MATLAB for Beginners - Constants, Variables and Expressions

Connect with me: Youtube | LinkedIn | WhatsApp Channel | Web | Facebook | Twitter

Data Types

Data Types in MATLAB

In MATLAB, data types define the kind of information a variable can store and the operations allowed on them. Choosing the correct data type is crucial for optimizing memory usage, calculation accuracy, and code efficiency. Here's a breakdown of the various data types available:

1. Numeric Data Types:

- **Double-precision floating-point (default):**
  - **Representation:** Uses 64 bits for storage, offering a balance between precision and memory usage.
  - **Type:** double
  - **Range:** Approximately `-1.7e+308` to `1.7e+308`.
  - **Precision:** Generally provides 15-16 decimal digits of precision.
  - **Applications:** Suitable for most general-purpose calculations due to its balanced characteristics.
- **Single-precision floating-point:** - **Representation:** Uses 32 bits for storage, offering faster calculations and less memory usage compared to double-precision. - **Type:** single - **Range:** Approximately `-3.4e+38` to `3.4e+38`. - **Precision:** Provides 6-7 decimal digits of precision. - **Applications:** Suitable when memory efficiency is a concern, and required precision is not as high as double-precision.
- **Integer data types:** - **Storage:** Use a specific number of bits to store whole numbers (integers) without decimal points, offering efficient memory usage and speed for integer operations. - **Types:** - `int8`: Signed, 8-bit integer (-128 to 127) - `uint8`: Unsigned, 8-bit integer (0 to 255) - `int16`: Signed, 16-bit integer (-32768 to 32767) - `uint16`: Unsigned, 16-bit integer (0 to 65535) - `int32`: Signed, 32-bit integer (-2147483648 to 2147483647) - `uint32`: Unsigned, 32-bit integer (0 to 4294967295) - **Applications:** For representing and manipulating whole numbers, particularly when memory or speed is critical.
- **Complex numbers:** - **Representation:** Combine real and imaginary parts, using double-precision floating-point elements for storage. - **Applications:** Used in calculations involving complex mathematical concepts like electrical engineering, signal processing, and wave mechanics.

2. Character and String Data Types: - **Character:** - Type: char - A character array is a sequence of individual characters stored one after another in memory. Each character takes 2 bytes of storage, typically representing a single letter, number, or symbol. - **Use Cases:** - Used for storing short pieces of text where individual characters matter, such as filenames, captions, or labels. - Useful for manipulating individual characters within the array using indexing and string functions. - **String:** - Type: string - A string array is a container that holds text data with additional features and functionalities. Internally, it uses a more complex structure than char arrays, leading to a slightly larger memory footprint. - Used for storing and manipulating textual data like words, sentences, or paragraphs. - **Use Cases:** - Used for storing text data in a more versatile way. - Offers

built-in functions for various string operations like concatenation, searching, and comparison. - Can hold text of any length without requiring padding or pre-allocation.

Example #: Creating char and string

```
% char array
myChar = 'Hello';

% string
myString = "Hello World!";
```

Example #: Accessing elements

```
% Accessing first character of char array (individual character)
firstChar = myChar(1);
```

3. Logical Data Type: * Represents logical values, either `true` or `false`. * Used for conditional statements and Boolean operations. * **Choosing the Right Data Type:**

Consider these factors when selecting a data type:

- **Range of values:** Choose a type that can accommodate the minimum and maximum values you need.
- **Required precision:** Select a type with sufficient precision to handle the level of accuracy needed.
- **Memory constraints:** If memory is limited, consider using efficient data types like integers when appropriate.
- **Functionality:** Choose the type that best suits the intended operations you'll perform on the data.

In MATLAB, there are several ways to check the data type of a variable:

1. `class` function:

This is the most common and recommended method. It returns a string indicating the data type of the variable.

```
variable_name = "Hello";
data_type = class(variable_name);
disp(data_type); % Output: "string"
```

2. `whos` command:

This command provides a detailed list of all currently defined variables in your workspace, including their name, size, class (data type), and a few other attributes.

```
whos
```

The output will show columns like "Name," "Size," and "Class," allowing you to check the data type for specific variables.

3. `isa` function:

This function allows you to check if a variable belongs to a specific data type. It returns `1` if the variable is of the specified type and `0` otherwise.

```
number = 10;
is_integer = isa(number, 'double');
disp(is_integer); % Output: 1
```

4. Built-in functions for specific types:

Certain data types have dedicated functions that return information about them. For example, `ischar` checks for characters, `isfloat` checks for floating-point numbers, and `islogical` checks for logical values.

```
myChar = 'This is a string';
is_char = ischar(myChar);
disp(is_char); % Output: 1
```

Choosing the best method:

- `class` **function** is the simplest and most versatile option for quickly determining the data type of a variable.
- `whos` **command** is useful when you need to see a complete list of variables and their attributes, not just the data type.
- `isa` **function** is beneficial for checking if a variable belongs to a specific type, often used in conditional statements.
- **Specific type functions** are helpful when working with particular data types and need additional information beyond just the basic class.

Remember, choosing the appropriate method depends on your specific needs and the context of your code.

Type Conversion

Some common MATLAB functions used to change data types:

1. `cast`: This is the most versatile function for converting data between various numeric types. It takes two arguments:

- **The data to be converted:** This can be any variable or expression that evaluates to a numerical value.
- **The desired data type:** Specify the new data type using a string like 'double', 'int32', 'single', etc.

```
% Convert a double to an integer
convertedInt = cast(12.34, 'int32');
```

```
% Convert a string to a double
convertedDouble = cast('3.14', 'double');
```

Additional points:

- These functions might not always be successful, especially when converting between incompatible data types.
- Always check the documentation for each function to understand its limitations and potential errors.
- Consider using `is*` functions like `isnumeric` or `ischar` to check the data type before conversion to avoid errors.

constants and Variables

Operators

Arithmetic Operators

Addition (+)

```
x = 5;
y = 10;
z = x + y;
disp(z); % Output: 15
```

Subtraction (-)

```
x = 10;
y = 5;
z = x - y;
disp(z); % Output: 5
```

Multiplication (*)

```
x = 2;
y = 3;
z = x * y;
disp(z); % Output: 6
```

Division (/)

```
x = 6;
y = 3;
```

```
z = x / y;
disp(z); % Output: 2
```

Exponentiation (^)

```
x = 2;
y = 3;
z = x ^ y;
disp(z); % Output: 8
```

Modulo (%)

```
x = 7;
y = 3;
z = mod(x,y);
disp(z); % Output: 1
```

Example #2.1 from book [2] **Example #2.2 from book** [2] **Example #2.3 from book** [2]

Relational Operators

Relational Operators in MATLAB (Beginner Level)

In MATLAB, relational operators help you compare values and create logical expressions that evaluate to either `true` (1) or `false` (0). These logical expressions are crucial for making decisions and controlling program flow in your code.

Here's a table summarizing the most common relational operators:

Operator	Description	Example
`==`	Equal to	`a == 5` checks if `a` is equal to 5
`~=`	Not equal to	`b ~= 3` checks if `b` is not equal to 3
`<`	Less than	`c < 10` checks if `c` is less than 10
`>`	Greater than	`d > 2` checks if `d` is greater than 2
`<=`	Less than or equal to	`e <= 0` checks if `e` is less than or equal to 0
`>=`	Greater than or equal to	`f >= 7.5` checks if `f` is greater than or equal to 7.5

Using Relational Operators in Expressions

You can combine relational operators with numerical values and variables to create logical expressions. Here are some examples:

```
age = 20;
isAdult = age >= 18;  % Checks if age is 18 or older (true)

grade = 85;
passedExam = grade > 70;  % Checks if grade is greater than 70 (true)

accountBalance = 100;
needsRefill = accountBalance < 50;  % Checks if balance is below 50 (false)
```

Example: String Comparisons

```
name = "Ali";
isFirstName = name == "Ali";  % Checks if name is exactly "Alice" (true)

fruit = "apple";
isFavorite = fruit ~= "banana";  % Checks if fruit is not "banana" (true)
```

Important: String comparisons in MATLAB are case-sensitive. "Ali" is not the same as "ali".

Logical Operators (and, or, not)

MATLAB provides additional operators to combine logical expressions:

- & (AND): Both conditions must be true for the overall expression to be true.
- | (OR): At least one condition must be true for the overall expression to be true.
- ~ (NOT): Inverts the truth value of the expression.

Here's an example:

```
temperature = 30;
isHot = temperature > 25;
isSunny = true;

goSwimming = isHot & isSunny;  % Only true if both hot and sunny (false)

goForWalk = isHot | ~isSunny;  % True if hot or not sunny (true)
```

Example : Combining Relational Operators

```
age = 16;
isTeenager = age >= 13 & age <= 19;  % Checks if age is between 13 and 19 (true)

accountBalance = 40;
needsRefill = accountBalance < 50 | ~isAdult;  % Needs refill if below 50 OR not
adult (true)
```

Applications of Relational Operators

Relational operators are fundamental for various tasks in MATLAB:

- **Conditional Statements:** Use `if`, `else if`, and `end` statements with logical expressions to control program flow based on conditions.
- **Loops:** Utilize `while` and `for` loops with logical expressions to repeat code as long as a condition remains true.
- **Data Filtering:** Employ logical expressions to select specific data points from matrices or arrays that meet certain criteria.

Remember:

- Always enclose variable names and numerical values in single quotes when using them in strings.
- Use parentheses to group complex logical expressions for proper evaluation.

Example: ``` Calculates the area of a rectangle

```
% Define the dimensions of the rectangle
width = 5;
height = 7;

% Calculate the area of the rectangle
area = width * height;

% Display the result
disp('The area of the rectangle is:');
disp(area);
```

Example: Calculate the area of circle with a radius of 5 cm

```
% Define the radius of the circle
r = 5;

% Calculate the area of the circle
area = pi * r^2;

% Display the result
fprintf('The area of the circle with radius %.2f is %.2f.\n', r, area);
```

In this example, we define the radius of the circle as `r=5`. Then, we use the formula for the area of a circle, which is `pi * r^2`, to calculate the area. The `pi` function is a built-in MATLAB function that returns the value of pi (approximately 3.1416). Finally, we use the `fprintf` function to display the result, which is the area of the circle with two decimal places.

You can adjust the value of `r` to calculate the area of a circle with a different radius.

Example: Calculate the circumference of the rectangle

```
% Define the width and height of the rectangle
w = 5;
h = 10;

% Calculate the circumference of the rectangle
circumference = 2 * (w + h);

% Display the result
disp(circumference)
```

In this example, we define the width and height of the rectangle as `w=5` and `h=10`, respectively. Then, we use the formula for the circumference of a rectangle, which is `2 * (width + height)`, to calculate the circumference. Finally, we use the `disp` function to display the result, which is the circumference of the rectangle with two decimal places.

You can adjust the values of `w` and `h` to calculate the circumference of a rectangle with different dimensions.

Example: Calculate the area of the triangle

```
% Define the base and height of the triangle
b = 6;
h = 4;

% Calculate the area of the triangle
area = 0.5 * b * h;

% Display the result
disp(area)
```

In this example, we define the base and height of the triangle as `b=6` and `h=4`, respectively. Then, we use the formula for the area of a triangle, which is `0.5 * base * height`, to calculate the area. Finally, we use the `disp` function to display the result, which is the area of the triangle with two decimal places.

You can adjust the values of `b` and `h` to calculate the area of a triangle with different dimensions.

Example: Calculate the semiperimeter of a spherical triangle

```
% Define the three sides of the spherical triangle
a = pi/6;  % in radians
b = pi/4;  % in radians
c = pi/3;  % in radians

% Calculate the semiperimeter of the spherical triangle
s = (a + b + c)/2;

% Display the result
```

```
fprintf('The semiperimeter of the spherical triangle with sides %.2f, %.2f, and
%.2f is %.2f.\n', a, b, c, s);
```

In this example, we define the three sides of the spherical triangle as `a=pi/6`, `b=pi/4`, and `c=pi/3`, which are angles measured in radians. Then, we use the semiperimeter formula, which is `s = (a + b + c)/2`, to calculate the semiperimeter of the spherical triangle.

Finally, we use the `disp` function to display the result, which is the semiperimeter of the spherical triangle with two decimal places.

You can adjust the values of `a`, `b`, and `c` to calculate the semiperimeter of a different spherical triangle.

Example: Calculate the area of a triangle using Heron's formula:

```
% Define the lengths of the sides of the triangle
a = 5;
b = 6;
c = 7;

% Calculate the semiperimeter of the triangle
s = (a + b + c)/2;

% Calculate the area of the triangle using Heron's formula
A = sqrt(s * (s - a) * (s - b) * (s - c));

% Display the result
disp(A)
```

In this example, we define the lengths of the sides of the triangle as `a=5`, `b=6`, and `c=7`. Then, we use the semiperimeter formula, which is `s = (a + b + c)/2`, to calculate the semiperimeter of the triangle. Finally, we use Heron's formula, which is `A = sqrt(s * (s - a) * (s - b) * (s - c))`, to calculate the area of the triangle.

The `sqrt` function is the square root function in MATLAB. The `fprintf` function is used to display the result, which is the area of the triangle with two decimal places.

You can adjust the values of `a`, `b`, and `c` to calculate the area of a different triangle using Heron's formula.

Example: Finding the Roots of a Quadratic Equation

```
% Define the coefficients a, b, and c of the quadratic equation ax^2 + bx + c = 0
a = 1;
b = 4;
c = 3;

% Calculate the roots of the quadratic equation using the quadratic formula
x1 = (-b + sqrt(b^2 - 4*a*c)) / (2*a);
x2 = (-b - sqrt(b^2 - 4*a*c)) / (2*a);
```

```
% Display the roots
disp(x1);
disp(x2);
```

factorial

f = factorial(n) returns the product of all positive integers less than or equal to n, where n is a nonnegative integer value. If n is an array, then f contains the factorial of each value of n. The data type and size of f is the same as that of n.

The factorial of n is commonly written in math notation using the exclamation point character as n!. Note that n! is not a valid MATLAB® syntax for calculating the factorial of n. [^1]

True/False (Mark T for True and F for False)

Multiple Choice (Select the best answer)

Exercises

Review Questions

References and Bibliography

[1] Raj Kumar Bansal, A. K. Goel, and Manoj Kumar Sharma, MATLAB and its applications in engineering : [based iôn MATLAB 7.5 (R2007b)]. Delhi: Pearson, 2012.

- [^1]: Factorial of input - MATLAB factorial - MathWorks