Different Types of 2D Plots:

3D Plotting Techniques 📊

Different Types of 3D Plots:

Advanced Examples:

Week 6: Advanced Function Files and Plotting 📊

Objectives:

- Learn advanced function file techniques 🔧.
- Understand and implement debugging practices 🐛.
- Create and customize 2D and 3D plots in MATLAB 🎨.

Topics Covered:

- Advanced Function Files 📂
- Checking for Optional Parameters 📝
- Subfunctions 🧩
- Debugging and Error Handling 🐞
- 2D Plotting Techniques 📉
- 3D Plotting Techniques 📊

Advanced Function Files 📂

User-Defined Functions and Scripts:

- **Functions vs. Commands**: Functions are separate from the command workspace, making them modular and reusable.
- **Function Syntax**:

```
function [output1, output2] = functionName(input1, input2)
    % Function body
    output1 = ...;
    output2 = ...;
end
```

💡 **Tip**: Functions are the building blocks of modular code. Keep them focused on specific tasks for better reusability.

Function Examples:

Simple Function:

```
function volume = volumeSphere(radius)
    volume = (4/3) * pi * (radius^3);
end
```

Function with Multiple Inputs:

```
function x = displacement(xo, vo, a, t)
    x = xo + vo.*t + (1/2).*a.*t.^2;
end
```

Function with Multiple Outputs:

```
function [a, f] = particleAcceleration(v2, v1, t2, t1, m)
    a = (v2 - v1) / (t2 - t1);
    f = m .* a;
end
```

Notes on Functions:

- **Naming Convention**: Use camelCase or underscores for readability.
- **Element-wise Operations**: Ensure correct use of element-wise operations (`.*`, `./`, `.^`) when working with arrays.
- **Comments and Documentation**: Essential for understanding and maintaining code. Always document what your function does and how to use it.

Checking for Optional Parameters 

Handling Optional Arguments:

- **Using `nargin`**: Check the number of input arguments.
- **Assign Default Values**: If an optional parameter is not provided, assign a default value.

Example:

```
function result = computeArea(radius, shape)
    if nargin < 2
        shape = 'circle'; % Default shape
    end
    switch shape
        case 'circle'
            result = pi * radius^2;
        case 'sphere'
            result = 4 * pi * radius^2;
        otherwise
            error('Unknown shape');
    end
end
```

💡 **Tip**: Handling optional parameters makes your functions more flexible and user-friendly.

Using `varargin` and `varargout`:

- **`varargin`**: Allows functions to accept any number of input arguments.
- **`varargout`**: Allows functions to return any number of output arguments.

Example:

```
function varargout = flexibleFunction(varargin)
    for k = 1:nargin
        varargout{k} = varargin{k}^2;
    end
end
```

Subfunctions 🧩

Defining Subfunctions:

- **Subfunctions** are additional functions defined in the same file as the main function.
- They are only accessible within the file they are defined.

Example:

```
function mainResult = mainFunction(input)
    % Main function code
    helperResult = helperFunction(input);
    mainResult = helperResult + 10;
end

function output = helperFunction(input)
    % Subfunction code
    output = input * 2;
end
```

💡 **Tip**: Use subfunctions to break down complex tasks and keep related functions together.

Benefits of Subfunctions:

- **Encapsulation**: Keeps helper functions hidden from the outside scope.
- **Organization**: Helps in organizing code logically within a single file.
- **Maintainability**: Easier to manage and debug related functions.

Persistent and Global Variables

Persistent Variables:

- Retain their value between function calls.
- Useful for maintaining state information.

Example:

```
function total = runningTotal(newValue)
    persistent sumTotal
    if isempty(sumTotal)
        sumTotal = 0;
    end
    sumTotal = sumTotal + newValue;
    total = sumTotal;
end
```

Global Variables:

- Are accessible from the command window and other functions.
- Should be used sparingly due to potential side effects.

Example:

```
function h = falling(t)
    global GRAVITY
    h = 0.5 * GRAVITY * t.^2;
end

% In the command window or another script:
global GRAVITY
GRAVITY = 9.81;
```

⚠ **Gotcha**: Use global variables sparingly. They can make code harder to debug and maintain because they introduce dependencies across different functions.

Debugging and Error Handling

Types of Errors:

- **Syntax Errors**: Detected during script execution, often highlighted in the editor.
- **Runtime Errors**: Occur during program execution, often due to unexpected input or operations.
- **Logic Errors**: Incorrect program logic that results in incorrect outputs, even though the code runs without errors.

Debugging Techniques:

- **Breakpoints**: Set breakpoints to pause execution and inspect variables.

```
dbstop if error
```

- **Try/Catch Blocks**: Handle errors gracefully.

```
try
    % Statements that may cause an error
catch exception
    % Statements to execute in case of an error
    disp('An error occurred.');
    disp(exception.message);
end
```

Example:

```
try
    result = 10 / 0;
catch exception
    disp('Error: Division by zero');
    disp(exception.message);
end
```

💡 **Tip**: Use the MATLAB debugger to step through your code line by line.

2D Plotting Techniques 📉

Basic 2D Plot:

- **Plot Command**:

```
x = 0:0.1:10;
y = sin(x);
plot(x, y);
title('Sine Wave');
xlabel('x');
ylabel('sin(x)');
grid on;
```

💡 **Tip**: Always label your axes and add a title to your plots for clarity.

Multiple Curves:

- **Using Hold**:

```
plot(x, y1);
hold on;
plot(x, y2);
hold off;
```

- **Multiple Plot Command**:

```
plot(x, y1, x, y2);
```

Creating Plot Grids with Subplots 

Using Subplots to Display Multiple Plots in One Figure:

- **Basic Syntax:**

```
subplot(m, n, p);
```

  - `m`: Number of rows
  - `n`: Number of columns
  - `p`: Position of the current plot

Grid of Plots with `subplot`:

Example Code:

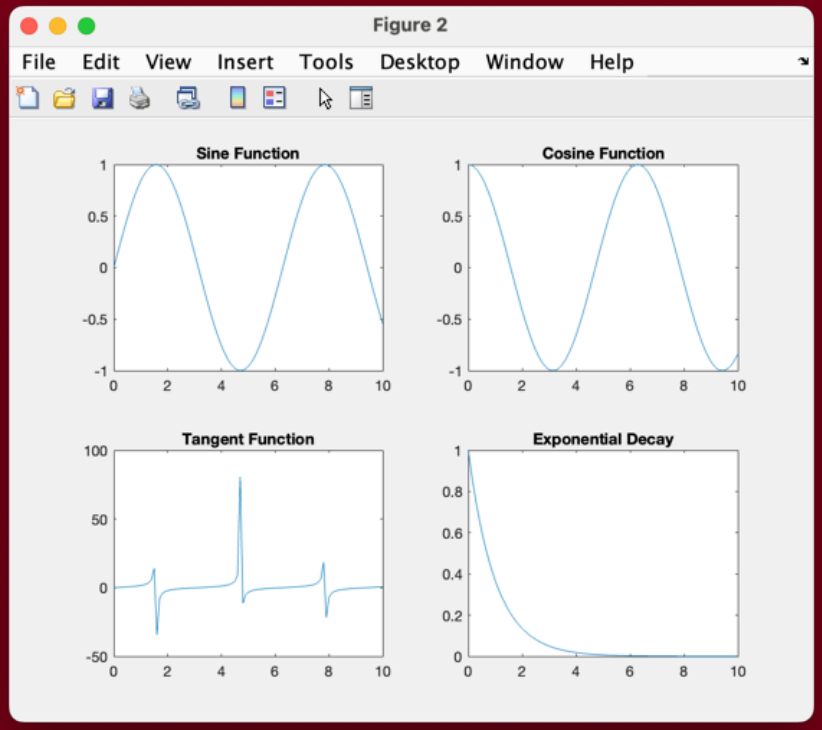
```
x = 0:0.1:10;
y1 = sin(x);
y2 = cos(x);
y3 = tan(x);
y4 = exp(-x);

figure;
subplot(2, 2, 1);
plot(x, y1);
title('Sine Function');

subplot(2, 2, 2);
plot(x, y2);
title('Cosine Function');

subplot(2, 2, 3);
plot(x, y3);
title('Tangent Function');

subplot(2, 2, 4);
plot(x, y4);
title('Exponential Decay');
```



Explanation: This code creates a 2x2 grid of plots in a single figure window.

Adjusting Layout:

- **Spacing and Margins:**

```
subplot('Position', [left, bottom, width, height]);
```

- **Using `tiledlayout` for More Control:**

```
tiledlayout(2,2);
nexttile;
plot(x, y1);
% Repeat for other plots
```

💡 **Tip:** Use `tiledlayout` for more advanced layout options in newer MATLAB versions.

Customizing Plots:

- **Line Style, Color, and Marker**:

```
plot(x, y, 'r-.*');
```

- **Axis Limits**:

```
axis([xmin, xmax, ymin, ymax]);
```

- **Annotations**:

```
text(2, 0.9, 'Annotated Text');
gtext('Place me anywhere');
```

- **Legends**:

```
legend('Curve 1', 'Curve 2');
```

💡 **Tip**: Use legends to distinguish between multiple data sets on the same plot. This makes your plot more informative.

Managing Figure Windows 🖼️

Creating Multiple Figures:

- **Opening New Figure Windows:**

```
figure(1); % Opens or switches to figure window 1
plot(x1, y1);
title('Figure 1');

figure(2); % Opens or switches to figure window 2
plot(x2, y2);
title('Figure 2');
```

- **Explanation:** Using `figure(n)` allows you to create and manage multiple figure windows.

Closing Figure Windows:

- **Close Specific Figure:**

```
close(1); % Closes figure window 1
```

- **Close All Figures:**

```
close all; % Closes all open figure windows
```

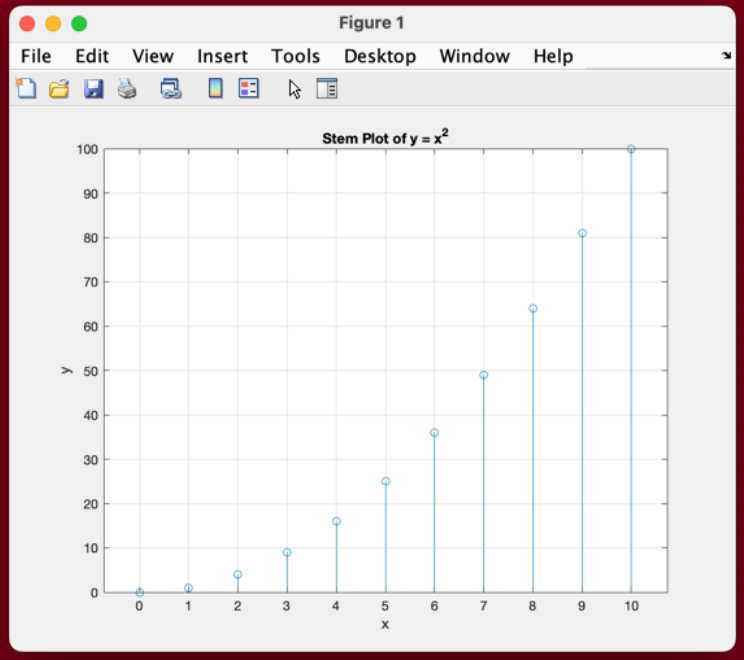
- 💡 **Tip:** It's good practice to use `close all` at the beginning of scripts to ensure no old figures interfere with new plots.

Stem Plots:

Usage: For discrete data points.

Example:

```
x = 0:10;
y = x.^2;
stem(x, y);
title('Stem Plot of y = x^2');
xlabel('x');
ylabel('y');
grid on;
```



Description: Displays data as lines extending from a baseline with markers at the data values.

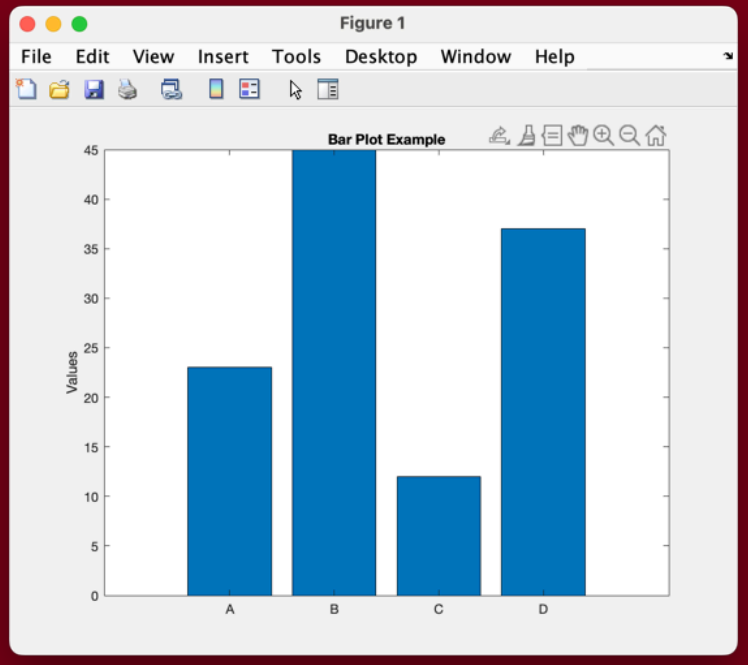
Bar Plots:

Usage: Comparing quantities across categories.

Example:

```
categories = {'A', 'B', 'C', 'D'};
values = [23, 45, 12, 37];
bar(values);
set(gca, 'XTickLabel', categories);
title('Bar Plot Example');
ylabel('Values');
```

Description: Represents data with rectangular bars proportional to the values they represent.

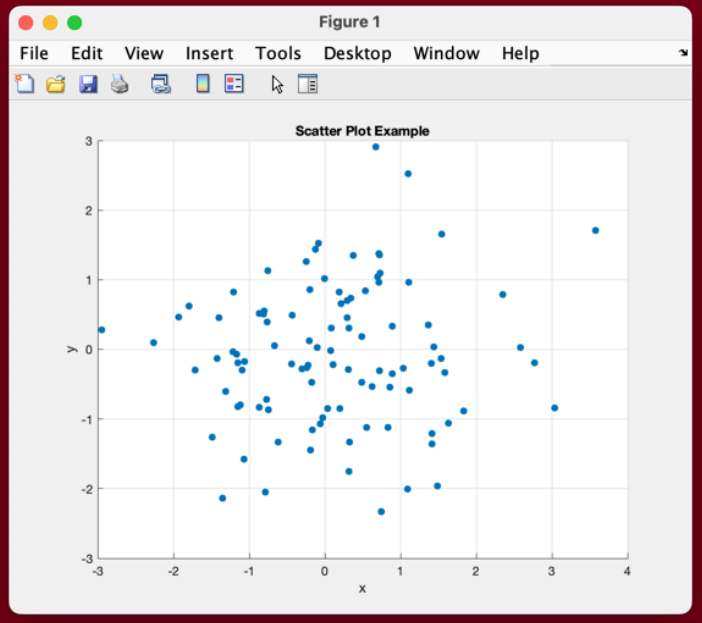


Scatter Plots:

Usage: Displaying relationships between two variables.

Example:

```
x = randn(100,1);
y = randn(100,1);
scatter(x, y, 'filled');
title('Scatter Plot Example');
xlabel('x');
ylabel('y');
grid on;
```



Description: Shows individual data points to highlight the distribution and relationship between variables.

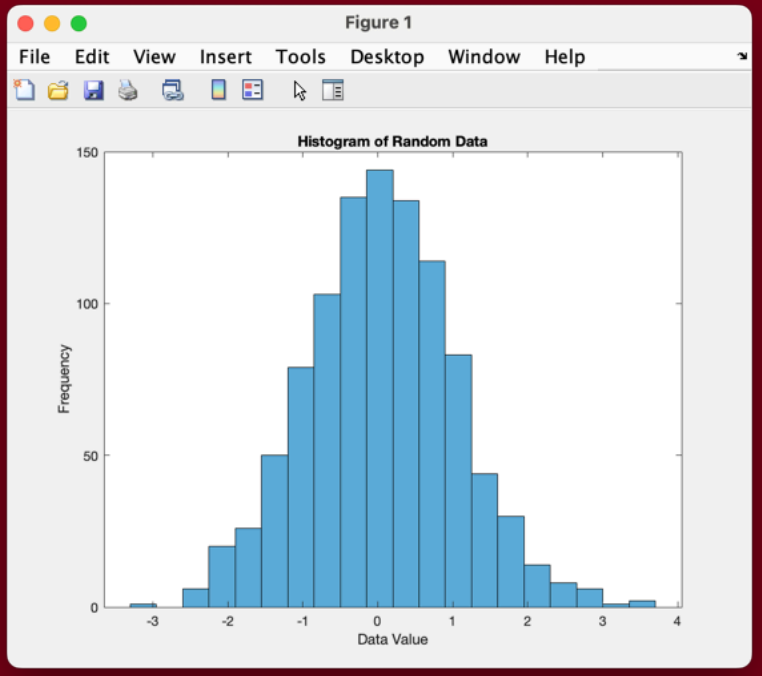
Histogram Plots:

Usage: Showing the distribution of a dataset.

Example:

```
data = randn(1000,1);
histogram(data, 20);
title('Histogram of Random Data');
xlabel('Data Value');
ylabel('Frequency');
```

Description: Groups data into bins and displays the frequency of data points in each bin.



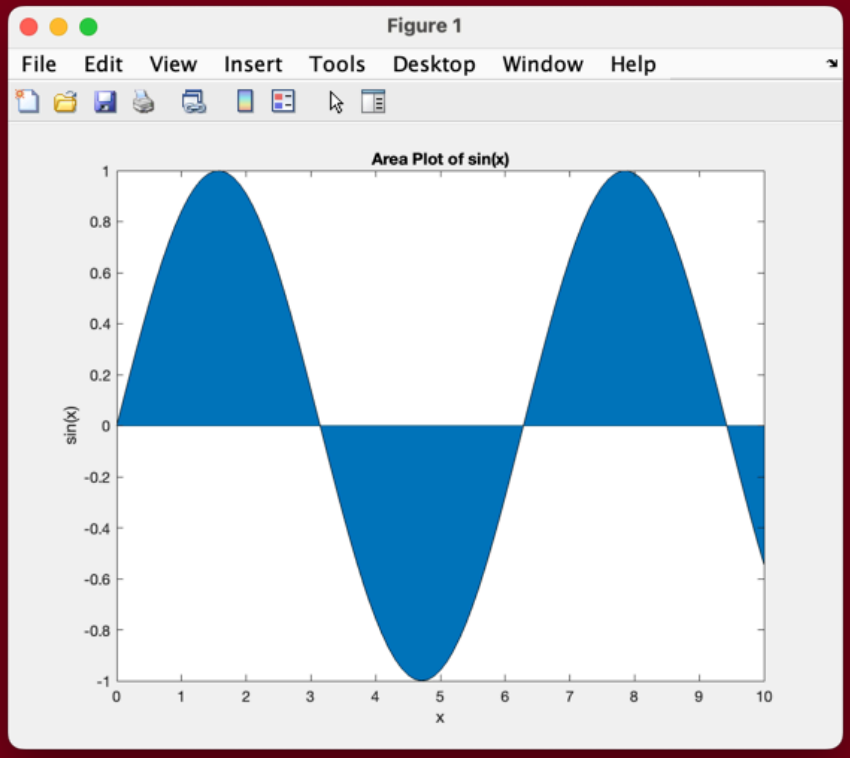
Area Plots:

Usage: Displaying cumulative totals over time or categories.

Example:

```
x = 0:0.1:10;
y = sin(x);
area(x, y);
title('Area Plot of sin(x)');
xlabel('x');
ylabel('sin(x)');
```

Description: Similar to line plots but with the area under the line filled.



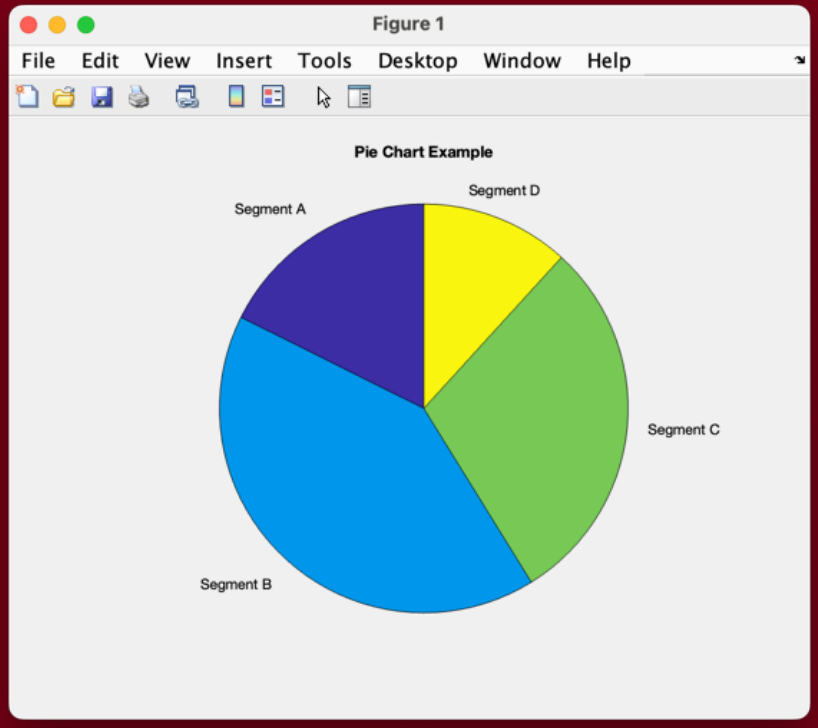
Pie Charts:

Usage: Showing proportions of a whole.

Example:

```
segments = [3, 7, 5, 2];
labels = {'Segment A', 'Segment B',
          'Segment C', 'Segment D'};
pie(segments, labels);
title('Pie Chart Example');
```

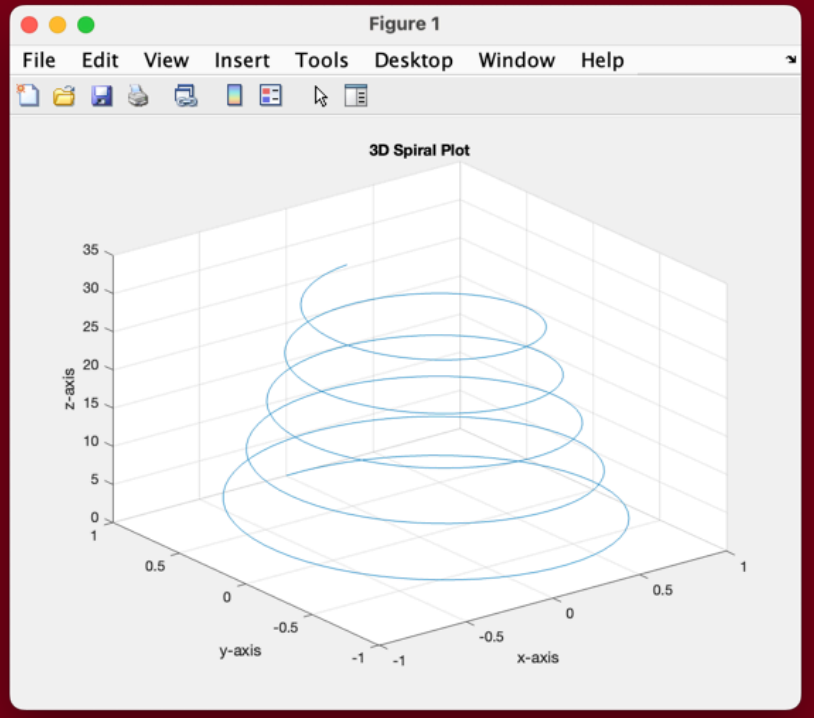
Description: Divides a circle into slices representing proportions of the total.



Basic 3D Plot:

Plot Command:

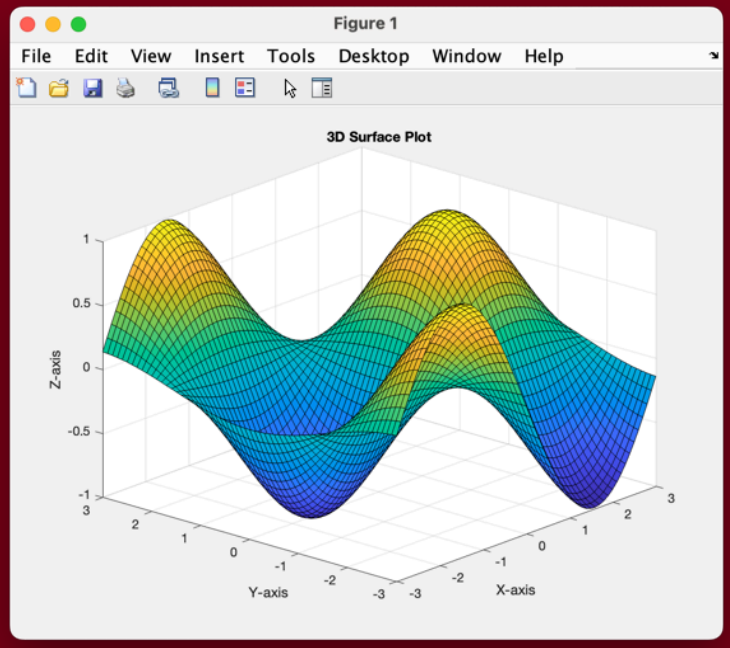
```
t = 0:0.1:10*pi;
x = exp(-0.02*t).*sin(t);
y = exp(-0.02*t).*cos(t);
z = t;
plot3(x, y, z);
xlabel('x-axis');
ylabel('y-axis');
zlabel('z-axis');
title('3D Spiral Plot');
grid on;
```



Mesh and Surface Plots:

Meshgrid and Mesh Plot:

```
[X, Y] = meshgrid(-5:0.5:5, -5:0.5:5);
Z = X.^2 - Y.^2;
mesh(X, Y, Z);
xlabel('X-axis');
ylabel('Y-axis');
zlabel('Z-axis');
title('Hyperbolic Paraboloid');
```



Example:

```
[X, Y] = meshgrid(-3:0.1:3, -3:0.1:3);
Z = sin(X).*cos(Y);
surf(X, Y, Z);
xlabel('X-axis');
ylabel('Y-axis');
zlabel('Z-axis');
title('3D Surface Plot');
```

💡 **Tip**: Use mesh and surface plots to explore the relationships between three variables visually.

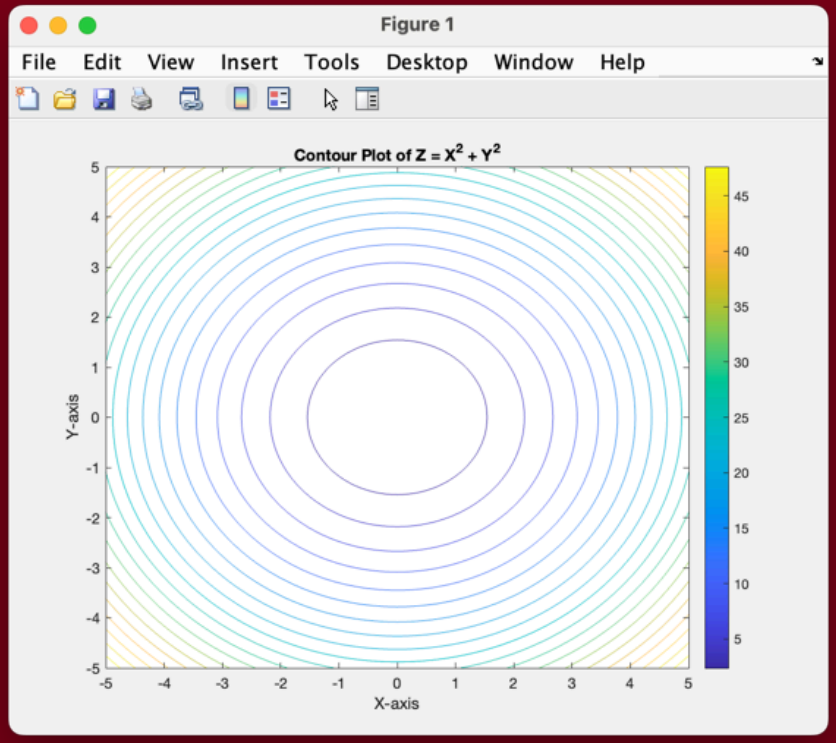
Contour Plots:

Usage: Representing 3D data in two dimensions using contour lines.

Example:

```
[X, Y] = meshgrid(-5:0.1:5, -5:0.1:5);
Z = X.^2 + Y.^2;
contour(X, Y, Z, 20);
title('Contour Plot of Z = X^2 + Y^2');
xlabel('X-axis');
ylabel('Y-axis');
colorbar;
```

Description: Shows level curves where the function has constant values.



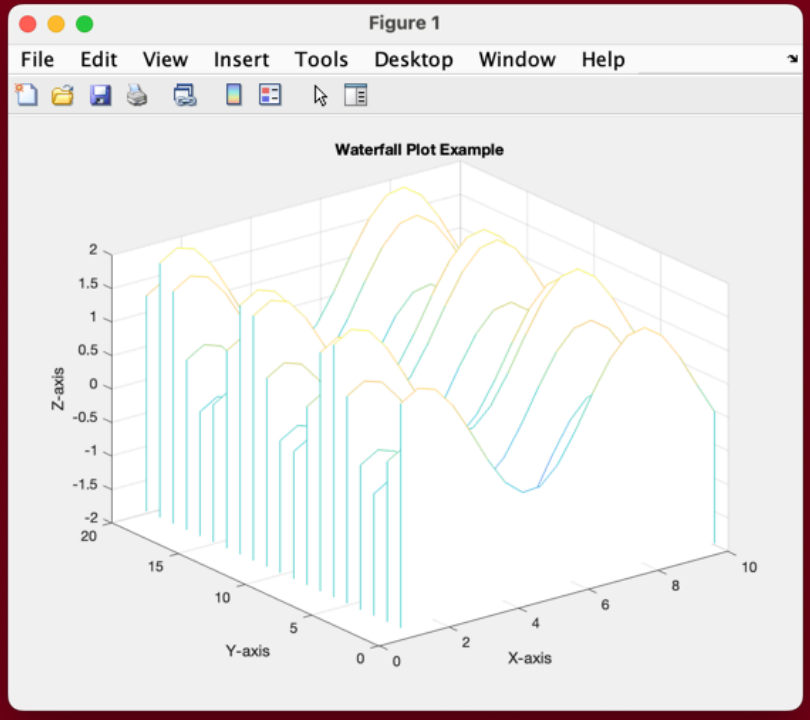
Waterfall Plots:

Usage: Visualizing slices of 3D data.

Example:

```
[X, Y] = meshgrid(1:0.5:10, 1:20);
Z = sin(X) + cos(Y);
waterfall(X, Y, Z);
title('Waterfall Plot Example');
xlabel('X-axis');
ylabel('Y-axis');
zlabel('Z-axis');
```

Description: Displays a series of lines in a 3D plot, useful for time-series data.



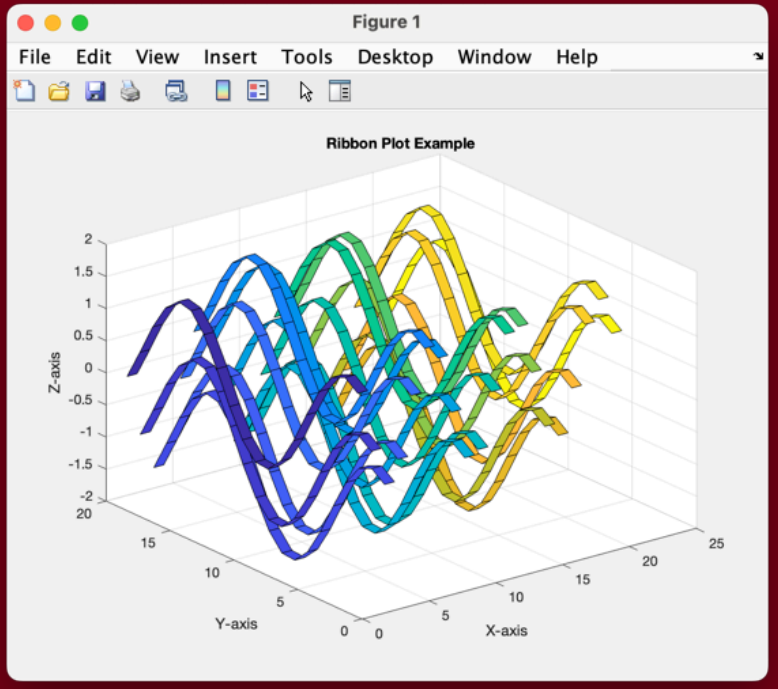
Ribbon Plots:

Usage: Plotting 3D lines with varying width or color.

Example:

```
[X, Y] = meshgrid(1:0.5:10, 1:20);
Z = sin(X) + cos(Y);
ribbon(Z');
title('Ribbon Plot Example');
xlabel('X-axis');
ylabel('Y-axis');
zlabel('Z-axis');
```

Description: Visualizes functions of one variable in 3D with a ribbon-like appearance.



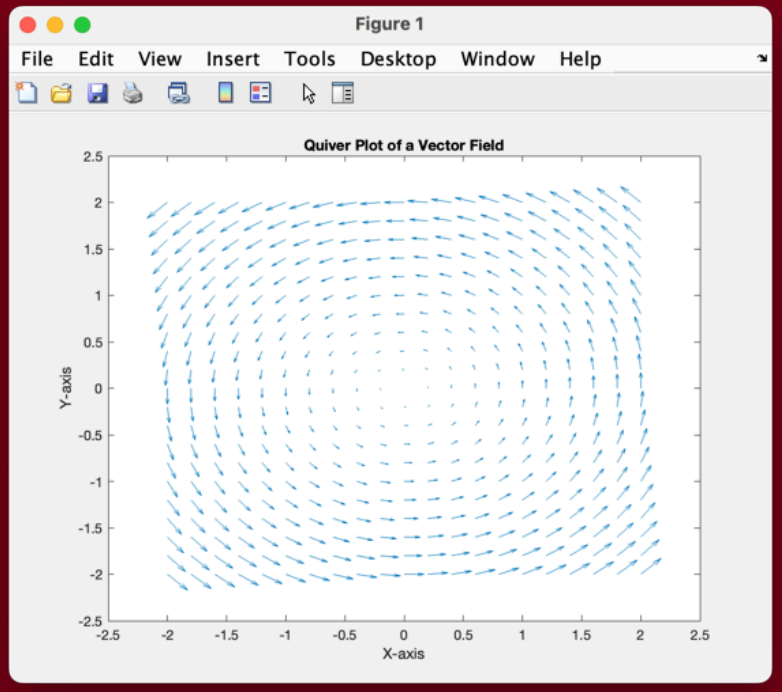
Quiver Plots:

Usage: Displaying vector fields.

Example:

```
[X, Y] = meshgrid(-2:0.2:2, -2:0.2:2);
U = -Y;
V = X;
quiver(X, Y, U, V);
title('Quiver Plot of a Vector Field');
xlabel('X-axis');
ylabel('Y-axis');
```

Description: Represents vector fields using arrows to show direction and magnitude.



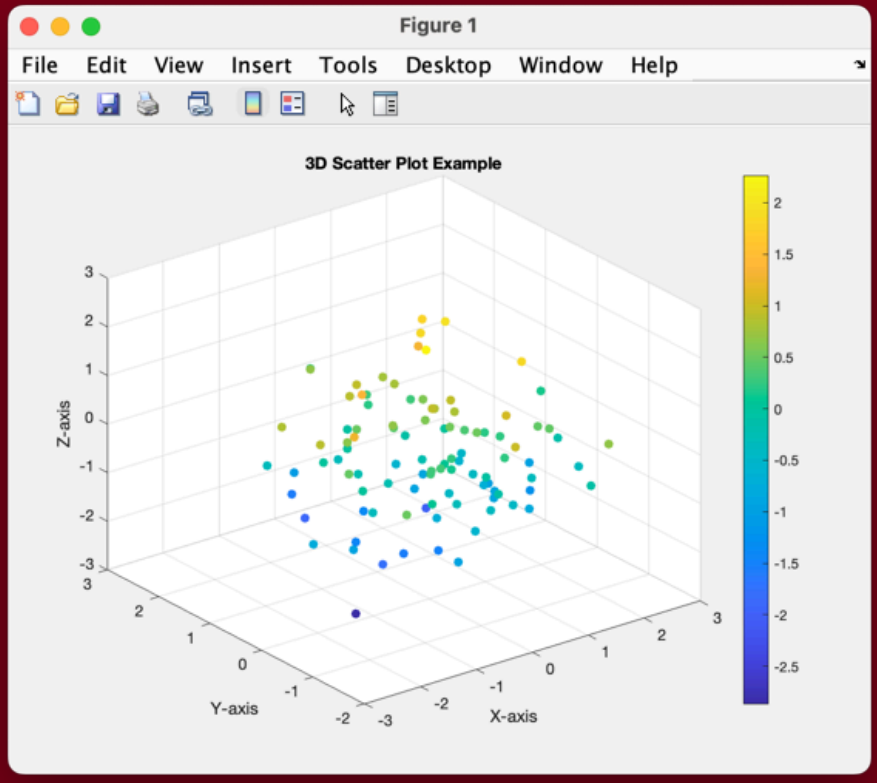
Scatter3 Plots:

Usage: Plotting 3D scatter plots.

Example:

```
x = randn(100,1);
y = randn(100,1);
z = randn(100,1);
scatter3(x, y, z, 36, z, 'filled');
title('3D Scatter Plot Example');
xlabel('X-axis');
ylabel('Y-axis');
zlabel('Z-axis');
colorbar;
```

Description: Displays individual data points in three dimensions.



Curve Fitting with `polyfit` and `polyval` 📈

Purpose:

- Fit polynomials to data for modeling and prediction.

Understanding `polyfit`:

- **Syntax**: `[p, S, mu] = polyfit(x, y, n);`
  - `p`: Coefficients of the fitted polynomial.
  - `S`: Structure containing error estimation information.
  - `mu`: Two-element vector with mean and standard deviation of `x` for centering and scaling.
- **Degree Selection**: Choose `n` based on the data's complexity.

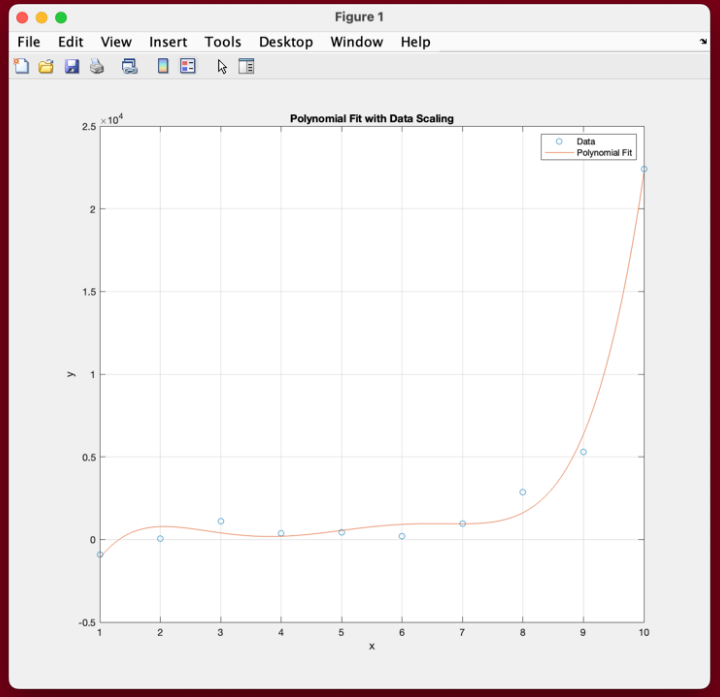
Evaluating with `polyval`:

- **Syntax**: `[y_fit, delta] = polyval(p, x, S, mu);`
  - `y_fit`: Evaluated polynomial values at `x`.
  - `delta`: Prediction error estimates.
- **Using `S` and `mu`**: Improves the accuracy of predictions, especially for higher-degree polynomials.

Data Scaling for Numerical Stability:

Example:

```
x = linspace(1, 10, 10);
% Noisy exponential data
y = exp(x) + randn(size(x))*1e3;
% Fit a 5th-degree polynomial with centering and scaling
[p, S, mu] = polyfit(x, y, 5);
x_fit = linspace(min(x), max(x), 100);
[y_fit, delta] = polyval(p, x_fit, S, mu);
plot(x, y, 'o', x_fit, y_fit, '-');
title('Polynomial Fit with Data Scaling');
xlabel('x');
ylabel('y');
legend('Data', 'Polynomial Fit');
grid on;
```

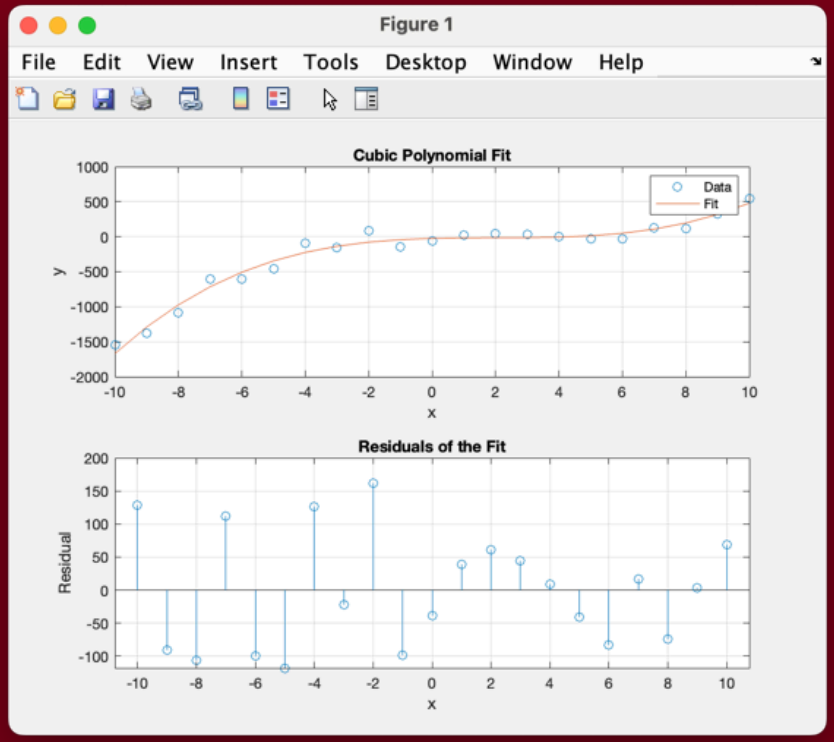


Explanation: Scaling improves the numerical properties of the fitting process.

Calculating and Plotting Residuals:

Example:

```
x = -10:1:10;
y = x.^3 - 6*x.^2 + 4*x + randn(size(x))*100;
p = polyfit(x, y, 3);
y_fit = polyval(p, x);
residuals = y - y_fit;
subplot(2,1,1);
plot(x, y, 'o', x, y_fit, '-');
title('Cubic Polynomial Fit');
xlabel('x');
ylabel('y');
legend('Data', 'Fit');
grid on;
subplot(2,1,2);
stem(x, residuals);
title('Residuals of the Fit');
xlabel('x');
ylabel('Residual');
grid on;
```

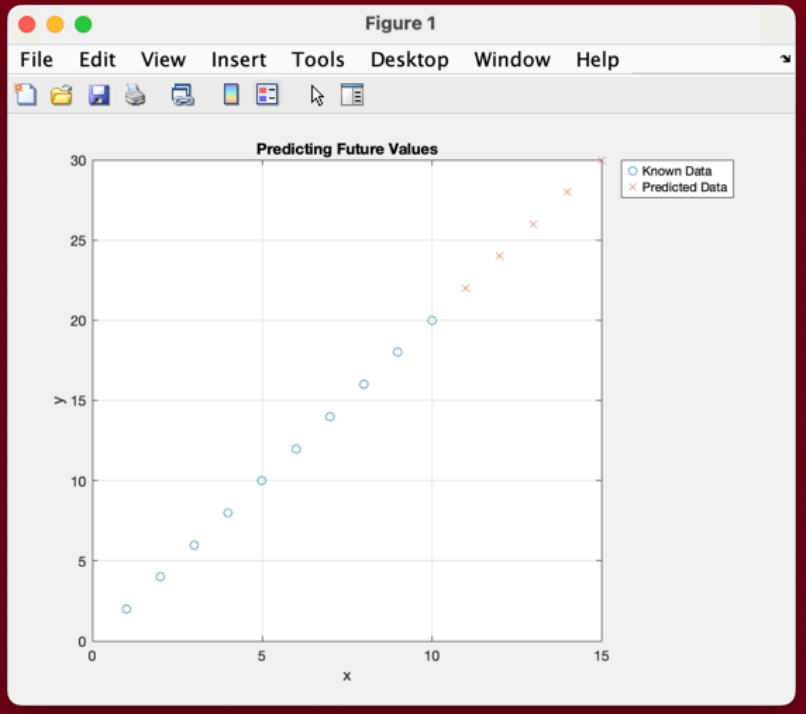


Explanation: Analyzing residuals helps assess the fit's adequacy.

Predicting Future Values:

Example:

```
x = (1:10)';
% Linear data
y = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]';
p = polyfit(x, y, 1);
x_future = (11:15)';
y_future = polyval(p, x_future);
plot(x, y, 'o', x_future, y_future, 'x');
title('Predicting Future Values');
xlabel('x');
ylabel('y');
legend('Known Data', 'Predicted Data');
grid on;
```



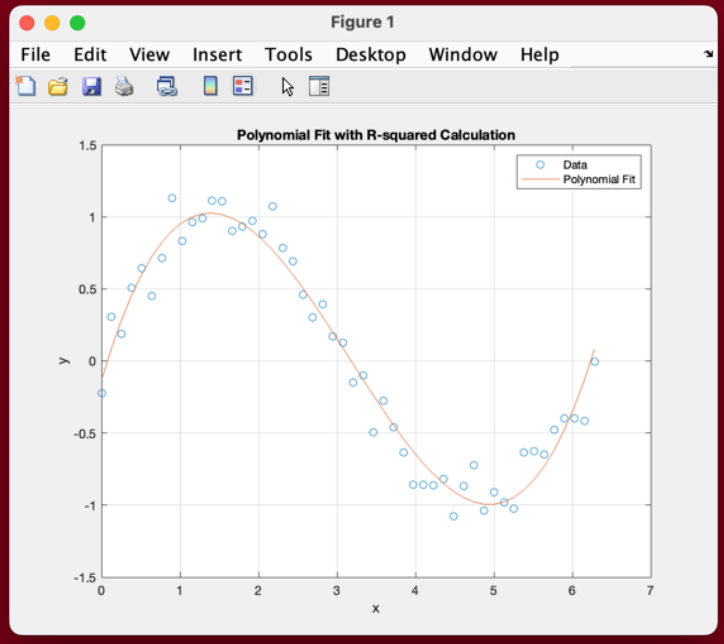
Explanation: Use the fitted model to predict values beyond the original dataset.

🥱 **Wake up** Who's been paying attention? What's wrong with this plot?

Assessing Goodness of Fit with R-squared:

Example:

```
x = linspace(0, 2*pi, 50);
y = sin(x) + randn(size(x))*0.1;
p = polyfit(x, y, 3);
y_fit = polyval(p, x);
SS_res = sum((y - y_fit).^2);
SS_tot = sum((y - mean(y)).^2);
R_squared = 1 - (SS_res / SS_tot);
fprintf('R-squared: %.4f\n', R_squared);
plot(x, y, 'o', x, y_fit, '-');
title('Polynomial Fit with R-squared Calculation');
xlabel('x');
ylabel('y');
legend('Data', 'Polynomial Fit');
grid on;
```



Explanation: The R-squared value indicates how well the model explains the variability of the data.

Animated Plots 

Creating Basic Animations:

Using a Loop to Update Plot Data:

```
x = linspace(0, 4*pi, 1000);
y = sin(x);

figure;
h = animatedline;
axis([0, 4*pi, -1, 1]);
title('Animated Sine Wave');
xlabel('x');
ylabel('sin(x)');
grid on;

for k = 1:length(x)
    addpoints(h, x(k), y(k));
    drawnow;
end
```

- **Explanation:** This code creates an animated plot of a sine wave by adding points incrementally.

Animating 3D Plots:

```
t = linspace(0, 10*pi, 500);
x = sin(t);
y = cos(t);
z = t;

figure;
h = animatedline('Marker', 'o');
axis([-1, 1, -1, 1, 0, max(z)]);
title('Animated 3D Spiral');
xlabel('x');
ylabel('y');
zlabel('z');
grid on;
view(3);

for k = 1:length(t)
    addpoints(h, x(k), y(k), z(k));
    drawnow;
end
```

- **Explanation:** Animates a 3D spiral by updating the plot in a loop.

Customizing Line Styles, Colors, and Markers 🎨

Symbol	Line Style
'-'	Solid line
'--'	Dashed line
':'	Dotted line
'-.'	Dash-dot line

Symbol	Marker
'o'	Circle
'+'	Plus sign
'*'	Asterisk
'.'	Point
'x'	Cross
's'	Square
'd'	Diamond
'^'	Upward triangle
'v'	Downward triangle
'>'	Right-pointing triangle
'<'	Left-pointing triangle
'p'	Pentagram
'h'	Hexagram

Symbol	Color
'r'	Red
'g'	Green
'b'	Blue
'c'	Cyan
'm'	Magenta
'y'	Yellow
'k'	Black
'w'	White

Combining Line Style, Marker, and Color:

- **Syntax:**

```
plot(x, y, 'style_marker_color');
```

- **Example:**

```
plot(x, y, 'r-o'); % Red solid line with circle markers
```

Line Width and Marker Size:

Adjusting Line Width:

```
plot(x, y, 'LineWidth', 2);
```

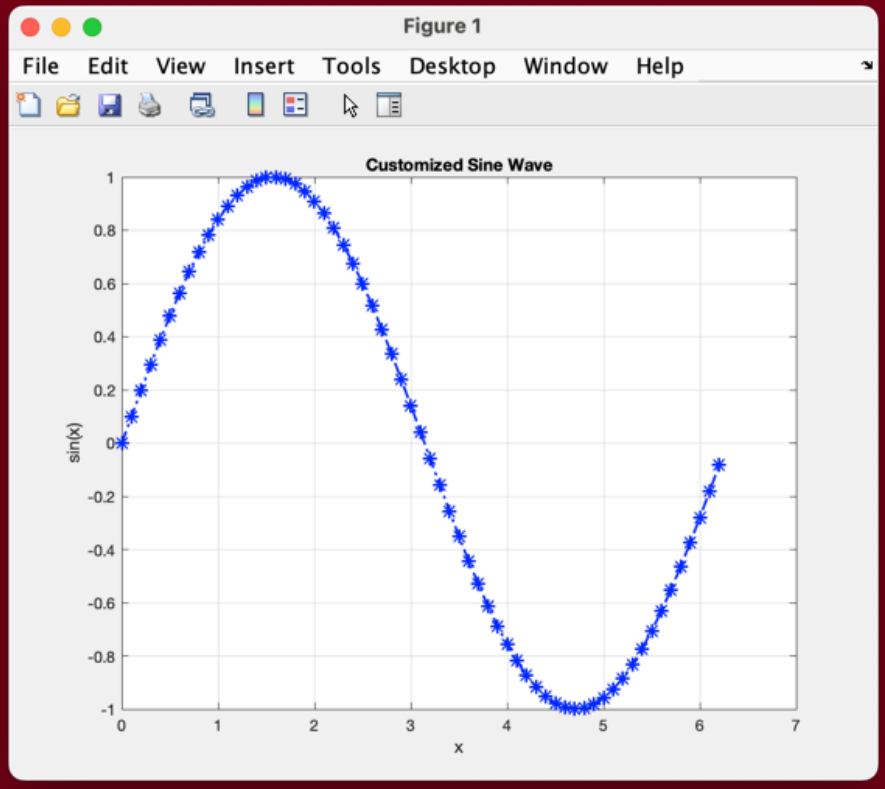
Adjusting Marker Size:

```
plot(x, y, 'MarkerSize', 10);
```

Example:

```
x = 0:0.1:2*pi;
y = sin(x);

figure;
plot(x, y, 'b-.*', 'LineWidth',
    1.5, 'MarkerSize', 8);
title('Customized Sine Wave');
xlabel('x');
ylabel('sin(x)');
grid on;
```



Explanation: This plot uses a blue solid line with asterisk markers, sets the line width to 1.5, and marker size to 8.

Key Takeaways 🎓

- Functions enhance code modularity and reusability.
- Handling optional parameters and using subfunctions increase function flexibility and organization.
- Debugging is essential for identifying and fixing errors in your code.
- MATLAB provides powerful tools for 2D and 3D plotting, which are crucial for data visualization.
- Curve fitting with `polyfit` and `polyval` is essential for data modeling and prediction.

Additional Tips 💡

- Always comment your code to improve readability and maintainability.
- Regularly test your functions with different inputs to ensure correctness.
- Use the MATLAB documentation and help functions to explore more plotting options and techniques.
- Experiment with different polynomial degrees when using `polyfit` to find the best model.

Gotchas to Watch Out For ⚠️

- **Variable Scope**: Be aware of variable scope, especially within functions. Avoid using global variables unless absolutely necessary.
- **Overwriting Functions**: Avoid naming variables the same as existing MATLAB functions to prevent unexpected behavior.
- **Debugging**: Use breakpoints and step through code to understand the flow and catch errors early.
- **Overfitting**: Be cautious of overfitting when using high-degree polynomials with `polyfit`.

Analogies 🔄

- **Functions**: Functions are like tools in a toolbox, each designed for a specific task.
- **Plotting**: Plotting data is like creating a map, where each plot type represents different ways to visualize data.
- **Debugging**: Debugging is like detective work, where you trace through the code to find the source of the problem.
- **Curve Fitting**: Curve fitting is like tailoring a suit to fit data perfectly, but too tight a fit may not be comfortable (overfitting).