Avoiding Common MATLAB Mistakes

Wyatt Cross

October 2025

Contents

1 Introduction

I set out to create this document to correct common errors that engineering students make when programming in MATLAB for Dr. B's Numerical Methods class at UF (EGM 3344). I have made many of them myself, and it can take practice to adjust to the way MATLAB works. The goal is to help you understand common engineering-focused misconceptions that might make you lose points or appear inexperienced. This is also not targeted at any student in particular.

In this document, valid MATLAB commands are written in monospace like this: `log(x)`.

2 Powers of 10

When given a percent error or tolerance value of $10^{-6}\%$, there are several correct ways to write it in MATLAB.

$$10^{-6}\% = 10^{-8} = 1.0 \times 10^{-8} = \texttt{10\^{}-8} = \texttt{1e-8}$$

The following will produce an inaccurate number:

```
10^-6, 1^-8, 10e-8, 1e-7
```

A percent sign divides the number by 100, so $10^{-6}\% = 10^{-8}$.

```
tolerance = 1e-8;  % Correct
tolerance = 10^-8; % Correct
```

3 Natural Logarithm

MATLAB uses `log(x)` for the **natural logarithm** (base e). This differs from many calculators or math textbooks, where `ln(x)` is used. The following table summarizes common notation conventions across fields.

	Mathematical Notation	**Engineering Notation**	**MATLAB Syntax**
log base e	$y = \ln(x)$	$y = \log(x)$	`y = log(x);`
log base 10	$y = \log_{10}(x)$	$y = \log_{10}(x)$	`y = log10(x);`

Table 1: Mathematicians may also use `log()` and Engineers may use `ln()`, but this table gives common notation.

In MATLAB, `log()` is always base e, while `log10()` is explicitly base 10.

4 Exponential Functions

In MATLAB, the exponential function, e^x, is defined as `exp()`.

$$y = e^x \quad \rightarrow \quad \texttt{y = exp(x);}$$

A common mistake is confusing the mathematical constant e (Euler's number, e ≈ 2.71828) with the exponential function e^x. Doing 2.71828^x is incorrect. You can use `exp(1)` to represent the mathematical constant e if you need 2.718281828459045... for anything.

5 Plotting Continuous and Discrete Data

Engineers must be able to distinguish between continuous functions and discrete data points. Understanding this difference helps you avoid making misleading plots.

Continuous Data

Continuous data come from mathematical expressions or simulations where you can evaluate many points. Crucially they share the property that greater refinement in an area of the plot can be achieved by increasing the density of points in the domain. In other words, if you desire greater detail between 0 and 1, you can sample more points in that interval. In the following example, you are able to cleanly increase the number of points in the domain, `x`, by altering the `linspace()` function.

```
x = linspace(0,2*pi,1000);    % Creates the domain to evaluate
y = sin(x);                   % Evaluates the domain using the sine function
plot(x, y, 'LineWidth', 1.5) % Plots the domain and range with a thicker line
xlabel('x')                   % Label horizontal axis
ylabel('sin(x)')              % Label vertical axis
title('Continuous Sine Function') % Add a descriptive title
grid on                       % A grid makes the plot easier to read
```

Discrete Data

Discrete data generally come from real-world measurements or samples. A key property of discrete data is that if you have points at 0 and 1 and want to know what happens between them, the only truly accurate method is to increase your sampling resolution when you run your experiment. Simply drawing a line between points can provide a visual estimate, but it may incorrectly suggest that the data is continuous as there are no markers on individual points.

A caveat is that if you fit a function such as a polynomial or sinusoid to your data, the resulting model is mathematically continuous, even if the original data were discrete.[1]

When plotting discrete data in MATLAB, it is best to use **markers** or **stem plots** to clearly indicate individual data points and avoid implying continuity. In the following example, circular markers on each point are sufficient to indicate that the points are discrete.

```
x = 0:1:10; % Domain, this could be measurements taken every second for 10 seconds
y = [3, 2, 1, 1.5, 2.5, 3, 2.5, 1.5, 1, 2, 3]; % Range of data collected
plot(x, y, 'o-')  % Plot result as circles connected by lines
```

FFT Example: The discrete Fourier transform (DFT) of a signal is another case where stem plots are useful. Here is an example computing the Fast Fourier Transform (FFT) of a simple discrete signal[2] and plotting its magnitude using `stem`.

```
x = 0:0.1:2*pi;                          % Create time domain
y = sin(2*pi*1*x) + 0.5*sin(2*pi*3*x); % Create the signal: 1Hz wave + 3Hz wave
Y = fft(y);          % Compute the Fast Fourier Transform (FFT)
n = length(Y);       % Count the number of data points
Fs = 1/0.1;          % Sampling Frequency = 1/sample interval
f = (0:n-1)*(Fs/n); % Frequency vector for plotting
stem(f, abs(Y)/n)   % Plot using stem(), 'abs(Y)/n' normalizes frequency magnitude
xlabel('Frequency (Hz)')
ylabel('Magnitude')
title('FFT of a 1Hz and 3Hz signal')
grid on
```

6 Functions and Function Handles

Functions in MATLAB let you reuse code efficiently. For example, define a file named `squareNum.m`:

```
function y = squareNum(x) % Specifies function name and the output variable, y
    y = x.^2; % The dot '.' ensures this works for vectors (element-wise)
end % An 'end' is required to define the MATLAB function
```

You can call this in your script as:

```
squareNum(3)
```

Functions can also be included in a script if they are placed at the very end of your script, after all other code that you plan to run.

A **Function Handle** (denoted by the @ symbol) is essentially a reference to a function. It allows you to treat a function like a variable, which is helpful when you need to pass a function as an argument to another function (such as in ODE solvers or optimization). Anonymous functions are a common use for function handles as you can define them anywhere inline in your code and are often sued for simple expressions. For example:

[1] This assumes that the fitted function itself is continuous. For example, a function such as `tanh()` is only continuous within its valid domain, and some fit types (piecewise or hyperbolic) may introduce discontinuities if handled incorrectly.

[2] *But I thought that the output of a mathematical function such as sine was continuous?* In this case as the domain consists of a finite number of points, so the FFT is attempting to fit continuous functions (sinusoids) to discrete points. The resulting frequencies are discrete as the FFT cannot resolve every possible continuous frequency due to the finite domain. This is a consequence of doing the DFT or FFT *numerically*.

```
f = @(x) x.^2 + 3*x - 5; % Squaring using '.' lets this work with vectors
fplot(f, [-5 5]) % fplot() is used to plot the ouput of f() over a given interval
f(2) % This is how you call a function handle somewhere else in your code
```

Function handles are especially useful when given various mathematical functions to evaluate every iteration in a numerical method. Specifically, they make defining functions and their derivatives very easy.[3]

```
f = @(x1,x2) x1.^2 + x2; % Define a 2-variable function
dfdx1 = @(x1,x2) 2*x1;   % Take the derivative with respect to x1
f(2, 1)     % Evaluate the original function (=5)
dfdx1(2, 1) % Evaluate the derivative (=4)
```

7 Complex Numbers

MATLAB reserves both the letters `i` and `j` as the imaginary unit $\sqrt{-1}$ (i.e., $i^2 = -1$). Computer programmers like to use i,j,k as loop indices because they correspond to common vector/matrix/tensor notation. This can cause issues with how MATLAB defines the imaginary unit, and makes your code unclear, as the reader has to make sure you are talking about an integer index in a loop, as opposed to a complex number.

```
x1 = 5 + 3i; % Defines a complex number: 5 + 3i
x2 = 5 + 3j; % Also defines a complex number: 5 + 3i
for ii=1:100 % Correct use of i as an index. ii and jj are convention in MATLAB
```

Another option is to use `index` or some other word as your loop index.

8 Infinity and Zero

MATLAB has special reserved keywords to represent mathematical concepts of infinity and numbers very close to zero. Using these keywords is better than manually entering a large or small number.

Infinity (`Inf`): The keyword `Inf` represents positive mathematical infinity (∞). It can arise from mathematical operations like dividing a non-zero number by zero.

Specifically, when setting up numerical methods with an error inequality, `Inf` should be used to define the error variable so the method works with any input. In the following example, the error of the function, `err`, must be defined before the loop, however it must also always be greater than the tolerance value, so the loop has an opportunity to run at least once.

```
tolerance = 1e-8;
err = Inf;
while err > tolerance
    % Numerical method here
    % err = {Some error frunction using the result of the method};
end
```

Not a Number (`NaN`): `NaN` is a value that represents undefined results from mathematical operations. This often occurs when a function is undefined (like $\tan(\pi/2)$ or $\frac{x}{0}$). Any operation involving a `NaN` will almost always result in `NaN`. When debugging, a `NaN` in your output is a strong indicator of a division by zero or a bad function call earlier in your code.

[3] Note, a human (the author) took the derivative using the brain. MATLAB did not do calculus.

Epsilon (`eps`): `eps` is the keyword for *Machine Epsilon*, the smallest number ϵ that can be added to the number 1 and still be distinguished from 1 in the computer's floating-point arithmetic system.

9 Loops

For Loops

For loops are generally used when iterating through data of fixed and known size such as a dataset you are provided with or looping through elements of a vector/matrix. If you are using a for loop when solving a numerical method to within a given tolerance, you are doing it wrong.[4]

While Loops

While loops are used when you do not know how long a method will take to converge. They follow the form `while {boolean}`, where you can use an inequality to check convergence every iteration.

One common incorrect *hack* is to set the loop always true and use an if condition in the loop to determine when to exit. This is redundant and confusing. Do not do this. Instead, you can usually take the contents of your if statement in the loop and put it in the line that creates your while loop.

```
while true % Do not do this
    % Method would go here
    if err > tolerance % Look how easy it is to move the inequality up
        break % This is redundant since the loop will break automatically
    end
end
```

Additionally, many decide to add a maximum iterations limit to their loop, likely because that is how the textbook/Chegg/Generative AI does it. Regardless, if you set up your loop correctly, using a iteration limit is unnecessary for all methods with a bracket and for good guesses in bracket-less methods.[5]

```
maxIter = 1000; % Not needed if you set the problem up correctly
count = 0;
while count < maxIter & err > tolerance % Do not include the maxIter part
    % Method would go here
    count = count + 1;
end
```

Using `while true` or `while count < maxIter` in your script is wrong and will lose you points.

10 How to Write a Good Comment

MATLAB uses the percent sign (%) for comments. Explain *why* you chose to do something that line, do not restate what your syntax does. Also, placing a `% X's method` and no additional comments at the top of your loop is not helpful. The goal is to allow someone unfamiliar with your specific code to understand what you are doing each line. This does not mean you have to give detailed comments when you are plotting (most MATLAB users know how labels and titles work) but, key inputs, loops, and lines in your method should be explained.

In the same vein, variables should have concise names. You do not need 20 characters to explain that you have a derivative, `d{func}d{derivative}` is sufficient.

[4] If you are writing flight software for the Space Shuttle, you get to use a 3 iteration fixed point method to solve Kepler's Equation. The rest of us have more that 25Hz computers (not MHz or GHz) and can afford to use a while loop to solve.

[5] Iteration limits are fundamentally a safety check to stop your code if a loop fails to converge. They should not be the primary mode of exiting the loop. In EGM 3344, understanding ways of quantifying error is an important part of the class and using `maxIter` defeats the whole purpose of understanding and implementing error calculations correctly.