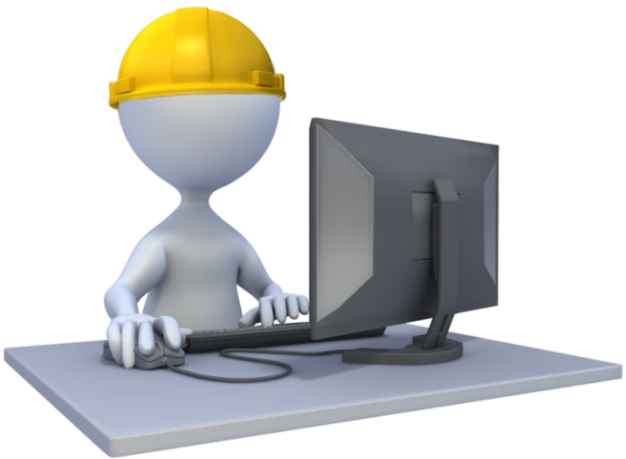
Computational Method (MATLAB Programming)

Chul Min Yeum
Associate Professor
Civil and Environmental Engineering
University of Waterloo, Canada



UNIVERSITY OF WATERLOO
FACULTY OF ENGINEERING

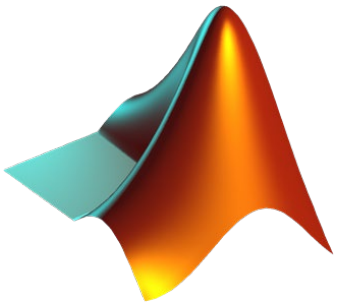
Last update: May 05, 2025

Table of Contents

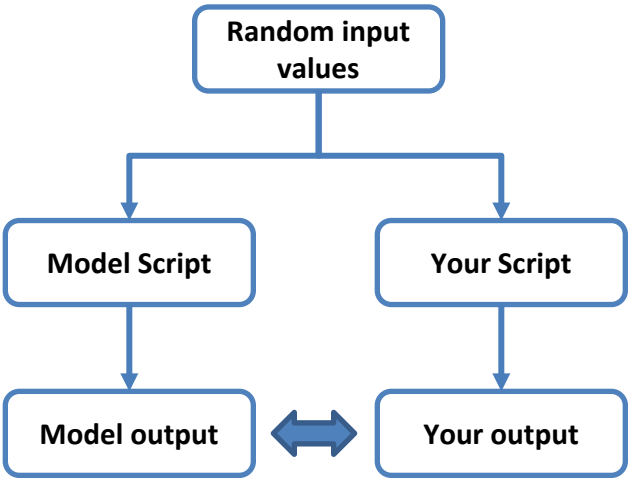
What & Why is MATLAB?

MATLAB is a high-level technical computing language and interactive environment for algorithm development, data visualization, data analysis, and numeric computation.

- Very powerful software package
- Many mathematical and graphical applications
- Has programming constructs
- Also has many built-in functions
- Can use interactively in the Command Window, or write your own programs
- Easy to debug your program
- Built-in Command Window
  - At the prompt, enter a command or expression
  - MATLAB will respond with a result



MATLAB Grader



The random number generator is to avoid your hard-coding in your assignments.

MATLAB® Grader™ is a browser-based environment for creating and sharing MATLAB coding problems and assessments. It's an auto-grading system.

An instructor designs several testers (assessments) to check your script if

- Correct outputs are generated from the given random inputs;
- Variables are properly defined;
- Keywords (e.g., built-in function) are present or absent.

Conventions Used in This Course Material

- **bold**, *Italic*, Red
  Used for highlighting or introducing a new concept, terms, or structure
- `Constant width`
  Refer to program elements such as variable or functions names used in MATLAB scripts

⚠: This element indicates a warning or caution.

☺: This element signifies a tip or suggestion.

📖 : This element signifies a general note.

Topic (placed at next to a slide title)

- Remind
  Math formula or concept
- Optional
  Optional topic or tip
- Challenging
  Challenging subjects

Conventions Used in This Course Material (Continue)

Editor or Script Window

```
% variable = expression
a1 = 3
a2 = 5
```

```
1  % variable = expression
2  a1 = 3
3  a2 = 5
4
```

Example level

 : Easy

 : Moderate

 : Difficult

Workspace

Name	Value
a1	3
a2	5

Command window

```
>> a

a =

     1
```

The above script is simplified as:

```
a        1
```

Navigating This Course Material

- This course material will be distributed as “images” in a PDF format so that students cannot copy scripts. Typing and running scripts yourself in MATAB is a crucial process to learn programming. Do not skim the code.

- Solve problems by yourself first and then review model solutions.

- Please do not jump into future modules because each module is built up from the prior module(s).

References

- Stormy Attaway, 2018, Matlab: A Practical Introduction to Programming and Problem Solving, 5th edition
- Lecture slides for “Matlab: A Practical Introduction to Programming and Problem Solving”
- Holly Moore, 2018, MATLAB for Engineers, 5th edition

Acknowledgement

- Pallavi Ahir (W25)
- Seth Bailey (S24)
- Ehtan Woo (S22), Violet Cottom (S22)
- Jesse St. John – Parker (W22)
- Bianca Angheluta (S21), Kasturi Ghosh (S21)
- Noreen Gao (S20), Vlad Fierastrau (S20)
- Jason Connelly (S19), Juan Park (S19)

Module 01:
Basic MATLAB Programming

Chul Min Yeum
Associate Professor
Civil and Environmental Engineering
University of Waterloo, Canada

UNIVERSITY OF WATERLOO
FACULTY OF ENGINEERING

Module 1: Intended Learning Outcomes

- Write an assignment statement to define a variable and assign value(s).
- Define a valid variable name acceptable in MATLAB
- Describe data types and a typecasting process.
- Explain numeric, logical operator, relational operators and compose expression using these operators
- Evaluate the expression with multiple operators following to an operator precedence in MATLAB
- Assess some basic math built-in functions (e.g., `round(x)`, `sign(x)`, `or abs(x)`)
- Generate random number(s) using a built-in random number function

Variables and Assignments

- To store a value, use a ***variable***
- A way to put a value in a variable is with an ***assignment statement***
- General form:

variable = expression

- The order is important:
  - *variable* name on the left
  - the assignment operator "="
  - *expression* on the right

⚠: Here, = is not meant to an equality.

⚠: Make sure that the variable name is always on the left.

```
% variable = expression
a1 = 3
a2 = 5
```

Name	Value
a1	3
a2	5

Variable Names

- Names must begin with a letter of the alphabet.
- After that names can only contain letters, digits, and the underscore character(_).
- Variable names must not have a space.
- You cannot use other characters except for '_'.
- MATLAB is case-sensitive.
- Names should be mnemonic: You and others know what are stored in your variables through its name.
- **`clear`** is a function to clear out variables and also functions.

Example: Variable Names

```
8val = 10; % error: must begin with a letter of the alphabet

_col = 0 ; % error: must begin with a letter of the alphabet

row_3_ = 1; % no error

row@3 = 10; % error: cannot contain characters other than underscore

col-03 = 10; % error: cannot contain characters other than underscore

% Following scripts have no error but the names should be mnemonic

asdf1 = 100; % no error
love = 10; % no error
aaaa3 = 10; % no error
```

⚠: Recommend mnemonic variable names

```
% define a variable of 'ga1'
ga1 = 100
```

Q. What value is in 'Ga1' ?

Example: Question in S19 Midterm

Q. Which of the following scripts have errors?

(1)	`Val1 = 10`
(2)	`4val = 5`
(3)	`new@data = 8`
(4)	`val*2 = 3`
(5)	`Jason = 10+2`

1. (1), (2), and (4)
2. (2), (3), and (5)
3. (2), (3), and (4)
4. (1), (2), and (5)

Constants

- In programming, variables, as the name suggests, store the values that **could be changed**.
- ***Constants*** are used when the value is **pre-defined** and not updated in the program.
- Examples in MATLAB

  - **`pi`** 3.14159….
  - **`i, j`** imaginary number
  - **`inf`** infinity
  - **`NaN`** stands for "not a number"; e.g. the result of 0/0

⚠: However, the constants are also variables so you can overwrite values to the constants, but I **do not** recommend the use of constants as variables. For example: `pi = 3 % no error`

Modifying Variables

```
myvar = 10; % initialize a variable
myvar = myvar + 3; % increment by 3
```

Name	Value
myvar	13

```
myvar = 10; % initialize a variable
val = 3; % initialize a variable

% identical opertaions
myvar1 = myvar + val;
myvar2 = 10 + 3;
```

Name	Value
myvar	10
val	3
myvar1	13
myvar2	13

Swap two values

```
myvar1 = 10; % initialize a var.
myvar2 = 5; % initialize a var.

tmp = myvar1;
myvar1 = myvar2;
myvar2 = tmp;
```

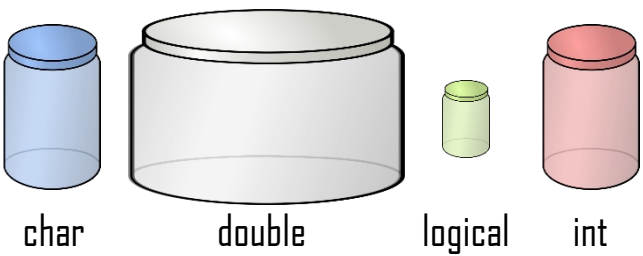
Name	Value
myvar1	5
myvar2	10
tmp	10

Data Types

- Every expression and variable has an associated *type* or *class*
  - Real numbers: *single*, *double* (**default type for numbers**)
  - Integer types: numbers in the names are the number of bits used to store a value of that type
    - Signed integers: *int8, int16, int32, int64*
    - Unsigned integers: *uint8, uint16, uint32, uint64*
  - Single characters and character vectors: *char*
  - Strings of characters: *string*
  - **True/false: *logical (represented as 1 or 0)***

☺: The same value can be defined as a different type.

Container



Examples

- 2.145, 0.15893, 3.0, 2.45
- 10, 11, 24, 30, 400
- ‘a’, ‘b’, ‘A’, ‘c’
- “sam”
- true, false

Data Types (Continue)

Example

- double : 2.145, 0.15893, 3.0,
- integer type: 10, 11, 24, 30
- char: ‘a’, ‘b’, ‘A’, ‘c’
- string: “sam”
- logical: true, false

`class` is a function of determining a class of variables or values.

```
>> class("sam")

ans =
    'string'
```

```
>> class('a')

ans =
    'char'

>> class(true)

ans =
    'logical'

>> class(2.145)

ans =
    'double'

>> class(10)

ans =
    'double'
```

⚠: The default type of numbers is ‘`double`’. Integers likes `10, 11, ..` are also defined as ‘`double`’. If you want to define these number as integer, you should do type-casting.

Type Cast

Challenging

There are many functions that convert values from one type to another. The names of these functions are the same as the names of the types. The names can be used as functions to convert a value to that type. This is called casting the value to a different type or type casting.

```
>> a = logical(1);
>> class(a)

ans =

    'logical'
>> b = double(a);
>> class(b)

ans =

    'double'
```

```
>> a = logical(1);
>> b = a + 2;
>> class(b)

ans =

    'double'
>> c = logical(b)
c =

  logical

   1
```

📖: In the second line in red, the type of `a` is converted to double and arithmetic operation is conducted with `2`. In the fourth line, `b` is double but the logical type only carries `0` or `1` so non-zero double (or other numeric type) values becomes `1`, otherwise `0`. This is a very important concept for logical operations.

Example: Type Cast (logical and double Types)

Q. What value is assigned to each variable? What is its type?

```
a = 3;
b = logical(a);
c = logical(1);
d = logical(0);

e = 1 + a;
f = 1 + double(b);
g = a - b;
h = logical(3 - a);
k = b - c;
```

Name	Value	Class
a	3	double
b	1	logical
c	1	logical
d	0	logical
e	4	double
f	2	double
g	2	double
h	0	logical
k	0	double

📖: For the variable of `k`, you cannot add or subtract values with a logical type. To conduct the operation, their types are implicitly changed to double types.

Operators

- There are in general two kinds of operators: *unary* operators, which operate on a single value and *binary* operators, which operate on two values.
- Operators include:
  - **+** addition
  - **–** subtraction or negation
  - ***** multiplication
  - **/** division
  - **^** exponentiation (e.g., 5^2 is 25)

```
val1 = 5 + 1;
val2 = 10*2;
val3 = 10^2;
val4 = 100/5;
val5 = 10 + 2 + 3;
```

Name	Value
val1	**6**
val2	**20**
val3	**100**
val4	**20**
val5	**15**

Operator Precedence Rules

- Some operators have precedence over others.
- Within given precedence, the expressions are evaluated from **left** to **right**.
- Precedence list (highest to lowest) :

`( )`	parentheses
`^`	exponentiation
`-`	negation
`*, /, \`	all multiplication and division
`+, -`	addition and subtraction

- Nested parentheses: expressions in inner parentheses are evaluated first

```
val1 = (5 + 1)*2;
val2 = 10^2*3;
val3 = 10^(2+3);
val4 = 4-3*2;
val5 = 3*((4+3)*2);
val6 = -10^2;
```

☺: A good practice is to use parentheses to clarify your operation: `-(10^2)` or `(-10)^2`.

Name	Value
`val1`	**12**
`val2`	**300**
`val3`	**100000**
`val4`	**-2**
`val5`	**42**
`val6`	**-100**

Example: Operator Precedence Rules

Q. What values are assigned to the variables?

```
val1 = 3*2+3-2*3
val2 = -10^4*3
val3 = (-10)^4*3
val4 = 5-3*2;
val5 = (3+3)^(2-1);
```

Name	Value
val1	3
val2	-30000
val3	30000
val4	-1
val5	6

Relational Operator

- The relational **operators** in MATLAB are:

>	greater than
<	less than
>=	greater than or equals
<=	less than or equals
==	equality
~=	inequality

⚠: Remember, **=** does not mean equality, which is used in the assignment statement. **==** is an equality operator

- It is also called *Boolean* expressions or *logical* expressions.
- **The resulting type is logical 1 for true or 0 for false**

📖 : "true" is represented by the logical value 1, and "false" is represented by the logical value 0. A logical type only contain one value, either 0 or 1.

Relational Operator (Continue)

```
% relation operator
ro1 = 3 < 4;
ro2 = 3 > 5;
ro3 = 3 == 5;
ro4 = 3 ~= 7;
ro5 = 3 <= 3;
ro6 = 3 >= 3;
ro7 = 3 > 3;
```

Name	Value
ro1	1
ro2	0
ro3	0
ro4	1
ro5	1
ro6	1
ro7	0

📖: In the second line, the code is first running the expression (`3<4`) and computing its value. This expression means "3 less than 4", which is a true. Thus, `true` (or logical `1`) is assigned to `ro1`.

⚠: Here, the values are all *logical* values, `0` or `1`, not *double*.

Logical Operator

Challenging

- The logical **operators** are:

`||`	or
`&&`	and
`~`	not

- Note that the logical operators are **commutative**
  - (e.g., `x || y` is equivalent to `y || x`)
- The resulting type is **`logical`** 1 for true or 0 for false

x	y	~x	x || y	x && y
true	true	false	true	true
true	false	false	true	false
false	false	true	false	false

📖: For example, `logical (1) && logical (1)` becomes `1`. This is the same with `true && true`.

Logical Operator (Continue)

```
% relation operator
lo1 = true && false
lo2 = 1 || 0
lo3 = 0 || 0
lo4 = true && true
lo5 = 0 || ~false
lo6 = ~false && true
lo7 = ~true || false
```

Name	Value
lo1	**0**
lo2	**1**
lo3	**0**
lo4	**1**
lo5	**1**
lo6	**1**
lo7	**0**

x	y
true	true
true	false
false	false

~x	x || y	x && y
false	true	true
false	true	false
true	false	false

General Operator Precedence

Challenging

Parentheses `()`
Power: `^`
Unary: negation (−), not (~)
Multiplication or division: `*`, `/`, `\`
Addition, subtraction: +, −
Relational: `<`, `<=`, `>`, `==`, `~=`
And: `&&`
Or: `||`

Highest

Lowest

```
val1 = 3 < (1 + 3)
val2 = (3 < 1) + 3
val3 = 3 < 1 + 3
```

⚠: Although there is no error for computing `val3`, this syntax is not recommended to avoid a potential mistake.

☺: I know it's bit hard but, it might be easy if you consider relational and logical operators as common operators like + , - and compute the values with this precedence order.

📖 : **Remember ! 0 is `false` otherwise `true`.**

Name	Value	Class
`val1`	**1**	**`logical`**
`val2`	**3**	**`double`**
`val3`	**1**	**`logical`**

Example: Operator Precedence

```
lg1 = (3 < 4) < 4;
lg2 = 3 < (4 < 5);
lg3 = (3 > 5) + 3;
lg4 = (10 > 4) && (4 > 1);
lg5 = (10 < 4) && (4 < 1);
lg6 = ~((10 < 4) && (4 < 1));
lg7 = 2 < 3 + 4;
```

Parentheses `()`
Power: `^`
Unary: negation (−), not (`~`)
Multiplication or division: `*`, `/`, `\`
Addition, subtraction: `+`, −
Relational: `<`, `<=`, `>`, `==`, `~=`
And: `&&`
Or: `||`

Highest

Lowest

Q. What values are assigned to the variables?

Name	Value
`lg1`	**1**
`lg2`	**0**
`lg3`	**3**
`lg4`	**1**
`lg5`	**0**
`lg6`	**1**
`lg7`	**1**

☺: Regardless of the operator precedence, you could use parentheses to clarify the operation order

Example: General Operator Precedence

Q. How to write a code to check if `x` lies in between `5` and `10`. If yes, `1` and otherwise `0`.

```
x1 = 6;
x2 = 11;

lg1 = (5 < x1) && (x1<10)
lg2 = 5 < x1 <10; % incorrect!
lg3 = (5 < x1) <10; % incorrect!

lg4 = (5 < x2) && (x2 < 10)
lg5 = 5 < x2 < 10; % incorrect!
lg6 = (5 < x2) < 10; % incorrect!
```

Name	Value
x1	6
x2	11
lg1	1
lg2	1
lg3	1
lg4	0
lg5	1
lg6	1

⚠: For `lg2`, the first expression `5<x` will be evaluated. It gives a logical value 1. Then, the rest of the expression will be evaluated, `1 < 10`. So, the final value to be assigned to `lg2` become logical 1, `true`.

Using Functions: Terminology

- To use a function, you ***call*** it
- To call a function, give its name followed by the ***argument(s)*** that are ***passed*** to it in parentheses:

```
out = fun(arg1, arg2,…)
```

- Functions basically calculate values and ***return*** the results:
- For example, to find the absolute value of `-4`

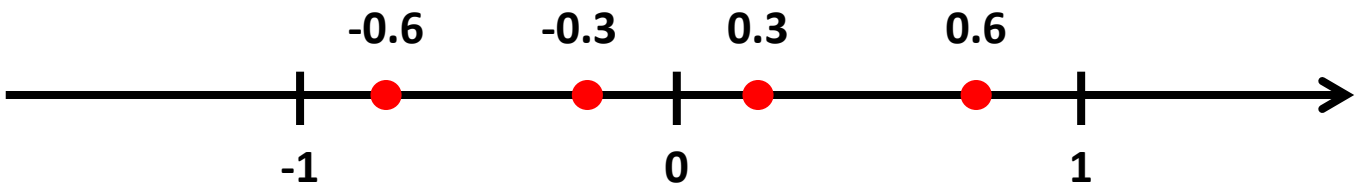
```
>> abs(-4)
ans =
     4
```

- The name of the function is "`abs`"
- One argument, `-4`, is passed to the `abs` function
- The `abs` function finds the absolute value of `-4` and returns the result, `4`.

Rounding Functions

Function	Description	Note
`round(x)`	Rounds `x` to the nearest integer	
`fix (x)`	Truncates `x` to the nearest integer toward zero.	
`floor(x)`	Rounds `x` to the nearest integer toward negative infinity.	
`ceil(x)`	Rounds `x` to the nearest integer toward positive infinity.	

Rounding Functions (Continue)



Function	Description
`round(x)`	Rounds `x` to the nearest integer
`fix (x)`	Truncates `x` to the nearest integer toward zero.
`floor(x)`	Rounds `x` to the nearest integer toward negative infinity.
`ceil(x)`	Rounds `x` to the nearest integer toward positive infinity.

	round	fix	floor	ceil
0.3	0	0	0	1
0.6	1	0	0	1
-0.3	0	0	-1	0
-0.6	-1	0	-1	0

Rounding Functions (Continue)

```
x1 = 10.3;
x2 = 12.7;
x3 = -1.3;

x1_ce = ceil(x1);
x1_fi = fix(x1);
x1_fl = floor(x1);

x2_ce = ceil(x2);
x2_fi = fix(x2);
x2_fl = floor(x2);

x3_ce = ceil(x3);
x3_fi = fix(x3);
x3_fl = floor(x3);
x3_ro = round(x3);
```

Name	Value
x1	**10.3**
x2	**12.7**
x3	**-1.3**
x1_ce	**11**
x1_fi	**10**
x1_fl	**10**
x2_ce	**13**
x2_fi	**12**
x2_fl	**12**
x3_ce	**-1**
x3_fi	**-1**
x3_fl	**-2**
x3_ro	**-1**

Function	Description
round(x)	Rounds **x** to the nearest integer
fix (x)	Truncates **x** to the nearest integer toward zero.
floor(x)	Rounds **x** to the nearest integer toward negative infinity.
ceil(x)	Rounds **x** to the nearest integer toward positive infinity.

☺: You do not have to memorize the functions and their usage. You can simply search for their usage in google or type `doc round` in command window (`doc fun_name`).

Common Math Functions

Function	Description	Script	Value
`abs(x)`	Finds the absolute value of `x`	`abs(-3)` `abs(2)`	`3` `2`
`sqrt(x)`	Finds the square root of `x`	`sqrt(4)` `sqrt(1.75)`	`2` `1.5`

`sign(x)`	Return `-1` if `x` is less than zero, a value of `0` if `x` equals zero, and a value of `1` if `x` is greater than zero.	`sign(-5)`
		`sign(3)`
		`sign(0)`
			`-1`
			`1`
			`0`

`rem(x,y)`	Computes the remainder of `x/y`	`rem(25,4)`
		`rem(4,2)`
		`rem(9,5)`
			`1`
			`0`
			`4`

Common Math Functions (Continue)

Function	Description
abs(x)	Finds the absolute value of **x**
sqrt(x)	Finds the square root of **x**
sign(x)	Return **-1** if **x** is less than zero, a value of **0** if **x** equals zero, and a value of **1** if **x** is greater than zero.
rem(x,y)	Computes the remainder of **x/y**

```
x = -4;
y = 9;
z = 2;

x_ab = abs(x)
x_si = sign(x)

xz_r = rem(x, z)

y_ab = abs(y)
y_sq = sqrt(y)
y_si = sign(y)

yz_r = rem(y, z)
```

Name	Value
x	**-4**
y	**9**
z	**2**
x_ab	**4**
x_si	**-1**
xz_r	**0**
y_ab	**9**
y_sq	**3**
y_si	**1**
yz_r	**1**

Logical Operation Functions

Function	Operator
`and(A, B)`	A && B
`or(A, B)`	A || B
`not(A)`	~A

☺: These functions improve readability.
For instance:

```
log1 = 3<x && x<10

log2 = and(3<x, x<10)
```

```
x1 = true;
x2 = false;

lg1a = and(x1, x2);
lg1b = x1 && x2;

lg2a = or(x1, x2);
lg2b = x1 || X2

lg3a = not(x1);
lg3b = ~x1;
```

Name	Value
x1	**1**
x2	**0**
lg1a	**0**
lg1b	**0**
lg2a	**1**
lg2b	**1**
lg3a	**0**
lg3b	**0**

Random Number

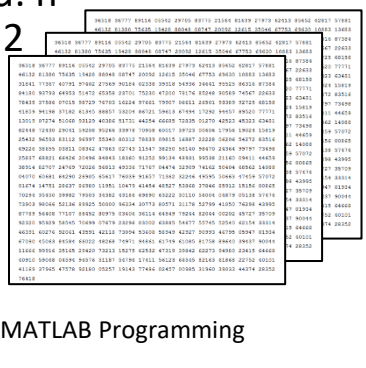
Challenging

- MATLAB generates pseudorandom numbers. These numbers are not strictly random and independent in the mathematical sense, but they pass various statistical tests of randomness and independence, and their calculation can be repeated for testing or diagnostic purposes.
- Several built-in functions generate random numbers.
- Random number generators start with a number called the `seed`. This is either a predetermined value or **from the clock**. This is considered as an "index" of a random number lookup table.

Seed: n

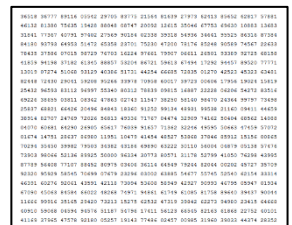
Seed: 2

Seed: 1



Seed: i

Seed: i



Random Number (Continue)

Challenging

- **`rand(n)`** creates an *nxn* matrix of random reals
- **`rand(n,m)`** create an *nxm* matrix of random reals
- **`randi([range],n,m)`** creates an *nxm* matrix of random integers in the specified range
- **`rng(seed)`** specifies the seed for the random generator

```
>> rand(1)

ans =

    0.8147

>> rand(1)

ans =

    0.9058
```

📖 : A seed is changed in each run of your script. Thus, different random numbers are generated

```
>> rng(10); rand(1)

ans =

    0.7713

>> rng(10); rand(1)

ans =

    0.7713
```

📖 : You can specify a seed. Then, the same numbers are generated.

Write a Numerical Expression

Optional

Suppose that you want to compute y when x = 10

$$y = \frac{\{x^2(100x + 10) + x^3(20x^2 + 3)\}}{-x^{-3} + 1}$$

```
x = 10;
y1 = (x^2*(100*x + 10) + x^3*(20*x^2 + 3))/(-x^(-3)+1)
```

☺: The more terms and parentheses in your equation, the larger the probability that you may be making a mistake. In this case, I usually do it like this

```
x = 10;
y_nom = (x^2)*(100*x + 10) + (x^3)*(20*(x^2) + 3);
y_den = -x^(-3)+1;
y = y_nom/y_den;
```

Arithmetic Operation in MATLAB

Optional

Suppose that you are solving a problem: If the car has a mass of 300kg and you push the car with an acceleration of 5 inch/s^2, compute the force that is generated from the car in newton

```
% MATLAB as a calculator
300*5*0.0254
```

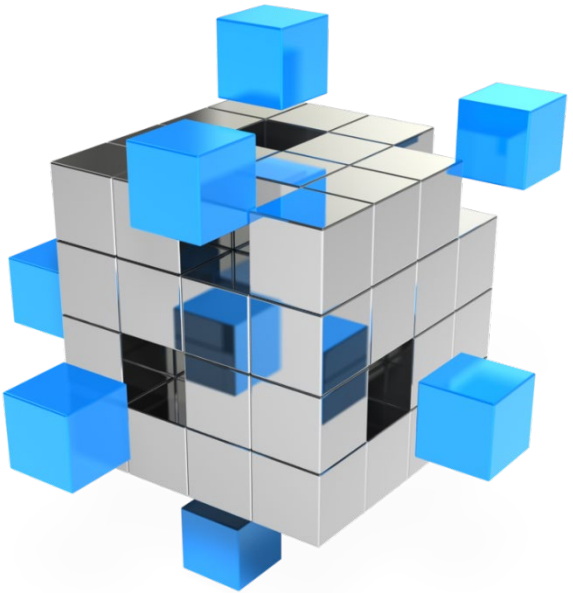
Name	Value
ans	38.1

☺: If you are using MATLAB as a programming tool, I would recommend writing the code below. This improves code readability and allows others to understand your code.

```
% MATLAB as programming tool

inch2m = 0.0254; % inch to m
mass = 300; % kg
accel = 5; % inch/s/s

% force  = mass(kg) * acceleration (m/s/s)
force = mass * accel * inch2m;
```

Name	Value
inch2m	0.0254
mass	300
accel	5
force	38.1

Module 02: Vectors and Matrices



Chul Min Yeum
Associate Professor
Civil and Environmental Engineering
University of Waterloo, Canada

UNIVERSITY OF WATERLOO
FACULTY OF ENGINEERING

Module 2: Intended Learning Outcomes

- Create an array in MATLAB, which includes a scalar, vector, and matrix.
- Refer elements in an array using index or subscript.
- Modify and concatenate an array.
- Perform array operations (element-wise operation and matrix operation)
- Define and evaluate a character and character vector
- Create a 3D matrix and refer to its element.
- Understand a linear indexing method

Matrix, Row Vector, and Column Vector

Remind

A matrix is an ordered rectangular array of numbers. A general matrix with *m* rows and *n* columns has the following structure:

$$\begin{array}{c} \\ \text{row, } i \end{array} \begin{array}{c} \text{column, } j \\ \begin{array}{c} 1 \\ 2 \\ \vdots \\ m \end{array} \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & \cdots & n \end{array} \\ \begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ \vdots & \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \dots & \dots & a_{mn} \end{bmatrix}_{m \times n} \end{array}$$

where a_{ij} is each number in the array indexed by its row position i and column position j.

Matrix, Row Vector, and Column Vector in MATLAB

- A ***matrix*** is used to store a set of values of the same type; every value is stored in an ***element.***
- A matrix looks like a table; it has both rows and columns
- A matrix with m rows and n columns is called ***m x n***; these are called its ***dimensions***;
- A ***vector*** is a special case of a matrix in which one of the dimensions is 1
- The term ***array*** is frequently used in MATLAB to refer generically to a matrix or a vector
  - a row vector with n elements is `1 x n`, e.g., `1 x 4`:
  - a column vector with m elements is `m x 1`, e.g. , `3 x 1`:
- A ***scalar*** is an even more special case; it is `1 x 1`, or in other words, just a single value

Creating Row Vectors

- Direct method: put the values you want in square brackets, separated by either **commas** or **spaces**
- **Colon operator**: iterates through values in the form `first:step:last`
  - `2:1:4` creates the vector of `[2 3 4]`.
  - If no step is specified, the default is `1` so for example `2:4` creates the vector `[2 3 4]`
  - Can go in reverse e.g. `4:-1:1` creates `[4 3 2 1]`
  - Cannot go beyond last e.g., `1:2:6` creates `[1 3 5]`

```
rvec1 = [1 2 3 4]; % separated by space
rvec2 = [1,2,3,4]; % separated by comma
rvec3 = [1   2   3 4];
rvec4 = 1:4; % use a colon operator
rvec5 = 1:1:4; % step by 1
rvec6 = [1:4]; % okay with putting bracket
```

Name	Value
rvec1	[1 2 3 4]
rvec2	[1 2 3 4]
rvec3	[1 2 3 4]
rvec4	[1 2 3 4]
rvec5	[1 2 3 4]
rvec6	[1 2 3 4]

Creating Row Vectors (Continue)

```
rvec1 = [1 2 3 4 5 6];
rvec2 = [1:6];

rvec3 = [1:2:6]; % step by 2

rvec4 = [1:3:6]; % step by 3

rvec5 = [6:-1:1]; % step by -1
```

Name	Value
rvec1	[1 2 3 4 5 6]
rvec2	[1 2 3 4 5 6]
rvec3	[1 3 5]
rvec4	[1 4]
rvec5	[6 5 4 3 2 1]

```
my_rvec1 = [1:5:6];

my_rvec2 = [5:-2:1];
```

Name	Value
rvec1	[1 6]
rvec2	[5 3 1]

📖: **Remember !** `variable = first:step:last`

Transpose

Remind

The **transpose** of a matrix is the operation of flipping the rows to columns and vice versa across the diagonal of the matrix. For example,

$$A = \left[\begin{array}{c|c} 1 & 4 \\ \hline 2 & 5 \\ \hline 3 & 6 \end{array}\right]_{3\times 2} \quad \text{has transpose} \quad A^T = \left[\begin{array}{c:c:c} 1 & 2 & 3 \\ \hline 4 & 5 & 6 \end{array}\right]_{2\times 3}$$

where the power of *T* indicates the transpose of a matrix. Note that in the example, the dimensions of the matrix changes from 3x2 to 2x3 (with the same total number of elements) and diagonal elements remain the same.

Creating Column Vectors

- A **column** vector is an *m x 1* vector
- **Direct method**: can create by separating values in square brackets with ***semicolons*** (e.g., `[4; 7; 2]`)
- You cannot directly create a column vector using methods such as the colon operator, but you can create a row vector and then ***transpose*** it to get a column vector using the transpose operator (e.g., `[4 7 2]'`)

```
cvec1 = [1;2;3;4];
cvec2 = [1 2 3 4]';
rvec3 = [1:4];
cvec4 = rvec3';
```

Name	Value
cvec1	**[1; 2; 3; 4]**
cvec2	**[1; 2; 3; 4]**
rvec3	**[1 2 3 4]**
cvec4	**[1; 2; 3; 4]**

Referring to Elements

- The elements in a vector are numbered sequentially; each element number is called the ***index***, or ***subscript*** :

```
myvec = [5 33 11 -4 2];
```

Index	1	2	3	4	5
Element	5	33	11	-4	2

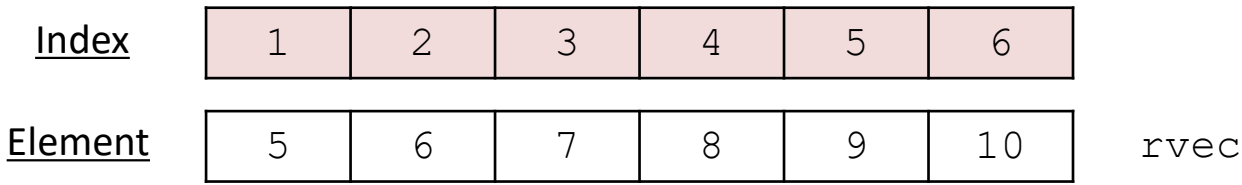
- Refer to an element using its ***index*** or ***subscript*** in parentheses, e.g. `vec(4)` is the 4th element of a vector `vec` (assuming it has at least 4 elements)
- Can also refer to a subset of a vector by using an ***index vector*** which is a vector of indices e.g. `vec([2 5])` refers to the 2nd and 5th elements of `vec`.

⚠: **The index in MATLAB starts from 1 !!**

Example: Referring to Elements

```
rvec = [5:10];
cvec = rvec';

val1 = rvec(2);
val2 = rvec(5);

vec1 = rvec([1 4]);
vec2 = rvec([1 2 6]);

vec3 = cvec([1 4]);
vec4 = cvec([1 2 6]);
```

Name	Value
rvec	[5 6 7 8 9 10]
cvec	[5; 6; 7; 8; 9; 10]
val1	6
val2	9
vec1	[5 8]
vec2	[5 6 10]
vec3	[5; 8]
vec4	[5; 6; 10]



Modifying Vectors

- Elements in a vector can be changed.

variable(index) = expression

```
rvec = 1:4;
rvec1 = rvec;
rvec1(4) = 10;
```

Name	Value
rvec	[1 2 3 4]
rvec1	[1 2 3 10]

📖 : Value(s) computed from *expression* are assigned to *variable* at *index* location(s).

```
rvec = 1:5;
rvec1 = rvec;
rvec2 = rvec;
rvec3 = rvec;

rvec1([1 3]) = [0 0];
rvec2(1:3) = 4:6;
rvec3([1 3]) = rvec3([3 1]);
```

Name	Value
rvec	[1 2 3 4 5]
rvec1	[0 2 0 4 5]
rvec2	[4 5 6 4 5]
rvec3	[3 2 1 4 5]

Modifying Vectors (Continue)

- A vector **can** be extended by referring to elements that do not yet exist
- A vector **cannot** be read by an index that does not yet exist

variable(index) = expression

```
rvec1 = 1:4;
rvec1(5) = 5;

rvec2 = 1:4;
rvec2([5 6]) = [3 2];
```

Name	Value
rvec1	[1 2 3 4 5]
rvec2	[1 2 3 4 3 2]

```
rvec1 = 1:4;
val1 = rvec1(5)
```

Error: Index exceeds the number of array elements (4)

```
rvec1 = 1:4;
rvec1(1:2) = [1 2 3];
```

Error: Unable to perform assignment because the left and right sides have a different number of elements.

Concatenation

- Vectors can be created by joining existing vectors together, or adding elements to existing vectors
- This is called ***concatenation***

```
rvec = [1 2];
rvec1 = [rvec 8 9];
rvec2 = [rvec rvec1];


cvec = [4;5];
cvecp = [6;7]
cvec1 = [cvec; cvecp];
```

Name	Value
rvec	[1 2]
rvec1	[1 2 8 9]
rvec2	[1 2 1 2 8 9]
cvec	[4; 5]
cvecp	[6; 7]
cvec1	[4; 5; 6; 7]

```
rvec = [1 2];
rvec1 = [rvec; 8];
```

Error: using vertcat
Dimensions of arrays being concatenated are not consistent.

Creating a Matrix

- Separate values within rows with ***blanks*** or ***commas***, and separate the rows with ***semicolons***
- Can use any method to get values in each row (any method to create a row vector, including colon operator)
- **There must <u>ALWAYS</u> be the same number of values in every row!!**

```
m1 = [1 2 3;4 5 6];

r1 = [1 2 3];
r2 = [4 5 6];

m2 = [r1;r2];
m3 = [r1;4 5 6];
% it works not recommend
```

Name	Value
m1	**[1 2 3;4 5 6]**
r1	**[1 2 3]**
r2	**[4 5 6]**
m2	**[1 2 3;4 5 6]**
m3	**[1 2 3;4 5 6]**

m1, m2, m3

1	2	3
4	5	6

Functions that Create Matrices

There are many built-in functions to create matrices

- **`zeros(n)`** creates an *nxn* matrix of all zeros
- **`zeros(n,m)`** creates an *nxm* matrix of all zeros
- **`ones(n)`** creates an *nxn* matrix of all ones
- **`ones(n,m)`** creates an *nxm* matrix of all ones
- **`eye(n)`** creates an *nxn* identity matrix.

⚠: **`zeros(n)`** or **`ones(n)`** is a matrix, not a column or row vector.

```
m1 = zeros(3);
m2 = zeros(3,2);

m3 = ones(2,1);
m4 = eye(3);
```

Name	Value
m1	`[0 0 0;0 0 0;0 0 0]`
m2	`[0 0; 0 0; 0 0]`
m3	`[1;1]`
m4	`[1 0 0; 0 1 0; 0 0 1]`

m1

0	0	0
0	0	0
0	0	0

m2

0	0
0	0
0	0

m3

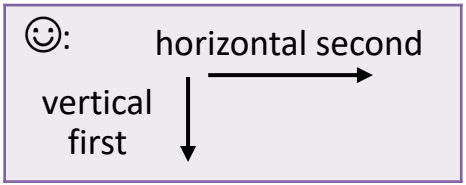
1
1

m4

1	0	0
0	1	0
0	0	1

Matrix Elements

- To refer to an element in a matrix, you use the matrix variable name followed by ***the index of the row, and then the index of the column***, in parentheses
- ALWAYS refer to the row first, column second
- This is called ***subscripted indexing***.



Variable (row indexes, column indexes)

```
m1 = [1 2 3;4 5 6];
```

1	2	3
4	5	6

Element

(1,1)	(1,2)	(1,3)
(2,1)	(2,2)	(2,3)

Indexing

Matrix Elements (Continue)

- You can index and refer to an entire row or column using a colon (:).
- Can also refer to any subset of a matrix
  - To refer to the entire m^{th} row: `m1(m,:)`
- To refer to the entire n^{th} column: `m1(:,n)`
- To refer to the last row or column use **`end`**, (e.g. `m1(end,m)` is the m^{th} value in the last row)
- Can modify an element or subset of a matrix in an assignment statement

Variable (row indexes, column indexes)

Used for **referring values** and **reading values**

Example: Matrix Elements

```
m1 = [1 2 3;4 5 6;7 8 9];

c1 = m1(:,1);
r1 = m1(2,:);

c2 = m1(:,end);
c3 = m1(1:3, end);
c4 = m1(1:end, end);

r2 = m1(end,:);
r3 = m1(3,:);

m2 = m1(2:3, 2:3);
m3 = m1(2:end, 2:end);
m4 = m1([1 3], [1 3]);
```

m1

1	2	3
4	5	6
7	8	9

c1

1
4
7

r1

4	5	6

c2

3
6
9

c3

3
6
9

c4

3
6
9

r2

7	8	9

r3

7	8	9

m2

5	6
8	9

m3

5	6
8	9

m4

1	3
7	9

Modifying Matrices

- An individual element in a matrix can be modified by assigning a new value to it.
- Entire rows and columns can also be modified.
- Any subset of a matrix can be modified, as long as what is being assigned has the same dimensions as the subset being modified.
- An exception to this: a scalar can be assigned to a subset of any size; the same scalar is assigned to every element in the subset.

Variable (row indexes, column indexes) = Expression

Used for **referring** **elements** and **reading** **values**

Example: Modifying Matrices

```
>> m1 = [1 2 3;4 5 6;7 8 9];
>> m2 = m1
m2 =

     1     2     3
     4     5     6
     7     8     9

>> m2(1,1) = 10
m2 =

    10     2     3
     4     5     6
     7     8     9

>> m2(1,end) = 30
m2 =

    10     2    30
     4     5     6
     7     8     9
```

… continue …

```
>> m2(:,1) = [10; 10; 10]
m2 =

    10     2    30
    10     5     6
    10     8     9

>> m2(end,:) = zeros(1, 3)
m2 =

    10     2    30
    10     5     6
     0     0     0

>> m2(3,:) = m2(1,:)
m2 =

    10     2    30
    10     5     6
    10     2    30
```

Example: Modifying Matrices (Continue)

… continue …

```
>> m2(:,1) = [10; 10; 10];
m2 =

    10     2    30
    10     5     6
    10     8     9

>> m2(end,:) = zeros(1, 3);
m2 =

    10     2    30
    10     5     6
     0     0     0

>> m2(2,:) = 1;
m2 =

    10     2    30
     1     1     1
     0     0     0
```

… continue …

```
>> m2(:, end) = 3;
m2 =

    10     2     3
     1     1     3
     0     0     3

>> m2([1 3], [1 3]) = 10;
m2 =

    10     2    10
     1     1     3
    10     0    10
```

📖: Exception to this: a scalar can be assigned to any size subset; the same scalar is assigned to every element in the subset.

Example: Modifying Matrices

Q: Swap the 2nd and 3rd columns in `mat1`

```
mat1 = [1 2 3; 4 5 6; 7 8 9];

% write your code here
vec = mat1(:, 2);

mat1(:,2) = mat1(:,3);
mat1(:,3) = vec;
```

```
mat1 = [1 2 3; 4 5 6; 7 8 9];

mat1(:, [3 2]) = mat1(:, [2 3]);
```

1	2	3
4	5	6
7	8	9

mat1

1	3	2
4	6	5
7	9	8

mat1

Matrix Dimension

- There are several functions to determine the dimensions of a vector or matrix:
  - `size` returns the # of rows and columns for a vector or matrix
  - Important: capture both output values in an assignment statement

    ```
    [r, c] = size(mat)
    ```
  - `numel` returns the total # of elements in a vector or matrix

⚠: Very important to generalize your script: do not assume that you know the dimensions of a vector or matrix – use `size` or `numel` to find out!

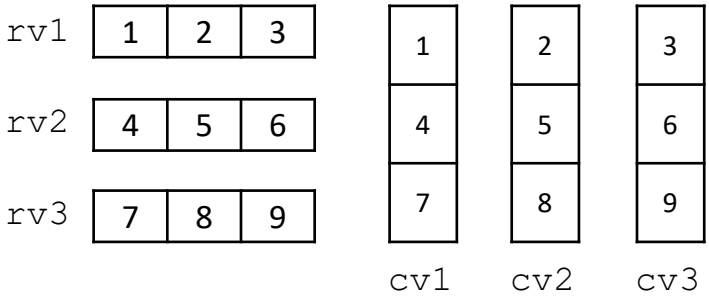
```
m1 = [1 2 3;4 5 6];

[r1, c1] = size(m1);

n_m1 = numel(m1);
```

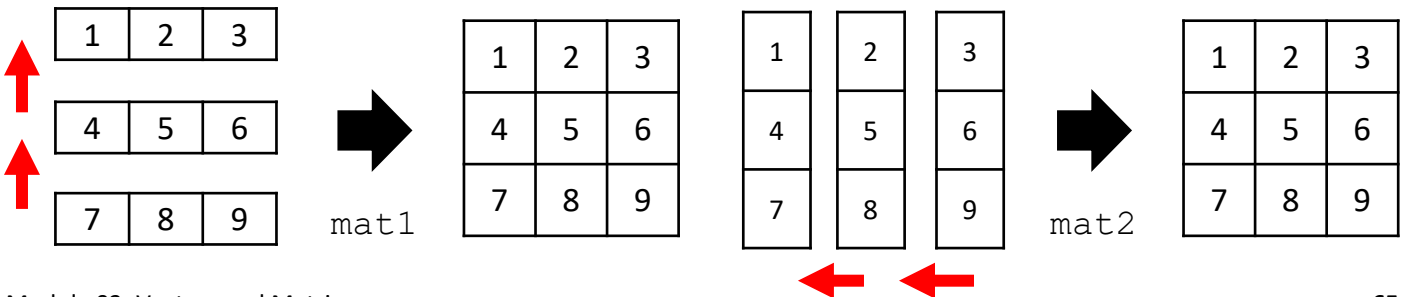
Name	Value
m1	[1 2 3;4 5 6]
r1	2
c1	3
n_m1	6

Matrix Concatenation

```
rv1 = [1 2 3];
rv2 = [4 5 6];
rv3 = [7 8 9];

mat1 = [rv1; rv2; rv3];

cv1 = [1; 4; 7];
cv2 = [2; 5; 8];
cv3 = [3; 6; 9];

mat2 = [cv1 cv2 cv3]
```

Matrices can be created by joining, vectors or adding elements.





Empty Vectors and Matrices

- An ***empty vector*** is a vector with no elements; an empty vector can be created using square brackets with nothing inside [ ]
- To **delete** an element from a vector, assign an empty vector to that element
- Delete an entire row or column from a matrix by assigning [ ]
  - Note: **cannot delete an individual element from a matrix**

```
>> v1 = 2:6
v1 =

     2     3     4     5     6

>> v1(3) = []
v1 =

     2     3     5     6

>> v1(end) = []
v1 =

     2     3     5
```

```
>> m1 = [1 2 3;4 5 6;7 8 9]
m1 =

     1     2     3
     4     5     6
     7     8     9

>> m1(2,:) = []
m1 =

     1     2     3
     7     8     9
```

Scalar Operations

- Numerical operations can be performed on every element in a vector or matrix
- For example, ***Scalar multiplication:*** multiply every element by a scalar

```
>> v1 = [4   0   11] *  3
v1 =

    12      0     33
```

- Another example: ***Scalar addition***; add a scalar to every element

```
>> v2 = zeros(1,3) + 5

v2 =

     5      5      5
```

Array Operations

Array operations on two matrices `A` and `B`.

- These are applied term-by-term, or element-by-element
- In MATLAB:
  - matrix addition: `A + B`
  - matrix subtraction: `A - B` or `B - A`
- For operations that are based on multiplication (multiplication, division, and exponentiation), a dot must be placed in front of the operator
  - array (element-wise) multiplication: `A .* B`
  - array (element-wise) division: `A ./ B`, `A .\ B`
  - array (element-wise) exponentiation `A .^ 2`
- **Matrix multiplication: NOT an array operation:** `A .* B` is not equal to `A * B`.

Example: Array Operations

```
m1 = [1 2 3; 4 5 6; 7 8 9];
m2 = [1 1 1; 2 2 2; 1 1 1];
sv = 3;

mat1 = m1 + m2;
mat2 = m1 .* m2;
mat3 = m1 - m2;
mat4 = m1 ./ m2;

mat5 = m1 + sv;
mat6 = m1 * sv;
```

m1

1	2	3
4	5	6
7	8	9

m2

1	1	1
2	2	2
1	1	1

sv

3

mat1 =

1	2	3
4	5	6
7	8	9

+

1	1	1
2	2	2
1	1	1

=

2	3	4
6	7	8
8	9	10

mat2 =

1	2	3
4	5	6
7	8	9

.*

1	1	1
2	2	2
1	1	1

=

1	2	3
8	10	12
7	8	9

mat3 =

1	2	3
4	5	6
7	8	9

–

1	1	1
2	2	2
1	1	1

=

0	1	2
2	3	4
6	7	8

mat4 =

1	2	3
4	5	6
7	8	9

./

1	1	1
2	2	2
1	1	1

=

1	2	3
2	2.5	3
7	8	9

mat5 =

4	5	6
7	8	9
10	11	12

mat6 =

3	6	9
12	15	18
21	24	27

Matrix Multiplication: Dimensions

- In MATLAB, the multiplication operator * performs matrix multiplication.
- To multiply a matrix A by a matrix B, the number of columns of A must be the same as the number of rows of B.
- If the matrix `A` has dimension of *m x n*, this means that matrix `B` must have dimensions of *n x something*;
  - In mathematical notation, `[A]`*m x n* `[B]`*n x p*
  - We say that the ***inner dimensions*** must be the same
- The resulting matrix `C` has the same number of rows as `A` and the same number of columns as `B`
  - in other words, the ***outer dimensions*** *m x p*
  - In mathematical notation, `[A]`*m x n* `[B]`*n x p* = `[C]`*m x p*.

⚠: Matrix multiplication is NOT an array operation. It does NOT mean multiplying term by term (element by element)

Matrix Times a Vector

Remind

A linear system of equations with coefficient matrix A, variable vector $\vec{x}$ and constant term vector $\vec{b}$, can be expressed as

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ \vdots & \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \dots & \dots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

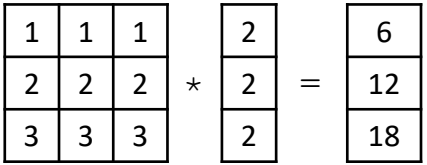
$$\overset{\text{matrix}}{(m \times n)} \overset{\text{vector}}{(n \times 1)} = \overset{\text{vector}}{(m \times 1)}$$

$$b_i = \sum_{j}^{n} x_j a_{ij}$$

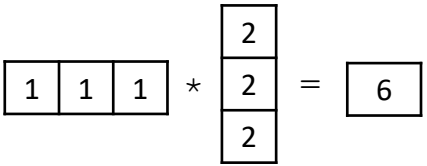
Compatibility – the number of columns in the matrix **must** equal the number of rows in the vector.

Example: Matrix Times a Vector

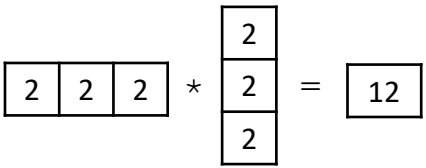
```
m1 = [1 1 1; 2 2 2; 3 3 3];
v1 = [2 2 2]';

c1 = m1*v1;

c11 = m1(1,:)*v1;
c21 = m1(2,:)*v1;
c31 = m1(3,:)*v1;

c2 = [c11;c21;c31];
```



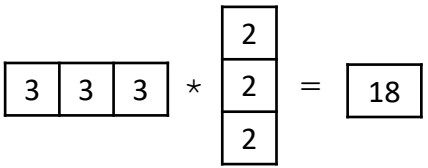
m1 * v1 = c1



m1(1,:) * v1 = c11



m1(2,:) * v1 = c21



m1(3,:) * v1 = c31

Matrix Times a Matrix

Remind

Matrix multiplication is an extension of matrix and vector multiplication. Consider the product of an $m \times n$ matrix A, and an $n \times p$ matrix B:

$$AB = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ b_{31} & b_{32} & \cdots & b_{3p} \\ \vdots & \vdots & & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{bmatrix}$$

$$\overset{\text{matrix}}{(m \times n)} \overset{\text{matrix}}{(n \times p)} = \overset{\text{matrix}}{(m \times p)}$$

Compatibility – the number of columns in the first matrix **must** equal the number of rows in the second matrix.

Example: Matrix Times a Matrix

```
m1 = [1 2;3 4];
m2 = [1 2;2 1];

m3 = m1*m2;

m411 = m1(1,:)*m2(:,1);
m421 = m1(2,:)*m2(:,1);
m412 = m1(1,:)*m2(:,2);
m422 = m1(2,:)*m2(:,2);

m4 = [m411 m412;m421 m422];
```

1	2
3	4

*

1	2
2	1

=

5	4
11	10

m1 * m2 = m3

⚠: * and .* are different!

1	2

*

1
2

=

5

m1(1,:) * m2(:,1) = m411

3	4

*

1
2

=

11

m1(2,:) * m2(:,1) = m421

1	2

*

2
1

=

4

m1(1,:) * m2(:,2) = m412

3	4

*

2
1

=

10

m1(2,:) * m2(:,2) = m422

Example: Matrix Times a Matrix or Vector

Q. Write a script to compute `A, B, C, D` using `m1` and `v1`.

$$\text{m1} = \begin{bmatrix} 2 & 4 \\ 3 & 2 \\ 5 & 3 \end{bmatrix} \qquad \text{v1} = \begin{bmatrix} 1 \\ 3 \\ 1 \end{bmatrix}$$

$$A = \begin{bmatrix} 1 & 3 & 1 \end{bmatrix} \begin{bmatrix} 2 & 4 \\ 3 & 2 \\ 5 & 3 \end{bmatrix}$$

$$B = \begin{bmatrix} 2 & 3 & 5 \\ 4 & 2 & 3 \end{bmatrix} \begin{bmatrix} 1 \\ 3 \\ 1 \end{bmatrix} \qquad C = \begin{bmatrix} 2 & 3 & 5 \end{bmatrix} \begin{bmatrix} 1 \\ 3 \\ 1 \end{bmatrix}$$

$$D = \begin{bmatrix} 1 & 3 \end{bmatrix} \begin{bmatrix} 2 & 3 & 5 \\ 4 & 2 & 3 \end{bmatrix}$$

```
m1 = [2 4; 3 2; 5 3];
v1 = [1; 3; 1];

A = v1'*m1

B = m1'*v1

C = m1(:,1)'*v1

D = v1(1:2)'*m1'
```

📖 : `A'` is a transpose of `A`.

Character and Character Vector

Challenging

- A ***character*** is a single character in single quotes
- A character vector is sequences of characters in single quotes, e.g. `'hello and how are you?'`
- **A character vector is a vector in which every element is a single character.**

```
ch_vec = 'MATLAB';

n_ch = numel(ch_vec);

ch3 = ch_vec(3);
ch4 = ch_vec(1:3);

ch_vec1 = [ch_vec ' is fun.']
```

Name	Value
ch_vec	'MATLAB'
n_ch	6
ch3	'T'
ch4	'MAT'
ch_vec1	'MATLAB is fun.'

Character Type Casting

Challenging

- The numeric type cast function can also convert a character to its equivalent numeric value.
- All characters are mapped to equivalent numeric values.

```
ch1 = 'a';
ch2 = 'b';
ch3 = 'd';

ch_vec1 = 'abc';
```

☺: This will be very useful for character operations.
www.asciitable.com

`double(ch1)`	`97`
`char(97)`	`'a'`
`char(98)`	`'b'`
`double('d')`	`100`
`char([97 99])`	`'ac'`
`'ab' + 1`	`[98 99]`
`'cd' - 1`	`[98 99]`
`char('cd'-1)`	`'bc'`
`ch_vec1 + 1`	`[98 99 100]`
`char(ch_vec1 + 1)`	`'bcd'`

3D Matrices

Challenging

- A three-dimensional matrix has dimensions of `m x n x p`
- For example, we can create a `3 x 4 x 2` matrix of random integers; there are 2 layers, each of which is a `3 x 4` matrix

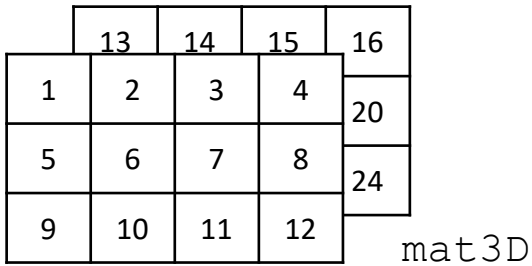
```
mat3D = zeros(3, 4, 2);

mat1 = [1 2 3 4; 5 6 7 8; 9 10 11 12];
mat2 = [13 14 15 16; 17 18 19 20; 21 22 23 24];

mat3D(:,:,1) = mat1;
mat3D(:,:,2) = mat2;
```

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

mat1

13	14	15	16
17	18	19	20
21	22	23	24

mat2



mat3D

3D Matrices (Continue)

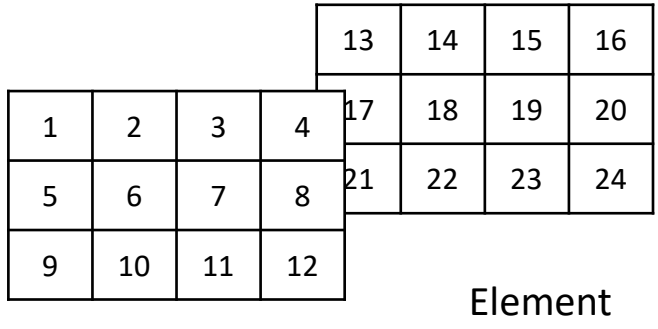
Challenging

```
val1 = mat3D(1, 3, 1);

val2 = mat3D(1, 1, 2);

mats = mat3D(1:2, 1:2, 1);
```

Name	Value
mat3D	3x4x2 *double*
val1	3
val2	13
mats	[1 2; 5 6]

📖: Large matrices are not printed out in *Workspace*.

mat3D



Linear Indexing

Challenging

- Linear indexing: only using one index into a matrix
- MATLAB will unwind it column-by column going from top to bottom.

```
mat1 = [1 5 3 7;3 2 2 4;4 2 8 9];
val1 = mat1(2, 1)
val2 = mat1(2)
val3 = mat1(3, 4)
val4 = mat1(12)

vec1 = mat1(3:5)
```

Name	Value
val1	3
val2	3
val3	9
val4	9
vec1	[4 5 2]

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

Element

(1,1)	(1,2)	(1,3)	(1,4)
(2,1)	(2,2)	(2,3)	(2,4)
(3,1)	(3,2)	(3,3)	(3,4)

Subscripted indexing

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

Linear indexing

Function: `reshape(x, sz)`

Challenging

`B = reshape(A,sz)` reshapes `A` using the size vector, `sz`, to define `size(B)`. `sz` must contain at least 2 elements, product of elements in `sz` must be the same as `numel(A)` (Note: Arrays are reshaped in a linear indexing order).

```
vec = 3:14; % 12 elements

mat0 = reshape(vec, [2 6]);
mat1 = reshape(vec, [3 4]);
mat2 = reshape(mat1, [6 2]);
```

vec

3	4	5	6	7	8	9	10	11	12	13	14

mat0

3	5	7	9	11	13
4	6	8	10	12	14

mat1

3	6	9	12
4	7	10	13
5	8	11	14

mat2

3	9
4	10
5	11
6	12
7	13
8	14

Types of Errors

Syntax error

```
a1 = [1 2 3];
3 = a1 + 2;
```

Error: Incorrect use of '=' operator. To assign a value to a variable, use '='. To compare values for equality, use '=='.

```
a@2 = [1 2 3];
```

Error: Invalid expression. Check for missing multiplication operator, missing or unbalanced delimiters, or other syntax error.

Runtime error

```
a2 =[11 2 3];
b2 = a2(4) + 1;
```

Index exceeds the number of array elements (3).

```
A = [1 2;3 4];
val1 = A(3,4);
```

Index in position 1 exceeds array bounds (must not exceed 2).

Logical error

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

m1 = A.*B;
m2 = A*B;
```

Example: Types of Errors

Q. Which of the following scripts occur run-time or syntax error?

(1)	`mat1 = ones(10,10);` `mat1(1:2, 5:10) = [];`
(2)	`mat1 = ones(10,10);` `mat1(1, 5) = [];`
(3)	`mat1 = ones(10,10);` `mat1(1, 1:2) = 3;`
(4)	`mat1 = ones(10,10);` `mat1 = [1 2; 3 4];`

1. (1), (2)
2. (1), (2), (3)
3. (1), (2), (4)
4. (3), (4)
5. (1), (3), (4)

Example (Midterm in S19)

Q. Write a code to compute `val1`, `val2`, and `mat1` using `A`,`B`,`r`, and `c`

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \\ a_{41} & a_{42} & a_{43} \end{bmatrix} \quad \boldsymbol{B} = \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \end{bmatrix} \quad \boldsymbol{c} = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} \quad \boldsymbol{r} = [r_1 \quad r_2 \quad r_3 \quad r_4]$$

$$\boldsymbol{val1} = \boldsymbol{c_1 r_1 + c_2 r_2 + c_3 r_3}$$

$$\boldsymbol{val2} = [\boldsymbol{c_1 r_1} \quad \boldsymbol{c_2 r_2} \quad \boldsymbol{c_3 r_3} \quad \boldsymbol{c_4 r_4}]$$

$$\boldsymbol{mat1} = \begin{bmatrix} b_{11}r_1 & b_{12}r_2 & b_{13}r_3 & b_{14}r_4 \\ b_{21}r_1 & b_{22}r_2 & b_{23}r_3 & b_{24}r_4 \\ b_{31}r_1 & b_{32}r_2 & b_{33}r_3 & b_{34}r_4 \end{bmatrix}$$

```
val1 = r(1:3)*c(1:3);
val2 = r.*c';

mat1 = B.*[r; r; r];
```

Module 03: Selection Statement

Chul Min Yeum
Associate Professor
Civil and Environmental Engineering
University of Waterloo, Canada

UNIVERSITY OF WATERLOO
FACULTY OF ENGINEERING

Module 3: Intended Learning Outcomes

- Describe a problem that requires if statement
- Construct an if-else statement and explain its operation
- Compare an if-else statement and its variants
- Create a script using a switch statement

If-Statement

- The **if** statement is used to determine whether or not a statement or group of statements is to be executed
- General form:

```
if condition
    action
end
```

- the *condition* is any relational expression (True or False)
- the *action* is any number of valid statements (including, possibly, just one)

If the condition is true, the action is executed – otherwise, it is skipped entirely.

Example: Relational Operator

Q. How to write a code to check if `x` lies in between `5` and `10`. If yes, assign `true` to `lg1` and otherwise `false`.

```
1
2
3
```

```
x1 = 6;

lg1 = (5 < x1) && (x1<10)
```

```
1
2
3
4
5
6
```

```
x1 = 6;
lg1 = false;

if (5 < x1) && (x1<10)
   lg1 = true
end
```

Q. How to write a code to check if `x` lies in between `5` and `10`. If yes, assign `10` to `val` and otherwise `5`.

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

```
x1 = 6;

val = 5;

if (5 < x1) && (x1<10)
   val = 10;
end
```

Example: Write a Code for Computing `abs(x)`

abs (x)	Finds the absolute value of **x**	abs (-3) abs (2)	3 2

```
1   x1 = -3
2
3   if x1>=0
4      xabs = x1;
5   end
6
7   if x1<0
8      xabs = x1*-1;
9   end
10
```

Name	Value
x1	-3
xabs	3

📖 : At line 4, `x1` is not more than `0`, and thus, the if-statement is skipped (move to line 8) . At line 8, since `x1` is less than `0`, the action (line 9) is executed. `x1*-1` is assigned to `xabs`.

```
x1 = -3
xabs = x1

if x1<0
   xabs = x1*-1;
end
```

This code also works.

Example: Write a Code for Computing `sign(x)`

`sign(x)`	Return **-1** if **x** is less than zero, a value of **0** if **x** equals zero, and a value of **1** if **x** is greater than zero.	`sign(-5)`
		`sign(3)`
		`sign(0)`
			`-1`
			`1`
			`0`

```
x1 = -3

if x1 == 0
   xsign = 0 ;
end

if x1<0
   xsign = -1;
end

if x1>0
   xsign = 1;
end
```

```
x1 = -3
xsign = 0;

if x1<0
   xsign = -1;
end

if x1>0
   xsign = 1;
end
```

Name	Value
x1	**-3**
xsign	**-1**

Example: Bulls and Cows

Bulls and Cows is a mind game played by two players. In the game, a random, 4-digit number is chosen and its values are compared to those of another trial number. **All four digits of the number are different**. If any digit in the chosen number is the exact same value and in the exact same position as any digit in the trial number, this is called a bull. If the digit is present in both the trial number and chosen number, but is not in the same location, this is called a cow.

In this figure, A: Bulls, B: Cows

Example: Bulls and Cows (Continue)

```
1   x_true = [1 2 3 4]; % true
2   x_test = [3 2 5 6]; % test
3
4   numb = 0; % number of Bull
5
6   if x_true(1) == x_test(1)
7      numb = numb + 1;
8   end
9
10  if x_true(2) == x_test(2)
11     numb = numb + 1;
12  end
13
14  if x_true(3) == x_test(3)
15     numb = numb + 1;
16  end
17
18  if x_true(4) == x_test(4)
19     numb = numb + 1;
    end
```

Q: Write a script to compute "Bull" and assign its value to `numb`. The true and test sequence are in `x_true` and `x_test`, respectively.

Name	Value
x1	**[1 2 3 4]**
x2	**[3 2 5 6]**
numb	**1**

☺: We will keep revisiting these examples later!!

If-else Statement

- The **`if-else`** statement chooses between two actions
- General form:

```
if condition
    action1
else
    action2
end
```

- ***Only one action is executed***; which one depends on the value of the condition. In the above statement, execute `action1` if `condition` is `true` or `action2` if `condition` is `not true` (`false`).

Example: If-else Statement

abs(x)	Finds the absolute value of **x**	**abs(-3)** **abs(2)**	3 2

```
x1 = -3

if x1>=0
   xabs = x1;
end

if x1<0
   xabs = x1*-1;
end
```

Name	Value
x1	-3
xabs	3

```
x1 = -3
xabs = 0

if x1>0
   xabs = x1;
else
   xabs = x1*-1;
end
```

Name	Value
x1	-3
xabs	3

☺: In this example, there is no difference in the final result. However, I recommend if-else statement because `abs(x)` is an obvious logical (binary) operation, meaning that if `action1` is performed, `action2` will not be executed.

Nested if-else Statements

- To choose from more than two actions, ***nested*** **if-else** statements can be used (an **if** or **if-else** statement as the action of another)

- General form:

```
if condition1
    action1
else
    if condition2
       action2
    else
       action3
    end
end
```

📖 : In this statement, only one action statement is executed!!

Recall the if-else statement:

```
if condition
    action1
else
    action2
end
```

Example: Nested if-else Statements

Q: If ‘scalar1’ is larger than 0 and less than 50, assign 10 to ‘out1’. Otherwise, assign 5 to ‘out1’.

```
scalar1 = 20;

if scalar1 > 0
   if scalar1<50
      out1 = 10;
   else
      out1 = 5;
   end
else
   out1 = 5;
end
```

```
scalar1 = 20;

if (scalar1 > 0) && (scalar1 < 50)
   out1 = 10;
else
   out1 = 5;
end
```

☺: In general, I avoid multiple levels of nested if-else statement. The script on the right is more readable and avoids potential mistakes.

if-else and elseif Statements

- MATLAB also has an `elseif` clause which shortens the code (and cuts down on the number of ends)
- General form:

```
1   if condition1
2       action1
3   elseif condition2
4       action2
5   elseif condition3
6       action3
7   else
8       action4
9   end
```

📖 : Again, in this statement, only one action statement must be executed!!

If `condition1` is `true`, `action1` is executed and go to `end` at line9. If `condition1` is `false`, go to `condition2` and execute `action2` if `condition2` is `true`. The same operation is performed for `condition3` and `action3`. If none of `condition1, 2, 3` is `true`, `action4` is executed.

Example: if-else and elseif Statements

Q: Write a script to determine a grade based on a score named `score`: A: score >=90, B: 80 <= score < 90, C:70 <= score < 80, D: score < 70. A character grade should be assigned to a variable named `grade`.

```
score = 81;

if score>= 90
    grade = 'A';
end
if score>=80 && score < 90
    grade = 'B';
end
if score>=70 && score < 80
    grade = 'C';
end
if score < 70
    grade = 'D';
end
```

Option 1

```
score = 81;

if score>= 90
    grade = 'A';
else
    if score >=80
        grade = 'B';
    else
        if score >=70
            grade = 'C';
        else
            grade = 'D';
        end
    end
end
```

Option 2

Example: if-else and elseif Statements (Continue)

Q: Write a script to determine a grade based on a score named `score`: A: score >=90, B: 80 <= score < 90, C:70 <= score < 80, D: score < 70. A character grade should be assigned to a variable named `grade`.

```
score = 81;

if score>= 90
    grade = 'A';
end
if score>=80 && score < 90
    grade = 'B';
end
if score>=70 && score < 80
    grade = 'C';
end
if score < 70
    grade = 'D';
end
```

Option 1

```
score =81;

if score>= 90
    grade = 'A';
elseif score >=80
    grade = 'B';
elseif score >=70
    grade = 'C';
else
    grade = 'D';
end
```

Option 3

☺: Option 3 is simpler and more readable.

Correct Use of if-else-elseif Statement

```
score =81;

if score>= 90
    grade = 'A';
elseif score >=70
    grade = 'C';
elseif score >=80
    grade = 'B';
else
    grade = 'D';
end
```

Option 3

⚠: If the order of the condition statements is changed, a total wrong result will be obtained. Here, `grade` become `'C'`, not `'B'`. Again, only one action is executed!

```
score =81;

if score>= 90
    grade = 'A';
elseif and(score < 90, score>=80)
    grade = 'B';
elseif and(score < 80, score>=70)
    grade = 'C';
else
    grade = 'D';
end
```

Option 4

☺: You can also define a clear condition statement to avoid confusion.

Correct Use of if-else-elseif Statement (Another Example)

Write a code if the value named ‘val’ is a multiples of 2, add 1 to ‘num’. If ‘val’ is multiples of 3, add additional 1 to ‘num’.

```
num = 0;
val = 6;

if rem(val, 2) == 0
   num = num + 1;
end

if rem(val, 3) == 0
   num = num + 1;
end
```

Correct

Name	Value
num	2

```
num = 0;
val = 6;

if rem(val, 2) == 0
   num = num + 1;
elseif rem(val, 3) == 0
   num = num + 1;
end
```

Wrong

Name	Value
num	1

Summary: if, if-else, if-elseif, if-elseif-else

```
if condition1
    action1
end
```

```
if condition1
    action1
else
    action2
end
```

```
if condition1
    action1
elseif condition2
    action2
end
```

```
if condition1
    action1
elseif condition2
    action2
else
    action3
end
```

How to Write a Formatted Script

- Use indentation to show the structure of a script or function for readability. Four empty spaces are indented in an action statement.
- The best way to format your script is to apply smart indenting while writing your script. Select your script and press ***Ctrl + I***.

```
x1 = -3
xabs = 0

if x1>0
   xabs = x1;
else
   xabs = x1*-1;
end
```

```
x1 = -3
xabs = 0
if x1>0
xabs = x1;
else
xabs = x1*-1;
end
```

📖 : Both are the same and working scripts. Which do you prefer?

Shorten Your Scripts

Optional, Challenge

```
x1 = -3
xabs = 0

if x1>=0
   xabs = x1;
end

if x1<0
   xabs = x1*-1;
end
```

Option 1

```
x1 = -3;
xabs = 0;

if x1>=0
    xabs = x1;
else
    xabs = x1*-1;
end
```

Option 2

abs (x)	Finds the absolute value of **x**

```
x1 = -3

xabs = (x1>=0)*x1 + -1*(x1<0)*x1
```

Option 3

📖: Here, `x1` is negative. Thus, `(x1>=0)` becomes `logical 0`. Due to intrinsic typecasting, this will be numeric 0 (type: double). Thus, the first term becomes 0. In the next term, `(x1<0)` becomes `logical 1`. Then, `-1*(x1<0)*x1` becomes `-1*x1`. Finally, `-1*x1` is assigned to `xabs`, which is the same operation using if-statement.

Switch-case Statement

- The **`switch-case`** statement can frequently be used in place of a nested **`if-else`** or **`if-elseif`** statement.
- This can be used when comparing the `switch_expression` to see if it is equal to the values on the case labels (the **`otherwise`** clause handles all other possible values)

General form:

```
switch switch_expression
  case caseexp1
    action1
  case caseexp2
    action2
  case caseexp3
    action3
  % etc: there can be many of these n actions
  otherwise
    actionn
end
```

☺: In practice, `switch-case` statement is used more with character values (or character vectors).

Example: switch-case Statements

```
grade = 'A';

if grade == 'A'
    disp('Your score is in the range of 90-100.');
elseif grade == 'B'
    disp('Your score is in the range of 80-90.');
elseif grade == 'C'
    disp('Your score is in the range of 70-80.');
elseif grade == 'D'
    disp('Your score is below 70.');
else
    disp('We do not have such grade.')
end
```

Q: Write a script to tell you the score range when your input is a grade. `disp()` is a function to print out text in a command window.

```
grade = 'A';

switch grade
    case 'A'
        disp('Your score is in the range of 90-100.');
    case 'B'
        disp('Your score is in the range of 80-90.');
    case 'C'
        disp('Your score is in the range of 70-80.');
    case 'D'
        disp('Your score is below 70.');
    otherwise
        disp('We do not have such grade.');
end
```

Throwing an Error

Optional

- MATLAB has `error` functions that can be used to display an error message in red, similar to the error messages generated by MATLAB or generated by MATLAB grader.
- When an error is thrown in a script, the script stops executing

```
if your_var ~= model_var
    error('Sorry: your value is not correct');
else
    disp('Pass the test');
end
```

- `assert(cond, msg)` throws an error if `cond` is `false`.

```
assert(your_var == model_var, 'Sorry: your value is not correct');
```

Module 04: Loop Statement

Chul Min Yeum
Associate Professor
Civil and Environmental Engineering
University of Waterloo, Canada

UNIVERSITY OF WATERLOO
FACULTY OF ENGINEERING

Module 4: Learning Outcomes

- Describe a problem that requires a loop statement
- Explain how a for-loop structure is operated
- Identify a break command and its operation
- Illustrate the difference between for-loop and while statement
- Solve and design problems using a loop statement and if-statement

For-Loop Statement

- Used as a **counted** loop (We know how many times are repeated)
- **Repeat** *an action* a specified number of times
- An *iterator* or loop variable specifies how many times to repeat the action
- General form:

```
for loopvar = range
    action
end
```

- The `range` is specified by a **vector**.
- The `action` is repeated for every value in the specified `range` and it is assigned to `loopvar`.
- If it is desired to repeat reading input a specified number of times (`N`), a for loop is used:

```
for ii = 1:N
    % do something with it!
end
```

Example: Summation of Values

Q. Sum 1 to 3 and assign the value to 'sumv'

```
1   sumv = 0;
2   for ii=1:3
3       sumv = sumv + ii;
4   end
5
```

Step	Operation	Workspace
1	line1: assign 0 to sumv	sumv → 0
2	line2: ii becomes 1	sumv → 0, ii → 1
3	line3: add ii to sumv and assign the value to sumv	sumv → 1, ii → 1
4	line4: end	sumv → 1, ii → 1
5	go back to line2 and ii becomes 2	sumv → 1, ii → 2
6	line3: add ii to sumv and assign the value to sumv	sumv → 3, ii → 2
7	line4: end	sumv → 3, ii → 2
8	go back to line2 and ii becomes 3	sumv → 3, ii → 3
9	line3: add ii to sumv and assign the value to sumv	sumv → 6, ii → 3
10	line4: end	sumv → 6, ii → 3
11	no more value in range and go to line5	sumv → 6, ii → 3

📖: sumv → 0 indicates that sumv contain a value 0.

Example: Summations of Values in a Vector

Q. Sum all values in a vector named ‘vec’ and assign the value to ‘sumv’

```
1  vec = [2 3 7 11];
2  sumv = 0;
3  for ii=1:4
4      sumv = sumv + vec(ii);
5  end
```

Step	Operation	Workspace
1	line 1 and 2: assign `[2 3 7 11]` to `vec` and `0` to `sumv`	`vec → [1 3 7 11], sumv → 0`
2	line 3 : `ii` becomes `1`	`sumv → 0, ii → 1`
3	line 4: add `vec(ii)` to `sumv` and assign the value to `sumv`	`sumv → 2, ii → 1`
4	line 5: `end`	`sumv → 2, ii → 1`
5	line 3 : `ii` becomes `2`	`sumv → 2, ii → 2`
6	line 4: add `vec(ii)` to `sumv` and assign the value to `sumv`	`sumv → 5, ii → 2`
7	line 5: `end`	`sumv → 5, ii → 2`
:	continue to run until `ii` becomes 4.	
9	line 5 : `end`	`sumv → 23, ii → 4`

Example: Summations of Values in a Vector (Continue)

Two different ways of designing a loop statement.

```
1   vec = [2 3 7 11];
2   sumv = 0;
3   nvec = numel(vec);
4   for ii=1:nvec
5       sumv = sumv + vec(ii);
    end
```

```
1   vec = [2 3 7 11];
2   sumv = 0;
3   for val=vec
4       sumv = sumv + val;
5   end
```

```
for ii = 1:N
    % do something!
end
```

```
for loopvar = range
    % do something!
end
```

Nested For-Loop Statement

- A nested for-loop is one inside of (as the action of) another for loop
- General form of a nested **for** loop:

```
for loopvar1 = range1
    action1
    for loopvar2 = range2
        action2
    end
end
```

- **The inner loop action is executed in its entirety for every value of the outer loop variable.**

Example: Summation of Values in a Matrix

Q. Sum all values in a matrix named '`mat1`' and assign the value to '`sumv`'

mat1

2	8	7
1	5	6

```
1  mat1 = [2 8 7; 1 5 6];
2  sumv = 0;
3  for ii=1:2
4      for jj=1:3
5          sumv = sumv + mat1(ii,jj);
6      end
7  end
```

Step	Operation	Workspace
1	line1: assign values to `mat1`	`mat1 → [2 8 7;1 5 6]`
2	line2: assign `0` to `sumv`	`mat1 → [2 8 7;1 5 6], sumv → 0`
3	line3: `ii` becomes 1	`sumv → 0, ii → 1`
4	line4: `jj` becomes 1	`sumv → 0, ii → 1, jj → 1`
5	line5: Read a value at the `1`st row and `1`st column in `mat1` and add the value to `sumv` and assign the value to `sumv`	`sumv → 2, ii → 1, jj → 1`
6	line6: `end`	`sumv → 2, ii → 1, jj → 1`
7	line4: go back to line 4 and `jj` becomes 2	`sumv → 2, ii → 1, jj → 2`

Omit **`mat1`** after Step 2 in Workspace

Example: Summation of Values in a Matrix (Continue)

Q. Sum all values in a matrix named '`mat1`' and assign the value to '`sumv`'

mat1

2	8	7
1	5	6

```
1  mat1 = [2 8 7; 1 5 6];
2  sumv = 0;
3  for ii=1:2
4      for jj=1:3
5          sumv = sumv + mat1(ii,jj);
6      end
7  end
```

Step	Operation	Workspace
7	line4: `jj` becomes 2	`sumv → 2, ii → 1, jj → 2`
8	line5: Read a value at the `1`st row and `2`nd column in `mat1` and add the value to `sumv` and assign the value to `sumv`	`sumv → 10, ii → 1, jj → 2`
9	line6: `end`	`sumv → 10, ii → 1, jj → 2`
10	line4: `jj` becomes 3	`sumv → 10, ii → 1, jj → 3`
11	line5: Read a value at the `1`st row and `3`rd column in `mat1` and add the value to `sumv` and assign the value to `sumv`	`sumv → 17, ii → 1, jj → 3`
12	line6: `end`	`sumv → 17, ii → 1, jj → 3`

Example: Summation of Values in a Matrix (Continue)

Q. Sum all values in a matrix named '`mat1`' and assign the value to '`sumv`'

mat1

2	8	7
1	5	6

```
1  mat1 = [2 8 7; 1 5 6];
2  sumv = 0;
3  for ii=1:2
4      for jj=1:3
5          sumv = sumv + mat1(ii,jj);
6      end
7  end
```

Step	Operation	Workspace
12	line6: `end`	`sumv → 17, ii → 1, jj → 3`
13	line7: `end`	`sumv → 17, ii → 1, jj → 3`
14	line3: `ii` becomes 2	`sumv → 17, ii → 2, jj → 3`
15	line4: `jj` becomes 1	`sumv → 17, ii → 2, jj → 1`
16	line5: Read a value at the 2nd row and 1st column in `mat1` and add the value to `sumv` and assign the value to `sumv`	`sumv → 18, ii → 2, jj → 1`
17	line6: `end`	`sumv → 18, ii → 2, jj → 1`

Example: Summation of Values in a Matrix (Continue)

Q. Sum all values in a matrix named '`mat1`' and assign the value to '`sumv`'

mat1

2	8	7
1	5	6

```
1  mat1 = [2 8 7; 1 5 6];
2  sumv = 0;
3  for ii=1:2
4      for jj=1:3
5          sumv = sumv + mat1(ii,jj);
6      end
7  end
```

Step	Operation	Workspace
17	line6: `end`	`sumv → 18, ii → 2, jj → 1`
18	line4: `jj` becomes 2	`sumv → 18, ii → 2, jj → 2`
19	line5: Read a value at the 2nd row and 2st column in `mat1` and add the value to `sumv` and assign the value to `sumv`	`sumv → 23, ii → 2, jj → 2`
20	line6: `end`	`sumv → 23, ii → 2, jj → 2`
21	line4: `jj` becomes 3	`sumv → 23, ii → 2, jj → 3`
22	line5: Read a value at the 2nd row and 3rd column in `mat1` and add the value to `sumv` and assign the value to `sumv`	`sumv → 29, ii → 2, jj → 3`
23	line6: `end`, line7: `end`	`sumv → 29, ii → 2, jj → 3`

Summation of Values using Linear Indexing

Challenging

Q. Sum all values in a matrix named '`mat1`' and assign the value to '`sumv`'

```
mat1 = [2 8 7; 1 5 6];
sumv = 0;
for ii=1:2
    for jj=1:3
        sumv = sumv + mat1(ii,jj);
    end
end
```

```
mat1 = [2 8 7; 1 5 6];
sumv = 0;
n_val = numel(mat1);
for ii=1:n_val
    sumv = sumv + mat1(ii);
end
```

mat1

2	8	7
1	5	6

Element

(1,1)	(1,2)	(1,3)
(2,1)	(2,2)	(2,3)

Subscripted indexing

1	3	5
2	4	6

Linear indexing

Pre-allocating a Vector

- Pre-allocation is to set aside enough memory for a vector to be stored
- The alternative, extending a vector, is very inefficient because it requires finding new memory and copying values every time
- Many functions can be used to pre-allocation like `zeros`, `ones`, `eyes`
- For example, to pre-allocate a vector `vec` to have `N` elements:
- `vec = zeros(1,N);`
- (Highly recommended) if you knew the array size that you allocate values computed from a loop !!!

Example: Summation of Values in Each Row of a Matrix

Q. Sum all values in each row of the matrix named '`mat1`' and assign the value to the corresponding row of a row vector named '`rvec`' .

```
mat1 = [2 8; 1 3; 2 3];
rvec = zeros(1,3);
for ii=1:3
    sumr = 0;
    for jj=1:2
        sumr = sumr + mat1(ii,jj);
    end
    rvec(ii) = sumr; % identical with rvec(ii,1)
end
```

Name	Value
mat1	[2 8; 1 3; 2 3]
rvec	[10 4 5]

```
mat1 = [2 8; 1 3; 2 3];
for ii=1:3
    sumr = 0;
    for jj=1:2
        sumr = sumr + mat1(ii,jj);
    end
    rvec(ii) = sumr;
end
```

⚠: `rvec` is not pre-allocated. Thus, in each iteration, the size of `rvec` will be changed

Working but not recommended

Example: Summation of Values in Each Row of a Matrix (Continue)

Q. Sum all values in each row of the matrix named '`mat1`' and assign the value to the corresponding row of a row vector named '`rvec`' .

```
mat1 = [2 8; 1 3; 2 3];
rvec = zeros(1,3);
for ii=1:3
    sumr = 0;
    for jj=1:2
        sumr = sumr + mat1(ii,jj);
    end
    rvec(ii) = sumr; % identical with rvec(ii,1)
end
```

Name	Value
mat1	[2 8; 1 3; 2 3]
rvec	[10 4 5]

```
mat1 = [2 8; 1 3; 2 3];
rvec = zeros(1,3);

for ii=1:3
    for jj=1:2
        rvec(ii) = rvec(ii) + mat1(ii,jj);
    end
end
```

Combining for-loop and if-else Statements

- **for**-**loop** and **if-else** statements can be combined
  - the action of a loop can include an **if-else** statement
  - the action of an **if-else** statement can include a **for-loop**
- This is also true for the nested **for**-**loops**; **if-else** statements can be part of the action(s) of the outer and/or inner loops
- This is done if an action is required on an element (of a vector or matrix) only if a condition is met.

```
for loopvar1 = range1
    if condition
       action
    end
end
```

📖 : Again, `action` is only executed if `condition` is `true`. This form is to execute `action` using selective values in `range1`.

Example: Combining for-loop and If-elseif Statement

Q. Change 1 to 5, 2 to 7, and the rest to 10 in a given vector named 'vec'

```
vec = [1 1 2 1 3 1 6 7 5];

n_vec = numel(vec);
for ii=1:n_vec
    if vec(ii) == 1
        vec(ii) = 5;
    elseif vec(ii) == 2
        vec(ii) = 7;
    else
        vec(ii) = 10;
    end
end
```

Name	Value
vec	[5 5 7 5 10 5 10 10 10]
n_vec	9

```
for loopvar1 = range1
    if condition
        action
    end
end
```

☺: We will revisit this examples later!!

Example: Value Replacement

Q. If values in ‘vec' are larger than and equal to 0 and less than 50, replace the values to 10. Otherwise, replace them to 5.

```
vec = [1 10 70 80 2];

n_vec = numel(vec);

for ii=1:n_vec
    t_val = vec(ii);
    if (t_val >= 0) && (t_val < 50)
        vec(ii) = 10;
    else
        vec(ii) = 5;
    end
end
```

Name	Value
vec	[1 10 70 80 2]
n_vec	[10 10 5 5 10]

Example: Find a Value

Q. Find index(es) of a vector where 5 is located. The vector named as 'vec' is given. The resulting indexes are assigned to 'loc'.

```
vec = [1 5 6 4 8 5 3 7 5];

n_vec = numel(vec);
loc = [];
for ii=1:n_vec
    if vec(ii) == 5
        loc = [loc ii];
    end
end
```

Option 1

Name	Value
loc	[2 6 9]

```
vec = [1 5 6 4 8 5 3 7 5];

n_vec = numel(vec);
loc = [];
for ii=1:n_vec
    if vec(ii) == 5
        loc(end+1) = ii;
    end
end
```

Option 2

```
vec = [1 5 6 4 8 5 3 7 5];

n_vec = numel(vec);
loc = 0;
count = 1;
for ii=1:n_vec
    if vec(ii) == 5
        loc(count) = ii;
        count = count + 1;
    end
end
```

Option 3

Example: Find a Value (Continue)

Q. Find index(es) of a vector where `5` is located. The vector named as '`vec`' is given and the indexes are assigned to '`loc`'.

```
vec = [1 5 6 4 8 5 3 7 5];

n_vec = numel(vec);
loc = [];
for ii=1:n_vec
    if vec(ii) == 5
        loc = [loc ii];
    end
end
```

Option 1

📖 : Since you do not know the size of loc in advance, you can pre-allocate the variable with its maximum size and delete the "unused" elements.

```
vec = [1 5 6 4 8 5 3 7 5];

n_vec = numel(vec);
loc = zeros(1, n_vec);

count = 0;
for ii=1:n_vec
    if vec(ii) == 5
        count = count + 1;
        loc(count) = ii;
    end
end

loc = loc(1:count);
```

Option 4

Example: Bulls and Cows

Bulls and Cows is a mind game played by two players. In the game, a random, 4-digit number is chosen and its values are compared to those of another trial number. **All four digits of the number are different**. If any digit in the chosen number is the exact same value and in the exact same position as any digit in the trial number, this is called a bull. If the digit is present in both the trial number and chosen number, but is not in the same location, this is called a cow.

In this figure, A: Bulls, B: Cows

Example: Bulls and Cows

```
x_true = [1 2 3 4]; % true
x_test = [3 2 5 6]; % test

numb = 0; % number of Bull
if x_true(1) == x_test(1)
   numb = numb + 1;
end

if x_true(2) == x_test(2)
   numb = numb + 1;
end

if x_true(3) == x_test(3)
   numb = numb + 1;
end

if x_true(4) == x_test(4)
   numb = numb + 1;
end
```

Q: Write a script to compute "Bull" and assign its value to `num_b`. The true and test sequence is in `x_true` and `x_test`, respectively.

```
x_true = [1 2 3 4]; % true
x_test = [3 2 5 6]; % test

numb = 0; % number of Bull
for ii=1:4
    if x_true(ii) == x_test(ii)
        numb = numb + 1;
    end
end
```

☺: Much simple and readable.

Example: Bulls and Cows (Continue)

Q: Write a script to compute “Cows” + “Bulls” and assign its value to `num_c`. In other word, you need to compute how many same digits are present in both sequences. The true and test sequence is in `x_true` and `x_test`, respectively.

```
x_true = [1 2 3 4]; % true
x_test = [3 2 5 6]; % test

numc = 0;
for ii=1:4
    for jj=1:4
        if x_true(ii) == x_test(jj)
            numc = numc + 1;
        end
    end
end
```

Name	Value
x1	[1 2 3 4]
x2	[3 2 5 6]
numc	2

📖: “Cows” becomes “numc” – “numb” obtained from the previous code.

Break

- **break** command can be used to terminate a loop prematurely (while the comparison in the first line is still true.
- A **break** statement will cause termination of the current (closest) enclosing while or for loop.

```
1   vec = zeros(1,5);
2
3   for ii=1:5
4
5       if ii == 3
6           break;
7       end
8       vec(ii) = ii;
9
10  end
```

Name	Value
vec	[1 2 0 0 0]
ii	3

📖: When `ii` is equal to `3`, `condition` in line 5 is `true` so `break` command is executed. Then, the code directly goes to `end` at line 10 (skip a script inside a loop that includes the `break` command.

Example: Use Break

Q: Find the first appearance location of `'a'` in a character vector named as `'char_seq'` and assign its location (index) to `'loc'`.

```
char_seq = 'Hello I am Matlab';

num_char = numel(char_seq);
loc = 0;
for ii=1:num_char
    if char_seq(ii) == 'a'
        loc = ii;
        break;
    end
end
```

Name	Value
char_seq	'Hello I am Matlab'
loc	9
num_char	17
ii	9

Q1. Think about an advantage of the use of `break` in this problem

Q2. Think about what value is assigned to `loc` if `break` command is deleted in the script.

While Loop

- used as a **conditional** loop
- used to **repeat** an action when ahead of time it is not known how many times the action will be repeated
- general form:

```
while condition
  action
end
```

📖 : `condition` is an exist condition for the loop.

- *the **action** is repeated as long as the condition is true*
- Note: since the condition comes before the action, it is possible that the condition will be false the first time and then, the action will not be executed at all

⚠: If `action` cannot make `condition` `true`, an infinite loop can occur. Use **Ctrl-C** to break out of the infinite loop.

Example: Summation

Q. Summation of 1 to 10 using a while loop

```
1  num = 1;
2  val = 0;
3
4  while num < 11
5      val = val + num;
6      num = num+ 1;
7  end
```

Step	Operation	Workspace
1	lines 1 & 2: assign `1` to `val` and assign `0` to `num`	`num → 1, val → 0`
2	line 4: since `num` is less than `11`, `condition` is `true` and `action` is executed.	`num → 1, val → 0`
3	lines 5 & 6: add `num` to `val` and add `1` to `num`	`num → 2, val → 1`
4	line 7: `end`, and go back to line 4	`num → 2, val → 1`
5	line 4: since `num` is less than `11`, `condition` is `true`, and `action` is executed.	`num → 2, val → 1`
6	lines 5 & 6: add `num` to `val` and add `1` to `num`	`num → 3, val → 3`
:	continue to run until `num` becomes 10.	`num → 10, val → 45`
8	line 4: since `num` is less than `11`, `condition` is `true`, and `action` is executed.	`num → 10, val → 45`
9	lines 5 & 6: add `num` to `val` and add `1` to `num`	`num → 10, val → 55`
10	line 7: `end`, and go back to line 4	`num → 11, val → 55`
11	line 4: since `num` is **NOT** less than `11`, `action` is skipped and go to line 7.	`num → 11, val → 55`

Example: Find a Location

Q: Find a location of the 3rd occurrence of number 5 in a vector named as `vec` and assign its location (index) to `loc`.

```
vec = [1 2 5 5 2 5 3 5 6];

num_5 = 0; % number of 5
idx = 0;

while num_5 < 3
    idx = idx + 1;
    if vec(idx) == 5
        num_5 = num_5 + 1;
    end
end

loc = idx;
```

Name	Value
vec	[1 2 5 5 2 5 3 5 6]
loc	6
num_5	3
idx	6

```
vec = [1 2 5 5 2 5 3 5 6];

num_5 = 0; % number of 5
loc = 0;

while num_5 < 3
    loc = loc + 1;
    if vec(loc) == 5
        num_5 = num_5 + 1;
    end
end
```

Example: Find a Location (Continue)

Optional

If you understand the operation of the while loop, you can design your code in a different way.

```
vec = [1 2 5 5 2 5 3 5 6];

num_5 = 0; % number of 5
loc = 0;

while num_5 < 3
    loc = loc + 1;
    if vec(loc) == 5
        num_5 = num_5 + 1;
    end
end
```

```
vec = [1 2 5 5 2 5 3 5 6];

num_5 = 0; % number of 5
loc = 0;

while num_5 ~= 3
    loc = loc + 1;
    if vec(loc) == 5
        num_5 = num_5 + 1;
    end
end
```

📖 : When `num_5` is equal to `3`, the while loop stop repeating but execute all script inside the loop (thus, `idx` becomes `7`).

Example: Find a Location (Continue)

Optional

You can build the same operation using for-loop with break.

```
vec = [1 2 5 5 2 5 3 5 6];

num_5 = 0; % number of 5
loc = 0;

while num_5 < 3
    loc = loc + 1;
    if vec(loc) == 5
        num_5 = num_5 + 1;
    end
end
```

```
vec = [1 2 5 5 2 5 3 5 6];

n_vec = numel(vec);

num_5 = 0;
loc = 0;

for ii=1:n_vec
    if vec(ii) == 5
        num_5 = num_5 + 1;
    end

    if num_5 == 3
        loc = ii;
        break;
    end
end
```

Dot Product

Remind

The dot product operation between two compatible vectors is defined as

$$\vec{a} \bullet \vec{b} = \begin{bmatrix} a_1 \\ \vdots \\ a_n \end{bmatrix} \bullet \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix} = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n = \sum_{k=1}^{n} a_k b_k$$

which is the linear combination of all elements in both vectors. The result is a scalar. Interestingly, this linear combination is the same as

$$\vec{a}^T \vec{b} = \begin{bmatrix} a_1 \\ \vdots \\ a_n \end{bmatrix}^T \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix} = \begin{bmatrix} a_1 & \cdots & a_n \end{bmatrix} \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix} = \begin{bmatrix} a_1 b_1 + a_2 b_2 + \cdots + a_n b_n \end{bmatrix}$$

The dot product is also known as the scalar product or inner product, and sometimes uses the syntax $\left(\vec{a}, \vec{b}\right) = \left\langle \vec{a}, \vec{b} \right\rangle = \vec{a} \bullet \vec{b}$.

Example: Dot Product

Q. Write a script to compute a dot product of `A` and `B`, which are the same size row vectors. Assign its value to `AB`.

```
A = [3 4 5 6 8];
B = [5 6 7 7 8];

% Loop structure
n_elem = numel(A);

AB_vec = zeros(1, n_elem);

for ii=1:n_elem
    AB_vec(ii) = A(ii)*B(ii);
end % altern: AB_vec = A.*B;

AB = 0;
for ii=1:n_elem
    AB = AB + AB_vec(ii);
end
```

```
A = [3 4 5 6 8];
B = [5 6 7 7 8];

% Loop structure
n_elem = numel(A);

AB = 0;
for ii=1:n_elem
    AB = AB + A(ii)*B(ii);
end
```

Name	Value
AB	180

```
% Use of a built-in function
AB = dot(A,B);
```

```
% Use of a operator
AB = A * B';
```

Module 05: Built-in Functions

Chul Min Yeum
Associate Professor
Civil and Environmental Engineering
University of Waterloo, Canada

UNIVERSITY OF WATERLOO
FACULTY OF ENGINEERING

Module 5: Learning Outcomes

- Relate an array as an input argument of built-in functions
- Print elements in variables and format output texts
- Apply built-in arithmetic functions and replicate the same capability using selection and loop statements
- Use a sort function to sort array elements

Built-in Function (Array as an Input Argument)

round

R2018b

Round to nearest decimal or integer

Syntax

```
Y = round(X)
Y = round(X,N)
Y = round(X,N,type)

Y = round(t)
Y = round(t,unit)
```

Description

example

`Y = round(X)` rounds each element of `X` to the nearest integer. In the case of a tie, where an element has a fractional part of exactly `0.5`, the `round` function rounds away from zero to the integer with larger magnitude.

example

`Y = round(X,N)` rounds to `N` digits:

- `N > 0`: round to `N` digits to the *right* of the decimal point.
- `N = 0`: round to the nearest integer.
- `N < 0`: round to `N` digits to the *left* of the decimal point.

example

`Y = round(X,N,type)` specifies the type of rounding. Specify `'significant'` to round to `N` significant digits (counted from the leftmost digit). In this case, `N` must be a positive integer.

Input Arguments

x — Input array

scalar | vector | matrix | multidimensional array

📖 : It is useful to conduct the function over each elements in an array. We do not have to implement a loop statement. You should check if the function support the array as an input format.

Built-in Function (Array as an Input Argument) (Continue)

- Entire arrays (vectors or matrices) can be used as input arguments to functions
- The result will have the same dimensions as the input

Q. Write a code to round to the nearest integer of each values in the vector.

```
vec = [1.1 2.3 -3.1 4.7 8.9];
n_vec = numel(vec);

vec_r = zeros(1, n_vec);
for ii=1:n_vec
    vec_r(ii) = round(vec(ii));
End
```

```
vec = [1.1 2.3 -3.1 4.7 8.9];

vec_r = round(vec);
```

Name	Value
vec	[1.1 2.3 -3.1 4.7 8.9]
n_vec	5
vec_r	[1 2 -3 5 9]

Example: Built-in Function

Q. Write a code to compute a square root of each value in the vector.

```
vec = [3 2 4 1 2 4];
n_vec = numel(vec);

for ii=1:n_vec
    vec(ii) = sqrt(vec(ii));
end
```

```
vec = [3 2 4 1 2 4];
vec = sqrt(vec);
```

Q. Write a code to compute a sign of each values in a matrix

```
mat1 = [3 2 4; -1 -2 4; 3 0 -1];
[nr, nc] = size(mat1);
for ii=1:nr
    for jj = 1: nc
        val= sign(mat1(ii, jj));
        mat1(ii, jj) = val;
    end
end
```

```
mat1 = [3 2 4; -1 -2 4; 3 0 -1];
mat1 = sign(mat1);
```

3	2	4
-1	-2	4
3	0	-1

1	1	1
-1	-1	1
1	0	-1

Array Operation

Function	Description
`reshape(x, sz)`	Changes dimensions of a matrix to any matrix with the same number of elements
`flip(x, dim)`	Flips a row vector left to right, column vector or matrix up to down, the elements in each column or row in a matrix
`cat(dim, A, B)`	Concatenate arrays along specified dimension
`diag(x)` `diag(A)`	Create diagonal matrix or get diagonal elements of matrix

Function: `reshape(x, sz)`

`B = reshape(A,sz)` reshapes `A` using the size vector, `sz`, to define `size(B)`. `sz` must contain at least 2 elements, product of elements in `sz` must be the same as `numel(A)` (Note: Arrays are reshaped in a linear indexing order).

```
vec = 3:14; % 12 elements

mat0 = reshape(vec, [2 6]);
mat1 = reshape(vec, [3 4]);
mat2 = reshape(mat1, [6 2]);
```

mat1

3	6	9	12
4	7	10	13
5	8	11	14

vec

3	4	5	6	7	8	9	10	11	12	13	14

mat0

3	5	7	9	11	13
4	6	8	10	12	14

mat2

3	9
4	10
5	11
6	12
7	13
8	14

Function: `flip(x, dim)`

`B = flip(A,dim)` reverses the order of the elements in `A` along dimension `dim`.

```
mat0 = reshape(3:14, [3 4]);
mat1 = flip(mat0, 1);
mat2 = flip(mat0, 2);
```

📖 : `dim` - If no value is specified, then the default is `1`. `1` indicates the vertical direction, and `2` indicates the horizontal direction.

`2`: horizontal

`1`: vertical

`mat0`

3	6	9	12
4	7	10	13
5	8	11	14

`mat1`

5	8	11	14
4	7	10	13
3	6	9	12

`mat2`

12	9	6	3
13	10	7	4
14	11	8	5

Function: `cat(dim, A, B)`

`C = cat(dim, A, B)` concatenates the arrays `A` and `B` along array the dimension specified by the number of dimensions.

```
mat0 = [1 1; 2 2];
mat1 = zeros(2,2);

mat2 = cat(1, mat0, mat1);
mat3 = cat(2, mat0, mat1);
mat4 = cat(3, mat0, mat1);
```

Name	Value
mat0	[1 1; 2 2]
mat1	[0 0; 0 0]
mat2	[1 1; 2 2; 0 0; 0 0]
mat3	[1 1 0 0; 2 2 0 0]
mat4	2 x 2 x 2 double

mat0

1	1
2	2

mat1

0	0
0	0

mat2

1	1
2	2
0	0
0	0

mat4

	0	0
		0
1	1	
2	2	

mat3

1	1	0	0
2	2	0	0

Function: `diag(x), diag(A)`

- `D = diag(v)` returns a square diagonal matrix `D` with the elements of vector `v` on the main diagonal.
- `x = diag(A)` returns a column vector of the main diagonal elements of matrix `A`.

```
vec = [1 2 3];
mat0 = [2 1 3; 4 5 2; 2 3 4];

D = diag(vec);
v = diag(mat0)
```

2	1	3
4	5	2
2	3	4

mat0

2
5
4

v

1	0	0
0	2	0
0	0	3

D

vec

1	2	3

```
vec = [1 2 3];

D = zeros(3,3);
for ii=1:3
   D(ii,ii) = vec(ii);
end

mat0 = [2 1 3; 4 5 2; 2 3 4];

v = zeros(3,1);
for ii=1:3
    v(ii) = mat0(ii,ii);
end
```

Output Functions

- There are two basic output functions:
  - `disp`, which is a quick way to display things
  - `fprintf`, which allows formatting

- The `fprintf` function uses format specifiers which include place holders; these have conversion characters:

```
fprintf(formatspec, A1, A2)
```

  - `%d` : integers
  - `%f` : floats (real numbers)
  - `%c` : single characters
  - `%s` : string of characters

📖 : This function and syntax is to print out numeric values or texts in a command window.

- `\n` newline character

Output Functions (Continue)

```
x = 1234.5678
v = [ 1 2 3 4]
```

```
disp(x)
disp(v)
disp('Hello! Matlab')
```

```
1.2346e+03
1     2     3     4
Hello! Matlab
```

```
fprintf('Hello! Matlab')
fprintf('Hello! %catlab', 'M')
fprintf('Hello! %slab', 'Mat')
fprintf('Hello! \nMatlab')
```

```
Hello! Matlab
Hello! Matlab
Hello! Matlab
Hello!
Matlab
```

```
x = 1234.5678
y = 10;
```

```
fprintf('case1: x is %f.', x);
fprintf('case2: y is %d.', y);
```

```
case1: x is 1234.567800.
case2: y is 10.
```

Sum, Average, Min & Max, and Sorting

Sum of array elements

- `S = sum(A)`
- `S = sum(A,'all')`
- `S = sum(A,dim)`

Average or mean value of array

- `S = mean(A)`
- `S = mean(A,'all')`
- `S = mean(A,dim)`

Median value of array

- `S = median(A)`
- `S = median(A,'all')`
- `S = median(A,dim)`

Maximum elements of an array

- `M = max(A)`
- `M = max(A, [], dim)`
- `[M, I] = max(A)`
- `C = max(A, B)`

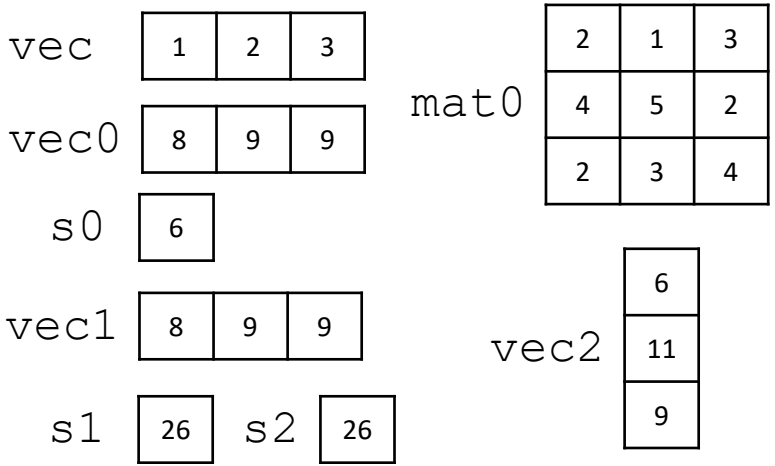
Sort array elements

- `B = sort(A)`
- `B = sort(A,dim)`
- `B = sort(A,direction)`
- `[B,I] = sort(A)`

Function: sum

- `S = sum(A)` returns the sum of the elements of `A`
  - If `A` is a vector, then `sum(A)` returns the sum of the elements.
  - If `A` is a matrix, then `sum(A)` returns a row vector containing the sum of each column (= `sum(A, 1)`).
- `S = sum(A,dim)` returns the sum along dimension `dim`.
- `S = sum(A,'all')` computes the sum of all elements of `A`.

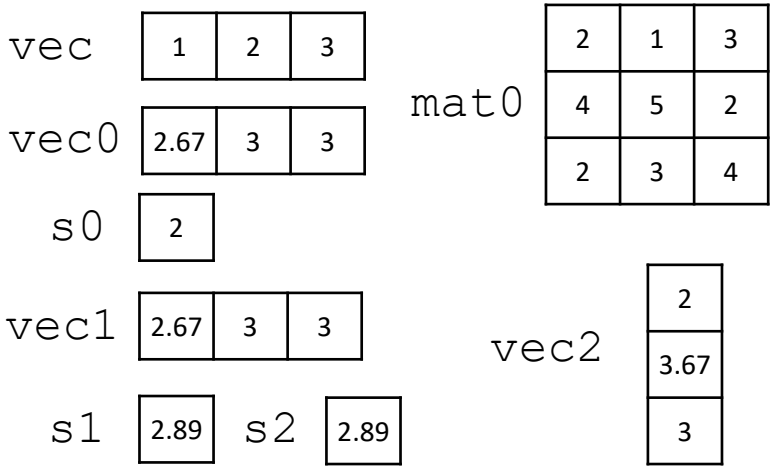
```
vec = [1 2 3];
mat0 = [2 1 3; 4 5 2; 2 3 4];

s0 = sum(vec);
vec0 = sum(mat0);

vec1 = sum(mat0, 1);
vec2 = sum(mat0, 2);

s1 = sum(mat0, 'all');
s2 = sum(mat0(:));
```



Function: `mean`

- `M = mean(A)` returns the mean of the elements of `A`
  - If `A` is a vector, then `mean(A)` returns the mean of the elements.
  - If `A` is a matrix, then `mean(A)` returns a row vector containing the mean of each column.
- `M = mean(A,'all')` computes the mean over all elements of `A`.
- `M = mean(A,dim)` returns the mean along dimension `dim`.

```
vec = [1 2 3];
mat0 = [2 1 3; 4 5 2; 2 3 4];

s0 = mean(vec);
vec0 = mean(mat0);

vec1 = mean(mat0, 1);
vec2 = mean(mat0, 2);

s1 = mean(mat0, 'all');
s2 = mean(mat0(:));
```



Example: `sum` and `mean`

Q. Write a code to compute `vec1`, `vec2`, `s1`, `s2` from `mat0` without using `sum` and `mean`

```
mat0 = [2 1 3; 4 5 2; 2 3 4];

[nr, nc] = size(mat0);

s1 = 0;
vec1 = zeros(nr, 1);
for ii=1:nr
    for jj=1:nc
        s1 = s1 + mat0(ii,jj);
        vec1(ii) = vec1(ii) + mat0(ii, jj);
    end
end

s2 = s1/(nr*nc);
vec2 = vec1/nc;
```

```
mat0 = [2 1 3; 4 5 2; 2 3 4];

vec1 = sum(mat0, 2);
vec2 = mean(mat0, 2);

s1 = sum(mat0, 'all');
s2 = mean(mat0, 'all');
```

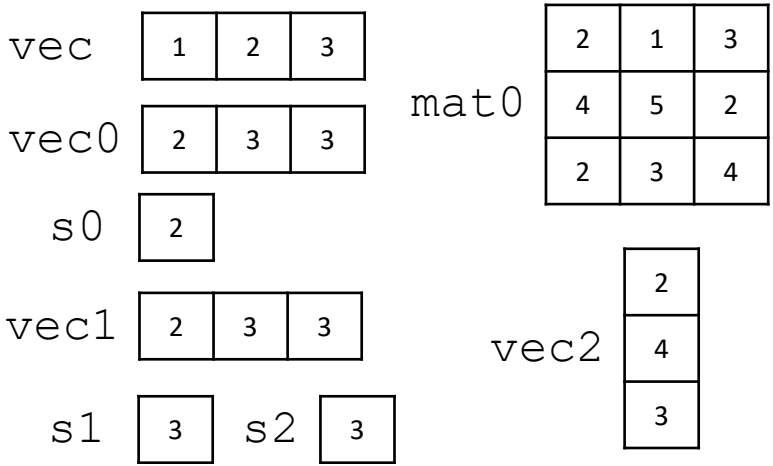

mat0

2	1	3
4	5	2
2	3	4

Function: median

- `M = median(A)` returns the median of the elements of `A`
  - If `A` is a vector, then `median(A)` returns the median of the elements.
  - If `A` is a matrix, then `median(A)` returns a row vector containing the median of each column.
- `M = median(A,'all')` computes the median over all elements of `A`.
- `M = median(A,dim)` returns the median along dimension `dim`.

```
vec = [1 2 3];
mat0 = [2 1 3; 4 5 2; 2 3 4];

s0 = median(vec);
vec0 = median(mat0);

vec1 = median(mat0, 1);
vec2 = median(mat0, 2);

s1 = median(mat0, 'all');
s2 = median(mat0(:));
```



Function: `max`

- `M = max(A)` returns the maximum elements of an array.
  - If `A` is a vector, then `max(A)` returns the maximum of `A`.
  - If `A` is a matrix, then `max(A)` is a row vector containing the maximum value of each column.
- `M = max(A,[],dim)` returns the largest elements of `A` along dimension `dim`. For example, if `A` is a matrix, then `max(A,[],2)` is a column vector containing the maximum value of each row.
- `M = max(A,[],'all')` finds the maximum over all elements of `A`.
- `[M,I] = max(___)` finds the indices of the maximum values of input `A`, which can be a vector or matrix, and returns them in output vector `I`, using any of the input arguments in the previous syntaxes. If the maximum value occurs more than once, then max returns the index **corresponding to the <u>first occurrence</u>**.
- `C = max(A,B)` returns an array with the largest elements taken from `A` or `B`.

Function: `max` (Continue)

```
vec = [1 3 2];
mat0 = [2 1 3; 4 5 2; 2 4 4];

s0 = max(1, 2);
s1 = max(vec);
[M, I1] = max(vec);

vec1 = max(mat0)
vec2 = max(mat0, [], 1);
vec3 = max(mat0, [], 2);

[~, I2] = max(mat0, [], 1);
[~, I3] = max(mat0, [], 2);

s2 = max(mat0, [], 'all')
s3 = max(mat0(:))
```

Name	Value
s0	2
s1	3
M	3
I1	2
vec1	[4 5 4]
vec2	[4 5 4]
vec3	[3; 5; 4]
I2	[2 2 3]
I3	[3; 2; 2]
s2	5
s3	5

mat0

2	1	3
4	5	2
2	4	4

vec

1	3	2

I1

2

I2

2	2	3

I3

3
2
2

Example: `max` (Implementation)

Q. Write a code to find a maximum number of a given array without using `max`.

```
vec = [2 1 5 7 4 2 3 9 4 2];

n_v = numel(vec);

max_val = vec(1);

for ii=2:n_v
    if vec(ii) > max_val
        max_val = vec(ii);
    end
end
```

📖 : Check if each number in an array is bigger than the current maximum value, `max_val`. If a new value is larger, `max_val` is replaced to the new value. In general, `max_val` is initialized with the first element of an input array.

Q. How to change the code if the given array is a matrix?

2	1	5	7	4	2	3	9	4	2

Function: `min`

```
vec = [1 3 2];
mat0 = [2 1 3; 4 5 2; 2 2 4];

s0 = min(1, 2);
s1 = min(vec);
[~, I1] = min(vec);

vec1 = min(mat0)
vec2 = min(mat0, [], 1);
vec3 = min(mat0, [], 2);

[~, I2] = min(mat0, [], 1);
[~, I3] = min(mat0, [], 2);

s2 = min(mat0, [], 'all')
s3 = min(mat0(:))
```

vec

1	3	2

mat0

2	1	3
4	5	2
2	2	4

Name	Value
s0	1
s1	1
I1	1
vec1	[2 1 2]
vec2	[2 1 2]
vec3	[1; 2; 2]
I2	[1 1 2]
I3	[2; 3; 1]
s2	1
s3	1

I1

1

I2

1	1	2

I3

2
3
1

Function: `sort`

- `B = sort(A)` sorts the elements of `A` in ascending order.
  - If `A` is a vector, then `sort(A)` sorts the vector elements.
  - If `A` is a matrix, then `sort(A)` treats the columns of `A` as vectors and sorts the elements within each vector column.
- `B = sort(A,dim)` returns the sorted elements of `A` along dimension `dim`.
- `B = sort(___,direction)` returns sorted elements of `A` in the order specified by direction using any of the previous syntaxes. `'ascend'` indicates ascending order (the default) and `'descend'` indicates descending order.
- `[B,I] = sort(___)` also returns a collection of index vectors for any of the previous syntaxes. `I` is the same size as `A` and describes the arrangement of the elements of `A` into `B` along the sorted dimension.

Function: `sort` (Continue)

```
vec = [2 1 3];
mat0 = [2 1 3; 4 5 2; 2 3 4];

v0 = sort(vec);
v1 = sort(vec, 'descend');
[B1, I1] = sort(vec, 'descend');
v2 = vec(I1);

mat2 = sort(mat0, 1);
[B2, I2] = sort(mat0, 1);

mat3 = sort(mat0, 2);
[B3, I3] = sort(mat0, 2);
```

vec

2	1	3

B1

3	2	1

v0

1	2	3

I1

3	1	2

v1

3	2	1

v2

3	2	1

mat2

2	1	2
2	3	3
4	5	4

I2

1	1	2
3	3	1
2	2	3

mat0

2	1	3
4	5	2
2	3	4

mat3

1	2	3
2	4	5
2	3	4

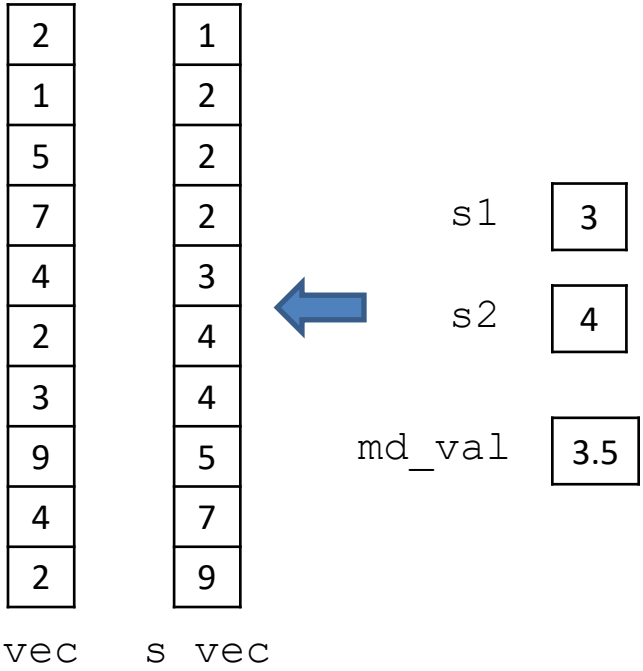
I3

2	1	3
3	1	2
1	2	3

Example: `median` (Implementation)

Q. Write a code to find a median of a given vector without using `median` (use `sort`).

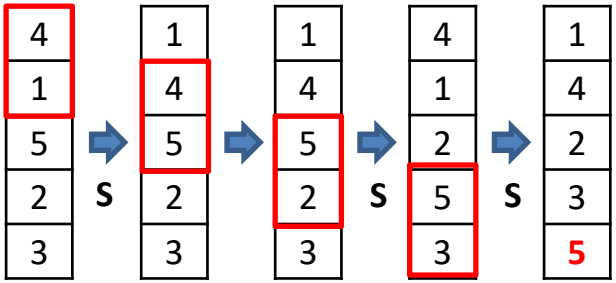
```
vec = [2 1 5 7 4 2 3 9 4 2]';

n_v = numel(vec);
s_vec = sort(vec);
idx = ceil(n_v/2);

if rem(n_v, 2) == 1
    md_val = s_vec(idx);
else
    idx = n_v/2;
    s1 = s_vec(idx);
    s2 = s_vec(idx+1);
    md_val = (s1+s2)/2;
end
```



Sort Algorithm: Bubble Sort

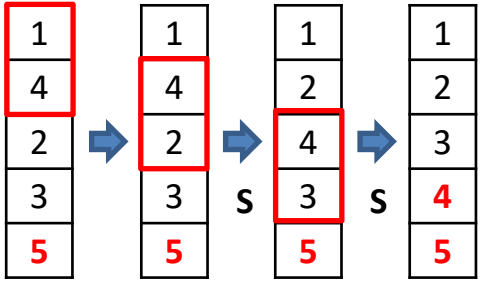
Challenging

Bubble sort, is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted.



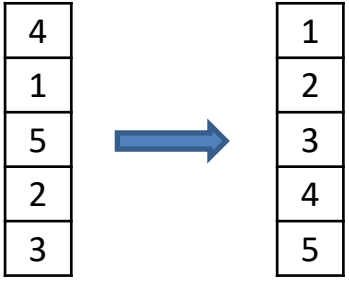
Step1: There are 4 conditions statements performed. At the end of the first step, we can find the last value.

Here '**S**' means swapping the values.



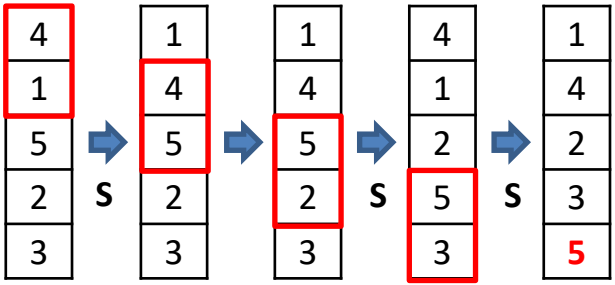
Step2: There are 3 conditions statements performed. At the end of the second step, we can find the 2nd last value.

Q. How to sort 'vec'

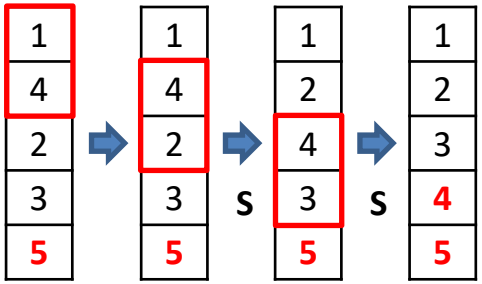


Sort Algorithm: Bubble Sort (Continue)

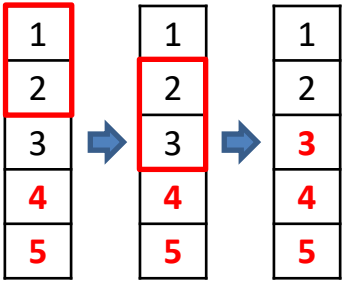
Challenging



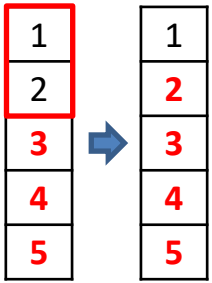
Step1



Step2



Step3



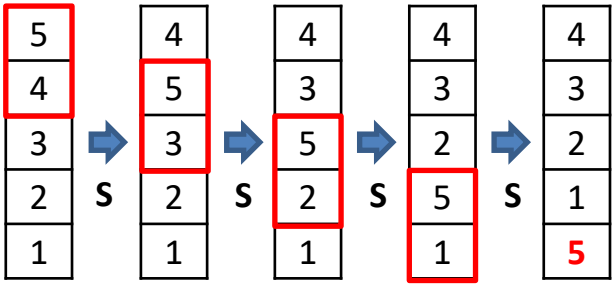
Step4

Summary

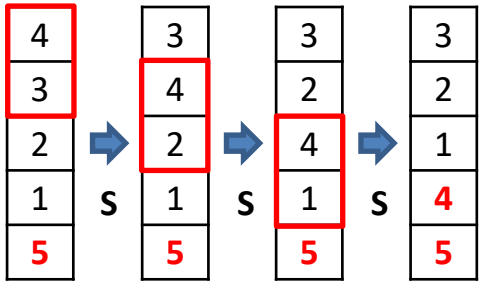
- 5 elements
- 4 steps needed
- Each steps, the number of actions (condition statement) is decreased by one.

Sort Algorithm: Bubble Sort (Another Example)

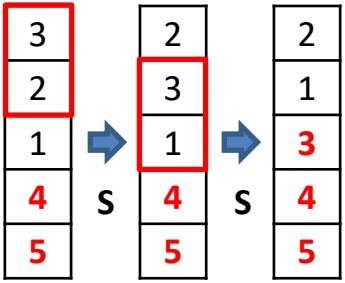
Challenging



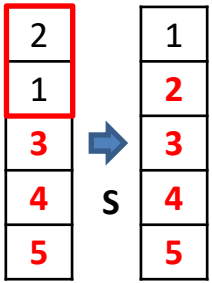
Step1



Step2



Step3



Step4

Summary

- 5 elements
- 4 steps needed
- Each steps, the number of actions (condition statement) is decreased by one.

Sort Algorithm: Bubble Sort (Implementation)

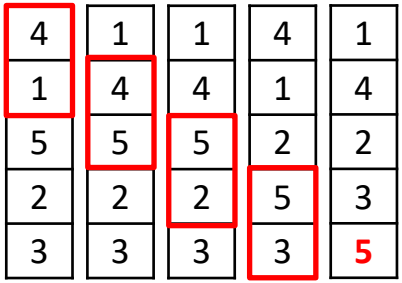
Challenging

Bubble sort, is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted.

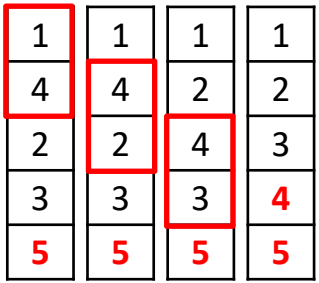
```
vec = [4 1 5 2 3]';
nv = numel(vec);

for ii=1:nv-1
    for jj=1:nv-ii
        if vec(jj)>vec(jj+1)
            tmp = vec(jj);
            vec(jj) = vec(jj+1);
            vec(jj+1) = tmp;
        end
    end
end
```

☺: The script in italic & bold can be replaced as

```
vec([jj jj+1]) = vec([jj+1 jj]);
```



At ii=1



At ii=2

Module 06: Operators

Chul Min Yeum
Associate Professor
Civil and Environmental Engineering
University of Waterloo, Canada

UNIVERSITY OF WATERLOO
FACULTY OF ENGINEERING

Module 6: Learning Outcomes

- Describe vectorization coding in MATLAB
- Apply numeric, relational, and logical operators to the combinations of scalar, vector, and matrix
- Redesign loop-based scripts using vectorization
- Solve problems using operators

Vectorization

- MATLAB® is optimized for operations involving matrices and vectors. The process of revising loop-based, scalar-oriented code to use MATLAB matrix and vector operations is called vectorization.

- Vectorizing your code is worthwhile for several reasons:
  - Appearance: Making the code easier to understand.
  - Less error prone: **Without loops, vectorized code is often shorter. Fewer lines of code mean fewer risks to introduce programming errors.**
  - Performance: Vectorized code often runs much faster than the corresponding code containing loops (in MATLAB).

📖: For example, instead of looping through all elements in a vector `vec` to add `3` to each element, just use scalar addition: `vec = vec + 3;`

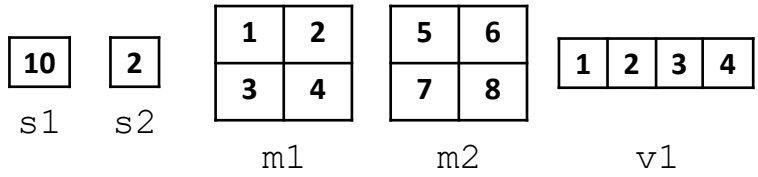
MATLAB Operator

Symbol	Role
+	Addition
–	Subtraction
*	Multiplication
/	Division
^	Matrix power
./	Element-wise right division
.*	Element-wise multiplication
.^	Element-wise power
'	Transpose

Symbol	Role
==	Equal to
~=	Not equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
&&	Logical AND (scalar logical)
||	Logical OR (scalar logical)
~	Logical NOT
&	Logical AND (array)
|	Logical OR (array)

MATLAB Operator: Arithmetic

```
s1 = 10; % scalar value
s2 = 2; % scalar value

v1 = [1 2 3 4]; % vector

m1 = [1 2; 3 4]; % matrix
m2 = [5 6; 7 8]; % matrix
```



Operator
+
-
*
/
^

Scalar ★ Scalar
Vector ★ Scalar
Matrix ★ Scalar
Vector ★ Vector
Vector ★ Matrix
Matrix ★ Matrix

★: Operator

`s1 + s2`	`12`
`s1 * s2`	`20`
`s1 / s2`	`5`
`s1 ^ s2`	`100`
`s1 + v1`	`[11 12 13 14]`
`v1 - s2`	`[-1 0 1 2]`
`v1 * s1`	`[10 20 30 40]`
`v1 / s2`	`[0.5 1 1.5 2]`
`m1 * s1`	`[10 20; 30 40]`
`m1 / s1`	`[0.1 0.2; 0.3 0.4]`
`m1 + s1`	`[11 12; 13 14]`
`v1 ^ s2`	*error*
`m1 ^ s2`	`[7 10; 15 22]`

MATLAB Operator: Arithmetic (Continue)

```
v1 = [1 2]; % vector
v2 = [1; 0]; % vector

m1 = [1 2; 3 4]; % matrix
m2 = [1 0; 0 1]; % matrix
```

Operator
+
-
*
/
^

Scalar ★ Scalar
Vector ★ Scalar
Matrix ★ Scalar
Vector ★ Vector
Vector ★ Matrix
Matrix ★ Matrix

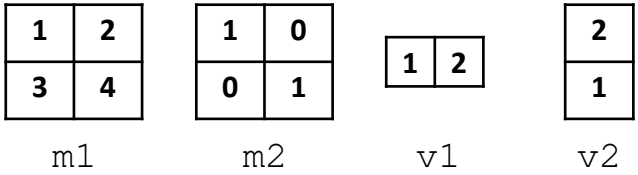
⚠: `m1*m1` is the same as `m1^2`.

m1		m2		v1		v2
1	2	1	0	1	2	1
3	4	0	1			0

`v1 * v2`	`1`
`v2 * v1`	`[1 2; 0 0]`
`m1 * v1`	*error*
`m1 * v2`	`[1; 3]`
`v1 * m1`	`[7 10]`
`v2 * m1`	*error*
`m1 + m2`	`[2 2; 3 5]`
`m1 * m2`	`[1 2; 3 4]`
`m1 * m1`	`[7 10; 15 22]`
`m1^2`	`[7 10; 15 22]`
`m2 / m1`	`[-2 1; 1.5 -0.5]`
`m2 * inv(m1)`	`[-2 1; 1.5 -0.5]`

MATLAB Operator (Element-wise Operation)

```
s1 = 2; % scalar
v1 = [1 2]; % vector
v2 = [2; 1]; % vector

m1 = [1 2; 3 4]; % matrix
m2 = [1 0; 0 1]; % matrix
```



Operator
+
-
*
/
^
./
.*
.^
'

Scalar ★ Scalar
Vector ★ Scalar
Matrix ★ Scalar
Vector ★ Vector
Vector ★ Matrix
Matrix ★ Matrix

★: Operator

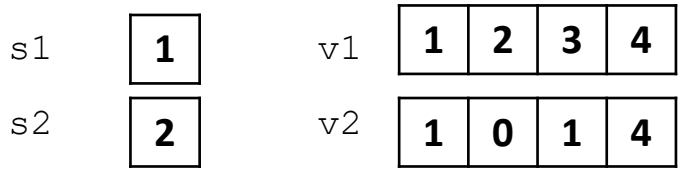
`v1 .* v1`	`[1 4]`
`v1 .* v2'`	`[2 2]`
`v1 ./ v2'`	`[0.5 2]`
`m1 .* m2`	`[1 0; 0 4]`
`m2 ./ m1`	`[1 0; 0 0.25]`
`m1 .^ s1`	`[1 4; 9 16]`
`m1 ^ s1`	`[7 10; 15 22]`

📖: `m1 ^ s1` is a matrix power that compute `m1` to `s1` power. `m1 .^ s1` is element-wise operation of `s1` power

MATLAB Operator: Relational

Challenging

```
s1 = 1; % scalar value
s2 = 2; % scalar value

v1 = [1 2 3 4]; % vector
v2 = [1 0 1 4]; % vector
```



Symbol	Role
==	Equal to
~=	Not equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to

```
s1 == s2                    0
s1 < s2                     1

v1 == s1                    [1 0 0 0]
v1 <= s2                    [1 1 0 0]
v1 ~= s2                    [1 0 1 1]

v1 == v2                    [1 0 0 1]
v1 >= v2                    [1 1 1 1]
v1 < v2                     [0 0 0 0]
```

⚠: `v1== v2` is not checking the equality of `v1` and `v2`. It is checking the element-wise equality and thus the output becomes a vector, not a logical value.

MATLAB Operator: Relational (Continue)

Challenging

```
s1 = 1; % scalar value
s2 = 2; % scalar value

m1 = [1 2; 3 4]; % matrix
m2 = [1 0; 5 4]; % matrix
```

Symbol	Role
==	Equal to
~=	Not equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to

s1 **1**

s2 **2**

1	**2**
3	**4**

m1

1	**0**
5	**4**

m2

`m1 == s1`	`[1 0;0 0]`
`m1 ~= s2`	`[1 0;1 1]`
`m1 == m2`	`[1 0;0 1]`
`m1 ~= m2`	`[0 1;1 0]`
`m1 >= m2`	`[1 1;0 1]`

MATLAB Operator: Logical

Challenging

```
s1 = 1; s2 = 0;
v1 = [1 0]; v2 = [0 0];

m1 = [1 1; 0 0];
m2 = [1 0; 0 1];
```

Symbol	Role
&&	Logical AND (scalar logical)
||	Logical OR (scalar logical)
~	Logical NOT
&	Logical AND (array)
|	Logical OR (array)

1	1
0	0

m1

1	0
0	1

m2

v1

1	0

v2

0	0

```
s1 && s1                1
s2 && s2                0
s1 || s2                1

~ s1                    0
~ s2                    1

m1 | s1                 [1 1; 1 1]

v2 | v1                 [1 0]
v2 & v1                 [0 0]

m1 | m2                 [1 1;0 1]
m1 & m2                 [1 0;0 0]
```

📖: || and && are used for scalars. For vectors or matrices, | and & are used to go through element-by-element and return `logical 1` or `0`.

Example: Count Number

```
vec = [1 3 4 2 5 10 5 9 11];

n_vec = numel(vec);

num5 = 0;
for ii=1:n_vec
    tecl_num = vec(ii);
    if tecl_num == 5
        num5 = num5 + 1;
    end
end
```

```
vec = [1 3 4 2 5 10 5 9 11];

idx = (vec == 5);
num5 = sum(idx);
```

Q: Write a script to count the number of 5 in the given vector named 'vec'. The resulting count is assigned to 'num5'.

vec

1	3	4	2	5	10	5	9	11

idx

0	0	0	0	1	0	1	0	0

num_5

2

Example: Bulls and Cows

Bulls and Cows is a mind game played by two players. In the game, a random, 4-digit number is chosen and its values are compared to those of another trial number. **All four digits of the number must be different**. If any digit in the chosen number is the exact same value and in the exact same position as any digit in the trial number, this is called a bull. If the digit is present in both the trial number and chosen number, but is not in the same location, this is called a cow.

In this figure, A: Bulls, B: Cows

Example: Bulls and Cows

```
x_true = [1 2 3 4]; % true
x_test = [3 2 5 6]; % test

numb = 0; % number of Bull
for ii=1:4
    if x_true(ii) == x_test(ii)
        numb = numb + 1;
    end
end
```

x_true	1	2	3	4
x_test	3	2	5	6
is_same	0	1	0	0

Q: Write a script to compute "Bull" and assign its value to `num_b`. The true and test sequence is in `x_true` and `x_test`, respectively .

```
x_true = [1 2 3 4]; % true
x_test = [3 2 5 6]; % test

is_same = (x_true == x_test);

numb = sum(is_same);
```

☺: Write these two lines as one line like

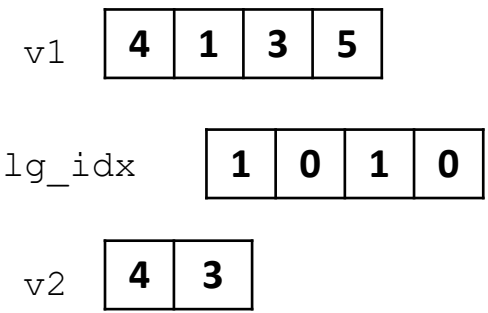
```
num_b = sum(x_true == x_test);
```

Logical Indexing (Vector)

Challenging

- Use this logical indexing to index into a vector or matrix when the index vector or matrix is the **logical** type.
- In logical indexing, you use a single, logical array for the matrix subscript.
- MATLAB extracts the matrix elements corresponding to the nonzero values of the logical array.

```
v1 = [4 1 3 5]; % vector

lg_idx = logical([1 0 1 0]);

v2 = v1(lg_idx);
```



⚠: I recommend to use the same size (the number of element) logical indexing vector.

Example: Logical Indexing

Q1: Find numbers more than or equal to `10` in `vec`

Q2: Find numbers not equal to `5` in `vec`

Q3: Find numbers larger than `3` and less than `7` in `vec`

```
1   vec = [1 3 4 2 5 10 2 9 11];
2
3   id1 = (vec >= 10);
4   vq1 = vec(id1);
5
6   id2 = (vec ~= 5);
7   vq2 = vec(id2);
8
9   id3 = (vec >3) & (vec<7);
10  vq3 = vec(id3);
```

☺: Here, all variables except for `vec` are a `logical` type. For lines 3,6,9, `()` helps readability of your code.

vec	1	3	4	2	5	10	2	9	11
id1	0	0	0	0	0	1	0	0	1
id2	1	1	1	1	0	1	1	1	1
id3	0	0	1	0	1	0	0	0	0
vq1	10	11							
vq2	1	3	4	2	10	2	9	11	
vq3	4	5							

Output from Linear and Logical Indexing

Optional, Challenging

- When the input array is a **row vector**, regardless of whether a linear indexing or logical indexing vector is a row or column vector, the output vector is a **row vector**. If the input array is a column vector, the output is a column vector.
- When the input is a matrix, if a linear indexing or logical indexing vector is a **row vector**, the output vector is a **row vector**. If the indexing vector is a column vector, the output vector is a column vector.

```
v1 = [4 1 3 5];
v2 = [4 1 3 5]';

m1 = [1 2; 3 4];

lg_idx1 = logical([1 0 1 0]);
lg_idx2 = lg_idx1';

li_idx1 = [2 3];
li_idx2 = li_idx1';
```

v1(lg_idx1)	[4 3]
v1(lg_idx2)	[4 3]
v1(li_idx1)	[1 3]
v1(li_idx2)	[1 3]
v2(lg_idx1)	[4; 3]
v2(lg_idx2)	[4; 3]
v2(li_idx1)	[1; 3]
v2(li_idx2)	[1; 3]
m1(lg_idx1)	[1 2]
m1(lg_idx2)	[1; 2]
m1(li_idx1)	[3 2]
m1(li_idx2)	[3; 2]

Example: Extracting All Odd Numbers

Q. Write the script that extracts all odd numbers in '`vec1`' and assign a resulting vector to '`odd1`'.

```
vec1 = [1 2 4 6 7 11 17 12 8];
n_vec1 = numel(vec1);

odd1 = [];
for ii=1:n_vec1
    t_num = vec1(ii);
    if rem(t_num,2) == 1
       odd1 = [odd1 t_num];
    end
end
```

```
vec1 = [1 2 4 6 7 11 17 12 8];

vec1_rem = rem(vec1,2);

logi_odd = (vec1_rem == 1);

odd1 = vec1(logi_odd);
```

📖: Here, `vec1_rem` is a numeric vector (`double` type) and `logi_odd` is a logical vector.

vec1	1	2	4	6	7	11	17	12	8
vec1_rem	1	0	0	0	1	1	1	0	0
logi_odd	1	0	0	0	1	1	1	0	0
odd1	1	7	11	17					

Logical Indexing (Matrix)

Challenging

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

ind_lg_3 = A > 3;

vec_lg_3 = A(ind_lg_3);

B(ind_lg_3) = 9;
B(~ind_lg_3) = 5;
```

1	2	3
4	5	6
7	8	9

A

0	0	0
1	1	1
1	1	1

ind_lg_3

4
7
5
8
6
9

vec_lg_3

📖: Here, the logical matrix `ind_lg_3` can be used to index the matrix. The logical matrix `ind_lg_3` is linearized before reading the elements (indexing).

1	1	1
0	0	0
0	0	0

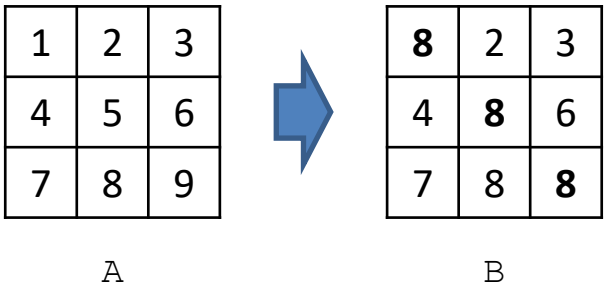
~ind_lg_3

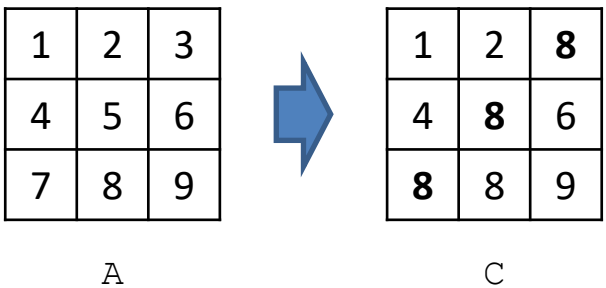
5	5	5
9	9	9
9	9	9

B

Example: Transform a Matrix

- Write a script to convert matrix A to matrix B and C.






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

B = A;
for ii=1: 3
    B(ii,ii) = 8;
end

C = A;
for ii=1:3
    col_idx = (3-ii + 1);
    C(ii, col_idx) = 8;
end
```

Example: Transform a Matrix (Continue)

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

B = A;
for ii=1: 3
    B(ii,ii) = 8;
end

C = A;
for ii=1:3
    col_idx = (3-ii + 1);
    C(ii, col_idx) = 8;
end
```

☺: `flip(x, 2)` is identical to `fliplr(x)`.

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

B = A;
d_idx = logical(eye(3));
B(d_idx) = 8;

C = A;
inv_d_idx = flip(d_idx, 2);
C(inv_d_idx) = 8;
```

1	2	3
4	5	6
7	8	9

A

1	0	0
0	1	0
0	0	1

d_idx

0	0	1
0	1	0
1	0	0

inv_d_idx

Compatible Array Sizes for Operation

Optional, Challenge

Operators and functions in MATLAB® support numeric arrays that have *compatible sizes*. **Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are either the same or one of them is 1.** MATLAB implicitly expands arrays with compatible sizes to be the same size during the execution of the element-wise operation or function.

```
s1 = 2;
m1 = [1 2; 3 4];
m2 = m1 * s1;
m2 = m1 .* [s1 s1; s1 s1];
```

1	2
3	4

2	=	2	4
6	8

1	2
3	4

.* | 2 | 2 |
|---|---|
| 2 | 2 |

= | 2 | 4 |
|---|---|
| 6 | 8 |

Q. Does it work?

1	4
2	5
3	6

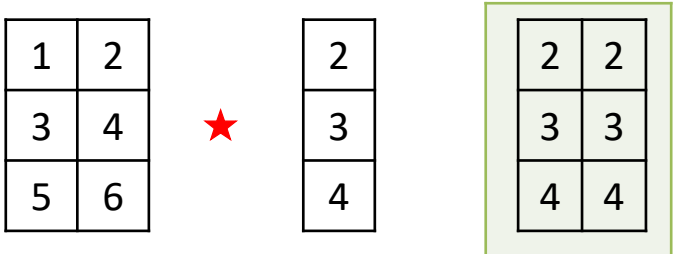
+

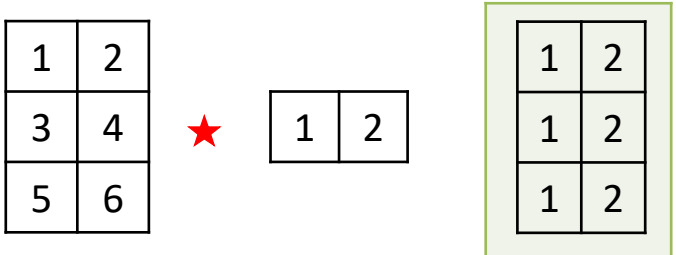
1
2
3

= ?

Compatible Array Sizes for Operation (Continue) Optional, Challenge

```
m1 = [1 2; 3 4; 5 6];
v1 = [2; 3; 4];
v2 = [1 2];
```





`m1 + v1;`	`[3 4; 6 7; 9 10]`
`m1 - v1;`	`[-1 0; 0 1; 1 2]`
`m1 .* v1;`	`[2 4; 9 12; 20 24]`
`m1 + v2;`	`[2 4; 4 6; 6 8]`
`m1 - v2;`	`[0 0; 2 2; 4 4]`
`m1 .* v2;`	`[1 4; 3 8; 5 12]`
`m1 == v1;`	`[0 1; 1 0; 0 0]`
`m1 == v2;`	`[1 1; 0 0; 0 0]`
`m1 > v1`	`[0 0; 0 1; 1 1]`
`m1 > v2`	`[0 0; 1 1; 1 1]`
`m1 * v1;`	*error*
`m1 * v2;`	*error*

MATLAB ***implicitly*** expands the vector to be the same size as the input matrix

Logical Built-in Function

- **`any`** returns true if anything in the input argument is true
- **`all`** returns true only if everything in the input argument is true
- **`find`** finds locations and returns indices

```
i1 = [1 1 1 1 1];
i2 = [0 0 0 0 0];
i3 = [0 1 0 1 0];
i4 = [0 0 0 1 0];
```

`i1,i2,i3,` and `i4` are logical vectors.

```
all(i1)                 1
all(i2)                 0
all(i3)                 0

any(i2)                 0
any(i3)                 1
any(i4)                 1

all(and(i1, i3))        0
any(and(i3, i4))        1

find(i1)                [1 2 3 4 5]
find(i3)                [2 4]
```

Example: Logical Built-in Function

Q1: Check if there is a number more than or equal to 10 in vec

Q2: Check if there is a number equal to 6 in vec

Q3: Check if there is a number larger than 3 and less than 7 in vec

If Yes, assign `true` in each variable, `q1`, `q2`, and `q3`. Otherwise, assign `false`.

```
1  vec = [1 3 4 2 5 10 2 9 11];
2
3  id1 = (vec >= 10);
4  id2 = (vec == 6);
5  id3 = (vec >3) & (vec<7);
6
7  q1 = any(id1);
8  q2 = any(id2);
9  q3 = any(id3);
```

vec	1	3	4	2	5	10	2	9	11
id1	0	0	0	0	0	1	0	0	1
q1	1								
id2	0	0	0	0	0	0	0	0	0
q2	0								
id3	0	0	1	0	1	0	0	0	0
q3	1								

Example: Bulls and Cows

Q: Write a script to compute “Cows” + “Bulls” and assign its value to `num_c`. In other word, you need to compute how many digits are present in both sequences. The true and test sequence is in `x_true` and `x_test`, respectively .

```
x_true = [1 2 3 4]; % true
x_test = [3 2 5 6]; % test

num_c = 0;
for ii=1:4
    for jj=1:4
        if x_true(ii) == x_test(jj)
            num_c = num_c + 1;
        end
    end
end
```

```
x_true = [1 2 3 4]; % true
x_test = [3 2 5 6]; % test

num_c = 0;
for ii=1:4
    if any(x_true == x_test(ii))
        num_c = num_c + 1;
    end
end
```

Example: Logical Built-in Function

Q. Find id(s) of students whose scores are more than or equal to 80 and less than 90 in their midterm. Here, the indexes of a score vector is student ids.

```
mid_score = [71 82 85 76 91 100 82 83 65 51];

cond1 = mid_score >=80;
cond2 = mid_score < 90;

idx = and(cond1, cond2); % or cond1 & cond2
cl_id = find(idx);
```

mid_score	71	82	85	76	91	100	82	83	65	51
cond1	0	1	1	0	1	1	1	1	0	0
cond2	1	1	1	1	0	0	1	1	1	1
idx	0	1	1	0	0	0	1	1	0	0
cl_id	2	3	7	8						

Example: For-loop and if vs Find vs Logical Indexing

Q. Change 1 to 5, 2 to 7, and the rest to 10 in the given vector named 'vec'

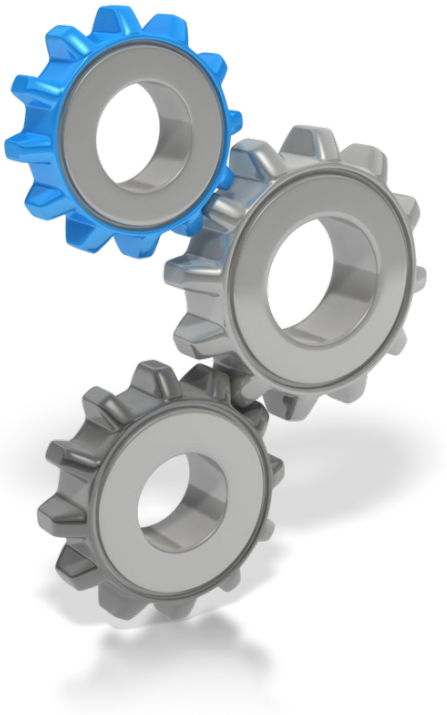
```
vec = [1 1 2 1 3 1 6 7 5];

n_vec = numel(vec);
for ii=1:n_vec
    if vec(ii) == 1
        vec(ii) = 5;
    elseif vec(ii) == 2
        vec(ii) = 7;
    else
        vec(ii) = 10;
    end
end
```

```
vec = [1 1 2 1 3 1 6 7 5];

loc1 = find(vec==1);
loc2 = find(vec==2);
locr = find((vec~=1 & vec~=2));

vec(loc1) = 5;
vec(loc2) = 7;
vec(locr) = 10;
```

```
vec = [1 1 2 1 3 1 6 7 5];

loc1 = (vec==1);
loc2 = (vec==2);
locr = ~(loc1 | loc2);

vec(loc1) = 5;
vec(loc2) = 7;
vec(locr) = 10;
```

Module 07: Function

Chul Min Yeum
Associate Professor
Civil and Environmental Engineering
University of Waterloo, Canada



UNIVERSITY OF WATERLOO
FACULTY OF ENGINEERING

Module 07: Learning Outcomes

- Explain the advantages of using functions
- Create the functions that supports various input and output variables
- Illustrate the difference between main and local functions
- Assess variable scope in the script having local functions.
- Program your own functions.

What is a Function?

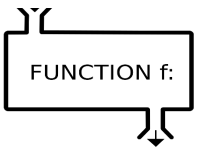
- Function is a type of procedure or routine that performs specific tasks.
- It is a block of **organized** and **reusable** scripts that is used to perform a single and related tasks.
- There are built-in functions provided by MATLAB, but users can write their own functions for reusing a certain operation.
- Advantage: **organized** (readable), **time saving** by avoiding mistake and reusing basic operations

Types of Functions

A function is a group of statements that together perform a task. In MATLAB, functions are defined in separate files. The name of the file and of the function should be the same.

Built-in Functions: Functions that are frequently used or that can take more time to execute are often implemented as executable files. These functions are called built-ins.

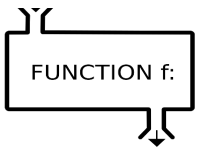
Single input



Single output

x = -1
y = abs(-1)

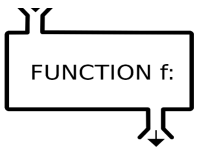
Multiple inputs



Single output

vec1 = randi(10,3,3)

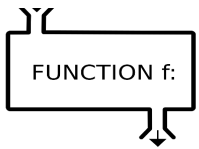
Single/Multiple inputs



Multiple output

mat1 = [1 1; 1 1];
[cz, rz] = size(mat1)

Single/Multiple inputs



No return value

disp('hello')

Example: Built-in Functions

sum

R2020a

Sum of array elements

collapse all in page

Syntax

```
S = sum(A)
S = sum(A,'all')
S = sum(A,dim)
S = sum(A,vecdim)
S = sum( __ ,outtype)
S = sum( __ ,nanflag)
```

Description

example

`S = sum(A)` returns the sum of the elements of A along the first array dimension whose size does not equal 1.

- If `A` is a vector, then `sum(A)` returns the sum of the elements.
- If `A` is a matrix, then `sum(A)` returns a row vector containing the sum of each column.
- If `A` is a multidimensional array, then `sum(A)` operates along the first array dimension whose size does not equal 1, treating the elements as vectors. This dimension becomes `1` while the sizes of all other dimensions remain the same.

example

`S = sum(A,'all')` computes the sum of all elements of `A`. This syntax is valid for MATLAB® versions R2018b and later.

example

`S = sum(A,dim)` returns the sum along dimension `dim`. For example, if `A` is a matrix, then `sum(A,2)` is a column vector containing the sum of each row.

example

`S = sum(A,vecdim)` sums the elements of `A` based on the dimensions specified in the vector `vecdim`. For example, if `A` is a matrix, then `sum(A,[1 2])` is the sum of all elements in `A`, since every element of a matrix is contained in the array slice defined by dimensions 1 and 2.

example

`S = sum( __ ,outtype)` returns the sum with a specified data type, using any of the input arguments in the previous syntaxes. `outtype` can be `'default'`, `'double'`, or `'native'`.

example

`S = sum( __ ,nanflag)` specifies whether to include or omit `NaN` values from the calculation for any of the previous syntaxes. `sum(A,'includenan')` includes all `NaN` values in the calculation while `sum(A,'omitnan')` ignores them.

Generic Function Definition

```
1  function [output arguments] = functionname (input arguments)
2
3  Your script
4
5  end
```

📖 : Texts in red and bold are syntax.

- The function header (line 1) includes
  - the reserved word **function**
  - output argument(s): listing multiple arguments separating with comma (Note that the square bracket in the output arguments is not needed if there is only one output.)
  - function name (optional: start with a capital letter for making difference from other variable names.)
  - Input argument(s): listing multiple arguments separating with comma
- The body of a function includes your script to compute values in the output arguments using the input arguments.
- Finished with the reserved word **end**

Notes on Functions

- The function header and function call have to match up:
  - the name must be the same
  - the number of input arguments must be the same
  - the number of variables in the left-hand side of the assignment should be the same as the number of output arguments

📖 : For example, if the function header is:

```
function [x,y,z] = fnname(a,b)
```

This indicates that the function is returning three outputs, so a call to the function might be (assuming a and b are numbers):

```
[g,h,t] = fnname(11, 4.3);
```

Or using the same names as the output arguments (**it doesn't matter since the workspace is not shared**):

```
[x,y,z] = fnname(11, 4.3);
```

Example: Make a Function to Compute an Absolute Value

```
% given an input scalar named
s1, compute the absolute value
of s1 and assign it to s2

s1 = 3;

if s1>=0
   s2 = s1;
end

if s1<0
   s2 = s1*-1;
end
```

Name	Value
s1	3
s2	3

Q. How to make a function called 'MyAbs' that can compute an absolute value like abs.

```
function x_abs = MyAbs(x)

if x>=0
    x_abs = x;
end

if x<0
    x_abs = x*-1;
end

end
```

```
s1 = -3;
s2 = MyAbs(s1);
```

What are the Difference between Scripts and Functions?

Script: Script files are program files with a .m extension. In these files, you write series of commands, in which you want to execute together. Scripts do not accept the inputs and do not return any outputs. They operate on data in the base workspace.

```
Input

Series of processing

Output
```

my_script.m

Function: functions file are also program files with a .m extension. Functions can accept inputs and return outputs. Internal variables are local to the function. They use separate workspace.

```
Input data

Output = MyFun(input)

Output data
```

call

```
function output = MyFun(input)

Series of processing

end
```

my_script.m **MyFun.m**

How to Save and Call Functions

Save

- The function is stored in a code file with the extension .m
- The file name **must be the same** as the function name.
- The file should be placed at the same folder where the script use the function. Otherwise, you need to add the script folder to search its path.
- To use a function you have created, you need to call that function in your script (similar to how to use built-in functions)

Call

- Calling the function should be in an assignment statement with the input and output argument(s), which is the same as the number of the input and output arguments in the function header.
- You can use any input and output variable names when you call the function because they are not sharing the Workspace (different variable scope).

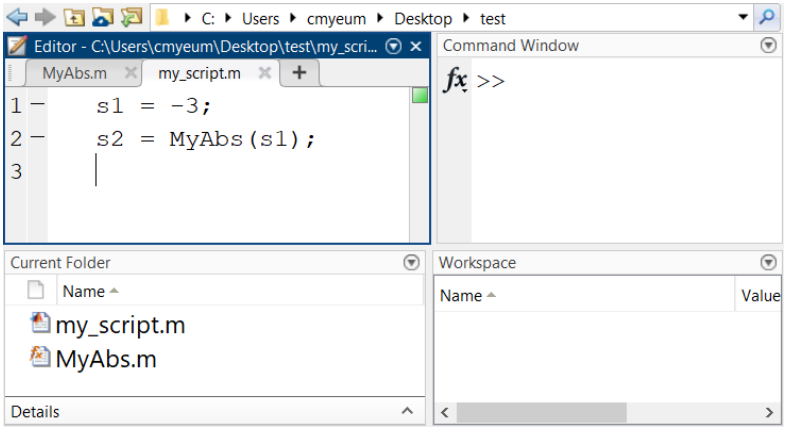
How to Use the Function

MyAbs.m

```
function x_abs = MyAbs(x)

if x>=0
    x_abs = x;
end

if x<0
    x_abs = x*-1;
end

end
```

my_script.m

```
s1 = -3;
s2 = MyAbs(s1);
```



📖 : Function and script files should be placed at the same folder. If not, we need add folders (`addpath`) where the functions are present so that MATLAB can search for the functions

Example: Bulls and Cows

Bulls and Cows is a mind game played by two players. In the game, a random, 4-digit number is chosen, and its values are compared to those of another trial number. **All four digits of the number must be different**. If any digit in the chosen number is the exact same value and in the exact same position as any digit in the trial number, this is called a bull. If the digit is present in both the trial number and chosen number, but is not in the same location, this is called a cow.

In this figure, A: Bulls, B: Cows

Example: Bulls and Cows – Compute Bulls

Q: Write a function named '`CompBulls`' to compute '`bulls`' when test and true numbers are given, which are named as '`x_true`' and '`x_test`'.

```
% This is the code that we
developed in the previous
module!

x_true = [1 2 3 4]; % true
x_test = [3 2 5 6]; % test

is_same = (x_true == x_test);
num_b = sum(is_same);
```

my_script.m

```
x_true = [1 2 3 4]; % true
x_test = [3 2 5 6]; % test

bulls = CompBulls(x_true, x_test);
```

CompBulls.m

```
function num_b = CompBulls(x_tr, x_ts)

is_same = (x_tr == x_ts);
num_b = sum(is_same);

end
```

Example: Bulls and Cows – Compute Cows

Q: Write a function named '`CompCows`' to compute '`cows`' when test and true numbers are given, which are named as '`x_true`' and '`x_test`'.

```
function cows = CompCows(x_tr, x_ts)

num_c = 0; % bulls + cows

for ii=1:4
    if any(x_tr == x_ts(ii))
        num_c = num_c + 1;
    end
end

is_same = (x_tr == x_ts);
num_b = sum(is_same); % bulls

cows = num_c - num_b; % cows

end
```

Option 1

```
x_true = [1 2 3 4]; % true
x_test = [3 2 5 6]; % test

cows = CompCows(x_true, x_test);
```

```
function cows = CompCows(x_tr, x_ts)

num_c = 0; % bulls + cows
for ii=1:4
    if any(x_tr == x_ts(ii))
        num_c = num_c + 1;
    end
end

num_b = CompBulls(x_tr, x_ts)
cows = num_c - num_b; % cows

end
```

Option 2

Example: Bulls and Cows – Compute Bulls and Cows

Q: Write a function named 'CompBC' to compute 'bulls' and 'cows' when test and true numbers are given, which are named as 'x_true' and 'x_test'.

```
function [bulls, cows] = CompBC(x_true, x_test)

bulls = CompBulls(x_true, x_test);
cows = CompCows(x_true, x_test);

end
```

```
function num_b = CompBulls(x_tr, x_ts)

is_same = (x_tr == x_ts);
num_b = sum(is_same);

end
```

📖: Assume that each function is stored in its m-file with the file name identical to its function name.

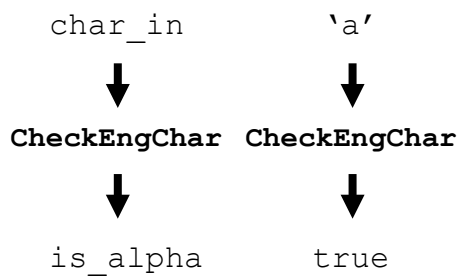
```
function cows = CompCows(x_tr, x_ts)

num_c = 0; % bulls + cows
for ii=1:4
    if any(x_tr == x_ts(ii))
        num_c = num_c + 1;
    end
end

num_b = CompBulls(x_tr, x_ts)
cows = num_c - num_b; % cows

end
```

Example: Is this Character English Alphabet?

Q: Given a character named 'one_char', write a function to check if 'one_char' is in the English alphabet. If yes, assign true to 'is_alpha', otherwise false.

```
one_char = 'a'; % input character

char_db = double(one_char);
cond1 = and(65 <= char_db, char_db <=90); % upper case
cond2 = and(97 <= char_db, char_db <=122); % lower case

is_alpha = or(cond1, cond2); % output
```

```
function is_alpha = CheckEngChar(char_in)

char_db = double(char_in);
cond1 = and(65 <= char_db, char_db <=90);
cond2 = and(97 <= char_db, char_db <=122);
is_alpha = or(cond1, cond2); % output

end
```



Example: How Many English Alphabets in a Character Vector?

```
char_vec = 'asbdec#43!@3';
num_char = numel(char_vec);

num_alpha = 0;
for ii=1:num_char

    one_char = char_vec(ii);
    char_db = double(one_char);
    cond1 = and(65 <= char_db, char_db <=90);
    cond2 = and(97 <= char_db, char_db <=122);

    is_alpha = or(cond1, cond2); % output

    if is_alpha == 1
        num_alpha = num_alpha + 1;
    end
end
```

Q: Given a character vector named '`char_vec`', write a script to count the number of English alphabet characters in '`char_vec`'. Assign the number to '`num_alpha`'.

☺: If you pack the script as a function, the code can be shortened and readable.

```
char_vec = 'asbdec#43!@3';
num_char = numel(char_vec);

num_alpha = 0;
for ii=1:num_char
    is_alpha = CheckEngChar(char_vec(ii));

    if is_alpha
        num_alpha = num_alpha + 1;
    end
end
```

Example: Is There Non-Alphabet Character in a Character Vector?

```
char_vec = 'asbdec#43!@3';
num_char = numel(char_vec);

is_all_alpha = true;
for ii=1:num_char

    one_char = char_vec(ii);
    char_db = double(one_char);
    cond1 = and(65 <= char_db, char_db <=90);
    cond2 = and(97 <= char_db, char_db <=122);

    is_alpha = or(cond1, cond2);

    if is_alpha ~= true
        is_all_alpha = false;
        break;
    end
end
```

Q: Given a character vector named '`char_vec`', write a script to check if the vector only contain English alphabet characters. If yes, assign `true` to '`is_all_alpha`', otherwise assign `false`.

```
char_vec = 'asbdec#43!@3';
num_char = numel(char_vec);

is_all_alpha = true;
for ii=1:num_char
    is_alpha = CheckEngChar(char_vec(ii));

    if is_alpha ~= true
        is_all_alpha = false;
        break;
    end
end
```

Example: Vectorization

```
function is_alpha = CheckEngChar(char_in)

char_db = double(char_in);

cond1 = and(65 <= char_db, char_db <=90);
cond2 = and(97 <= char_db, char_db <=122);
is_alpha = or(cond1, cond2); % output

end
```

📖 : Here, the input does not have to be a single character. The script supports a character vector as an input. Then, the output will be the same size as its input vector.

Q. How many English alphabet?

```
char_vec = 'asbdec#43!@3';

is_alpha = CheckEngChar(char_vec);

num_alpha = sum(is_alpha);
```

Q. Is there non-alphabet character in a character vector?

```
char_vec = 'asbdec#43!@3';

is_alpha = CheckEngChar(char_vec);

is_all_alpha = ~all(is_alpha);
```

Example: How Many Upper-Case and Lower-Case Letters?

```
function [num_up, num_low] = CheckCharCase(char_in)

char_db = double(char_in);

lg_up = and(65 <= char_db, char_db <=90); % index upper case
lg_low = and(97 <= char_db, char_db <=122); % index lower case

num_up = sum(lg_up);
num_low = sum(lg_low);

end
```

```
>> char_vec = 'asbdEc#43!@3';
>> [num_up, num_low] = CheckCharCase(char_vec)

num_up =

     1


num_low =

     5
```

📖 : The function designed here computes numbers of both upper and lower-case letters when one input vector is provided.

Example: How Many Lower-Case and Upper-Case Letters? (Continue)

Challenging

```
function num_char = CheckCharCase(char_in, lt_case)

char_db = double(char_in);

if isequal(lt_case, 'upper')
    lg_up = and(65 <= char_db, char_db <=90);
    num_char = sum(lg_up);

elseif isequal(lt_case, 'lower')
    lg_low = and(97 <= char_db, char_db <=122);
    num_char = sum(lg_low);

else
   error('wrong second argument of lt_case');
end

End
```

📖 : We can make the function to selectively compute either lower- or upper-case letter by providing the second argument.

```
>> char_vec = 'asbdEc#43!@3';
>> num_low_char = CheckCharCase(char_vec, 'lower')

num_low_char  =

     5
```

Local Function

MATLAB® program files can contain code for more than one function. In a function **script** file, the first function in the file is called the **main function**. This function is visible to functions in other files, or you can call it from the command line. Additional functions within the file are called **local functions**, and they can occur in any order after the main function.

MyFun.m

```
function out = MyFun(in)

call MyFunSub1
call MyFunSub2

processing using outputs
from sub functions

end
```

MyFunSub1.m

```
function out = MyFunSub1 (in)

a series of processing

end
```

MyFunSub2.m

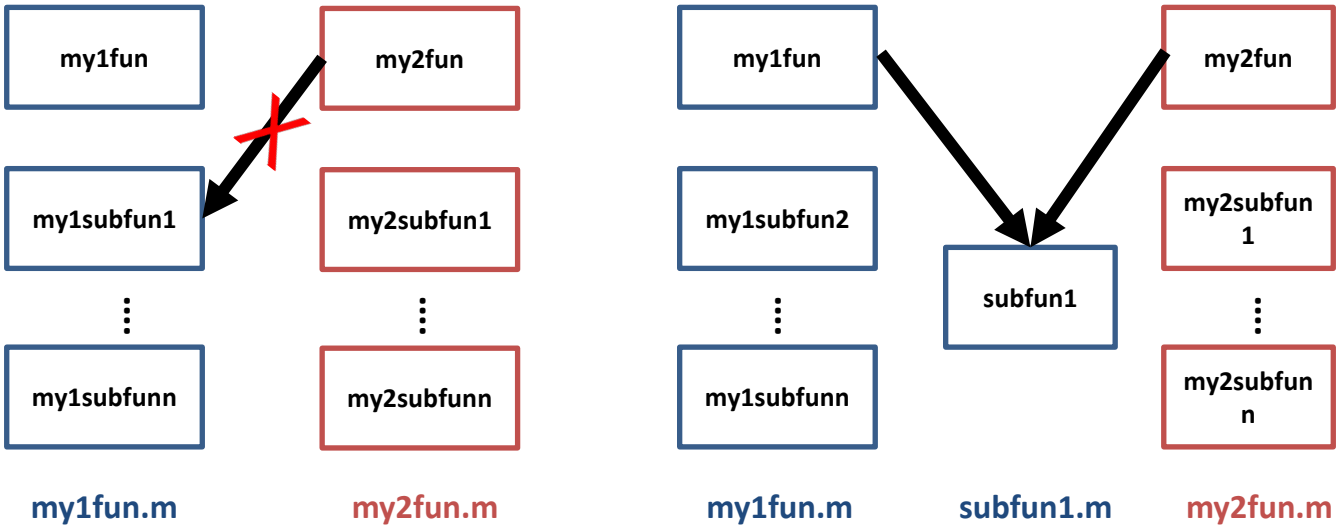
```
function out = MyFunSub2 (in)

a series of processing

end
```

MyFunAll.m

```
function out = MyFunAll(in)        <-- main function
call MyFunSub1
call MyFunSub2
processing using outputs from sub functions
end

function out = MyFunSub1 (in)      <-- local function
a series of processing
end

function out = MyFunSub2 (in)      <-- local function
a series of processing
end
```

Call Local Functions

Local functions are only visible to other functions **in the same file**. They are equivalent to subroutines in other programming languages and are sometimes called subfunctions. For example, 'subfun1.m' below should be stored in a separate file to be accessed by other functions.



Example: Bulls and Cows – Compute Bulls and Cows

CompBC.m

```
function [bulls, cows] = CompBC(x_true, x_test)
bulls = CompBulls(x_true, x_test);
cows = CompCows(x_true, x_test);
end

function num_b = CompBulls(x_tr, x_ts)

is_same = (x_tr == x_ts);
num_b = sum(is_same);

end

function num_c = CompCows(x_tr, x_ts)

num_c = 0;
for ii=1:4
    if any(x_tr == x_ts (ii))
        num_c = num_c + 1;
    end
end
bulls = CompBulls(x_tr, x_ts);
num_c = num_c - bulls;

end
```

```
>> x_tr = [1 2 3 6]; % true
>> x_ts = [3 2 5 6]; % test
>> [bls, cws] = CompBC(x_tr, x_ts)
bls =

     2


cws =

     1
```

my_script.m

```
x_tr = [1 2 3 6]; % true
x_ts = [3 2 5 6]; % test

[bls, cws] = CompBC(x_tr, x_ts)
```

Add Functions to Scripts

- MATLAB® scripts, including live scripts, can **contain code to define functions.** These functions are called local functions. Local functions are useful if you want to reuse code within a script. By adding local functions, you can avoid creating and managing separate function files. They are also useful for experimenting with functions, which can be added, modified, and deleted easily as needed. Functions in scripts are supported **in R2016b or later**.
- Local functions are only visible within the file where they are defined, both to the script code and other local functions within the file. They are not visible to functions in other files and cannot be called from the command line. They are equivalent to subroutines in other programming languages and are sometimes called subfunctions.

Example: Add Functions to Your Script

```
x_tr = [1 2 3 6]; % true
x_ts = [3 2 5 6]; % test
[bulls, cows] = CompBC(x_tr, x_ts)

function [bulls, cows] = CompBC(x_true, x_test)

bulls = CompBulls(x_true, x_test);
cows = CompCows(x_true, x_test);

end

function num_b = CompBulls(x_tr, x_ts)

is_same = (x_tr == x_ts);
num_b = sum(is_same);

end

function num_c = CompCows(x_tr, x_ts)
num_c = 0;
for ii=1:4
    if any(x_tr == x_ts (ii))
        num_c = num_c + 1;
    end
end
bulls = CompBulls(x_tr, x_ts);
num_c = num_c - bulls;

end
```

📖: We can write both script and function at the same m-file. In this case, all functions become local function, which means these cannot be accessed from the other files.

📖: In this case, the script name does not have to be one of local function names.

☺: This is very useful when you test your function. Before 2016, we need to have a separate script to test function or using a command window.

Variable Scope

- The ***scope*** of any variable is the workspace in which it is valid.
- The workspace created in the Command Window is called the ***base workspace***.
- Scripts also create variables in the base workspace. That means that variables created in the Command Window can be used in scripts and vice versa
- Functions do not use the base workspace. Every function has **its own function workspace.** Each function workspace is separate from the base workspace and all other workspaces to protect the integrity of the data. Even local functions in a common file have their own workspaces. Variables specific to a function workspace are called local variables. Typically, local variables do not remain in memory from one function call to the next.

Change Workspace and Variable Scope

Challenging

```
1   x_tr = [1 2 3 6]; % true
2   x_ts = [3 2 5 6]; % test
3
4   [bls, cws] = CompBC(x_tr, x_ts);
5
6   disp('Done!');
7
8   function [bulls, cows] = CompBC(x_true, x_test)
9   bulls = CompBulls(x_true, x_test);
10  cows = CompCows(x_true, x_test);
11  end
12
13  function num_b = CompBulls(x_tr, x_ts)
14  is_same = (x_tr == x_ts);
15  num_b = sum(is_same);
16  end
17
18  function num_c = CompCows(x_tr, x_ts)
19  num_c = 0;
20  for ii=1:4
21      if any(x_tr == x_ts(ii))
22          num_c = num_c + 1;
23      end
24  end
25  bulls = CompBulls(x_tr, x_ts);
26  num_c = num_c - bulls;
27  end
```

Workspace - CompBulls

Name	Value
is_same	1x4 logical
num_b	2
x_tr	[1,2,3,6]
x_ts	[3,2,5,6]

Stop at line 16

Workspace - CompBC

Name	Value
bulls	2
x_test	[3,2,5,6]
x_true	[1,2,3,6]

Stop at line 10

Workspace - CompCows

Name	Value
ii	4
num_c	3
x_tr	[1,2,3,6]
x_ts	[3,2,5,6]

Stop at line 25

Workspace - CompBulls

Name	Value
is_same	1x4 logical
num_b	2
x_tr	[1,2,3,6]
x_ts	[3,2,5,6]

Stop at line 16

Workspace - CompBC

Name	Value
bulls	2
cows	1
x_test	[3,2,5,6]
x_true	[1,2,3,6]

Stop at line 11

Workspace - my_script

Name	Value
bls	2
cws	1
x_tr	[1,2,3,6]
x_ts	[3,2,5,6]

Stop at line 6

Example: Variable Scope (Which is a Valid Code ?)

```
in_vec = [1 2 3 4];
n_elem = numel(in_vec);
mean_val = myfun_mean(in_vec);

function mean_val = myfun_mean(vec)

mean_val = sum(vec)/n_elem;

end
```

Error

```
vec = [1 2 3 4];
val1 = myfun_mean(vec);

function mean_val = myfun_mean(vec)

n_elem = numel(vec);
mean_val = sum(vec)/n_elem;

end
```

No Error

```
vec = [1 2 3 4];
n_elem = numel(vec);
m_val = mf_mean(vec);

function m_val = mf_mean(vec, n_elem)

m_val = sum(vec)/n_elem;

end
```

Error

```
vec = [1 2 3 4];
n_elem = numel(vec);
mean_val = myfun_mean(vec);

function mean_val = myfun_mean(vec)

n_elem = numel(vec);
mean_val = sum(vec)/n_elem;

end
```

No error

Module 08: Plotting

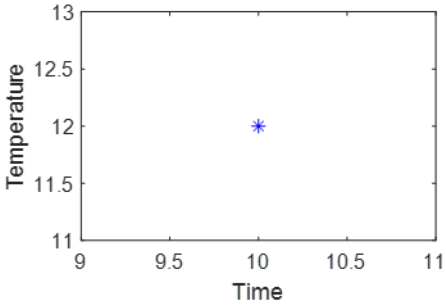
Chul Min Yeum
Associate Professor
Civil and Environmental Engineering
University of Waterloo, Canada

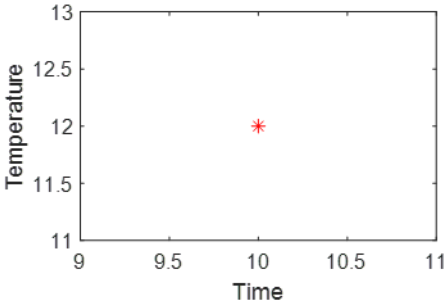
UNIVERSITY OF WATERLOO
FACULTY OF ENGINEERING

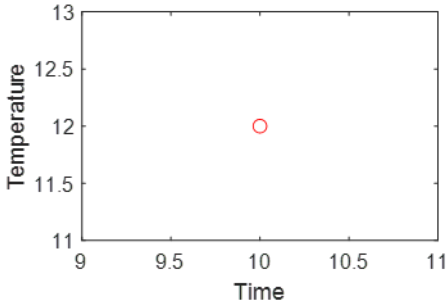
Module 08: Learning Outcomes

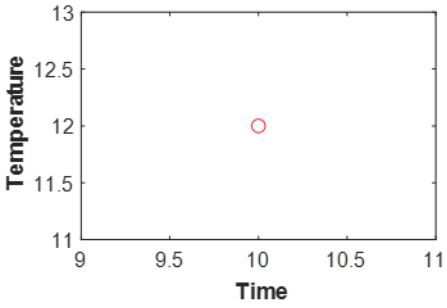
- Create a graph using the '`plot`' function
- Customize your plot: marker color & type, line color, labels, set range, etc
- Draw multiple plots or graphs in a single figure window
- Save your graph

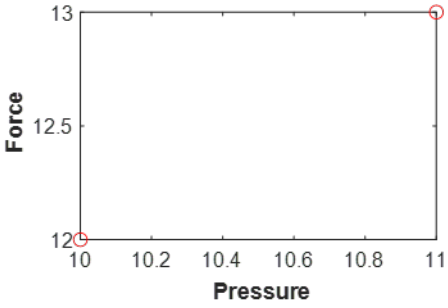
Plot a Point (Overview)

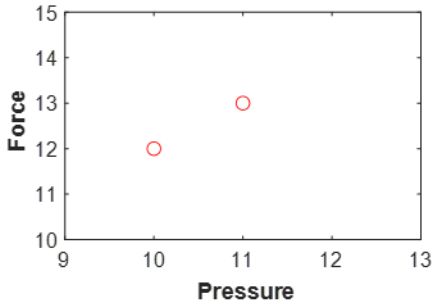






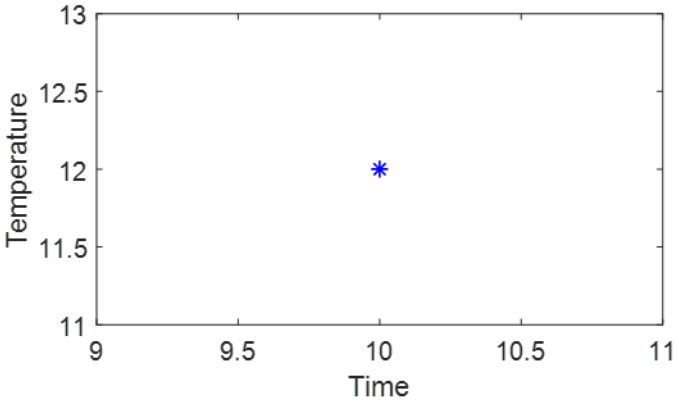




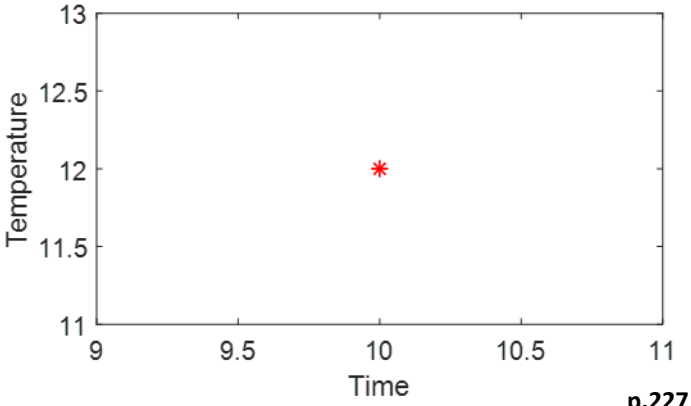


Plot a Point: Change Marker Color

```
x = 10;
y = 12;
plot(x,y, 'b*')

xlabel('Time');
ylabel('Temperature');
```



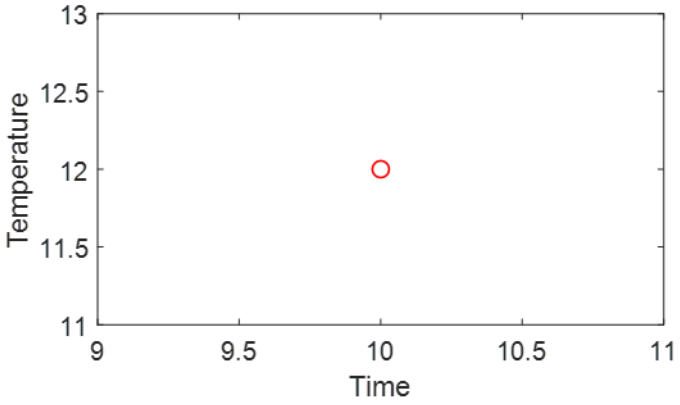

```
x = 10;
y = 12;
plot(x,y, 'r*')

xlabel('Time');
ylabel('Temperature');
```



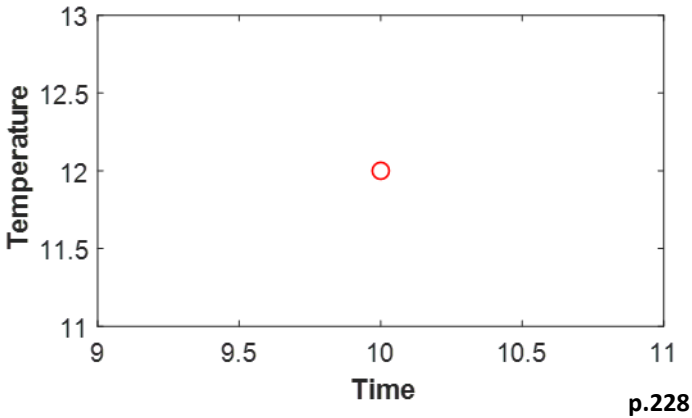
📖: Color can be specified as a character. `'r'`: red, `'b'`: blue, `'m'`: magenta, `'g'`: green, `'k'`: black, `'y'`: yellow.

Plot a Point: Change Marker Type & Make Labels as a Bold Font

```
x = 10;
y = 12;
plot(x,y, 'ro')

xlabel('Time');
ylabel('Temperature');
```



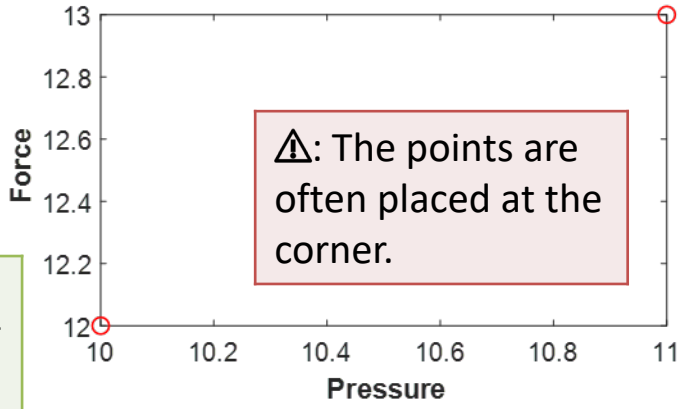

```
x = 10;
y = 12;
plot(x,y, 'ro')

xlabel('\bf Time');
ylabel('\bf Temperature');
```



📖 : Marker type can be specified as a character. `'o'`: circle, `'x'`: cross, `'*'`: asterisk, `'+'`: plus sign

Plot Point(s): Add Point(s) & Change Min. and Max Value in X & Y-axis

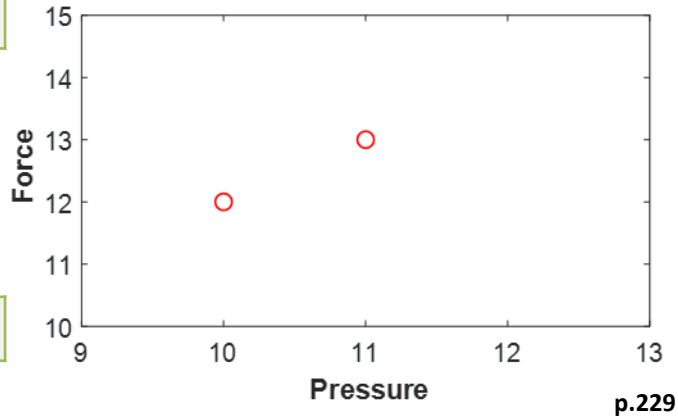
```
x = [10 11];
y = [12 13];
plot(x,y, 'ro')

xlabel('\bf Pressure');
ylabel('\bf Force');
```



⚠: The points are often placed at the corner.

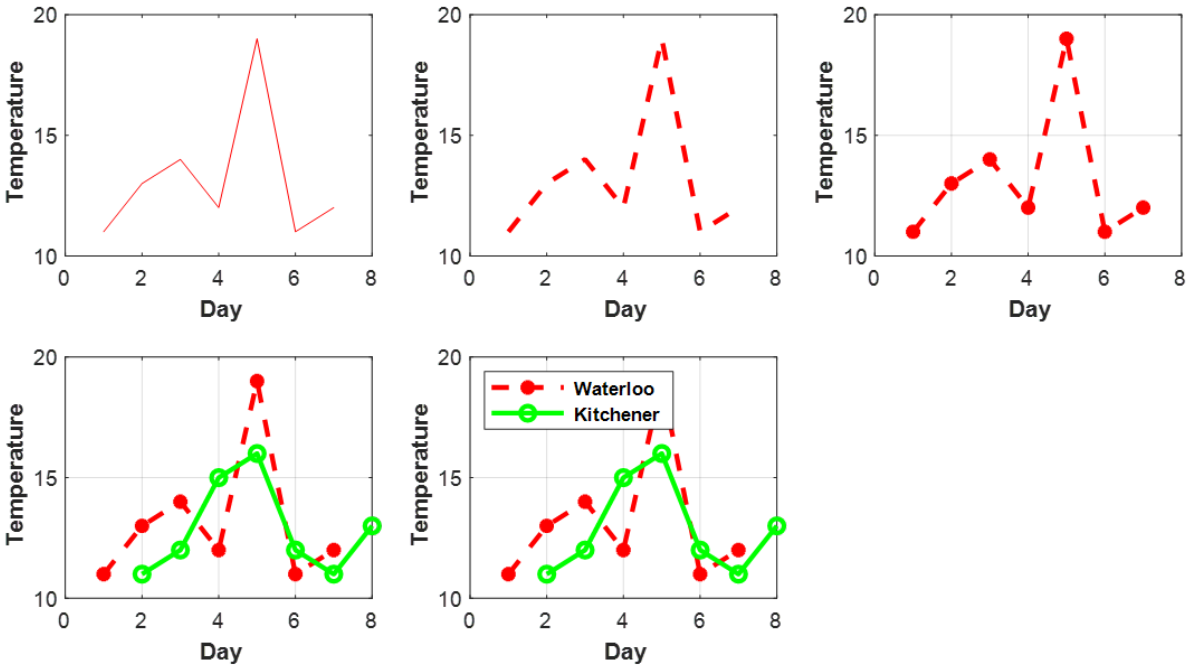
📖: `x` and `y` are inputted as a vector to plot multiple points.

```
x = [10 11];
y = [12 13];
plot(x,y, 'ro')

xlabel('\bf Pressure');
ylabel('\bf Force');
axis([9 13 10 15]);
```



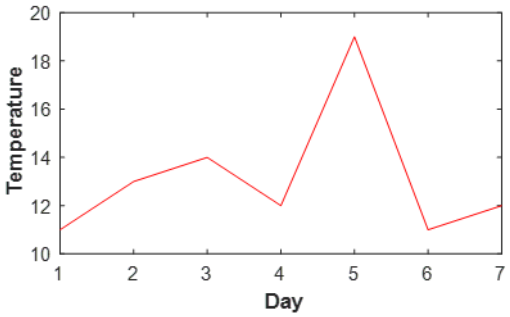
📖: `axis([xmin xmax ymin ymax])`

Plot a Vector (Overview)



Plot a Vector: Change a Line as a Dashed Line & its Line Width

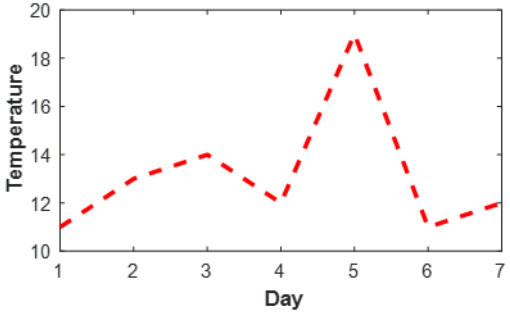
```
x = 1:7;
y = [11 13 14 12 19 11 12];
plot(x,y, 'r')

xlabel('\bf Day');
ylabel('\bf Temperature');
```




```
x = 1:7;
y = [11 13 14 12 19 11 12];
plot(x,y, '--r', 'linewidth', 2)

xlabel('\bf Day');
ylabel('\bf Temperature');
```

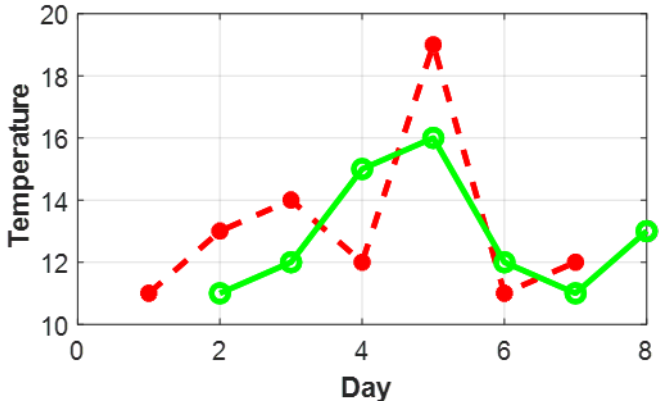
📖: `Plot(… ,'linewidth', value)` is to change line width. As increasing value, the line width is thicker.

📖: If you include color without specifying marker type, the points are connected (default: solid line). Line style character can specify the line style. `'-'`: solid, `'--'`: dash, `':'`: dotted, `'-.'`: dash-dot.



Plot a Vector: Add Data Point Marker and Grid & Add Another Line

```
x = 1:7;
y = [11 13 14 12 19 11 12];
plot(x,y, '--r*', 'linewidth', 2)

hold on;
x2 = 2:8;
y2 = [11 12 15 16 12 11 13];
plot(x2,y2, '-go', 'linewidth', 2)

xlabel('Day'); grid on;
ylabel('Temperature');
```



📖: The way of plotting two graphs in one plot is to overlay (combine) two graphs using `hold on`.

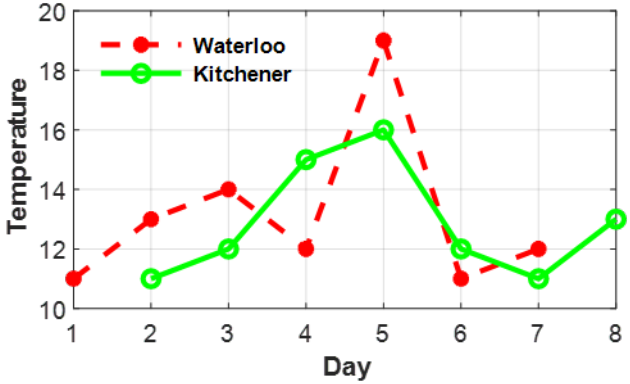
📖: When you combine line and marker style with a color like `'--r*'`, you can plot the markers on the line, which are used for drawing the corresponding line.

📖: `grid on` is to display a grid.

☺: By selecting different line styles, color, line width, marker styles, you can draw nice looking graphs!

Plot a Vector: Add Labels & Their Configuration

```
x = 1:7; y = [11 13 14 12 19 11 12];
plot(x,y, '--r*', 'linewidth', 2); hold on

x2 = 2:8;  y2 = [11 12 15 16 12 11 13];
plot(x2,y2, '-go', 'linewidth', 2)

legend('\bf Waterloo', '\bf Kitchener');
legend('location', 'northwest')
legend('boxoff')

xlabel('\bf Day');grid on
ylabel('\bf Temperature');
```



📖: You can create a legend with descriptive labels for each plotted data using `legend(label1, ...)`

📖: You can remove legend background and outline using `legend('boxoff')`.

📖: The graphs might overlap with the legends. Then, you can change the location of the legend using `legend(..., 'Location', lcn)`. Here, `lcn` includes `'north'`, `'south'`, .. `'northwest'`, etc.

Trigonometric Functions

Function	Description	Script	Value
`deg2rad(x)`	Converts degrees to radians	`deg2rad(90)` `deg2rad(180)`	`1.5708` `3.1416`
`rad2deg(x)`	Converts radians to degrees	`rad2deg(pi)` `rad2deg(pi/2)`	`180` `90`

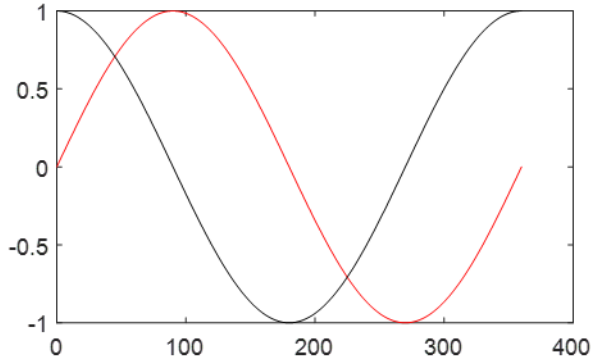
`sin(x)`
`sind(x)`
	Find the sine of **x** when **x** is expressed in **radian/degree**	`sin(pi)`
		`sind(90)`
		`sin(90)`
			`0`
			`1`
			`0.8940`

`cos(x)`
`cosd(x)`
	Find the cosine of **x** when x is expressed in radian/degree	`cos(pi)`
		`cosd(90)`
		`cos(-90)`
			`-1`
			`0`
			`-0.4481`

`asin(x)`
`asind(x)`
	Find the inverse sin of **x** and reports the result in radian (-pi .. pi)/degree(-180 .. 180)	`asin(1)`
		`asind(1)`
		`asind(0.5)`
			`1.5708`
			`90`
			`30`

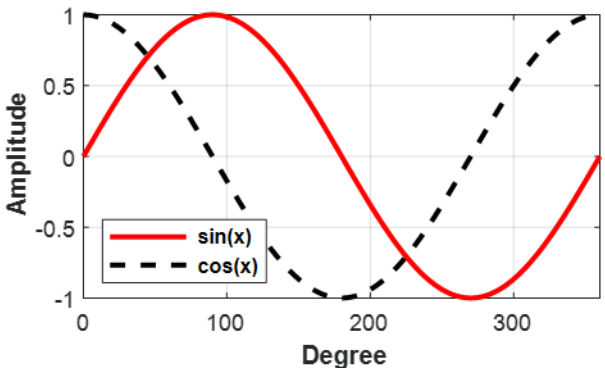
📖: `'-d'` at the end of the name stands for degree.

Example: How to Change and Draw a Graph



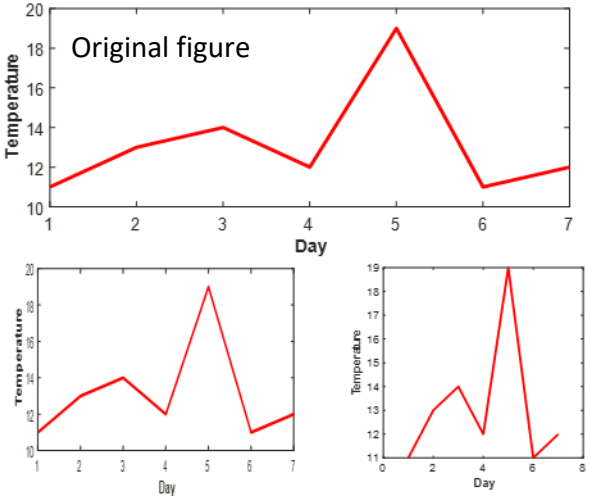

```
x = 0:360;
y1 = sind(x);
y2 = cosd(x);
plot(x, y1,'r'); hold on;
plot(x, y2, ':k')
```

Q. How to change the style of the graph?




```
x = 0:360;
y1 = sind(x);
y2 = cosd(x);
plot(x, y1,'r', 'linewidth', 2); hold
on;
plot(x, y2, '--k', 'linewidth', 2)

legend('\bf sin(x)', '\bf cos(x)');
legend('location', 'southwest')

xlabel('\bf Degree');grid on
ylabel('\bf Amplitude')
axis([0 360 -1 1 ]);
```

Figure Size Control



Resizing with/without keeping aspect ratio of graph contents

```
x = 1:7;
y = [11 13 14 12 19 11 12];

figure(1);
plot(x,y, 'r', 'linewidth', 2)
xlabel('\bf Day');
ylabel('\bf Temperature');
set(gcf, 'Position', [100 100 500 200]);

figure(2);
plot(x,y, 'r', 'linewidth', 2)
xlabel('\bf Day');
ylabel('\bf Temperature');
set(gcf, 'Position', [100 100 300 300]);
```

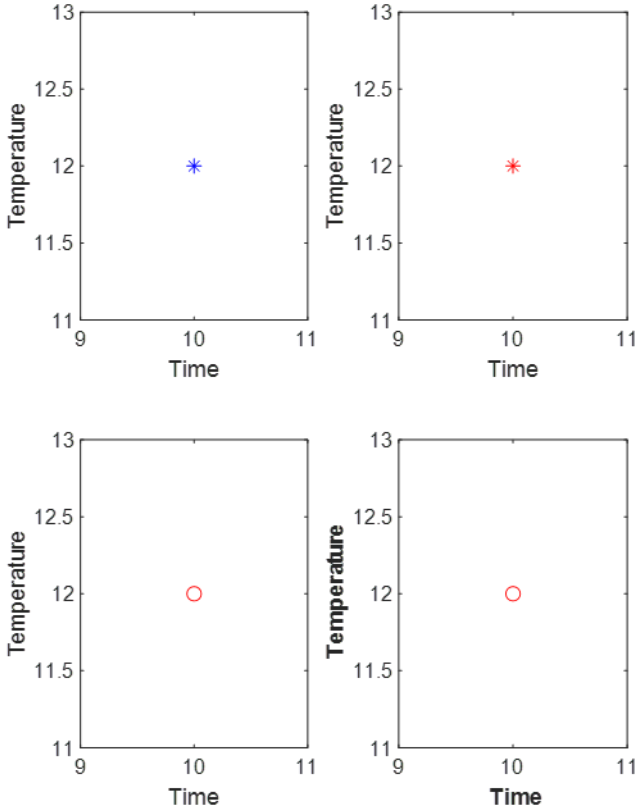
📖: `figure(n)` create a new figure window in which name is `n`.

📖: When you resize the figure after copying your image, you cannot keep the aspect ratio of graph contents or the original figure. There are two ways to resize figure with the original aspect ratio. Change the figure window before copying the figure or use '`position`' argument.

```
set (gcf, 'position', [left bottom width height])
```

Here, `[left bottom ...]` indicates the location of left-bottom corner of your figure window.

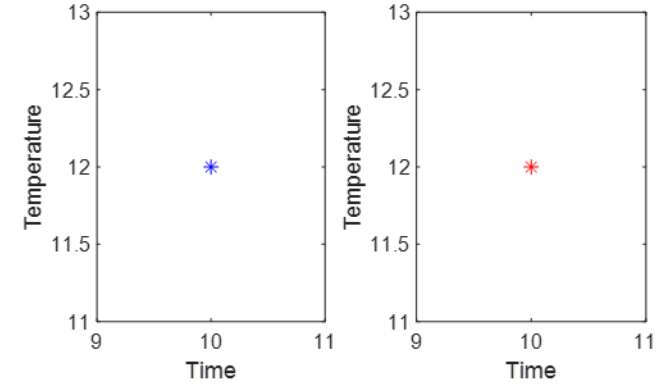
Multiple Figures in One Figure Window

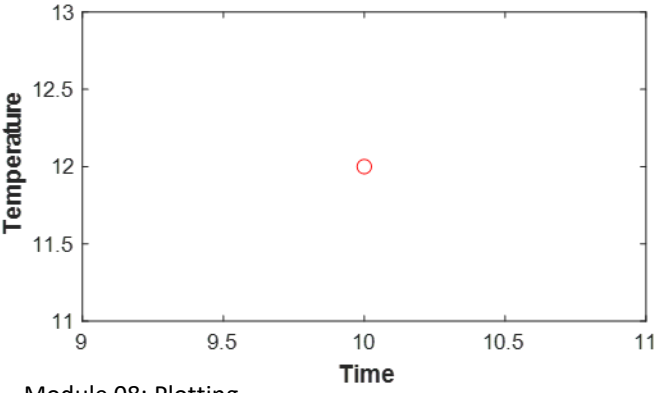



```
figure(1);

x = 10; y = 12;

subplot(2,2,1);
plot(x,y, 'b*')
xlabel('Time'); ylabel('Temperature');

subplot(2,2,2);
plot(x,y, 'r*')
xlabel('Time'); ylabel('Temperature');

subplot(2,2,3);
plot(x,y, 'ro')
xlabel('Time'); ylabel('Temperature');

subplot(2,2,4);
plot(x,y, 'ro')
xlabel('\bf Time');
ylabel('\bf Temperature');

set(gcf,'Position', [100 100 400 500]);
```

Multiple Figures (Continue)

Challenging






```
figure(1);
x = 10; y = 12;

subplot(2,2,1);
plot(x,y, 'b*')
xlabel('Time'); ylabel('Temperature');

subplot(2,2,2);
plot(x,y, 'r*')
xlabel('Time'); ylabel('Temperature');

subplot(2,2,[3 4]);
plot(x,y, 'ro')
xlabel('\bf Time');
ylabel('\bf Temperature');

set(gcf,'Position', [100 100 400 500]);
```

subplot(2,2,n)

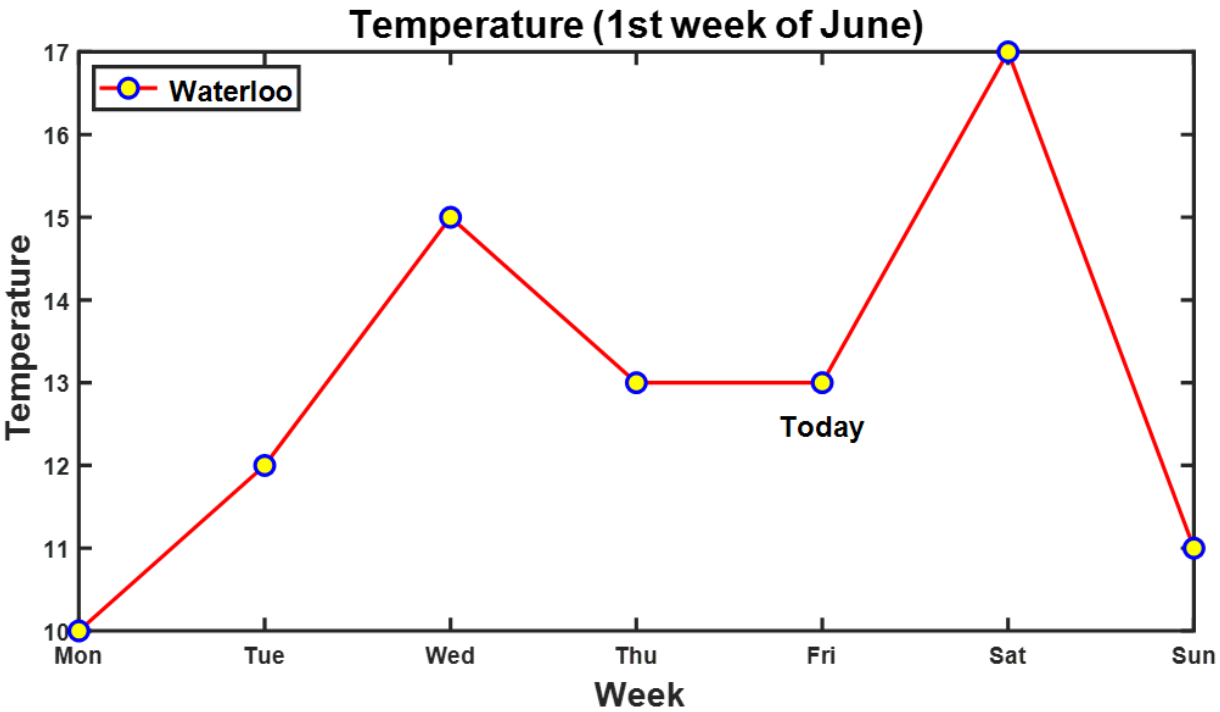
1	2
3	4

subplot(2,3,n)

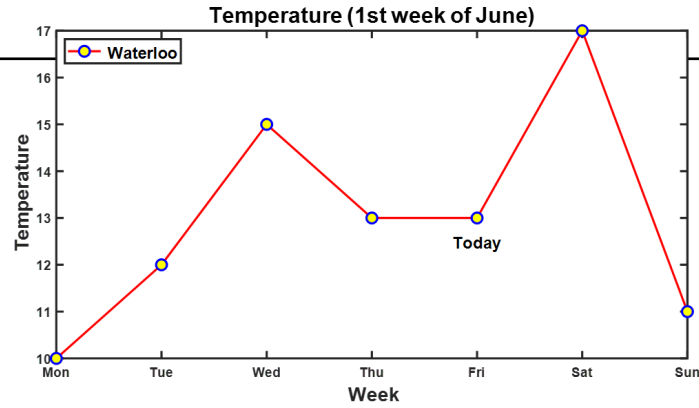
1	2	3
4	5	6

Advanced Plotting

Challenging



Advanced Plotting (Continue)

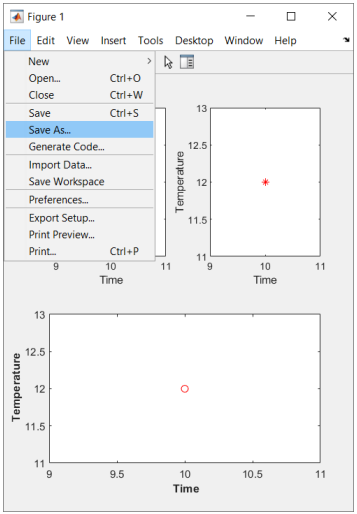


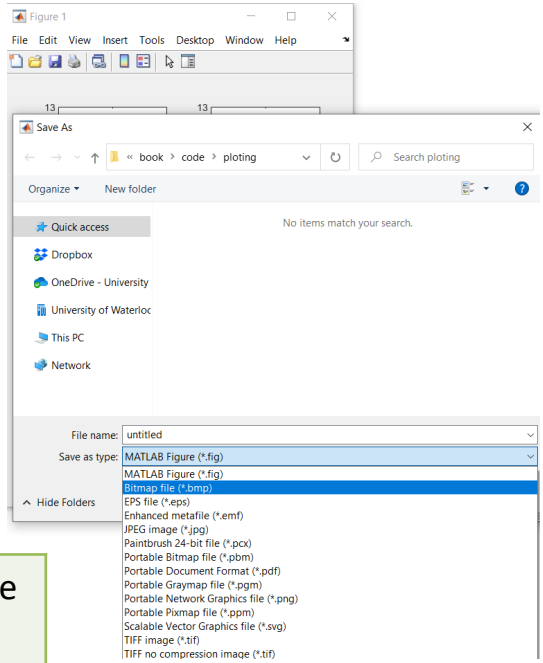

```
x = 1:7; y = [10 12 15 13 13 17 11];
plot(x,y, '-ro', ...
    'LineWidth', 2, ...
    'MarkerSize', 10, ...
    'MarkerEdgeColor', 'b', ...
    'MarkerFaceColor', 'y')

text(x(5), y(5)-0.5, 'Today', ...
    'FontSize' , 15, ...
    'FontWeight', 'bold', ...
    'HorizontalAlignment', 'center');

xticks(1:7);
xticklabels({'Mon','Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun'});

legend('\bf Waterloo', 'Location', 'northwest', 'FontSize', 15);
set(gca, 'Fontsize', 12, 'FontWeight', 'bold');
set(gca, 'LineWidth', 2);
set(gcf, 'Position', [100 100 1000 500]);

xlabel('\bf Week', 'fontsize', 18);
ylabel('\bf Temperature', 'fontsize', 18);
title('Temperature (1st week of June)', 'fontsize', 20);
```

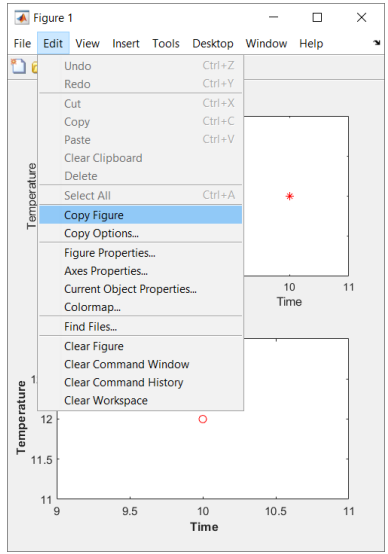
📖: `...` is to continue long statements on multiple lines

p.240

Save Your Graphs







📖 : Save a graph in an image format: Go to “File-Save As” and select file types to save the figure.

📖 : You can copy and paste a graph.

Module 09: Data Structure

Chul Min Yeum
Associate Professor
Civil and Environmental Engineering
University of Waterloo, Canada

UNIVERSITY OF WATERLOO
FACULTY OF ENGINEERING

Module 09: Learning Outcomes

- Explain cell and structure data type
- Describe a problem that require these new data type
- Access values in cell and structure array
- Illustrate the difference between cell and structure data type
- Understand vectorization of the operations involving cell arrays

Cell Array

- A **cell array** is a **data type** that can **store different types and sizes of values** in its elements (cell).
- For example, when you store combinations of text and numbers as one variable which are often read from **spreadsheets**, this cell type is very useful.
- A cell array could be a vector (row or column) or matrix of cells.
- A cell array is an array, so indices are used to refer to the elements.
- The syntax used to create a cell array is curly braces { } instead of [ ] (e.g. `var = {'abc', 3, [1 2 3]}`)
- The cell function can also be used to preallocate empty cells by passing the dimensions of the cell array, e.g. `cell(4,2)`.
- The contents in cells can be access (write and read) by indexing with curly braces {}.
- () is to **refer a set of cells**, for example, to define a subset of the cell array.

Create Cell Array

```
cl_info = {'Chul Min', 'CIVE', 1076123, [80 90], [70 30 50], ...
[4 5 1 2], '4B'};
```

option 1

```
cl_info = cell(1,7);

cl_info{1,1} = 'Chul Min';
cl_info{1,2} = 'CIVE';
cl_info{1,3} = 1076123;
cl_info{1,4} = [80 90];
cl_info{1,5} = [70 30 50];
cl_info{1,6} = [4 5 1 2];
cl_info{1,7} = '4B';
```

option 2

☺: Option 2 is more general when you put multiple input data.

```
>> cl_info

cl_info =

  1×7 cell array

  Columns 1 through 3

    {'Chul Min'}    {'CIVE'}    {[1076123]}

  Columns 4 through 5

    {1×2 double}    {1×3 double}

  Columns 6 through 7

    {1×4 double}    {'4B'}
```

📖: `...` is to continue long statements on multiple lines

Example: How to Create and Access Cell Array

Suppose that you want to store the following data using a cell array:

Name	Program	ID	Exam	Quiz	Homework	Cohort
Chul Min	CIVE	1076123	[80 90]	[70 30 50]	[4 5 2 1]	4B
Noreen	ENVE	3026327	[100 70]	[10 20 70]	[2 7 8 9]	2A
Vlad	ENVE	2046426	[50 90]	[90 60 80]	[1 2 6 2]	2A

```
cl_info = cell(3,7);

cl_info{1,1} = 'Chul Min';
cl_info{1,2} = 'CIVE';
cl_info{1,3} = 1076123;
cl_info{1,4} = [80 90];
cl_info{1,5} = [70 30 50];
cl_info{1,6} = [4 5 1 2];
cl_info{1,7} = '4B';

cl_info{2,1} = 'Noreen';
cl_info{2,2} = 'ENVE';
```

```
% continue
cl_info{2,3} = 3026327;
cl_info{2,4} = [100 70];
cl_info{2,5} = [10 20 70];
cl_info{2,6} = [2 7 8 9];
cl_info{2,7} = '2A';

cl_info{3,1} = 'Vlad';
cl_info{3,2} = 'ENVE';
cl_info{3,3} = 2046426;
cl_info{3,4} = [50 90];
cl_info{3,5} = [90 60 80];
cl_info{3,6} = [1 2 6 2];
cl_info{3,7} = '2A';
```

⚠: We cannot store multiple character vectors using basic array type when their lengths are different.

Example: How to Create and Access Cell Array (Continue)

Name	Program	ID	Exam	Quiz	Homework	Cohort
Chul Min	CIVE	1076123	[80 90]	[70 30 50]	[4 5 2 1]	4B
Noreen	ENVE	3026327	[100 70]	[10 20 70]	[2 7 8 9]	2A
Vlad	ENVE	2046426	[50 90]	[90 60 80]	[1 2 6 2]	2A

Q1. What's Chul Min's ID? Assign to '`test_id`'.

```
[nr, nc] = size(cl_info);
for ii=1:nr
   if strcmp(cl_info{ii, 1}, 'Chul Min')
      test_id = cl_info{ii,3};
      break;
   end
end
```

☺: You can shorten the following script

```
test_exam = cl_info{ii,4};
test_grade = test_exam(1);

% shortened
test_grade = cl_info{ii,4}(1);
```

Q2. What's Noreen's the first exam grade? Assign to '`test_grade`'.

```
[nr, nc] = size(cl_info);
for ii=1:nr
   if strcmp(cl_info{ii, 1}, 'Noreen')
      test_exam = cl_info{ii,4};
      test_grade = test_exam(1);
      break;
   end
end
```

What's the Difference Between {} and ()?

Challenging

```
cl_info = cell(3,7);

cl_info{1,1} = 'Chul Min';
cl_info{1,2} = 'CIVE';
cl_info{1,3} = 1076123;
cl_info{1,4} = [80 90];
cl_info{1,5} = [70 30 50];
cl_info{1,6} = [4 5 1 2];
cl_info{1,7} = '4B';

cl_info{2,1} = 'Noreen';
cl_info{2,2} = 'ENVE';
cl_info{2,3} = 3026327;
cl_info{2,4} = [100 70];
cl_info{2,5} = [10 20 70];
cl_info{2,6} = [2 7 8 9];
cl_info{2,7} = '2A';

cl_info{3,1} = 'Vlad';
cl_info{3,2} = 'ENVE';
cl_info{3,3} = 2046426;
cl_info{3,4} = [50 90];
cl_info{3,5} = [90 60 80];
cl_info{3,6} = [1 2 6 2];
cl_info{3,7} = '2A';
```

```
cl_info = cell(3,7);

cl_info(1,:) = {'Chul Min', 'CIVE', 1076123, [80 90], ...
[70 30 50], [4 5 1 2], '4B'};

cl_info(2,:) = {'Chul Min', 'CIVE', 1076123, [80 90], ...
[70 30 50], [4 5 1 2], '4B'};

cl_info(3,:) = {'Chul Min', 'CIVE', 1076123, [80 90], ...
[70 30 50], [4 5 1 2], '4B'};
```

```
>> class(cl_info{1,1})

ans =

    'char'
```

Read the value inside the cell

```
>> class(cl_info(1,1))

ans =

    'cell'
```

Read the cell itself

📖: `()` is to refer a set of cells. You refer cell(s) themselves and assign a cell or cell array to the space. `{}` is to refer a value in a cell. You refer a space inside the cell and assign the value to the cell. Remember that cell is a data type!

📖: **`class`** is a function of determining a class of object.

How to Create and Access Cell Array using Vectorization

Challenging

Name	Program	ID	Exam	Quiz	Homework	Cohort
Chul Min	CIVE	1076123	[80 90]	[70 30 50]	[4 5 2 1]	4B
Noreen	ENVE	3026327	[100 70]	[10 20 70]	[2 7 8 9]	2A
Vlad	ENVE	2046426	[50 90]	[90 60 80]	[1 2 6 2]	2A

Q1. What's Chul Min's ID? Assign to '`test_id`'.

```
lg_vec = strcmp(cl_info(:,1), 'Chul Min');
idx = find(lg_vec);
test_id = cl_info{idx, 3};
```

☺: We don't need to do `find(lg_vec == 1)` because `find` function is designed to find the 1 (`true`) values.

`lg_vec`

1
0
0

`idx` | 1 |

`test_id` | 1076123 |

📖: `strcmp` supports a cell array of character vectors as inputs. The output produces the logical array with the same size as the input array. Here, the input is the column vector of a cell array that contains character vectors (names). The function compares each element with the second input of the character vector.

How to Create and Access Cell Array using Vectorization Challenging

Name	Program	ID	Exam	Quiz	Homework	Cohort
Chul Min	CIVE	1076123	[80 90]	[70 30 50]	[4 5 2 1]	4B
Noreen	ENVE	3026327	[100 70]	[10 20 70]	[2 7 8 9]	2A
Vlad	ENVE	2046426	[50 90]	[90 60 80]	[1 2 6 2]	2A

Q2. What's name of the student whose ID is 3026327? Assign the name to '`test_name`'.

```
mat_id = [cl_info{:,3}];
lg_vec = mat_id == 3026327;
idx = find(lg_vec);
test_name = cl_info{idx, 1};
```

📖: This is the way of vectorizing the numeric cell array. `cl_info{:,3}` is the script to read all values in the referred cells. If it is inside `[]`, this becomes a numeric array. You can use a `cell2mat` built-in function. Once we convert the cell array to the numeric array, we can solve this problem using a vectorized script.

`mat_id`

1076123
3026327
2046426

`lg_vec`

0
1
0

`idx`

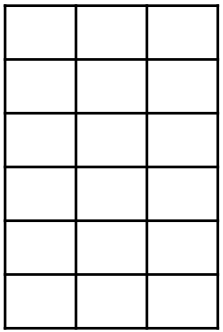
2

`test_name`

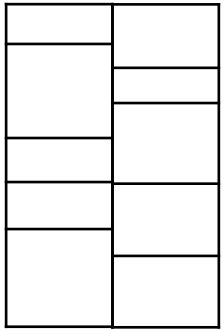
N	o	r	e	e	n

Structure Variables

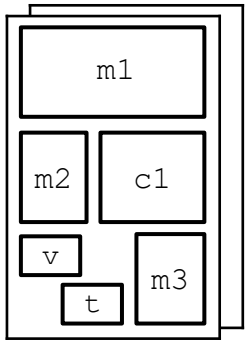
- A structure array is a data type that groups related data using data containers called field.
- Each field can contain any type of data.
- Fields are given names; they are referred to as **structurename.fieldname** using the dot operator.



Basic array (A)



Cell array (B)



Structure array (C)

```
A = [1 2 3; 4 5 6];
B = {'A', 'B', 'C'};
C.a = 'CIVE';
C.b = 'ENVE';
```

A(1)	1
B{2}	'B'
C.a	'CIVE'

Create Structure Array

```
st_info(1) = struct('name', 'Chul Min', 'program', 'CIVE', 'id', ...
    1076123, 'exam', [80 90], 'quiz', [70 30 50], 'homework', ...
    [4 5 1 2], 'cohort', '4B');
```

option 1

```
st_info(1).name = 'Chul Min';
st_info(1).program = 'CIVE';
st_info(1).id = 1076123;
st_info(1).exam = [80 90];
st_info(1).quiz = [70 30 50];
st_info(1).homework = [4 5 1 2];
st_info(1).cohort = '4B';
```

option 2

☺: You can initialize the structure array in the beginning however, there might not be a formal way to do it. If you know the size of the structure array (e.g., # of students in the above case), you can do

```
st_info(10) = struct;
```

```
>> st_info

st_info =

  struct with fields:

        name: 'Chul Min'
     program: 'CIVE'
          id: 1076123
        exam: [80 90]
        quiz: [70 30 50]
    homework: [4 5 1 2]
      cohort: '4B'
```

Example: How to Create and Access Structure Array

Suppose that you want to store the following data using a structure array:

Name	Program	ID	Exam	Quiz	Homework	Cohort
Chul Min	CIVE	1076123	[80 90]	[70 30 50]	[4 5 2 1]	4B
Noreen	ENVE	3026327	[100 70]	[10 20 70]	[2 7 8 9]	2A
Vlad	ENVE	2046426	[50 90]	[90 60 80]	[1 2 6 2]	2A

```
st_info(1).name = 'Chul Min';
st_info(1).program = 'CIVE';
st_info(1).id = 1076123;
st_info(1).exam = [80 90];
st_info(1).quiz = [70 30 50];
st_info(1).homework = [4 5 1 2];
st_info(1).cohort = '4B';

st_info(2).name = 'Noreen';
st_info(2).program = 'ENVE';
st_info(2).id = 3026327;
st_info(2).exam = [100 70];
```

```
st_info(2).exam = [100 70];
st_info(2).quiz = [10 20 70];
st_info(2).homework = [2 7 8 9];
st_info(2).cohort = '2A';

st_info(3).name = 'Vlad';
st_info(3).program = 'ENVE';
st_info(3).id = 2046426;
st_info(3).exam = [50 90];
st_info(3).quiz = [90 60 80];
st_info(3).homework = [1 2 6 2];
st_info(3).cohort = '2A';
```

Example: How to Create and Access Structure Array (Continue)

Name	Program	ID	Exam	Quiz	Homework	Cohort
Chul Min	CIVE	1076123	[80 90]	[70 30 50]	[4 5 2 1]	4B
Noreen	ENVE	3026327	[100 70]	[10 20 70]	[2 7 8 9]	2A
Vlad	ENVE	2046426	[50 90]	[90 60 80]	[1 2 6 2]	2A

Q1. What's Chul Min's ID? Assign to '`test_id`'.

```
nst = numel(st_info);
for ii=1:nst
   if strcmp(st_info(ii).name, 'Chul Min')
      test_id = st_info(ii).id;
      break;
   end
end
```

☺: You can shorten the following script

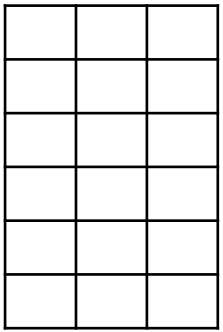
```
test_exam = st_info(ii).exam;
test_grade = test_exam(1);

% shortened
test_grade = st_info(ii).exam(1);
```

Q2. What's Noreen's the first exam grade? Assign to '`test_grade`'.

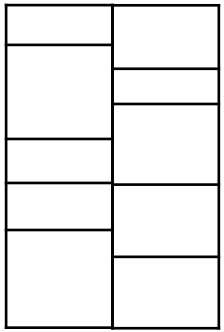
```
nst = numel(st_info);
for ii=1:nst
   if strcmp(st_info(ii).name, 'Noreen')
      test_exam = st_info(ii).exam;
      test_grade = test_exam(1);

   end
end
```

Cell Arrays vs Structures

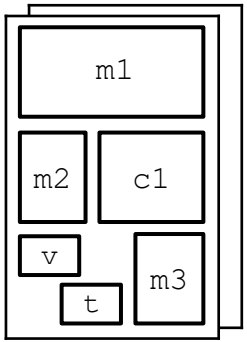
- Cell arrays are arrays, so they can be indexed. That means that you can loop though the elements in a cell array – or have MATLAB do that for you by using a vectorized code
- **Structs are not indexed**, so you cannot loop to access the value. However, the field names are mnemonic, so it is clearer what is being stored in a struct.
- For example: variable{1} vs. variable.weight: which is more mnemonic?



Basic array (A)



Cell array (B)



Structure array (C)

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

```
B = {'A', 'B', 'C'};
B{1}
```

```
B.a = 'CIVE';
B.b = 'ENVE';
B.a
```

Passing Multiple Input Variable to Function using Structure Optional

Suppose that you are making a function named 'CompGrade' to compute a student grade using his/her record during a term. The ten records are stored in each variable named 'r1', 'r2', …. 'r10'. Thus, the function accepts for 10 inputs and produce one output. How to design your function?

```
function final_grade = CompGrad(r1, r2, ..., r10)
Do something
end
```

```
function final_grade = CompGrad(record)
r1 = record.r1;
r2 = record.r2;
:
r10 = record.r10;
end
```

☺: Rather than passing all ten input variables to the function, we can pass one singe structure variable and store all ten variables in the structure variable.

script

```
...
record.r1 = r1;
record.r2 = r2;
:
record.r10 = r10;
final_grade = ...
CompGrad(record);
...
```

Module 10: File I/O

Chul Min Yeum
Associate Professor
Civil and Environmental Engineering
University of Waterloo, Canada

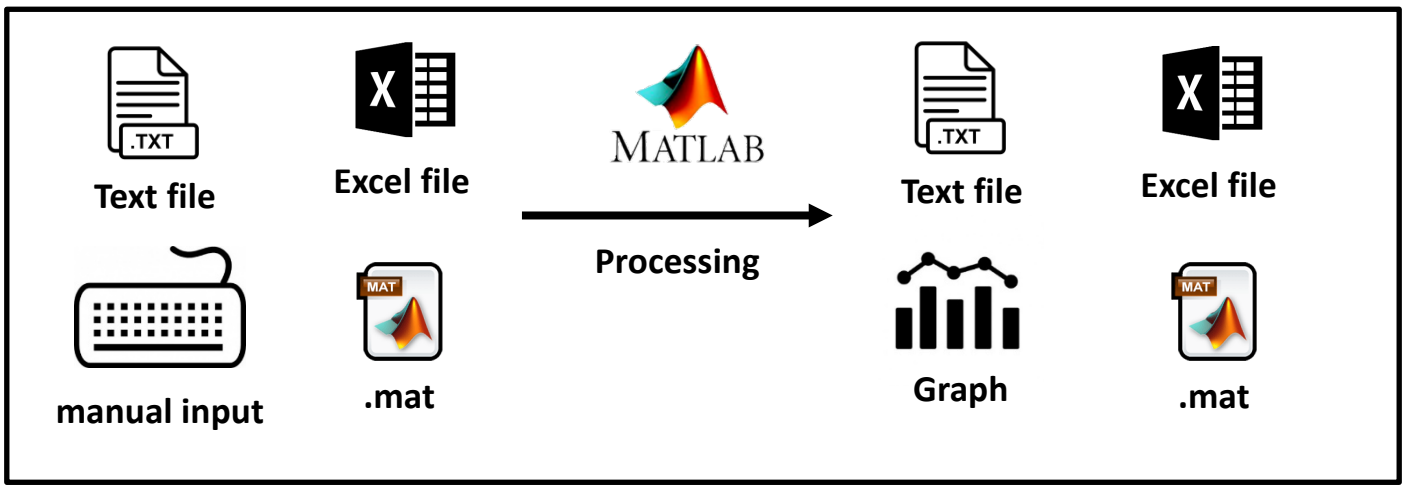
UNIVERSITY OF WATERLOO
FACULTY OF ENGINEERING

Module 10: Learning Outcomes

- Store variables in MATLAB Workspace to a MAT-file
- Import text files to MATLAB Workspace
- Explain the difference between text and numeric data when they are read from the file
- Read and Write MS Excel files

File Input & Output (I/O)

- MATLAB has functions to read from and write to many different file types, for example, spreadsheets.
- MATLAB has a special binary file type that can be used to store variables and their contents in MAT-files.



Using MAT-files for Variables

- MATLAB has functions that allow reading and saving variables from files.
- These files are called MAT-files (because the extension is `.mat`).
- Variables can be written to MAT-files, appended to them, and read from them.
- Rather than just storing data, MAT-files **store variable names and their values**

- To save all workspace variables in a file, the command is:

```
save filename
```

- To save just one variable to a file, the format is:

```
save filename variablename
```

- To read variables from a MAT-file into the base workspace:

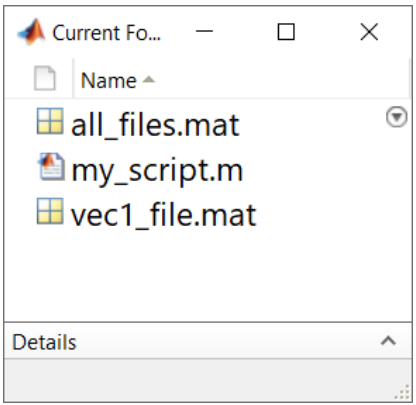
```
load filename variablelist
```

Example Save and Load Variables Using MAT files

my_script. m

```
vec1 = zeros(1, 3);
mat1 = ones(2, 2);
char1 = 'a';

save all_files
save vec1_file vec1
```

Current folder




```
load all_files
```

Name	Value
vec1	[0 0 0]
mat1	[1 1; 1 1]
char1	'a'

```
load vec1_file
```

Name	Value
vec1	[0 0 0]

```
load all_files mat1
```

Name	Value
mat1	[1 1; 1 1]

⚠: When you are working on a large-scale project, saving all files causes long saving and loading time. Selectively saving and loading is recommended.

Importing Data from a .txt File

- The text file must be saved in the current folder that you are working in on MATLAB

- Delimiter: sequence of one or more characters used to specify boundaries between separate regions (e.g., comma (,), semicolon (;), space( ))

- Three importing scenarios:
  - Importing numeric data
  - Importing character (string) data
  - Importing numeric and character data

Print Formatting Text

- The print formatting texts specify in what layout and what data type a column of data will be imported/exported as – for multiple columns of data, use multiple print format operators

- Common print formatting texts:
  - `%d` –For integer numbers
  - `%f` –For floating point (decimal) numbers
  - `%s` –For entire character vectors or strings

- Note that you can only use 1 format text per column (you cannot specify `'%s'` for the first value in a column and '`%f`' for all other values

- Example: If you wanted your first column of data to be integer data, the second column to be stored as character data and the third to be floating point numbers, the correct format text would be: `'%d %s %f'`

textscan Function

- The `textscan` function will create a **cell** array from the data read on the `.txt` file

```
C = textscan(fileID,formatSpec)
```

- This function can be used to import **numeric data, text data, and both types of data** from the same file easily
- When importing using `textscan`, the data is imported column-by-column.
- You can specify using print formatting texts what type of data you want each column to be.
- Data will be stored in a cell, **where each column is a different cell array element**
- You have the option of specifying a delimiter. The default delimiter is white-space.

fopen and fclose function

- Used to open a file for a specific purpose
- You can specify your purpose using a permission specifier(the second input)
- The file does not have to already exist (although for reading data it should)

- Read only access (default):

```
fid = fopen('sample_data.txt', 'r')
```

- Write only access (discard original contents of a file):

```
fid = fopen('sample_data.txt', 'w')
```

- Read and write access (discard original contents of file if writing):

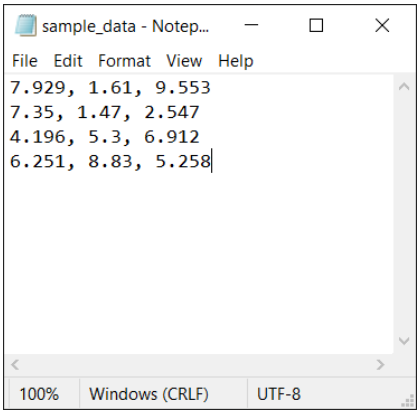
```
fid = fopen('sample_data.txt', 'w+')
```

- Once you finish read/write operation on the file, you need to close the file

```
fclose(fid)
```

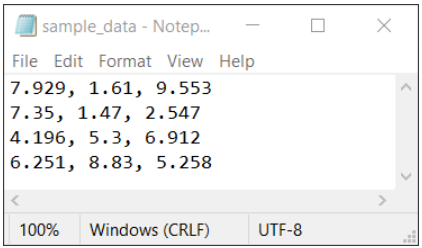
General Procedure for Importing Data using textscan

```
1  fid = fopen('sample_data.txt');
2
3  test_data = textscan(fid, '%f %f %f', 'delimiter', ',');
4
5  fclose(fid);
```



- Line 1: Open the file to be read by creating a 'fid'. This is creating a numeric ID that represents the file.
- Line 3: Use the textscan function to import the data using the proper format string and delimiter. If the delimiter is whitespace, you do not need to specify a delimiter.
- Line 5: Close the file you have just read.

Example: Read Numeric Data

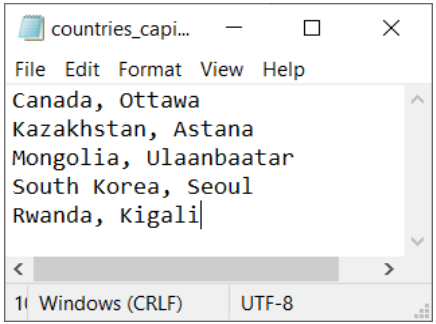


Three columns of numeric data – when importing, you will need to specify three print formatting texts.

```
fid = fopen('sample_data.txt');

samp_data = textscan(fid, '%f %f %f', 'delimiter', ',');

fclose(fid);
```

```
>> samp_data

samp_data =

  1×3 cell array

    {4×1 double}    {4×1 double}    {4×1 double}

>> samp_data{1}

ans =

    7.9290
    7.3500
    4.1960
    6.2510
```

Example: Read Text Data



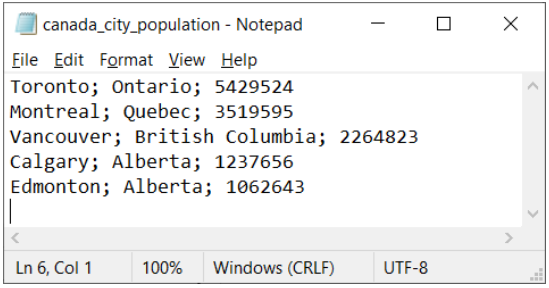
⚠: When you read or write text information in a text file, you should not use the white-space delimiter because texts likely include the white space.

```
fid = fopen('countries_capitals.txt');

text_data = textscan(fid, '%s %s', 'delimiter', ',');

fclose(fid);
```

```
>> text_data

text_data =

  1×2 cell array

    {5×1 cell}    {5×1 cell}

>> text_data{1}

ans =

  5×1 cell array

    {'Canada'     }
    {'Kazakhstan' }
    {'Mongolia'   }
    {'South Korea'}
    {'Rwanda'     }
```

Example: Read Numeric and Text Data

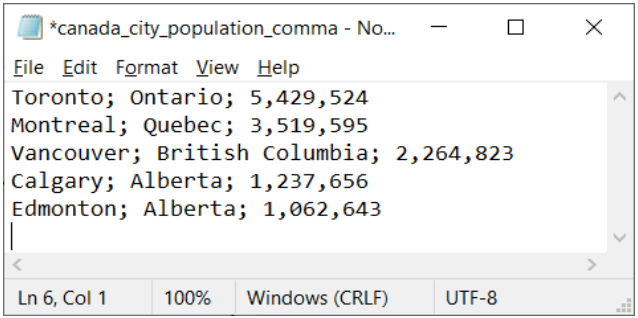
```
fid = fopen('canada_city_population.txt');

city_data = textscan(fid, '%s %s %f', 'delimiter', ';');

fclose(fid);
```




```
>> city_data

city_data =

  1×3 cell array

    {5×1 cell}    {5×1 cell}    {5×1 double}

>> city_data{3}

ans =

     5429524
     3519595
     2264823
     1237656
     1062643
```

Example: Read Number (with Comma) and Text Data

Optional




```
fid = fopen( ...
'canada_city_population_comma.txt');

text_data = textscan(fid, ...
'%s %s %f',  'delimiter', ';');

fclose(fid);
```

Not working

⚠: 5,429, 524 is not a numeric type because of commas. Thus, the corresponding column can't be read using '%f'.

```
fid = ...
fopen('canada_city_population_comma.txt');

text_data = textscan(fid, '%s %s %s',...
'delimiter', ';');
fclose(fid);

pop_data_cell = text_data{3};

n_data = numel(pop_data_cell);
pop_data = zeros(n_data,1);
for ii=1:n_data
   text_text = pop_data_cell{ii};
   num_text = text_text(text_text ~= ',');
   pop_data(ii) = str2double(num_text);
end
```

```
>> pop_data

pop_data =

     5429524
     3519595
     2264823
     1237656
     1062643
```

Read and Write Excel Data

- The `xlsread` function can be used for importing MS Excel data into MATLAB. However, the use of `xlsread` function is not recommend starting in R2019a.
- MATLAB introduce a **new data type** called **`table`**. It is a suitable for column-oriented or tabular data that is often stored as columns in a text file or a spreadsheet. However, we will not cover this new data type in this course.
- Instead, we will import the data as a cell array using `readcell` or `readmatrix`

readmatrix and readcell Function

- `readmatrix` creates an array by reading column-oriented data from a file.

```
M = readmatrix(filename)
M = readmatrix(filename,'Sheet',sheet,'Range',range)
```

- `readcell` creates a cell array by reading column-oriented data from a file.

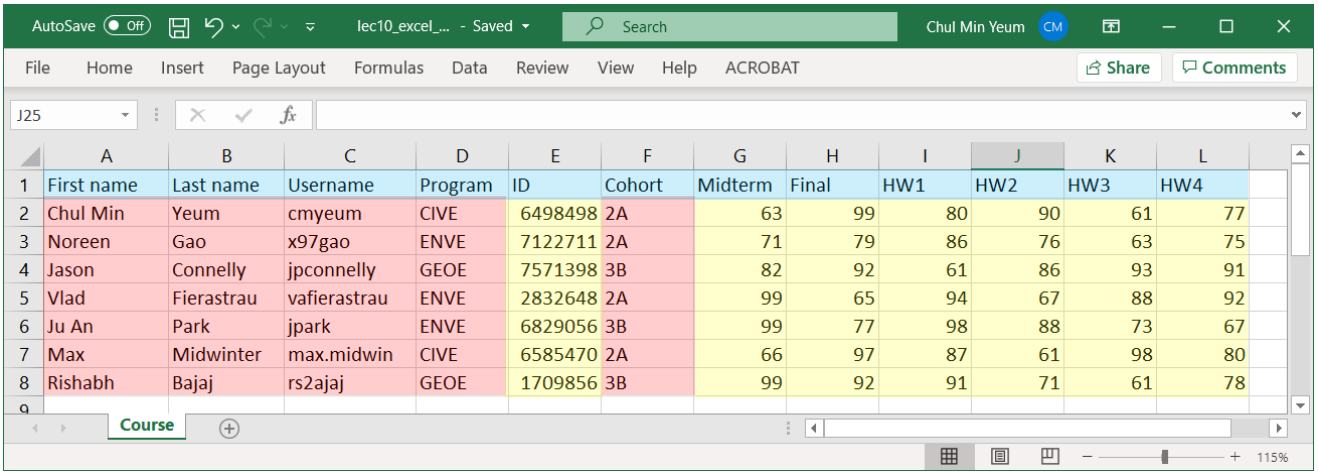
```
C = readcell(filename)
C = readcell(filename,'Sheet',sheet,'Range',range)
```

☺: `readmatrix` or `readcell` determines the file format from the file extension:

- .txt, .dat, or .csv for delimited text files
- .xls, .xlsb, .xlsm, .xlsx, .xltm, .xltx, or .ods for spreadsheet files

Example: Read Data from an Excel File (File)

lec10_excel_file.xlsx



	A	B	C	D	E	F	G	H	I	J	K	L
1	First name	Last name	Username	Program	ID	Cohort	Midterm	Final	HW1	HW2	HW3	HW4
2	Chul Min	Yeum	cmyeum	CIVE	6498498	2A	63	99	80	90	61	77
3	Noreen	Gao	x97gao	ENVE	7122711	2A	71	79	86	76	63	75
4	Jason	Connelly	jpconnelly	GEOE	7571398	3B	82	92	61	86	93	91
5	Vlad	Fierastrau	vafierastrau	ENVE	2832648	2A	99	65	94	67	88	92
6	Ju An	Park	jpark	ENVE	6829056	3B	99	77	98	88	73	67
7	Max	Midwinter	max.midwin	CIVE	6585470	2A	66	97	87	61	98	80
8	Rishabh	Bajaj	rs2ajaj	GEOE	1709856	3B	99	92	91	71	61	78

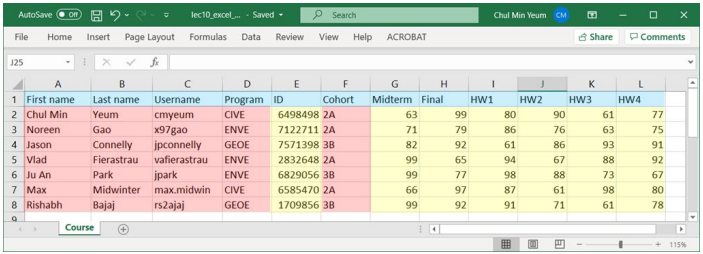
Header

Text values

Numeric values

⚠: We need to use a cell array to contain all these data in one variable.

Example: Read Data from an Excel File (readmatrix)



	A	B	C	D	E	F	G	H	I	J	K	L
1	First name	Last name	Username	Program	ID	Cohort	Midterm	Final	HW1	HW2	HW3	HW4
2	Chul Min	Yeum	cmyeum	CIVE	6498498	2A	63	99	80	90	61	77
3	Noreen	Gao	x97gao	ENVE	7122711	2A	71	79	86	76	63	75
4	Jason	Connelly	jpconnelly	GEOE	7571398	3B	82	92	61	86	93	91
5	Vlad	Fierastrau	vafierastrau	ENVE	2832648	2A	99	65	94	67	88	92
6	Ju An	Park	jpark	ENVE	6829056	3B	99	77	98	88	73	67
7	Max	Midwinter	max.midwin	CIVE	6585470	2A	66	97	87	61	98	80
8	Rishabh	Bajaj	rs2ajaj	GEOE	1709856	3B	99	92	91	71	61	78

```
filename = 'lec10_excel_file.xlsx';
M = readmatrix(filename);
```

```
K>> M

M =

       NaN       NaN       NaN       NaN   6498498
       NaN       NaN       NaN       NaN   7122711
       NaN       NaN       NaN       NaN   7571398
```

```
filename = 'lec10_excel_file.xlsx';
M = readmatrix(filename);

M_num = M;
lg_mat = isnan(M_num);
idx = logical(sum(lg_mat));
M_num(:,idx) = [];
```

option1

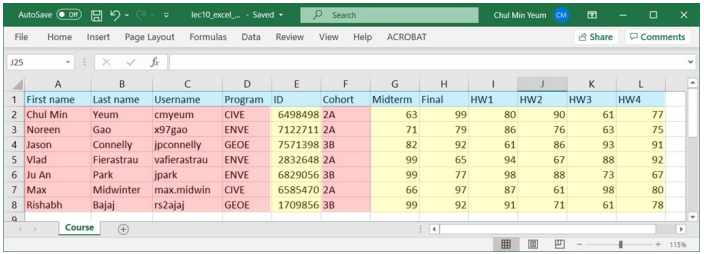
📖: `isnan` returns a logical array containing `1` where the elements are `NaN`. Here `NaN` means it is not a numeric (number) value. We extract numeric array using a vectorized code.

```
filename = 'lec10_excel_file.xlsx';
M = readmatrix(filename);
```

option2

```
M1 = readmatrix(filename, 'Range', ...
'E2:E8'); % set range
M2 = readmatrix(filename, 'Range', ...
'G2:L8'); % set range

M_num = [M1 M2]; % joining cell arrays
```

Example: Read Data from an Excel File (readcell)



	A	B	C	D	E	F	G	H	I	J	K	L
1	First name	Last name	Username	Program	ID	Cohort	Midterm	Final	HW1	HW2	HW3	HW4
2	Chul Min	Yeum	cmyeum	CIVE	6498498	2A	63	99	80	90	61	77
3	Noreen	Gao	x97gao	ENVE	7122711	2A	71	79	86	76	63	75
4	Jason	Connelly	jpconnelly	GEOE	7571398	3B	82	92	61	86	93	91
5	Vlad	Fierastrau	vafierastrau	ENVE	2832648	2A	99	65	94	67	88	92
6	Ju An	Park	jpark	ENVE	6829056	3B	99	77	98	88	73	67
7	Max	Midwinter	max.midwin	CIVE	6585470	2A	66	97	87	61	98	80
8	Rishabh	Bajaj	rs2ajaj	GEOE	1709856	3B	99	92	91	71	61	78

```
filename = 'lec10_excel_file.xlsx';
M = readcell(filename);
```

```
>> M

M =

  8×12 cell array

    {'First name'}    {'Last name' }    {'Username'    }    {'Program'}    {'ID'      }
    {'Chul Min'  }    {'Yeum'      }    {'cmyeum'      }    {'CIVE'   }    {[6498498]}
    {'Noreen'    }    {'Gao'       }    {'x97gao'      }    {'ENVE'   }    {[7122711]}
    {'Jason'     }    {'Connelly'  }    {'jpconnelly'  }    {'GEOE'   }    {[7571398]}
```

```
filename = 'lec10_excel_file.xlsx';

M1 = readcell(filename, 'Range', ...
'E2:E8');
M2 = readcell(filename, 'Range', ...
'G2:L8');

M_num_cell = [M1 M2];
M_num = cell2mat(M_num_cell);
```

📖: All data will be read using cell types. We can access both text and numeric data. However, when you process only numeric data, and spreadsheet contains large volume of text data, the size of M will get larger causing taking up large memory and slow processing speed. Thus, in such case, I recommend using `reamatrix` rather than `readcell`.

writematrix and writecell Function

- `writematrix` writes numeric array `A` to a file with the name and extension specified by `filename`

```
writematrix(A, filename)
writematrix(A, filename,'Sheet',sheet,'Range',range)
```

- `writecell` write cell array `C` to a file with the name and extension specified by `filename`

```
writecell(C, filename)
writecell(C, filename,'Sheet',sheet,'Range',range)
```

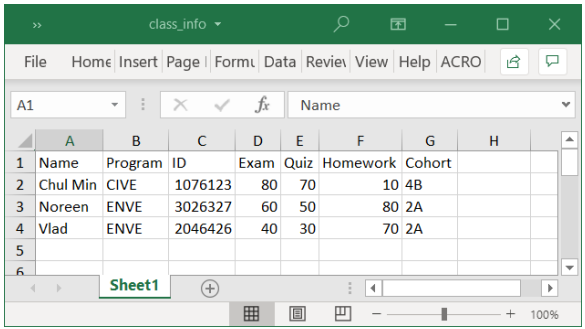
☺: `writematrix` or `writecell` determines the file format based on the specified extension:

- .txt, .dat, or .csv for delimited text files
- .xls, .xlsb, .xlsm, .xlsx, .xltm, .xltx, or .ods for spreadsheet files

Example: Write Data to an Excel File (writecell)

```
cl_info = cell(3,7);

cl_info{1,1} = 'Chul Min';
cl_info{1,2} = 'CIVE';
cl_info{1,3} = 1076123;
cl_info{1,4} = [80 90];
cl_info{1,5} = [70 30 50];
cl_info{1,6} = [4 5 1 2];
cl_info{1,7} = '4B';

cl_info{2,1} = 'Noreen';
cl_info{2,2} = 'ENVE';
cl_info{2,3} = 3026327;
cl_info{2,4} = [100 70];
cl_info{2,5} = [10 20 70];
cl_info{2,6} = [2 7 8 9];
cl_info{2,7} = '2A';

cl_info{3,1} = 'Vlad';
cl_info{3,2} = 'ENVE';
cl_info{3,3} = 2046426;
cl_info{3,4} = [50 90];
cl_info{3,5} = [90 60 80];
cl_info{3,6} = [1 2 6 2];
cl_info{3,7} = '2A';
```

```
cl_header = {'Name', 'Program', 'ID', 'Exam', 'Quiz',...
'Homework', 'Cohort'};

cl_all = cat(1, cl_header, cl_info);
% cl_all = [cl_header; cl_info];

writecell(cl_all, 'cl_info.xlsx');
```



class_info.xlsx

📖 : In the original cell, there is no header. Thus, before you store the cell array data, you need to append the header to the cell data.

Tip for Quick Testing

Optional

Sometimes, you want to do quick testing on a part of data in an excel file, but you do not want to write a script to read/extract the values from the file.

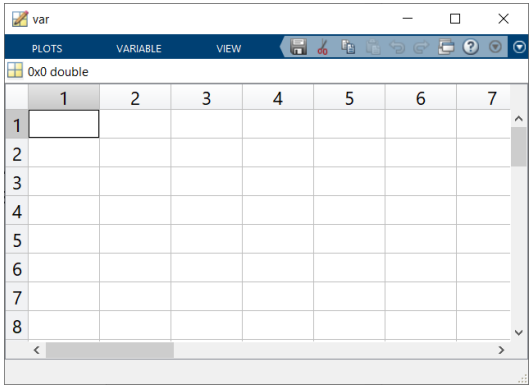
Step 1. Make an empty variable

Step 2. Double click the variable in Workspace to edit the values of the variable.

Step 3. Copy the data from the excel file or text file and paste them in the values of the variable.

Step 4. Save the variable in .mat file.

Step 5. Use the variable for your computation



☺: You can copy excel data and paste them in the variable through this variable window. You can open it by double-clicking the variable name in Workspace. The format of the variable window is similar to the ones in Excel.

Module 11: Text Manipulation

Chul Min Yeum
Associate Professor
Civil and Environmental Engineering
University of Waterloo, Canada

UNIVERSITY OF WATERLOO
FACULTY OF ENGINEERING

Module 11: Learning Outcomes

- Distinguish the difference between a character vector and a string scalar
- Distinguish the difference between a string scalar and a string array.
- Explain how to read and extract individual characters in a string
- Apply built-in functions specially designed for operations using string or character vector
- Explain how to use operators (==, +) when string variables are involved.
- Convert strings or character vectors to numbers and vice versa.

Text Terminology

- Text in MATLAB can be represented using:
  - character vectors (in single quotes)
  - string arrays, introduced in R2016b (in double quotes)
- Prior to R2016b, the word “string” was used for what are now called character vectors.
- Many functions that manipulate text can be called using **either a character vector or a string.**
- Additionally, several new functions have been created for the string type

Review: Character vs Character Vector

- **Characters** include letters of the alphabet, digits, punctuation marks, white space, and control characters. Individual characters in **single quotation** marks are the type char (so, the class is char)
- Character vectors consist of any number of characters and contained in single quotation marks. Their type are **char.**
- Since these are **vectors** of characters, **built-in functions and operators that we've seen already work with character vectors as well as numbers**. You can also index into a character vector to get individual characters or to get subsets

```
>> letter = 'x';
>> size(letter)

ans =

     1     1

>> class(letter)

ans =

    'char'
```

```
>> myword = 'Hello';
>> size(myword)

ans =

     1     5

>> class(myword)

ans =

    'char'
```

String Scalar

- A string scalar is a data type of '**string**', a single string, and is used to store a group of characters.
- The group of characters defined as a string is a single scalar variable, not a vector.
- String can be created
  - using double quotes, e.g., "`Chul Min`"
  - using the string type cast function, e.g., `string('Chul Min')`
- Strings are displayed using double quotes.

```
>> myword = 'Hello';
>> size(myword)

ans =

     1     5

>> class(myword)

ans =

    'char'
```

```
>> myword = "Hello";
>> size(myword)

ans =

     1     1

>> class(myword)

ans =

    'string'
```

⚠: A character vector is an array of characters. A string is a single data type (dimension) that can contain a string of characters. It is not an array of characters!

String Indexing

Challenging

- You can index into a string array using curly braces `{}`, to access characters directly.
- Indexing with curly braces provides compatibility for code that could work with either string arrays or cell arrays of character vectors.

```
mystr = "CM121";

mycharv = 'CM121';

mystrArry = ["ENVE121",...
"GEOE121", "AE121"];
```

📖: `strlength` is to count the number of characters in a string value.

```
size(mystr)              [1  1]
mystr(1)                 "CM121"
mystr{1}                 'CM121'
mystr{1}(1)              'C'
mystr{1}(4)              '2'

size(mycharv)            [1  5]
mycharv(1)               'C'
mycharv(4)               '2'

size(mystrarry)          [1  3]
mystrArry(2)             "GEOE121"
mystrArry{1}             'ENVE121'
mystrArry{2}(1)          'G'
mystrArry{2}(5)          '1'

numel(mystr)             1
numel(char(mystr))       5
strlength(mystr)         5
```

Group of Strings

Challenging

- Groups of strings can be stored in (1) **string arrays**, (2) **character matrices**, (3) **cell arrays**.
- Strings are created using double quotes or by passing a character vector to the string function
- The plus function or operator can **join**, or **concatenate**, two strings together (e.g., `"abc" + "xyz" => "abcxyz"`)

`[char1 char2]`	`'ENVE121'`
`char1 + char2`	`error`
`char1 + char1`	`[138 156 172 138]`
`[str1 str2]`	`"AE"  "Hello"`
`str1 + str2` `strcat(str1, str2)`	`"AEHello"` `"AEHello"`
`str1 + string(char1)`	`"AEENVE"`
`[char(str1) char(str2)]`	`'AEHello'`

```
char1 = 'ENVE';
char2 = '121';

str1 = "AE";
str2 = "Hello";
```

☺: You can initialize a string array using `strings`.

```
str = strings(sz1,...,szN)

str_arry = strings(1, 2);
str_arry(1,1) = "AE"
str_arry(1,2) = "Hello"
```

Example: Print Texts

Q. Print out the following texts in the command window when variables containing the first and last names are given.

```
Welcome ! Park, Ju An
Welcome ! Yeum, Chul Min
Welcome ! Gao, Noreen
Welcome ! Fierastrau, Vlad
Welcome ! Connelly, Jason
```

```
num_name = 5;
fname = ["Ju An", "Chul Min", "Noreen", "Vlad", "Jason"];
lname = ["Park", "Yeum", "Gao", "Fierastrau", "Connelly"];
```

```
for ii=1:num_name
    fprintf('Welcome ! %s, %s \n', lname(ii), fname(ii));
end
```

option1

```
for ii=1:num_name
    str = lname(ii) + ", " + fname(ii);
    fprintf('Welcome ! %s \n', str);
end
```

option2

⚠: '… %s,%s \n'
Here, the comma is not to separate arguments. It is in the text.

📖 : %s is to print both strings and characters.

Built-in Functions for Text

Many functions can have either character vectors or strings as input arguments. Generally, **if a character vector is passed to the function, a character vector will be returned, and if a string is passed, a string will be returned**

☺: Again, you do not have to memorize the functions and their usage. You can simply search for their usage in google or type `doc fun_name` in command window.

Function	Description
`upper`	Convert strings to uppercase
`lower`	Convert strings to lowercase
`reverse`	Reverse order of characters in strings
`strcmp`	Compare strings
`strcmpi`	Compare strings (case insensitive)
`strncomp`	Compare first n characters of strings (case sensitive)
`strncmpi`	Compare first n characters of strings (case insensitive)
`contains`	Determine if pattern is in strings
`strfind`	Find strings within other strings
`strrep`	Find and replace substrings
`count`	Count occurrences of pattern in strings

Example: upper Built-in Function

Q. Write a custom function named '`MyUpper`' to replicate the same operation of `upper`. The input and output types are assumed to be string.

```
function newStr = MyUpper(str)

char_db = double(char(str));

lgv = 97 <= char_db & char_db <= 122;

char_db(lgv) = char_db(lgv)-32;

newStr = string(char(char_db));

end
```

```
>> str = "abc!ABC?";
>> upper(str)

ans =

    "ABC!ABC?"

>> MyUpper(str)

ans =

    "ABC!ABC?"
```

📖 : Here is the syntax for `upper` built-in function: `newStr = upper(str)` converts all lowercase characters in `str` to the corresponding uppercase characters and leaves all other characters unchanged.

⚠: string type has no equivalent numeric values. Thus, we cannot do the type casting of strings to numeric values using `double()`.

Example: strfind and count Built-in Function

Q. Write a script to count the number of '`word`' in '`char_vec`'

option1

```
char_vec = 'abcardcsecar';
word = 'car';

vec_len = numel(char_vec);
n_char = numel(word);

n_word = 0;
for ii=1:vec_len-n_char+1
    test_loc = ii:ii+n_char-1;
    if isequal(char_vec(test_loc), word)
       n_word = n_word + 1;
    end
end
```

option2

```
char_vec = 'abcardcsecar';
word = 'car';

idx = strfind(char_vec, word);
n_word = numel(idx);
```

option3

```
char_vec = 'abcardcsecar';
word = 'car';

n_word = count(char_vec, word);
```

📖: `k = strfind(str,pattern)` searches str for occurrences of pattern. The output, `k`, indicates the starting index of each occurrence of pattern in `str`.

`A = count(str,pattern)` returns the number of occurrences of `pattern` in `str`.

📖: Again, the function support both **strings** and **character** vectors as the input for `str`.

Example: strfind Built-in Function (Word Finder Puzzle)

Q. Rewrite a script for a word finder puzzle using `strfind`.

```
word_loc = zeros(n_word, 2);
for ii=1:puzzle_size
    col_vec = puzzle(:,ii); % column
    row_vec = puzzle(ii,:); % row

    for jj=1:(puzzle_size-n_word+1)
        test_loc = jj:(jj+n_word-1);

        test_word_col = col_vec(test_loc);
        test_word_row = row_vec(test_loc);

        if isequal(test_word_col, word_db')
            word_loc(:,2) = ii;
            word_loc(:,1) = test_loc';
        elseif isequal(test_word_row, word_db)
            word_loc(:,1) = ii;
            word_loc(:,2) = test_loc';
        end
    end
end
```

```
for ii=1:puzzle_size
    col_vec = char(puzzle(:,ii)); % column
    row_vec = char(puzzle(ii,:)); % row

    id_col = strfind(col_vec', char(word_db));
    id_row = strfind(row_vec, char(word_db));

    if ~isempty(id_col)
        word_loc(:,2) = ii;
        word_loc(:,1) = (id_col:id_col+n_word-1)';
    elseif ~isempty(id_row)
        word_loc(:,1) = ii; % row
        word_loc(:,2) = (id_row:id_row+n_word-1)';
    end
end
```

📖: `k = strfind(str,pattern)` searches `str` for occurrences of `pattern`. The output, `k`, indicates the starting index of each occurrence of `pattern` in `str`.

Comparing Strings: Equality Operator (String)

To use the equality operator == with character vectors, they must be the same length, and each element will be compared. The output is the same size as the input vector size. However, for strings, it will simply return 1 or 0 instead of comparing each element.

```
char1 = 'ENVE';
char2 = 'GEOE';
char3 = 'ENVE';
char4 = 'AE';
char5 = 'G';

str1 = "AE";
str2 = "Hello";
str3 = "AE";
str4 = "A";
```

`char1 == char2`	`[0 0 0 1]`
`char1 == char3`	`[1 1 1 1]`
`char1 == char4`	`error`
`char2 == char5`	`[1 0 0 0]`
`char2 ~= char5`	`[0 1 1 1]`
`str1 == str2`	`0`
`str1 == str3`	`1`
`str1 == str4`	`0`
`str1 ~= str4`	`1`
`[str1 str2] == str3`	`[1 0]`
`[str1 str2] == str4`	`[0 0]`
`string(char1) == ... string(char3)`	`1`

⚠: You need to clearly understand the usage of the equality operator when character or strings are involved.

Create Strings using sprintf

- `sprintf` creates text so it can be used to customize the format of text.

```
str = sprintf(formatSpec,A1,...,An)
```

- Formats the data in arrays `A1,...,An` using the formatting texts specified by `formatSpec` and returns the resulting text.
- In `formatSpec`, if you use **a single quote**, the output becomes a **character vector**. If it has **double quote**, the output becomes a **string**.

⚠: `sprintf` and `fprintf` have the same syntax for the input. The difference is `sprintf` creates a string or character vector whereas `fprintf` prints it in a command window or file.

```
>> str1 = "AEG";
>> str2 = "121";

>> "Welcome! " + str1 + str2

ans =

    "Welcome! AEG121"

>> sprintf("Welcome! %s%s", str1, str2)

ans =

    "Welcome! AEG121"

>> sprintf('Welcome! %s%s', str1, str2)

ans =

    'Welcome! AEG121'
```

Example: Generate a Card Sequence String

The standard 52-card deck has 13 numbers and 4 different suits. The suit order is 'Clubs (♣)', 'Diamonds (♦)', 'Hearts (♥)', and 'Spades (♠)'. Each integer from 1 to 52 will represent the value and suit of a card, where from 1 to 52, and the value and suit of the cards will proceed in the following order:

Num	Card
1	1C
2	1D
3	1H
4	1S
5	2C
6	2D
:	:
50	13D
51	13H
52	13S

```
function str_cards = StrCard(cards)

suits = 'CDHS';
str_cards = "";
for ii = 1:numel(cards)
    card_num = ceil(cards(ii)/4);
    card_rem = rem(cards(ii),4);

    if card_rem==0
        card_rem=4;
    end

    card_suit = suits(card_rem);

    str_cards = str_cards +  ...
        sprintf("%d%s ", card_num, card_suit);
end

str_cards{1}(end) = [];

end
```

Q. Create a function named '`StrCard`' that generate a card sequence string. The input named '`cards`' is a 1 x 7 numeric vector and include the integer card number. The output named '`str_cards`' is a string scalar and include all name of the cards separated by a space. The text for suits is '`C`', '`D`', '`H`' and '`S`', corresponding to 'Clubs', 'Diamonds', 'Hearts', and 'Spades'.

```
>> cards = [1 2 3 4 5 6 7];
>> str_cards = StrCard(cards);
>> str_cards

str_cards =

    "1C 1D 1H 1S 2C 2D 2H"
```

String/Number Functions

Challenging

Converting from strings or character vectors to numbers and vice versa:

- `num2str` converts a real number to a character vector containing the number
- `string` converts number(s) to strings
- `str2double` converts from a string or character vector containing number(s) to a number array

```
num1 = 75;
num2 = 12345;

char1 = 'abcd'
char2 = '678910'
```

`char(num1)`	`'K'`
`char(num2)`	`'卄'`
`char(num1)`	`'K'`
`num2str(num1)`	`'75'`
`string(num1)`	`"75"`
`char(num2)`	`'卄'`
`num2str(num2)`	`'12345'`
`string(num2)`	`"12345"`
`double(char1)`	`[97 98 99 100]`
`str2double(char1)`	`NaN`
`double(char2)`	`[54 55 56 57 49 48]`
`str2double(char2)`	`678910`

⚠: This is very confusing!! You should not be confused with **type casting** and number conversion from **string or character vectors**.

Example: How Many Digit in Your Number?

Q. Create a function named '`CountNum`' to count the appearance of the given digit in a single numeric number. The digit should be within 0 and 9. For example:

`CountNum(454214, 4)`	3
`CountNum(454221, 2)`	2

```
function countd = CountNum(num, digit)

chard_num = num2str(num);
chard_digit = num2str(digit);
lg_vec = chard_num == chard_digit;

countd = sum(lg_vec);
end
```

option1

```
function countd = CountNum(num, digit)

countd = count(num2str(num), num2str(digit));
% countd = count(string(num), string(digit));

end
```

option2

```
CountNum(454214, 4)
```

Name	Value
num	454214
digit	4
card_num	'454214'
card_digit	'4'
lg_vec	[1 0 1 0 0 1]
countd	3