



MATLAB Tutorial: An Introduction to MATLAB

AUGUST 31, 2015 BY UDEMYTUTORIALS

TOP UDEMY COURSES:

[Top Java Courses](#)

[Top Python Courses](#)

[Top Excel Courses](#)

[Learn Excel With This GIF Tutorial](#)


 [Become a Web Developer from Scratch! \(810 students\)](#)

 [Advanced Excel Training \(42,660+ students\)](#)

 [Coding for Entrepreneurs \(4810+ students\)](#)

 [Advanced Java Programming \(735+ students\)](#)

POPULAR POSTS

 [How to Build an iPhone App from Scratch for Technical People: Your quick and dirty guide](#)


 [Excel Formulas: 10 Formulas That Helped Me Keep My Job](#)

 [Code Wars: Ruby vs Python vs PHP \[Infographic\]](#)

 [Top 10 Programming Languages to Learn in 2019](#)

 [How to Add Ringtones To Your iPhone \(Updated for iOS 7\)](#)

 [8 Best PowerPoint Presentations: How To Create Engaging Presentations](#)

 [Java Interview Questions: How to crack the Top 15 questions](#)

 [Drupal vs Joomla vs WordPress: CMS Show \[infographic\]](#)

 [Making an App: 6 Things You Should Consider Before Getting Started](#)

 [10 Fórmulas de Excel para ser Más Productivo](#)

TABLE OF CONTENTS: CLICK TO JUMP TO A SPECIFIC SECTION

[What is Matlab?](#)

[Setting it Up](#)

[Interactive Mode and Script Files](#)

[Writing Matlab Code](#)

Brief overview

Variables

Arrays and Matrices

Control Flow

Conditional Branches

While/For Loops

Structs

Classes

Properties

Functions/Methods

Inheritance and handle

[Applications](#)

Plotting functions

Conway's Game of Life

[Conclusion](#)

What is Matlab?

Matlab, an abbreviation of Matrix Laboratory, is a commercial programming language that offers a range of built-in functions and tools. It was developed as a language to synthesize programming, as in C, C++, Pascal, or Java, into a stronger and easier-to-use math development environment. Its primary users range from mathematics students and academics to those in other science fields, for use with analysis, manipulation, extrapolation, and visualization of data.

Setting it Up

Because Matlab is a commercial programming language, it requires a **purchase** in order to access its features. Mathworks, the company that develops Matlab, provides a **free trial**, as well as **subsidized rates for students**. Some schools and companies also offer product keys and online access through, for instance, Citrix.

Open source projects that replicate much of the Matlab framework also exist. Most of this guide will be identical for **Octave**, which is under active development, and **FreeMat**, which had its last release in 2013. Modifications can make it compatible with **Scilab**, which has a slightly different syntax. More information about the differences between Scilab and Matlab can be found [here](#)

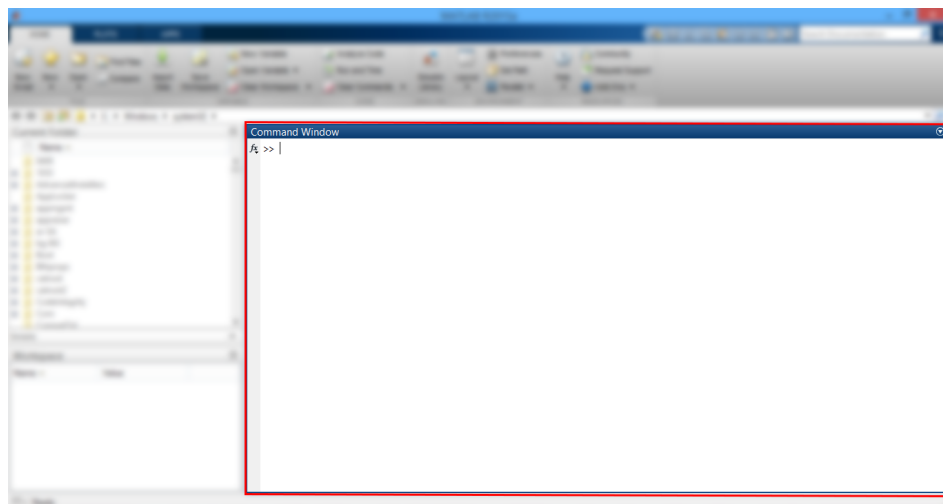
In addition to the core features, Mathworks provides a powerful package with **Simulink**, which works closely with Matlab, offering visualizations of various systems, including 3D models and statistical data as they change with time. An open source equivalent is **Xcos**, which comes packaged with Scilab.



Interactive Mode and Script Files

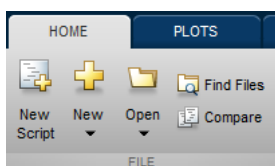
Every program written for Matlab will consist of a sequence of lines of code that eventually produce a result. There are two main ways of inputting these commands: interactive mode and script files.

When Matlab is launched, a command window will appear in the middle of the display

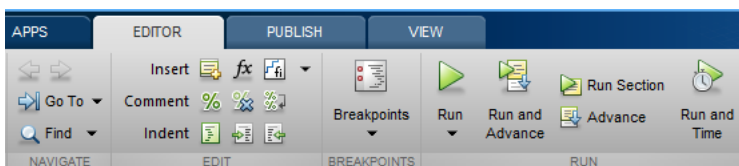


Similar to some other languages, such as Python, Perl, and Haskell, Matlab has a fully functional interactive mode in which any code may be entered and immediately evaluated, one line at a time.

An alternative that is more ubiquitous in programming is using script files with a .m file extension. These can be created by pressing the *New Script* button in the *File* menu under the *Home* tab.



After creating a new script and saving it, the script can be run with the *Editor* tab:



There are a number of useful tools in the *Editor* tab that we'll go over later. For now, though, you simply need to know about the *Run* button. Most of the other buttons are used when debugging.

In this tutorial, we'll mostly be using interactive code for small bits of code and the scripting approach for more full programs. For clarity, the `>>` at the beginning of each interactive mode line will be preserved whenever it appears. This said, programs will run identically in each mode, albeit the order in which output appears will be different.

Writing Matlab Code

Brief overview

Before typing any code, there are a few things to clarify.

1. Excess **whitespace** will not affect the function of a program, excluding new lines. In this way, `a=3` is exactly identical to `a = 3`. Generally, sections that are together will be offset with a similar indentation, for clarity.
2. Lines generally will end with a `;`. In interactive mode, though, leaving the semicolon off will print details about the statement, as will be seen in examples below.
3. Lines beginning with `%` are comments for the reader. Matlab compilers ignore these lines automatically, and it is generally advisable to include these, so as to have cleanly readable code.
4. Names of variables should generally follow coding guidelines as given [here](#)

Variables

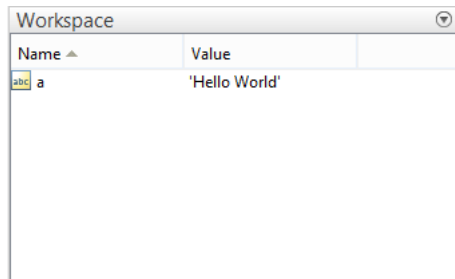
Going into the command window, we can start to write our first line of code:

```
>> a = 'Hello World'
```

After hitting enter, immediately a response is given:

```
a =  
  
Hello World
```

You may also notice a change in the bottom right window:



After entering in the statement, a variable named `a` will appear in our workspace with the value 'Hello World'. `a` can be substituted with any name within reason, and coding convention generally dictates naming variables with capitalization as in `myVariableName`.

On the left side, we have set `a` to 'Hello World'. There are a number of primitive choices for variables, including:

1. characters, written as '@'. A group, or *array* of characters is written similarly, as 'Hello'
2. integers, written plainly as a number like 2009. Booleans are also stored as integers, where `false` is 0 and `true` is 1.
3. double precision floating points, or decimals, written similarly like 3.14159
4. Arrays and matrices, written as [1 2 3 4; 5 6 7 8]. We'll discuss these more in a later section.

Once a variable is in the workspace, it can be referenced and modified. For instance, consider this simple program:

```
>> exampleNumber=3  
  
exampleNumber =  
  
    3  
  
>> exampleNumber*3  
  
ans =  
  
    9
```

In addition to the variables that have been defined, another one pops up: `ans`. This appears as the output, whenever a suitable variable isn't given.

Arrays and Matrices

What would *Matlab* be without a healthy supply of functions and input for matrices?

Let's construct an example matrix:

```
>> exMatrix = [1 4 7; 2 5 8; 3 6 9]  
  
exMatrix =  
  
    1    4    7  
    2    5    8
```

```
3 6 9
```

Both arrays and matrices are declared with square brackets, `[]`, and are input row by row. Separation between row elements can be done either with a comma or with spaces. Separation between rows should be done with a semicolon.

In addition, an index notation can be used for row vectors. In MatLab, generated ranges are made with the form 'start:end' or, when needed, `start:increment:end`. Unlike some programming languages, both `start` and `end` are inclusive, so `1:4` will return `[1 2 3 4]`, `1:2:5` will return `[1 3 5]`, and `1:4:7` will return `[1 5]`

Now that we have a matrix in the workspace, it can be manipulated with a large number of built in functions for **matrices**. In addition, nearly all other functions will act element by element; `sin(matrix)` will perform the `sin` on every element. The notable exceptions are for functions like **matrix multiplication** and **exponentiation**, where their expected forms in math take over:

```
>> exMatrix^2

ans =

    30    66   102
    36    81   126
    42    96   150
```

This works, as Matlab views the input as this:

$$\begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix}^2 = \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix} \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix} = \begin{pmatrix} 30 & 66 & 102 \\ 36 & 81 & 126 \\ 42 & 96 & 150 \end{pmatrix}$$

The lesser used behavior of doing these operations element-wise can be done by inserting a `.` before the operation, as in `matrix.^2`, `matrix.*matrix`, and `3.^matrix`.

When referencing parts of a matrix, indexing is done with parentheses and is 1-indexed. This means that the first element of our example matrix is given by `matrix(1)`, which returns `1`. The example matrix further shows the position numbers for a 3x3 matrix. In addition, matrices can be referenced by coordinates in the same way, by (row, column), so position 6 can be referenced by `matrix(3,2)`. Matlab also allows for range, in the same way as the index notation:

```
>> matrix(1:2,2:3)

ans =

     4     7
     5     8
```

Both of these can be used in a third way to create matrices, where any missing, but necessary, values are automatically filled in with zeros:

```
>> e(2:3,2:3)=3

e =

     0     0     0
     0     3     3
     0     3     3
```

Control Flow

After we have a few different data types under our belt, explaining how to manipulate them into a usable program is important.

Conditional Branches

One of the major building blocks of coding in general is the ability to only execute code when certain criteria are met. In Matlab, as with most languages, this is done with `if`, `elseif`, and `else` blocks. This can be constructed by the following:

```
if condition1
    %This executes if condition1 is true
elseif condition2
    %This executes if condition1 is false, but condition2 is true
else
    %This executes if condition1 is false, and condition2 is false
end
```

Each condition can be a single variable or an expression. Variables are evaluated as true or false by whether or not they are 0 or empty; [], 0, and [0, 0] all evaluate as false, whereas 'b', 3, and [0 0 1] all evaluate as true.

While/For Loops

Sometimes, instead of simply branching, looping while a constraint holds or for a certain number of iterations is useful. For a large number of tasks in Matlab, using loops is not only useless but less efficient; operations that can be performed on arrays will almost always be preferable. When this isn't possible, `while` and `for` loops are relatively universal constructs that are used. To start, the syntax for a `while` loop is very similar to an `if` statement:

```
while conditionHolds
    %Perform an action
end
```

These are frequently used to keep an activity repeating until it is finished. An example would be in the collatz conjecture:

```
while i ~= 1
    if mod(i, 2)
        % i is odd
        i = 3*i + 1;
    else
        % i is even
        i = i/2;
    end
    display(i)
end
```

In this example, `~=` means "does not equal", so the loop runs until `i` is 1. The condition is only checked at the beginning of the loop, though, so 1 will still be displayed at the end.

Further, `for` loops provide similar functionality to `while` loops, but a counter is provided in the body of the loop. Typically, the syntax is given by something of the form `for i = 1:5`. Any value can be given in the range, in lines with the index notation Matlab provides. For very CPU-intensive loop bodies, where the order in which the loop is executed does not matter, there is also an identically stated `parfor`, which uses multiple cores to run the `for` loop in *parallel*.

In both types of loops, some additional control is needed for management. These are given by the following:

- `break`: Any time a `break` is encountered, the rest of the loop will be ignored, and the statement after the `while` loop will be run.
- `continue`: A `continue` is a softer "break", in which the rest of the current iteration of the loop is skipped, and then the program continues back to the top of the loop; it skips an iteration and continues with the other iterations.

More information about any control flow, including a few other keywords can be found [here](#).

Structs

Sometimes, we need to use sets of data that are grouped together with more structure than a matrix. In many cases, this can be accomplished with the use of a struct. With more complexity, an entire class might need to be created, which will be discussed in a later section.

Imagine having a few people for whom we would like to store information. A single person could be created immediately by specifying its attributes:

```
person.firstName = 'Joe';
person.lastName = 'Smith';
person.age = 25;
display(person);
-----
person =

    firstName: 'Joe'
    lastName: 'Smith'
         age: 25
```

Arrays of structs can also be used in a way that works much like tables

```
people(1).firstName = 'Sigmund';
people(1).lastName = 'Freud';
people(1).age = 25;

people(2).firstName = 'Carl';
people(2).lastName = 'Jung';
people(2).age = 30;

display(people);
-----
people =

1x2 struct array with fields:

    firstName
    lastName
         age
```

Querying for all firstNames by doing `people.firstName` will produce each, one by one:

```
ans =

Sigmund

ans =

Carl
```

Classes

Matlab classes provide additional functionality and **encapsulation** than structs. Note that classes and functions are not available in interactive mode. We'll start with a basic setup, declared appropriately in `NameOfClass.m`:

```
classdef NameOfClass
    properties
        % Class variables
    end
    methods
```

```

    % Class methods
end
end

```

In its current form, an instance of this class could be created by doing `instanceOfClass = NameOfClass()`.

Properties

All variables contained in the properties section operate similarly to the way a struct acts: A named variable is given a value of one of our declarable types. In addition, the class gives **additional attributes** that may be assigned to sets of properties. Some of the most common attributes are setting the type of variable (`Constant`, or `Static` for methods) and setting access to a variable (`private`, `protected`, and `public`). All properties with similar attributes are grouped in a section, so the properties for one's own polar vector class might look something like the following:

```

properties
    %Set variables with some default value
    radius=1;
    angle=0;
end
properties (Constant, Access = protected)
    PI = 3.14159265358
end

```

In this case, methods for the vector might require constant access to `PI`, so it is easier to declare it as part of the class (In practice, `PI` in particular is unnecessary to declare, since Matlab has `pi` as a default constant to the mathematical value). Other useful properties that need to be the same for all instances of a class should be declared in a constant section, so that changing them is as easy as swapping out their value in the properties section.

The `Access` of a variable, which can also be split up into `GetAccess` and `SetAccess`, refers to the context in which a class variable can be referenced. The three access levels in Matlab are the following:

- `public`: This is the default access level that allows any context to reference a variable
- `protected`: The variable can only be referenced from within the class or any of its subclasses
- `private`: The variable can only be referenced from within the class

Functions/Methods

After the properties of a class have been defined, the way they interact with method calls can be defined. First, let's start with the syntax and meaning of a function. Put simply, a function is a way to tell Matlab how to take some input, called arguments, and produce a defined output. The following is an easy function that shows how one may be written:

```

function result = addTwo( input )
result = input+2;
end

```

After saving this with filename `addTwo.m` in a location that matlab can find, it can be used as:

```

>> addTwo(3)

ans =

    5

```

These kinds of functions may be placed in the `methods` section to encapsulate uses for the class. There are also two main kinds of methods: instance methods, which work on a particular instance of a class, and `Static` methods, which do not. Of particular note are the instance methods that override functionality. Let's start with the constructor, which is an important interface for readily creating an instance of a class:

```

function obj = PolarVector(radius, angle)
    obj.radius = radius;
    obj.angle = angle;
end

```


Unlike the initial example, an instance of this class can be created with quickly assigned default values by doing, for instance, `instanceOfClass = PolarVector(3, 0.0)`. Note that this constructor cannot go under a `Static` attribute, as it implicitly implies that `obj` is an instance of the class.

In addition to a constructor, there are other types of functions that may be overridden, including the use of operators such as `+` or `-`. A list of these operators and their associated functions can be found [here](#). Overloading these operators follows a more consistent scheme for instance methods. For instance, take an example overridden version of the addition operator for a `PolarVector`:

```
function sum = plus(obj, objOther)
    x = obj.radius*cos(obj.angle)+objOther.radius*cos(objOther.angle);
    y = obj.radius*sin(obj.angle)+objOther.radius*sin(objOther.angle);
    rad = sqrt(x^2+y^2);
    ang = atan2(y,x);
    sum = PolarVector(rad,ang);
end
```

Given two instances of `PolarVector`, `obj1` and `obj2`, this method may be referenced in three different ways: statically with two arguments (`PolarVector.add(obj1, obj2)`), as an instance with one argument (`obj1.add(obj2)`), or with the overloaded operator (`obj1+obj2`). In the second way, `obj1` is sent in as the first argument to the function, as will happen with all instance methods.

Finally, we have static methods. Imagine in our polar class we want a method to create a north facing vector of some radius. This can be done in various ways, including having a static method that takes a radius argument. With the full method signature, it would be created in the following way:

```
methods (Static)
    function obj = north(radius)
        obj = PolarVector(radius, pi/2);
    end
end
```

This may then be called by `PolarVector.north(3)`.

Inheritance and handle

Often, there is a need for a specific type of an existing class that has special features over the old class. This is when “object inheritance” comes into play. If a class is a *subclass* of another class, then it can be declared as follows by `classdef subClassName < superClassName`.

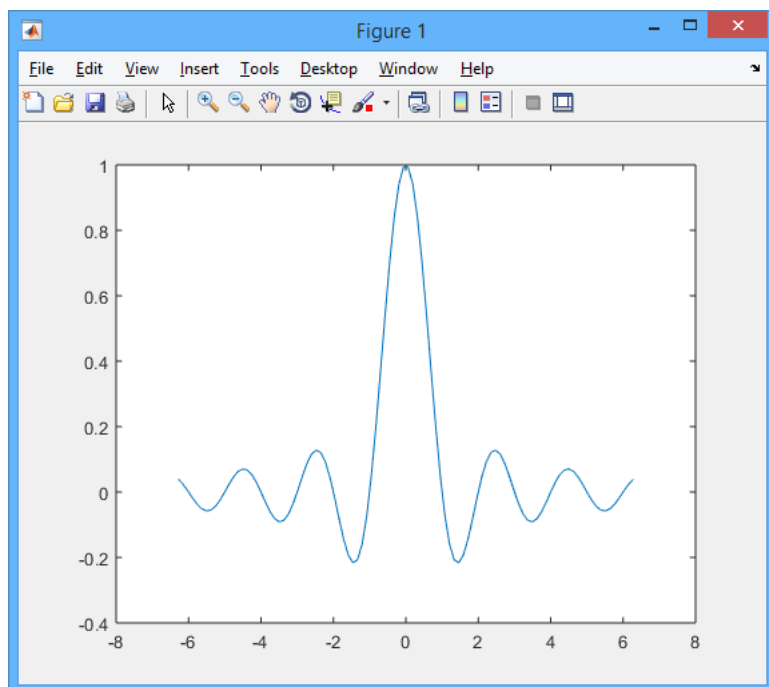
One of the more useful classes to subclass is the `handle` class, provided by Matlab. Unlike many other programming languages, creating an object `x = PolarVector(3.0, 0.0)` and setting `a = x` will assign `a` an entirely new value from `x`. This means that setting values in `a` will have no bearing on activities in `x`. In some contexts, this is a useful tool, but in others, particularly those where there are methods that make changes to values in the class, this is counterintuitive. Adding `< handle` to the `classdef` line will change it to this expected result.

Applications

Plotting functions

As a math resource, Matlab should easily be able to handle plotting a function:

```
X = linspace(0,2*pi);
plot(X, sinc(X));
```



`linspace` is a useful function in these contexts. It takes a range and produces, by default, 100 points, starting and ending at given arguments. In addition, a third argument may be provided for the number of points, when 100 isn't reasonable. Many arguments to plot can be found [here](#). Many **other kinds of plots** exist that give other ways of visualizing data more effectively. When a plot is called, it returns a handle that may be used to modify what appears.

Conway's Game of Life

Conway's Game of Life is an example of a complex system that evolves from very simple rules. Try copying and pasting the following class into `GameOfLife.m`:

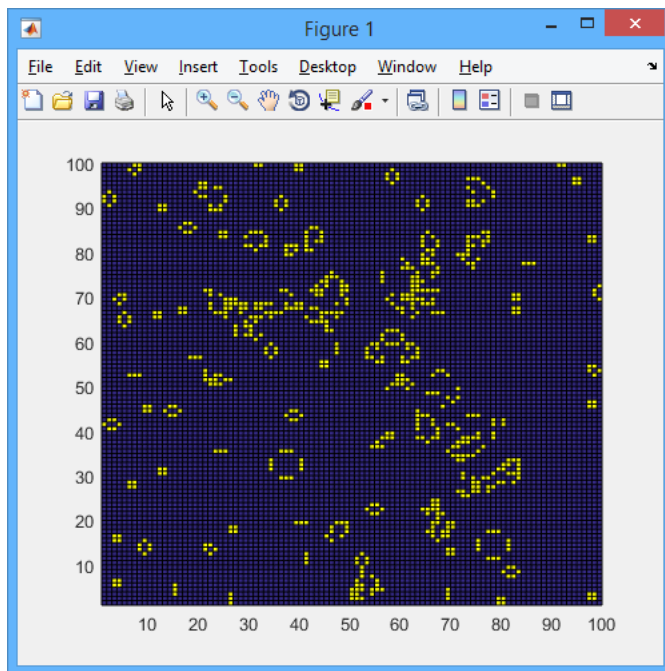
```
classdef GameOfLife < handle
    properties (Access = protected)
        grid;
        speed;
    end
    properties (Constant, Access = protected)
        %This is a filter that is used on each point to add up the
        %surrounding points
        addNeighborsFilter = [1.0 1.0 1.0;1.0 0.0 1.0;1.0 1.0 1.0];
    end
    methods
        function obj = GameOfLife(density, size, speed)
            %rand returns a random array of dimensions size * size. This is
            %filtered by the density, meaning more 1s appear if the density
            %is higher. Convultions require doubles, so it is then changed
            %to a double.
            obj.grid = double(rand(size,size)<density);
            %The object's speed determines the length of a pause between
            %frames. The higher the speed, the lower the pause
            obj.speed = 1/speed;
            obj.start();
        end
        function start(obj)
            %Get a handle to the plot we create
            handle = pcolor(obj.grid);
            %The ishandle will return false when the window is destroyed
            while ishandle(handle)
                %Increment to the next frame
                obj.increment();
                %Set the data to be displayed to the new grid
            end
        end
    end
end
```

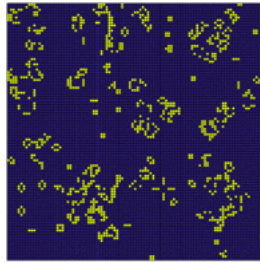
```

        handle.CData = obj.grid;
        %refreshdata tells the handle that new information should
        %be plotted
        refreshdata;
        %
        pause(obj.speed);
    end
end
end
methods (Access = protected)
    %Progresses the grid from one frame to the next, by the rules of
    %Conway's Game of Life
    function increment(obj)
        %Convolutions can be used to find the number of adjacent live
        %neighbors around each point
        sumOfNeighbors = conv2(obj.grid, GameOfLife.addNeighborsFilter, 'same');
        %Filtering can be done with element-wise multiplication to get
        %results for live neighbors only
        liveSumOfNeighbors = sumOfNeighbors.*obj.grid;
        %Similar filtering can be done for dead neighbors
        deadSumOfNeighbors = sumOfNeighbors-liveSumOfNeighbors;
        %A point on a grid lives on if it is dead and has three
        %neighbors or if it is alive and has two or three
        obj.grid=(deadSumOfNeighbors==3)+((1<liveSumOfNeighbors).*(liveSumOfNeighb
    end
end
end
-----
GameOfLife(.4, 100, 20);

```

When an instance of the class is instantiated, something like this will appear:





Conclusion

Matlab provides a considerable number of resources for a math development environment while still maintaining relative speed and fine tunability. This gives it an important and powerful role in many fields. Learning Matlab can be a big step as a first programming language since its syntax style is markedly similar, but simpler, to many other languages. Hopefully, this tutorial provided enough of a resource to assist readers in being confident using Matlab for the variety of uses it was intended for!

Filed Under: [Uncategorized](#)

[Return to top of page](#)



Copyright © 2019 · Udemy, Inc. · Built on the Genesis Framework