in the differential equation, we obtain

$$\mathcal{R}e\left[V_c(1+j\Omega_0)e^{j\Omega_0 t}\right] = \mathcal{R}e\left[Ae^{j\Omega_0 t}\right]$$

so that

$$V_c = \frac{A}{1+j\Omega_0} = \frac{A}{\sqrt{1+\Omega_0^2}}e^{-j\tan^{-1}(\Omega_0)}$$
$$= Ce^{j\psi}$$

and the sinusoidal steady-state response is

$$v_c(t) = \mathcal{R}e\left[V_c e^{j\Omega_0 t}\right]$$
$$= \frac{A}{\sqrt{1+\Omega_0^2}}\cos(\Omega_0 t - \tan^{-1}(\Omega_0))$$

which coincides with the response obtained above. The ratio of the output phasor V_c to the input phasor V_i,

$$\frac{V_c}{V_i} = \frac{1}{1+j\Omega_0}$$

gives the response of the circuit at frequency Ω_0. If the frequency of the input is a generic Ω, changing Ω_0 above for Ω gives the frequency response for all possible frequencies.

The concepts of *linearity* and *time invariance* will be used in both continuous-time as well as discrete-time systems, along with the Fourier representation of signals in terms of sinusoids or complex exponentials, to simplify the analysis and to allow the design of systems. Thus, transform methods such as Laplace and the Z-transform will be used to solve differential and difference equations in an algebraic setup. Fourier representations will provide the frequency perspective. This is a general approach for both continuous-time and discrete-time signals and systems. The introduction of the concept of the transfer function will provide tools for the analysis as well as the design of linear time-invariant systems. The design of analog and discrete filters is the most important application of these concepts. We will look into this topic in Chapters 5, 6, and 11.

0.5 SOFT INTRODUCTION TO MATLAB

MATLAB is a computing language based on vectorial computations.[10] In this section, we will introduce you to the use of MATLAB for numerical and symbolic computations.

[10] MATLAB stands for matrix laboratory. MatWorks, the developer of MATLAB, was founded in 1984 by Jack Little, Steve Bangert, and Cleve Moler. Moler, a math professor at the University of New Mexico, developed the first version of MATLAB in Fortran in the late 1970s. It only had 80 functions and no M-files or toolboxes. Little and Bangert reprogrammed it in C and added M-files, toolboxes, and more powerful graphics [49].

0.5.1 Numerical Computations

The following instructions are intended for users who have no background in MATLAB but are interested in using it in signal processing. Once you get the basic information on how to use the language you will be able to progress on your own.

1. Create a directory where you will put your work, and from where you will start MATLAB. This is important because when executing a program, MATLAB will look at the current directory, and if the file is not present in the current directory, and if it is not a MATLAB function, MATLAB gives an error indicating that it cannot find the desired program.
2. There are two types of programs in MATLAB: the script, which consists in a list of commands using MATLAB functions or your own functions, and the functions, which are programs that can be called with different inputs and provide the corresponding outputs. We will show examples of both.
3. Once you start MATLAB, you will see three windows: the command window, where you will type commands; the command history, which keeps a list of commands that have been used; and the workspace, where the variables used are kept.
4. Your first command on the command window should be to change to your data directory where you will keep your work. You can do this in the command window by using the command CD (change directory) followed by the desired directory. It is also important to use the command clear all and clf to clear all previous variables in memory and all figures.
5. Help is available in several forms in MATLAB. Just type helpwin, helpdesk, or demo to get started. If you know the name of the function, help will give you the necessary information on the particular function, and it will also give you information on help itself. Use help to find more about the functions used in this introduction to MATLAB.
6. To type your scripts or functions you can use the editor provided by MATLAB; simply type edit. You can also use any text editor to create scripts or functions, which need to be saved with the .m extension.

Creating Vectors and Matrices

Comments are preceded by percent, and to begin a script, as the following, it is always a good idea to clear all previous variables and all previous figures.

```
% matlab primer
clear all                % clear all variables
clf                      % clear all figures
% row and column vectors
x = [ 1 2 3 4]           % row vector
y = x'                   % column vector
```

The corresponding output is as follows (notice that there is no semicolon (;) at the end of the lines to stop MATLAB from providing an output when the above script is executed).

```
x =
     1     2     3     4
```

```
y =
     1
     2
     3
     4
```

To see the dimension of x and y variables, type

```
whos        % provides information on existing variables
```

to which MATLAB responds

```
Name      Size              Bytes  Class
x         1x4                  32  double array
y         4x1                  32  double array
Grand total is 8 elements using 64 bytes
```

Notice that a vector is thought of as a matrix; for instance, vector x is a matrix of one row and four columns. Another way to express the column vector y is the following, where each of the row terms is separated by a semicolon (;)

```
y = [1;2;3;4]    % another way to write a column
```

To give as before:

```
y =
     1
     2
     3
     4
```

MATLAB does not allow arguments of vectors or matrices to be zero or negative. For instance, if we want the first entry of the vector y we need to type

```
y(1)    % first entry of vector y
```

giving as output

```
ans =
     1
```

If we type

```
y(0)
```

it will give us an error, to which we get the following warning:

```
??? Subscript indices must either be real positive integers or logicals.
```

MATLAB also has a peculiar way to provide information in a vector, for instance:

```
y(1:3)          % first to third entry of column vector y
```

giving as expected the first to the third entries of the column vector y:

```
ans =
     1
     2
     3
```

The following will give the third to the first entry in the row vector x (notice the difference in the two outputs; as expected the values of y are given in a column, while the requested entries of x are given in a row).

```
x(3:-1:1)          % displays entries x(3) x(2) x(1)
```

Thus,

```
ans =
     3     2     1
```

Matrices are constructed as an concatenation of rows (or columns):

```
A = [ 1 2; 3 4; 5 6]   % matrix A with rows [1 2], [3 4] and [5 6]
```

```
A =
     1     2
     3     4
     5     6
```

To create a vector corresponding to a sequence of numbers (in this case integers) there are different approaches, as follows:

```
n = 0:10    % vector with entries 0 to 10 increased by 1
```

This approach gives the following as output:

```
n =
  Columns 1 through 10
     0     1     2     3     4     5     6     7     8     9
  Column 11
    10
```

which is the same as the command

```
n = [0:10]
```

If we wish the increment different from 1 (default value), then we indicate it as in the following:

```
n1 = 0:2:10  % vector with entries from 0 to 10 increased by 2
```

which gives

```
n1 =
     0     2     4     6     8    10
```

We can combine the above vectors into one as follows:

```
nn1 = [n n1]   % combination of vectors
```

to get

```
nn1 =
    Columns 1 through 10
      0   1   2   3   4   5   6   7   8   9
    Columns 11 through 17
     10   0   2   4   6   8  10
```

Vectorial Operations

MATLAB allows the conventional vectorial operations as well as facilitates others. For instance, if we wish to multiply by 3 every entry of the row vector x given above, the command

```
z = 3*x          % multiplication by a constant
```

would give

```
z =
    3   6   9  12
```

Besides the conventional multiplication of vectors with the correct dimensions, MATLAB allows two types of multiplications of one vector by another. The first one is where the entries of one vector are multiplied by the corresponding entries of the other. To effect this the two vectors should have the same dimension (i.e., both should be columns or rows with the same number of entries) and it is necessary to put a dot before the multiplication operator—that is, as shown here:

```
v = x.*x  % multiplication of entries of two vectors

v =
    1   4   9  16
```

The other type of multiplication is the conventional multiplication allowed in linear algebra. For instance, with that of a row vector by a column vector,

```
w = x*x'  % multiplication of x (row vector) by x'(column vector)

w = 30
```

the result is a constant—in this case, the length of the row vector should coincide with that of the column vector. If you multiply a column (say x') of dimension 4×1 by a row (say x) of dimension 1×4 (notice that the 1s coincide at the end of the first dimension and at the beginning of the second), the multiplication $z = x' * x$ results in a 4×4 matrix.

The solution of a set of linear equations is very simple in MATLAB. To guarantee that a unique solution exists, the determinant of the matrix should be computed before inverting the matrix. If the determinant is zero MATLAB will indicate the solution is not possible.

```
% Solution of linear set of equations Ax = b
A = [1 0 0; 2 2 0; 3 3 3];     % 3x3 matrix
t = det(A);                    % MATLAB function that calculates determinant
b = [2 2 2]';                  % column vector
x = inv(A)*b;                  % MATLAB function that inverts a matrix
```

The results of these operations are not given because of the semicolons at the end of the commands. The following script could be used to display them:

```
disp(' Ax = b')          % MATLAB function that displays the text in ' '
A
b
x
t
```

which gives

```
Ax = b
A =
     1     0     0
     2     2     0
     3     3     3
b =
     2
     2
     2
x =
    2.0000
   -1.0000
   -0.3333
t =
     6
```

Another way to solve this set of equations is

```
x = b'/A'
```

Try it!

MATLAB provides a fast way to obtain certain vectors/matrices; for instance,

```
% special vectors and matrices
x = ones(1, 10)   % row of ten 1s
x =
     1     1     1     1     1     1     1     1     1     1
A = ones(5, 5)    % matrix of 5 x 5 1s
A =
     1     1     1     1     1
     1     1     1     1     1
     1     1     1     1     1
     1     1     1     1     1
     1     1     1     1     1
x1 = [x zeros(1, 5)]   % vector with previous x and 5 0s
```

```
x1 =
  Columns 1 through 10
    1   1   1   1   1   1   1   1   1   1
  Columns 11 through 15
    0   0   0   0   0
A(2:5, 2:5) = zeros(4, 4)  % zeros in rows 2–5, columns 2–5
A =
    1   1   1   1   1
    1   0   0   0   0
    1   0   0   0   0
    1   0   0   0   0
    1   0   0   0   0
y = rand(1,10)      % row vector with 10 random values (uniformly
                    % distributed in [0,1]
y =
  Columns 1 through 6
   0.9501   0.2311   0.6068   0.4860   0.8913   0.7621
  Columns 7 through 10
   0.4565   0.0185   0.8214   0.4447
```

Notice that these values are between 0 and 1. When using the normal or Gaussian-distributed noise the values can be positive or negative reals.

```
y1 = randn(1,10)        % row vector with 10 random values
                        % (Gaussian distribution)
y1 =
  Columns 1 through 6
   -0.4326  -1.6656   0.1253   0.2877  -1.1465   1.1909
  Columns 7 through 10
    1.1892  -0.0376   0.3273   0.1746
```

Using Built-In Functions and Creating Your Own

MATLAB provides a large number of built-in functions. The following script uses some of them.

```
% using built-in functions
t = 0:0.01:1;   % time vector from 0 to 1 with interval of 0.01
x = cos(2*pi*t/0.1);    % cos processes each of the entries in
    % vector t to get the corresponding value in vector x
% plotting the function x
figure(1)  % numbers the figure
plot(t, x)  % interpolated continuous plot
xlabel('t (sec)')  % label of x-axis
ylabel('x(t)')  % label of y-axis
```

```
% let's hear it
sound(1000*x, 10000)
```

The results are given in Figure 0.14.

To learn about any of these functions use *help*. In particular, use *help* to learn about MATLAB routines for plotting *plot* and *stem*. Use *help sound* and *help waveplay* to learn about the sound routines available in MATLAB. Additional related functions are put at the end of these help files. Explore all of these and become aware of the capabilities of MATLAB. To illustrate the plotting and the sound routines, let us create a chirp that is a sinusoid for which the frequency is varying with time.

```
y = sin(2*pi*t.^2/.1);   % notice the dot in the squaring
                                  % t was defined before
sound(1000*y, 10000)  % to listen to the sinusoid
figure(2) % numbering of the figure
plot(t(1:100), y(1:100))  % plotting of 100 values of y
figure(3)
plot(t(1:100), x(1:100), 'k', t(1:100), y(1:100), 'r')  % plotting x and y on same plot
```

Let us hope you were able to hear the chirp, unless you thought it was your neighbor grunting. In this case, we plotted the first 100 values of t and y and let MATLAB choose the color for them. In the second plot we chose the colors: black (dashed lines) for x and blue (continuous line) for the second signal $y(t)$ (see Figure 0.15).

Other built-in functions are sin, tan, acos, asin, atan, atan2, log, log10, exp, etc. Find out what each does using help and obtain a listing of all the functions in the signal processing toolbox.

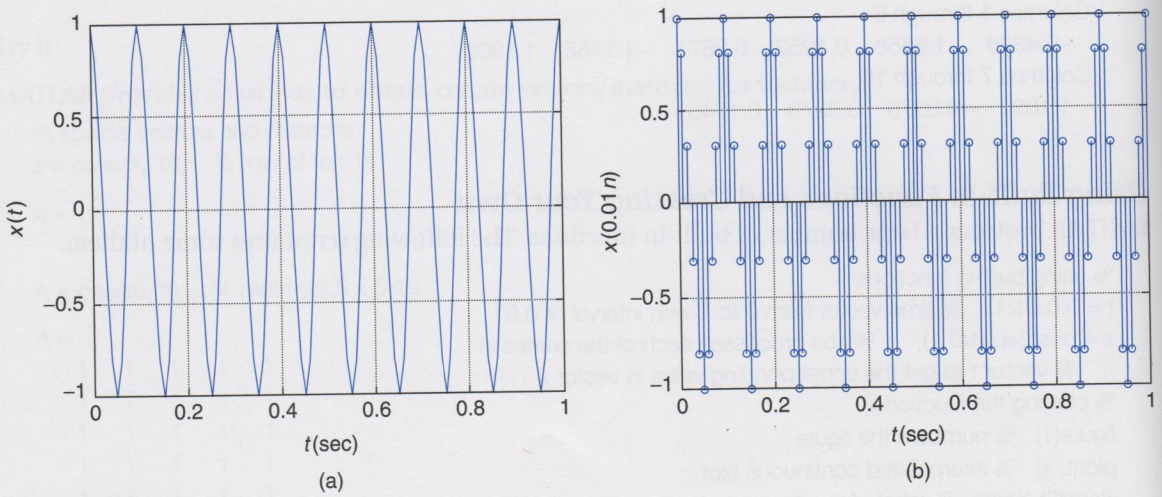

FIGURE 0.14
(a) Plotting of a sinusoid using *plot*, which gives a continuous plot, and (b) *stem*, which gives a discrete plot.

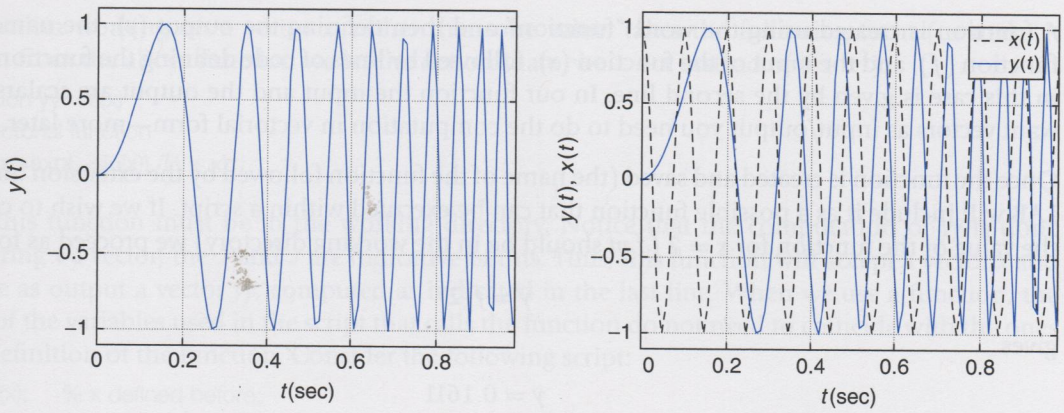

FIGURE 0.15
(a) Plotting chirp (MATLAB chooses color), (b) sinusoid and chirp (the sinusoid is plotted with dashed lines and the chirp with solid lines).

You do not need to define π, as it is already done in MATLAB. For complex numbers also you do not need to define the square root of -1, which for engineers is 'j' and for mathematicians 'i' (they have no current to worry about).

```
% pi and j
pi
j
i

ans =
    3.1416
ans =
        0 + 1.0000i
ans =
        0 + 1.0000i
```

Creating Your Own Functions

MATLAB has created a lot of functions to make our lives easier, and it allows us also to create—in the same way—our own. The following file is for a function f with an input of a scalar x and output of a scalar y related by a mathematical function:

```
function y = f(x)
y = x*exp(-sin(x))/(1 + x^2);
```

Functions cannot be executed on their own—they need to be part of a script. If you try to execute the above function MATLAB will give the following:

```
??? format compact;function y = f(x)
                   |
Error: A function declaration cannot appear within a script M-file.
```

A function is created using the word "function" and then defining the output (y), the name of the function (f), and the input of the function (x), followed by lines of code defining the function, which in this case is given by the second line. In our function the input and the output are scalars. If you want vectors as input/output you need to do the computation in vectorial form—more later.

Once the function is created and saved (the name of the function followed by the extension .m), MATLAB will include it as a possible function that can be executed within a script. If we wish to compute the value of the function for $x = 2$ ($f.m$ should be in the working directory) we proceed as follows:

$$y = f(2)$$

gives

$$y = 0.1611$$

To compute the value of the function for a vector as input, we compute for each of the values in the vector the corresponding output using a for loop as shown in the following.

```
x = 0:0.1:100;          % create an input vector x
N = length(x);          % find the length of x
y = zeros(1,N);         % initialize the output y to zeros
for n = 1:N,            % for the variable n from 1 to N, compute
    y(n) = f(x(n));     % the function
end
figure(3)
plot(x, y)
grid                    % put a grid on the figure
title('Function f(x)')
xlabel('x')
ylabel('y')
```

This is not very efficient. A general rule in MATLAB is: Loops are to be avoided, and vectorial computations are encouraged. The results are shown in Figure 0.16.

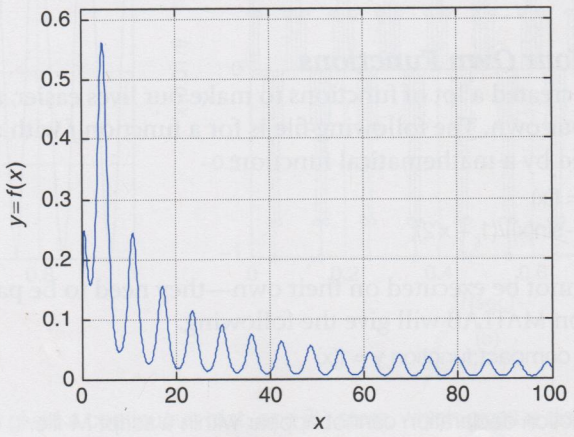

FIGURE 0.16
Result of using the function $f(.)$

The function working on a vector x, rather than one value, takes the following form (to make it different from the above function we let the denominator be $1 + x$ instead of $1 + x^2$):

```
function yy = ff(x)
% vectorial function
yy = x.*exp(-sin(x))./(1 + x);
```

Again, this function must be in the working directory. Notice that the computation of yy is done considering x a vector; the .* and ./ are indicative of this. Thus, this function will accept a vector x and will give as output a vector yy, computed as indicated in the last line. When we use a function, the names of the variables used in the script that calls the function do not need to coincide with the ones in the definition of the function. Consider the following script:

```
z = ff(x);     % x defined before,
                % z instead of yy is the output of the function ff
figure(4)
plot(x, z); grid
title('Function ff(x)') % MATLAB function that puts title in plot
xlabel('x')  % MATLAB function to label x-axis
ylabel('z')  % MATLAB function to label y-axis
```

The difference between plot and stem is important. The function plot interpolates the vector to be plotted and so the plot appears continuous, while stem simply plots the entries of the vector, separating them uniformly. The input x and the output of the function are discrete time and we wish to plot them as such, so we use stem.

```
stem(x(1:30), z(1:30))
grid
title('Function ff(x)')
xlabel('x')
ylabel('z')
```

The results are shown in Figure 0.17.

More on Plotting

There are situations where we want to plot several plots together. One can superpose two or more plots by using hold on and hold off. To put several figures in the same plot, we can use the function subplot. Suppose we wish to plot four figures in one plot and they could be arranged as two rows of two figures each. We do the following:

```
subplot(221)
plot(x, y)
subplot(222)
plot(x, z)
subplot(223)
stem(x, y)
subplot(224)
stem(x, z)
```

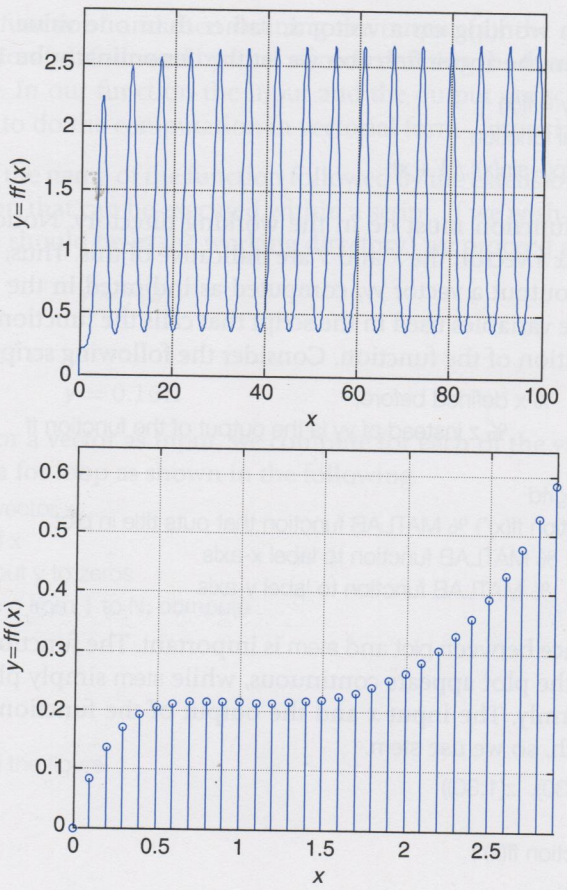

FIGURE 0.17
Results of using the function ff(.) (notice the difference in scale in the x axis).

In the subplot function the first two numbers indicate the number of rows and the number of columns, and the last digit refers to the order of the graph that is, 1, 2, 3, and 4 (see Figure 0.18).

There is also a way to control the values in the axis, by using the function (you guessed!) **axis**. This function is especially useful after we have a graph and want to improve its looks. For instance, suppose that the professor would like the above graphs to have the same scales in the y-axis (picky professor). You notice that there are two scales in the y-axis, one 0-0.8 and another 0-3. To have both with the same scale, we choose the one 0-3, and modify the above code to the following

```
subplot(221)
plot(x, y)
axis([0 100 0 3])
subplot(222)
plot(x, z)
axis([0 100 0 3])
subplot(223)
stem(x, y)
```

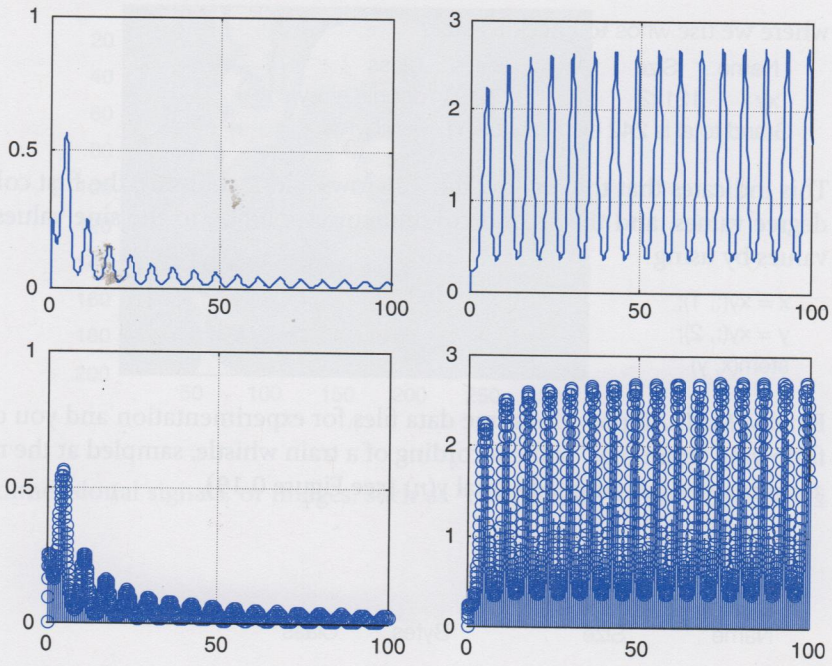

FIGURE 0.18
Plotting four figures in one.

```
axis([0 100 0 3])
subplot(224)
stem(x, z)
axis([0 100 0 3])
```

Saving and Loading Data

In many situations you would like to either save some data or load some data. The following is one way to do it. Suppose you want to build and save a table of sine values for angles between 0 and 360 degrees in intervals of 3 degrees. This can be done as follows:

```
x = 0:3:360;
y = sin(x*pi/180);  % sine computes the argument in radians
xy = [x' y'];  % vector with 2 columns one for x'
                % and  another for y'
```

Let's now save these values in a file "sine.mat" by using the function save (use help save to learn more):

```
save sine.mat xy
```

To load the table, we use the function load with the name given to the saved table "sine" (the extension *.mat is not needed). The following script illustrates this:

```
clear all
load sine
whos
```

where we use whos to check its size:

```
Name      Size            Bytes  Class
xy        121x2            1936  double array
Grand total is 242 elements using 1936 bytes
```

This indicates that the array *xy* has 121 rows and 2 columns, the first colum corresponding to x, the degree values, and the second column corresponding to the sine values, y. Verify this and plot the values by using

```
x = xy(:, 1);
y = xy(:, 2);
stem(x, y)
```

Finally, MATLAB provides some data files for experimentation and you only need to load them. The following "train.mat" is the recording of a train whistle, sampled at the rate of F_s samples/sec, which accompanies the sampled signal $y(n)$ (see Figure 0.19).

```
clear all
load train
whos
```

```
Name        Size              Bytes  Class

Fs             1x1                 8  double array
y          12880x1             103040  double array

Grand total is 12881 elements using 103048 bytes
```

```
sound(y, Fs)
plot(y)
```

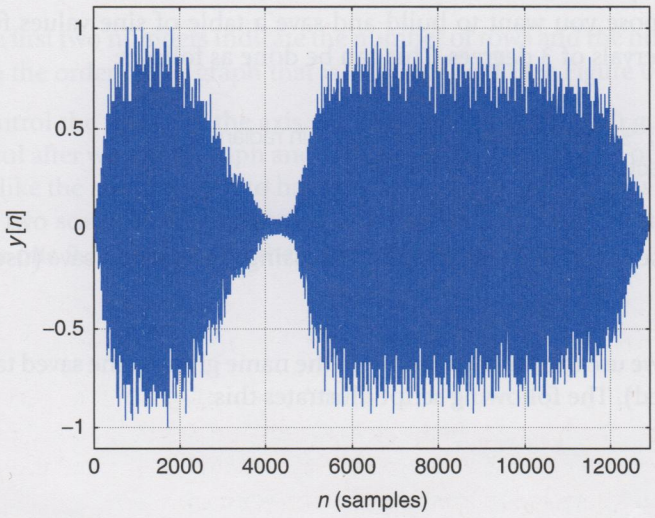

FIGURE 0.19
Train signal.

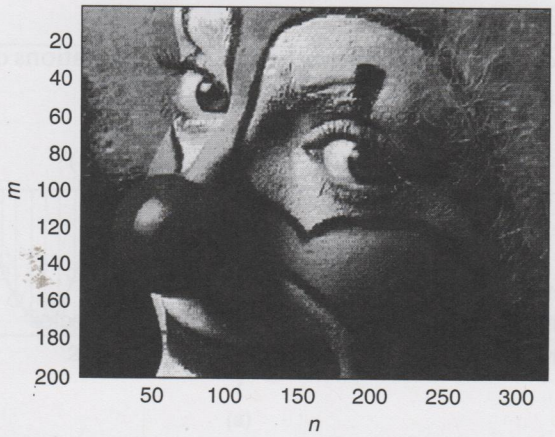

FIGURE 0.20
Clown in gray scale.

MATLAB also provides two-dimensional signals, or images, such as "clown.mat," a 200 × 320 pixels image.

```
clear all
load clown
whos
```

Name	Size	Bytes	Class
X	200x320	512000	double array
caption	2x1	4	char array
map	81x3	1944	double array

Grand total is 64245 elements using 513948 bytes

We can display this image in gray levels by using the following script (see Figure 0.20):

```
colormap('gray')
imagesc(X)
```

Or in color using

```
colormap('hot')
imagesc(X)
```

0.5.2 Symbolic Computations

We have considered the numerical capabilities of MATLAB, by which numerical data are transformed into numerical data. There will be many situations when we would like to do algebraic or calculus operations resulting in terms of variables rather than numerical data. For instance, we might want to find a formula to solve quadratic algebraic equations, to find a difficult integral, or to obtain the Laplace or the Fourier transform of a signal. For those cases MATLAB provides the Symbolic Math Toolbox, which uses the interface between MATLAB and MAPLE, a symbolic computing system. In this section, we provide you with an introduction to symbolic computations by means of examples, and hope to get you interested in learning more on your own.

Derivatives and Differences

The following script compares symbolic with numeric computations of the derivative of a chirp signal (a sinusoid with changing frequency) $y(t) = \cos(t^2)$, which is

$$z(t) = \frac{dy(t)}{dt} = -2t\sin(t^2)$$

```
clf; clear all
% symbolic
   syms t y z  % define the symbolic variables
   y = cos(t^2)  % chirp signal -- notice no . before ^ since t is no vector
   z = diff(y)  % derivative
   figure(1)
   subplot(211)
   ezplot(y, [0, 2*pi]);grid  % plotting for symbolic y between 0 and 2*pi
   hold on
   subplot(212)
   ezplot(z, [0, 2*pi]);grid
   hold on
%numeric
   Ts = 0.1;  % sampling period
   t1 = 0:Ts:2*pi;  % sampled time
   y1 = cos(t1.^2);  % sampled signal --notice difference with y above
   z1 = diff(y1)./diff(t1);  % difference  -- approximation to derivative
   figure(1)
   subplot(211)
   stem(t1, y1, 'r');axis([0 2*pi 1.1*min(y1) 1.1*max(y1)])
   subplot(212)
   stem(t1(1:length(y1) - 1), z1, 'r');axis([0 2*pi 1.1*min(z1) 1.1*max(z1)])
   legend('Derivative (black)','Difference (blue)')
   hold off
```

The symbolic function syms defines the symbolic variables (use help syms to learn more). The signal $y(t)$ is written differently than $y_1(t)$ in the numeric computation. Since t_1 is a vector, squaring it requires a dot before the symbol. That is not the case for t, which is not a vector but a variable. The results of using diff to compute the derivative of $y(t)$ is given in the same form as you would have obtained doing the derivative by hand—that is,

```
y = cos(t^2)
z = −2*t*sin(t^2)
```

The symbolic toolbox provides its own graphic routines (use help to learn about the different ez-routines). For plotting $y(t)$ and $z(t)$, we use the function ezplot, which plots the above two functions for $t \in [0, 2\pi]$ and titles the plots with these functions.

The numeric computations differ from the symbolic in that vectors are being processed, and we are obtaining an approximation to the derivative $z(t)$. We sample the signal with $T_s = 0.1$ and use again

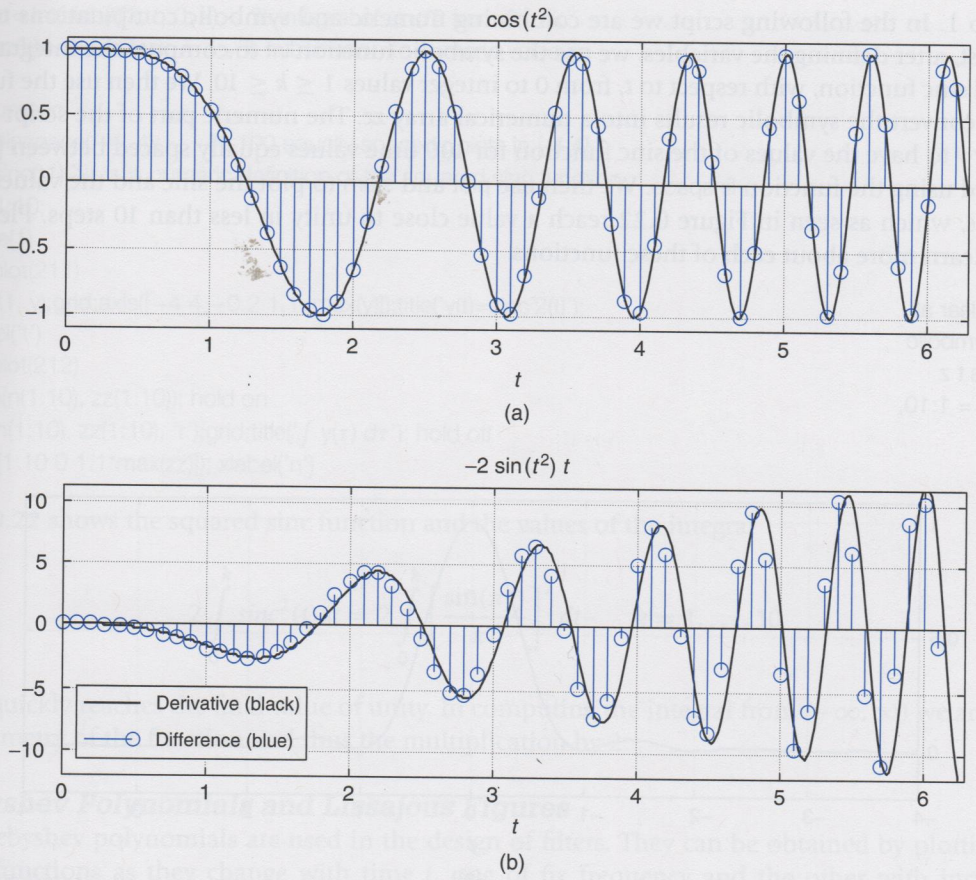

FIGURE 0.21

Symbolic and numeric computation of the derivative of the chirp $y(t) = \cos(t^2)$. (a) $y(t)$ and the sampled signal $y(nT_s)$, $T_s = 0.1$ sec. (b) Displays the exact derivative (continuous line) and the approximation of the derivative at samples nT_s. Better approximation to the derivative can be obtained by using a smaller value of T_s.

the function diff to approximate the derivative (the denominator *diff(t1)* is the same as T_s). Plotting the exact derivative (continuous line) with the approximated one (samples) using stem clarifies that the numeric computation is an approximation at nT_s values of time. See Figure 0.21.

The Sinc Function and Integration

The sinc function is very significant in the theory of signals and systems. It is defined as

$$y(t) = \frac{\sin \pi t}{\pi t} \qquad -\infty < t < \infty$$

It is symmetric with respect to the origin, and defined from $-\infty$ to ∞. The value of $y(0)$ can be found using L'Hôpital's rule. We will see later (Parseval's result in Chapter 5) that the integral of $y^2(t)$ is

equal to 1. In the following script we are combining numeric and symbolic computations to show this. First, after defining the variables, we use the symbolic function int to compute the integral of the squared sinc function, with respect to t, from 0 to integer values $1 \leq k \leq 10$. We then use the function subs to convert the symbolic results into a numerical array zz. The numeric part of the script defines a vector y to have the values of the sinc function for 100 time values equally spaced between $[-4, 4]$, obtained using the function linspace. We then use plot and stem to plot the sinc and the values of the integrals, which as seen in Figure 0.22 reach a value close to unity in less than 10 steps. Please use help to learn more about each of these functions.

```
clf; clear all
% symbolic
syms t z
for k = 1:10,
```

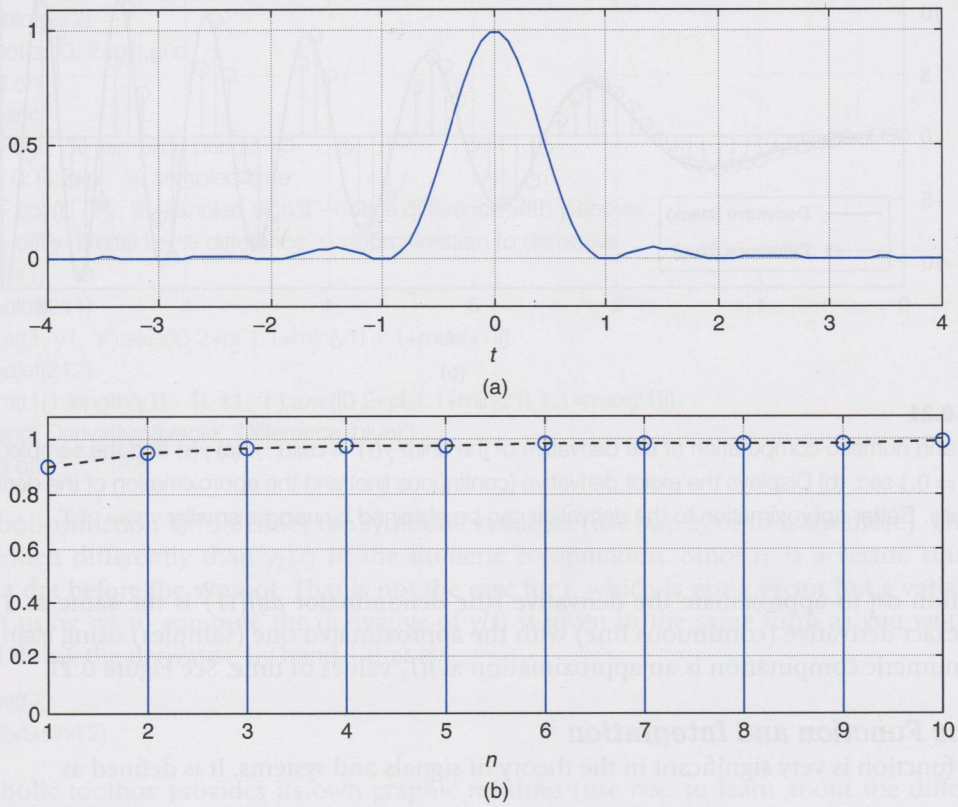

FIGURE 0.22

(a) Computation of the integral of the squared sinc function (b) Illustrates that the area under the curve of this function, or its integral, is unity. Using the symmetry of the function only the integral for $t \geq 0$ needs to be computed.

```
    z = int(sinc(t)^2, t, 0, k);    % integral of sinc^2 from 0 to k
    zz(k) = subs(2*z);              % substitution to numeric value zz
end
% numeric
t1 = linspace(-4, 4);       % 100 equally spaced points in [-4,4]
y = sinc(t1).^2;    % numeric definition of the squared sinc function
n = 1:10;
figure(1)
subplot(211)
plot(t1, y);grid;axis([-4 4 -0.2 1.1*max(y)]);title('y(t)=sinc^2(t)');
xlabel('t')
subplot(212)
stem(n(1:10), zz(1:10)); hold on
plot(n(1:10), zz(1:10), 'r');grid;title('∫ y(τ) dτ'); hold off
axis([1 10 0 1.1*max(zz)]); xlabel('n')
```

Figure 0.22 shows the squared sinc function and the values of the integral

$$2\int_0^k sinc^2(t)dt = 2\int_0^k \left[\frac{\sin(\pi t)}{\pi t}\right]^2 dt \qquad k = 1, \ldots, 10$$

which quickly reaches the final value of unity. In computing the integral from $(-\infty, \infty)$ we are using the symmetry of the function and thus the multiplication by 2.

Chebyshev Polynomials and Lissajous Figures

The Chebyshev polynomials are used in the design of filters. They can be obtained by plotting two cosine functions as they change with time t, one of fix frequency and the other with increasing frequency:

$$x(t) = \cos(2\pi t)$$
$$y(t) = \cos(2\pi kt) \qquad k = 1, \ldots, N$$

The $x(t)$ gives the x axis coordinate and $y(t)$ the y axis coordinate at each value of t. If we solve for t in the top equation, we get

$$t = \frac{1}{2\pi}\cos^{-1}(x(t))$$

which then replaced in the bottom equation gives

$$y(t) = \cos\left[k\cos^{-1}(x(t))\right] \qquad k = 1, \ldots, N$$

as an expression for the Chebyshev polynomials (we will see in Chapter 6 that these equations can be expressed as regular polynomials). Figure 0.23 shows the Chebyshev polynomials for $N = 4$. The following script is used to compute and plot these polynomials.

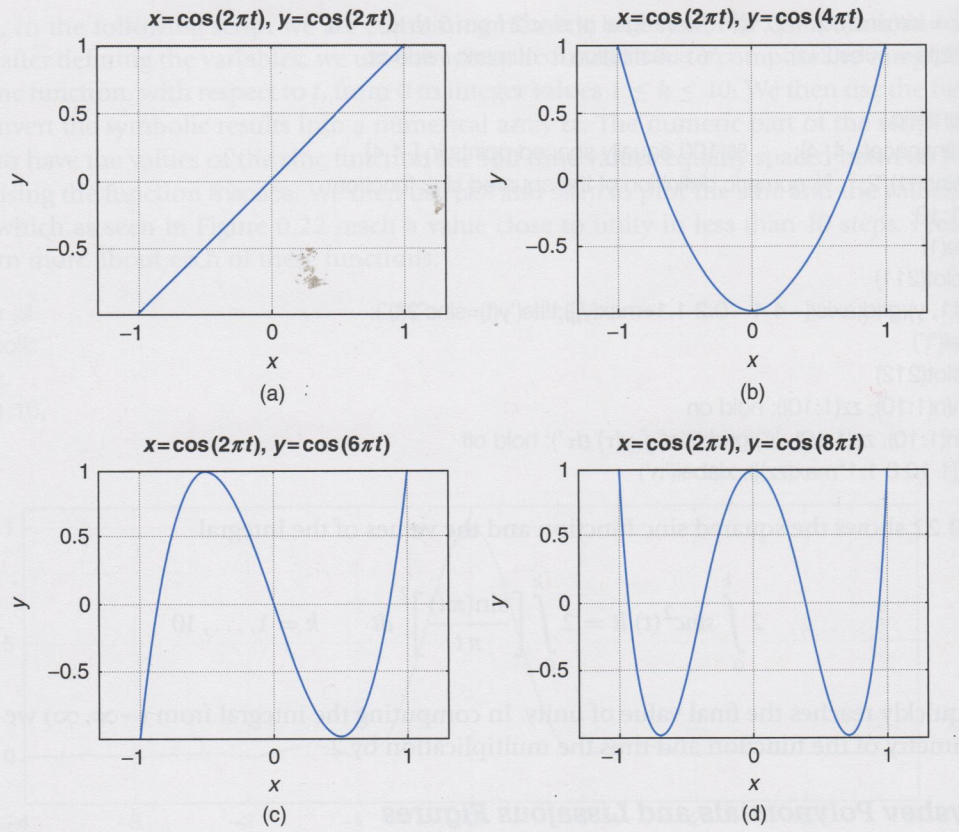

FIGURE 0.23
The Chebyshev polynomials for $n = 1, 2, 3, 4$. First (a) to fourth (d) polynomials. Notice that these polynomials are defined between $[-1, 1]$ in the x axis.

```
clear all;clf
syms x y t
x = cos(2*pi*t); theta=0;
figure(1)
for k = 1:4,
  y = cos(2*pi*k*t + theta);
  if k == 1, subplot(221)
    elseif k == 2, subplot(222)
    elseif k == 3, subplot(223)
    else subplot(224)
  end
  ezplot(x, y);grid;hold on
end
hold off
```

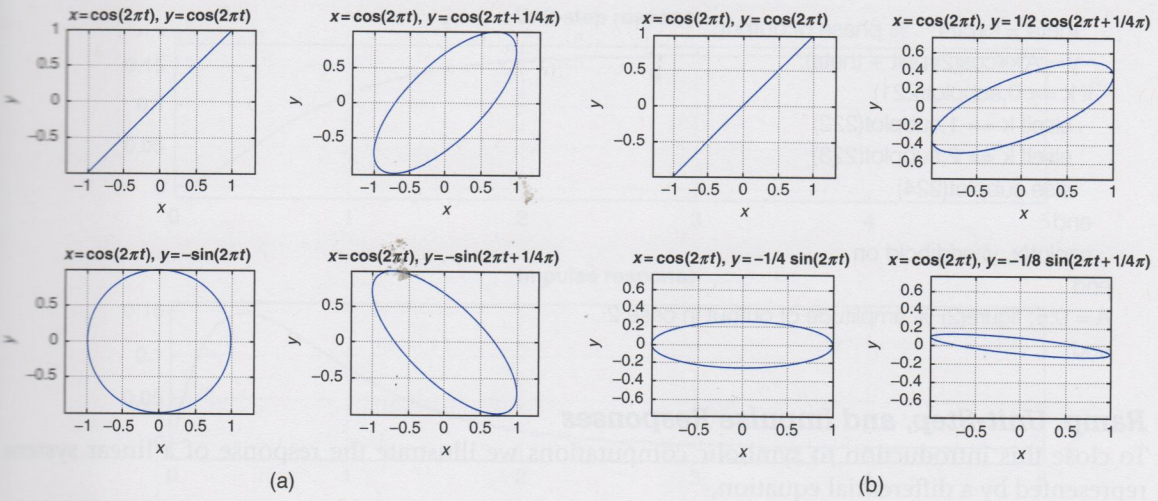

FIGURE 0.24

Lissajous figures: (a) (four left plots) case 1 input and output of same amplitude ($A = 1$) but phase differences of $0, \pi/4, \pi/2$, and $3\pi/4$; (b) (four right plots) case 2 input has unit amplitude but output has decreasing amplitudes and same phase differences as in case 1.

The Lissajous figures we consider next are a very useful extension of the above plotting of sinusoids in the x and y axes. These figures are used to determine the difference between a sinusoidal input and its corresponding sinusoidal steady state. In the case of linear systems, which we will formally define in Chapter 2, for a sinusoidal input the outputs of the system are also sinusoids of the same frequency, but they differ with the input in the amplitude and phase.

The differences in amplitude and phase can be measured using an oscilloscope for which we put the input in the horizontal sweep and the output in the vertical sweep, giving figures from which we can find the differences in amplitude and phase. Two situations are simulated in the following script, one where there is no change in amplitude but the phase changes from zero to $3\pi/4$, while in the other case the amplitude decreases as indicated and the phase changes in the same way as before. The plots, or Lissajous figures, indicate such changes. The difference between the maximum and the minimum of each of the figures in the x axis gives the amplitude of the input, while the difference between the maximum and the minimum in the y axis gives the amplitude of the output. The orientation of the ellipse provides the difference in phase with respect to that of the input. The following script is used to obtain the Lissajous figures in these cases. Figure 0.24 displays the results.

```
clear all;clf
syms x y t
x = cos(2*pi*t); % input of unit amplitude and frequency 2*pi
A = 1;figure(1)  % amplitude of output in case 1
for i = 1:2,
for k = 0:3,
```

```
    theta = k*pi/4;   % phase of output
    y = A^k*cos(2*pi*t + theta);
  if k == 0,subplot(221)
    elseif k == 1,subplot(222)
    elseif k == 2,subplot(223)
    else subplot(224)
  end
  ezplot(x, y);grid;hold on
end
A = 0.5; figure(2) % amplitude of output in case 2
end
```

Ramp, Unit-Step, and Impulse Responses

To close this introduction to symbolic computations we illustrate the response of a linear system represented by a differential equation,

$$\frac{d^2y(t)}{dt^2} + 5\frac{dy(t)}{dt} + 6y(t) = x(t)$$

where $y(t)$ is the output and $x(t)$ the input. The input is a constant $x(t) = 1$ for $t \geq 0$ and zero otherwise (MATLAB calls this function *heaviside*, but we will call it the *unit-step signal*). We then let the input be the derivative of $x(t)$, which is a signal that we will call *impulse*, and finally we let the input be the integral of $x(t)$, which is what we will call the *ramp* signal. The following script is used to find the responses, which are displayed in Figure 0.25.

```
clear all; clf
syms y t x z
% input a unit-step  (heaviside) response
y = dsolve('D2y + 5*Dy + 6*y = heaviside(t)','y(0) = 0','Dy(0) = 0','t');
x = diff(y); % impulse response
z = int(y); % ramp response
figure(1)
subplot(311)
ezplot(y, [0,5]);title('Unit-step response')
subplot(312)
ezplot(x, [0,5]);title('Impulse response')
subplot(313)
ezplot(z, [0,5]);title('Ramp response')
```

This example illustrates the intuitive appeal of linear systems. When the input is a constant value (or a unit-step signal or a heaviside signal) the output tries to follow the input after some initial inertia and it ends up being constant. The impulse signal (obtained as the derivative of the unit-step signal) is a signal of very short duration equivalent to shocking the system with a signal that disappears very fast, different from the unit-step signal that is like a dc source. Again the output tries to follow the input, eventually disappearing as t increases (no energy from the input!), and the ramp that is

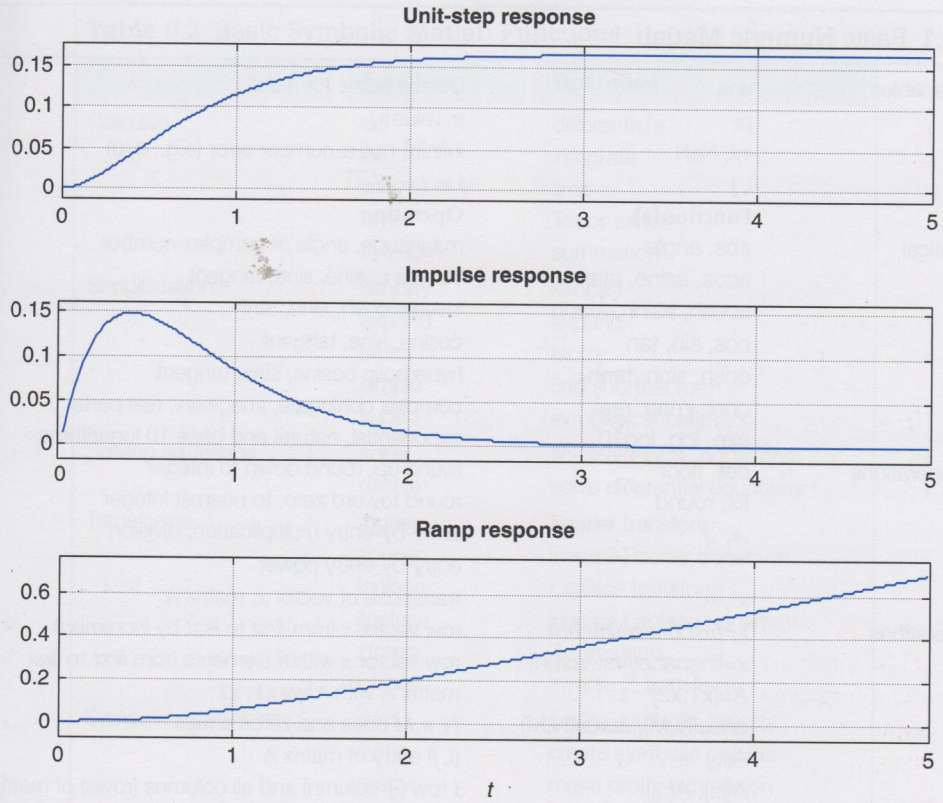

FIGURE 0.25
Response of a second order system represented by a differential equation for input of the unit-step signal, its derivative, or the impulse signal and the ramp signal that is the integral of the unit-step input.

the integral of the unit-step signal grows with time, providing more and more energy to the system as time increases, thus the response we obtained. The function dsolve solves differential equations explicitly given (D stands for the derivative operator, so D is the first derivative and D2 is the second derivative). A second-order system requires two initial conditions, the output and its derivative at $t = 0$.

We hope this introduction to MATLAB has provided you with the necessary background to understand the basic way MATLAB operates, and shown you how to continue increasing your knowledge of it. Your best source of information is the *help* command. Explore the different modules that MATLAB has and you will become quickly convinced that these modules provide a great number of computational tools for many areas of engineering and mathematics. Try it—you will like it! Tables 0.1 and 0.2 provide a listing of the numeric and symbolic variables and operations.

Table 0.1 Basic Numeric Matlab

Special variables	ans	Default name for result
	pi	π value
	inf, NaN	infinity, not-a-number error (e.g., 0/0)
	i, j	$i = j = \sqrt{-1}$
	Function(s)	**Operation**
Mathematical	abs, angle	magnitude, angle of complex number
	acos, asine, atan	inverse cosine, sine, tangent
	acosh, asinh, atanh	inverse cosh, sinh, tanh
	cos, sin, tan	cosine, sine, tangent
	cosh, sinh, tanh	hyperbolic cosine, sine, tangent
	conj, imag, real	complex conjugate, imaginary, real parts
	exp, log, log10	exponential, natural and base 10 logarithms
Special operations	ceil, floor	round up, round down to integer
	fix, round	round toward zero, to nearest integer
	.*, ./	entry-by-entry multiplication, division
	.^	entry-by-entry power
	x', A'	transpose of vector x, matrix A
Array operations	x=*first*:*increment*:*last*	row vector x from *first* to *last* by *increment*
	x=linspace(first,last,n)	row vector x with n elements from *first* to *last*
	A=[x1;x2]	matrix A with rows $x1$, $x2$
	ones(N,M), zeros(N,M)	$N \times M$ ones and zeros arrays
	A(i,j)	(i, j) entry of matrix A
	A(i,:), A(:,j)	i row (j-column) and all columns (rows) of matrix A
	whos	display variables in workspace
	size(A)	(number rows, number of colums) of matrix A
	length(x)	number rows (colums) of vector x
Control flow	for, if, elseif	for loop, if, else-if loop
	while	while loop
	pause, pause(n)	pause and pause n seconds
Plotting	plot, stem	continuous, discrete plots
	figure	figure for plotting
	subplot	subplots
	hold on, hold off	hold plot on or off
	axis, grid	axis, grid of plots
	xlabel, ylabel, title, legend	labeling of axes, plots, and subplots
Saving and loading	save, load	saving and loading data
Information and managing	help	help
	clear, clf	clear variables from memory, clear figures
Operating system	cd, pwd	change directory, current working directory

Table 0.2 Basic Symbolic Matlab Functions

	Function	Operation
Calculus	diff	differentiate
	int	integrate
	limit	limit
	taylor	Taylor series
	symsum	summation
Simplification	simplify	simplify
	expand	expand
	factor	factor
	simple	find shortest form
	subs	symbolic substitution
Solving equations	solve	solve algebraic equations
	dsolve	solve differential equations
Transforms	fourier	Fourier transform
	ifourier	inverse Fourier transform
	laplace	Laplace transform
	ilaplace	inverse Laplace transform
	ztrans	Z-transform
	iztrans	inverse Z-transform
Symbolic operations	sym	create symbolic objects
	syms	create symbolic objects
	pretty	make pretty expression
Special functions	dirac	Dirac or delta function
	heaviside	unit-step function
Plotting	ezplot	function plotter
	ezpolar	polar coordinate plotter
	ezcontour	contour plotter
	ezsurf	surface plotter
	ezmesh	mesh (surface) plotter

PROBLEMS

For the problems requiring implementation in MATLAB, write scripts or functions to solve them numerically or symbolically. Label the axes of the plots, give a title, and use legend to identify different signals in a plot. To save space use subplot to put several plots into one. To do the problem numerically, sample analog signals with a small T_s.