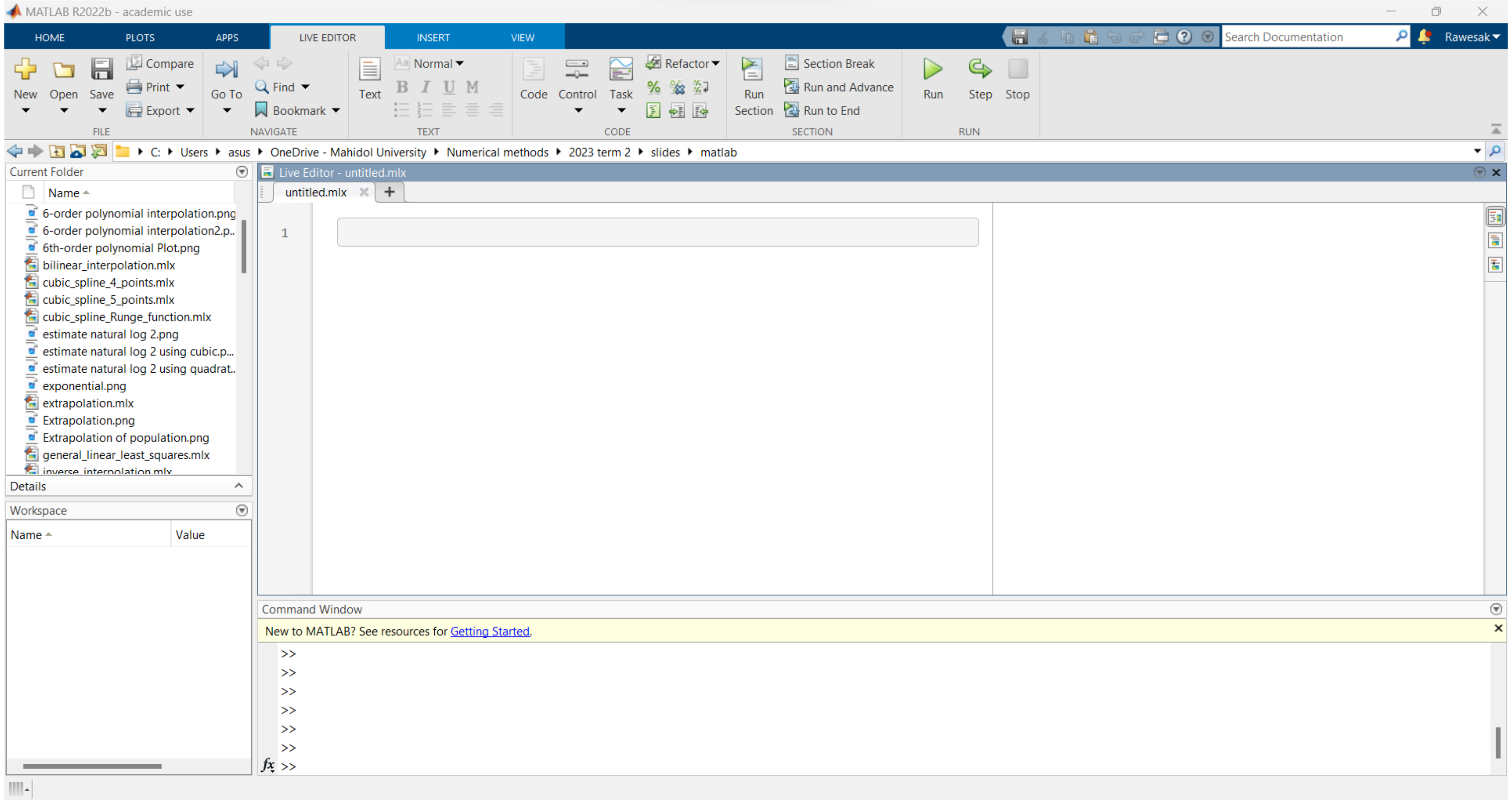
Mahidol University
Faculty of Information and Communication Technology

MATLAB Fundamentals

Rawesak Tanawongsuwan, Ph.D.
rawesak.tan@mahidol.ac.th

This slide is part of teaching materials for ITCS122 Numerical Methods
Semester 2/2023, Calendar year 2024

The MATLAB Environment



MATLAB Variables

- MATLAB allows you to assign values to variable names. This results in the storage of values to memory locations corresponding to the variable name.
- MATLAB can store individual values as well as arrays; it can store numerical data and text (which is actually stored numerically as well).
- MATLAB does not require that you pre-initialize a variable; if it does not exist, MATLAB will create it for you.

Scalars

- To assign a single value to a variable, simply type the variable name, the = sign, and the value:

```
>> a = 4
a =
  4
```

Note that variable names must start with a letter, though they can contain letters, numbers, and the underscore (_) symbol

- You can tell MATLAB not to report the result of a calculation by appending the semi-colon (;) to the end of a line. The calculation is still performed.

Formats

- You can tell MATLAB to report the values back with several different formats using the format command. Note that the values are still *stored* the same way, they are just displayed on the screen differently. Some examples are:
- `short` - scaled fixed-point format with 5 digits.
- `long` - scaled fixed-point format with 15 digits for double and 7 digits for single.
- `short eng` - engineering format with at least 5 digits and a power that is a multiple of 3 (useful for SI prefixes).

Format Examples

- ```
  >> format short; pi
  ans =
      3.1416
  >> format long; pi
  ans =
     3.14159265358979
  >> format short eng; pi
  ans =
       3.1416e+000
  >> pi*10000
  ans =
      31.4159e+003
  ```
- Note - the format remains the same unless another `format` command is issued.
  ```

Arrays, Vectors, and Matrices

- MATLAB can automatically handle rectangular arrays of data - one-dimensional arrays are called vectors and two-dimensional arrays are called *matrices*.
- Arrays are set off using square brackets [and] in MATLAB
- Entries within a row are separated by spaces or commas
- Rows are separated by semicolons

Array Examples

```
>> a = [1 2 3 4 5 ]
a =
     1     2     3     4     5
>> b = [2;4;6;8;10]
b =
     2
     4
     6
     8
    10
```

- Note 1 - MATLAB does not *display* the brackets
- Note 2 - if you are using a monospaced font, such as Courier, the displayed values should line up properly

Matrices

A 2-D array, or matrix, of data is entered row by row, with spaces (or commas) separating entries within the row and semicolons separating the rows:

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

Useful Array Commands

- The transpose operator (apostrophe) can be used to flip an array over its own diagonal. For example, if `b` is a row vector, `b'` is a column vector. If `b` is complex `b'` contains the complex conjugate of `b`.

Accessing Array Entries, 1

- Individual entries within an array can be both read and set using either the *index* of the location in the array or the row and column.
- The index value starts with 1 for the entry in the top left corner of an array and increases down a column - the following shows the indices for a 4 row, 3 column matrix:

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

Accessing Array Entries, 2

Assuming some matrix C:

```
C =
   2       4       9
   3       3      16
   3       0       8
  10      13      17
```

`C(2)` would report 3

`C(4)` would report 10

`C(13)` would report an error!

Entries can also be access using the row and column:

`C(2,1)` would report 3

`C(3,2)` would report 0

`C(5,1)` would report an error!

Array Creation - Built In

- There are several built-in functions to create arrays:

  `zeros(r,c)` will create an `r` row by `c` column matrix of zeros.

  `zeros(n)` will create an `n` by `n` matrix of zeros.

  `ones(r,c)` will create an `r` row by `c` column matrix of ones.

  `ones(n)` will create an `n` by `n` matrix one ones.

- `help elmat` has, among other things, a list of the elementary matrices

Array Creation - Colon Operator

- The colon operator : is useful in several contexts. It can be used to create a linearly spaced array of points using the notation `start:diffval:limit` where `start` is the first value in the array, `diffval` is the difference between successive values in the array, and `limit` is the *boundary* for the last value (though not necessarily the last value).

```
>>1:0.6:3
ans =
    1.0000    1.6000    2.2000    2.8000
```

Colon Operator - Notes

If `diffval` is omitted, the default value is 1:

```
>>3:6
ans =
     3     4     5     6
```

To create a decreasing series, `diffval` must be negative:

```
>> 5:-1.2:2
ans =
    5.0000    3.8000    2.6000
```

If `start+diffval>limit` for an increasing series or `start+diffval<limit` for a decreasing series, an empty matrix is returned:

```
>>5:2
ans =
   Empty matrix: 1-by-0
```

To create a column, transpose the output of the colon operator, not the limit value; that is, `(3:6)'` not `3:6'`

Array Creation - `linspace`

- To create a row vector with a specific number of linearly spaced points between two numbers, use the `linspace` command.
- `linspace(x1, x2, n)` will create a linearly spaced array of `n` points between `x1` and `x2`

```
>>linspace(0, 1, 6)
ans =
   0   0.2000   0.4000   0.6000   0.8000   1.0000
```

- If `n` is omitted, 100 points are created.
- To generate a column, transpose the output of the `linspace` command.

Character Strings & Ellipsis

Alphanumeric constants are enclosed by apostrophes (')

```
  >> f = 'Miles ';
  >> s = 'Davis'
```

Concatenation: pasting together of strings

```
  >> x = [f s]
  x =
  Miles Davis
```

Ellipsis (...): Used to continue long lines

```
  >> a = [1 2 3 4 5 ...
6 7 8]
a =
     1     2     3     4     5     6     7     8
```

You cannot use an ellipsis within single quotes to continue a string. But you can piece together shorter strings with ellipsis

```
  >> quote = ['Any fool can make a rule,' ...
' and any fool will mind it']
quote =
Any fool can make a rule, and any fool will mind it
```

Some Useful Character Functions

Function	Description
`n = length(s)`	Number of characters, `n`, in a string, `s`.
`b = strcmp(s1,s2)`	Compares two strings, `s1` and `s2`; if equal returns true (`b = 1`). If not equal , returns false (`b = 0`).
`n = str2num(s)`	Converts a string, `s`, to a number, `n`.
`s = num2str(n)`	Converts a number, `n`, to a string, `s`.
`s2 = strrep(s1,c1,c2)`	Replaces characters in a string with different characters
`i = strfind(s1,s2)`	Returns the starting indices of any occurrences of the string `s2` in the string `s1`
`S = upper(s)`	Converts a string to upper case
`s = lower(S)`	Converts a string to lower case

Mathematical Operations

- Mathematical operations in MATLAB can be performed on both scalars and arrays.
- The common operators, in order of priority, are:

^	Exponentiation	$4 \mathbin{^\wedge} 2 = 8$
−	Negation (unary operation)	$-8 = -8$
* /	Multiplication and Division	$2 * \text{pi} = 6.2832$ $\text{pi}/4 = 0.7854$
\	Left Division	$6\backslash 2 = 0.3333$
+ −	Addition and Subtraction	$3 + 5 = 8$ $3 - 5 = -2$

Within each precedence level, operators are evaluated from left to right

Order of Operations

The order of operations is set first by parentheses, then by the default order given above, then left-to-right:

`y = -4 ^ 2` gives `y = -16`
since the exponentiation happens first due to its higher default priority, but

`y = (-4) ^ 2` gives `y = 16`
since the negation operation on the 4 takes place first because of the parentheses

`y = 8/2*6` gives `y = 24`
since the left-to-right rule implements the division first to give 4 and then the multiplication to give 4*6 to yield 24

Vector-Matrix Calculations

MATLAB can also perform operations on vectors and matrices.

The * operator for matrices is defined as the *outer product* or what is commonly called “matrix multiplication.”

- The number of columns of the first matrix must match the number of rows in the second matrix.
- The size of the result will have as many rows as the first matrix and as many columns as the second matrix.
- The exception to this is multiplication by a 1 by 1 matrix, which is actually an array operation.

The ^ operator for matrices results in the matrix being matrix-multiplied by itself a specified number of times.

Note - in this case, the matrix must be square!

Element-by-Element Calculations

At times, you will want to carry out calculations item by item in a matrix or vector. The MATLAB manual calls these *array operations*. They are also often referred to as *element-by-element* operations.

MATLAB defines .* and ./ (note the dots) as the array multiplication and array division operators.

- For array operations, both matrices must be the same size *or* one of the matrices must be 1 by 1 (that is, a scalar).

Array exponentiation (raising each element to a corresponding power in another matrix) is performed with .^

- Again, for array operations, both matrices must be the same size *or* one of the matrices must be 1 by 1.

Built-In Functions

There are several built-in functions you can use to create and manipulate data.

The built-in help function can give you information about both what exists and how those functions are used:

`help elmat` will list the elementary matrix creation and manipulation functions, including functions to get information about matrices.

`help elfun` will list the elementary math functions, including trig, exponential, complex, rounding, and remainder functions.

The built-in `lookfor` command will search help files for occurrences of text and can be useful if you know a function's purpose but not its name

Graphics

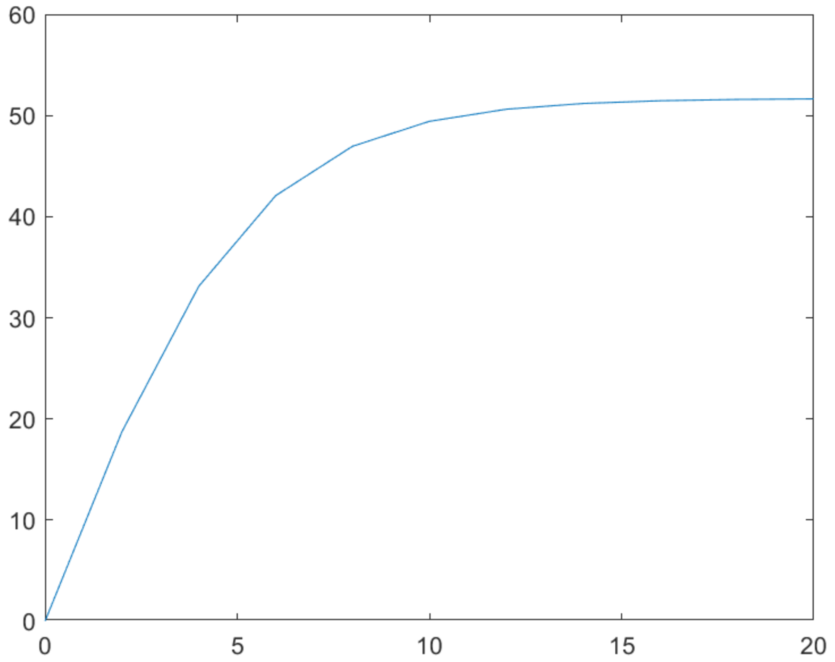
- MATLAB has a powerful suite of built-in graphics functions.

- Two of the primary functions are `plot` (for plotting 2-D data) and `plot3` (for plotting 3-D data).

- In addition to the plotting commands, MATLAB allows you to label and annotate your graphs using the `title`, `xlabel`, `ylabel`, and `legend` commands.

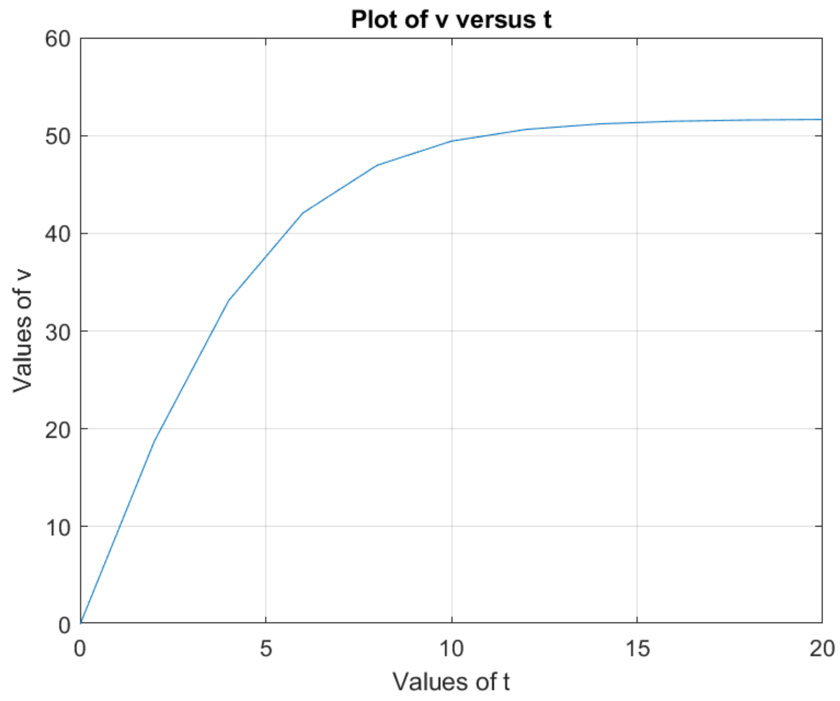
Plotting Example

```
t = [0:2:20]';
g = 9.81; m = 68.1; cd = 0.25;
v = sqrt(g*m/cd) * tanh(sqrt(g*cd/m)*t);
plot(t, v)
```



Plotting Annotation Example

```
title('Plot of v versus t')
xlabel('Values of t')
ylabel('Values of v')
grid
```



Plotting Options

When plotting data, MATLAB can use several different colors, point styles, and line styles. These are specified at the end of the `plot` command using *plot specifiers* as found in Table 2.2.

The default case for a single data set is to create a blue line with no points. If a line style is specified with no point style, no point will be drawn at the individual points; similarly, if a point style is specified with no line style, no line will be drawn.

Examples of plot specifiers:

‘ro:’ - red dotted line with circles at the points.

‘gd’ - green diamonds at the points with no line.

‘m--’ - magenta dashed line with no point symbols.

Other Plotting Commands

`hold on` and `hold off`

`hold on` tells MATLAB to keep the current data plotted and add the results of any further plot commands to the graph. This continues until the `hold off` command, which tells MATLAB to clear the graph and start over if another plotting command is given. `hold on` should be used *after* the first plot in a series is made.

`subplot(m, n, p)`

`subplot` splits the figure window into an `m×n` array of small axes and makes the p^{th} one active. Note - the first subplot is at the top left, then the numbering continues across the row. This is different from how elements are numbered within a matrix!

Programming with MATLAB

M-files

- Whereas commands can be entered directly to the command window, MATLAB also allows you to put commands in text files called *M-files*. *M-files* are so named because the files are stored with an `.m` extension.
- There are two main kinds of M-file

  Script files.

  Function files.

Script Files

- A *script file* is merely a set of MATLAB commands that are saved on a file - when MATLAB runs a script file, it is as if you typed the characters stored in the file on the command window.
- Scripts can be executed either by typing their name (without the .m) in the command window, by selecting the `Debug`, `Run` (or `Save and Run`) command in the editing window, or by hitting the `F5` key while in the editing window. Note that the latter two options will save any edits you have made, while the former will run the file as it exists on the drive.

Function Files

- *Function files* serve an entirely different purpose from script files. Function files can accept input arguments from and return outputs to the command window, but variables created and manipulated within the function do not impact the command window.
- They are commonly used to perform useful tasks that you might want to use again in other M-files.

Function File Syntax

The general syntax for a function is:

```
function outvar = funcname(arglist)
% helpcomments
statements
outvar = value;
```

where

`outvar`: output variable name.

`funcname:` function's name.

`arglist:` input argument list; comma-delimited list of values that the function requires to execute properly.

`helpcomments:` text to show with ***`help funcname`***.

`statements:` MATLAB commands for the function.

Subfunctions

- A function file can contain a single function, but it can also contain a *primary function* and one or more *subfunctions*
- The primary function is whatever function is listed first in the M-file - its function name should be the same as the file name.
- Subfunctions are listed below the primary function. Note that they are *only* accessible by the main function and subfunctions within the same M-file and *not* by the command window or any other functions or scripts.

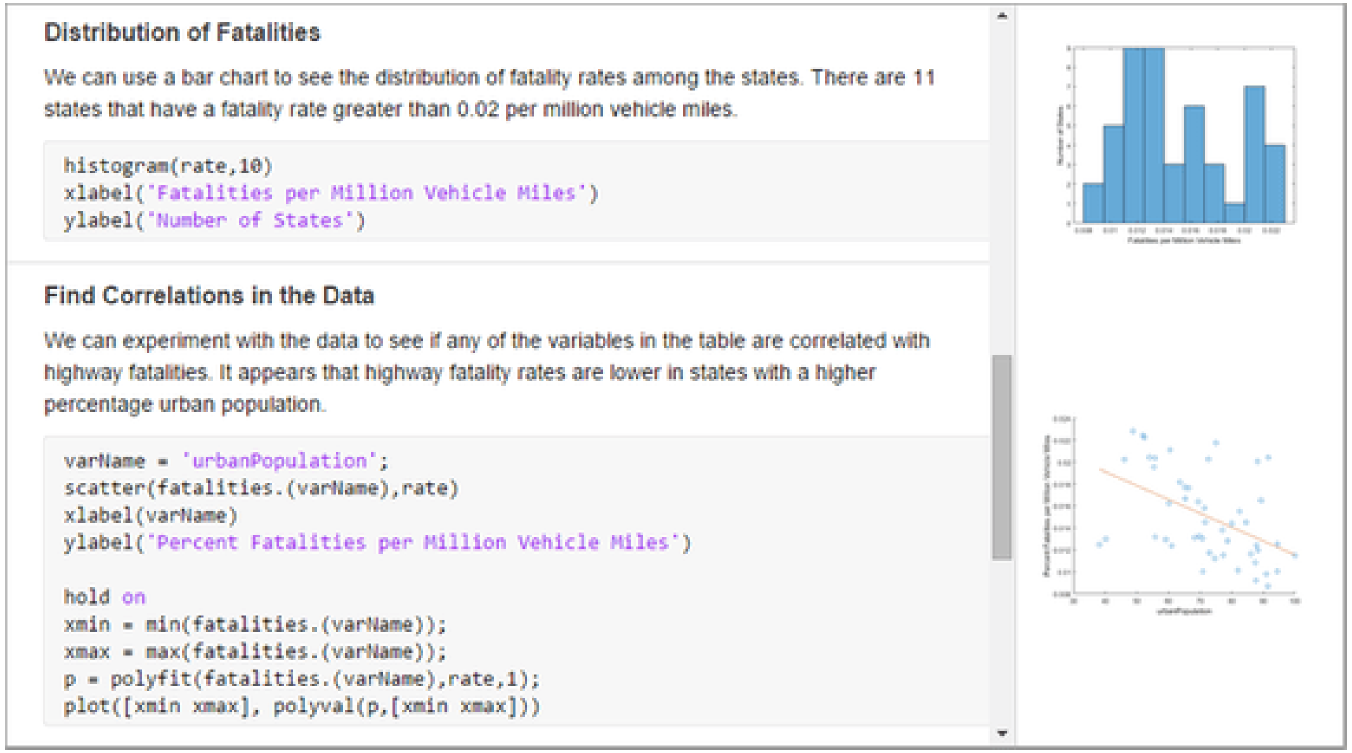
MATLAB live scripts and live functions

- Interactive documents that combine MATLAB code with formatted text, equations, and images in a single environment called the Live Editor.
- Live scripts store and display output alongside the code that creates it.

https://www.mathworks.com/help/matlab/matlab_prog/what-is-a-live-script-or-function.html#bvg6yjm-2

Visually explore and analyze problems

- Write, execute, and test code in a single interactive environment.
- Run blocks of code individually (live scripts only) or as a whole file, and view the results and graphics with the code that produced them.

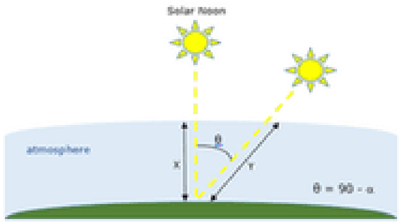


https://www.mathworks.com/help/matlab/matlab_prog/what-is-a-live-script-or-function.html#bvg6yjm-2

Share richly formatted, executable narratives

- Add titles, headings, and formatted text to describe a process and include equations, images, and hyperlinks as supporting material.
- Save your narratives as richly formatted, executable documents and share them with colleagues or the MATLAB community, or convert them to HTML files, PDF files, Microsoft® Word documents, LaTeX files, Markdown files, and Jupyter® notebooks for publication.

Air Mass and Solar Radiation



The larger the air mass, the less radiation reaches the ground. The air mass can be calculated from the equation

$$AM = \frac{1}{\cos(90-\alpha) + 0.5057(6.0799+\alpha)^{-1.6364}}.$$

Then the solar radiation (in Kw/m^2) reaching the ground can be calculated from the empirical equation

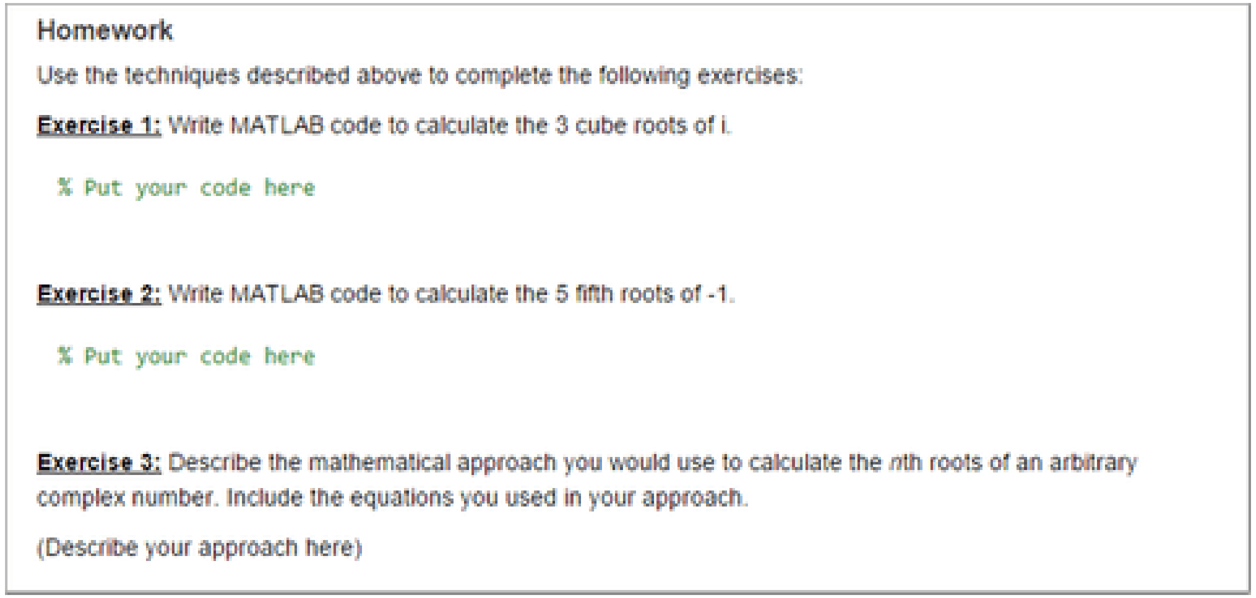
$$sRad = 1.353 * 0.7^{AM^{0.678}}.$$

```
AM = 1/(cosd(90-alpha) + 0.50572*(6.07955+alpha)^-1.6354);
sRad = 1.353*0.7^(AM^0.678);                                  % kW/m^2
disp(['Air Mass = ' num2str(AM) '   Solar Radiation = ' num2str(sRad) ' kW/m^2'])
```

https://www.mathworks.com/help/matlab/matlab_prog/what-is-a-live-script-or-function.html#bvg6yjm-2

Create interactive lectures for teaching

- Combine code and results with formatted text and mathematical equations.
- Create step-by-step lectures and evaluate them incrementally to illustrate a topic.
- Modify code on the fly to answer questions or explore related topics.
- Share lectures with students as interactive documents or in hard copy format, and distribute partially completed files as assignments.



Homework

Use the techniques described above to complete the following exercises:

Exercise 1: Write MATLAB code to calculate the 3 cube roots of i.

```
% Put your code here
```

Exercise 2: Write MATLAB code to calculate the 5 fifth roots of -1.

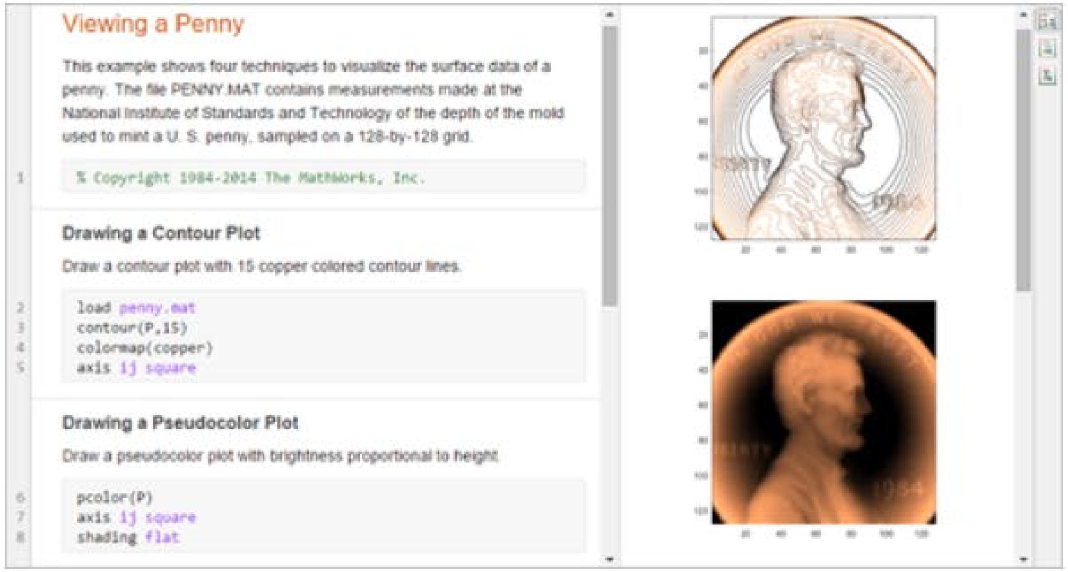
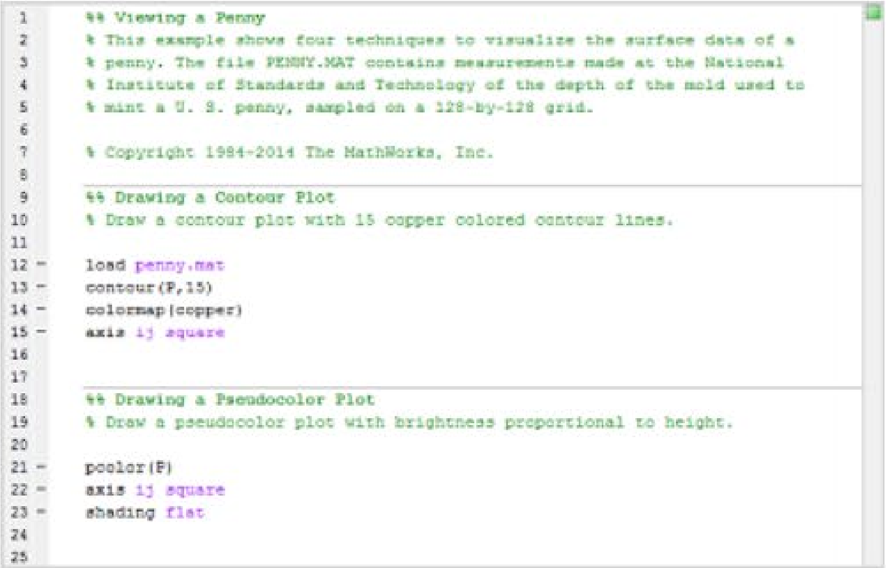
```
% Put your code here
```

Exercise 3: Describe the mathematical approach you would use to calculate the *n*th roots of an arbitrary complex number. Include the equations you used in your approach.

(Describe your approach here)

https://www.mathworks.com/help/matlab/matlab_prog/what-is-a-live-script-or-function.html#bvg6yjm-2

Differences with Plain Code Scripts and Functions

	Live Scripts and Functions	Plain Code Scripts and Functions
File Format	Live Code file format. For more information, see Live Code File Format (.mlx)	Plain Text file format
File Extension	`.mlx`	`.m`
Output Display	With code in the Live Editor (live scripts only)	In Command Window
Text Formatting	Add and view formatted text in the Live Editor	Use publishing markup to add formatted text, publish to view
Visual Representation		

https://www.mathworks.com/help/matlab/matlab_prog/what-is-a-live-script-or-function.html#bvg6yjm-2

Input

The easiest way to get a value from the user is the input command:

`n = input('promptstring')`

MATLAB will display the characters in `promptstring`, and whatever value is typed is stored in `n`. For example, if you type `pi`, `n` will store 3.1416…

`n = input('promptstring', 's')`

MATLAB will display the characters in `promptstring`, and whatever characters are typed will be stored as a string in `n`. For example, if you type `pi`, `n` will store the letters *p* and *i* in a 2×1 char array.

Output

The easiest way to display the value of a matrix is to type its name, but that will not work in function or script files. Instead, use the `disp` command

```
disp(value)
```

will show the *`value`* on the screen.

If *`value`* is a string, enclose it in single quotes. For example, `disp('Yes')` will display `Yes`

Formatted Output

For formatted output, or for output generated by combining variable values with literal text, use the *`fprintf`* command:

`fprintf('`*`format`*`', `*`x`*`, `*`y`*`,...)`

where *`format`* is a string specifying how you want the value of the variables *`x`*, *`y`*, and more to be displayed - including literal text to be printed along with the values.

The values in the variables are formatted based on format codes.

Format and Control Codes

Within the *format* string, the following format codes define how a numerical value is displayed:
%d - integer format
%e - scientific format with lowercase e
%E - scientific format with uppercase E
%f - decimal format
%g - the more compact of %e or %f

The following control codes produce special results within the *format* string:
\n - start a new line
\t - tab
\\ - print the \ character

Example of including text in a *format* string

```
fprintf('The value of pi is %8.4f \n',pi)
```

```
The value of pi is   3.1416
```

Creating and Accessing Files

MATLAB has a built-in file format that may be used to save and load the values in variables.

`save` *`filename var1 var2`* `...` *`varn`* saves the listed variables into a file named *`filename`*`.mat`. If no variable is listed, all variables are saved.

`load` *`filename var1 var2`* `...`*`varn`* loads the listed variables from a file named *`filename`*`.mat`. If no variable is listed, all variables in the file are loaded.

Note - these are not text files!

ASCII Files

To create user-readable files, append the flag `-ascii` to the end of a `save` command. This will save the data to a text file in the same way that `disp` sends the data to a screen.

Note that in this case, MATLAB does *not* append anything to the file name so you may want to add an extension such as *.txt* or *.dat*.

To load a rectangular array from a text file, simply use the *load* command and the file name. The data will be stored in a matrix with the same name as the file (but without any extension).

Structured Programming

Structured programming allows MATLAB to make decisions or selections based on conditions of the program.

Decisions in MATLAB are based on the result of logical and relational operations and are implemented with `if`, `if…else`, and `if…elseif` structures.

Selections in MATLAB are based on comparisons with a test expression and are implemented with `switch` structures.

Relational Operators

Example	Operator	Relationship
`x == 0`	`==`	Equal
`unit ~= 'm'`	`~=`	Not equal
`a < 0`	`<`	Less than
`s > t`	`>`	Greater than
`3.9 <= a/3`	`<=`	Less than or equal to
`r >= 0`	`>=`	Greater than or equal to

Logical Operators

`~x` (Not): true if `x` is false (or zero); false otherwise

`x & y` (And): true if both `x` and `y` are true (or non-zero)

`x || y` (Or): true if either `x` or `y` are true (or non-zero)

Order of Operations

- Priority can be set using parentheses. After that, Mathematical expressions are highest priority, followed by relational operators, followed by logical operators. All things being equal, expressions are performed from left to right.
- *Not* is the highest priority logical operator, followed by *And* and finally *Or*
- Generally, do not combine two relational operators!
  If `x`=5, 3<`x`<4 should be false (mathematically), but it is calculated as an expression in MATLAB as:
  `3<5<4`, which leads to `true<4` at which point `true` is converted to `1`, and `1<4` is true!
- Use `(3<x)&(x<4)` to properly evaluate.

Decisions

Decisions are made in MATLAB using `if` structures, which may also include several `elseif` branches and possibly a catch-all `else` branch.

Deciding which branch runs is based on the result of *conditions* which are either true or false.

- If an `if` tree hits a *true* condition, that branch (and that branch only) runs, then the tree terminates.
- If an `if` tree gets to an `else` statement without running any prior branch, that branch will run.

Note - if the *condition* is a matrix, it is considered true if and only if all entries are true (or non-zero).

Selections

Selections are made in MATLAB using switch structures, which may also include a catch-all otherwise choice.

Deciding which branch runs is based on comparing the value in some test expression with values attached to different cases.

- If the test expression matches the value attached to a case, that case's branch will run.
- If no cases match and there is an `otherwise` statement, that branch will run.

Loops

Another programming structure involves loops, where the same lines of code are run several times. There are two types of loop:

- A `for` loop ends after a specified number of repetitions established by the number of columns given to an index variable.
- A `while` loop ends on the basis of a logical condition.

`for` Loops

One common way to use a `for…end` structure is:

```
for index = start:step:finish
  statements
end
```

where the *`index`* variable takes on successive values in the vector created using the : operator.

Vectorization

Sometimes, it is more efficient to have MATLAB perform calculations on an entire array rather than processing an array element by element. This can be done through *vectorization*.

<table>
<tr><th><code>for</code> loop</th><th>Vectorization</th></tr>
<tr><td><pre>i = 0;
for t = 0:0.02:50
  i = i + 1;
  y(i) = cos(t);
end</pre></td><td><pre>t = 0:0.02:50;
y = cos(t);</pre></td></tr>
</table>

`while` Loops

A **while loop** is fundamentally different from a **for loop** since while loops can run an indeterminate number of times. The general syntax is

```
while condition
  statements
end
```

where the `condition` is a logical expression. If the `condition` is true, the `statements` will run and when that is finished, the loop will again check on the `condition`.

Note - though the `condition` may become false as the `statements` are running, the only time it matters is after all the statements have run.

Early Termination

Sometimes it will be useful to break out of a **for** or **while** loop early - this can be done using a break statement, generally in conjunction with an if structure.

Example:

```
x = 24
while (1)
  x = x - 5
  if x < 0, break, end
end
```

will produce x values of 24, 19, 14, 9, 4, and -1, then stop.

`continue` Command

The **continue command** jumps to the loop's `end` statement and then back to the loop's initial statement (*`for`* or *`while`*) allowing the loop to continue until the completion condition is met.

Example:

```
for i = 1:100
 if mod (i, 17)~=0
  continue
  end
  disp([num2str(i)‘ is evenly divisible by 17’])
end
```

will display all the numbers up to 100 that are evenly divisible by 17.

Nesting and Indentation

Structures can be placed within other structures. For example, the `statements` portion of a `for` loop can be comprised of an `if...elseif...else` structure.

For clarity of reading, the `statements` of a structure are generally indented to show which lines of controlled are under the control of which structure.

Anonymous & Inline Functions

Anonymous functions are simple one-line functions created without the need for an M-file

```
fhandle = @(arg1, arg2, ...) expression
```

Inline functions are essentially the same as anonymous functions, but with a different syntax:

```
fhandle = inline('expression', 'arg1', 'arg2',...)
```

Anonymous functions can access the values of variables in the workspace upon creation, while inline functions cannot.

Function Functions

Function functions are functions that operate on other functions which are passed to it as input arguments. The input argument may be the handle of an anonymous or inline function, the name of a built-in function, or the name of a M-file function.

Using function functions will allow for more dynamic programming.