Computing and Software Engineering

GET211

September 23, 2025

Data Types

The basic types used to represent numbers. They can be broadly categorized into two main types: **floating-point** and **integer** types.

1. Floating-Point Types These types are used to represent real numbers, including those with decimal points. They can store very large or very small numbers due to their ability to handle a wide range of magnitudes.

2. Integer Types These types are used to represent whole numbers (without fractions). MATLAB provides different sizes of integers, both **signed** and **unsigned**.

Signed Integer Types Signed integers can represent both positive and negative whole numbers.

Unsigned Integer Types Unsigned integers can only represent non-negative whole numbers (positive numbers and zero).

These primitive numeric types allow MATLAB to efficiently store and process numerical data with varying levels of precision and range. The **default type** for most numerical computations in MATLAB is `double`, but for memory optimization and specific use cases, you can use other numeric types such as `single` or integer types (`int`/`uint`).

Data Type	Description	Size	Range	Example
`double`	Double-precision floating-point (default)	64 bits (8 bytes)	$\pm 1.7 \times 10^{\pm 308}$ (15–17 decimal digits)	`x = 3.14159`
`single`	Single-precision floating-point	32 bits (4 bytes)	$\pm 3.4 \times 10^{\pm 38}$ (6–9 decimal digits)	`y = single(3.14159)`
`int8`	8-bit signed integer	8 bits (1 byte)	-128 to 127	`x = int8(-50)`
`int16`	16-bit signed integer	16 bits (2 bytes)	-32,768 to 32,767	`x = int16(1000)`
`int32`	32-bit signed integer	32 bits (4 bytes)	-2,147,483,648 to 2,147,483,647	`x = int32(100000)`
`int64`	64-bit signed integer	64 bits (8 bytes)	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	`x = int64(10000000000)`
`uint8`	8-bit unsigned integer	8 bits (1 byte)	0 to 255	`x = uint8(255)`
`uint16`	16-bit unsigned integer	16 bits (2 bytes)	0 to 65,535	`x = uint16(60000)`
`uint32`	32-bit unsigned integer	32 bits (4 bytes)	0 to 4,294,967,295	`x = uint32(40000000)`
`uint64`	64-bit unsigned integer	64 bits (8 bytes)	0 to 18,446,744,073,709,551,615	`x = uint64(1000000000000)`

Table 1: MATLAB Primitive Numeric Data Types

Non-numeric representation

Non-numeric data types are used to store and represent information that is not strictly numerical, such as text, logical values, complex data structures, or symbolic expressions.

Text Representation

Character arrays are used to store textual data. In newer versions of MATLAB, strings are represented by the string data type, which is more flexible than character arrays (char).

1. Character Arrays (`char`)

A character array is a sequence of characters, where each character is treated as an element in an array. It is enclosed in single quotes (`' '`) and is the traditional way to handle text data in MATLAB.

Example:

```
charArray = 'Hello, World!';  % 1x13 character array
```

Multiline Character Array Example:

```
names = ['Alice'; 'Bob  '; 'Carol'];  % Character
    array
```

2. String Arrays (`string`)

The `string` data type is more flexible and easier to work with than `char`. String arrays are scalar values, unlike character arrays. They are enclosed in double quotes (" ").

Example:

```
str = "Hello, World!";  % String scalar
```

Multiline String Array Example:

```
greetings = ["Hello", "Hi", "Greetings"];  % 1x3
    string array
```

Both `char` and `string` types are used to represent text in MATLAB. The `string` data type offers more flexibility and better text manipulation features, making it ideal for modern applications, while `char` is still commonly used for compatibility with older code.

Feature	char	string
Syntax	Single quotes (`'text'`)	Double quotes (`"text"`)
Array Representation	2D character arrays	1D/2D string arrays
Padding Requirement	Requires padding for alignment	No padding required
Mutability	Modifiable like arrays	Immutable
Introduced in	Available since early MATLAB	Introduced in R2016b

Table 2: Differences between `char` and `string` in MATLAB

Unicode and Special Characters
MATLAB supports Unicode, allowing characters from different languages and special symbols to be represented using either `char` or `string`.
Unicode Example:

```
smiley = char(9786);  % Unicode character
```

Special Characters Example:

```
specialStr = 'Hello\nWorld!';  % Newline between
    "Hello" and "World!"
```

Operation	`char` Example	`string` Example
Concatenation	`['Hi ' 'there!']`	`"Hi " + "there!"`
Substring	`str(1:5)`	`extractBetween(str, 1, 5)`
Length	`length('Hello')`	`strlength("Hello")`
Uppercase	`upper('abc')`	`upper("abc")`
Lowercase	`lower('ABC')`	`lower("ABC")`
Comparison	`strcmp('Hi', 'hi')`	`"Hi" == "hi"`
Replace	`strrep('MAT', 'A', 'I')`	`replace("MAT", "A", "I")`
Trim	`strtrim(' Hi ')`	`strtrim(" Hi ")`
Convert to `char`	N/A	`char("MATLAB")`
Convert to `string`	`string('MATLAB')`	N/A
Check if Empty	`isempty('')`	`strlength("") == 0`

Table 3: Common Operations on **`char`** and **`string`** in MATLAB

Boolean Algebra

Boolean representation in MATLAB is crucial for logical operations, conditionals, and flow control in programming.

Representation

Boolean values are represented by the `logical` data type, which can take on two values:

- `true` (represented by `1`)
- `false` (represented by `0`)

This data type is particularly useful for performing logical operations and making decisions in code.

Creating Boolean Values

Direct Assignment You can create Boolean values directly using:

```
a = true;   % a is 1
b = false;  % b is 0
```

From Numeric Values Numeric values can be converted to logicals, where `0` becomes `false` and any non-zero value becomes `true`:

```
c = logical(5);  % c is true (1)
d = logical(0);  % d is false (0)
```

The following table summarizes the logical operators available in MATLAB, including their symbols, descriptions, and examples.

Operator	Symbol	Description	Example
AND	`&`	Returns `true` if both operands are `true`.	`result = true & false;`
OR	`|`	Returns `true` if at least one operand is `true`.	`result = true | false;`
NOT	`~`	Inverts the value of the operand.	`result = ~true;`
Exclusive OR	`xor`	Returns `true` if only one operand is `true`.	`result = xor(true, false);`
Equal to	`==`	Checks if two values are equal.	`result = (5 == 5);`
Not equal to	`~=`	Checks if two values are not equal.	`result = (5 ~= 6);`

Table 4: Summary of Logical Operators in MATLAB

Truth Table for Logical Operators

A	B	A & B	A | B	~A	~B	A xor B	A == B
true	true	true	true	false	false	false	true
true	false	false	true	false	true	true	false
false	true	false	true	true	false	true	false
false	false	false	false	true	true	false	true

Computer Operators

1. Arithmetic Operators

Operator	Description	Example	Result
$+$	Addition	$5 + 3$	8
$-$	Subtraction	$5 - 3$	2
$*$	Matrix multiplication	$[1, 2] * [3; 4]$	Matrix product(11)
$.*$	Element-wise multiplication	$[1, 2]. * [3, 4]$	$[3, 8]$
$/$	Matrix right division	$10/2$	5
$./$	Element-wise division	$[10, 20]./2$	$[5, 10]$
$\backslash$	Matrix left division	$8\backslash 4$	Solves $8 * x = 4$
$.\hat{}$	Element-wise power	$[2, 3].^2$	$[4, 9]$
$\hat{}$	Power	2^3	8
`mod`	Modulus (remainder)	`mod(10, 3)`	1

Table 6: Arithmetic Operators in MATLAB

2. Logical Operators

Operator	Description	Example
&	Element-wise AND	`a & b`
|	Element-wise OR	`a | b`
$\sim$	NOT	$\sim$ `a`
xor	Exclusive OR	`xor(a, b)`
&&	Short-circuit AND (scalars only)	`a && b`
||	Short-circuit OR (scalars only)	`a || b`

Table 7: Logical Operators in MATLAB

3. Relational Operators

Operator	Description	Example	Result
==	Equal to	$5 == 5$	`true`
=	Not equal to	$5 = 3$	`true`
<	Less than	$3 < 5$	`true`
<=	Less than or equal to	$3 <= 3$	`true`
>	Greater than	$5 > 3$	`true`
>=	Greater than or equal to	$5 >= 5$	`true`

Table 8: Relational Operators

4. Bitwise Operators

Operator	Description	Example	Result
`bitand`	Bitwise AND	`bitand(6, 3)`	2
`bitor`	Bitwise OR	`bitor(6, 3)`	7
`bitxor`	Bitwise XOR	`bitxor(6, 3)`	5
`bitnot`	Bitwise NOT	`bitnot(6)`	-7
`bitshift`	Bitwise shift	`bitshift(3, 2)`	12

Table 9: Bitwise Operators

5. Assignment Operators

Operator	Description	Example
=	Assignment	`a = 5`
+=	Addition Assignment	`a += 5`
-=	Subtraction Assignment	`a -= 5`
*=	Multiplication Assignment	`a *= 5`
/=	Division Assignment	`a /= 5`
^=	Power Assignment	`a ^= 2`

Table 10: Assignment Operators

6. Special Operators

Operator	Description	Example
:	Colon (range/slicing)	`1:10`
;	Statement Separator	`a = 5; b = 10;`
[]	Array Concatenation	`A = [1, 2, 3]`
’	Transpose	`A’`
.	Element-wise Operator	`a .* b`

Table 11: Special Operators

Special Functions for Mathematical Operations

Function	Description	Example	Result
`sqrt`	Square root	`sqrt(16)`	4
`exp`	Exponential function	`exp(1)`	e
`log`	Natural logarithm	`log(10)`	Natural log of 10
`log10`	Base 10 logarithm	`log10(100)`	2
`abs`	Absolute value	`abs(-10)`	10
`sign`	Signum function	`sign(-5)`	-1
`floor`	Round down	`floor(5.9)`	5
`ceil`	Round up	`ceil(5.1)`	6
`round`	Round to nearest integer	`round(5.5)`	6
`mod`	Modulus	`mod(10, 3)`	1
`rem`	Remainder	`rem(10, 3)`	1

Table 12: Special Functions in MATLAB

Precedence Level	Operator	Description
1	`( )`	Parentheses (for grouping expressions)
2	`.`, `->`	Structure field access
3	`'` `.'`	Transpose Element-wise transpose
4	`^` `.^`	Power Element-wise power
5	`+`, `-` `~`	Unary plus, unary minus Logical NOT

6	`*`, `/`, `\`
	
	`.*`, `./`, `.\`
		Multiplication, right division, left division
		Element-wise multiplication, division

7	`+`, `-`	Addition, subtraction
8	`:`	Colon operator (range creation)
9	`<`, `<=`, `>`, `>=` `==`, `~=`	Relational operators Equality, inequality
10	`&`	Element-wise AND
11	`|`	Element-wise OR
12	`&&`	Short-circuit AND
13	`||`	Short-circuit OR
14	`=` `+=`, `-=`, `*=`, `/=`, `^=`	Assignment Compound assignment operators

Table 13: Operator Precedence in MATLAB

Type Casting in MATLAB

Type casting in MATLAB refers to the process of converting a variable from one data type to another. This is often necessary when performing operations that require specific data types or when working with data from different sources.

Implicit Type Casting

Implicit type casting occurs automatically when MATLAB converts data types without receiving explicit instructions.

MATLAB often performs implicit type casting when necessary. For example, when performing arithmetic operations, MATLAB will automatically convert operands to a common data type. However, it's important to be aware of potential data loss or precision issues that may occur during implicit conversions.

```
1 x = 5 + 3.2;   % Result is double (8.2)
2 y = [1 2] + 3;   % Integer array implicitly cast to
      double
```

Explicit Type Casting

Explicit type casting is when you deliberately convert a value from one data type to another using a specific function or operation.
- Explicit type casting provides more control over the conversion process. - Implicit type casting can simplify code but may introduce unintended side effects. - Be mindful of data type compatibility and potential precision loss during conversions. - Use explicit type casting when necessary to ensure correct data types and avoid unexpected behavior.
MATLAB provides explicit functions for type casting:
- int8(x): Converts to 8-bit signed integer.
- uint8(x): Converts to 8-bit unsigned integer.
- int16(x): Converts to 16-bit signed integer.
- uint16(x): Converts to 16-bit unsigned integer.
- int32(x): Converts to 32-bit signed integer.
- uint32(x): Converts to 32-bit unsigned integer.
- int64(x): Converts to 64-bit signed integer.
- uint64(x): Converts to 64-bit unsigned integer.
- single(x): Converts to single-precision floating-point number.
- double(x): Converts to double-precision floating-point number.
- logical(x): Converts to logical (true/false) value.
- char(x): Converts to character array.
- cell(x): Converts to cell array.
- struct(x): Converts to structure array.

Explicit Type casting

```
% Convert a double to an integer
x = 3.14;
y = int32(x);  % y = 3

% Convert a character array to a double
str = '123';
num = double(str);  % num = [49 50 51] (ASCII values)

% Convert a logical value to a character
flag = true;
charFlag = char(flag);  % charFlag = '1'
```

Exercise

Create four variables of different data types:
- An integer variable intVar with a value of 8
- A double variable doubleVar with a value of 3.14
- A character variable charVar with a value of 'A'
- A string variable strVar with the value "Hello, MATLAB!"

Display each variable along with its data type using the class function.

Exercise

Create the following variables:
- A string variable **name** with the value "Alice"
- A numerical variable **score** with a value of 90
- Combine these variables to display a sentence that says, "Alice scored 90 points."

Exercise

Create a variable y with the value 10.5. Use isnumeric, isinteger, and isfloat to check if it is:
- A numeric type.
- An integer type.
- A floating-point type.

Display the results of each check.

Exercise

Create a double precision variable x with the value 25.6. Then, convert it to:
- An integer type int32.
- A logical type.

Display the results of the conversions.

Exercise

- Create a string variable str1 with the value "MATLAB is great!".
- Find the length of the string using the strlength function.
- Convert the string to uppercase and lowercase using upper and lower.
- Create a character array charArray = 'Data Science'.
- Concatenate charArray with another character array, ' in MATLAB', using strcat.
- Convert str1 to a character array and charArray to a string.

Exercise

Assign the values 15 and 4 to the variables a and b, respectively. Next, perform addition, subtraction, multiplication, division, and modulus operations on a and b, storing each result in a new variable and displaying it.
Following that, use the exponentiation operator ^ to compute a raised to the power of b. Assign the value 3.5 to the variable c and perform the operation of c raised to the power of a using the element-wise power operator .^.

Create two vectors, x = [1, 2, 3] and y = [4, 5, 6]. Perform element-wise addition, subtraction, and multiplication on x and y using the operators +, -, and .*.

Exercise

Assign the values 5, 10, and 3 to the variables x, y, and z, respectively. Evaluate the expression result = x + y * z / 2 - z ˆ2 without using parentheses and rewrite the expression with parentheses to control the order of operations as result = ((x + y) * (z / 2)) - (z ˆ2). Compare both results and explain the difference observed.

Next, assign the values 6, 2, and 8 to the variables a, b, and c. Evaluate the expression $output = a > b \&\& b < c || a == 6$, and modify this expression to use parentheses $output = (a > b) \&\& (b < c) || (a == 6)$, observing how the change affects the result.

Exercise

Assign the value 19 to the variable num and 4 to div. Calculate the remainder when num is divided by div using the modulus operator.. Additionally, perform floor division of num by div using the expression floor(num/div), and compare this result with the standard division num / div.