Signal Processing using Matlab

Lesson 1

Overview of Matlab Syntax and Programming

1.1 Introduction

In this lesson, we will take an overview of the Matlab language syntax and programming. In the first section, we will take an overview of Matlab syntax. We will start by explaining how we could define arrays in Matlab. Afterwards, we will study Matlab's basic numeric operations (arithmetic, relational, and logical operations). We will then proceed to Matlab's basic array operations (concatenation, indexing, and transposing). We will study an important notation called the colon notation. In the last section of this lesson, we will study some important Matlab functions that will be used in the following lessons. In the second section, we will discuss M-files and programming statements.

Section I: Syntax Basics

1.2 Defining Arrays

The flexibility of Matlab in solving engineering problems comes from its ability to operate with arrays in an easier way than the other programming languages. The first step in learning Matlab is to know how to define arrays. In (1.1), we introduce some new terminology related to Matlab arrays. In (1.2), we learn how to define arrays.

1.2.1 Scalars, Vectors, Arrays and Dimensioning

An array is a data unit that consists of many elements. A 4-by-3 array of numbers is a data unit consisting of 4 rows by 3 columns of numbers. Another name for an array is a matrix. A special case of an array is the vector. A vector is a 1-by-N or an N-by-1 array, i.e. it is an array with only one row or only one column. Another special case of an array is the scalar. A scalar is an array with only one element. Its dimensions are said to be 1-by-1. The following example shows an array A, a row vector VR, a column vector VC and a scalar S.

$$\text{A} = \begin{bmatrix} 4 & 2 & 3 & 7 \\ 5 & 8 & 8 & 1 \\ 2 & 1 & 7 & 5 \\ 9 & 3 & 1 & 2 \end{bmatrix} \qquad \text{VR} = \begin{bmatrix} 3 & 0 & -1 \end{bmatrix} \qquad \text{VC} = \begin{bmatrix} 2 \\ 6 \\ 3 \end{bmatrix} \qquad \text{S} = 2.7$$

1.2.2 Defining Arrays, Vectors and Scalars

Defining arrays in Matlab is a general technique out of which we can derive the special cases of defining vectors and scalars. There are four rules for defining arrays in Matlab:

1. Elements on the same row must be separated by white-spaces or commas.
2. A semicolon, wherever it appears, causes a transition to the next row of elements.
3. The definition requires the use of square brackets " [ ] ".
4. The defined array must be rectangular, i.e. the number of elements in all rows must be the same.

The instructions that create the variables in the previous example are as follows:

```
>>A=[4 2 3 7 ; 5 8 8 1 ; 2 1 7 5 ; 9 3 1 2]
>>VR=[3 0 -1]
>>VC=[2 ; 6 ; 3]
>>S=[2.7] %(or >>S=2.7)
```

Note that scalars can be defined without square brackets. This is an exception to the rule to facilitate working with scalars because they occur very frequently.

1.3 Arithmetic, Relational and Logical Operations

1.3.1 Arithmetic Operations

The following table lists the available arithmetic operators, and a brief description of what each of these operators does.

	Exampl e	***Description***
+	A+B	Adds B to A element by element
-	A-B	Subtracts B from A element by element
.*	A.*B	Multiplies A by B element by element
*****	A*B	Matrix multiplies A by B
./	A./B	Divides A by B element by element
/	A/B	Matrix multiplies A by the inverse of B
.^	A.^3	Cubes A element by element, equivalent to A.*A.*A
^	A^3	Equivalent to matrix multiplying A by A by A (A*A*A)

Here are some examples that operate on arrays A, B, C and D shown below.

$$A=\begin{bmatrix}1&2&3\\4&5&6\\7&5&10\end{bmatrix}\quad B=\begin{bmatrix}3&3\\3&3\\3&3\end{bmatrix}\quad C=\begin{bmatrix}1&1&1\\1&1&1\\1&1&1\end{bmatrix}\quad D=\begin{bmatrix}2&3\\2&3\\2&3\end{bmatrix}$$

```
>> X=A+C

X =

   2    3    4
   5    6    7
   8    6   11

>> X=C-A

X =

   0   -1   -2
  -3   -4   -5
  -6   -4   -9
```

```
>> X=B.*D

X =

   6    9
   6    9
   6    9

>> X=A*C

X =

   6    6    6
  15   15   15
  22   22   22
```

```
>> X=B./D

X =

   1.5000   1.0000
   1.5000   1.0000
   1.5000   1.0000

>> X=C/A

X =

  -0.333   0.333       0
  -0.333   0.333       0
  -0.333   0.333       0
```

```
>> X=A^2

X =

  30   27   45
  66   63  102
  97   89  151

>> X=A.^2

X =

   1    4    9
  16   25   36
  49   25  100
```

1.3.2 Relational Tests

The following table lists the available relational operators, and a brief description of what each of these operators does.

	Example	***Description***
>	A>B	Tests if the elements of A are greater than the elements of B
>=	A>=B	Tests if the elements of A are greater than or equal to the elements of B
<	A<B	Tests if the elements of A are smaller than the elements of B
<=	A<=B	Tests if the elements of A are smaller than or equal to the elements of B
==	A==B	Tests if the elements of A are equal to the elements of B
~=	A~=B	Tests if the elements of A are not equal to the elements of B

Here are some examples that operate on arrays A, B, C and D shown below.

$$A=\begin{bmatrix}1&2&3\\4&5&6\\7&5&10\end{bmatrix}\qquad B=\begin{bmatrix}3&3\\3&3\\3&3\end{bmatrix}\qquad C=\begin{bmatrix}1&1&1\\1&1&1\\1&1&1\end{bmatrix}\qquad D=\begin{bmatrix}2&3\\2&3\\2&3\end{bmatrix}$$

```
>> X=A>C

X =

     0     1     1
     1     1     1
     1     1     1

>> X=A>=C

X =

     1     1     1
     1     1     1
     1     1     1
```

```
>> X=B<D

  X =

       0     0
       0     0
       0     0

  >> X=B<=D

  X =

       0     1
       0     1
     0     1
```

```
>> X=B==D

X =

     0     1
     0     1
     0     1

>> X=B~=D

X =

     1     0
     1     0
     1     0
```

Note that relational operations are element-by-element operations.

1.3.3 Logical Operations

The following table lists the available logical operators, and a brief description of what each of these operators does.

	Example	***Description***
&	A&B	Performs logical ANDing of elements of A and B
|	A|B	Performs logical ORing of elements of A and B

~	~A	Complements the logic of elements of A

Here are some examples that operate on arrays A and B shown below.

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 5 & 6 \\ 7 & 0 & 10 \end{bmatrix} \qquad B = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

```
>> X=A&B

X =

     0     1     0
     0     0     1
     0     0     0
```

```
>> X=A|B

X =

     1     1     1
     1     1     1
     1     1     1
```

```
>> X=~A

X =

     0     0     0
     1     0     0
     0     1     0
```

Note that Matlab treats numbers from the logical point of view as zero and non-zero. A non-zero number is equivalent to logic 1. Also note that logical operations are element-by-element operations.

1.3.4 Size Rules

The following table shows the arithmetic, relational and logical operations and the size rule that must be satisfied by their operands.

Operations	Size Rule
Addition and Subtraction	All operands must have the same size.
Array Multiplication, Division and Power	All operands must have the same size.
Matrix Multiplication	Inner matrix dimensions must agree.
Matrix Division and Power	Matrices must be square.
All Relational Operations	All operands must have the same size.
All Logical Operations	All operands must have the same size.

1.3.5 Scalar Expansion

Matlab has an exception to the size rule of array operations when it comes to operations involving scalars. An operation involving an array and a scalar will cause Matlab to expand the scalar to the size of the array. Here are examples that operate on the array A shown below.

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 5 & 6 \\ 7 & 0 & 10 \end{bmatrix}$$

```
>> X=A-7

X =

    -6    -5    -4
    -7    -2    -1
     0    -7     3
```

```
>> X=2*A

X =

     2     4     6
     0    10    12
    14     0    20
```

```
>> X=A>=5

X =

     0     0     0
     0     1     1
     1     0     1
```

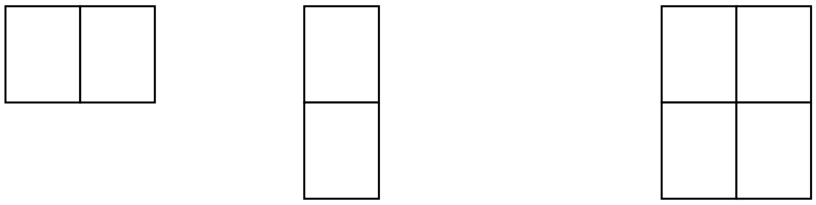
1.3.6 Combining Arithmetic, Relational and Logical Operations

Part of the flexibility of Matlab comes from the ability of combining expressions. The following expression will execute successfully provided that A and B have the same size.

```
>> C=(2*A-B)*~((2*A-B)>=B)
```

1.4 Concatenation

To concatenate two pieces of paper is to join them end-to-end. You can concatenate two pieces of paper horizontally or vertically. You can also concatenate more than two pieces of paper as shown below.



1.4.1 Concatenation Syntax

You can concatenate arrays in Matlab just as you concatenate pieces of paper. Here are the instructions to carry out horizontal and vertical concatenation. Note that the syntax of concatenation is similar to the syntax of array creation. This is because array creation is actually a concatenation of single numbers.

$$A=\begin{bmatrix}1 & 1\\1 & 1\end{bmatrix} \qquad B=\begin{bmatrix}2 & 2\\2 & 2\end{bmatrix}$$

```
>> C=[A B]

C =

    1    1    2    2
    1    1    2    2

>> C=[B;A]

C =

    2    2
    2    2
    1    1
    1    1

>> C=[A B;B A]

C =

    1    1    2    2
    1    1    2    2
    2    2    1    1
    2    2    1    1
```

1.4.2 Concatenation Size Rules

Since any array must be rectangular, there are size rules that govern concatenation. Arrays concatenated horizontally must have the same number of rows, and arrays concatenated vertically must have the same number of columns.

1.5 Indexing into Vectors

"Indexing into an array" means dealing with part of the array, either reading from this part or writing to it. The easiest array we can start learning how to index into is the vector. We will first learn how to read single and multiple elements from vectors, then we will learn how to write single or multiple elements into vectors.

1.5.1 Single Element Indexing into Vectors on the Right Hand Side

Suppose we define the vector V as shown below. We then want to read the third element in V. The instruction V(3) obtains the third element in V. The number between round brackets is called the "index". The index may take any integer value from 1 to the length of V. The index may also be end, which corresponds to the last element in the vector.

```
>> V=[7 9 6 2]

V =

     7     9     6     2

>> A=V(3)

A =

     6

>> A=V(end)

A =

     2
```

1.5.2 Multiple Element Indexing into Vectors on the Right Hand Side

If the index is a vector instead of a scalar, Matlab returns multiple elements from the vector. The following instructions provide examples that index into the same vector V used in (4.1).

```
>> A=V([1 3])

A =

     7     6

>> B=V([end-2 end])

B =

     9     2
```

Note that the length of the result is the same as the length of the index.

1.5.3 Single Element Indexing into Vectors on the Left Hand Side

We can use indexing on the left hand side to store a value in a specific element of a vector. Look at the following example.

```
>> V=[7 9 6 2]

V =

     7     9     6     2

>> V(end)=4

V =

     7     9     6     4
```

1.5.4 Multiple Element Indexing into Vectors on the Left Hand Side

We can also index on the left hand side using a vector index to store a vector of values in specific locations in a vector. Look at the following example:

```
>> V=[7 9 6 2]


V =


     7     9     6     2


>> V([1 end-1 end])=[4 1 2]


V =


     4     9     1     2
```

1.6 Indexing into 2-D Arrays

We have learnt how to index into vectors. Vectors are 1-D arrays where only one index per element is needed. In 2-D arrays, we will need a row index and a column index to specify the position of an element.

1.6.1 Single Element Indexing into 2-D Arrays on the Right Hand Side

The general syntax for indexing into a 2-D array is A(R,C) where A is the array name, R is the row index and C is the column index. If R and C are scalars, Matlab returns a single element from A.

```
>> A=magic(3)

A =

     8     1     6
     3     5     7
     4     9     2
```

```
>> A(3,1)

ans =

     4
```

1.6.2 Multiple Element Indexing into 2-D Arrays on the Right Hand Side

If R and C are vectors rather than scalars, Matlab returns multiple elements according to the rows and columns listed in the vector indices.

```
>> A=magic(3)

A =

     8     1     6
     3     5     7
     4     9     2

>> A([1 2],1)

ans =

     8
     3

>> A([1 3],[1 2])

ans =

     8     1
     4     9
```

The colon (:) may be used as a special index meaning "all". If it appears as a row index it means all rows. If it appears as a column index it means all columns.

```
>> A([1 end],:)

ans =

     8     1     6
     4     9     2
```

1.6.3 Single Element Indexing into 2-D Arrays on the Left Hand Side

We can use indexing on the left hand side to write a value into an element in an array specificied by its row and column indices. Here is an example.

```
>> A=magic(3)

A =

     8     1     6
     3     5     7
     4     9     2
```

```
>> A(end,2)=0

A =

     8     1     6
     3     5     7
     4     0     2
```

1.6.4 Multiple Element Indexing into 2-D Arrays on the Left Hand Side

```
>> A=magic(3)

A =

     8     1     6
     3     5     7
     4     9     2

>> A(1,:)=[4 4 4]

A =

     4     4     4
     3     5     7
     4     9     2

>> A([2 3],:)=0

A =

     4     4     4
     0     0     0
     0     0     0
```

The instruction A(1,:)=[4 4 4] demonstrates a size rule. The right hand side must have the same dimensions as the left hand side specification. The instruction A([2 3],:)=0 demonstrates the exception from this rule, which we called scalar expansion. Matlab understands that you want to write zeros in all elements of the second and third rows of A.

1.7 Transposing Arrays

The transpose operation turns the array such that its rows become columns and its columns become rows. Here is an example.

```
>> A=magic(4)

A =

    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1

>> A=A'
```

```
A =

    16     5     9     4
     2    11     7    14
     3    10     6    15
    13     8    12     1
```

1.8 The Colon Notation

The colon (:) has two uses in Matlab. We looked at its first use as an index meaning "all". Its second application is in a special notation called the colon notation. The colon notation is used to generate sequences of numbers given a start number, a step and a stopping limit. Look at the following examples.

```
>> A=4:2:12

A =

     4     6     8    10    12

>> A=8:-3:-5

A =

     8     5     2    -1    -4
```

If the step is equal to 1, it may be omitted from the colon notation, so that the notation appears as follows.

```
>> A=120:130

A =

   120   121   122   123   124   125   126   127   128   129   130
```

The colon notation produces a row vector by default. If you want a column vector, you will have to transpose the result.

1.9 Some Important Matlab Functions

The "zeros" and "ones" Functions

These two functions are responsible for generating arrays of zeros or ones. The syntax of these functions is as follows:

```
>>A=zeros(M,N);
>>B=5*ones(M,N);
```

The first instruction generates an array of zeros of dimensions M×N. The second instruction generates an array of dimensions M×N full of fives based on the "ones" function.

The "length" Function

The "length" function returns the length (number of elements) of a vector. Let V be a vector containing 4 elements. The following instruction would return 4.

```
>>L=length(V);
```

The "find" Function

The "find" function returns the indices of non-zero elements in its input array. This function is used mainly to search for elements of an array that satisfy a certain condition. To understand, consider the following instruction.

```
>>I=find(A>=2);
```

The input to the "find" function is the array resulting from the relational test (A>=2). This array has the same size of A, but consists entirely of ones and zeros. Ones appear in place of the elements of A that are greater than or equal to 2. Thus, in effect, the find function will return the indices of the elements of A that are greater than or equal to 2. This is a very important function in signal processing.

Section II: Programming

1.11 M-Files

M-files are files in which we can write and save Matlab code to be executed later. We can open a new M-file by writing "edit" in the command prompt, or by clicking on the "New" button in the Matlab button toolbar. You write the instructions in the M-file, and can then run the M-file by pressing the "Run" button in the M-file editor's toolbar, or by writing the M-file's name in the command window.

1.12 Programming Statements

In this section, we will discuss the four different statements used in programming: the "for" statement, the "while" statement, the 'if" statement, and the "switch" statement. The syntax for each statement is demonstrated below.

The "for" statement

```
for i=10:-2:5
    Code …
end
```

The "while" statement

```
while (a>2)&(b<7)
    Code …
end
```

The "if statement

```
if a<0
    Code 1 …
elseif a>2
    Code 2 …
else
    Code 3 …
end

if b<c
    Code …
end
```

```
if d<e
    Code 1 …
else
    Code …
end
```

The "switch" statement

```
switch x
    case 0
        Code 1 …
    case 3
        Code 2 …
end

switch y+z
    case 1
        Code 1 …
    case 2
        Code 2 …
    otherwise
        Code 3 …
end
```

[illegible]

e 2005, the salaries were raised by 10 units. In June 2006, the salaries were raised by 1[illegible]

[illegible]

[illegible] on to produc[illegible]

[illegible]

The v[illegible] each of the following operations. [illegible] down a sin[illegible]

)R[illegible]

[illegible]

[illegible]

536 … 9 4 1.

[illegible]

Use the appropriate indexing techniques to:

[illegible]

Signal Processing using Matlab

Lesson 2

Signal Basics

2.1 Introduction

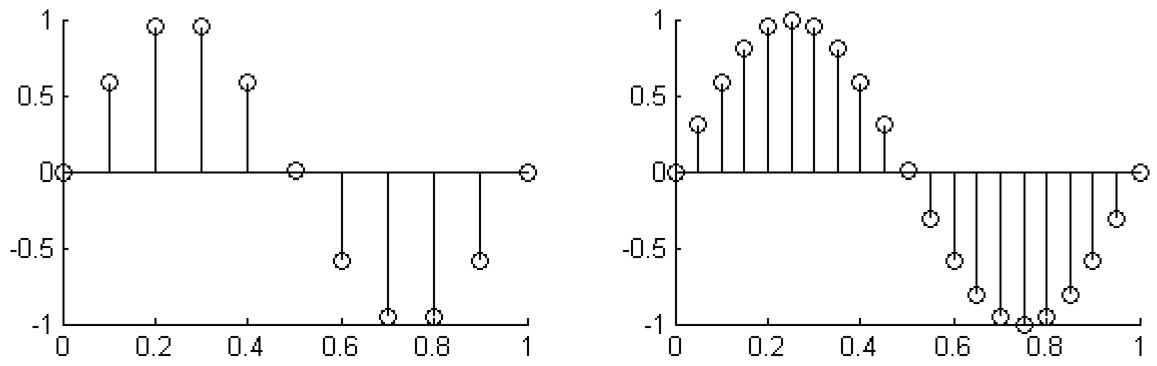
After you have finished studying the first lesson, you will have read that Matlab is the most powerful technical programming tool in signal processing and communications, but you don't yet know why. The reason is that Matlab has the power of dealing directly with large arrays in the same degree of ease of dealing with single numbers. This gives Matlab a very high preference to people studying signal processing and communications. Signals are usually large arrays of numbers. People want to operate on those signals as fast and simple as possible. Thus, Matlab is the best choice. This automatically makes Matlab the top choice for people studying communication, as communication consists completely of signal processing operations.

In this chapter, we will study the basics of using Matlab in signal processing. We will focus in this chapter on four fundamental issues: creating elementary signals, performing elementary operations on those signals, plotting signals, and computing signal statistics. Thus, we have divided this chapter into four sections. Section I deals with creating elementary signals such as sine waves, square waves, impulses … etc. Section II deals with performing elementary signal operations such as arithmetic operations, upsampling and downsampling, rectification … etc. Section III deals with plotting and signal visualization. Section IV deals with signal statistics and how they may be computed using Matlab.

Section I: Creating Elementary Signals

2.2 Understanding the Sample Time: The First Step in Creating any Signal

All signals in Matlab are discrete signals. Any discrete signal has an important attribute called the "sample time". The sample time is the time interval between two successive samples in the definition of the signal. To understand further, consider the following figure. In figure 2.1.a, the signal has a sample time of 0.1 seconds. In figure 2.1.b, the signal has a sample time of 0.05 seconds.



(a) (b)

Figure 2.1

Looking carefully at figure 2.1.a, you can see that the time interval between each sample and the next is 0.1. In figure 2.1.b, however, this value is 0.05. Thus, we have seen how changing the sample time will change the shape of the signal.

It seems usually that working with smaller sample times makes the representation of the signals more accurate. We will discuss this issue in the next point. However, for now we will say that this is true. Working with smaller sample times, however, may lead to a storage problem. To see how, suppose that we want to define a signal of sample time 1 ns over a time duration of 1 second. If each sample is defined using 8 bytes, the total storage size required to store this signal will be approximately 8 Gigabytes! Keep in mind that signals are stored in the RAM. Such signal would be very difficult to create, and will probably make your machine crash.

Thus, it is very important to determine the sample rate of the signal carefully. Determining the proper sample rate is a search for the best value on the following line.

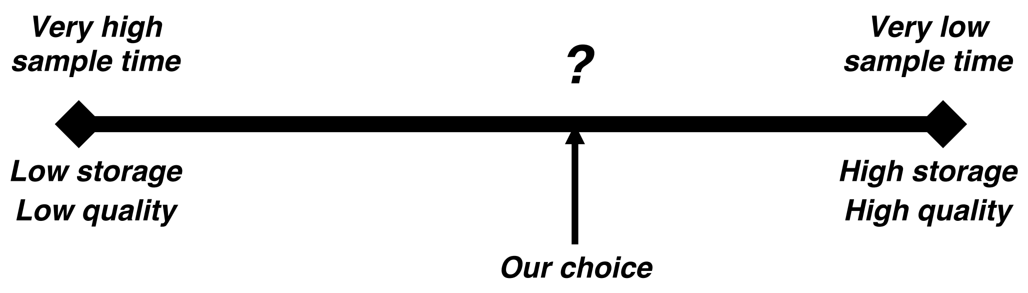


Figure 2.2

When we try to create any periodic signal, we will choose the sample time such that there are at least 10 samples per period. In our words, the sample rate will be at least 10 times the frequency of the signal.

Example:

For a triangular wave of frequency 250 Hz, the minimum sample rate will be 2500 Hz. This corresponds to a sample time of 400 μs. The sample time must be less than or equal to 400 μs.

2.3 Generating Signals

After we choose the sample time, we should start the actual generation steps. These steps differ according to the signal we want to generate. In general, any signal may be generated using either of two generation methods: the direct method or the indirect method.

The direct method is used if the signal may be generated using one or more functions that exist already in Matlab. Examples of functions that may be generated using the direct method are the trigonometric signals, logarithmic signals, exponential signals … etc. Accordingly, the class of signals that may be generated using the direct method extends to include all signals that may be generated by performing one or more operations on signals belonging to those groups. For instance, the following signal may be generated directly.

$$x(t) = (\cos(20\pi t + \pi/6) + \sin(30\pi t))\log(2t^2) \qquad t \geq 1$$

Periodic square pulse signals may also be generated directly using the function "square" as we shall explain shortly. Periodic triangular pulse signals, however, cannot be generated directly, because there is no Matlab function that generates a periodic triangular signal.

A signal that cannot be generated directly may be generated indirectly using basic Matlab operations, for example by generating small parts of the signal and joining them together using concatenation. The signal in figure 2.3 is an example of a signal that cannot be generated directly. An indirectly generated signal may be easily generated once it could be analyzed into a set of directly generable functions. The signal of figure 2.3 may be analyzed into 3 contiguous segments. The first segment is from -3 to -1 seconds. The second segment is from -1 to 1 second. The third segment is from 1 to 3 seconds. The first and third segments feature a DC signal. The middle segment features a cosinusoidal half-cycle with a DC offset. Each of these segments will be generated individually, and they will then be concatenated to give the signal shown.

In the rest of this section, we will proceed as follows. We will start by explaining the "linspace" function, because it is very important in the direct generation of many elementary signals such as trigonometric signals, exponential signals, logarithmic signals, square signals. We will then explain the Matlab functions responsible for the direct generation of all the elementary signals. Afterwards, we will discuss the techniques of indirect generation.

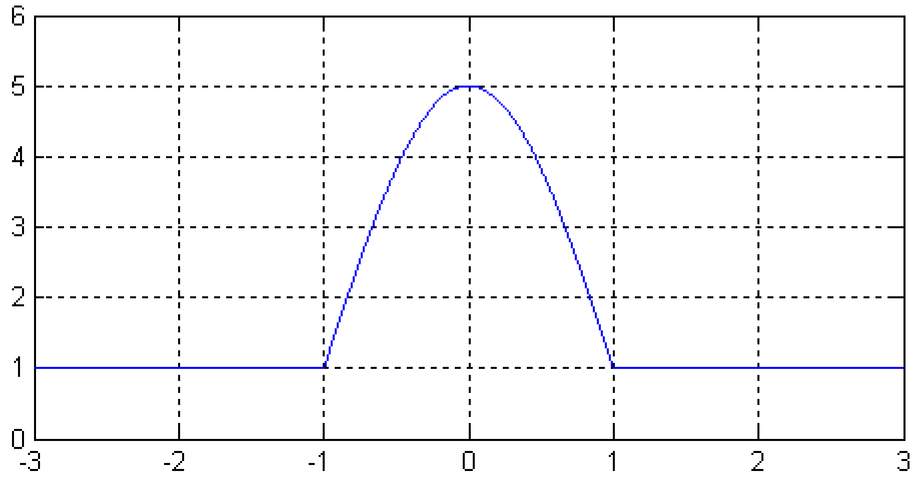


Figure 2.3

2.3.1 The "linspace" Function

All directly generable signals that rely on mathematical functions such as trigonometric functions, exponential functions, logarithmic functions, hyperbolic functions … etc need what we call a time base signal. To understand what the time base signal is and why it is required, keep in mind that you cannot compute the value of a sine without knowing the value of its input argument. If we are talking about sinusoidal signals, for instance, the signal is computed as follows:

$$x(t) = A\sin(2\pi ft + \theta)$$

A is the amplitude, *f* is the frequency, and θ is the phase shift in radians. In order to evaluate the value of the signal at any time instant, you must substitute with the value of *t* along with the values of the parameters.

When we want to generate a signal, we have to compute the signal at a set of values of *t* corresponding to the time interval of interest. Thus, *t* should be a vector not just a single value. This vector is called the time base signal. It contains the values of time at which the sine should be computed. The values of time in the time vector start and end at the limits of the interval in which we are interested. The difference between these two values is called the ***duration*** of the signal. The total number of points in the time base signal (N_t) satisfies the relation:

$$N_t = \text{duration} \times \text{sample rate}$$

Thus, if we want to generate a sine wave in the time interval from 2 to 10 seconds with a sample rate of 100 samples per second, the time base signal must consist of 800 points whose values increment uniformly from 2 to 10.

The linspace function is used in generating the time base signal. The linspace function has the syntax shown below:

$$t = linspace(a, b, N)$$

The first and second input arguments (*a* and *b*) are the initial and final values of the generated signal. The third argument (*N*) is the number of points of the signal. Thus, we must have $N=(a\text{-}b)f_s$. As an example, the following command could be used to generate a time base signal for a sine wave in the time interval from 3 to 5 seconds with a sample frequency of 100 samples per second.

```
>>t=linspace(3,5,(5-3)*100);
```

2.3.2 Direct Generation of Elementary Signals

In this subsection, we will discuss the commands through which we could generate elementary functions such as DC signals, ramp signals, trigonometric functions … etc.

DC Signal

A DC signal may be easily generated using the "zeros" or "ones" functions. The syntax of both functions is the same. This syntax is shown below.

$$x1 = zeros(1, N)$$
$$x2 = ones(1, N)$$

The "zeros" function generates a 1×N vector of zeros. The "ones" function generates a 1×N vector of ones. Using basic operations, we could generates DC signals of any other value, as the following examples demonstrate.

Examples

The first example below shows an instruction used to generate a DC signal of value -5 and sample rate 100 Hz in the time interval from -3 to 3 seconds using the "zeros" function. The second example shows an instruction used to generate a DC signal of value 17.3 and sample rate 20 Hz in the time interval from 0 to 4 seconds using the "ones" function. Note that generating a DC signal does not need a time base signal, because the signal is not a function in time.

```
>>x=zeros(1,600)-5;
>>x=17.3*ones(1,80);
```

Ramp Signal

A ramp signal is a signal whose value increases or decreases uniformly with time. A ramp signal has the following general form:

$$y(t) = at + b$$

The parameter *a* is called the slope, and the parameter *b* is called the intercept of the signal.

Example

In the following example, we are generating a ramp signal of sample rate 1000 Hz, slope -2 volts/second and intercept -3 volts in the time range from -2 seconds to 4 seconds.

```
>>t=linspace(-2,4,6*1000);
>>x=-2*t-3;
```

Polynomial Signals

Ramp signals are a special case of polynomial signals. In the following example, we will see how a polynomial signal of higher degree may be generated.

Example

The polynomial signal we want to generate a polynomial signal that follows the equation:

$$x(t) = 3t^3 - 11t^2 + 7$$

We want to generate this signal in the time range from 0 to 1 second with a sample rate of 1000 Hz. The following instructions generate this signal.

```
>>t=linspace(0,1,1000);
>>x=3*t.^3-11*t.^2+7;
```

Polynomial signals require a time base signal to be generated, because they are computed as functions in time.

Trigonometric, Hyperbolic, Exponential, and Logarithmic Signals

Now, we will discuss how to generate trigonometric, hyperbolic, exponential, and logarithmic signals. These groups of signals are generated similarly, so we will give one example from each group. However, we will list the functions that are used to generate the signals in each group.

Trigonometric functions

sin, cos, tan, asin, acos, atan, sec, csc, cot, asec, acsc, acot.

Hyperbolic functions

sinh, cosh, tanh, asinh, acosh, atanh, sech, csch, coth, asech, acsch, acoth.

Exponential functions
exp
Logarithmic functions
log, log10

Examples

In the following examples, we will generate the following six signals. All signals have an interval of interest from 1 to 5 seconds, and a sample rate of 150 Hz.

$$y_1(t) = 3\cos(4\pi t + \pi/4)$$
$$y_2(t) = 4\sinh(3.5t + 2)$$
$$y_3(t) = 4e^{-0.5t}$$
$$y_4(t) = 2^{-0.6t+0.3}$$
$$y_5(t) = \ln(3/t)$$
$$y_6(t) = \log(0.6t)$$

```
>>t=linspace(1,5,600);
>>y1=3*cos(4*pi*t+pi/4);
>>y2=4*sinh(3.5*t+2);
>>y3=4*exp(-0.5*t);
>>y4=2.^(-0.6*t+0.3);
>>y5=log(3./t);
>>y6=log10(0.6*t);
```

2.3.3 Indirect Generation

Indirect generation usually involves generating a signal by analyzing the signal into elementary segments that may be generated directly, and then conjoining the individual segments using concatenation. The following example shows the generation of the signal shown in figure 2.3.

The sample rate of the signal is 1000 Hz. The first segment is a DC signal of value 1 from -3 to -1 seconds. The second segment is half a cycle of a cosine wave of frequency 1/4 Hz (because half a cycle occupies 2 seconds), amplitude 4, phase shift 0, and DC offset 1. The third segment is an instant of the first segment. The following instructions may be used to generate this signal.

```
>>y1=ones(1,2000);
>>t=linspace(-1,1,2000);
>>y2=4*cos(2*pi*t/4);
>>y=[y1 y2 y1];
```

Section II: Elementary Operations on Signals

In this section, we will present a multitude of operations that may be carried out on signals. We will start by discussing arithmetic operations, relational tests, and logical operations. We will then discuss operations that change the sample time (called upsampling and downsampling). Afterwards, we will deal with operations on the independent variable such as time scaling and time shifting. We will then discuss a set of miscellaneous operations such as rectification, clipping … etc.

2.4 Arithmetic Operations

```
>>c=a+b;
```

This instruction adds two signals (a and b) together. Note that the two signals must have the same dimensions. The result is stored in c.

```
>>c=a-b;
```

This instruction subtracts the signal b from the signal a. Again, a and b must have the same dimensions.

```
>>c=a.*b;
```

This instruction multiplies the two signals together. The two signals must have the same dimensions. Note that the multiplication operator is (.*) not (*).

```
>>c=c=a./b;
```

This instruction divides the signal a by the signal b. The two signals must have the same dimensions. Note that the division operator is (./) not (/).

```
>>c=a.^b
```

This instruction raises the elements in a to the powers of elements in b.

```
>>c=a+3;
```

This instruction adds a DC offset to signal a. This is an exception to the size rule we mentioned before.

```
>>c=a-3;
```

This instruction subtracts a DC offset from signal a. This is an exception to the size rule we mentioned before.

```
>>c=0.3*a;
```

This instruction multiplies the signal a by a constant gain. We may use the (*) operator or the (.*) operator when one of the operand is a scalar.

```
>>c=c=a/0.4;
```

This instruction divides the signal a by the scalar 0.4. We may use the (./) or (/) operand when the divisor is a scalar..

```
>>c=a^2;
```

This instruction raises the elements in a to a fixed power. We may use the (^) or (.^) operand when the power is a scalar.

2.5 Relational Tests

```
>>c=a==b;
```

Returns 1 where a and b are equal, and 0 otherwise.

```
>>c=a==3;
```

Returns 1 where a equals 3, and 0 otherwise.

```
>>c=a>b;
```

Returns 1 where a is greater than b, and 0 otherwise.

```
>>c=a>3;
```

Returns 1 where a is greater than 3, and 0 otherwise.

```
>>c=a>=b;
```

Returns 1 where a is greater than or equal to b, and 0 otherwise.

```
>>c=a>=3;
```

Returns 1 where a is greater than or equal to 3, and 0 otherwise.

```
>>c=a<b;
```

Returns 1 where a is less than b, and 0 otherwise.

```
>>c=a==3;
```

Returns 1 where a is less than 3, and 0 otherwise.

```
>>c=a<=b;
```

Returns 1 where a is less than or equal to b, and 0 otherwise.

```
>>c=a<=3;
```

Returns 1 where a is less than or equal to 3, and 0 otherwise.

```
>>c=a~=b;
```

Returns 1 where a and b are not equal, and 0 otherwise.

```
>>c=a~=3;
```

Returns 1 where a does not equal 3, and 0 otherwise.

2.6 Logical Operations

In logical operations, Matlab views the elements of any operand as zero or non-zero. Zero maps to logic zero, whereas non-zero maps to logic 1. Logical operations are useful when combined with relational tests.

```
>>c=a&b;
```

Returns 1 where a and b are both non-zero, and 0 otherwise.

```
>>c=(a>3)&(b<=7);
```

Returns 1 where a is greater than 3 and b is less than or equal to 7, and 0 otherwise.

```
>>c=a|b;
```

Returns 1 where a or b or both are non-zero, and 0 where both a and b are zero.

```
>>c=(a>3)|(b<=7);
```

Returns 1 where a is greater than 3 or b is less than or equal to 7, and 0 otherwise.

```
>>c=~a;
```

Returns 1 where a is zero, and 0 otherwise..

```
>>c=~a&(b<=7);
```

Returns 1 where a is zero and b is less than or equal to 7, and 0 otherwise.

2.7 Resampling Operations

In this section, we will deal with three functions that change the sample rate of a signal: "downsample", "upsample", and "resample". The first two functions are easy to use when we wish to decrease or increase the sample time by an integer factor. The third is mainly used when the up- or downsampling factor is not an integer.

```
>>x1=[1 5 3 7 4 2 6 9 3 1 5 7];
>>x1d=downsample(x1,3);
>>x1u=upsample(x1,3);
>>x1r_3_over_2=resample(x1,3,2);
>>x1r_2_over_3=resample(x1,2,3);
```

The first instruction downsamples x1 by a factor of 3 by taking one sample and skipping 2. The output x1d will have the following elements: [1 7 6 1]. An optional third input argument may be written to indicate the initial offset, i.e. a number of samples to be skipped before taking the first sample.

The second instruction upsamples x1 by zero insertion. The output x1u will be [1 0 0 5 0 0 3 0 0 7 0 0 4 0 0 2 0 0 6 0 0 9 0 0 3 0 0 1 0 0 5 0 0 7 0 0].

The third instruction upsamples x1 by a factor of 1.5. The result is a 1×18 vector with the elements: [1 4.45 4.36 3 …]. The values are obtained by reconstructing the signal, and then sampling the reconstructed signal at the destination sample rate.

The fourth instruction downsamples x1 by a factor of 1.5. The result is a 1×8 vector with the values: [1.94 3.88 …]. These numbers are obtained by reconstructing the signal and then sampling the reconstructed signal at the destination sample rate.

2.8 Operations on the Independent Variable

In this section, we will discuss operations carried out on the independent variable of the signal: time. Such operations include time shifting, time scaling, and reflection about the y-axis. We will demonstrate these operations in the following example.

In this example, we will start by an exponentially-damped sinusoidal signal called x defined over the time interval from -2 to 4 seconds. The signal has a sample rate of 1000 Hz. We will interest ourselves in generating a number of other signals based on x. The first signal is obtained by shifting x 2 seconds to the right. The second signal is obtained by cyclically shifting x 2 seconds to the right. The third signal is obtained by scaling the time by a factor of 2. The fourth signal is obtained by scaling the time by a factor of 0.25. The fifth signal is obtained by reflecting the signal about the t=0 axis. The sample rates of all generated signals are equal to that of the original signal. We will list the instructions and then explain how they work.

```
>>t=linspace(-2,4,6000);
>>x=4*sin(pi*t).*exp(-0.4*t);
>>x1=[zeros(1,2000) x(1:4000)];
>>x2=[x(4001:end) x(1:4000)];
>>t3=linspace(-4,8,12000);
>>x3=resample(x,2,1);
>>t4=linspace(-1/2,1,1500);
>>x4=downsample(x,4);
>>t5=linspace(-4,2,6000);
>>x5=x(end:-1:1);
>>plot(t,x);
>>figure; plot(t,x1);
>>figure; plot(t,x2);
>>figure; plot(t3,x3);
>>figure; plot(t4,x4);
>>figure; plot(t5,x5);
```

In the first two instructions, the time base signal for x and the signal x are developed. In the third instruction, the signal is shifted by 2 seconds to the left. This is modeled as follows. The shift is acyclic. This means that the part that will disappear beyond the right end of the interval will not reappear at the left. The left segment of the signal empted by the shift will consist of zeros. Since the sample rate is 1000 Hz and the amount of shift is 2 seconds, we shall have 2000 zeros. The instruction concatenates a segment consisting of 2000 zeros with the first 4000 elements of x. The zeros will represent the signal in the interval from -2 to 0 seconds, and the next 4000 samples will represent the signal in the interval from 0 to 4 seconds. Since the interval of interest of the new signal is from -2 to 4 seconds, the same time base signal of x may be used with this new signal.

The fourth instruction takes the last 2000 samples (corresponding to the last 2 seconds) of the signal and brings them before the first 4000 samples. This corresponds to a cyclic shift of magnitude 2 seconds. The part that disappears from the right reappears at the left. Again, this signal may use the same time base signal as the original signal.

The fifth and sixth instructions are used to generate the fourth required signal. This signal extends along the interval from -4 to 8 seconds. This is a new interval of interest, which requires us to construct a new time base signal. Since the sample rate is also 1000 Hz, the total number of points in the time base signal will be 12000. The time base signal is generated using the expression linspace(-4,8,12000). The signal stretched in time domain across a new interval of interest that is twice the original may be obtained by upsampling the signal by a factor of 2. We may upsample the signal using the "upsample" function or the "resample" function. If we are considering the signal to be discrete, using "upsample" will be better, because discrete signals are upsampled by zero insertion between the original samples. If we are considering the signal continuous, using "resample" will be better, because continuous signals are upsampled by interpolating between the original samples. In our example, we are considering all our signals continuous. Thus, we have used "resample" to upsample the signal up to the number of points of the new interval of interest.

The seventh and eighth instructions are used to generate the fifth required signal. Since this signal is compressed in time by a factor of 4, the interval of interest of the new compressed signal will be from -0.5 to 1. This new interval of interest requires us to define a new time base signal. The number of points of this time base signal will be 1500, because the sample rate equals 1000 Hz. Thus, the new time base signal is obtained using linspace(-0.5,1,1500). Now, the compressed signal is obtained using the "downsample" function because the downsampling factor of 4 is an integer. If the downsampling factor was not an integer, the "downsample" function would not have been able to perform, and thus we would have used "resample" in this case.

The remaining instructions are used to open figures and plot the signals. We will deal with these instructions in the next section of this lesson.

2.9 Miscellaneous Signal Operations

In this subsection, we will demonstrate a multitude of diverse, yet common, signal processes that could not be classified under the classes of operations we have explained so far. Examples of such operations are clipping, rectification, signal switching, autocorrelation, and cross correlation.

Clipping

In the following example, a sine wave of amplitude 5 is generated. This wave will then be clipped at +3 volts and -4 volts. This means that any input value greater than +3 should be equal to +3 at the output, and any input value less than -4 should be equal to -4 at the output. The following instructions generate the sine wave and then subject it to the required clipping operation.

```
>>t=linspace(0,4,400);
>>x=5*sin(2*pi*t);
>>I1=find(x>3);
>>x(I1)=3;
>>I1=find(x<-4);
>>x(I1)=-4;
```

The third instruction finds the elements whose magnitudes are greater than +3, and outputs their location indices in I1. The fourth instruction writes +3 in these locations in x. The fifth instruction finds the elements whose magnitudes are less than -4, and outputs their location indices in I2. The sixth instruction writes -4 in these locations in x. Obviously, the new x is clipped at +3 and -4.

Half Wave Rectification

In the following example, a sine wave is generated. The wave is then half-wave rectified. Half wave rectification means that the negative portions of the signal should be clipped at 0. Thus, half wave rectification is a special case of clipping. Half wave rectification is demonstrated in the following instructions using two different methods. The two methods will yield identical outputs.

```
>>t=linspace(0,4,400);
>>x=3*sin(2*pi*t);
>>x1=x;
>>x1(find(x1<0))=0;
>>x2=x.*(x>0);
```

In the fourth instruction, the search for elements in x1 that are less than zero is written in the place of, and thus represents, the indices in x1 in which 0 will be written. In the last instruction, x is multiplied by the result of the relational test x>0. This test will return 1 at the locations of positive signal samples and 0 otherwise. Multiplying these two signals together is equivalent to "zeroing" the negative portions of x.

Full Wave Rectification

Full wave rectification is the process of negating the negative signal portions, such that the signal becomes all-positive. Full wave rectification may be simply done using the "abs" function, which removes any negative signs in the signal. The following example demonstrates this.

```
>>t=linspace(0,4,400);
>>x=3*sin(2*pi*t);
>>x1=abs(x);
```

Signal Switching

Signal switching is a common operation in which we have multiple input signals and one output signal. The output signal may follow either of the input signals depending on a certain condition. For example, if the condition is satisfied, the output equals the first signal. If not, it follows the second signal. The following example demonstrates a signal switching operation with three input signals and one output signal. The output continuously follows the maximum of the three inputs.

```
>>t=linspace(0,4,400);
>>x1=4*sin(2*pi*t);
>>x2=3*sin(2*pi*t+2*pi/3);
>>x3=3*cos(2*pi*t);
>>x1greatest=(x1>x2)&(x1>x3);
>>x2greatest=(x2>x1)&(x2>x3);
>>x3greatest=(x3>x1)&(x3>x2);
>>y=x1greatest.*x1+ x2greatest.*x2+ x3greatest.*x3;
```

Differentiation

Determining the derivative of a signal is a crucial part in many systems. The following example explains how the first derivative of a signal may be obtained. In a repetitive manner, we may obtain higher derivatives. The differencing process is carried out using the "diff" function. However, we must multiply by the sample rate (or divide by the sample time) because the derivative should equal the sample differences over the time differences.

```
>>t=linspace(0,4,400);
>>x=5*sin(2*pi*t).*exp(-0.5*t);
>>x_dash=100*diff(x);
```

Autocorrelation

Autocorrelation is a very important operation. It is a correlation of a signal with itself. Autocorrelation is useful in detecting the fundamental period of a noisy signal of unknown frequency, in determining the power spectral density of a signal, and much more. Thus, autocorrelation appear very frequently in speech or voice analysis, spectral analysis … etc. Given a signal of dimensions 1×N, the autocorrelation output will be a signal of dimensions 1×2N-1. The following example shows how the autocorrelation of a signal may be computed.

```
>>x=[0 0 1 1 1 1 0 0];
>>xACR=xcorr(x);
```

Cross Correlation

Cross correlation is the correlation between two different signals. Measuring the correlation between two different signals is a very important operation in matched filters and optimum receiver design, object detection, signal analysis … etc. The following example shows how the cross correlation between two signals may be computed.

```
>>x1=[0 0 1 1 1 1 0 0];
>>x2=[0 0 1 1 1 0 1 0]
>>CCSignal=xcorr(x1,x2);
```

Section III: Plotting Signals

2.10 Plotting Techniques

2.10.1 Using "plot" and "stem"

In this section, we will explain how signals could be plotted. We will handle two plotting functions: "plot" and "stem". The "plot" function is used in plotting continuous signals as shown in figure 2.3. The "stem" function is used in plotting discrete signals as shown in figure 2.1.

You can use plot or stem with one, two, or three input arguments as shown below. In the first two instructions, you have provided Matlab with the signal values, but you did not provide a time base signal through which Matlab can know how to label the t-axis. Thus, Matlab plots the signal against the index of the points as shown in figure 2.4 and 2.5. Matlab uses blue solid lines without markers by default with the "plot" function, and blue stems with circle markers by default with the "stem" function.

If you supply two inputs, a time base vector and a signal vector, as shown in the third and fourth instructions, the time base vector will be used to set the labeling on the t-axis as shown in figures 2.6 and 2.7.

Whether you decide to pass the time base vector or not, you may add an optional input argument called the line specification string. The line specification string allows you to override the default line color, line type, and marker. The line specification string typically consists of three elements, the line color character, the line type character(s), and the marker shape character. The line color character may be 'r', 'g', 'b', 'w', 'k', 'c', 'm', or 'y'. The line type may be '-', '--', ':', or none. The marker shape may be 'd', 'h', 'o', 'p', 's', 'v', 'x', '+', '-','*', '^', '<', '>', or none. For example, the line specification string 'k--o' draws a black dashed line with circle markers. The string 'rx' draws a red line with cross-shaped markers only (no line is drawn). The fifth and sixth instructions draw signals with modified line specifications, as shown in figures 2.8 and 2.9.

```
>>plot(x1);
>>stem(x1);
>>plot(t1,x1);
>>stem(t1,x1);
>>plot(t1,x1,'k:o');
>>stem(t1,x1,'k:o');
```

2.10.2 Drawing Multiple Graphs on the Same Axes

If you try to execute the previous instructions one after another, you will find out that each new plot erases the previous one. We can plot several lines together by invoking the "hold on" command. In the following example, hold on prevents Matlab from erasing the first plot when the second plot command is issued. The result is shown in figure 2.10.

```
>>plot(t1,x1,'k-x');
>>hold on;
>>stem(t2,x2,'k-o');
```

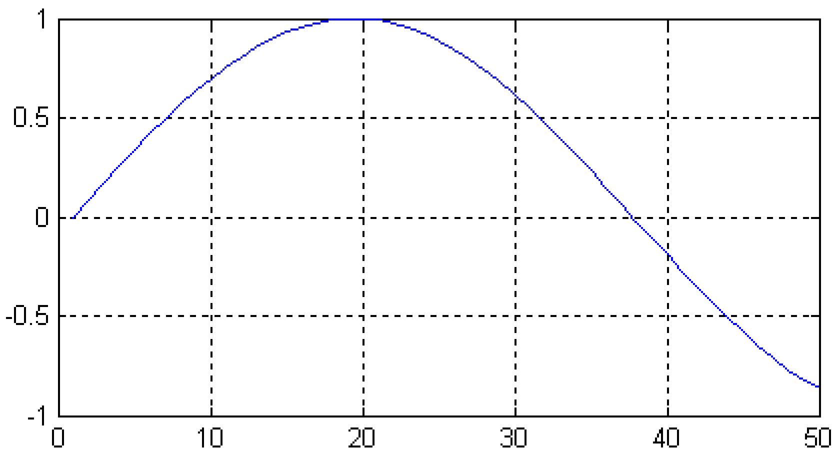


Figure 2.4

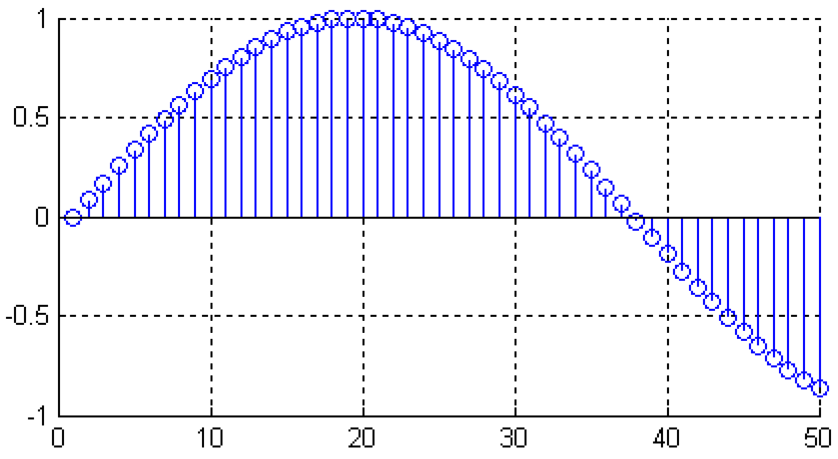


Figure 2.5

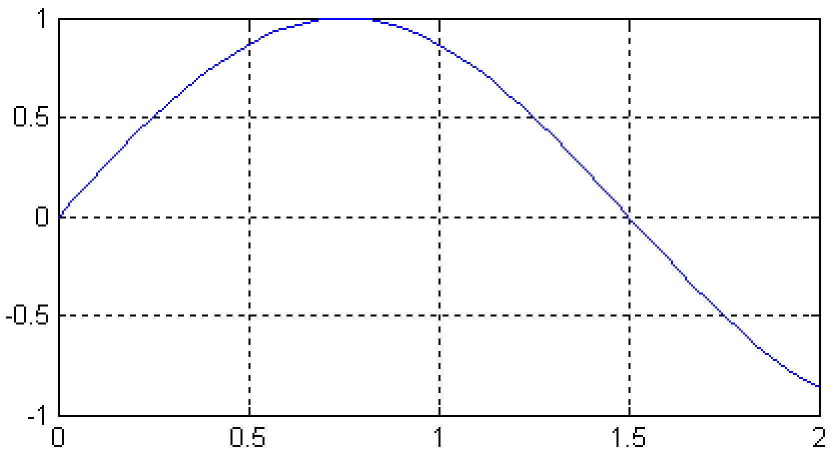


Figure 2.6

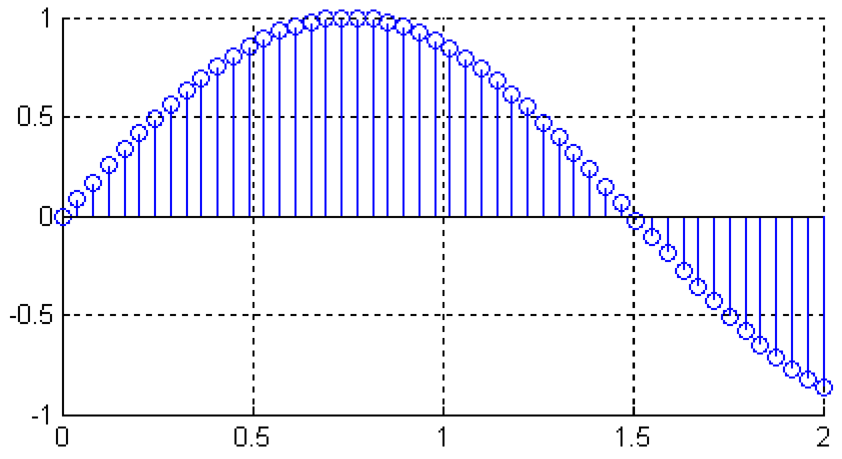


Figure 2.7

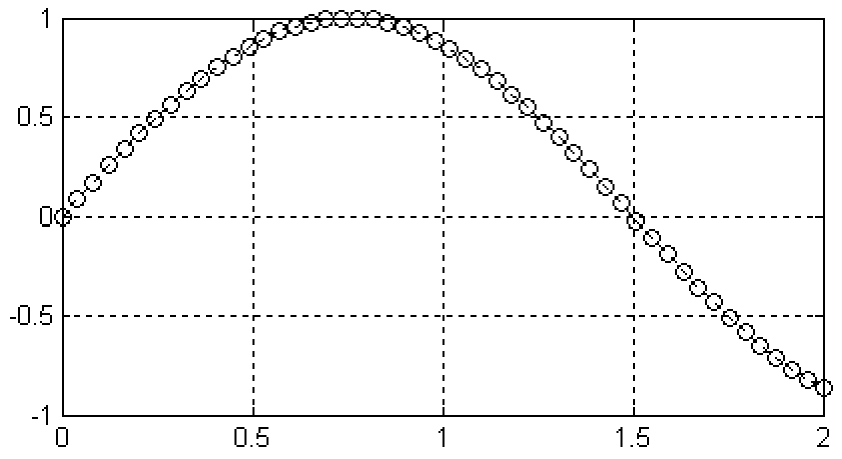


Figure 2.8

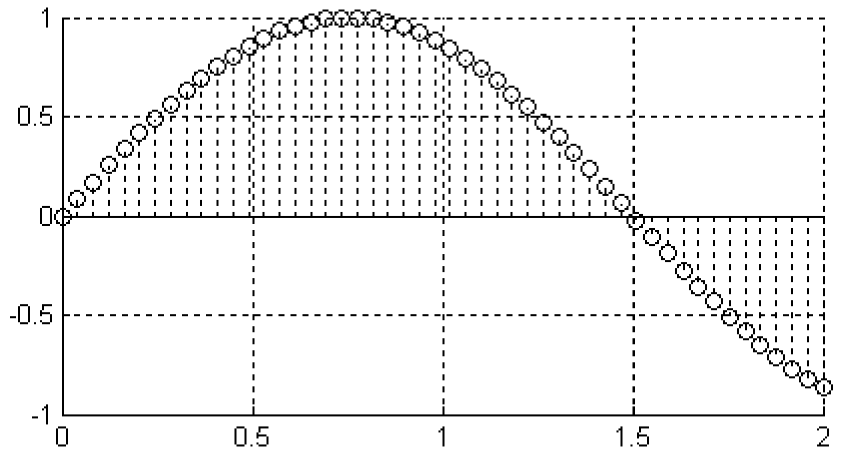


Figure 2.9

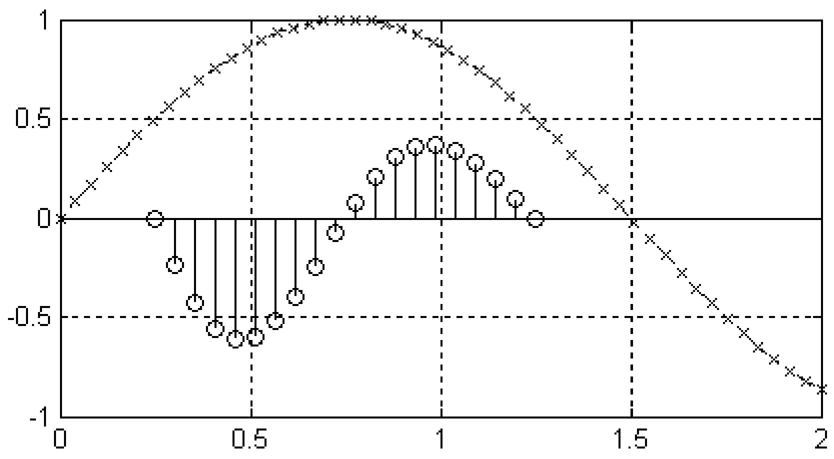


Figure 2.10

2.10.3 Using "subplot"

The "subplot" function allows us to plot multiple signals in multiple axis boxes on the same figure. The following figure depicts a typical figure generated using subplot.

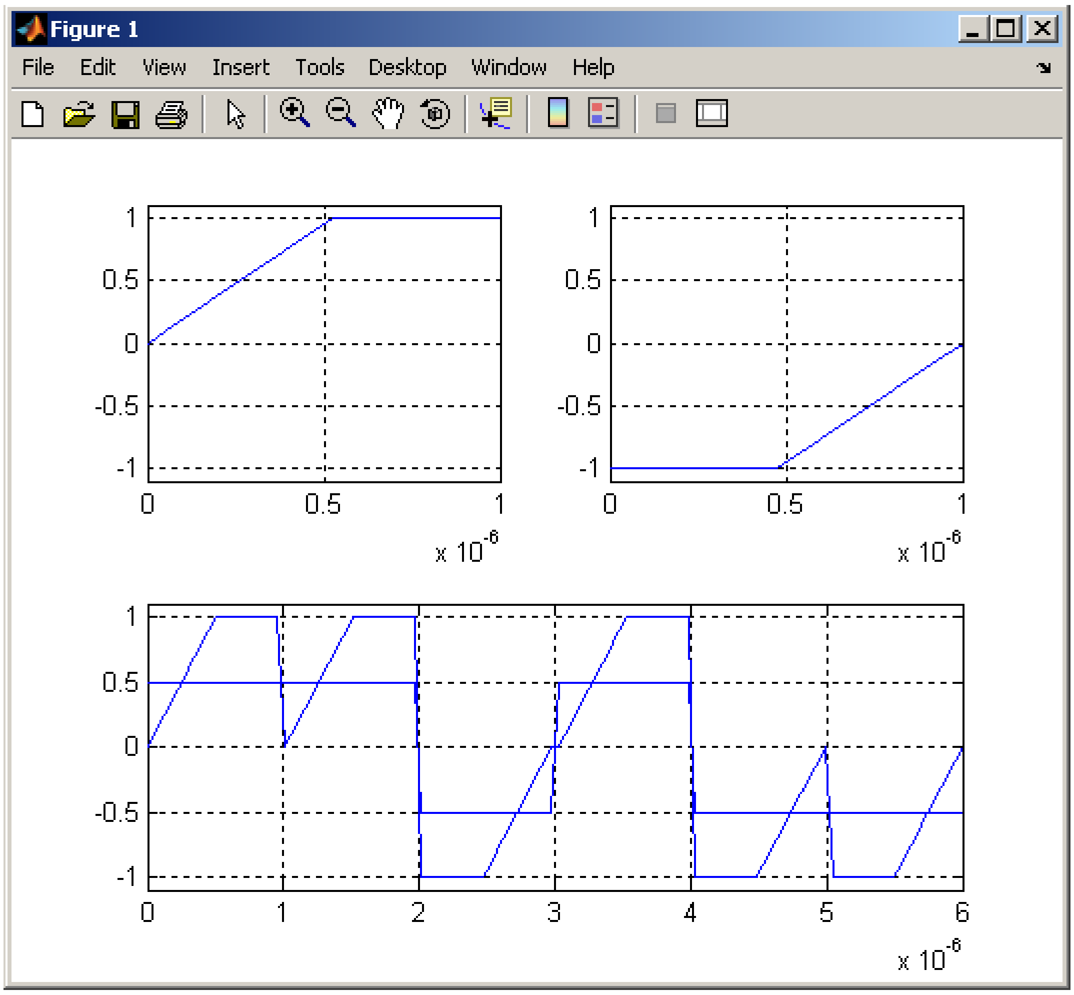


Figure 2.11

The following instructions were written to generate this figure. The two signals in the top two boxes are called s1 and s2. The two signals in the wide axes box are called s and g.

```
>>t1=linspace(0,1e-6,20);
>>s1=[0:0.1:0.9 ones(1,10)];
>>s2=-s1(end:-1:1);
```

```
>>t6=linspace(0,6e-6,120);
>>s=[s1 s1 s2 s1 s2 s2];
>>P=o.5*ones(1,20);
>>g=[P P –P P –P -P];
>>subplot(2,2,1);
>>plot(t1,s1);
>>subplot(2,2,2);
>>plot(t1,s2);
>>subplot(2,2,[3 4]);
>>plot(t6,g);
>>hold on;
>>plot(t6,s);
```

The first subplot command divides the figure into 2×2 partitions, and creates and activates the first axes box, which is situated in the top left partition of the figure. Any subsequent plotting command will target this axes box. The second subplot command creates and activates the second axes box, which is situated in the top right partition of the figure. Thus, the subsequent plot goes to this axes box. The third subplot command creates and activates an axes box that spans the third and fourth partitions. The three subsequent plotting commands target this wide axes box.

Section IV: Computing Signal Statistics

2.11 Computing Signal Statistics

In this section, we will explain how we may compute various common statistics of signals. Some statistics are standard such as the mean, range, variance, and standard deviation. Others are non-standard, such as the percentage of time a signal satisfies a certain condition. We will also explain how we may generate the histogram of a signal.

2.11.1 Standard Statistics

Mean

The mean of a signal may be computed using the "mean" function as follows.

```
>> mu=mean(x);
```

Range (Minimum and Maximum Values)

The minimum and maximum values of a function may be determined using the "min" and "max" functions as follows.

```
>> MinVal=min(x);
>> MaxVal=max(x);
```

Variance and Standard Deviation

The variance and standard deviation of a signal may be computed using the "var" and "std" functions as follows.

```
>> Variance=var(x);
>> StdDev=std(x);
```

2.11.2 Non-Standard Statistics

In this subsection, we demonstrate how we may compute some non-standard statistics of a signal. We have a signal called x, and we want to compute 2 things for this signal. Firstly, we want to determine the number of zero crossings of the signal. This is a very important quantity in signal analysis. Secondly, we want to determine the percentage of time in which the signal is above the value midway between its mean and maximum. We will write down the instructions, and then explain how they work.

```
>>%Computing the number of zero crossings
>>x1=x;
>>x1(find(x1>0))=1;
>>x1(find(x1<0))=-1;
>>Crossings=(-(x1*[0 x1(1:end-1)])==1)|(x1==0);
>>NumberofCrossings=sum(Crossings);
>>ComparisonLevel=(mean(x)+max(x))/2;
>>PercentageTime=100*length(find(x>ComparisonLevel))/length(x);
```

To determine the number of zero crossings, we have generated an auxiliary signal called x1. We have modified x1 such that it is +1 at all locations in x holding positive values, 0 at all locations in x holding 0, and -1 at all locations in x holding negative values. The number of zeros crossings may then be determined from the number of zero-valued elements plus the number of sign flips in x1. The zero-valued elements are detected using x1==0. The sign flips may be detected by multiplying x1 by a shifted version of it. Whenever there is a sign change, the product will be -1. By multiplying this product by -1, the result will be +1 wherever there is a sign flip. The instruction starting with "Crossings=…" returns a vector that is one wherever there is a zero or a sign flip, and zero otherwise. The next instruction sums this vector, which is equal to counting the number of ones in "Crossings".

The last two instructions are responsible for computing the percentage of time in which the signal is above the value midway between its mean and max. The first instruction computes this value, and the second instruction computes the required percentage by dividing the number of signal points that are greater than the value by the total number of points in the signal and then multiplying by 100.

2.11.3 Histograms

A signal histogram is a histogram that illustrates the distribution of the values of signal samples. We may generate the histogram of any signal x by typing the following instruction:

```
>>hist(x,N)
```

N is the number of bins in the histogram. Figures 2.12 and 2.13 show two histograms of a sinusoidal signal generated using 30 bins in the first case and 100 bins in the second case.

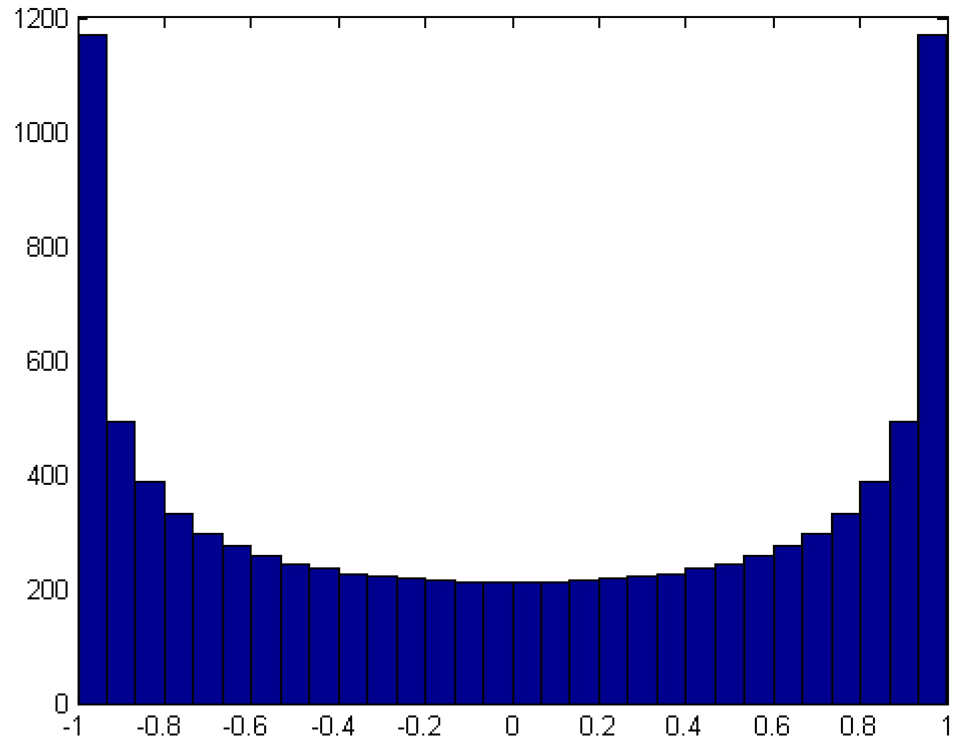


Figure 2.12

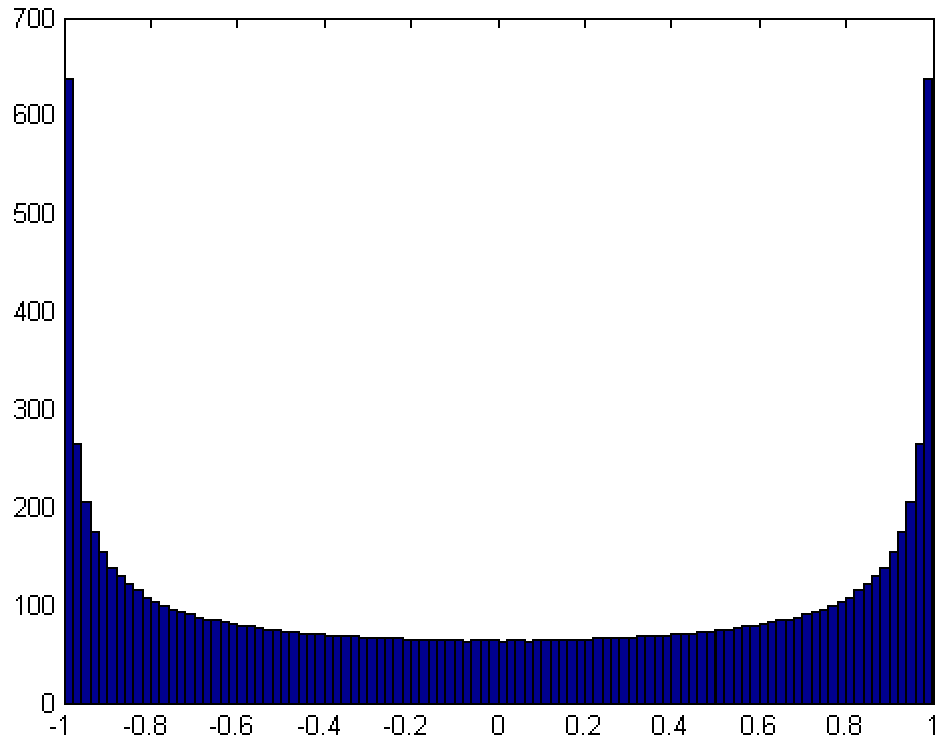


Figure 2.13

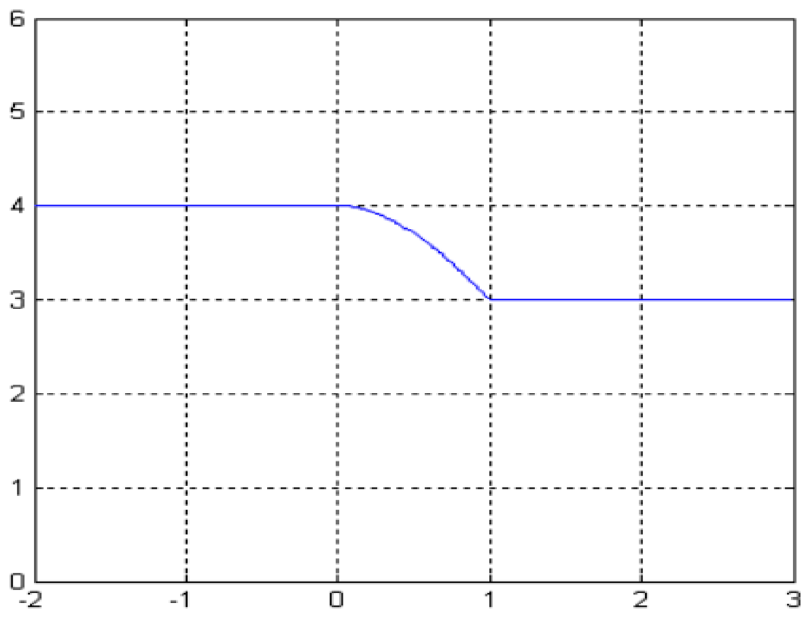
Assignment 1

1. The vector V is given by V=[2 8 7 3 1 0 8 9]. Write down ***a single Matlab instruction*** to produce a vector that contains 1 in the place of the odd numbers and -1 in the place of the even numbers.

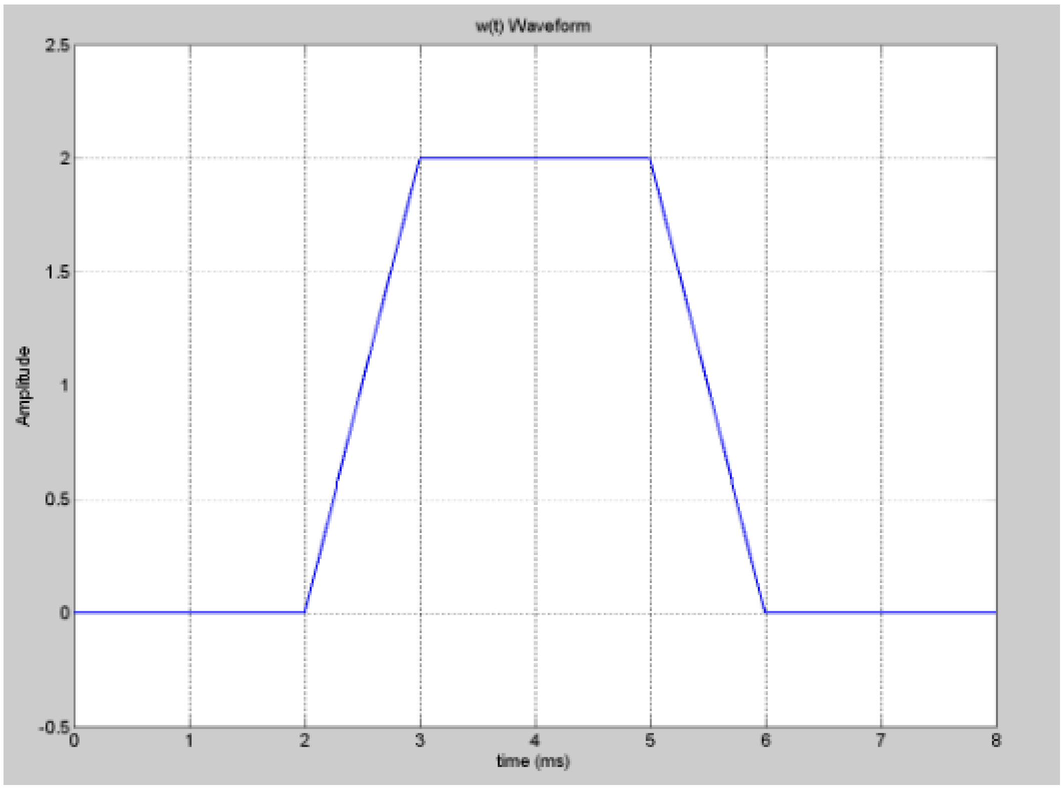
2. Generate the following sequence: V=1 4 9 16 25 ………………… 16 9 4 1   Where V is a sequence of length of 49 elements.
   For the previous generated sequence V, write down ***a single Matlab instruction*** that carries out the operation:
   a) Add 2 to the last 3 elements.
   b) Reverse the order of the last 10 elements.
   c) For the first 48 elements, add the elements in the even places(2,4,6,…) to that in the odd places (1,3,5,…) and store the output in the odd places.

3.

Write down the Matlab instructions that will generate the signal depicted below. This signal consists of a DC segment from -2 to 0 seconds, a quarter cycle of a sinusoidal wave from 0 to 1 seconds, and another DC segment from 1 to 3 seconds. The sample rate is 100 Hz.



4. Plot the following signal with FS=100 HZ.


w(t) Waveform
Amplitude
time (ms)