Module 06: Operators

Chul Min Yeum
Assistant 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 opportunities 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
```

s1

10

s2

2

m1

1	2
3	4

m2

5	6
7	8

v1

1	2	3	4

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
```

m1

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

m2

1	**0**
0	**1**

v1

1	**2**

v2

1
0

Operator
+
-
*****
/
^

Scalar ★ Scalar

Vector ★ Scalar

Matrix ★ Scalar

Vector ★ Vector

Vector ★ Matrix

Matrix ★ Matrix

★: Operator

`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
```

$$\underset{\texttt{m1}}{\begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix}} \quad \underset{\texttt{m2}}{\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}} \quad \underset{\texttt{v1}}{\begin{bmatrix}1 & 2\end{bmatrix}} \quad \underset{\texttt{v2}}{\begin{bmatrix}2\\ 1\end{bmatrix}}$$

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
```

s1	1
s2	2

v1	1	2	3	4
v2	1	0	1	4

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]`

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
```

s1 $\boxed{1}$

s2 $\boxed{2}$

1	2
3	4

m1

1	0
5	4

m2

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

`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; s0 = 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`
`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
    test_num = vec(ii);
    if test_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

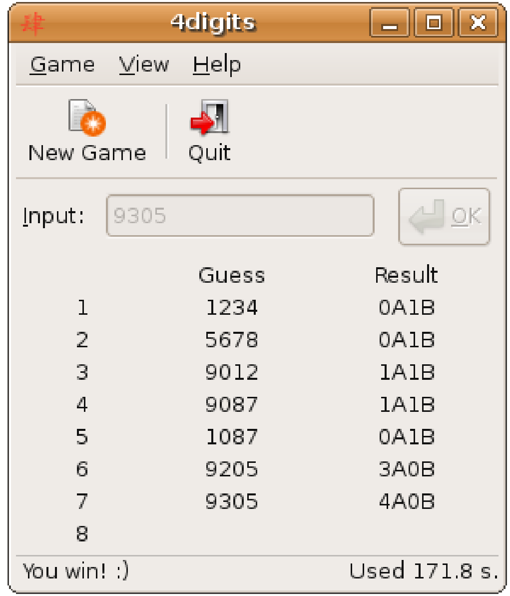
0	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**0**

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.



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)

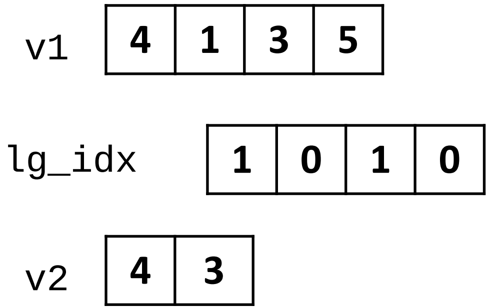
Challenging

- Can use this to index into a vector or matrix (only if 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);
```

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							

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

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);
```

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					

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

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);

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

: 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	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

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.

1	2	3
4	5	6
7	8	9

A

8	2	3
4	**8**	6
7	8	**8**

B

1	2	3
4	5	6
7	8	9

A

1	2	**8**
4	**8**	6
8	8	9

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];
```

$$\begin{array}{|c|c|}\hline 1 & 2 \\ \hline 3 & 4 \\ \hline \end{array} \; * \; \begin{array}{|c|}\hline 2 \\ \hline \end{array} \; = \; \begin{array}{|c|c|}\hline 2 & 4 \\ \hline 6 & 8 \\ \hline \end{array}$$

$$\begin{array}{|c|c|}\hline 1 & 2 \\ \hline 3 & 4 \\ \hline \end{array} \; .* \; \begin{array}{|c|c|}\hline 2 & 2 \\ \hline 2 & 2 \\ \hline \end{array} \; = \; \begin{array}{|c|c|}\hline 2 & 4 \\ \hline 6 & 8 \\ \hline \end{array}$$

Q. Does it work?

$$\begin{array}{|c|c|}\hline 1 & 4 \\ \hline 2 & 5 \\ \hline 3 & 6 \\ \hline \end{array} \; + \; \begin{array}{|c|}\hline 1 \\ \hline 2 \\ \hline 3 \\ \hline \end{array} \; = \; ?$$

Compatible Array Sizes for Operation (Continue) Optional, Challenge

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

m1

1	2
3	4
5	6

★

v1

2
3
4

Implicit → v1

2	2
3	3
4	4

m1

1	2
3	4
5	6

★

v2

1	2

Implicit → v2

1	2
1	2
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`.

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

id1 = (vec >= 10);
id2 = (vec == 6);
id3 = (vec >3) && (vec<7);

q1 = any(id1);
q2 = any(id2);
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
st_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
st_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;
```