MATLABQUICK VOLUME 1

Learning MATLAB® is easy once a few basic concepts are introduced and used to drive the overall programing experience. This ebook focuses on teaching MATLAB® skills by introducing time series data using built-in math functions for Gaussian noise randn(), and the MATLAB® histogram() and plot() functions.

Scott W. Teare

Name: Scott W. Teare
Title: MATLABQuick, Volume 1.
Subject: Programming using MATLAB®

Copyright © 2017 by Scott W. Teare
Email: scott.teare@gmail.com
All rights are reserved. Permission is granted to copy and/or distribute this document in whole, provided that the work remains unaltered.

This work is provided for educational purposes only. Every effort has been made to provide accurate information hereon, but neither the publisher nor the author is responsible for the accuracy of the information or for any outcome from its use.

1.0 Overview

MATLABQuick was written for the reader who wants to obtain basic MATLAB® skills as quickly as possible so that they can transition these skills to their own projects. Often, the most difficult component of learning to program is finding something to program that is sufficiently interesting and intricate that it is worth doing, yet not so complicated that it cannot be completed in a reasonable amount of time. The topics we will be programing are related to signals, that is, creating a time varying function, in this case a noise signal. This has the advantage of introducing some basic math functions, data organization and visualization tools. While there is much more to this topic than we will be covering here, we will get a good start on it.

2.0 MATLAB® Basics

Programming in MATLAB® involves entering lists of commands into a text editor and saving them as text files known as scripts. These scripts contain commands that can be used to make loops, conditional decisions, mathematical equations and much more allowing very complicated programs to be developed. Many users find that the visualization tools in MATLAB® make it well worth the effort to learn.

MATLAB® scripts have a file extension of ".m" and the file name must conform to a set of rules. Rather than worry about the details of these rules we will use of the naming convention of "my<Name>.m" for our files. This will prevent us getting bogged down in rules by using things that will work. If you want more details, there is extensive MATLAB® help at www.mathworks.com.

There is no requirement to define a format for writing programs in MATLAB®; however, having a standard approach to writing programs will make them easier to read. The approach used here is to have the program run from a script and the script calls functions as needed to complete the task. The form of a MATLAB® script file that we will be using is show below as:

```
%% myProg.m
% SWT-6-12-17
% Basic format for a script file

%% Housekeeping
clear;clc; % Clears variables and command window

%% Stuff

%% End
```

This simple program structure represents the template for the programs we will be writing. In the area between % Stuff and % End is the program area where the commands that are to be run will be place. Notice that there are a number of “%” signs and these are used to define comments. Two percentage signs “%%” are used to indicate a section comment and a single sign shows a lesser comment. These are not executed by MATLAB®, they are there for the reader so comment often to make your code understandable.

The program area contains the commands that are to be executed to perform specific tasks. These can be functions that are part of MATLAB® such as trigonometry functions like sin(), cos(), or any of a wide range functions to provide specific functionality such as a random number generator like randn(). In addition users can develop their own functions and use them within the main program. One of the strengths of MATLAB® is its visualization tools and here we will look at two such functions, plot() and histogram().

Functions, whether built-in or custom, are separate program blocks that can be called as needed by the main script. Functions are reusable code that can accept variables and return values to the main program. Functions tend to make programs more readable and maintainable over time by isolating specific sections of code. Such functions can also be stored in their own files making it simpler to edit and revise this code as you are dealing with smaller blocks. However, custom functions will be left as a topic for another day.

3.0 First Program

The first program we will construct will be as straight forward as possible; we will construct a constant signal with time and plot it. The equation of a straight line is **y=mx+b,** in this case there will be no slope so the equation is just y = b, and we will construct a time base for it over 1 second. The code is shown in Table 1, with the lines that contain commands numbered to differentiate them from comments.

Table 1. The code for myFirst.m program.

```
      %% myFirst.m
      % SWT-6-13-17
      % Plotting a constant signal with time

      %% Housekeeping
 1    clear;clc; % Clears variables and command window

      %% Calculations
 2    maxCount = 10; % maximum number of points
 3    maxTime = 1; % maximum time
 4    b = 1; % signal value
 5    count = 1: maxCount; % make an array of points
 6    time = count*maxTime/maxCount; % scale time to array of points
 7    y(1, 1:maxCount) = b; % make the output signal

      %% Plotting
 8    figure(1); plot(time, y, 'ko'); % plot the function
 9    axis([0 1.1 0 2]); % set axis manually
10    xlabel('Time (s)'); % some labels
11    ylabel('Signal');

      %% End
```

The result of our first program is not overly spectacular, but it is a start and it introduces a number of the basic operations that we need to become familiar with when writing programs. The result of the code is a straight line with the data points shown as circles as shown in Fig. 1. There is a lot of programing technique that we can explore in this program and we will follow the line numbers to understand them.

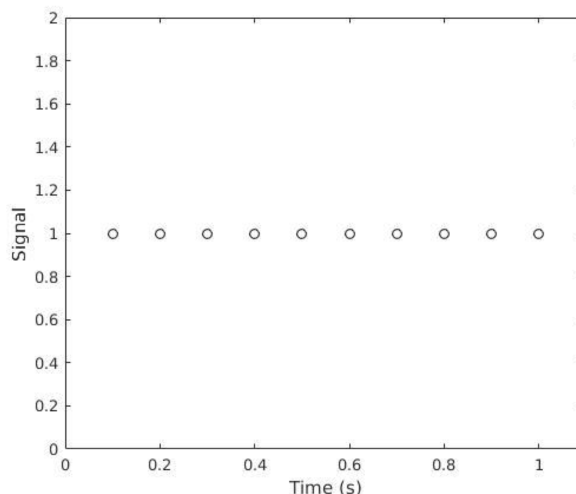

Figure 1. The plot from myFirst.m.

A line by line description of the code used in myFirst.m will help each of the different pieces of code make sense.

Line 1: The clear() and clc() functions are used to clear the variables from MATLAB® and to clear the Command Window. This provides a fresh start for the script. Notice that it is not necessary to include the parenthesis that commonly identifies a function.

Lines 2-4: These three lines define variables that a user may choose to change for any given execution of the program. These variables are 'maxCount' which is the number of points to be used in calculation; 'maxTime' which is the maximum time to be considered; and 'b' is the value to be used for the signal which in this case is a constant.

Line 5: This statement creates a vector, that is, a list of numbers, that begins with the value 1 and increments by one until the value in 'maxCount' is reached. The length of the vector is set by 'maxCount'.

Line 6: The vector 'count' is scale by the ratio of 'maxTime' to 'maxCount' to create an array of vectors of length 'maxCount' scaled to give the 'time'. Individual values of the time can be extracted from the vector using 'time(12)' where 12 is the

index value of interest. The dimensions of the vector can be obtained using the command 'size(count)' which will return the values 1 and 5 indicating a single row and 10 columns.

Line 7: Here is a bit more complicated example of forming a vector. If the statement was just 'y=b' the value of 'y' would have just a single value. However, we want to have an individual value of 'y' for each index 'count'. Thus we write the vector 'y' such that for a single row, and each of the columns the value of 'b' is generated.

Line 8: This is the starting point where the plot is generated. First a figure(), that is a plotting area is created, followed by the type of graph to be created in this case a plot() which is an x-y plot. The plot() function takes as inputs, in order, the x-axis then y-axis values followed by control characters for the data indicators and the color, 'ko' indicates the color black (k) and to use circles as data points (o).

Line 9: The axis() function controls the range of the numbers on the axes. The numbers in the square brackets correspond to [xstart xend ystart yend].

Lines 10 &11: The xlabel() and ylabel() functions provide the ability to label the axes.

In our example, the code ends with the %% End statement which is a comment and not required; however, can be valuable in providing clarity for the reader.

4.0 Adding Noise to a Signal

Now we will add a little complexity to our program by introducing the idea of random numbers which provide a noise component to the value of 'b'. We will do this by adding new code following line 7. This new code is: **noise = randn(size(y)); sig = y+noise;** and to replace the 'y' in Line 8 with sig. Now when we run the program we see that the straight line is no longer, the points show random behavior and each time you run the program the line will have a different shape. One instance of this is shown in Fig 2.

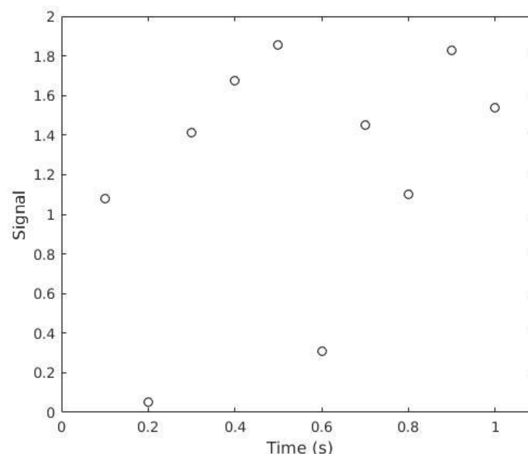

Figure 2. The effect of adding random noise to the constant value 'b'.

At this point we have introduced some new functions and have generated some plots while getting a good feel for how to program in MATLAB®. MATLAB® has many powerful built-in functions and we will use one of them to look at a more detailed problem related to analyzing noise.

5.0 Visualization and Analysis

The randn() function generates random numbers that are normally distributed also known as a Gaussian distribution. One of the best tools to appreciate what is meant by these ideas is to use the histogram() function and the related function histcounts() to both see and analyze the data. The code for this example is shown below.

```
%% myHist.m
% SWT-6-16-17
%
% Uses randn() to create data set and histogram() scaled gaussian
% function to analyze.

%% Housekeeping
clear; clc;

%% Parameters
maxCount = 1e6; % number of points
maxTime = 1; % time range
sigma = 0.1;  % amplitude of noise
binsPerSigma = 10; % number of bins in a standard deviation
rangeSigma = 5; % number of standard deviations to view

%% Create histogram bin vector
stop = rangeSigma*sigma; % stop bin
```

```
start = -stop;         % start bin
step = sigma/binsPerSigma; % bin size

binVector = start:step:stop; % bin vector

%% Create time sequence vector of noise
count = 1: maxCount;
t = maxTime/maxCount*count; % time based on index counts
noise = sigma*randn(maxCount,1) ; % create line + noise vector

%% Find mean and standard deviation
dataMean = mean(noise);  % calculate mean of time series
dataSTD = std(noise);    % calculate standard deviation of time series

%% Create envelope equation and scale to histogram
gaussianEqn = (1/(dataSTD*sqrt(2*pi)))* ...
              exp(-0.5*((binVector-dataMean)/dataSTD).^2);

histSum = sum(histcounts(noise, binVector)); % area under histogram
eqnSum = sum(gaussianEqn);       % area under gaussian
scalingFactor = histSum/eqnSum; % compare gaussian to histogram
scaledEqn = gaussianEqn*scalingFactor; % match gaussian to histogram

%% Plot
% plot of time series data
figure(1); plot(t, noise);
axis([0 maxTime start stop]);
xlabel('Time(s)'); ylabel('Signal');
set(gca,'FontSize',9,'FontWeight','Bold','LineWidth',1.5)
saveas(gca,'Fig_a','jpg');

% Plot of histogram and gaussian function
figure(2); h = histogram(noise, binVector); hold on;
plot(binVector,scaledEqn,'LineWidth', 2); hold off;
xlabel('Bins'); ylabel('Counts');
axis([start stop 0 1.1*max(scaledEqn)]);
set(gca,'FontSize',9,'FontWeight','Bold','LineWidth',1.5)
saveas(gca,'Fig_b','jpg');

% Visualization of relationship of time series to distribution
figure(3); subplot(1,2,1); plot(t, noise);
set(gca,'FontSize',9,'FontWeight','Bold','LineWidth',1.5)
xlabel('Time(s)'); ylabel('Signal');
subplot(1,2, 2); h = histogram(noise,binVector); view(90,90); hold on;
plot(binVector,scaledEqn,'LineWidth', 2); hold off;
xlabel('Bins'); ylabel('Counts');
set(gca,'FontSize',9,'FontWeight','Bold','LineWidth',1.5)
saveas(gca,'Fig_c','jpg');
```

The code has expanded in size considerably, but, much of the code we have seen before and there are a number of reused bits of code throughout the program. Before detailing how the code works, let’s consider what the code does as shown in Fig. 3. These plots were created using the last section of code and show side by side the time series signal and the histogram that describes the variation in the signal. The histogram on the right was formed by dividing the amplitude of the time series data on left into bins and counting the number of points in each bin. The result is

the normal or Gaussian distribution to describe the frequency of the points landing in each bin. This plot combines a number of useful details in making plots which will be considered in detail. Two other plots are created, and are used to show the same information contained in Fig 3, just separated out.

The histogram representation of the data is used to show the distribution of the time series data, it contain the same information as the time series plot, without the time. This is a useful way to look at the distribution of data points in the time series data as well as providing a first visualization function to be considered as histogram() provides both numerical values and a plotting tool.

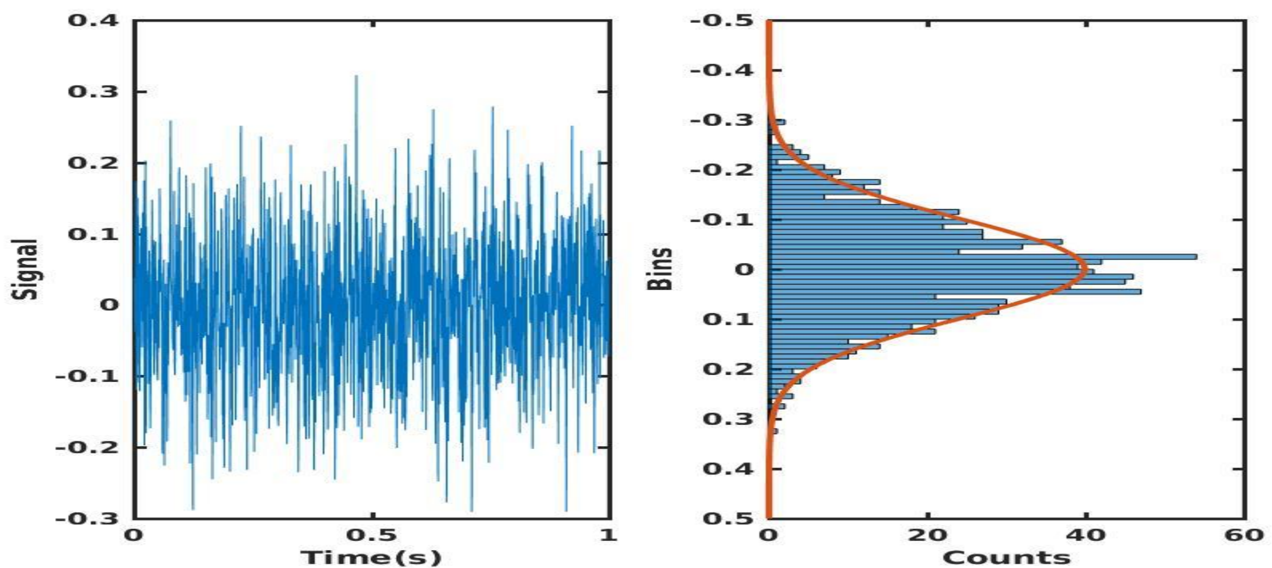

Figure 3. Histogram and envelope function for 1 thousand points showing the relationship between the histogram and the time series data.

In the coding area, above the %% Plot section, the code is just an expanded version of what we have already considered in the first program. The new information relates to the use of the histogram() and histcounts() functions so we will be taking a detailed look at these two related functions.

The first to review is the histcounts() function which provides the counts for each bin generating the histogram without actually plotting the results. The inputs to the

function are the time series data and the vector that describes the bins from their start to finish and the bin step size. This was accomplished by creating the binVector which has its own section of code.

In the histogram() function, the inputs are the same as the histcounts() function, however this function also includes the generation of a plot. The histogram plot is complemented by generating a line function that describes the shape of a perfect normal distribution or Gaussian. In Fig 4, there is little difference between the histogram bin heights and the Gaussian curve as a million data points were plotted, however, this is not as apparent when only 1000 data points are used in the time series data and the resulting histogram plot is less well defined as seen in Fig 5.

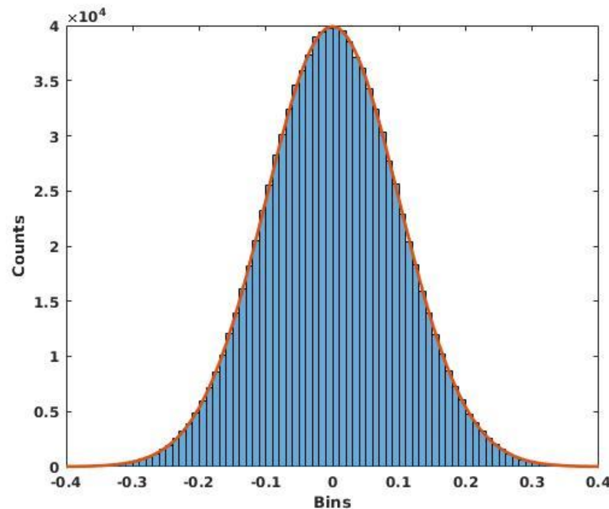

Figure 4. Histogram and envelope function for 1 million points.

The quality of the histogram plot depends very much on the size of the bins and the number of data points. When lower numbers of points are available it is common to reduce the number of bins; however, when too few bins are used the data looks more like a set of blocks rather than a distribution which can reduce the usefulness of the representation.

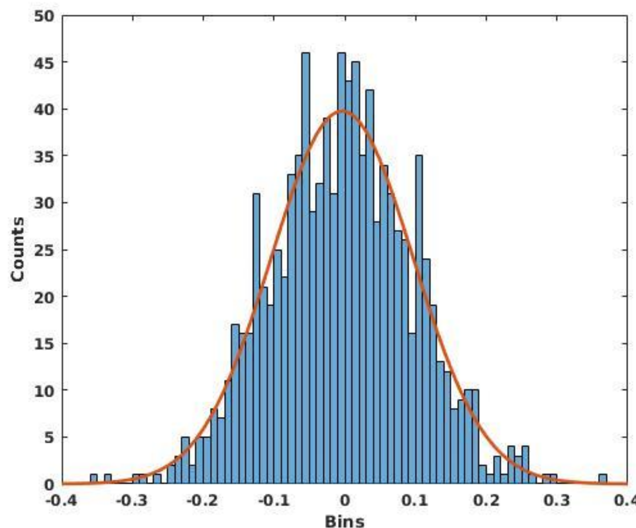

Figure 5. Histogram and envelope function for 1 thousand points.

The area of the code where there are considerable changes is in the plotting section. The individual plots are identified using the figure() function and a number is assigned to each. The axis(), xlabel(), ylabel() commands are straight forward, the label functions take strings that are plotted and the axis() function is used to set the extent of the numbering on each axis according to numbers entered as axis([xstart xstop ystart ystop]). The set() command is fairly complicated, but it is fair to say that nearly everything in MATLAB® is controllable by the user and the set command here is used to format the look of the plots. The set() command takes many inputs and the version used above has set(gca,'FontSize',9,'FontWeight','Bold','LineWidth',1.5), where the gca keyword identifies the current axes of a graph being affected. The rest is straight forward, there are other key words entered as strings followed by other strings or numbers that can be adjusted. There are many other adjustable components of graphs.

The final unique command is subplot(). This is used so that we can have multiple plots in a single figure. The numbers refer to the number and position of the plots in the figure window. The form of the numbers in the subplot(a, b, c) function refer to a = the number rows of plots to be included, b = the number of columns to be included, and c the position number. Thus subplot(1, 2, 2) refers to their being two plots, located side by side and in this case the right most position. The plot positions in the figure are numbered increasing monotonically left to right, top to bottom.

6.0 Summary

There are many built-in functions in MATLAB® and there is the ability to write custom functions. These are very effective tools for developing complex code and retaining its readability and maintainability! We have worked through some very commonly encountered MATLAB® components and programing. While there are many more functions available for use, they can be understood and used from the skills we have developed here.

7.0 Bibliography

1. Hanselman, D., Littlefield, B., ***Mastering MATLAB® 7***, Prentice-Hall (2011).

2. Moore, H., ***MATLAB® for Engineers, 4th Edition***, Prentice-Hall (2014).

3. Pratap, R., ***Getting Started with MATLAB® 7***, A quick introduction to scientists and engineers. Oxford University Press (2006).

4. Teare, S.W., ***Optics Using MATLAB®***, SPIE Press (2017).

5. Teare, S.W., ***Practical Electronics for Optical Design and Engineering***, SPIE Press (2016).

.