Week 15A: Useful MATLAB Toolboxes

Contents

- What Are MATLAB Toolboxes?
- Installing Toolboxes
- Essential Toolboxes for Undergraduate Engineers
- Checking Which Toolboxes You Have
- Using Help Documentation
- Summary: Toolboxes for Your Course Project
- Conclusion

A Guide for Introductory Programming Students

What Are MATLAB Toolboxes?

MATLAB toolboxes are collections of specialized functions, apps, and algorithms that extend MATLAB's capabilities for specific application areas. Think of them as add-on packages that provide domain-specific tools—similar to libraries in Python. While base MATLAB provides fundamental programming capabilities, toolboxes give you powerful, pre-built functions for areas like signal processing, statistics, image processing, and control systems.

Toolboxes save you time by providing tested, optimized implementations of complex algorithms that would take weeks or months to write from scratch. For engineers, they're essential for real-world problem solving.

Back to top

Installing Toolboxes

To install a toolbox in MATLAB:

- Go to the **Home** tab in the MATLAB ribbon
- Click the **Add-Ons** button
- Select **Get Add-Ons**
- Search for the toolbox you need (e.g., Signal Processing Toolbox)
- Click **Install**

Note: You need a valid MATLAB license with toolbox access. BU students typically have access through the campus license.

Essential Toolboxes for Undergraduate Engineers

1. Signal Processing Toolbox

The Signal Processing Toolbox is fundamental for analyzing and manipulating signals—time-series data like audio, sensor readings, biomedical signals (ECG, EEG), and more. This toolbox is essential for electrical, biomedical, and mechanical engineers.

Key Function: findpeaks()

One of the most useful functions in this toolbox is `findpeaks()`, which automatically identifies peaks (local maxima) in your data. This is incredibly valuable for:

- Finding heartbeats in ECG signals
- Detecting peaks in spectroscopy data
- Identifying resonant frequencies
- Locating maxima in any waveform

Basic Syntax:

```
[pks, locs] = findpeaks(data);
```

Where:

- **pks** returns the values of the peaks
- **locs** returns the indices where peaks occur
- **data** is your signal vector

Advanced Parameters:

`findpeaks()` becomes even more powerful when you use its optional parameters to filter out unwanted peaks:

Parameter	**Description**
`MinPeakHeight`	Finds only peaks greater than a threshold value
`MinPeakProminence`	Finds peaks with a minimum vertical drop on both sides
`MinPeakDistance`	Finds peaks separated by more than a minimum distance

Example: ECG Peak Detection

```
% Load ECG data
time = ecgData(:, 1);
voltage = ecgData(:, 2);

% Find R-peaks (QRS complex) in ECG
[pks, locs] = findpeaks(voltage, ...
    'MinPeakHeight', 0.5, ...
    'MinPeakDistance', 300);

% Plot results
plot(time, voltage);
hold on;
plot(time(locs), pks, 'ro');
xlabel('Time (s)');
ylabel('Voltage (mV)');
```

Other Useful Signal Processing Functions:

- `fft()` - Fast Fourier Transform for frequency analysis
- `filter()` - Apply digital filters to data
- `butter()` - Design Butterworth filters
- `spectrogram()` - Time-frequency analysis

2. Statistics and Machine Learning Toolbox

This toolbox provides functions for statistical analysis, data visualization, and machine learning. It's essential for analyzing experimental data, performing hypothesis testing, and building predictive models.

Key Functions for Data Analysis

histogram()

Creates histogram plots to visualize data distributions:

```
histogram(heartRate);
xlabel('Heart Rate (bpm)');
ylabel('Frequency');
```

boxchart()

Creates box plots to compare distributions across groups:

```
% Combine data from different conditions
combinedData = [restingHR; exerciseHR];
group = [ones(size(restingHR)); 2*ones(size(exerciseHR))];

% Create box plot
boxchart(group, combinedData);
xticklabels({'Resting', 'Exercise'});
ylabel('Heart Rate (bpm)');
```

Print to PDF ▶

ttest2() - Two-Sample t-Test

The t-test is a statistical hypothesis test used to determine if two datasets are significantly different from each other. It's one of the most common statistical tests in engineering and science.

Syntax:

```
h = ttest2(data1, data2);
```

Where:

- **h = 1** means the datasets are significantly different (reject null hypothesis)
- **h = 0** means there's not enough evidence to claim they're different

Example:

```
% Test if resting and exercise heart rates are different
h = ttest2(restingHeartRate, exerciseHeartRate);

if h == 1
    disp('Heart rates are significantly different!');
else
    disp('Cannot claim they are different.');
end
```

Other Useful Statistical Functions:

- `mean()`, `median()`, `std()` - Basic statistics
- `corrcoef()` - Correlation coefficients
- `fitlm()` - Linear regression
- `kmeans()` - K-means clustering

3. Optimization Toolbox

The Optimization Toolbox helps you find optimal solutions to engineering problems—minimizing cost, maximizing efficiency, or finding the best design parameters. Essential for mechanical, industrial, and systems engineers.

Key Functions:

- `fminunc()` - Unconstrained optimization
- `fmincon()` - Constrained optimization
- `linprog()` - Linear programming
- `ga()` - Genetic algorithm

Example: Finding the Minimum of a Function

Suppose you want to find the minimum value of a cost function. Here's how to use `fminunc()`:

```
% Define a cost function (e.g., f(x) = x^2 + 3x + 5)
costFunction = @(x) x^2 + 3*x + 5;

% Set starting guess
x0 = 0;

% Find the minimum
[xmin, fval] = fminunc(costFunction, x0);

fprintf('Minimum occurs at x = %.2f\n', xmin);
fprintf('Minimum value = %.2f\n', fval);
```

Example: Constrained Optimization

For a problem with constraints, such as designing a cylindrical can with minimum surface area for a given volume:

```
% Objective: minimize surface area (2*pi*r^2 + 2*pi*r*h)
% Constraint: volume must equal 500 cm^3 (pi*r^2*h = 500)

% Objective function (surface area)
objective = @(x) 2*pi*x(1)^2 + 2*pi*x(1)*x(2);

% Constraint: pi*r^2*h = 500, rewritten as pi*r^2*h - 500 = 0
constraint = @(x) deal([], pi*x(1)^2*x(2) - 500);

% Initial guess [radius, height]
x0 = [5, 5];

% Lower bounds (radius and height must be positive)
lb = [0.1, 0.1];

% Solve
options = optimoptions('fmincon', 'Display', 'off');
[x, fval] = fmincon(objective, x0, [], [], [], [], lb, [], constraint, options);

fprintf('Optimal radius: %.2f cm\n', x(1));
fprintf('Optimal height: %.2f cm\n', x(2));
fprintf('Minimum surface area: %.2f cm^2\n', fval);
```

4. Symbolic Math Toolbox

This toolbox lets you perform symbolic mathematics—working with equations algebraically rather than numerically. Perfect for calculus, differential equations, and analytical solutions.

Example:

```
syms x
y = x^2 + 3*x + 2;

% Take derivative symbolically
dy = diff(y, x);    % Returns: 2*x + 3

% Solve equation
roots = solve(y == 0, x);    % Returns: -2, -1
```

5. Control System Toolbox

Essential for mechanical, electrical, and aerospace engineers working with feedback control systems, system modeling, and stability analysis.

Key Functions:

- `tf()` - Create transfer functions
- `step()` - Step response analysis
- `bode()` - Bode plot
- `pid()` - PID controller design

Example: Analyzing a Transfer Function

Let's analyze a simple first-order system, like an RC circuit or thermal system:

```
% Create a transfer function: H(s) = 1/(s + 2)
num = 1;           % Numerator coefficients
den = [1, 2];      % Denominator coefficients (s + 2)
sys = tf(num, den);

% Display the transfer function
disp('Transfer Function:');
disp(sys);

% Plot the step response
figure;
step(sys);
title('Step Response');
grid on;

% Plot the Bode diagram (frequency response)
figure;
bode(sys);
grid on;
```

Example: Designing a PID Controller

Design a PID controller for a second-order plant:

```
% Define the plant (system to control): G(s) = 1/(s^2 + 2s + 1)
plant = tf(1, [1, 2, 1]);

% Design a PID controller
Kp = 10;   % Proportional gain
Ki = 5;    % Integral gain
Kd = 2;    % Derivative gain
controller = pid(Kp, Ki, Kd);

% Create closed-loop system
closedLoop = feedback(controller * plant, 1);

% Plot the step response of the controlled system
figure;
step(closedLoop);
title('PID Controlled System Response');
grid on;

% Display settling time and overshoot
info = stepinfo(closedLoop);
fprintf('Settling Time: %.2f seconds\n', info.SettlingTime);
fprintf('Overshoot: %.2f%%\n', info.Overshoot);
```

6. Image Processing Toolbox

Work with images, video, and computer vision applications. Useful for computer science, electrical, and biomedical engineers.

Key Functions:

- `imread()`, `imshow()` - Read and display images
- `rgb2gray()` - Convert to grayscale
- `edge()` - Edge detection
- `imfilter()` - Apply filters

Example: Basic Image Processing

```
% Read and display an image
img = imread('photo.jpg');
figure;
imshow(img);
title('Original Image');

% Convert to grayscale
grayImg = rgb2gray(img);
figure;
imshow(grayImg);
title('Grayscale Image');

% Save the processed image
imwrite(grayImg, 'photo_grayscale.jpg');
```

Example: Edge Detection

Detecting edges is useful for identifying boundaries and features in images:

```
% Read image and convert to grayscale
img = imread('photo.jpg');
grayImg = rgb2gray(img);

% Apply different edge detection methods
edgesCanny = edge(grayImg, 'Canny');
edgesSobel = edge(grayImg, 'Sobel');

% Display results
figure;
subplot(1, 3, 1);
imshow(grayImg);
title('Original');

subplot(1, 3, 2);
imshow(edgesCanny);
title('Canny Edge Detection');

subplot(1, 3, 3);
imshow(edgesSobel);
title('Sobel Edge Detection');
```

Example: Image Filtering

Apply filters to reduce noise or enhance features:

```
% Read image
img = imread('photo.jpg');
grayImg = rgb2gray(img);

% Create a Gaussian filter to blur the image
h = fspecial('gaussian', [5 5], 2);
blurredImg = imfilter(grayImg, h);

% Create a sharpening filter
h_sharp = fspecial('unsharp');
sharpenedImg = imfilter(grayImg, h_sharp);

% Display results
figure;
subplot(1, 3, 1);
imshow(grayImg);
title('Original');

subplot(1, 3, 2);
imshow(blurredImg);
title('Blurred (Gaussian)');

subplot(1, 3, 3);
imshow(sharpenedImg);
title('Sharpened');
```

Checking Which Toolboxes You Have

To see which toolboxes are installed on your system, type:

```
ver
```

This will list all installed toolboxes and their version numbers.

Using Help Documentation

MATLAB's built-in documentation is excellent. To learn more about any toolbox function:

- `help functionname` - Quick syntax reference in Command Window
- `doc functionname` - Opens full documentation with examples

For example:

```
doc findpeaks
```

This opens comprehensive documentation with syntax, parameters, examples, and tips.

Summary: Toolboxes for Your Course Project

For the ECG data analysis project, you'll need:

- **Signal Processing Toolbox** - for `findpeaks()` to detect heartbeats
- **Statistics and Machine Learning Toolbox** - for `histogram()`, `boxchart()`, and `ttest2()`

Make sure both are installed before starting your project!

Conclusion

MATLAB toolboxes transform MATLAB from a general-purpose programming environment into a powerful domain-specific tool. By leveraging these pre-built, optimized functions, you can focus on solving engineering problems rather than reinventing algorithms.

As you progress through your engineering education, you'll encounter many more toolboxes. The patterns you learn now—reading documentation, understanding parameters, combining functions—will serve you throughout your career.

Happy coding!