      Date       Programmer          Description of change
%      ====       ==========          =====================
%    01/24/14    S. J. Chapman        Original code
% 1. 01/24/14    S. J. Chapman        Correct divide-by-0 error if
%                                     0 or 1 input values given.
%
% Define variables:
%   n       -- The number of input samples
%   std_dev -- The standard deviation of the input samples
%   sum_x   -- The sum of the input values
%   sum_x2  -- The sum of the squares of the input values
%   x       -- An input data value
%   xbar    -- The average of the input samples

% Initialize sums.
n = 0; sum_x = 0; sum_x2 = 0;

% Read in first value
x = input('Enter first value: ');

% While Loop to read input values.
while x >= 0

   % Accumulate sums.
   n      = n + 1;
   sum_x  = sum_x + x;
   sum_x2 = sum_x2 + x^2;

   % Read in next value
   x = input('Enter next value:  ');

end

% Check to see if we have enough input data.
if n < 2   % Insufficient information

   disp('At least 2 values must be entered!');

else % There is enough information, so
     % calculate the mean and standard deviation
```

```
    x_bar = sum_x / n;
    std_dev = sqrt( (n * sum_x2 - sum_x^2) / (n * (n-1)) );

    % Tell user.
    fprintf('The mean of this data set is: %f\n', x_bar);
    fprintf('The standard deviation is:    %f\n', std_dev);
    fprintf('The number of data points is: %f\n', n);

end
```

MATLAB documentation by MathWorks on `while` loop is available at

`https://www.mathworks.com/help/matlab/ref/while.html`

For interested readers, there is a general discussion on the while construct at Wikipedia:

`https://en.wikipedia.org/wiki/While_loop`

6.2 The `for` Loop

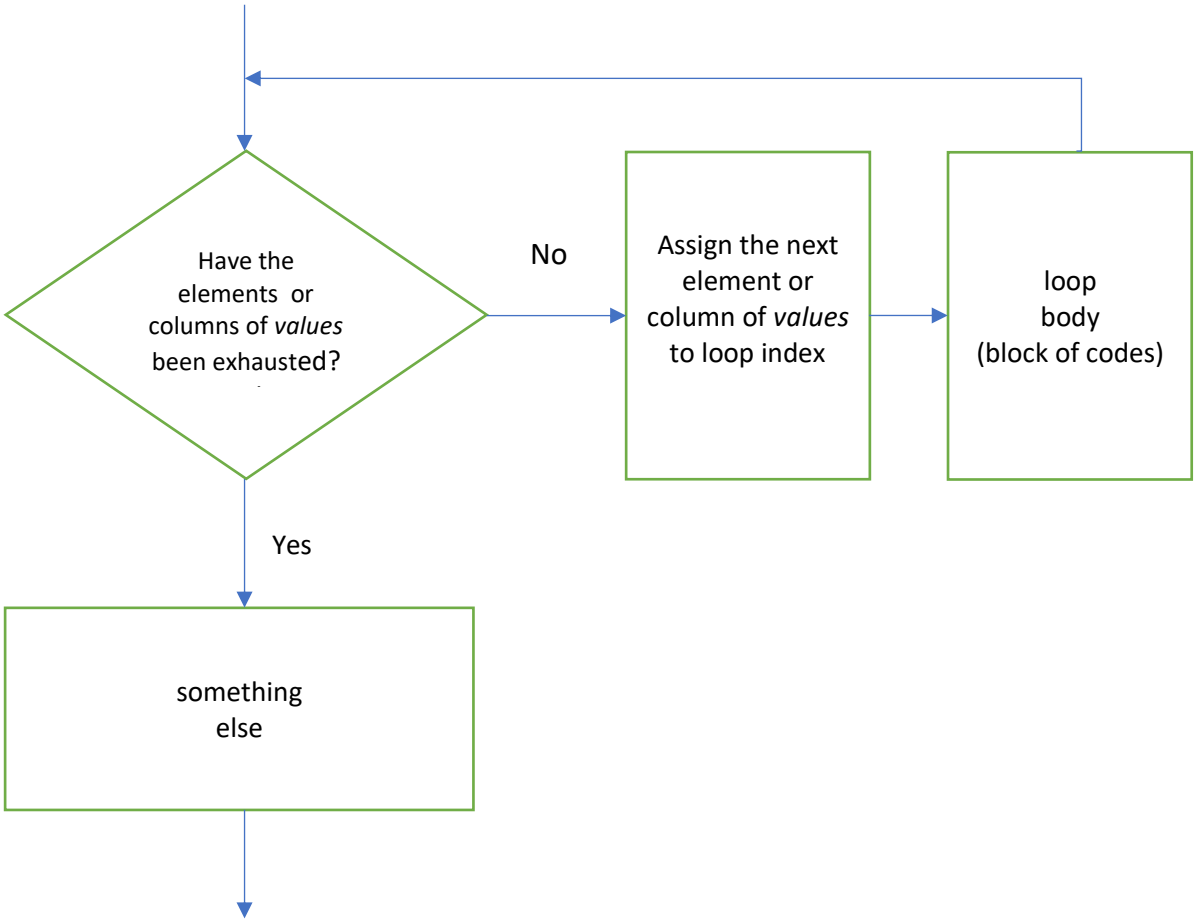
A **for loop** is a block of statements which is repeated a specified number of times. The general form of such a loop is

```
for index = values
     loop body (block of codes)
end
```

The variable `index` is called a **loop variable** (or a **loop index**), while `values` is the loop control expression. If `values` is a vector, then each of its elements is stored one at a time in `index` and the loop body is executed for that particular value of `index`. That is, the loop body is run once for each element of `values`.

Otherwise, if `values` is a matrix, then each of its columns is stored one at a time in `index` and the loop body is executed for that particular value, which is a column vector, of `index`. This way the loop body is run once for each column of `values`.

Illustration of a **for** loop



Good practice: Codes are much more readable if the bodies of loops are indented properly.

Good practice: The loop variable should not be modified anywhere within the loop body. Modifying the loop index within the loop body often causes hard-to-find errors.

Example 6.2.1 *The following function m-file uses a* `for` *loop to compute the factorial of a nonnegative integer* `n`*:*

```
function fact2 = factorial2(n)
% FACTACTORIAL2 Calculate the factorial of n
% Function FACTACTORIAL2 calculates the product 1*2*...*n
% provided n is a positive integer or returns 0 if n = 0.
% It returns an error message if n < 0 or if n is non-integer.
%
% Calling sequence:
%   factorial2 = fact(n)

% Define variables:
%   n    = the integer n as defined in the description of
%          the function.
% fact2 = factorial of n
% iter  = iteration variable

if (n < 0) || (floor(n) ~= n)
    fprintf('Error: n must be a nonnegative integer.\n\n');
else
    fact2 = 1;
    for iter = 2:n
        fact2 = fact2 * iter;
    end
end

end
```

A session of calling the function yields

```
>> factorial2(-5)
Error: n must be a nonnegative integer.

>> factorial2(3.7)
Error: n must be a nonnegative integer.

>> factorial2(10)
```

```
ans =

     3628800
```

Note that MATLAB has already a more powerful built-in factorial function called `factorial`.

Example 6.2.2 *The following MATLAB script is adapted from [1, Example 5.3]. It accepts a day, mpnth, and year, and calculates the day of year corresponding to that date.*

```
%  Script file: doy.m
%
%  Purpose:
%    This program calculates the day of year corresponding
%    to a specified date.  It illustrates the use switch
%    and for constructs.
%
%  Record of revisions:
%      Date       Programmer          Description of change
%      ====       ==========          =====================
%    01/27/10    S. J. Chapman        Original code
%
% Define variables:
%   day           -- Day (dd)
%   day_of_year   -- Day of year
%   ii            -- Loop index
%   leap_day      -- Extra day for leap year
%   month         -- Month (mm)
%   year          -- Year (yyyy)

% Get day, month, and year to convert
disp('This program calculates the day of year given the ');
disp('specified date.');
month = input('Enter specified month (1-12): ');
day   = input('Enter specified day(1-31):    ');
year  = input('Enter specified year(yyyy):   ');

% Check for leap year, and add extra day if necessary
if mod(year,400) == 0
   leap_day = 1;          % Years divisible by 400 are leap years
elseif mod(year,100) == 0
   leap_day = 0;          % Other centuries are not leap years
elseif mod(year,4) == 0
   leap_day = 1;          % Otherwise every 4th year is a leap year
else
   leap_day = 0;          % Other years are not leap years
end

% Calculate day of year by adding current day to the
% days in previous months.
day_of_year = day;
```

```
for ii = 1:month-1

   % Add days in months from January to last month
   switch (ii)
   case {1,3,5,7,8,10,12},
      day_of_year = day_of_year + 31;
   case {4,6,9,11},
      day_of_year = day_of_year + 30;
   case 2,
      day_of_year = day_of_year + 28 + leap_day;
   end

end

% Tell user
fprintf('The date %2d/%2d/%4d is day of year #%d.\n', ...
         month, day, year, day_of_year);
```

MATLAB documentation by MathWorks on `for` loop is available at

`https://www.mathworks.com/help/matlab/ref/for.html`

For interested readers, there is a general discussion on the for loop at Wikipedia:

`https://en.wikipedia.org/wiki/For_loop`

6.3 The `break` and `continue` Statements

The **`break` statement** terminates the execution of a loop and passes control to the next statement right after the end of the loop, while the **`continue` statement** terminates the current pass through the loop and returns control to the top of the loop.

Example 6.3.1 *The following `for` loop will run only for the first five iterations:*

```
for k = 1:10
     fprintf('k = %d\n', k);
     if k == 5
          break
     end
end
```

This is confirmed by observing the output:

```
k = 1
k = 2
k = 3
k = 4
k = 5
```

The following `for` *loop bypasses the fifth and the eighth iterations:*

```
for k = 1:10
     if k == 5 || k == 8
          continue
     end
     fprintf('k = %d\n', k);
end
```

The output is

```
k = 1
k = 2
k = 3
k = 4
k = 6
k = 7
k = 9
k = 10
```

Example 6.3.2 *The* `break` *and* `continue` *statements exhibit the same behavior within* `while` *loops as they would within* `for` *loops. The following* `while` *loop*

```
k = 0;
while k <= 10
     k = k + 1;
     fprintf('k = %d\n', k);
     if k == 5
```

```
          break
     end
end
```

is equivalent to the first `for` *loop in the previous example, while the following* `while` *loop*

```
k = 0;
while k < 10
     k = k + 1;
     if k == 5 || k == 8
          continue
     end
     fprintf('k = %d\n', k);
end
```

is equivalent to the second `for` *loop.*

Example 6.3.3 *If one loop is completely within another, the loops are called* **nested loops**. *For instance,*

```
for m = 1:3
     for n = 1:3
          fprintf('m = %d   n = %d\n', m, n);
     end
end
```

yields

```
m = 1   n = 1
m = 1   n = 2
m = 1   n = 3
m = 2   n = 1
m = 2   n = 2
m = 2   n = 3
m = 3   n = 1
m = 3   n = 2
m = 3   n = 3
```

We may have more than two nest loops, say

```
for m = 1:2
     for n = 1:2
          for p = 1:2
               fprintf('m = %d   n = %d   p = %d\n', m, n, p);
          end
     end
end
```

yields

```
m = 1   n = 1   p = 1
m = 1   n = 1   p = 2
m = 1   n = 2   p = 1
m = 1   n = 2   p = 2
m = 2   n = 1   p = 1
m = 2   n = 1   p = 2
m = 2   n = 2   p = 1
m = 2   n = 2   p = 2
```

Good practice: *Codes are much more readable if the bodies of loops are indented properly. (Yes, this is the second time this advice is given.)*

The `break` *and the* `continue` *statements always refer to the loop containing it. For instance,*

```
for m = 1:3
     for n = 1:3
          if m == 2
               break
          end
     fprintf('m = %d   n = %d\n', m, n);
     end
end
```

yields

```
m = 1   n = 1
m = 1   n = 2
m = 1   n = 3
m = 3   n = 1
m = 3   n = 2
m = 3   n = 3
```

It has the same effect as the following script using the `continue` *statement:*

```
for m = 1:3
     for n = 1:3
          if m == 2
               continue
          end
     fprintf('m = %d   n = %d\n', m, n);
     end
end
```

So has the script

```
for m = 1:3
     if m == 2
          continue
     end
     for n = 1:3
          fprintf('m = %d   n = %d\n', m, n);
     end
end
```

On the other hand,

```
for m = 1:3
     if m == 2
          break
     end
     for n = 1:3
```

```
            fprintf('m = %d   n = %d\n', m, n);
        end
end
```

yields

```
m = 1   n = 1
m = 1   n = 2
m = 1   n = 3
```

MATLAB documentation by MathWorks on the `break` statement is available at

`https://www.mathworks.com/help/matlab/ref/break.html`

MATLAB documentation by MathWorks on the `continue` statement is available at

`https://www.mathworks.com/help/matlab/ref/continue.html`

Warning: Neither `break` nor `continue` can be used outside a `for` or `while` loop. To exit a function, use `return`.

MATLAB documentation by MathWorks on the `return` statement is available at

`https://www.mathworks.com/help/matlab/ref/return.html`

References

[1] Stephen J. Chapman. *MATLAB Programming with Applications for Engineers*. Cengage Learning, 1st edition, 2013.

[2] Brian D. Hahn and Daniel T. Valentine. *Essential MATLAB for Engineers and Scientists*. Elsevier, 2017.

[3] Craig S. Lent. *Learning to Program with MATLAB Building GUI*. John Wiley & Sons, Inc., 2013.

[4] Ed Overman. *A MATLAB Tutorial*. August 2018.