Islamic University of Technology (IUT)

Organization of Islamic Cooperation (OIC)

Department of Electrical and Electronic Engineering (EEE)

COURSE NO	**:**	**EEE 4416**
LAB NO	**:**	**07**
TOPIC	**:**	**MISCELLANEOUS**

You have come a long way in your journey of learning MATLAB programming. By now, you should have a solid understanding of the fundamental concepts required to write programs in MATLAB. These foundational skills are crucial as they serve as the building blocks for solving more complex problems.

To summarize the main topics that we have covered so far –

i. Data structures: Numeric array, Cell array, Character array, String, Logical
ii. Indexing: Linear, Subscripts, Logical
iii. Conditionals: If-else, Switch-case
iv. Loops: For, While, Nested loop
v. Functions: Built-in, User-defined, Variable-length
vi. Techniques: Boolean masking, Matrix manipulation

If you have been actively engaging with the practice problems, you've likely developed a good level of proficiency in writing MATLAB code. This hands-on experience not only strengthens your programming logic but also helps you become more confident and efficient in applying MATLAB to real-world scenarios.

Before we move on to exploring practical applications and how MATLAB can be used in those contexts, it's important to first solidify a few key programming concepts. These are foundational yet often overlooked, and they play a vital role in writing efficient and effective code.

In addition, we'll cover some useful techniques and best practices that can significantly simplify your work in MATLAB. Mastering these will not only make your programming smoother but also help you solve problems more systematically and with greater ease.

The topics that will be covered in today's lab are as follows:

i. Vectorization
ii. Measuring Code Execution Time in MATLAB
iii. Pre-allocation
iv. Polymorphism
v. Broadcasting
vi. Multi-dimensional arrays

Vectorization

Vectorization is a powerful technique in MATLAB that allows you to replace loops with matrix and vector operations. This not only makes your code more concise but also significantly improves performance. While loops are slower and time-consuming, vectorized code takes significantly less time.

Let's look at a couple of examples to understand the concept.

Exercise – 01

Problem Statement: Given an array of integers, return the square of all elements in a separate array.

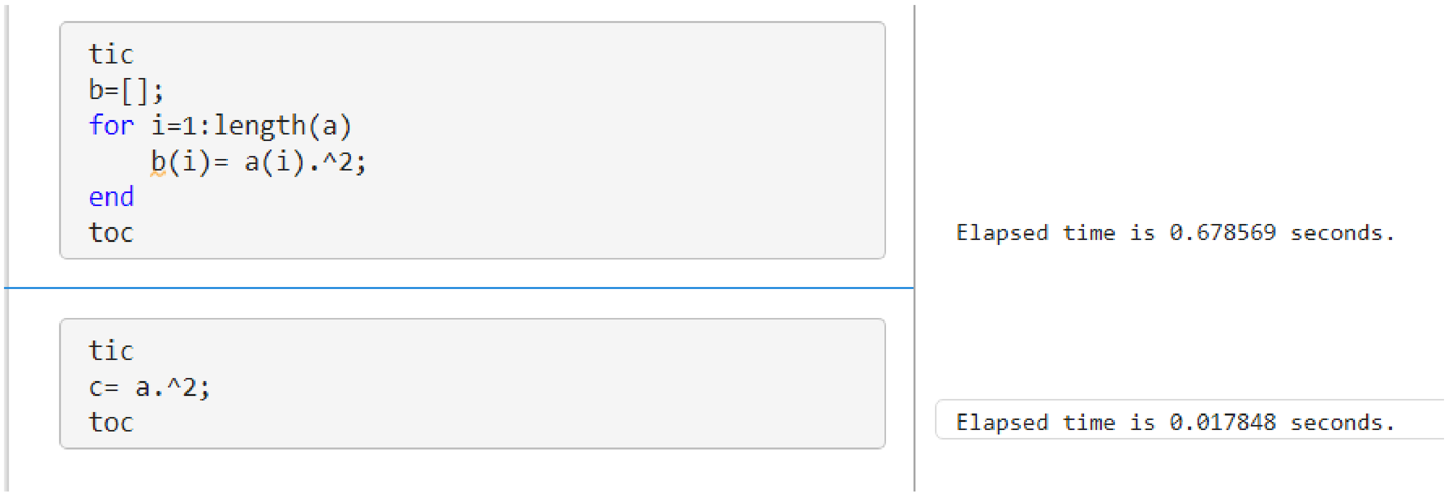
You have already solved this problem in a previous lab. However, one thing we did not consider before is the length of the array. You have worked with small arrays so far such as an array with a length of 10. But what if the size of the array is much larger – say, 10 million elements (data points) in the array? Given how you write your code, it can take a few seconds, or it can also take minutes to return the results – which is not desirable. When you have to deal with very large datasets (large arrays), efficiency becomes an issue. You want to write your code with a smaller execution time.

Efficiency is a crucial aspect of programming, especially when dealing with large datasets or real-time systems. An efficient program not only produces the correct output but does so quickly and with minimal use of system resources such as memory and CPU.

Consider the simple task of finding a specific word in a dictionary. If the dictionary contains only a few words, a basic approach—such as checking each word one by one—might be sufficient. However, if the dictionary contains hundreds of thousands of entries, this method becomes slow and inefficient. You don't want to wait a minute to return search results when you are trying to simply find a word. In such cases, using optimized techniques like binary search or MATLAB's built-in functions (which are highly optimized) can dramatically reduce execution time.

This example highlights why choosing the right algorithm and taking advantage of efficient data structures and built-in operations is essential. As problems scale, inefficiencies that seem negligible in small cases can quickly become bottlenecks, impacting performance and usability. Therefore, writing efficient code is not just about making things faster—it's about making solutions scalable, reliable, and suitable for real-world applications.

We can solve the problem using a for loop or we can utilize MATLAB's element-wise operation. Let's try both.



```
tic
b=[];
for i=1:length(a)
    b(i)= a(i).^2;
end
toc
```

```
Elapsed time is 0.678569 seconds.
```

```
tic
c= a.^2;
toc
```

```
Elapsed time is 0.017848 seconds.
```

✓ The *tic-toc* function allows you to measure elapsed time for running a code block.

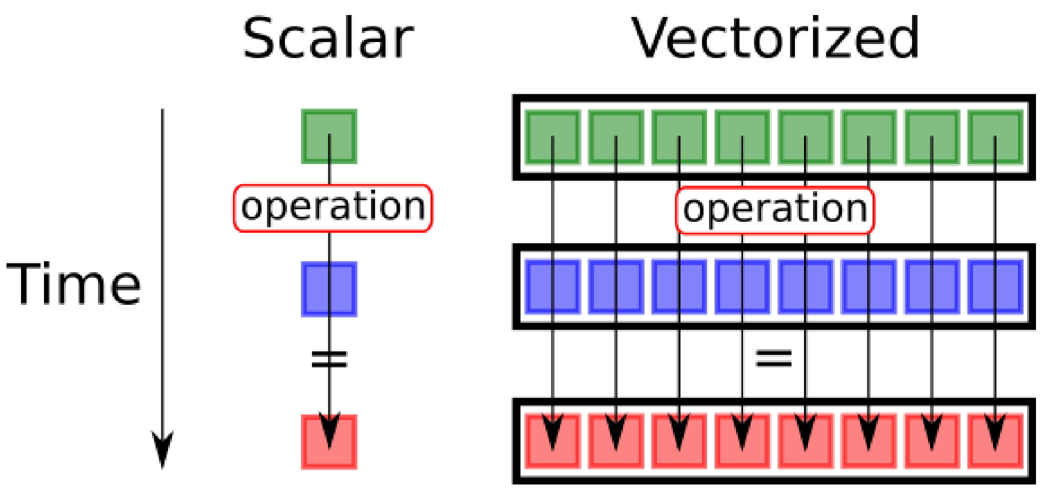
Here, as you can see, for the given array ‘a’ (random vector of size 10 M), using a for loop to compute the square values takes much longer than using an element-wise power operation. What are the reasons behind this?

How Vectorization Helps

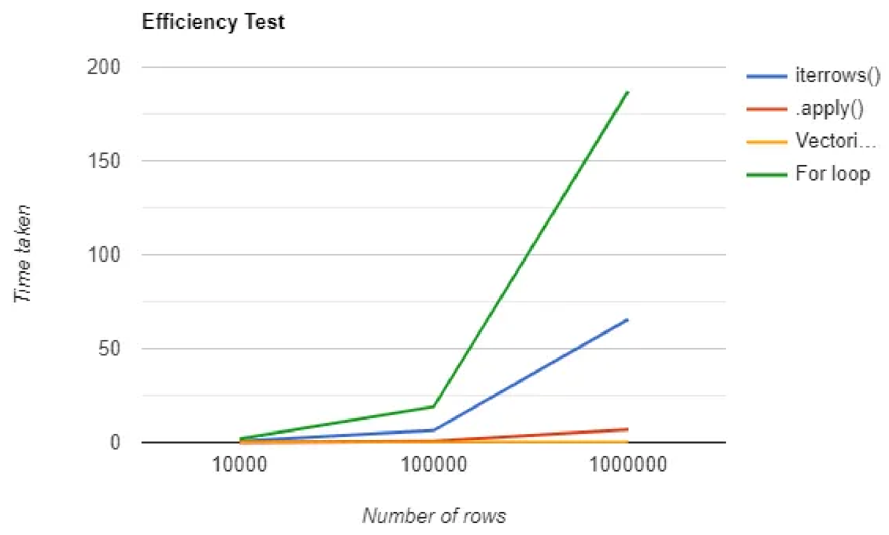
- When using a for loop, you are performing the operation on each element **individually and sequentially**, one after the other. Executing the ‘same’ operation again and again for millions of elements takes time.

- In the second approach, you are performing the operation on each element **in parallel**; the operation is performed in all the elements **simultaneously**. This approach taps into the high computational and processing power of computers – that they can perform multiple tasks at the same time.
  This vectorized approach (a.^2) leverages MATLAB’s ability to perform operations on entire arrays at once. Internally, these operations are highly optimized and executed in parallel at the lower (compiler) level, making them much faster and more efficient.

- There is another reason behind the higher execution time in the first case with the for loop. We will discuss it in the next section.



Vectorization removes the need for explicit loops and allows MATLAB to take advantage of its parallel processing capabilities**.** When you are performing the same operation, there is no need to use loops. You can directly apply the operation on a large array for parallel execution – this will significantly boost performance.

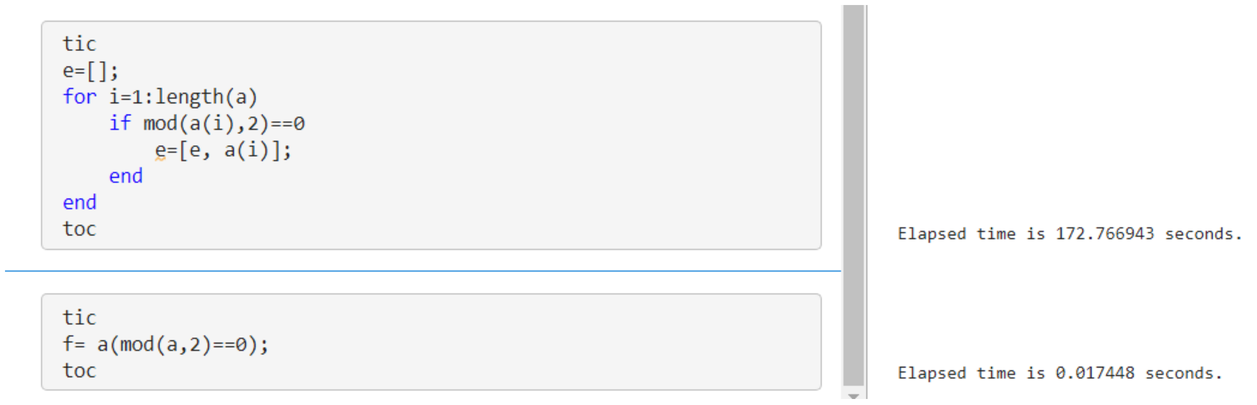


Exercise – 02

Problem Statement: Given an array of integers, you have to extract the even numbers from the list.

In this problem, you need to perform the modulo operation to check whether the integer is even or odd. If even, you need to place it in a separate array.

You have to perform the same 'modulo' operation on all the elements in the array. You can do it one by one using a for loop or you can execute the operation on the whole array simultaneously using Boolean masking.



```
tic
e=[];
for i=1:length(a)
    if mod(a(i),2)==0
        e=[e, a(i)];
    end
end
toc
```

```
Elapsed time is 172.766943 seconds.
```

```
tic
f= a(mod(a,2)==0);
toc
```

```
Elapsed time is 0.017448 seconds.
```

In the above case, an array of size 1 million was used. Using a for loop on a very large array (10M elements) will take a significantly large amount of time to execute. For 1 million instances, it took 3 minutes to return the results.

❖ Apart from using a for loop, there's another important factor contributing to the delay in execution time.

- At the beginning, an empty array 'e' is defined to store the even numbers.
- Inside the loop, each element of the original array is checked to determine whether it is even.
- If it is even, the number is appended to the array e.

- Each time a new element is added, MATLAB has to **resize the array** ‘e’ to accommodate the new value.
- Resizing an array repeatedly during execution leads to **frequent memory reallocation**, which significantly slows down performance.

This dynamic resizing is inefficient, especially for large arrays.

In contrast, a **vectorized approach** avoids this problem entirely. The memory for the result is allocated in a **single, optimized step**, and the operation is applied to the entire array at once. This not only eliminates loop overhead but also ensures efficient memory usage—making the vectorized code much faster and more scalable.

❖ Sometimes, it may become possible to avoid this frequent memory reallocation issue. We will discuss it in the next section.

Exercise – 03

Problem Statement: *Distance of Each Point from the Origin*

Suppose you have a set of 2D points stored in an N×2 matrix, and you want to calculate the Euclidean distance of each point from the origin (0,0).

The matrix looks like this –

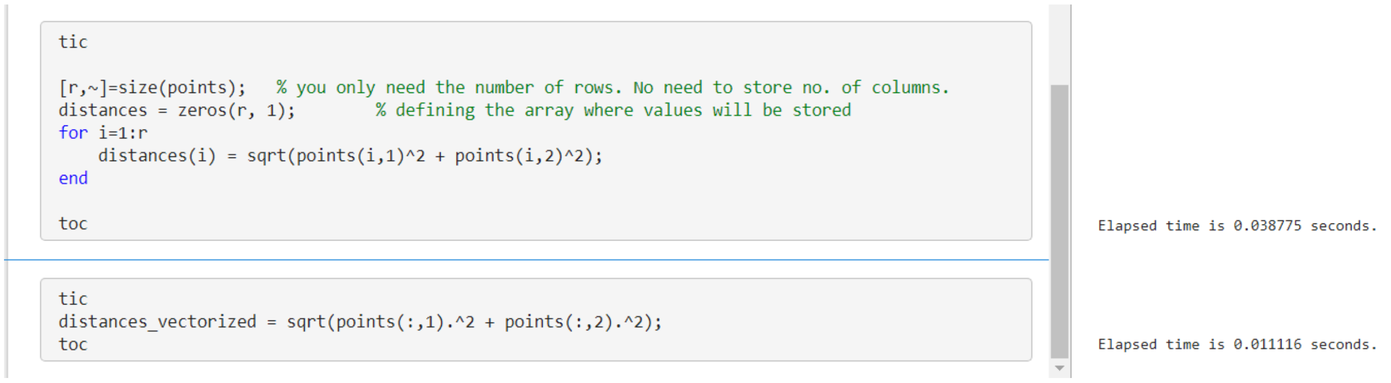
```
[452  552
 259   30
 995  641
 … … … ]
```

Every row represents a point in the XY plane where the 1st column value is the x-axis distance and the 2nd column value is the y-axis distance. You need to find the distance from the origin of this point and all other points in each row.

You can compute the distance of each point from the origin using the following formula:

$$distance = \sqrt{x^2 + y^2}$$

Using this formula, you can compute the distance for every point (row) using a for loop. Or you can write a vectorized solution.



```
tic

[r,~]=size(points);   % you only need the number of rows. No need to store no. of columns.
distances = zeros(r, 1);        % defining the array where values will be stored
for i=1:r
    distances(i) = sqrt(points(i,1)^2 + points(i,2)^2);
end

toc
```

```
Elapsed time is 0.038775 seconds.
```

```
tic
distances_vectorized = sqrt(points(:,1).^2 + points(:,2).^2);
toc
```

```
Elapsed time is 0.011116 seconds.
```

Exercise – 04

Problem Statement: *Perrin sequence*

You are familiar with the Fibonacci sequence. The Perrin sequence is much alike and can be solved in a similar way.

- Recurrence relation:

$$P(n) = P(n-2) + P(n-3), \qquad where, P(0) = 3,\ P(1) = 0,\ P(2) = 2$$

- Sequence: 3, 0, 2, 3, 2, 5, 5, 7, ...

⇨ Can we solve the problem without using a for loop?
⇨ Is there any vectorized solution possible to obtain the sequence?

Since each term depends on two earlier terms, full vectorization isn't possible. Each value must be computed sequentially, just like with Fibonacci. However, we can optimize it without using dynamic resizing or recursion.

Class Task: Write a function to return the first ‘n’ Perrin numbers.

Measuring Code Execution Time in MATLAB

Tic/toc

Tic/toc do a great job measuring elapsed wall-clock time in MATLAB, but they aren't perfect — there are a few nuances to keep in mind.

- They measure wall-clock time — i.e., real-time from start to finish, similar to a stopwatch.
- This includes everything in that block: computed operations, I/O, memory allocation, and even time your system spends multitasking in the background.
- Loop overhead and dynamic memory can impact results—e.g., **resizing arrays inside loops adds delays** unrelated to the logic you're timing.
- Some internal system processes may introduce jitters in timing results.

One thing you should keep in mind is that the time tic/toc returns is not absolute. If you run the same block multiple times, tic/toc won't return the same elapsed time every time. It may vary to some extent. However, overall, it gives you a good idea for comparison among multiple ways of writing your program.

Timeit

If you need more precise and repeatable timing of functions, MATLAB's timeit is better.

- It repeatedly runs the function — automatically warming up and averaging times.
- timeit calls the specified function multiple times and returns the median of the measurements.
- Ideal for micro benchmarking or comparing algorithms without external noise.

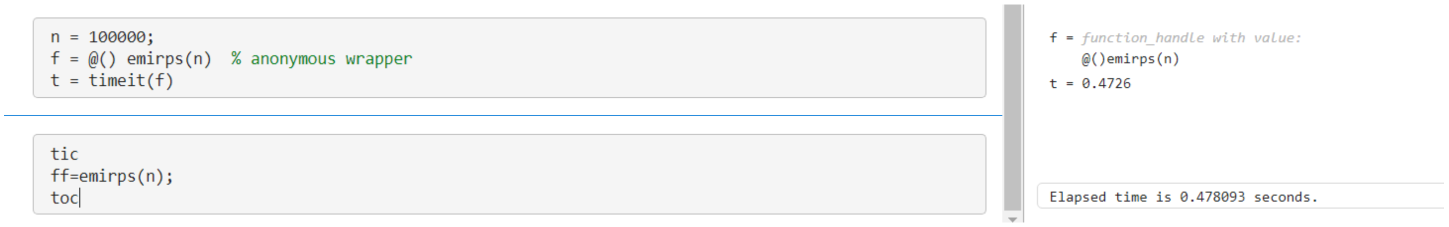
While tic/toc is accurate and convenient for simple timing measurements in MATLAB, for high-precision or benchmark comparisons, consider using timeit instead.

How to use timeit?

⇨ timeit takes a function handle (i.e., a function with @ and no input arguments).

⇨ Your function must not require input arguments. If it does, you need to wrap it inside an anonymous function.

⇨ I have a function called 'emirps' that takes one input argument. Let's try to use timeit to check its runtime.

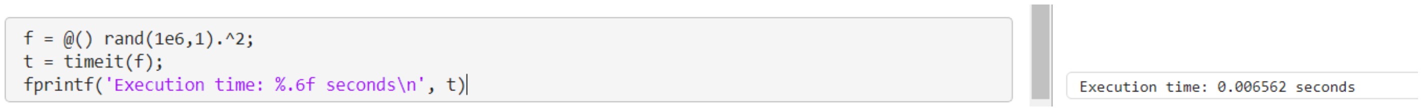


```
n = 100000;
f = @() emirps(n)  % anonymous wrapper
t = timeit(f)
```

```
f = function_handle with value:
   @()emirps(n)
t = 0.4726
```

```
tic
ff=emirps(n);
toc
```

```
Elapsed time is 0.478093 seconds.
```



```
f = @() rand(1e6,1).^2;
t = timeit(f);
fprintf('Execution time: %.6f seconds\n', t)
```

```
Execution time: 0.006562 seconds
```

Asif Newaz
Lecturer, EEE, IUT

Pre-allocation

Previously you have seen how resizing an array repeatedly during execution (for loop) leads to frequent memory reallocation, which significantly slows down performance. There are certain scenarios where it may be possible to avoid this – which can save a lot of time and significantly improve runtime.

- When you **know the size of the output array** prior to running your for loop, you can use pre-allocation.

Pre-allocation means reserving memory space for a variable (usually an array) before using it in a loop or iterative operation.

Let's try to understand the concept using some examples.

Without pre-allocation:

```
for i = 1:10000
   A(i) = i^2;       % Array grows on each iteration
end
```

- MATLAB dynamically resizes A every time a new element is added.
- This involves repeated memory allocation + copying, which is very slow, especially in large loops.

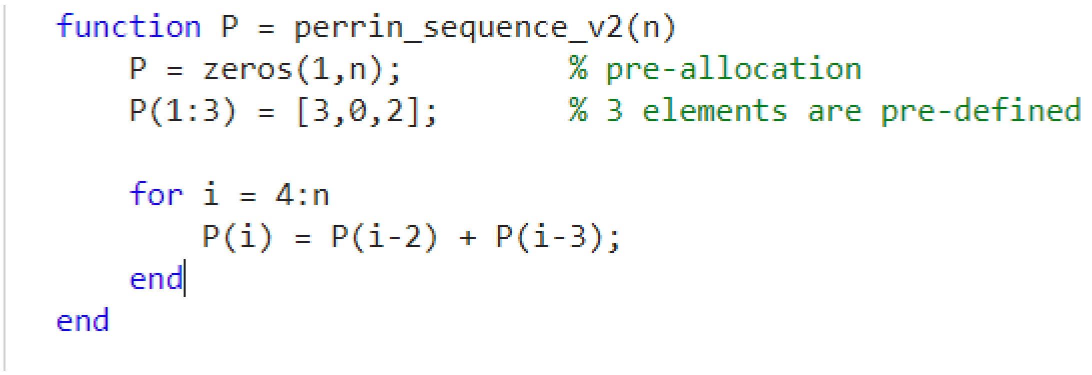
With Pre-allocation:

```
A = zeros(1, 10000);               % Preallocate memory
for i = 1:10000
   A(i) = i^2;
end
```

- Here, memory is allocated once.
- No resizing happens during the loop.
- At each iteration, the value is placed in an already-created memory space.
- Much faster and more efficient.

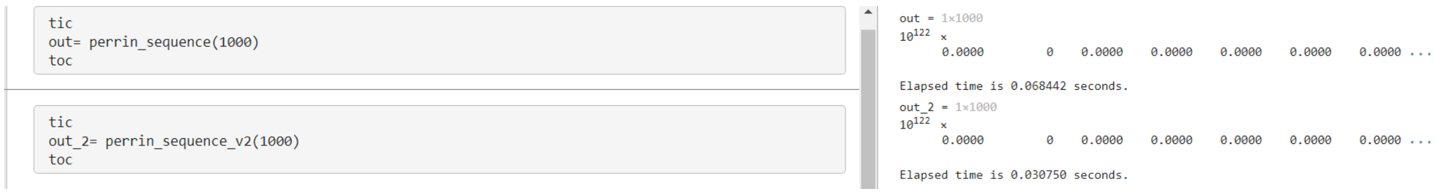
Let's try and solve exercise – 04 with or without pre-allocation and see the difference in runtime.

```
function P = perrin_sequence(n)
    P= [3,0,2];       % 3 elements are pre-defined
    for i = 4:n
        P(i) = P(i-2) + P(i-3);
    end
end
```



```
function P = perrin_sequence_v2(n)
    P = zeros(1,n);          % pre-allocation
    P(1:3) = [3,0,2];        % 3 elements are pre-defined

    for i = 4:n
        P(i) = P(i-2) + P(i-3);
    end
end
```

Check the difference in execution time for n = 1000. The time required has almost halved.



```
tic
out= perrin_sequence(1000)
toc
```

```
tic
out_2= perrin_sequence_v2(1000)
toc
```

```
out = 1×1000
10^122 ×
      0.0000         0    0.0000    0.0000    0.0000    0.0000    0.0000 ...

Elapsed time is 0.068442 seconds.
out_2 = 1×1000
10^122 ×
      0.0000         0    0.0000    0.0000    0.0000    0.0000    0.0000 ...

Elapsed time is 0.030750 seconds.
```

❖ It may not be always possible to use pre-allocation. For instance, in exercise – 02, you had to extract the even numbers from the list. You have no way to learn prior to executing your code how many even numbers are going to be there. So, you cannot pre-define your output array.

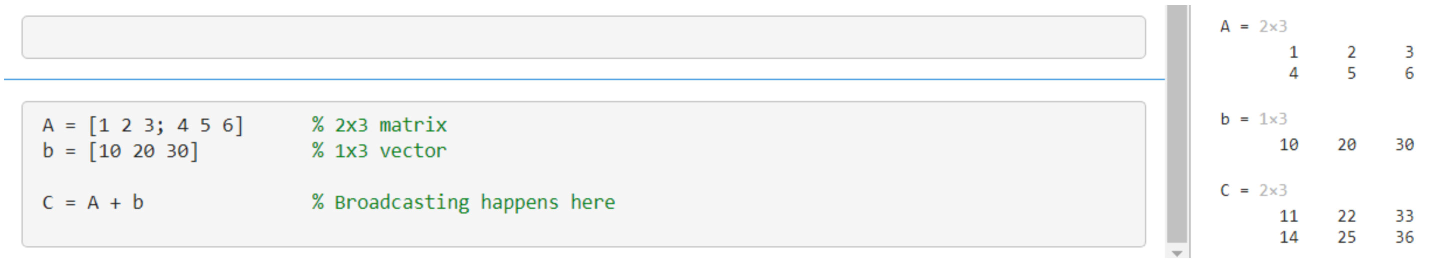
❖ So, from now on, whenever you are aware of the output array size before running a for loop, use pre-allocation using functions like zeros or ones.

Broadcasting

Broadcasting refers to MATLAB’s ability to **automatically expand arrays** of different sizes so that operations between them can still be performed element-wise, without explicit looping or replication.

This behavior was introduced in MATLAB R2016b, and it's similar to broadcasting in NumPy (Python).

Look at the following example –



```
A = [1 2 3; 4 5 6]       % 2x3 matrix
b = [10 20 30]           % 1x3 vector

C = A + b                % Broadcasting happens here
```

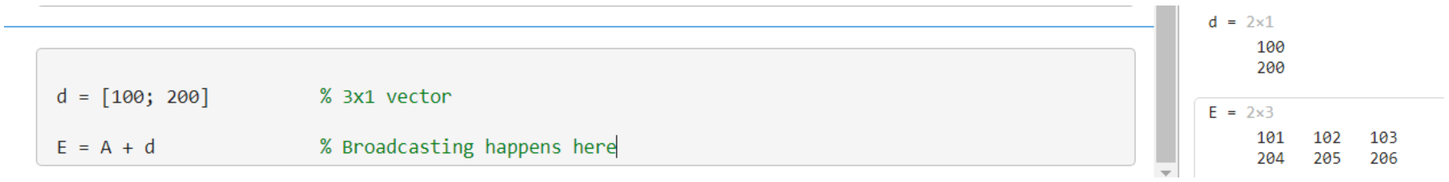
```
A = 2×3
     1     2     3
     4     5     6

b = 1×3
    10    20    30

C = 2×3
    11    22    33
    14    25    36
```

Here, as you can see –

- A is a matrix of size 2 by 3.
- B is a matrix of size 1 by 3.
    - How is this possible to add two matrices of different sizes?

This is achieved by MATLAB’s in-built broadcasting property. MATLAB automatically expands b to match the size of A row-wise. Then b is added to A in an element-wise add operation.

It is also possible to add a 2 by 1 matrix with A. Here, matrix d is automatically expanded column-wise to match the size of A.

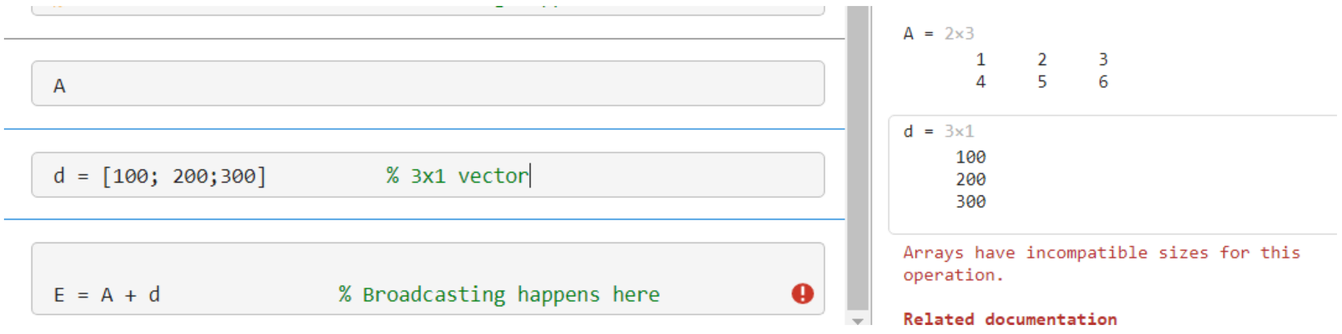


```
d = [100; 200]           % 3x1 vector

E = A + d                % Broadcasting happens here
```

```
d = 2×1
   100
   200

E = 2×3
   101   102   103
   204   205   206
```

Broadcasting Rules

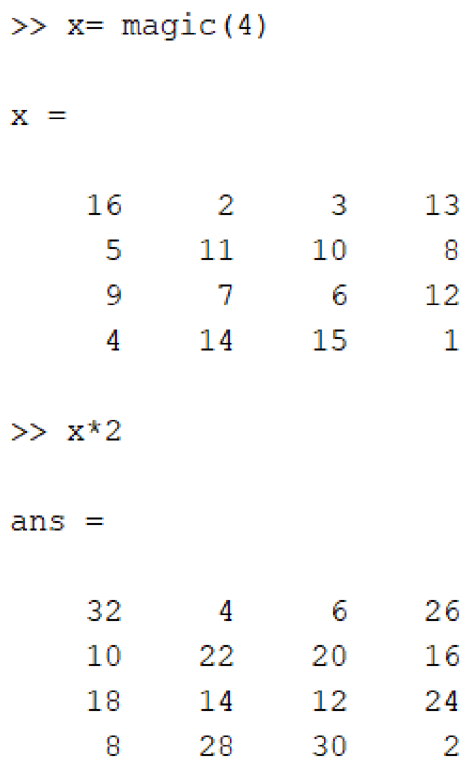
- Arrays must be compatible in size.
- If the sizes differ, MATLAB will implicitly **expand the smaller array** to match the larger one **along singleton dimensions.**
- Dimensions must either be equal, or one of them must be 1.



```
A

d = [100; 200;300]          % 3x1 vector

E = A + d                   % Broadcasting happens here
```

```
A = 2×3
     1     2     3
     4     5     6

d = 3×1
   100
   200
   300

Arrays have incompatible sizes for this
operation.

Related documentation
```

Asif Newaz
Lecturer, EEE, IUT



```
>> x= magic(4)

x =

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

>> x*2

ans =

    32     4     6    26
    10    22    20    16
    18    14    12    24
     8    28    30     2
```

Exercise – 05

Problem Statement: Multiplication Table

Given a list of numbers(n), create a multiplication table up to x. Write a function called 'multi_table' that takes n and x as input parameters.

Test Case – 01:

- Input: n = 1:3, x = 5
- Output:

```
[1    2    3    4    5
  2    4    6    8    10
  3    6    9    12    15]
```

Test Case – 02:

- Input: n = [10, 20], x = 10
- Output:

```
[10    20    30    40    50
  20    40    60    80    100]
```

❖ You can use a for loop for this. Can you solve the problem without a loop, using broadcasting?

3D Array

So far, we've worked with only a 2-dimensional matrix. We can also create a 3D, 4D, or higher dimensional matrix very easily. Let's try to understand first what a 3D matrix is.

Say, you are taking notes in your diary. The page on which you're writing can have an x-axis and y-axis i.e. can be divided into a 20-by-20 matrix. The next page also has the same size and can be divided into the same 20-by-20 matrix.

Now, try to look at your diary from the outside. There are say 100 pages stacked on top of one another; each of which (page) is a 20-by-20 matrix. So, your diary is actually a 3D matrix of size – (20, 20, 100).

Here, 100 is no. of pages which is the 3rd dimension. Each page has a 2D matrix of size (20,20).

Now, Say I want to create a 3D matrix of size (5, 5, 3).

So, I have three 2D matrices of size (5, 5). Let's create 3 such matrices first.

- M1= eye (5)
- M2= spiral (5)
- M3= magic (5)

Now we've to stack them on top of one another.

- Mat_3d (:, :, 1) = M1
- Mat_3d (:, :, 2) = M2
- Mat_3d (:, :, 3) = M3

You can check from your workspace that Mat_3d is a 3D matrix.

- zeros(3, 3, 6)
  - ⇨ this will create a 3D matrix with six 2D matrix of size 3x3.

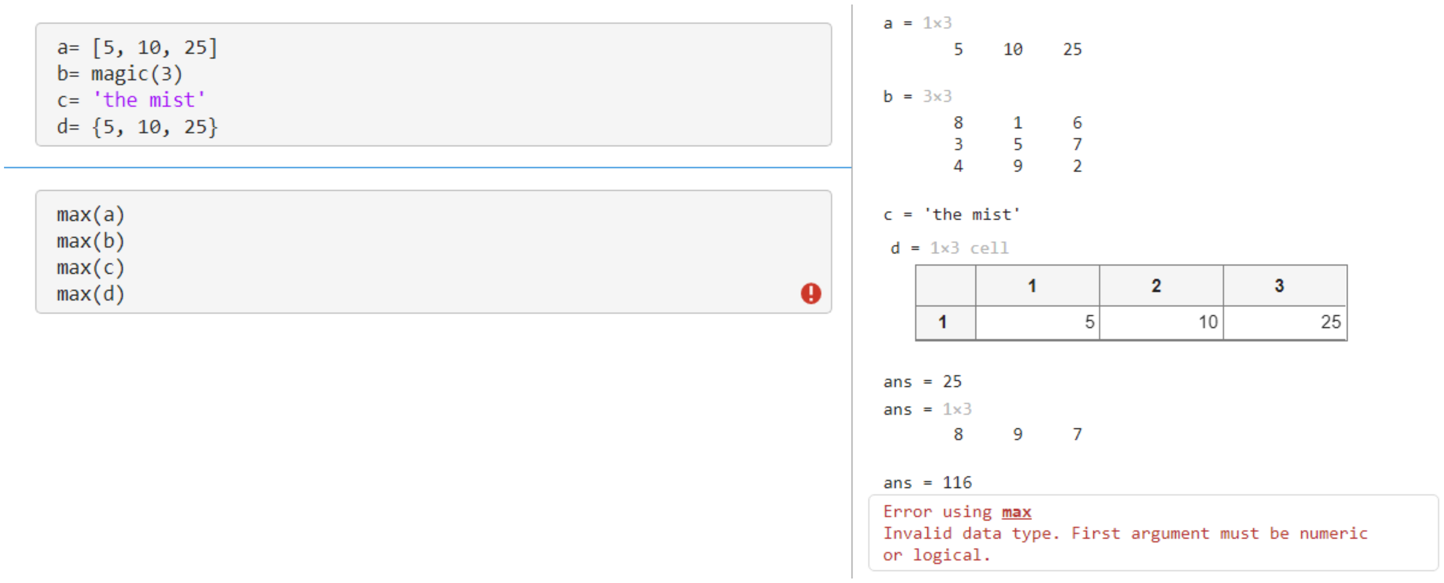
- ❖ Now, if you can consider 10 such books stacked in a box – then, the box can be considered to be representing a 4D object where no. of books is the 3rd dimension.

- ❖ When Are 3D Matrices Useful => Image processing: RGB images are 3D matrices (height × width × 3 channels).

Polymorphism

MATLAB functions can handle different types of inputs like vectors, matrices, char arrays, and even tables - without having to be defined by the user what kind of input is fed to the function.

This particular feature that MATLAB offers is called ‘Polymorphism’. It is not available in many other programming languages.



```
a= [5, 10, 25]
b= magic(3)
c= 'the mist'
d= {5, 10, 25}
```

```
max(a)
max(b)
max(c)
max(d)
```

```
a = 1×3
       5    10    25

b = 3×3
       8     1     6
       3     5     7
       4     9     2

c = 'the mist'
d = 1×3 cell
```

	1	2	3
1	5	10	25

```
ans = 25
ans = 1×3
       8     9     7

ans = 116
Error using max
Invalid data type. First argument must be numeric
or logical.
```

As you can see, the ‘max’ can operate with arrays of different sizes as well as character arrays, but not cell arrays. The data type needs to be homogeneous.

Asif Newaz
Lecturer, EEE, IUT