**Islamic University of Technology (IUT)**

Organization of Islamic Cooperation (OIC)

Department of Electrical and Electronic Engineering (EEE)

| | | |
|---|---|---|
| COURSE NO | : | EEE 4416 |
| LAB NO | : | 01 (Part – C) |
| TOPIC | : | ARRAY |

# ARRAY

Arrays (vectors and matrices) are MATLAB’s primary data structures. Understanding how to generate, modify, and combine arrays efficiently is key to leveraging MATLAB’s power for numerical computing. An array can be aptly described as a collection of items stored at contiguous memory locations.

An array is used to store values in one or more dimensions. Arrays can be vectors, matrices, or multidimensional arrays, and they can hold numbers, characters, or even structures and objects. For now, we will focus on the numeric array. In later labs, you will learn about some other array types available in MATLAB.

Say, you want to store the IDs of the students attending today’s class. You can simply store them using an array. To create an array, type the elements with a space or a comma between the elements inside square brackets.

```
ID = [302, 304, 306, 310, 312]

Id = [301 303 305 307 311]
```

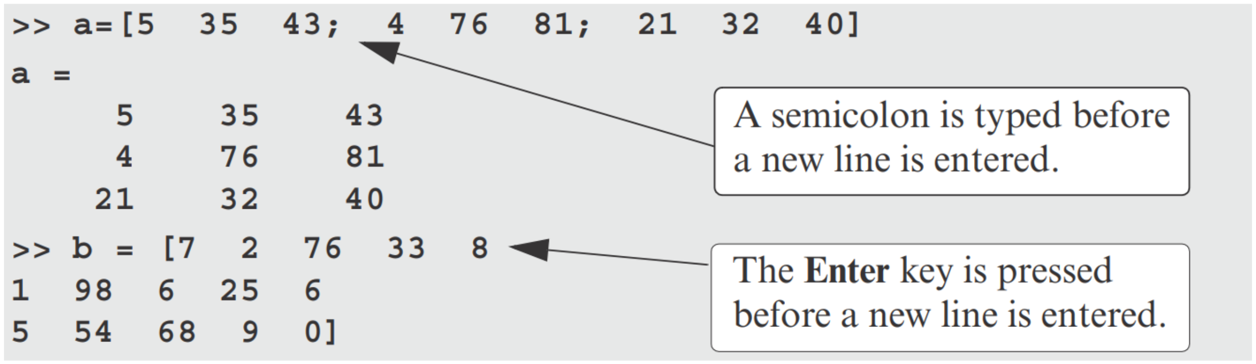
This is a one-dimensional array, also called a vector. More specifically, a **row vector**. A row vector is of shape 1×n, where n is the number of elements. A **column vector** is exactly opposite and has a shape of n×1. You can create a column vector by separating the values using a semicolon or by pressing the Enter key after each element.

```
Year = [ 2003; 2010; 2020]

Z =      [2003

          2010

          2020]
```

*Prepared by: Asif Newaz, A. K. M. Rakib*
*Lecturer, EEE, IUT*

A two-dimensional array is often called a **matrix**. It has a shape of m×n.

A = [ 2, 4; 5, 10]



```
>> a=[5  35  43;  4  76  81;  21  32  40]
a =
     5    35    43
     4    76    81
    21    32    40
>> b = [7  2  76  33  8
1  98  6  25  6
5  54  68  9  0]
```

A semicolon is typed before a new line is entered.
The **Enter** key is pressed before a new line is entered.

| Type | Description | Example |
|---|---|---|
| Row vector | 1×n array | A = [1 2 3 4] |
| Column vector | n×1 array | B = [1; 2; 3; 4] |
| Matrix | 2D array (m×n) | C = [1 2; 3 4] |
| Multidimensional | 3D or higher-dimensional arrays | D = rand(3, 4, 2) |
| Empty array | No elements | E = [ ] |
| Cell array | Holds different types of data | F = {1, 'text', [2 3]} |
| Logical array | Holds true or false values | G = logical([1 0 1]) |

There are many ways to create an array:

i. Directly assigning values
ii. Creating a range using the colon operator
iii. Creating fixed-value or random-value matrices using built-in functions
iv. Using linspace, logspace functions

## Colon Operator

**Creating a vector with constant spacing by specifying the first term, the spacing, and the last term**

In a vector with constant spacing, the difference between the elements is the same. For example, in the vector v = 2 4 6 8 10, the spacing between the elements is 2. A vector in which the first term is m, the spacing is q, and the last term is n is created by typing:

array_1 = [m:q:n]

array_2 = m:q:n

**Syntax:** start : step : end

- Automatically chooses step = +1 if omitted.
- If the numbers m, q, and n are such that the value of n cannot be obtained by adding q's to m, then (for positive n) the last element in the vector will be the last number that does not exceed n.

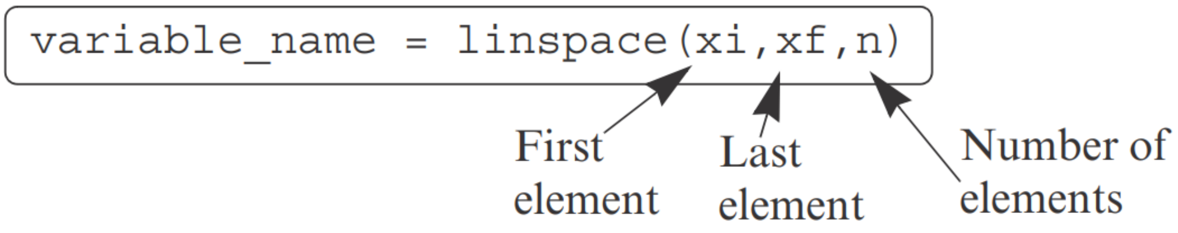
**Examples:**

```
v1 = 1:5;            % [1 2 3 4 5]

v2 = 0:0.5:2;        % [0 0.5 1.0 1.5 2.0]

v3 = 10:-2:2;        % [10 8 6 4 2]
```

## Linspace Function

**Creating a vector with linear (equal) spacing by specifying the first and last terms, and the number of terms:**

A vector with n elements that are linearly (equally) spaced, in which the first element is xi and the last element is xf, can be created by typing the linspace command (MATLAB determines the correct spacing). When the number of elements is omitted, the default is 100.



**Example:**

v4 = linspace(0, 25, 5)

v5= linspace(30,10,5)

v6= linspace(0, 1000)

## Built-in Commands

- ⇨ zeros(3) % will create a 3×3 square matrix filled with zeros
- ⇨ zeros (2,5) % will create a 2×5 matrix filled with zeros
- ⇨ ones(1,4) % will create a row vector of size 4
- ⇨ magic (3) % will create a 3×3 matrix with fixed integers
- ⇨ randi (10, 3, 4) % will create a 3×4 matrix with values <=10

| Function | Description | Example | Result |
|---|---|---|---|
| `zeros(m, n)` | Matrix of all zeros | `zeros(2,3)` | 2×3 matrix of 0s |
| `ones(m, n)` | Matrix of all ones | `ones(3,2)` | 3×2 matrix of 1s |
| `eye(n)` | Identity matrix | `eye(3)` | 3×3 identity matrix |
| `rand(m, n)` | Uniform random values in (0, 1) | `rand(2,2)` | 2×2 matrix of random values |
| `Randi (m,n)` | Random integer values within m | `Randi (2,5)` | 5×5 matrix of random integer values <=m |
| `randn(m, n)` | Normally distributed random numbers (mean 0, std 1) | `randn(2,2)` | 2×2 normal distribution matrix |
| `magic(n)` | Magic square matrix | `magic(3)` | 3×3 magic square |
| `diag(v)` | Creates a diagonal matrix from vector `v` | `diag([1 2 3])` | 3×3 diagonal matrix |

- A magic matrix is a square matrix in which the sum of each row, each column, and both diagonals is the same. This constant sum is called the magic constant.
- You can also use different ways in combination to create a matrix.

```
>> A=[1:2:11; 0:5:25; linspace(10,60,6); 67 2 43 68 4 13]
A =
     1     3     5     7     9    11
     0     5    10    15    20    25
    10    20    30    40    50    60
    67     2    43    68     4    13
>>
```

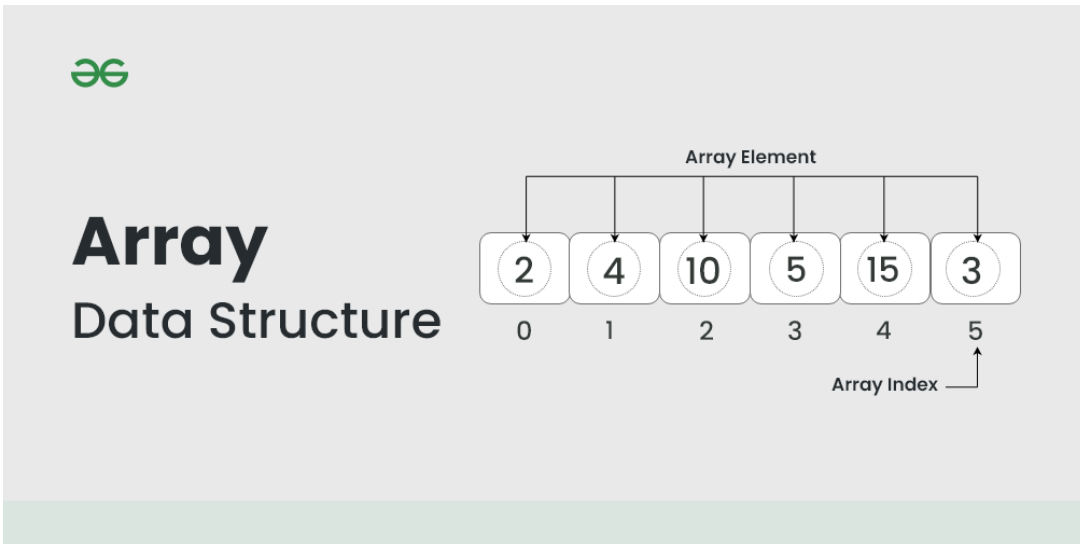
✓ One thing to keep in mind is that the dimension has to match. Otherwise, an error will be reported. For example,

```
x= [2:5; 10:2:22]
```

This command will cause an error: “dimensions of arrays being concatenated are not consistent.” This is because the two rows have different numbers of elements. They should be the same.

# Indexing

An index refers to the position of an element in an array. It can be used to access, extract, or modify that element. MATLAB's index starts from 1, unlike other programming languages such as C or Python, where the index starts with 0.



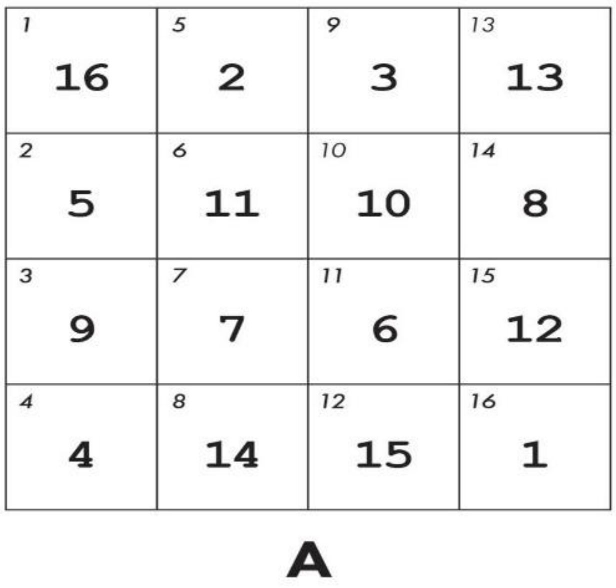
Every element in an array is assigned an index. So, if we create the array shown in the figure, element 2 will have an index value of 1. Use parentheses to extract elements.

```
X= [ 2, 4, 10, 5, 15, 3]

  ⇨  X(1)     % will return 2
  ⇨  X(6)     % will return 3
  ⇨  X(0)     % will return error!!! Index must be positive.
  ⇨  X(8)     % will return error!!! Out of bounds.
```

This is called **linear indexing**, where all the elements are assigned an integer index value starting from 1. This is also valid for matrix or multidimensional arrays. For a matrix, the index values are assigned column-wise.

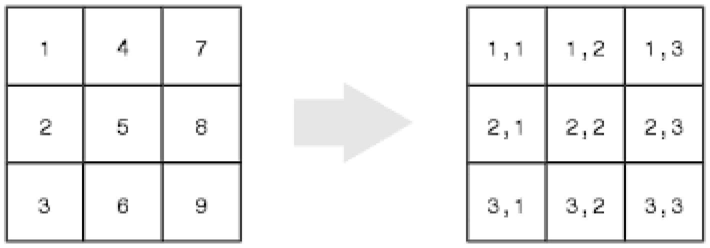
Indexing is used to extract elements from an array. They can also be used to modify the elements in the array.

```
  ⇨  X(1) = -2000

The original array X will be updated as follows: X =[-2000, 4, 10, 5, 15, 3]
```



- MATLAB supports 3 different ways of indexing:
  i. Linear indexing
  ii. Subscript indexing
  iii. Logical indexing

When working with a matrix, **subscript indexing** can be more intuitive because it directly refers to the element's row and column position, making it easier to understand and visualize.



```
A = [1 2 3;

      4 5 6;

     7 8 9];

   ⇨  A(2,3)   % Subscript indexing → 2nd row, 3rd column = 6
   ⇨  A(8)     % 8th element = 6
```

Logical indexing is conceptually equivalent to '**Boolean Masking**' used in other programming languages. This will be discussed later.

## Extracting elements from an array

We can use linear indices or subscripts to extract elements from an array or a matrix. Previously, you have seen how to extract a single element. What if you need to extract multiple elements?

This can be achieved by using the colon operator. A colon can be used to address a range of elements in a vector or a matrix. This is called '**slicing**'.

| | |
|---|---|
| **Vec (:)** | Refers to all the elements of the vector/matrix vec. This command will return all the elements (sequentially) in a row vector. While extracting elements this way using ':', it is pronounced as 'all'. |
| **Vec (m:n)** | Refers to elements m through n of the vector vec. While extracting elements this way using ':', it is pronounced as "m is to n". |

```
Vec = [ 100, 250, 300, 400, 600, 760]

  ⇨  Vec (3:5)      % will return [ 300, 400, 600]
  ⇨  Vec(2:2:5)     % will return [ 250, 400]
  ⇨  Vec (4: end)   % will return [400, 600, 760]
```

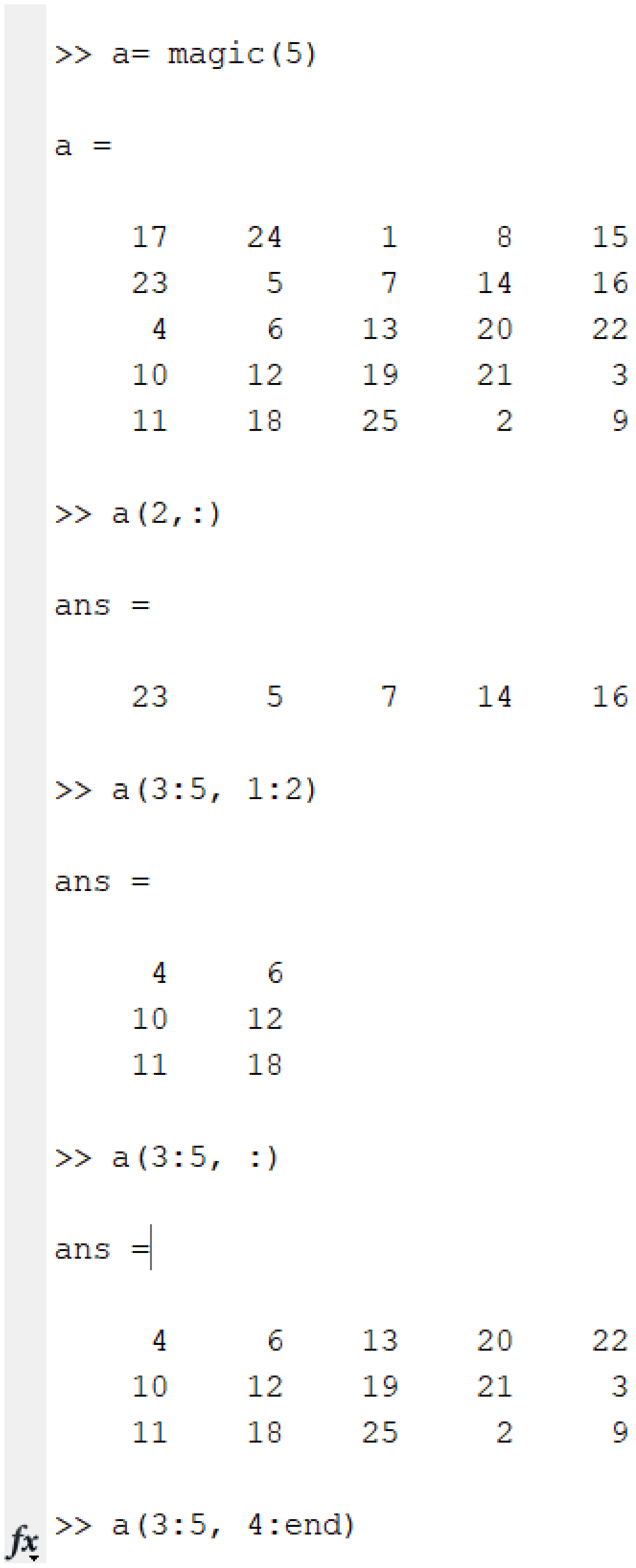
*What about a matrix? How to extract multiple elements?*

| | |
|---|---|
| A(:, n) | Refers to the elements in all the rows of column n of the matrix A. |
| A(n, :) | Refers to the elements in all the columns of row n of the matrix A. |
| A(:, m:n) | Refers to the elements in all the rows between columns m and n of the matrix A. |
| A(m:n, :) | Refers to the elements in all the columns between rows m and n of the matrix A. |
| A(m:n, p:q) | Refers to the elements in rows m through n and columns p through q of the matrix A. |
| A(m:end, :) | Refers to the elements in rows m through the end, and all columns |

- You can use the keyword '**end**' if you want to extract up to the last row/column. You can use this to avoid hard-coding.

**Lab Task:** *Create a magic matrix of size 8. Extract different rows/columns, different subsets from it.*

*Prepared by: Asif Newaz, A. K. M. Rakib*
*Lecturer, EEE, IUT*



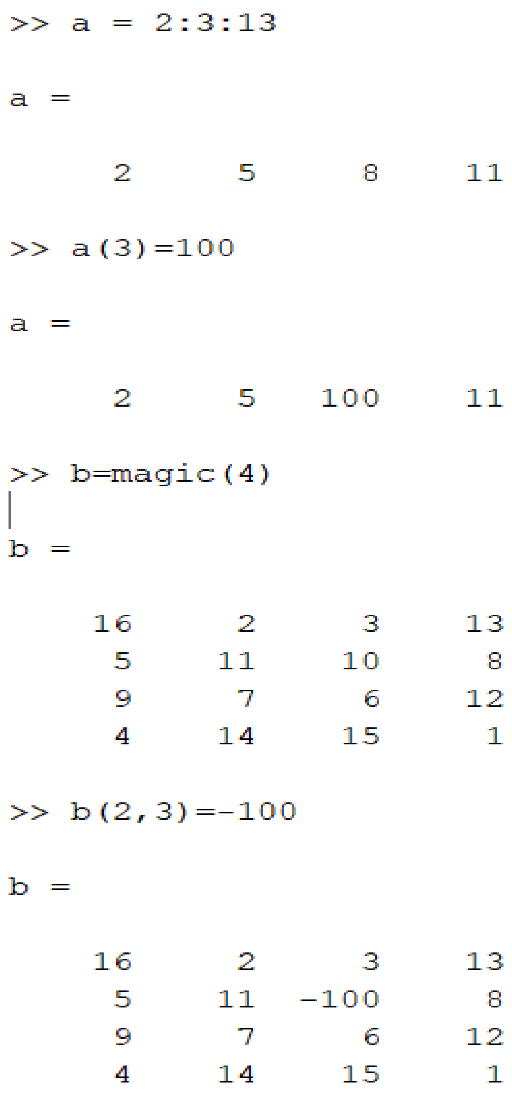
```
>> a= magic(5)

a =

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

>> a(2,:)

ans =

    23     5     7    14    16

>> a(3:5, 1:2)

ans =

     4     6
    10    12
    11    18

>> a(3:5, :)

ans =

     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9

>> a(3:5, 4:end)
```

⇨ *What will be the output from the last command?*

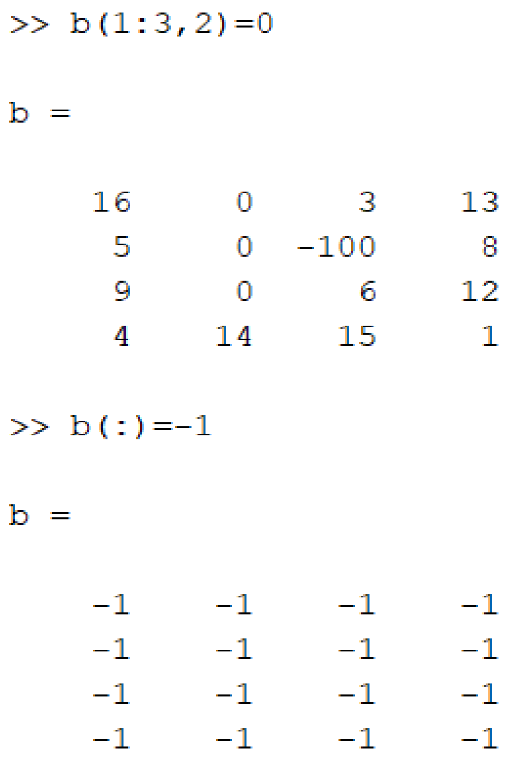
# Array Manipulation

## Modifying Elements:

You can modify the values in an array or a matrix using their index.



```
>> a = 2:3:13

a =

     2     5     8    11

>> a(3)=100

a =

     2     5   100    11

>> b=magic(4)

b =

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

>> b(2,3)=-100

b =

    16     2     3    13
     5    11  -100     8
     9     7     6    12
     4    14    15     1
```

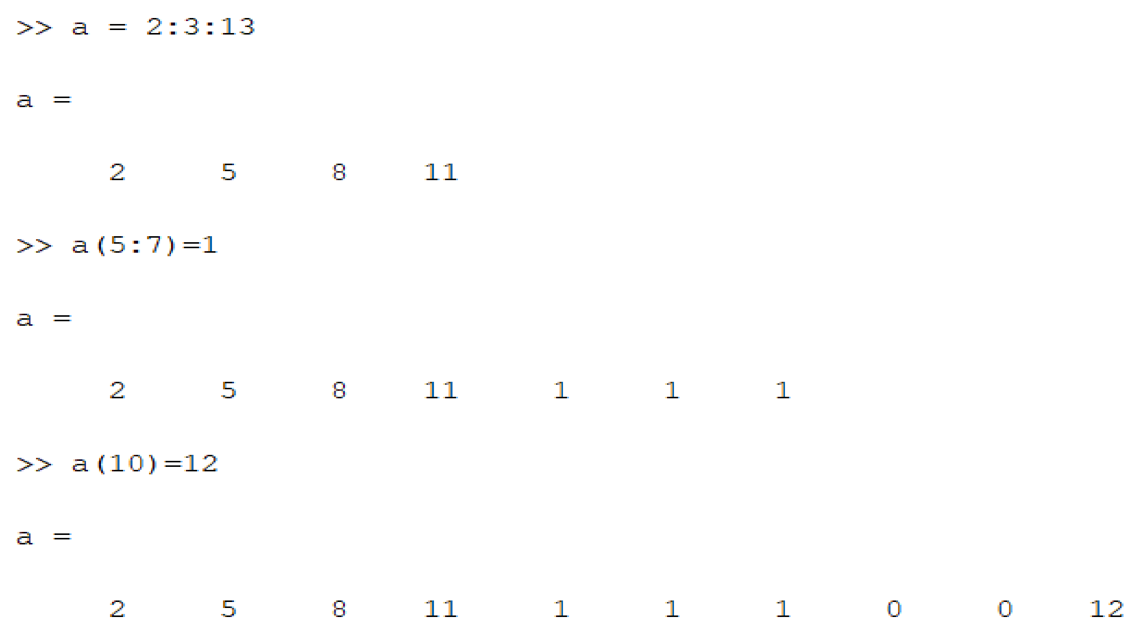


```
>> b(1:3,2)=0

b =

    16     0     3    13
     5     0  -100     8
     9     0     6    12
     4    14    15     1

>> b(:)=-1

b =

    -1    -1    -1    -1
    -1    -1    -1    -1
    -1    -1    -1    -1
    -1    -1    -1    -1
```

## Adding Elements:

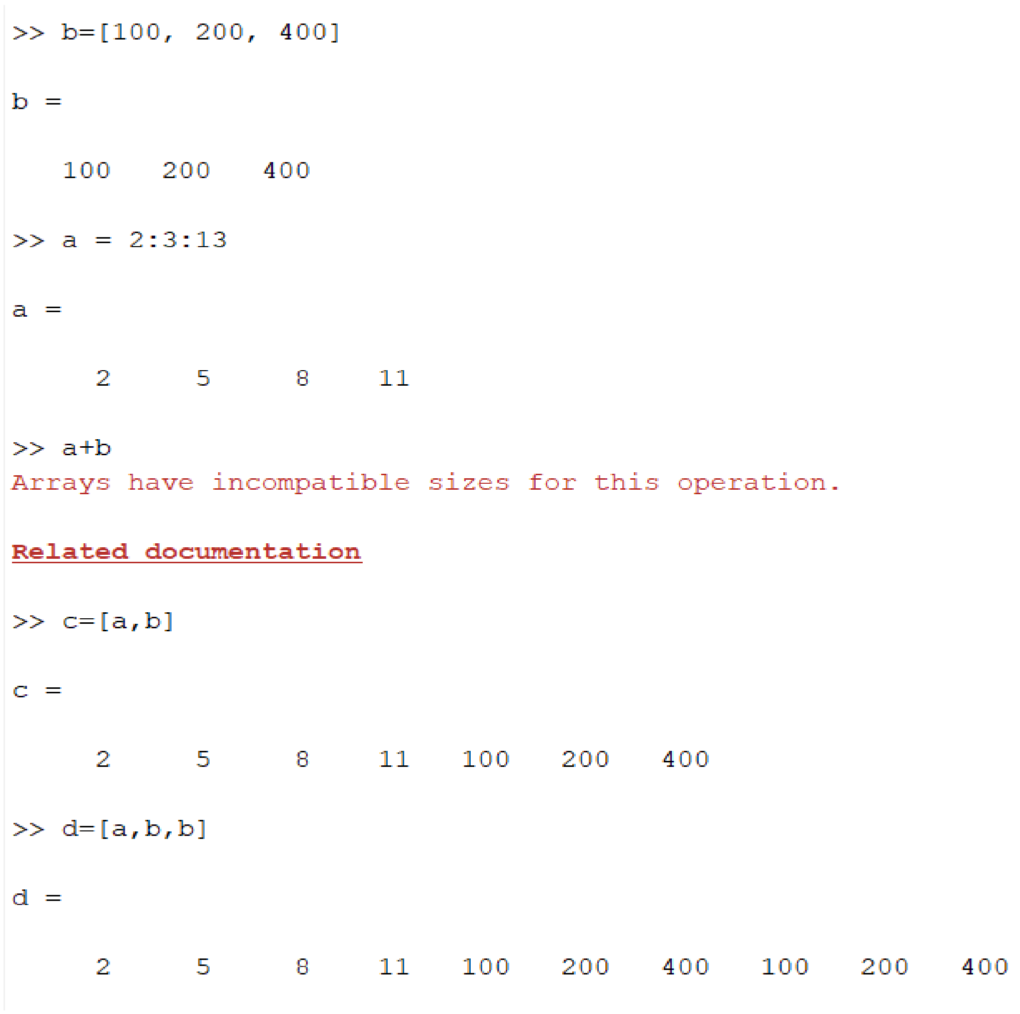
- You can add elements to your vector directly using the index (Figure 10)
- You can append existing vectors (Figure 11)

✓ Undefined index positions will be automatically assigned a 0 value.
✓ You can append multiple vectors together.



```
>> a = 2:3:13

a =

     2     5     8    11

>> a(5:7)=1

a =

     2     5     8    11     1     1     1

>> a(10)=12

a =

     2     5     8    11     1     1     1     0     0    12
```

*Figure 10*



```
>> b=[100, 200, 400]

b =

   100   200   400

>> a = 2:3:13

a =

     2     5     8    11

>> a+b
Arrays have incompatible sizes for this operation.

Related documentation

>> c=[a,b]

c =

     2     5     8    11   100   200   400

>> d=[a,b,b]

d =

     2     5     8    11   100   200   400   100   200   400
```

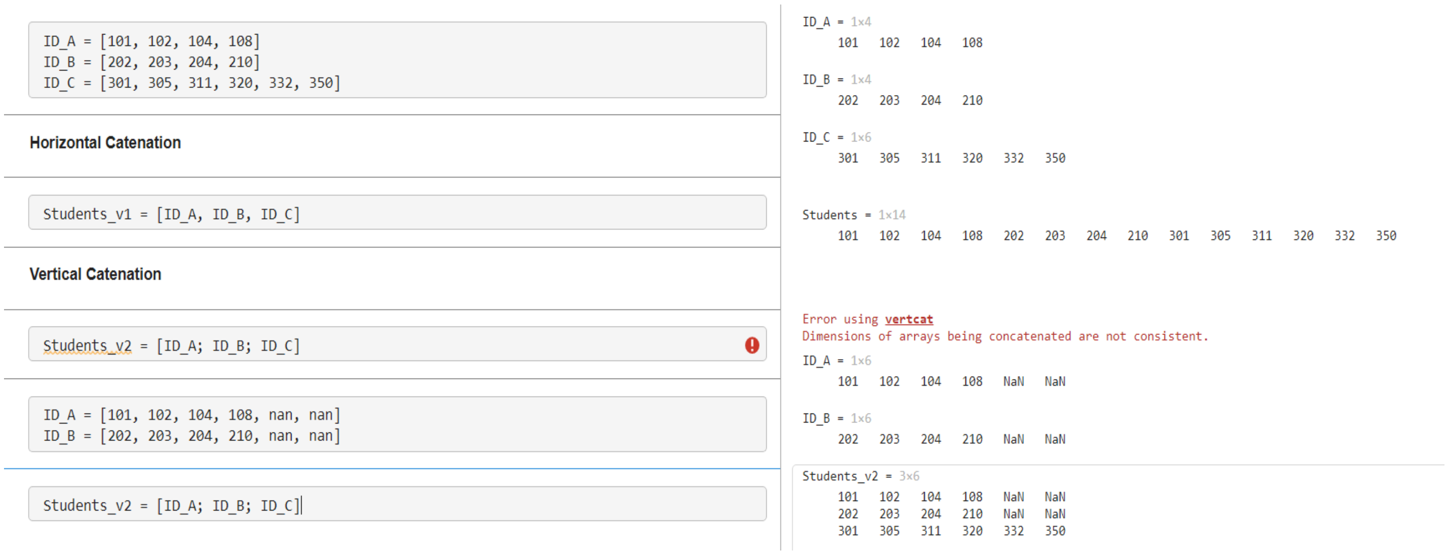
*Figure 11*

## Concatenation

Appending multiple arrays together to form a larger array – this operation is called ‘**Concatenation**’. This can be performed in two ways –

i. Horizontal catenation
ii. Vertical catenation

- During vertical catenation, you need to take into consideration the array size – they must match.



```
ID_A = [101, 102, 104, 108]
ID_B = [202, 203, 204, 210]
ID_C = [301, 305, 311, 320, 332, 350]
```

```
ID_A = 1×4
   101   102   104   108

ID_B = 1×4
   202   203   204   210

ID_C = 1×6
   301   305   311   320   332   350
```

**Horizontal Catenation**

```
Students_v1 = [ID_A, ID_B, ID_C]
```

```
Students = 1×14
   101   102   104   108   202   203   204   210   301   305   311   320   332   350
```

**Vertical Catenation**

```
Students_v2 = [ID_A; ID_B; ID_C]
```

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

```
ID_A = [101, 102, 104, 108, nan, nan]
ID_B = [202, 203, 204, 210, nan, nan]
```

```
ID_A = 1×6
   101   102   104   108   NaN   NaN

ID_B = 1×6
   202   203   204   210   NaN   NaN
```

```
Students_v2 = [ID_A; ID_B; ID_C]
```

```
Students_v2 = 3×6
   101   102   104   108   NaN   NaN
   202   203   204   210   NaN   NaN
   301   305   311   320   332   350
```

This is equally valid for a matrix.

```
>> x= [ones(3), zeros(3), eye(3), magic(3)]

x =

     1     1     1     0     0     0     1     0     0     8     1     6
     1     1     1     0     0     0     0     1     0     3     5     7
     1     1     1     0     0     0     0     0     1     4     9     2

>> y= [ones(3),zeros(3); eye(3), magic(3)]

y =

     1     1     1     0     0     0
     1     1     1     0     0     0
     1     1     1     0     0     0
     1     0     0     8     1     6
     0     1     0     3     5     7
     0     0     1     4     9     2
```

## Removing Elements

In MATLAB, you can remove elements from an array by assigning them as empty ([ ]). It can be done in several ways:

i. Removing by index

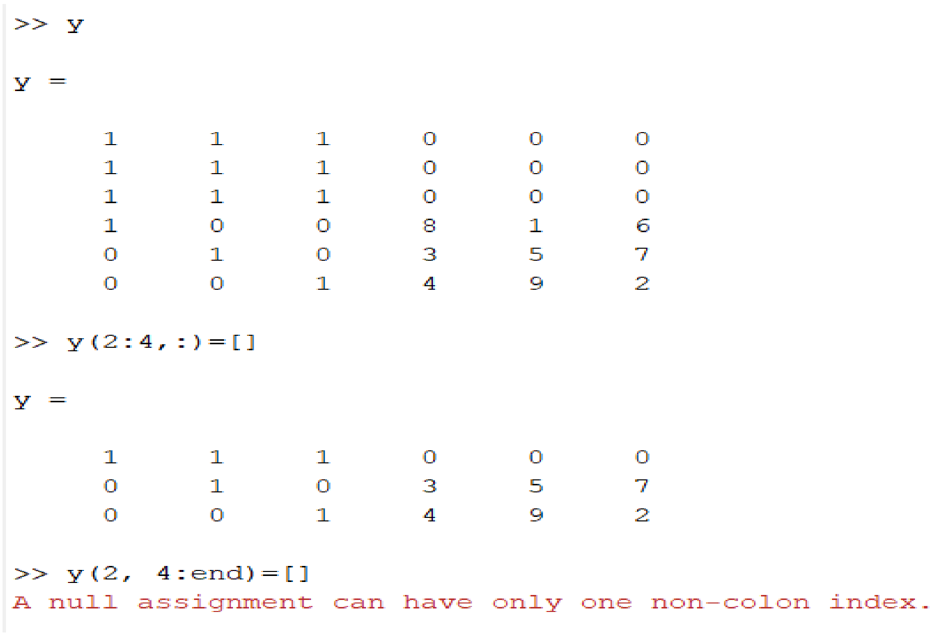
```
A = [10, 20, 30, 40, 50];

A(3) = [ ];  % Removes the 3rd element (30)

% Result: A = [10, 20, 40, 50]
```

ii. Removing multiple elements

```
A = [10, 20, 30, 40, 50];

A([2, 4]) = [];  % Removes 2nd and 4th elements (20 and 40)

% Result: A = [10, 30, 50]
```

iii. Remove by Value (Using Logical Indexing)

```
A = [10, 20, 30, 40, 50];

A(A == 20) = [];  % Removes all elements equal to 20

% Result: A = [10, 30, 40, 50]
```



```
>> y

y =

     1     1     1     0     0     0
     1     1     1     0     0     0
     1     1     1     0     0     0
     1     0     0     8     1     6
     0     1     0     3     5     7
     0     0     1     4     9     2

>> y(2:4,:)=[]

y =

     1     1     1     0     0     0
     0     1     0     3     5     7
     0     0     1     4     9     2

>> y(2, 4:end)=[]
A null assignment can have only one non-colon index.
```

*Prepared by: Asif Newaz, A. K. M. Rakib*
*Lecturer, EEE, IUT*

## Operations with Arrays

| Operation | Syntax | Description |
|---|---|---|
| **Addition & Subtraction** | C = A + B<br><br>C = A - B | Element-wise sum or difference of two arrays of the same size |
| **Array (Matrix) Multiplication** | C = A * B | Standard linear-algebraic (matrix) multiplication; inner dims must agree |
| **Array Division** | C = A / B<br><br>C = A \ B | Right-division (/) solves X*B = A;<br><br>left-division (\) solves A*X = B |
| **Inverse of a Matrix** | A_inv = inv(A) | Computes the (multiplicative) inverse of a matrix A |
| **Determinant** | d = det(A) | Returns the scalar determinant of a matrix A |
| **Element-by-Element Operations** | C = A .* B<br><br>C = A ./ B<br><br>C = A .^ B | Multiply, divide, or raise to the power of each corresponding element of A and B. |

**Homework:** Create some vectors and matrices. Test the above operations.

- **length (vec):** built-in function to obtain the length of the array (1D).
- **size(mat):** built-in function to obtain the size of the matrix or multidimensional array.
- **numel (x):** built-in function to obtain the number of elements in an array.

*Prepared by: Asif Newaz, A. K. M. Rakib*
*Lecturer, EEE, IUT*