Basic Syntax

Plotting

1. Creating Simple Plots:

- Line Plot: plot(x, y) or plot(x1, y1, x2, y2, ...)
- Scatter Plot: scatter(x, y)

2. Adding Titles and Axis Labels:

- Title: title('Plot Title')
- X-Axis Label: xlabel('X-Axis Label')
- Y-Axis Label: ylabel('Y-Axis Label')

3. Annotations:

- Text Annotation: text(x, y, 'Text')
- Arrow Annotation: annotation('arrow', [x1, x2], [y1, y2])

4. Multiple Data Sets in One Plot:

- "hold on" to enable multiple plots on the same axes.

```
plot(x1, y1)
hold on
plot(x2, y2)
legend('Data 1', 'Data 2')
```

5. Specifying Line Styles and Colors:

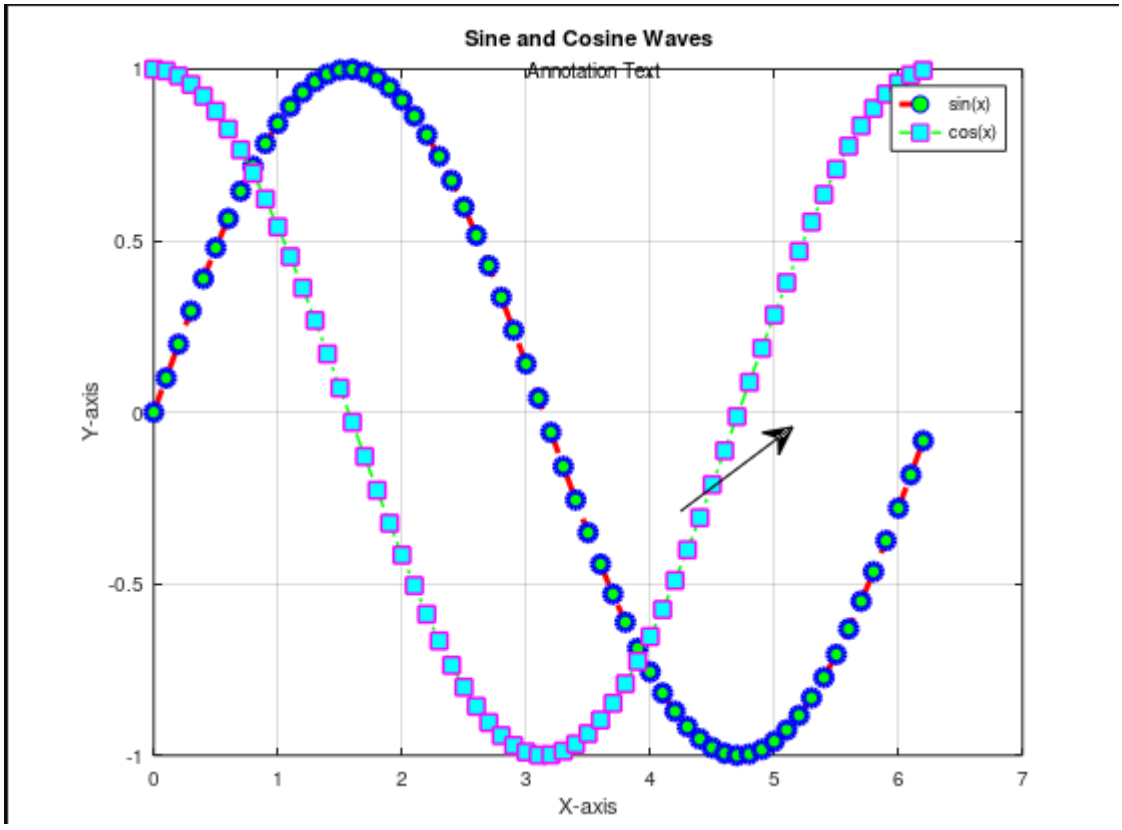
- Line Style and Color in **plot**:
  - **'r-'** for red solid line
  - **'g--'** for green dashed line
- Marker Style and Color in **plot**:
  - **'ro'** for red circles
  - **'bs'** for blue squares

```
plot(x, y, 'r--', 'LineWidth', 2, 'Marker', 'o', 'MarkerEdgeColor', 'b', 'MarkerFaceColor', 'g')
```

Complete Example:

```
x = 0:0.1:2*pi;
y1 = sin(x);
y2 = cos(x);
plot(x, y1, 'r--', 'LineWidth', 2, 'Marker', 'o', 'MarkerEdgeColor', 'b', 'MarkerFaceColor', 'g')
hold on
plot(x, y2, 'g-.', 'LineWidth', 1, 'Marker', 's', 'MarkerEdgeColor', 'm', 'MarkerFaceColor', 'c')
title('Sine and Cosine Waves')
xlabel('X-axis')
ylabel('Y-axis')
legend('sin(x)', 'cos(x)')
text(3, 1, 'Annotation Text')
annotation('arrow', [0.6, 0.7], [0.4, 0.5])
grid on
```

Output:



Functions

```
function [output1, output2, ...] = functionName(input1, input2, ...)
    % Function description and comments
    % Function code here
    % Assign values to output variables
end
```

Example:

```
function [sum_result, product_result] = sum_and_product(a, b)
 sum_result = a + b;
product_result = a * b;
end
```

If Else Conditionals

```
if condition
    % Code to execute when the condition is true
else
    % Code to execute when the condition is false
end
```

Example

```
x = 10;
if x > 5
    disp('x is greater than 5');
else
    disp('x is not greater than 5');
end
```

Relational and Logical Operators:

- Relational Operators: >, <, >=, <=, ==, ~= (greater than, less than, etc.)
- Logical Operators: && (AND), || (OR), ~ (NOT)

Example:

```
a = 10;
b = 7;
if (a > 5) && (b < 8)
   disp('Both conditions are true');
elseif (a < 5) || (b > 8)
   disp('At least one condition is true');
end
```

For Loop

```
for index = start:step:end
   % Loop code using the 'index' variable
end
```

Example:

```
for i = 1:5
   disp(['Iteration ', num2str(i)]);
end
```

While Loop

```
while condition
   % Loop code
End
```

Example

```
n = 1;
while n <= 3
   disp(['Value of n: ', num2str(n)]);
   n = n + 1;
end
```

Switch Case

```
switch expression
   case caseValue1
      % Code to execute when expression matches caseValue1
   case caseValue2
      % Code to execute when expression matches caseValue2
   % Additional case blocks
   otherwise
      % Code to execute if none of the cases match
end
```

Example:

```
day = 'Wednesday';
```

```
switch day
    case 'Monday'
        disp('Start of the workweek.');
    case 'Wednesday'
        disp('Midweek day.');
    otherwise
        disp('Weekend or unknown day.');
end
```

Try Catch

```
try
    % Code that may generate an error
catch exception
    % Code to handle the exception
End
```

Example

```
try
    a = 10;
    b = 0;
    result = a / b;  % This operation will throw a division by zero error.
catch exception
    disp('An error occurred:');
    disp(exception.message);
end
```

Saving Output to a File

```
fid = fopen('output.txt', 'w'); % Open or create a file for writing
fprintf(fid, 'Hello, world!\n'); % Write data to the file
fclose(fid); % Close the file
```

Q. M-File Scripts in MATLAB:

1. **Definition:** M-File scripts are plain text files containing a sequence of MATLAB commands.
2. **Execution:** Scripts run in the order they are written, and you can execute them all at once.
3. **No Function:** Unlike functions, scripts don't accept input arguments or return values.
4. **Workspace:** Variables created in a script are stored in the MATLAB workspace.
5. **Script Side-Effects:** Be cautious of unintended side-effects on variables in the workspace.
6. **Order Matters:** Execution order affects variable values, so be mindful of variable dependencies.
7. **Clear Workspace:** Clear variables if necessary to avoid conflicts between scripts.
8. **Use Functions:** For reusable code, consider using MATLAB functions instead of scripts.
9. **Debugging:** Use debugging tools to identify and fix issues in your scripts.
10. **Documentation:** Add comments and documentation to make scripts more understandable.

M-File Function in MATLAB/Anatomy of A function:

```
function [output1, output2, ...] = functionName(input1, input2, ...)
    % Function description and comments
    % Function code here
    % Assign values to output variables
End
```

1. **function Keyword:** The function definition starts with the **function** keyword.

2. **Output Arguments:** Enclosed in square brackets, define the variables that the function will return.
3. **Function Name:** Specify the function name immediately after the **function** keyword.
4. **Input Arguments:** Enclosed in parentheses, list the variables the function will accept as inputs.
5. **Function Description:** Add comments and documentation to describe the function's purpose and usage.
6. **Function Code:** Write the actual code that performs the desired computations or operations.
7. **Assign Outputs:** Assign values to the output variables inside the function.
8. **End:** Conclude the function with the **end** keyword.

A MATLAB M-File function is a program that takes input arguments and produces output arguments. Here's an example:

```
function [sum_result, product_result] = sum_and_product(a, b)
 sum_result = a + b;
product_result = a * b;
end
```

In this example:

- The function is named **sum_and_product**.
- It takes two input arguments, **a** and **b**.
- It calculates the sum and product of **a** and **b**.
- The results are returned as output arguments, **sum_result** and **product_result**.

You can call this function from the MATLAB Command Window like this:

```
[x, y] = sum_and_product(3, 5);
disp(['Sum: ' num2str(x)]);
disp(['Product: ' num2str(y)]);
```

This will display:

```
Sum: 8
Product: 15
```

In this example, the function accepts inputs, performs calculations, and provides results as output.

MATLAB Debugging:

1. **Use the MATLAB Editor:** Write your code in the MATLAB Editor, as it provides features like syntax highlighting and code analysis that can help catch errors early.
2. **Set Breakpoints:** Place breakpoints in your code at locations where you suspect problems. You can do this by clicking in the left margin of the Editor or using the **dbstop** command.
3. **Run in Debug Mode:** Execute your code in debug mode by clicking the "Run" button with the "Debug" option enabled or by typing **dbstop if error** in the Command Window.
4. **Step Through Code:** Use the debugging toolbar or the **dbstep**, **dbcont**, and **dbquit** commands to step through your code line by line.
5. **Inspect Variables:** Use the "Variables" tab in the Debugging pane to inspect the values of variables. You can also use the **disp**, **fprintf**, or **disp()** functions to print variable values to the Command Window.
6. **Use Breakpoints and Watchpoints:** Set breakpoints, conditional breakpoints, and watchpoints to monitor and control program flow.
7. **Examine Errors:** If an error occurs, read the error message in the Command Window. It provides information about what went wrong and where the issue is located.
8. **Modify Code:** Once you identify the problem, modify your code accordingly. Make sure to save your changes in the Editor.
9. **Re-Run and Test:** After making changes, re-run your code in debug mode to verify that the issue has been fixed.
10. **Documentation and Help:** If you're unsure about functions or syntax, consult MATLAB's documentation and use the **help** command.

Q. Setting Breakpoints

In MATLAB, setting breakpoints and running code with breakpoints is a crucial part of the debugging process. Here's how to set breakpoints and run your code with them:

1. **Setting Breakpoints:**
    - Open your MATLAB script or function in the MATLAB Editor.
    - Identify the line in your code where you want to set a breakpoint.

- Click in the left margin of the Editor, just to the left of the line number, to set a breakpoint. A red dot will appear, indicating the breakpoint.

Alternatively, you can set breakpoints programmatically using the **dbstop** command. For example, to set a breakpoint at line 10 in your script, you can use:

dbstop in yourScriptName at 10

Running with Breakpoints:

- With breakpoints set, you can run your code in debug mode. There are a few ways to do this:

a. In the MATLAB Editor, click the "Run" button, and select the "Debug" option.

b. In the Command Window, type **dbstop if error** to set breakpoints if an error occurs and then run your script.

c. Use the **dbquit** command to start debugging from the beginning of your script or function.

During Debugging:

- When you run the code in debug mode, MATLAB will stop at each breakpoint.
- You can use the debugging toolbar at the top of the MATLAB Editor to control the flow of execution. It provides options like "Step In," "Step Out," "Continue," and more.
- Inspect variable values in the "Variables" tab in the Debugging pane to monitor how they change during execution

Continue or Stop Debugging:

- To continue running your code without stopping at breakpoints, use the "Continue" button on the debugging toolbar or type **dbcont** in the Command Window.
- To stop debugging, use the "Quit Debugging" button in the MATLAB Editor or type **dbquit** in the Command Window.

Q. Examining values, correcting and ending debugging, and correcting an M-file in MATLAB

1. Examining Values During Debugging:

- While debugging, you can inspect variable values at each breakpoint using the "Variables" tab in the Debugging pane.
- Simply click on a variable to see its current value.
- You can also use the **disp**, **fprintf**, or **disp()** functions to print variable values to the Command Window.

2. Correcting Issues During Debugging:

- If you identify an issue while examining variable values, you can make corrections directly in the MATLAB Editor.
- Modify the code where you believe the problem lies.
- Be sure to save your changes in the Editor after making corrections.

3. Continuing Debugging:

- To continue debugging after making corrections, use the "Continue" button on the debugging toolbar or type **dbcont** in the Command Window.
- MATLAB will execute your code from the current breakpoint.

4. Ending Debugging:

- When you've finished debugging, you can end the debugging session.
- Use the "Quit Debugging" button in the MATLAB Editor's debugging toolbar or type **dbquit** in the Command Window.

5. Correcting an M-File:

- To correct an M-File (a script or a function), open it in the MATLAB Editor.
- Locate the section of code that needs correction based on your debugging findings.
- Make the necessary changes to the code.
- Save the M-File in the Editor.