The following is taken from "Linear Systems and Signals" by B. P. Lathi; Oxford University Press , 2nd ed. 2004.

REFERENCES

1. Asimov, Isaac. *Asimov on Numbers.* Bell Publishing, New York, 1982.
2. Calinger, R., ed. *Classics of Mathematics.* Moore Publishing, Oak Park, IL, 1982.
3. Hogben, Lancelot. *Mathematics in the Making.* Doubleday, New York, 1960.
4. Cajori, Florian. *A History of Mathematics,* 4th ed. Chelsea, New York, 1985.
5. Encyclopaedia Britannica. *Micropaedia* IV, 15th ed., vol. 11, p. 1043. Chicago, 1982.
6. Singh, Jagjit. *Great Ideas of Modern Mathematics.* Dover, New York, 1959.
7. Dunham, William. *Journey Through Genius.* Wiley, New York, 1990.

MATLAB SESSION B: ELEMENTARY OPERATIONS

MB.1 MATLAB Overview

Although MATLAB® (a registered trademark of The MathWorks, Inc.) is easy to use, it can be intimidating to new users. Over the years, MATLAB has evolved into a sophisticated computational package with thousands of functions and thousands of pages of documentation. This section provides a brief introduction to the software environment.

When MATLAB is first launched, its command window appears. When MATLAB is ready to accept an instruction or input, a command prompt (>>) is displayed in the command window. Nearly all MATLAB activity is initiated at the command prompt.

Entering instructions at the command prompt generally results in the creation of an object or objects. Many classes of objects are possible, including functions and strings, but usually objects are just data. Objects are placed in what is called the MATLAB workspace. If not visible, the workspace can be viewed in a separate window by typing `workspace` at the command prompt. The workspace provides important information about each object, including the object's name, size, and class.

Another way to view the workspace is the `whos` command. When `whos` is typed at the command prompt, a summary of the workspace is printed in the command window. The `who` command is a short version of `whos` that reports only the names of workspace objects.

Several functions exist to remove unnecessary data and help free system resources. To remove specific variables from the workspace, the `clear` command is typed, followed by the names of the variables to be removed. Just typing `clear` removes all objects from the workspace. Additionally, the `clc` command clears the command window, and the `clf` command clears the current figure window.

Often, important data and objects created in one session need to be saved for future use. The `save` command, followed by the desired filename, saves the entire workspace to a file, which has the .mat extension. It is also possible to selectively save objects by typing `save` followed by the filename and then the names of the objects to be saved. The `load` command followed by the filename is used to load the data and objects contained in a MATLAB data file (.mat file).

Although MATLAB does not automatically save workspace data from one session to the next, lines entered at the command prompt are recorded in the command history. Previous command lines can be viewed, copied, and executed directly from the command history window. From the command window, pressing the up or down arrow key scrolls through previous commands and redisplays them at the command prompt. Typing the first few characters and then pressing the arrow keys scrolls through the previous commands that start with the same characters. The arrow keys allow command sequences to be repeated without retyping.

Perhaps the most important and useful command for new users is `help`. To learn more about a function, simply type `help` followed by the function name. Helpful text is then displayed in the command window. The obvious shortcoming of `help` is that the function name must first be known. This is especially limiting for MATLAB beginners. Fortunately, help screens often conclude by referencing related or similar functions. These references are an excellent way to learn new MATLAB commands. Typing `help help`, for example, displays detailed information on the `help` command itself and also provides reference to relevant functions such as the `lookfor` command. The `lookfor` command helps locate MATLAB functions based on a keyword search. Simply type `lookfor` followed by a single keyword, and MATLAB searches for functions that contain that keyword.

MATLAB also has comprehensive HTML-based help. The HTML help is accessed by using MATLAB's integrated help browser, which also functions as a standard web browser. The HTML help facility includes a function and topic index as well as full text-searching capabilities. Since HTML documents can contain graphics and special characters, HTML help can provide more information than the command-line help. With a little practice, MATLAB makes it very easy to find information.

When MATLAB graphics are created, the `print` command can save figures in a common file format such as postscript, encapsulated postscript, JPEG, or TIFF. The format of displayed data, such as the number of digits displayed, is selected by using the `format` command. MATLAB help provides the necessary details for both these functions. When a MATLAB session is complete, the `exit` command terminates MATLAB.

MB.2 Calculator Operations

MATLAB can function as a simple calculator, working as easily with complex numbers as with real numbers. Scalar addition, subtraction, multiplication, division, and exponentiation are accomplished using the traditional operator symbols `+`, `-`, `*`, `/`, and `^`. Since MATLAB predefines $\texttt{i} = \texttt{j} = \sqrt{-1}$, a complex constant is readily created using cartesian coordinates. For example,

```
>> z = -3-j*4
z = -3.0000 - 4.0000i
```

assigns the complex constant $-3 - j4$ to the variable z.

The real and imaginary components of z are extracted by using the `real` and `imag` operators. In MATLAB, the input to a function is placed parenthetically following the function name.

```
>> z_real = real(z); z_imag = imag(z);
```

When a line is terminated with a semicolon, the statement is evaluated but the results are not displayed to the screen. This feature is useful when one is computing intermediate results, and it allows multiple instructions on a single line. Although not displayed, the results `z_real = -3` and `z_imag = -4` are calculated and available for additional operations such as computing $|z|$.

There are many ways to compute the modulus, or magnitude, of a complex quantity. Trigonometry confirms that $z = -3 - j4$, which corresponds to a 3-4-5 triangle, has modulus $|z| = |-3 - j4| = \sqrt{(-3)^2 + (-4)^2} = 5$. The MATLAB `sqrt` command provides one way to compute the required square root.

```
>> z_mag = sqrt(z_real^2 + z_imag^2)
z_mag = 5
```

In MATLAB, most commands, include `sqrt`, accept inputs in a variety of forms including constants, variables, functions, expressions, and combinations thereof.

The same result is also obtained by computing $|z| = \sqrt{zz^*}$. In this case, complex conjugation is performed by using the `conj` command.

```
>> z_mag = sqrt(z*conj(z))
z_mag = 5
```

More simply, MATLAB computes absolute values directly by using the `abs` command.

```
>> z_mag = abs(z)
z_mag = 5
```

In addition to magnitude, polar notation requires phase information. The `angle` command provides the angle of a complex number.

```
>> z_rad = angle(z)
z_rad = -2.2143
```

MATLAB expects and returns angles in a radian measure. Angles expressed in degrees require an appropriate conversion factor.

```
>> z_deg = angle(z)*180/pi
z_deg = -126.8699
```

Notice, MATLAB predefines the variable `pi` $= \pi$.

It is also possible to obtain the angle of z using a two-argument arc-tangent function, `atan2`.

```
>> z_rad = atan2(z_imag,z_real)
z_rad = -2.2143
```

Unlike a single-argument arctangent function, the two-argument arctangent function ensures that the angle reflects the proper quadrant. MATLAB supports a full complement of trigonometric functions: standard trigonometric functions `cos`, `sin`, `tan`; reciprocal trigonometric functions `sec`, `csc`, `cot`; inverse trigonometric functions `acos`, `asin`, `atan`, `asec`, `acsc`, `acot`; and hyperbolic variations `cosh`, `sinh`, `tanh`, `sech`, `csch`, `coth`, `acosh`, `asinh`, `atanh`, `asech`, `acsch`, and `acoth`. Of course, MATLAB comfortably supports complex arguments for any trigonometric function. As with the `angle` command, MATLAB trigonometric functions utilize units of radians.

The concept of trigonometric functions with complex-valued arguments is rather intriguing. The results can contradict what is often taught in introductory mathematics courses. For example, a common claim is that $|\cos(x)| \leq 1$. While this is true for real x, it is not necessarily true for complex x. This is readily verified by example using MATLAB and the `cos` function.

```
>> cos(j)
ans = 1.5431
```

Problem B.19 investigates these ideas further.

Similarly, the claim that it is impossible to take the logarithm of a negative number is false. For example, the principal value of $\ln(-1)$ is $j\pi$, a fact easily verified by means of Euler's equation. In MATLAB, base-10 and base-e logarithms are computed by using the `log10` and `log` commands, respectively.

```
>> log(-1)
ans = 0 + 3.1416i
```

MB.3 Vector Operations

The power of MATLAB becomes apparent when vector arguments replace scalar arguments. Rather than computing one value at a time, a single expression computes many values. Typically, vectors are classified as row vectors or column vectors. For now, we consider the creation of row vectors with evenly spaced, real elements. To create such a vector, the notation `a:b:c` is used, where `a` is the initial value, `b` designates the step size, and `c` is the termination value. For example, `0:2:11` creates the length-6 vector of even-valued integers ranging from 0 to 10.

```
>> k = 0:2:11
k =  0    2    4    6    8   10
```

In this case, the termination value does not appear as an element of the vector. Negative and noninteger step sizes are also permissible.

```
>> k = 11:-10/3:0
k = 11.0000    7.6667    4.3333    1.0000
```

If a step size is not specified, a value of one is assumed.

```
>> k = 0:11
k =  0    1    2    3    4    5    6    7    8    9   10   11
```

Vector notation provides the basis for solving a wide variety of problems.

For example, consider finding the three cube roots of minus one, $w^3 = -1 = e^{j(\pi+2\pi k)}$ for integer k. Taking the cube root of each side yields $w = e^{j(\pi/3+2\pi k/3)}$. To find the three unique solutions, use any three consecutive integer values of k and MATLAB's `exp` function.

```
>> k = 0:2;
>> w = exp(j*(pi/3 + 2*pi*k/3))
w = 0.5000 + 0.8660i  -1.0000 + 0.0000i   0.5000 - 0.8660i
```

The solutions, particularly $w = -1$, are easy to verify.

Finding the 100 unique roots of $w^{100} = -1$ is just as simple.

```
>> k = 0:99;
>> w = exp(j*(pi/100 + 2*pi*k/100));
```

A semicolon concludes the final instruction to suppress the inconvenient display of all 100 solutions. To view a particular solution, the user must specify an index. In MATLAB, ascending positive integer indices specify particular vector elements. For example, the fifth element of w is extracted using an index of 5.

```
>> w(5)
ans = 0.9603 + 0.2790i
```

Notice that this solution corresponds to $k = 4$. The independent variable of a function, in this case k, rarely serves as the index. Since k is also a vector, it can likewise be indexed. In this way, we can verify that the fifth value of k is indeed 4.

```
>> k(5)
ans = 4
```

It is also possible to use a vector index to access multiple values. For example, index vector `98:100` identifies the last three solutions corresponding to $k = [97, 98, 99]$.

```
>> w(98:100)
ans = 0.9877 - 0.1564i    0.9956 - 0.0941i    0.9995 - 0.0314i
```

Vector representations provide the foundation to rapidly create and explore various signals. Consider the simple 10 Hz sinusoid described by $f(t) = \sin(2\pi 10t + \pi/6)$. Two cycles of this sinusoid are included in the interval $0 \leq t < 0.2$. A vector t is used to uniformly represent 500 points over this interval.

```
>> t = 0:0.2/500:0.2-0.2/500;
```

Next, the function $f(t)$ is evaluated at these points.

```
>> f = sin(2*pi*10*t+pi/6);
```

The value of $f(t)$ at $t = 0$ is the first element of the vector and is thus obtained by using an index of one.

```
>> f(1)
ans = 0.5000
```

Unfortunately, MATLAB's indexing syntax conflicts with standard equation notation.[†] That is, the MATLAB indexing command `f(1)` is not the same as the standard notation $f(1) = f(t)|_{t=1}$. Care must be taken to avoid confusion; remember that the index parameter rarely reflects the independent variable of a function.

MB.4 Simple Plotting

MATLAB's `plot` command provides a convenient way to visualize data, such as graphing $f(t)$ against the independent variable t.

```
>> plot(t,f);
```

[†]Advanced structures such as MATLAB inline objects are an exception.

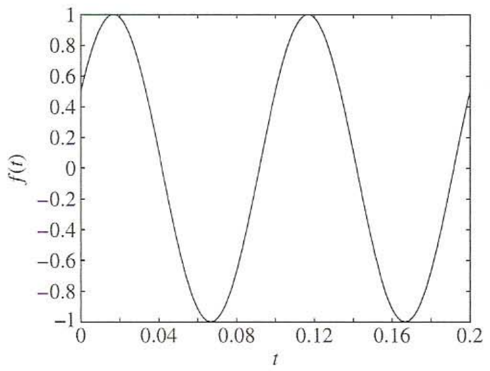

Figure MB.1 $f(t) = \sin(2\pi 10t + \pi/6)$.

Axis labels are added using the `xlabel` and `ylabel` commands, where the desired string must be enclosed by single quotation marks. The result is shown in Fig. MB.1.

```
>> xlabel('t'); ylabel('f(t)')
```

The `title` command is used to add a title above the current axis.

By default, MATLAB connects data points with solid lines. Plotting discrete points, such as the 100 unique roots of $w^{100} = -1$, is accommodated by supplying the `plot` command with an additional string argument. For example, the string `'o'` tells MATLAB to mark each data point with a circle rather than connecting points with lines. A full description of the supported plot options is available from MATLAB's help facilities.

```
>> plot(real(w),imag(w),'o');
>> xlabel('Re(w)'); ylabel('Im(w)');
>> axis equal
```

The `axis equal` command ensures that the scale used for the horizontal axis is equal to the scale used for the vertical axis. Without `axis equal`, the plot would appear elliptical rather than circular. Figure MB.2 illustrates that the 100 unique roots of $w^{100} = -1$ lie equally spaced on the unit circle, a fact not easily discerned from the raw numerical data.

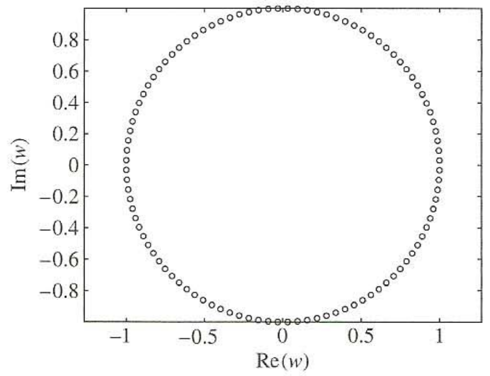

Figure MB.2 Unique roots of $w^{100} = -1$.

MATLAB also includes many specialized plotting functions. For example, MATLAB commands `semilogx`, `semilogy`, and `loglog` operate like the `plot` command but use base-10 logarithmic scales for the horizontal axis, vertical axis, and the horizontal and vertical axes, respectively. Monochrome and color images can be displayed by using the `image` command, and contour plots are easily created with the `contour` command. Furthermore, a variety of three-dimensional plotting routines are available, such as `plot3`, `contour3`, `mesh`, and `surf`. Information about these instructions, including examples and related functions, is available from MATLAB help.

MB.5 Element-by-Element Operations

Suppose a new function $h(t)$ is desired that forces an exponential envelope on the sinusoid $f(t)$, $h(t) = f(t)g(t)$ where $g(t) = e^{-10t}$. First, row vector $g(t)$ is created.

```
>> g = exp(-10*t);
```

Given MATLAB's vector representation of $g(t)$ and $f(t)$, computing $h(t)$ requires some form of vector multiplication. There are three standard ways to multiply vectors: inner product, outer product, and element-by-element product. As a matrix-oriented language, MATLAB defines the standard multiplication operator `*` according to the rules of matrix algebra: the multiplicand must be conformable to the multiplier. A $1 \times N$ row vector times an $N \times 1$ column vector results in the scalar-valued inner product. An $N \times 1$ column vector times a $1 \times M$ row vector results in the outer product, which is an $N \times M$ matrix. Matrix algebra prohibits multiplication of two row vectors or multiplication of two column vectors. Thus, the `*` operator is not used to perform element-by-element multiplication.[†]

Element-by-element operations require vectors to have the same dimensions. An error occurs if element-by-element operations are attempted between row and column vectors. In such cases, one vector must first be transposed to ensure both vector operands have the same dimensions. In MATLAB, most element-by-element operations are preceded by a period. For example, element-by-element multiplication, division, and exponentiation are accomplished using `.*`, `./`, and `.^`, respectively. Vector addition and subtraction are intrinsically element-by-element operations and require no period. Intuitively, we know $h(t)$ should be the same size as both $g(t)$ and $f(t)$. Thus, $h(t)$ is computed using element-by-element multiplication.

```
>> h = f.*g;
```

The `plot` command accommodates multiple curves and also allows modification of line properties. This facilitates side-by-side comparison of different functions, such as $h(t)$ and $f(t)$. Line characteristics are specified by using options that follow each vector pair and are enclosed in single quotes.

```
>> plot(t,f,'-k',t,h,':k');
>> xlabel('t'); ylabel('Amplitude');
>> legend('f(t)','h(t)');
```

[†]While grossly inefficient, element-by-element multiplication can be accomplished by extracting the main diagonal from the outer product of two N-length vectors.

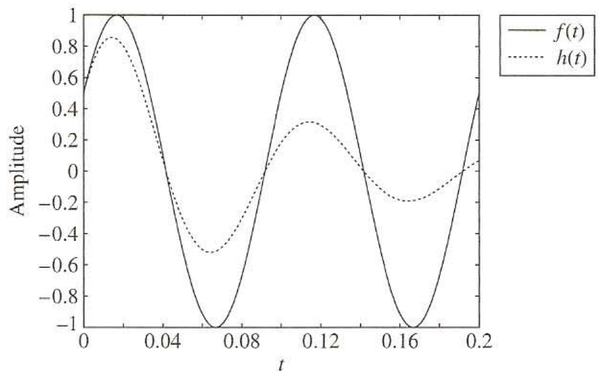

Figure MB.3 Graphical comparison of $f(t)$ and $h(t)$.

Here, `'-k'` instructs MATLAB to plot $f(t)$ using a solid black line, while `':k'` instructs MATLAB to use a dotted black line to plot $h(t)$. A legend and axis labels complete the plot, as shown in Fig. MB.3. It is also possible, although more cumbersome, to use pull-down menus to modify line properties and to add labels and legends directly in the figure window.

MB.6 Matrix Operations

Many applications require more than row vectors with evenly spaced elements; row vectors, column vectors, and matrices with arbitrary elements are typically needed.

MATLAB provides several functions to generate common, useful matrices. Given integers `m`, `n`, and vector `x`, the function `eye(m)` creates the $m \times m$ identity matrix; the function `ones(m,n)` creates the $m \times n$ matrix of all ones; the function `zeros(m,n)` creates the $m \times n$ matrix of all zeros; and the function `diag(x)` uses vector `x` to create a diagonal matrix. The creation of general matrices and vectors, however, requires each individual element to be specified.

Vectors and matrices can be input spreadsheet style by using MATLAB's array editor. This graphical approach is rather cumbersome and is not often used. A more direct method is preferable.

Consider a simple row vector **r**,

$$\mathbf{r} = [1 \quad 0 \quad 0]$$

The MATLAB notation `a:b:c` cannot create this row vector. Rather, square brackets are used to create **r**.

```
>> r = [1 0 0]
r = 1     0     0
```

Square brackets enclose elements of the vector, and spaces or commas are used to separate row elements.

Next, consider the 3×2 matrix **A**,

$$\mathbf{A} = \begin{bmatrix} 2 & 3 \\ 4 & 5 \\ 0 & 6 \end{bmatrix}$$

Matrix **A** can be viewed as a three-high stack of two-element row vectors. With a semicolon to separate rows, square brackets are used to create the matrix.

```
>> A = [2 3;4 5;0 6]
A = 2     3
    4     5
    0     6
```

Each row vector needs to have the same length to create a sensible matrix.

In addition to enclosing string arguments, a single quote performs the complex conjugate transpose operation. In this way, row vectors become column vectors and vice versa. For example, a column vector **c** is easily created by transposing row vector **r**.

```
>> c = r'
c = 1
    0
    0
```

Since vector **r** is real, the complex-conjugate transpose is just the transpose. Had **r** been complex, the simple transpose could have been accomplished by either `r.'` or `(conj(r))'`.

More formally, square brackets are referred to as a concatenation operator. A concatenation combines or connects smaller pieces into a larger whole. Concatenations can involve simple numbers, such as the six-element concatenation used to create the 3×2 matrix **A**. It is also possible to concatenate larger objects, such as vectors and matrices. For example, vector **c** and matrix **A** can be concatenated to form a 3×3 matrix **B**.

```
>> B = [c A]
B = 1     2     3
    0     4     5
    0     0     6
```

Errors will occur if the component dimensions do not sensibly match; a 2×2 matrix would not be concatenated with a 3×3 matrix, for example.

Elements of a matrix are indexed much like vectors, except two indices are typically used to specify row and column.† Element (1, 2) of matrix **B**, for example, is 2.

```
>> B(1,2)
ans = 2
```

Indices can likewise be vectors. For example, vector indices allow us to extract the elements common to the first two rows and last two columns of matrix **B**.

```
>> B(1:2,2:3)
ans = 2     3
      4     5
```

†Matrix elements can also be accessed by means of a single index, which enumerates along columns. Formally, the element from row m and column n of an $M \times N$ matrix may be obtained with a single index $(n-1)M+m$. For example, element (1, 2) of matrix **B** is accessed by using the index $(2-1)3+1=4$. That is, `B(4)` yields 2.

One indexing technique is particularly useful and deserves special attention. A colon can be used to specify all elements along a specified dimension. For example, B(2,:) selects all column elements along the second row of **B**.

```
>> B(2,:)
ans = 0     4     5
```

Now that we understand basic vector and matrix creation, we turn our attention to using these tools on real problems. Consider solving a set of three linear simultaneous equations in three unknowns.

$$\begin{aligned} x_1 - 2x_2 + 3x_3 &= 1 \\ -\sqrt{3}x_1 + x_2 - \sqrt{5}x_3 &= \pi \\ 3x_1 - \sqrt{7}x_2 + x_3 &= e \end{aligned}$$

This system of equations is represented in matrix form according to $\mathbf{Ax} = \mathbf{y}$, where

$$\mathbf{A} = \begin{bmatrix} 1 & -2 & 3 \\ -\sqrt{3} & 1 & -\sqrt{5} \\ 3 & -\sqrt{7} & 1 \end{bmatrix}, \qquad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}, \text{ and } \qquad \mathbf{y} = \begin{bmatrix} 1 \\ \pi \\ e \end{bmatrix}$$

Although Cramer's rule can be used to solve $\mathbf{Ax} = \mathbf{y}$, it is more convenient to solve by multiplying both sides by the matrix inverse of **A**. That is, $\mathbf{x} = \mathbf{A}^{-1}\mathbf{Ax} = \mathbf{A}^{-1}\mathbf{y}$. Solving for **x** by hand or by calculator would be tedious at best, so MATLAB is used. We first create **A** and **y**.

```
>> A = [1 -2 3;-sqrt(3) 1 -sqrt(5);3 -sqrt(7) 1];
>> y = [1;pi;exp(1)];
```

The vector solution is found by using MATLAB's inv function.

```
>> x = inv(A)*y
x = -1.9999
    -3.8998
    -1.5999
```

It is also possible to use MATLAB's left divide operator x = A\y to find the same solution. The left divide is generally more computationally efficient than matrix inverses. As with matrix multiplication, left division requires that the two arguments be conformable.

Of course, Cramer's rule can be used to compute individual solutions, such as x_1, by using vector indexing, concatenation, and MATLAB's det command to compute determinants.

```
>> x1 = det([y,A(:,2:3)])/det(A)
x1 = -1.9999
```

Another nice application of matrices is the simultaneous creation of a family of curves. Consider $h_\alpha(t) = e^{-\alpha t}\sin(2\pi 10t + \pi/6)$ over $0 \le t \le 0.2$. Figure MB.3 shows $h_\alpha(t)$ for $\alpha = 0$ and $\alpha = 10$. Let's investigate the family of curves $h_\alpha(t)$ for $\alpha = [0, 1, \ldots, 10]$.

An inefficient way to solve this problem is create $h_\alpha(t)$ for each α of interest. This requires 11 individual cases. Instead, a matrix approach allows all 11 curves to be computed simultaneously. First, a vector is created that contains the desired values of α.

```
>> alpha = (0:10);
```

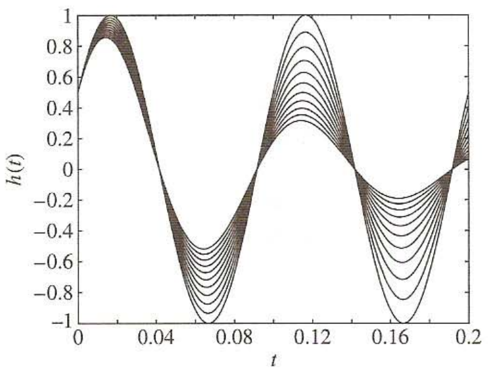

Figure MB.4 $h_\alpha(t)$ for $\alpha = [0, 1, \ldots, 10]$.

By using a sampling interval of one millisecond, $\Delta t = 0.001$, a time vector is also created.

```
>> t = (0:0.001:0.2)';
```

The result is a length-201 column vector. By replicating the time vector for each of the 11 curves required, a time matrix T is created. This replication can be accomplished by using an outer product between t and a 1×11 vector of ones.[†]

```
>> T = t*ones(1,11);
```

The result is a 201×11 matrix that has identical columns. Right multiplying T by a diagonal matrix created from α, columns of T can be individually scaled and the final result is computed.

```
>> H = exp(-T*diag(alpha)).*sin(2*pi*10*T+pi/6);
```

Here, H is a 201×11 matrix, where each column corresponds to a different value of α. That is, $\mathbf{H} = [\mathbf{h}_0, \mathbf{h}_1, \ldots, \mathbf{h}_{10}]$, where $\mathbf{h}_\alpha$ are column vectors. As shown in Fig. MB.4, the 11 desired curves are simultaneously displayed by using MATLAB's plot command, which allows matrix arguments.

```
>> plot(t,H); xlabel('t'); ylabel('h(t)');
```

This example illustrates an important technique called vectorization, which increases execution efficiency for interpretive languages such as MATLAB.[‡] Algorithm vectorization uses matrix and vector operations to avoid manual repetition and loop structures. It takes practice and effort to become proficient at vectorization, but the worthwhile result is efficient, compact code.

MB.7 Partial Fraction Expansions

There are a wide variety of techniques and shortcuts to compute the partial fraction expansion of rational function $F(x) = B(x)/A(x)$, but few are more simple than the MATLAB `residue`

[†]The `repmat` command provides a more flexible method to replicate or tile objects. Equivalently, `T = repmat(t,1,11)`.

[‡]The benefits of vectorization are less pronounced in recent versions of MATLAB.

command. The basic form of this command is

```
>> [R,P,K] = residue(B,A)
```

The two input vectors B and A specify the polynomial coefficients of the numerator and denominator, respectively. These vectors are ordered in descending powers of the independent variable. Three vectors are output. The vector R contains the coefficients of each partial fraction, and vector P contains the corresponding roots of each partial fraction. For a root repeated r times, the r partial fractions are ordered in ascending powers. When the rational function is not proper, the vector K contains the direct terms, which are ordered in descending powers of the independent variable.

To demonstrate the power of the `residue` command, consider finding the partial fraction expansion of

$$F(x) = \frac{x^5+\pi}{(x+\sqrt{2})(x-\sqrt{2})^3} = \frac{x^5+\pi}{x^4-\sqrt{8}x^3+\sqrt{32}x-4}$$

By hand, the partial fraction expansion of $F(x)$ is difficult to compute. MATLAB, however, makes short work of the expansion.

```
>> [R,P,K] = residue([1 0 0 0 0 pi],[1 -sqrt(8) 0 sqrt(32) -4])
R = 7.8888    5.9713    3.1107    0.1112
P = 1.4142    1.4142    1.4142   -1.4142
K = 1.0000    2.8284
```

Written in standard form, the partial fraction expansion of $F(x)$ is

$$F(x) = x + 2.8284 + \frac{7.8888}{x-\sqrt{2}} + \frac{5.9713}{(x-\sqrt{2})^2} + \frac{3.1107}{(x-\sqrt{2})^3} + \frac{0.1112}{x+\sqrt{2}}$$

The signal processing toolbox function `residuez` is similar to the `residue` command and offers more convenient expansion of certain rational functions, such as those commonly encountered in the study of discrete-time systems. Additional information about the `residue` and `residuez` commands are available from MATLAB's help facilities.

PROBLEMS

B.1 Given a complex number $w = x + jy$, the complex conjugate of w is defined in rectangular coordinates as $w^* = x - jy$. Use this fact to derive complex conjugation in polar form.

B.2 Express the following numbers in polar form:

(a) $1 + j$
(b) $-4 + j3$
(c) $(1+j)(-4+j3)$
(d) $e^{j\pi/4} + 2e^{-j\pi/4}$
(e) $e^j + 1$
(f) $(1+j)/(-4+j3)$

B.3 Express the following numbers in Cartesian form:

(a) $3e^{j\pi/4}$
(b) $1/e^j$
(c) $(1+j)(-4+j3)$
(d) $e^{j\pi/4} + 2e^{-j\pi/4}$
(e) $e^j + 1$
(f) $1/2^j$

B.4 For complex constant w, prove:

(a) $\text{Re}(w) = (w + w^*)/2$
(b) $\text{Im}(w) = (w - w^*)/2j$