Scientific Computing Software. Matlab

Lecture 6. Matlab programming

Data types and structures, Control flow, Functions, Scripts and Performance

Nasedkina A. A.



Data types and structures

- arrays
- cells
- structs
- tables

Why Data Types and Structures Matter

- MATLAB is built around the concept of matrices, but modern engineering workflows require diverse data types to handle complex datasets.
- Choosing the correct data container is the first step toward writing efficient code.
- Choosing the right container affects:
  - Clarity — does the code express intent?
  - Access speed — named access vs. positional access
  - Memory — homogeneous vs. heterogeneous storage
  - Interoperability — export to CSV, database, etc.

Type	Homogeneous?	Named Access?	Typical Use
Numeric array	Yes	No	Linear algebra, signals
Cell array	No	No (index only)	Mixed collections
Struct	No	Yes (fields)	Records, metadata
Table	No (per column)	Yes (variables)	Tabular data

Core Data Structures in Matlab

- **Numerical Arrays / Matrices:**
  - The fundamental MATLAB data type.
  - Elements must be of the identical primitive type (usually double).
  - Operations are highly optimized for hardware acceleration.
- **Cell Arrays ({}):**
  - Containers that hold data of different types and sizes in separate cells.
  - Useful for collections of text, or arrays of varying lengths.
- **Structures (struct):**
  - Data containers accessed via named fields (e.g., patient.name, patient.id).
  - Excellent for grouping related parameters.
- **Tables (table):**
  - Tabular containers for mixed-type data, matching spreadsheets or SQL databases.
  - Columns have names, and metadata can track variable units.
    - .

Arrays in Matlab

- In MATLAB, every variable is a multidimensional array.
- MATLAB stands for *MATrix LABoratory*, meaning it treats everything—from a single scalar number to large multi-dimensional grids—as an array by default

- You construct arrays manually using square brackets [].
- **Row Vector:** Separate numbers with spaces or commas ,
- **Column Vector:** Separate numbers with semicolons ;
- **Matrix (2D Array):** Separate elements with spaces/commas, and rows with semicolons.

```
row_vec = [1, 2, 3, 4]
% Result: 1x4 row vector
```

```
row_vec = 1×4
     1     2     3     4
```

```
col_vec = [1; 2; 3; 4]
% Result: 4x1 column vector
```

```
col_vec = 4×1
     1
     2
     3
     4
```

```
matrix  = [1 2 3; 4 5 6]
% Result: 2x3 matrix
```

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

Arrays in Matlab: regular spacing

- **Colon Operator (`start:step:end`):** Creates evenly spaced values. If you leave out the `step` , it defaults to `1` .
- **`linspace(start, end, n)` :** Generates exactly `n` points evenly split between the start and end values.

```
grid1 = 1:2:9
```

```
grid1 = 1×5
     1     3     5     7     9
```

```
grid2 = 1:5
```

```
grid2 = 1×5
     1     2     3     4     5
```

```
grid3 = linspace(0, 10, 5)
```

```
grid3 = 1×5
         0    2.5000    5.0000    7.5000   10.0000
```

Built-in Utility Functions for Common Array Matrices

Function	Description	Example
`zeros(r, c)`	Fills an array with all zeros	`zeros(2, 3)`
`ones(r, c)`	Fills an array with all ones	`ones(3, 1)`
`rand(r, c)`	Fills uniformly distributed random numbers between 0 and 1	`rand(2, 2)`
`eye(n)`	Generates an n × n identity matrix (1s on the main diagonal)	`eye(3)`

```
Zeros=zeros(2,3)
```

```
Zeros = 2×3
     0     0     0
     0     0     0
```

```
Ones=ones(3,1)
```

```
Ones = 3×1
     1
     1
     1
```

```
% Random real matrix with
% values in the range (0,1)
% (uses uniform distribution)
A=rand(2,2)
```

```
A = 2×2
    0.2348    0.8212
    0.3532    0.0154
```

```
B=rand(3)
```

```
B = 3×3
    0.0430    0.7317    0.5470
    0.1690    0.6477    0.2963
    0.6491    0.4509    0.7447
```

```
% Random interger matrix
% with values in the range [-5,5]
C=randi([-5 5],2,3)
```

```
C = 2×3
     3    -2     4
     0     5     1
```

```
% Identity matrix
I=eye(3)
```

```
I = 3×3
     1     0     0
     0     1     0
     0     0     1
```

Array Indexing & Accessing Data

- **1-Based Indexing:** The first element in a MATLAB array is always at index `1`.
- **Row-Column Subscripts:** Standard lookup uses `Array(row, column)`.
- **The Colon Operator (`:`):** Acts as a wildcard to mean "all elements" in that dimension.

```
A = [10 20 30; 40 50 60; 70 80 90]
```

```
A = 3×3
     10    20    30
     40    50    60
     70    80    90
```

```
% Matrix is stored by columns
% as a long vector-column
A(:)
```

```
ans = 9×1
     10
     40
     70
     20
     50
     80
     30
     60
     90
```

```
A(3) % Returns 70 (3rd element)
```

```
ans = 70
```

```
A(1) % Returns 10 (first element)
```

```
ans = 10
```

```
A(end) % Returns 90 (last element)
```

```
ans = 90
```

```
% Returns 60 (Row 2, Column 3)
val = A(2, 3)
```

```
val = 60
```

```
% Returns the entire second row:
% [40, 50, 60]
row2 = A(2, :)
```

```
row2 = 1×3
     40    50    60
```

```
% Returns the entire
% third column:
% [30; 60; 90]
col3 = A(:, 3)
```

```
col3 = 3×1
     30
     60
     90
```

```
% Returns a sub-matrix
sub = A(1:2, 2:3)
```

```
sub = 2×2
     20    30
     50    60
```

Matrix Operations vs. Element-Wise Operations

Standard operators (`*` , `/` , `^`) perform strict linear algebra matrix mathematics. If you want to apply the operation to **every individual element** rather than the whole matrix, add a period (`.`) before the operator.

```
X = [1 2; 3 4]
```

```
X = 2×2
     1     2
     3     4
```

```
Y = [2 0; 1 2]
```

```
Y = 2×2
     2     0
     1     2
```

```
% Matrix Multiplication
% Multiplies X and Y using linear algebra rules
M = X * Y
```

```
M = 2×2
     4     4
    10     8
```

```
% Element-by-Element Multiplication
% Result: [1*2, 2*0; 3*1, 4*2]
% -> [2 0; 3 8]
E = X .* Y
```

```
E = 2×2
     2     0
     3     8
```

```
% Element-by-Element Exponentiation
% Result: Squares each individual number
% -> [1 4; 9 16]
Sq = X .^ 2
```

```
Sq = 2×2
     1     4
     9    16
```

Numeric Arrays — The Default

```
% Matrix 5x4 of random integers
A = randi([1 100], 5, 4)
```

```
A = 5x4
    53    44    38    93
    42     2    20     6
    66    99    49    74
    63    17    34    27
    30    11    96    43
```

```
% Logical indexing (key to efficiency)
A1=A;
mask = A1 > 50
```

```
mask = 5x4 logical array
   1   0   0   1
   0   0   0   0
   1   1   0   1
   1   0   0   0
   0   0   1   0
```

```
% Replace all >50 with 0
A1(mask) = 0
```

```
A1 = 5x4
     0    44    38     0
    42     2    20     6
     0     0    49     0
     0    17    34    27
    30    11     0    43
```

```
% Multiply all odd elements by 100
A2=A;
odds=mod(A2,2)==1
```

```
odds = 5x4 logical array
   1   0   0   1
   0   0   0   0
   0   1   1   0
   1   1   0   1
   0   1   0   1
```

```
A2(odds)=100*A2(odds)
```

```
A2 = 5x4
        5300          44          38        9300
          42           2          20           6
          66        9900        4900          74
        6300        1700          34        2700
          30        1100          96        4300
```

```
% Deleting elements
v = [1 2 3 4 5]
```

```
v = 1x5
     1     2     3     4     5
```

```
v(v > 3) = []
```

```
v = 1x3
     1     2     3
```

Cell Arrays — Heterogeneous Containers

```
myCell = {42, 'Temperature Data', [1, 2, 3, 4]}
```

myCell = 1×3 cell

	1	2	3
1	42	'Temperature Data'	[1,2,3,4]

```
firstCellContent = myCell{1} % Access content using curly braces
```

firstCellContent = 42

```
C = {rand(2), 'experiment', true, {1, 2, 3}}
```

C = 1×4 cell

	1	2	3	4
1	[0.8797,0.2607;0.8178,0.5944]	'experiment'	1	*1×3 cell*

```
% Access: () returns a cell; {} returns the contents
C(1)        % 1×1 cell array
```

ans = *1×1 cell array*

{2×2 double}

```
C{1}        % 3×3 numeric matrix
```

ans = 2×2

```
    0.8797    0.2607
    0.8178    0.5944
```

Note
A cell array holds **anything in each cell** — numbers, strings, matrices, even other cells

When to use: collecting results of varying size/type before converting to a table

Structs — Named Fields

Structs group related data with **meaningful field names**

```
% Structure (Named fields)
motorParams.Resistance = 0.5;
motorParams.Inductance = 0.01;
motorParams.Model = 'Brushless';
motorParams
```

```
motorParams = struct with fields:
    Resistance: 0.5000
    Inductance: 0.0100
         Model: 'Brushless'
```

```
student.name   = 'Alice';
student.id     = 12345;
student.grades = [90 85 88];
student.avg    = mean(student.grades);

% Nested struct
student.address.city = 'Reno';
student.address.zip  = '89557';
student
```

```
student = struct with fields:
       name: 'Alice'
         id: 12345
     grades: [90 85 88]
        avg: 87.6667
    address: [1×1 struct]
```

```
% Struct arrays (records)
exp(1).name = 'run_A'; exp(1).temp = 25.3;
exp(1).data = rand(100,1);
exp(2).name = 'run_B'; exp(2).temp = 27.1;
exp(2).data = rand(100,1);
exp
```

exp = 1×2 struct

Fields	name	temp	data
1	'run_A'	25.3000	*100×1 double*
2	'run_B'	27.1000	*100×1 double*

```
% Collect field values across array
temps = [exp.temp] % Concatenate all temperatures
```

```
temps = 1×2
    25.3000   27.1000
```

```
max(temps)   % Hottest temperature
```

```
ans = 27.1000
```

```
disp(exp(idx).name)
```

```
run_B
```

Tables — Spreadsheet-Like Data

Tables are for **column-oriented, heterogeneous data** with named variables

```
% Table (Tabular data with mixed types)
Name   = {'Alice'; 'Bob'; 'Carol'};
Age    = [28; 35; 42];
Height = [165; 180; 170];
Smoker = logical([0; 1; 0]);
T = table(Name, Age, Height, Smoker)
```

T = 3×4 table

	Name	Age	Height	Smoker
1	'Alice'	28	165	0
2	'Bob'	35	180	1
3	'Carol'	42	170	0

```
% Access
T.Age       % Column as numeric vector
```

```
ans = 3×1
     28
     35
     42
```

```
T{:, 'Age'}  % Same, explicit
```

```
ans = 3×1
     28
     35
     42
```

```
T(2, :)     % Row as table
```

ans = 1×4 table

	Name	Age	Height	Smoker
1	'Bob'	35	180	1

```
T(T.Age > 30, :)  % Filter rows
```

ans = 2×4 table

	Name	Age	Height	Smoker
1	'Bob'	35	180	1
2	'Carol'	42	170	0

Tables — Spreadsheet-Like Data

```
% Summary and export
summary(T)
```

```
Variables:

    Name: 3×1 cell array of character vectors

    Age: 3×1 double

        Values:

            Min          28
            Median       35
            Max          42

    Height: 3×1 double

        Values:

            Min         165
            Median      170
            Max         180

    Smoker: 3×1 logical

        Values:

            True         1
            False        2
```

```
writetable(T, 'patients.csv');
```

patients.csv

	A	B	C	D
	patients			
	Name	Age	Height	Smoker
	Text	Number	Number	Number
1	Name	Age	Height	Smoker
2	Alice	28	165	0
3	Bob	35	180	1
4	Carol	42	170	0

Table vs. struct:

Table → rectangular, column-oriented, ideal for *observations × variables*

Struct → non-rectangular, flexible nesting, ideal for *configuration/records*

Control flow

- Conditional Statements: if elseif else
- Loop Statements: for/while
- Program Control Keywords

Conditional Statements: if elseif else

Conditional Statements evaluate conditions at runtime to decide which block of code to execute.

`if, elseif, else`

Executes code based on whether a logical condition is true or false. Note that `elseif` is written as a single word with no spaces.

```
score = 85;

if score >= 90
    disp('Grade: A');
elseif score >= 80
    disp('Grade: B'); % This block will execute
else
    disp('Grade: C or below');
end
```

```
Grade: B
```

Conditional Statements: switch, case, otherwise

Executes one group of statements from multiple choices by matching a variable against specific case values. Unlike languages like C, MATLAB `switch` statements do not require a `break` keyword at the end of each case; they execute only the first matching case and then exit.

```
day = 'Tuesday';

switch day
    case 'Monday'
        disp('Start of the work week.');
    case 'Tuesday'
        disp('Second day of the week.'); % This block will execute
    otherwise
        disp('Another day.');
end
```

```
Second day of the week.
```

Loop Control Statements: for and while

Loops allow you to repeatedly execute a block of code

`for`

Repeats a block of code a predetermined number of times, typically using a loop counter variable or traversing elements in a vector/array.

`while`

Repeats a block of code indefinitely as long as a specified logical condition remains true.

```
for i = 1:5
    fprintf('Iteration number: %d\n', i);
end
```

```
Iteration number: 1
Iteration number: 2
Iteration number: 3
Iteration number: 4
Iteration number: 5
```

```
count = 1;
while count <= 3
    disp(count);
    count = count + 1;
end
```

```
     1

     2

     3
```

Efficient Programming: the cost of loops

- MATLAB is optimized for **matrix and vector operations.**
- Loop-based scalar code is often **an order of magnitude slower .**

Example: compute sine of 1001 values

```
% Loop version
i = 0;
for t = 0:0.01:10
    i = i + 1;
    y(i) = sin(t);
end
whos y
```

```
  Name      Size              Bytes  Class     Attributes

  y         1x1001             8008  double
```

```
% Vectorized version
t = 0:0.01:10;
y = sin(t);
disp(y)
```

```
  Columns 1 through 13

         0    0.0100    0.0200    0.0300    0.0400    0.0500    0.0600    0.0699    0.0799    0.0899    0.0998    0.1098    0.1197

  Columns 14 through 26

    0.1296    0.1395    0.1494    0.1593    0.1692    0.1790    0.1889    0.1987    0.2085    0.2182    0.2280    0.2377    0.2474
```

Efficient Programming: Point-Wise Operations

Use .*, ./, .^ when you want **element-by-element** computation.

Example: Areas of 10 circles

```
R=randi([1 100],1,10) % radius in cm
```

```
R = 1×10
     85    79    28    23    33    83    83    58    58    29
```

```
% Loop (most languages)
for i = 1:length(R)
    A(i) = pi * (R(i)^2);
end
disp(A)
```

```
   1.0e+04 *

    2.2698    1.9607    0.2463    0.1662    0.3421    2.1642    2.1642    1.0568    1.0568    0.2642
```

```
% Vectorized
A = pi *(R.^2)
```

```
A = 1×10
10^4 ×
    2.2698    1.9607    0.2463    0.1662    0.3421    2.1642    2.1642    1.0568    1.0568    0.2
```

Logical Indexing and find

- **Logical indexing** selects elements where a condition is true — often faster and clearer than **find**.
- **find** returns linear indices — useful when you need *positions*, not just values

```
Magic = magic(3) % magic square
```

```
Magic = 3×3
     8     1     6
     3     5     7
     4     9     2
```

```
% Negate every element > 4

M=Magic;
% Loop version (slow)
for i = 1:3
    for j = 1:3
        if M(i,j) > 4
            M(i,j) = -M(i,j);
        end
    end
end
M
```

```
M = 3×3
    -8     1    -6
     3    -5    -7
     4    -9     2
```

```
% Logical indexing (fast)
M=Magic;
mask = M > 4;
M(mask) = -M(mask)
```

```
M = 3×3
    -8     1    -6
     3    -5    -7
     4    -9     2
```

```
M=Magic;
% Linear indices
idx = find(M > 4)
```

```
idx = 5×1
     1
     5
     6
     7
     8
```

```
M=Magic;
% Linear indices
idx = find(M > 4)
```

```
idx = 5×1
     1
     5
     6
     7
     8
```

```
% Elements>4
M(idx)
```

```
ans = 5×1
     8
     5
     9
     6
     7
```

```
% Row and column subscripts
[row, col] = find(M > 4)
```

Vectorization Patterns

```
% Pattern 1: Apply operation to subset
x=randi([-10 10],1,10)
```

```
x = 1×10
    -1    -5    -1   -10     2    10    -8   -10     8    -5
```

```
x(x < 0) = 0                % Clamp negatives to zero
```

```
x = 1×10
     0     0     0     0     2    10     0     0     8     0
```

```
% Pattern 2: Compute per-column statistic without loop
A = randi(100, 10, 5);
colMeans = mean(A)          % 1×5 vector
```

```
colMeans = 1×5
   48.3000   50.6000   57.8000   56.4000   51.6000
```

```
deviation = A - colMeans  % Implicit expansion
```

```
deviation = 10×5
   -47.3000   -6.6000   26.2000  -27.4000   30.4000
    33.7000  -36.6000  -52.8000    9.6000  -41.6000
   -33.3000   25.4000   -2.8000   -7.4000   34.4000
    39.7000  -25.6000   37.2000   41.6000  -48.6000
   -38.3000   15.4000  -24.8000   18.6000   38.4000
   -12.3000   35.4000   23.2000    0.6000   38.4000
    11.7000  -41.6000    3.2000  -26.4000    1.4000
    10.7000   47.4000   21.2000  -30.4000  -38.6000
    18.7000  -46.6000   22.2000   32.6000  -33.6000
    16.7000   33.4000  -52.8000  -11.4000   19.4000
```

```
% Pattern 3: Running/cumulative operations
x = 1:6
```

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

```
y = cumsum(x)
```

```
y = 1×6
     1     3     6    10    15    21
```

```
% Every 3rd cumulative sum
result = y(3:3:end)
```

```
result = 1×2
     6    21
```

```
% Pattern 4: cellfun / arrayfun for containers
myCell = {42, 'Temperature Data', [1, 2, 3, 4]}
```

```
myCell = 1×3 cell
```

	1	2	3
1	42	'Temperature Data'	[1,2,3,4]

```
lengths = cellfun(@numel, myCell)
```

```
lengths = 1×3
     1    16     4
```

```
% What is 16? Explain the result
```

From slow to fast

```
v = rand(1e8, 1);
fprintf('Slow:    '); tic; slowSumSq(v, 1); toc
```

```
Slow:    Elapsed time is 0.159320 seconds.
```

```
fprintf('Fast:    '); tic; fastSumSq(v, 1); toc
```

```
Fast:    Elapsed time is 0.065521 seconds.
```

```
function s = slowSumSq(v, thresh)
    s = 0;
    for k = 1:length(v)
        if v(k) > thresh
            s = s + v(k)^2;
        end
    end
end

function s = fastSumSq(v, thresh)
    s = sum(v(v > thresh).^2);
end
```

Scripts and Functions

Feature	Scripts	Functions
Workspace	Share the base workspace (can accidentally overwrite variables).	Maintain their own local workspace (variables clear when finished).
Inputs / Outputs	No formal arguments; rely on existing workspace variables.	Accept explicit inputs and pass back explicit outputs.
Use Case	Data analysis, sequencing tasks, plotting, and quick experiments.	Code reuse, building algorithms, math transformations, and libraries.

- **Use Scripts** for your top-level driver files—the master routine that sets configurations, loads data files, triggers algorithms, and plots final figures.
- **Use Functions** for every sub-step that calculates a formula, filters a signal, or formats data.

How to Write a Function

- First line defines signature:

function [outputs] = name(inputs)

- For the functions in a separate file the file name must match function name (myFunction.m)
- H1 line (first comment line) appears in help and lookfor
- Variables are local — they do not appear in the base workspace
- Use end to close (required for nested functions, recommended always)

In Matlab2024 and above all function definitions can be located either **before** or **after** the function call

In Matlab2022 all function definitions should be located **after** the function call and separated by a section break

```
myFunction(2, 3)
```

```
ans = 5
```

```
a1=myFunction(2, 3)
```

```
a1 = 5
```

```
[a1,a2]=myFunction(2, 3)
```

```
a1 = 5
a2 = 6
```

```
function [out1, out2] = myFunction(in1, in2)
%MYFUNCTION One-line description
%   Detailed description of behavior.
%
%   [out1, out2] = myFunction(in1, in2) computes ...

    % Body
    out1 = in1 + in2;
    out2 = in1 * in2;
end
```

Functions inside the script and function call

```
%Matlab2024
function m=MaxDiag(a)
m=max(diag(a));
end
A=randi(10,3)
max_diag=MaxDiag(A)
```

```
A = 3×3
    10    10     2
     2     5     5
    10     9    10

max_diag = 10
```

```
% Call
[s, d] = algebra(5, 3)   % Only request what you need
```

```
s = 8
d = 2
```

```
function [s, d, p, q] = algebra(a, b)
    s = a + b;
    d = a - b;
    p = a * b;
    q = a / b;
end
```

Function handles to named functions

When to use:

- Short mathematical expressions
- Passing functions to optimizers (fminsearch, fzero, integral)
- Callbacks (UI, timers)

```
h = @sin;
h(pi/2)
```

```
ans = 1
```

```
% Useful for passing behavior as an argument
applyTwice(@(x) x + 3, 5)
```

```
ans = 11
```

```
function result = applyTwice(f, x)
    result = f(f(x));
end
```

Function Handles and Anonymous Functions

- **Function Handles (`@`):** A data type that stores a reference to a function, allowing you to pass functions as arguments to other functions (known as higher-order functions like `integral`, `fzero`, or `ode45`).
- **Anonymous Functions:** Simple inline functions created on the fly without saving code to a separate `.m` file. They capture variables present in the current workspace when created.

```
% 1. Creating a standard function handle
sineHandle = @sin
```

```
sineHandle = function_handle with value:
    @sin
```

```
result = sineHandle(pi/2) % Evaluates to 1
```

```
result = 1
```

```
% 2. Creating an anonymous function (Single-line math expression)
% Syntax: handle = @(inputVars) expression
quadratic = @(x) 2*x^2 + 5*x - 3
```

```
quadratic = function_handle with value:
    @(x)2*x^2+5*x-3
```

```
y_val = quadratic(3) % Evaluates to 30
```

```
y_val = 30
```

- A **function handle** is a variable that *refers to* a function.
- An **anonymous function** is a one-line function defined inline

Variable Number of Arguments: varargin and varargout

In MATLAB, **varargin**, **varargout**, **nargin**, and **nargout** work together to build flexible functions that accept any number of inputs and return any number of outputs.

Flexible Signatures via `varargin` and `varargout`

- `varargin` : A cell array that packages any number of optional input arguments trailing the explicit inputs.
- `varargout` : A cell array that packages dynamic, optional outputs based on how many outputs the caller requests.

- `nargin` **(Number of Arguments In):** Returns the total number of input arguments supplied by the caller during execution.
- `nargout` **(Number of Arguments Out):** Returns the total number of output arguments requested by the caller.

Variable inputs

Write a function that takes an arbitrary number of vectors and returns the average value of the elements across all the vectors.

```
function m=AverageV(varargin)
 fprintf('Number of arguments: %d\n',nargin);
 c=[];
 for k = 1:nargin
      c=[c,varargin{k}];
 end
 m=mean(c);
end
```

```
AverageV(v1,v2,v3)
```

```
Number of arguments: 3
ans = 109
```

```
v1=randi(10,1,10)
```

```
v1 = 1×10
     4    10     9     3    10     1     5     5     5     2
```

```
v2=randi(100,1,8)
```

```
v2 = 1×8
    43    62    99    22    36    27    30    19
```

```
v3=randi(1000,1,5)
```

```
v3 = 1×5
    23   450   244   869   529
```

Variable outputs

```
v=randi([1 100],1,10)
```

```
v = 1×10
    92    98    59    12    93    60    89    43    61     8
```

```
[a, b] = minMaxMedian(v)
```

```
a = 8
b = 98
```

```
[a, b, c] = minMaxMedian(v)
```

```
a = 8
b = 98
c = 60.5000
```

```
function varargout = minMaxMedian(v)
    varargout{1} = min(v);
    varargout{2} = max(v);
    varargout{3} = median(v);
end
```

Variable inputs and outputs

```
function [summary, varargout] = processData(data, varargin)
% PROCESSDATA Handles arbitrary inputs and
% optional statistical outputs.

    % Default parameter configuration
    filterType = 'none';
    if nargin > 1
        % Extract from varargin cell array
        filterType = varargin{1};
    end

    % Core operations
    if strcmp(filterType, 'smooth')
        data = movmean(data, 3);
    end
    summary = mean(data);

    % Dynamic output generation using nargout
    if nargout > 1
        % Second output: Standard Deviation
        varargout{1} = std(data);
    end
    if nargout > 2
        % Third output: Minimum value
        varargout{2} = min(data);
    end
end
```

```
% --- Executive Calls from a Script ---
v=[10, 20, 30]
```

```
v = 1×3
     10    20    30
```

```
% Call 1: Basic execution
s1 = processData(v)
```

```
s1 = 20
```

```
% Call 2: Providing optional inputs
% and requesting optional outputs
[s2, dataStd] = processData(v, 'smooth')
```

```
s2 = 20
dataStd = 5
```

```
% Call 3: Requesting all available outputs
[s3, dataStd, dataMin] = processData(v, 'smooth')
```

```
s3 = 20
dataStd = 5
dataMin = 15
```

Performance Optimization

Preallocation

- When you assign to an array index beyond its current size, MATLAB must:
  - Allocate new, larger memory
  - Copy old data
  - Free old memory

  This repeats **every iteration** if you grow inside a loop.
- **Preallocating** sets the memory space once.

```
% INEFFICIENT: Dynamic resizing
clear array
for i = 1:10000
    array(i) = i^2;
end

% EFFICIENT: Memory preallocated via zeros()
array = zeros(1, 10000);
for i = 1:10000
    array(i) = i^2;
end
```

Profiling

Code Profiling

- Never guess what part of your code is slow.
- Use the **MATLAB Profiler** to measure execution time down to individual lines.

The Profiler shows:

- **Total time** per function
- **Self time** (excluding called functions)
- **Line-by-line** execution time
- **Call count**

```
profile on
myslowScript
profile off
profile viewer
```

Typical workflow:

1. Run profiler on representative input
2. Identify the **hottest function/line**
3. Apply vectorization or preallocation there
4. Re-profile to verify improvement
5. Repeat

Profiling

```
profile on          % Start recording execution metrics
demoProfile         % Run your target function or script
profile viewer      % Stop recording and open the interactive Profiler window
```

```
function demoProfile()
    n = 15000;

    % Task 1: Slow loop without preallocation
    slowResult = runSlowLoop(n);

    % Task 2: Fast vectorized operation
    fastResult = runFastVector(n);
end

function out = runSlowLoop(n)
    out = []; % No preallocation
    for i = 1:n
        out(i) = sin(i) * cos(i); % Resizes array on every iteration
    end
end

function out = runFastVector(n)
    x = 1:n;
    out = sin(x) .* cos(x); % Single vectorized call
end
```

Profiling

```
profile on          % Start recording execution metrics
demoProfile         % Run your target function or script
profile viewer      % Stop recording and open the interactive Profiler window
```

```
function demoProfile()
    n = 15000;

    % Task 1: Slow loop without preallocation
    slowResult = runSlowLoop(n);

    % Task 2: Fast vectorized operation
    fastResult = runFastVector(n);
end

function out = runSlowLoop(n)
    out = []; % No preallocation
    for i = 1:n
        out(i) = sin(i) * cos(i); % Resizes array on every iteration
    end
end

function out = runFastVector(n)
    x = 1:n;
    out = sin(x) .* cos(x); % Single vectorized call
end
```

Optimization Checklist

Technique	When to Apply	Expected Gain
Vectorize	Loops with independent iterations	2–100×
Preallocate	Loops that append to arrays	10–50×
Logical indexing	Subset selection, conditional replacement	5–10× vs. loops
Use built-in functions	Any standard operation	C-optimized, fastest
Avoid `eval` and dynamic field names in hot loops	Only if profiler flags them	Variable
Use `tic` / `toc` or `timeit`	Measure before and after	—