

An Introduction to MATLAB

Enrique P. Blair, Ph.D.

August 19, 2018

Contents

1	Overview of MATLAB	2
1.1	Installing MATLAB - Students, Staff, and Faculty of Baylor University	2
2	Using MATLAB	3
2.1	The MATLAB User Interface	3
2.2	Using MATLAB as a Calculator	4
2.2.1	MATLAB Functions	5
2.3	Using MATLAB to Run Scripts	6
2.3.1	Creating a New Script	6
2.3.2	Writing a Script	7
2.3.3	Executing a Script	7
3	Matrix Manipulations	8
3.1	Defining Matrices and Vectors	8
3.2	Subscripting Matrices	10
3.3	Subscripting Matrices and Assignment Statements	12
3.4	Concatenation	13
3.5	Single-subscripting	13
3.6	The <code>char</code> Data Type and Strings	14
3.6.1	Strings	15
4	Basic Visualization	16
5	for Loops	19
5.1	Some Useful Non-numerical Functions	19
5.1.1	The <code>disp()</code> Function	19
5.1.2	The <code>num2str()</code> Function	19
5.2	for Loops	19
5.2.1	for-loop Automation	21
5.2.2	for Loops Aren't Always the Best Tool	22
6	Extending MATLAB: Writing User-defined Functions	23
6.1	Writing Functions	23

1 Overview of MATLAB

“MATLAB” is a contraction of “**matrix laboratory**,” and it was designed to allow matrix manipulations. MATLAB is very powerful, allowing users to manipulate, analyze, and visualize data. Users can write functions, classes, and build graphical user interfaces (GUIs).

1.1 Installing MATLAB - Students, Staff, and Faculty of Baylor University

If you are currently enrolled at or affiliated with Baylor University, you can use Baylor’s site license and download/install MATLAB on a university-owned laptop **free of charge**.

To install MATLAB on your own personal laptop:

1. Point a browser to <https://www.baylor.edu/library/techpoint/>
2. Then choose “Software & Student Discounts” → “Software You Can Install”
3. Next, follow the link for “Installation and download instructions HERE”

2 Using MATLAB

In this section, we describe how to begin using MATLAB.

2.1 The MATLAB User Interface

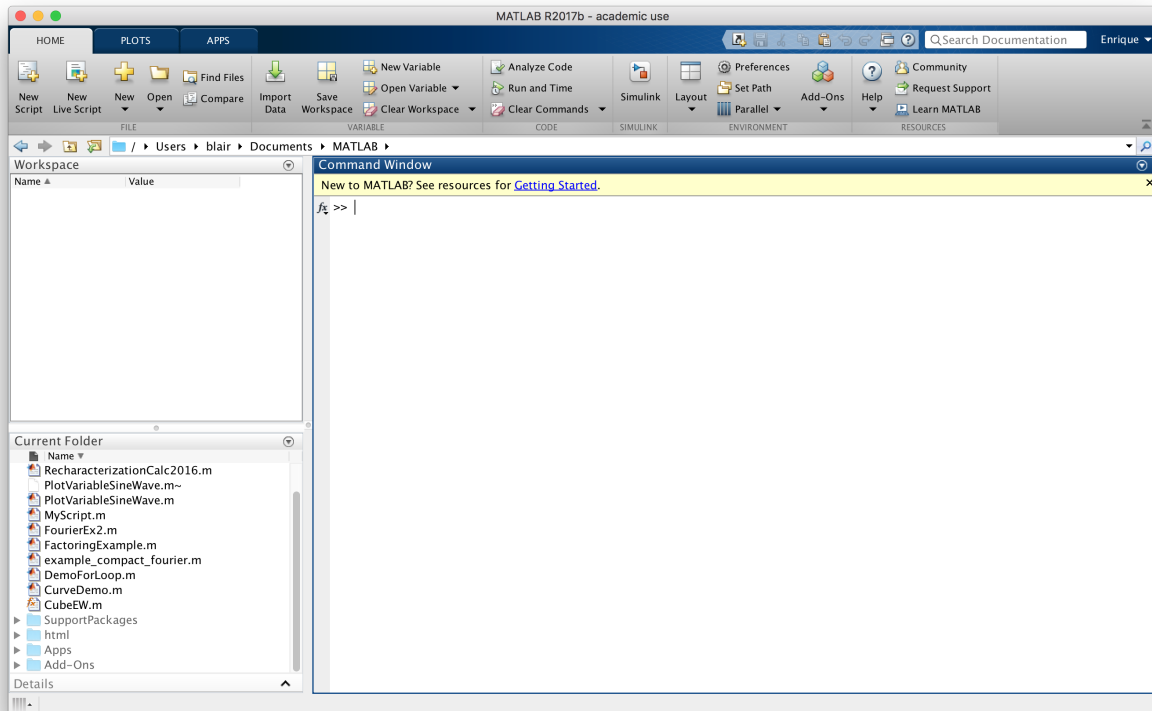



Figure 1: A typical layout for the MATLAB IDE window. It consists of three sub-windows: (1) the Command Window [right]; (2) the Workspace [upper left], and (3) the Current Folder [lower left].

The primary and most commonly-used MATLAB user interface is the MATLAB integrated developing environment (IDE), as shown in Fig. 1. This figure is customizable, but this configuration resembles the default configuration. This configuration has three windows: (1) the Command Window, (2), the Workspace, and (3) the Current Folder. We briefly describe these here:

1. **The Command Window.** A user types MATLAB commands at the “>>” prompt. The Command Window also displays the unsuppressed text output user commands. Commands may be as simple as calculator-like arithmetic to scripts and MATLAB functions.
2. **The Workspace** lists variables, structures, and objects stored in program memory. The contents of program memory are the result of MATLAB commands and calculations.
3. **The Current Folder** lists files and subfolders contained within the current folder. The local path to the current folder is also listed in a toolbar near the top. This is important because text files define MATLAB commands, and MATLAB can only use commands associated with files found in the current folder (listed in the Current Folder window) or found within other folders stored in a special `MATLABPATH` variable. If you attempt to execute a command associated

with a file not found in the current folder or in the MATLABPATH folders, then you will get an error in the Command Window that says “Undefined function or variable ‘xxx’,” where “xxx” is replaced by the offending command name.

2.2 Using MATLAB as a Calculator

Perhaps the simplest way to use MATLAB is to use it as a calculator by typing mathematical operations in the Command Window. For example, at the MATLAB command prompt “>>” typing “5+6” followed by RETURN () causes MATLAB to evaluate the sum of 5 and 6. The MATLAB output is shown here:

```
1 >> 5+6
2
3 ans =
4
5     11
6
7 >>
```

Listing 1: A simple addition calculation in MATLAB returns its result.

In this code listing, line 1 begins with the prompt, “>>”, which is followed by the input typed by the user. MATLAB uses lines 2-6 to report that the answer, 11, is stored in the variable `ans`. The re-emergence of the prompt signifies that MATLAB is ready for a new command.

Another important variation on this command is to assign the result of the mathematical operation to a variable. For example, the following results if we input “`x=5+6`”:

```
1 >> x=5+6
2
3 x =
4
5     11
6
7 >>
```

Listing 2: The result of the previous simple addition in MATLAB (see Listing 2) is stored in variable `x`.

The command of line 1 is not a statement of equality, but rather an assignment command (we call “=” the *assignment operator*). This input instructs MATLAB to evaluate the sum 5+6 and store the result in the variable `x`. In lines 2-6, MATLAB reports the value that is stored in the variable `x`. Line 7 again prompts the user for a new command.

Valid MATLAB variable names may contain letters, numbers and underscores. Valid MATLAB variables may not have any spaces, nor can they begin with numbers.

Often, it is desirable to suppress the result of a MATLAB command. This can be accomplished by adding a semicolon ; to the command, as follows:

```
1 >> x=5+6;
2 >>
```

Listing 3: A semicolon (;) may be used at the end of a MATLAB command to suppress the output.

Here, the result of line 1 is suppressed, and the new MATLAB prompt indicates that MATLAB is ready to receive new command. Additionally, the ; can be used to divide a single line into several individual commands, as follows:

```
1 >> x=5+6;y=x*2
2
3 y =
4
5     22
6
7 >>
```

Listing 4: The result of a simple addition calculation is saved in the variable x , which is used for a follow-on calculation.

Here, the same command is issued as before, and a second command instructs MATLAB to multiply the value stored in x by 2, and to store the result in a second variable y . Since the second command did not end with a ;, its output was not suppressed, and MATLAB reports this result as 22.

2.2.1 MATLAB Functions

At this point, we have used the additive and multiplicative binary operators, “+” and “*”, respectively; as well as the assignment operator “=”. Other valid operators include “-” (subtraction), “/” (division), and “^” (exponentiation). Important functions include `exp()` (the exponential), `sin()` (the sine function), `cos()` (the cosine function) and `tan()` (the tangent function). MATLAB’s trigonometric functions accept arguments in radians, but the functions `sind()`, `cosd()`, and `tand()` are degree-input versions of the other trigonometric functions.

MATLAB also recognizes certain predefined constants. We can use “pi” to represent the irrational number π , and “1i” is used to represent $i = \sqrt{-1}$. Consider the use of these constants with functions:

```
1 >> x = cos(pi/2)
2
3 x =
4
5     6.1232e-17
6
7 >> y = cosd(90)
8
9 y =
10
11     0
12
13 >> z = exp(1i*pi)
14
15 z =
16
17    -1.0000 + 0.0000i
```

Listing 5: Some examples of pre-defined constants in MATLAB.

Here, lines 1 and 7 evaluate the cosine of $\pi/2$ and 90° . The results are stored in “x” and “y”, respectively. Note that `cos(pi/2)` evaluates to a very small number (approximately zero), whereas `cosd(90)` evaluates to zero.

2.3 Using MATLAB to Run Scripts

2.3.1 Creating a New Script

It often is helpful to combine many commands in a single file and execute them in sequence as a script. There are several ways to make a new script. One can do any of the following:

1. Click the “New Script” button in the upper right region of the toolbar.
2. Click the “New Document” button in the upper right region of the toolbar, and then select “Script”
3. Press `⌘` + `N` on Mac (or `ctrl` + `N` on a Windows keyboard)
4. Type `edit` followed by `↵` in the command window.

Each of these actions opens a new, untitled script for editing in a new Editor window, as seen in Fig. 2.

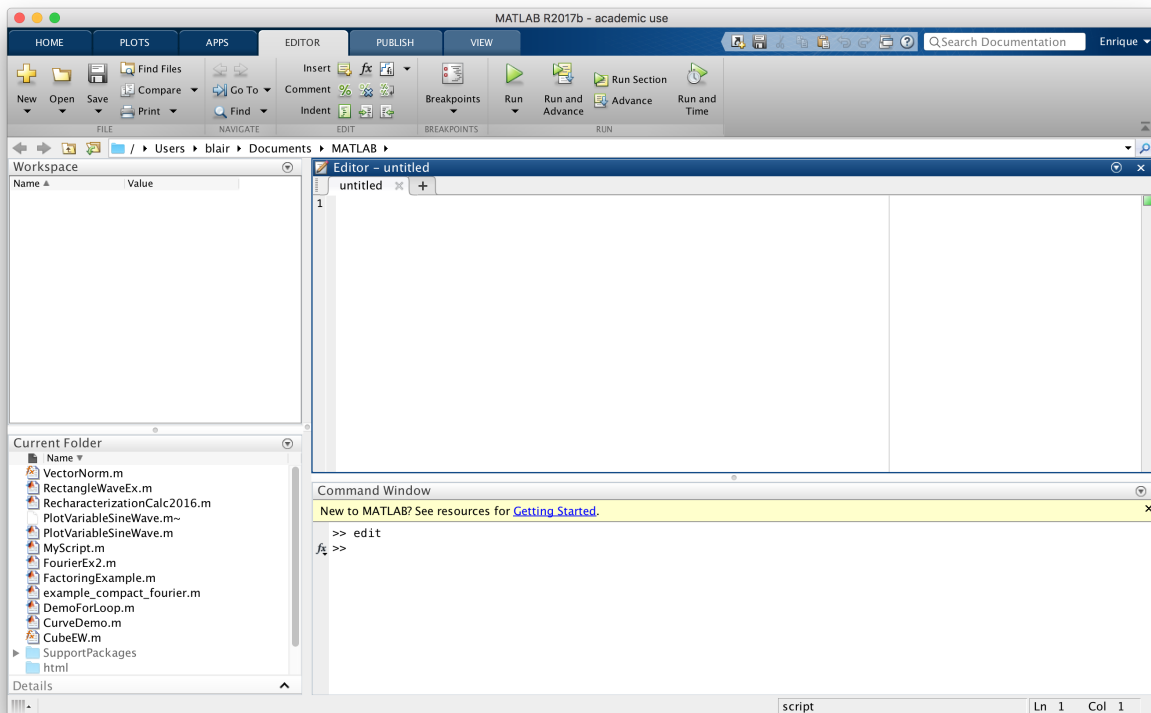


Figure 2: A typical layout for the MATLAB IDE window with an Editor window.

2.3.2 Writing a Script

To write the script, write commands in the script just as in section 2.2. As before, use “;” to suppress output or divide a line into multiple commands. Comments may be added to any line of script by using the “%” symbol. For a given line in the script, any text following a “%” will be ignored and will not be evaluated as a command.

2.3.3 Executing a Script

To execute a script, first make sure that the file containing the script is saved in either the current folder or in a folder listed in the `MATLABPATH` variable. Then, simply type the name of the script exactly (but, without the `.m` extension) in the command line. Alternately, with the Editor as the active window and the desired script as the active tab within the Editor, click the “**Save and Run**” icon (green right-ward-pointing triangle) in the Editor tab of the toolbar.

3 Matrix Manipulations

In MATLAB, all numbers are treated as matrices. The simplest matrix **A** is an $m \times n$ array of numbers, where m refers to the number of (horizontal) rows and n refers to the number of (vertical) columns in **A**. Even when we manipulate a single number, MATLAB treats this single number as a 1×1 matrix.

Matrices are important in solving systems of equations. They also are useful in MATLAB for storing arrays of numerical data. In this context, matrices may also be called “storage arrays.”

3.1 Defining Matrices and Vectors

A matrix **A** can be formed by listing each element row by row within a single set of brackets (“[” and “]”). Within a row, elements may be separated by white space or by a column, and rows are delimited using semicolons. An example of this is given in the following command and output:

```
1 >> A = [1 2 3 4; 5 6 7 8; 9 10 11 12] % this defines a 3x4 matrix A
2
3 A =
4
5     1     2     3     4
6     5     6     7     8
7     9    10    11    12
```

Listing 6: A matrix is defined by enumerating its elements and delineating the matrix with brackets (‘[’ and ‘]’). Individual elements within a row are separated by a white space, or optionally a comma (,). Rows are separated using a semicolon (;).

Alternately, functions such as `rand()`, `eye()`, `zeros()`, `ones()`, and `diag()` may be used to create matrices. Consider the following MATLAB script, entitled “MatrixCreation.m”:

```
1 % MatrixCreation.m
2
3 m = 3; n = 2; B = rand(m) % defines a pseudorandom m-by-m matrix
4
5 C = rand([m,n]) % defines a pseudorandom m-by-n matrix
6
7 In = eye(n) % defines an n-by-n identity matrix
8
9 Znm = zeros(n, m) % defines an n-by-m matrix of zeros
10
11 Omn = ones(m, n) % defines a m-by-n matrix of ones
12
13 D = diag([1 2 3 4 5]) % defines a diagonal matrix with the input
    vector
14                      % on the diagonal
```

Listing 7: Built-in MATLAB functions may be used to create/define matrices.

Line 3 of `MatrixCreation.m` defines a $m \times m$ matrix of pseudorandom numbers using the `rand()` function and stores them in the variable **B**. Another way to use `rand()` is to specify a matrix input, where each element of the input matrix specifies a dimension size (the row specification is first,

and the column specification comes second). This is done in line 5. In line 7, we use `eye(n)` to construct a $n \times n$ identity matrix. Line 9 creates a matrix of zeros, and line 11 creates a matrix of ones. Line 13 creates a diagonal matrix. Running `MatrixCreation.m` results in the following output:

```

1 >> MatrixCreation
2
3 B =
4
5     0.4898     0.7094     0.6797
6     0.4456     0.7547     0.6551
7     0.6463     0.2760     0.1626
8
9
10 C =
11
12     0.1190     0.3404
13     0.4984     0.5853
14     0.9597     0.2238
15
16
17 In =
18
19     1     0
20     0     1
21
22
23 Znm =
24
25     0     0     0
26     0     0     0
27
28
29 Omn =
30
31     1     1
32     1     1
33     1     1
34
35
36 D =
37
38     1     0     0     0     0
39     0     2     0     0     0
40     0     0     3     0     0
41     0     0     0     4     0
42     0     0     0     0     5

```

Listing 8: The Command Window output for an invocation of Listing 7.

Row vectors are simply $m \times n$ matrices with $m = 1$, and column vectors are $m \times n$ matrices with $n = 1$. Here are some ways to construct numerical row vectors:

1. A simple row vector of n integers may be formed by using the syntax “**k:k+n-1**”. For example, the input **2:5** defines a vector **[2 3 4 5]**. This is referred to as a *range* of integers, with unity increment between successive elements. A increment may be specified using “**min:inc:max**”. For example, the input “**2:2:10**” specifies the row vector **[2 4 6 8 10]**
2. A row vector of floating-point double values may be formed using the syntax “**min:inc:max**”, where **min** is the low value, **max** is the lowest value and the first element in the vector, and each successive element larger than the previous value by **inc**. No values larger than **max** are included in the vector.
3. The **linspace()** command can be used to create row vector of n evenly-spaced points that includes the minimum and maximum values **min** and **max** using the syntax “**linspace(min, max, n)**”

3.2 Subscripting Matrices

MATLAB allows us to refer to individual elements of a matrix by using parentheses. If **A** is a matrix, then **A(row_spec, col_spec)** provides a syntax for subscripting a subset of matrix elements. Here, **row_spec** and **col_spec** are row specifier and column specifier expressions, respectively. The **row_spec** and **col_spec** are also called *indices* (plural of the word, “index”), and subscripting is also called “indexing.”

While **row_spec** and **col_spec** must be integers or logical values, there are many ways to specify rows and columns.

- If **row_spec** and **col_spec** both are single integers, **A(row_spec, col_spec)** specifies a single element of **A**.
- A subset of rows or columns may be specified using vectors or ranges. A full row-span or column-span may be achieved by using “**:**” as either the **row_spec** or **col_spec** expression.
- When using a vector as a **row_spec** or **col_spec** expression, the **end** keyword identifies the last valid index for the row or column.

Examples of matrix subscripting are given in the code listing of “**MatrixSubscripting.m**”:

```
1 A = [1 2 3 4; 5 6 7 8; 9 10 11 12] % this defines a 3x4 matrix A
2
3 A34 = A(3,4) % extract the (3,4) element of A
4
5 B = A(2:3,1:3) % extract elements in rows 2-3 and columns 1-3 of A
6
7 v = A(:, 2) % extract elements in all rows and column 2 of A [col.
   vector]
8
9 w = A(3, :) % extract elements row 3 and all columns of A [row
   vector]
10
11 A2end = A(2,end) % extract the last element of row 2
12
```

```

13 A2endm2_end = A(2, end-2:end) % extract the last three elements of
    row 2

```

Listing 9: Some examples of matrix subscripting, the referencing of a subset of matrix elements.

When this script is run, the following output results:

```

1 >> MatrixSubscripting
2
3 A =
4
5     1     2     3     4
6     5     6     7     8
7     9    10    11    12
8
9
10 A34 =
11
12     12
13
14
15 B =
16
17     5     6     7
18     9    10    11
19
20
21 v =
22
23     2
24     6
25    10
26
27
28 w =
29
30     9    10    11    12
31
32
33 A2end =
34
35     8
36
37
38 A2endm2_end =
39
40     6     7     8

```

Listing 10: Command Window output for Listing 9.

3.3 Subscripting Matrices and Assignment Statements

We can assign individual elements or subsets of elements of a matrix by using a subscripted matrix on the left-hand side (LHS) of an assignment statement. Consider the code listing of `SubscriptedAssignments.m`:

```
1 % SubscriptedAssignments.m
2
3 M = zeros(4, 5)
4
5 M(2, 5) = 1
6
7 M(1:2, 2:3) = 1
8
9 M(3:4, 2:4) = rand(2,2)
```

Listing 11: Matrix subscripting may be used to assign values to a subset of matrix elements.

Line 3 defines `M` as a 4×5 matrix of zeros, with the following result:

```
1 M =
2
3      0      0      0      0      0
4      0      0      0      0      0
5      0      0      0      0      0
6      0      0      0      0      0
```

Line 5 of `SubscriptedAssignments.m` stores the value 1 in the $M_{2,5}$, the (2,5) element of `M`, with this output:

```
1 M =
2
3      0      0      0      0      0
4      0      0      0      0      1
5      0      0      0      0      0
6      0      0      0      0      0
```

A single scalar value can be assigned to multiple elements of a matrix `M` simultaneously. As an example of this, line 7 of `SubscriptedAssignments.m` assigns the single scalar value 1 to a 2×2 block of elements in `M`:

```
1 M =
2
3      0      1      1      0      0
4      0      1      1      0      1
5      0      0      0      0      0
6      0      0      0      0      0
```

In general, however, the LHS and right-hand-side (RHS) of an assignment statement must match in size. Line 9 of `SubscriptedAssignments.m` demonstrates this: MATLAB throws an error because the target of the assignment on the LHS of line 9 is a 2×3 block of elements, but the value on the RHS of line 9 is a 2×2 block.

```

1 Subscripted assignment dimension mismatch.
2
3 Error in SubscriptedAssignments (line 9)
4 M(3:4, 2:4) = rand(2,2)

```

3.4 Concatenation

Matrices can be combined via concatenation. Horizontal concatenation is the act of creating a larger matrix **C** from two smaller matrices **A** and **B** by setting **A** and **B** side-by-side. The syntax for horizontal concatenation is **{C = [A B]}**, or equivalently, **{C = [A, B]}**. This involves listing the constituent matrices within brackets (**[** and **]**) and separating them with either a comma (**,**) or a white space. An example of this is:

```

1 >> A = [1 2 3]; B = [4 5 6]; C = [A B]
2
3 C =
4
5      1      2      3      4      5      6

```

Listing 12: Matrices may be concatenated horizontally by using brackets and either a comma or a white space to delimit individual elements.

In the input command line, we define **A** and **B** as row vectors, which themselves are horizontal concatenations of three numbers, each. Then, we form **C** by horizontally concatenating **A** and **B**. Horizontal concatenation requires that constituents have the same number of rows. Concatenation can be thought of as making a matrix of matrices.

Vertical concatenation is accomplished in the same way as horizontal concatenation, except that constituent matrices must have the same number of columns, and constituent matrices must be separated by a semicolon (**;**). The following listing forms the matrix **D** by vertically concatenating **A** and **B**:

```

1 >> D = [A; B]
2
3 D =
4
5      1      2      3
6      4      5      6

```

Listing 13: Matrices may be concatenated vertically by using brackets and either a semicolon to delimit individual elements.

3.5 Single-subscripting

Matrices also may be subscripted with a single index, instead of using both **row_spec** and **col_spec** specifiers. For an $m \times n$ matrix **A**, the element **A(1)** is the $A_{1,1}$ element in the upper left. MATLAB is known as a *column-major* language, in which elements are counted down the columns. Thus, **A(2)** refers to $A_{2,1}$, which is the element immediately below $A_{1,1}$ in column 1; and **A(3)** refers to $A_{2,1}$. When the end of the column is reached, the counting continues at the top of the next column.

A simple application of this is the subscripting of a vector **A**. The expression **A(k)** refers to the k -th element of **A**, regardless of whether **A** is a row vector or a column vector.

Example 3.1. Example Define a row vector \mathbf{t} that includes $n_t = 75$ time points from $t = 0$ to $t = 1$ (inclusive). Then, define a matrix \mathbf{fn} to represent $f_n(t) = \cos(2\pi nt)$ over the interval $t \in [0, 1]$ for $n \in \{1, 2, 3, 4\}$.

Solution The solution is given in the code listing of `Sinusoids.m`.

```
1 % Sinusoids.m
2 %
3 % By E.P. Blair
4 % Baylor University
5 %
6
7 nt = 75; % number of time points
8 t = linspace(0, 1, nt); % [s] define a time vector
9
10 fn = zeros(4, nt); % Define a storage array to hold fn calculations
11                     % The n^th row of fn stores fn = cos(2*pi*n*t)
12
13 % calculate fn and store fn for each value of n in the n^th row
14 fn(1,:) = cos(2*pi*t); % n = 1
15 fn(2,:) = cos(2*pi*2*t); % n = 2
16 fn(3,:) = cos(2*pi*3*t); % n = 3
17 fn(4,:) = cos(2*pi*4*t); % n = 4
```

So far, we have discussed MATLAB numerical data. Numbers in MATLAB are double-precision floating-point values, or simply `double`. MATLAB uses 8 bytes of memory to store each one. When storing complex numbers, MATLAB uses two doubles for each individual number: one for the real part, and one for the complex part.

3.6 The char Data Type and Strings

Now we consider data of type `char`, which is short for “character.” MATLAB `chars` store individual letters (uppercase [A-Z] or lowercase [a-z]), numerals (0-9), and special characters (~, !, @, #, etc.). Data of type `char` may be stored in a variable using an assignment statement. In MATLAB, we identify a character using a pair of single quotes, as follows in this MATLAB command line input and resulting output:

```
1 >> x = '1'
2
3 x =
4
5     '1'
6
7 >> y = 1
8
9 y =
10
11     1
12
13 >>
```

*Listing 14: Simple definition of **char** variables in the Command Window.*

Here, we have stored the **char** data '1' in the variable **x** (line 1, with output on line 3). We contrast this with the case where we store numerical (**double** data (the value 1) in the variable **y** (line 7, with output on line 9). The **char** data '1' is distinguished from the **double** data 1 by the single quotes.

3.6.1 Strings

A string is simply an array of **chars**. A string may be defined using the following syntax:

```
1 >> x = 'hello world'
2
3 x =
4
5     'hello world'
```

*Listing 15: Simple definition of a **char** string and its output.*

It is most typical to work with strings as row vectors, but column vectors and general matrices also may be constructed. Matrix concatenations are available for strings:

```
1 >> x = 'cat'; y = 'dog'; z = [x y]
2
3 z =
4
5     'catdog'
6 >> w = [x; y]
7
8 w =
9
10    2 3 char array
11
12    'cat'
13    'dog'
```

Listing 16: Horizontal and vertical string concatenations.

4 Basic Visualization

The code of Example 3.1 executes, but we have not yet shown a way to visualize the result to verify the calculation. To do this, we introduce basic plotting.

A very useful command is the `plot()` command, which can make line graphs. One very general syntax for `plot()` is of the form `plot(X1, Y1, X2, Y2, X3, Y3, ...)`, where X_k and Y_k are vectors of equal length storing x -data and y -data, respectively, with $k \in \{1, 2, 3, \dots\}$. To adapt this syntax to the code of Example 3.1, we use `t` in the place of each of X_1, X_2, X_3 , and X_4 . Also, we use `fn(:,k)` in place of Y_k .

```
1 plot( t, fn(:,1), t, fn(:,2), t, fn(:,3), t, fn(:,4) );
```

In the case of Example 3.1, however, X_1, X_2, X_3 , and X_4 are indentially equal to the `t` vector; additionally, all the sets of y -data are stored as rows of the `fn` matrix. In this case, we can use the simpler syntax

```
1 plot( t, fn);
```

The result of this listing or the prior one are equivalent. We append this visualization code to the end of `Sinusoids.m`, resulting in a new script, `Sinusoids_v01.m`:

```
1 % Sinusoids_v01.m
2 %
3 % This is the same as Sinusoids.m, but adds visualization.
4 %
5 % By E.P. Blair
6 % Baylor University
7 %
8
9 nt = 75; % number of time points
10 t = linspace(0, 1, nt); % [s] define a time vector
11
12 fn = zeros(4, nt); % Define a storage array to hold fn calculations
13                     % The n^th row of fn stores fn = cos(2*pi*n*t)
14
15 % calculate fn and store fn for each value of n in the n^th row
16 fn(1,:) = cos(2*pi*t); % n = 1
17 fn(2,:) = cos(2*pi*2*t); % n = 2
18 fn(3,:) = cos(2*pi*3*t); % n = 3
19 fn(4,:) = cos(2*pi*4*t); % n = 4
20
21 plot(t, fn);
```

The result of this listing is shown in Fig. 3(a). Here, we see the set of sinusoids represented by $f_n(t)$ plotted for $n \in \{1, 2, 3, 4\}$. However, this plot may be improved in several ways:

1. The sinusoids are not smooth: they look as though they are only approximated using straight line segments. `plot` requires labels that help assign meaning to the data.
2. This plot requires labels that help assign meaning to the data.
 - (a) The x -axis and y -axis both need *labels*.

- (b) There should be a *legend* to disambiguate the various lines in the plot.
- 3. A grid may be added to assist in reading the graph.
- 4. The data lines themselves may be thickened for improved visibility.

These changes can be achieved by using `Sinusoids_v02.m`:

```

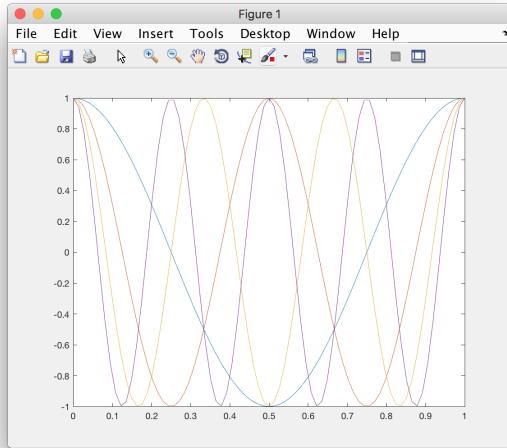
1 % Sinusoids_v02.m
2 %
3 % This is the same as Sinusoids_v01.m, but improves the
  % visualization.
4 %
5 % By E.P. Blair
6 % Baylor University
7 %
8
9 nt = 150; % number of time points
10 t = linspace(0, 1, nt); % [s] define a time vector
11
12 fn = zeros(4, nt); % Define a storage array to hold fn calculations
13 % The nth row of fn stores fn = cos(2*pi*n*t)
14
15 % calculate fn and store fn for each value of n in the nth row
16 fn(1,:) = cos(2*pi*t); % n = 1
17 fn(2,:) = cos(2*pi*2*t); % n = 2
18 fn(3,:) = cos(2*pi*3*t); % n = 3
19 fn(4,:) = cos(2*pi*4*t); % n = 4
20
21 plot(t, fn, 'LineWidth', 2); % thickens the data lines
22 grid on; % makes the grid visible
23 set(gca, 'FontSize', 18, 'FontName', 'Times');
24 xlabel('$t$ (s)', 'Interpreter', 'latex')
25 ylabel('$f_n(t)$', 'Interpreter', 'latex')
26 sine_lgnd = legend('$f_1$', '$f_2$', '$f_3$', '$f_4$')
27 sine_lgnd.Interpreter = 'latex';

```

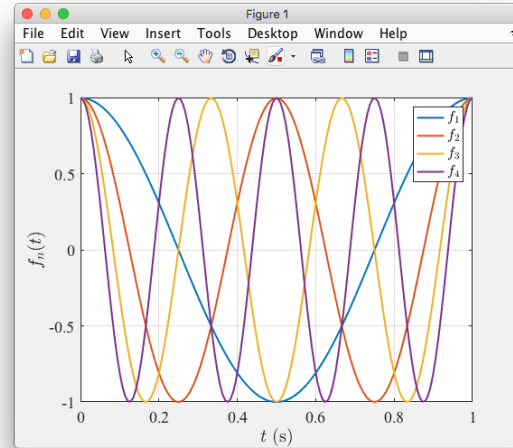
Listing 17: Listing of the `Sinusoids_v02.m` script.

In this listing, line 9 increases the number of data points so that the plots become smoother. Line 21 uses optional inputs to `plot()` in a **property-value** pair, specifically, `'LineWidth', 2` to increase the weight of the lines drawn. Line 22 turns the grid on. Line 23 uses `gca` (get current axes) to get a handle to the current axes. This handle, then is used within the `set()` function to modify properties of the current axes using property-value pairs. Specifically, we set the font size to 18, and the font name to `'Times'`. Lines 24 and 25 generate the labels for the x - and y -axes. Notice here that the first argument to `xlabel()` or `ylabel()` is the desired string, and that some of the string is enclosed in dollar signs ($\$$). These tell MATLAB to use \LaTeX syntax to mathematically typeset the expression or equation contained inside the paired $\$$ signs. In order for this to work, we can to use the property value pair `'Interpreter', 'latex'` to indicate that MATLAB needs to perform some \LaTeX typesetting. In line 26, we use the `legend()` command to create the legend, and we

save the handle by which we reference the legend object by assigning it to a variable, `sine_lgnd`. The inputs to `legend()` are character strings, and again, we used \LaTeX formatting. In order for this to work, we set the `Interpreter` property of `sine_lgnd` to `'latex'`. The result of this code is shown in Fig. 3(b).



(a) Output for listing of `Sinusoids_v01.m`.



(b) Output for listing of `Sinusoids_v02.m`.

Figure 3: Plots of the sinusoids calculated in Example 3.1.

5 for Loops

5.1 Some Useful Non-numerical Functions

We briefly introduce some MATLAB functions that will help us illustrate a `for` loop.

5.1.1 The `disp()` Function

MATLAB comes with some helpful non-numerical functions. For example, the `disp()` function prints a string to the output. Here are two example uses of the `disp()` function:

```
1 >> disp('hello world')
2 hello world
3 >> someStr = 'hello world'; disp(someStr)
4 hello world
```

In the first example (line 1), we provide the string `'hello world'` within the input to `disp()` itself. In the second example (line 3), we first store `'hello world'` in the variable `someStr`, and then we pass `someStr` to the `disp()` function. In both cases, the output is the same.

5.1.2 The `num2str()` Function

If we wish to print a number, we first convert it to a string using the `num2str()` function.

```
1 >> b = num2str('42')
2
3 b =
4
5     '42'
6
7 >> k = 27; disp(['k = ', num2str(k)])
8 k = 27
9 >>
```

In this example, we convert the number 42 to the string `'42'`, and store it in the variable `b` (line 1). Then, in line 7, we assign the number 27 to the variable `k`. We append the string representing the value of `k` onto the string `'k = '`, and we pass this to `disp()`, with the result of line 8.

5.2 for Loops

A `for` loop has the following syntax:

```
1 for variable = expr, statement, ..., statement END
```

It begins with the key word `for` and ends with the key word `end`. The placeholder `expr` usually is an expression that defines or references a matrix or an array. The placeholder `statement, ..., statement` represents the body of the `for` loop. The body is executed as many times as dictated by the instruction `for variable = expr`. In each iteration, the variable `variable`—known as the `for`-loop variable—references some subset of `expr` and may be manipulated within the body.

A more typical and readable format for a `for` loop is as follows:

```

1 for variable = expr
2     statement
3     ...
4     statement
5 end

```

This produces the same result as the previous listing, but the structure of the loop and the body are (desirably) more clearly visible to a reader.

If `expr` is an array, then the `for` loop iterates once for each element of `expr`. In the k -th iteration of the the loop, `variable` is accessible and takes the value of the k -th element of `expr`.

Consider the following listing, in which we define `v` as a 1×5 array of type `double` (`v = [1 2 3 4 5]`), and we make MATLAB display each element, one at a time:

```

1 % basicForLoop.m
2 % Demonstrates a basic for loop
3
4 v = 1:5;
5 for k = v
6     disp(['The value of k is ', num2str(k)]);
7 end

```

For each loop iteration, the value of `k` is displayed, and `k` takes the value of a different element of `v` in each iteration.

```

1 >> basicForLoop
2 The value of k is 1
3 The value of k is 2
4 The value of k is 3
5 The value of k is 4
6 The value of k is 5

```

In the following listing, `basicForLoop_v01.m` a modified loop vector `v` starts with `v_idx = v(1)` (starting at five), and each successive `v(k)` is two less than the previous value. This continues as long as `v_idx` is greater than equal to the far end of `v`. The code of `basicForLoop_v01.m` is as follows:

```

1 % basicForLoop_v01.m
2 % Demonstrates a basic for loop
3
4 v = 5:-2:-12;
5 for k = v
6     disp(['The value of k is ', num2str(k)]);
7 end

```

This yields the result

```

1 >> basicForLoop_v01
2 The value of k is 5
3 The value of k is 3
4 The value of k is 1

```

```

5 The value of k is -1
6 The value of k is -3
7 The value of k is -5
8 The value of k is -7
9 The value of k is -9
10 The value of k is -11

```

5.2.1 for-loop Automation

The listing `Sinusoids_v02.m` (Listing 17) may be written elegantly using a `for` loop. This is accomplished in `Sinusoids_v03.m`:

```

1 % Sinusoids_v03.m
2 %
3 % This is the same as Sinusoids_v02.m, but uses a for loop.
4 %
5 % By E.P. Blair
6 % Baylor University
7 %
8
9 N = 4; % number of sinusoids (maximum value of N)
10 nt = 150; % number of time points
11 t = linspace(0, 1, nt); % [s] define a time vector
12
13 fn = zeros(4, nt); % Define a storage array to hold fn calculations
14                     % The nth row of fn stores fn = cos(2*pi*n*t)
15
16 % calculate fn and store fn for each value of n in the nth row
17 for n_idx = 1:N
18     fn(n_idx,:) = cos(2*pi*n_idx*t);
19 end
20
21 plot(t, fn, 'LineWidth', 2); % thickens the data lines
22 grid on; % makes the grid visible
23 set(gca, 'FontSize', 18, 'FontName', 'Times');
24 xlabel('$t$ (s)', 'Interpreter', 'latex')
25 ylabel('$f_n(t)$', 'Interpreter', 'latex')
26 sine_lgnd = legend('$f_1$', '$f_2$', '$f_3$', '$f_4$')
27 sine_lgnd.Interpreter = 'latex';

```

Listing 18: The functionality of `Sinusoids_v02.m` is duplicated here, this time using a `for` loop.

Lines 15-19 of Listing 17 map to Lines 17-19 here. While the gain is not impressive, it can be helpful, especially if we must repeat a task tens, hundreds, or thousands of times. It makes code much easier to read when `for` loops are used.

Some best practices when using `for` loops include the use of a meaningful `for`-loop variable, and using indentation within the loop to distinguish the body from the beginning and end, as well as from code outside of the loop. Finally, if we do not necessarily know how long a vector is, and we wish to operate on each element, we can use the `(length())` function:

```

1 ...
2 for x_idx = 1:length(x)
3     statements
4 end

```

Listing 19: The `length()` function can be used to determine the number of times a `for` loop should iterate to operate on each value of the vector \mathbf{x} .

5.2.2 `for` Loops Aren't Always the Best Tool

`for` loops have their place, but they are not necessarily the best tool for certain tasks. For example, if we define \mathbf{x} using `x = 1:7`, we could use a `for` loop to calculate the elements of \mathbf{y} , for which $y(k) = x(k)^2$, but it will be faster and more elegant if we use the `.`² (element-wise exponentiation) operator. This is illustrated in Listing 20, below.

```

1 % poor_for_usage.m
2
3 x = 1:7;
4 y = zeros(1, length(x)); % pre-define a storage vector for the
   result
5 for x_idx = 1:length(x)
6     y(x_idx) = x(x_idx)^2;
7 end
8 y
9
10 y2 = x.^2

```

Listing 20: Element-wise operations may be faster and more elegant than using a `for` loop. Here, we use a `for` loop to square every element of \mathbf{x} and store the result in \mathbf{y} (lines 5-7). Alternately, we use an element-wise exponentiation to square each element of \mathbf{x} and save the results in $\mathbf{y2}$. The element-wise exponentiation is faster, more efficient, clearer, and more concise than the `for`-loop approach.

This produces the following output

```

1 >> poor_for_usage
2
3 y =
4
5     1     4     9    16    25    36    49
6
7
8 y2 =
9
10     1     4     9    16    25    36    49

```

Listing 21: The output of Listing 20 shows a case where an element-wise operation yields the same result as a `for`-loop execution.

6 Extending MATLAB: Writing User-defined Functions

A powerful way to extend a programming language is to write functions. A well-written function serves as a robust piece of code that can be seamlessly and simply used repeatedly and in different contexts with an invocation to the function. A simple function invocation can belie very complex inner workings, which a user need not reprogram or copy and paste. Complex software can be built in a clear and concise way from robust, rigorously-tested functions.

A function performs a specific task. It may receive input parameters or *input arguments* that determine the results of the function. Some output values—called *output arguments*—may be returned by the function.

6.1 Writing Functions

A MATLAB function is typically defined in a text file with the extension `*.m`. Functions also may be defined within a script, but in this case, they are only accessible within that script and must appear at the bottom of the script. The former method provides more flexibility and power than the latter, so we prefer to give each function its own appropriately-named `*.m` file. Here are some salient features of such a function definition file.

1. The name of the `*.m` file must match the function name exactly (MATLAB is case-sensitive).
2. The file begins with the keyword `function`, followed by the name of the function. This first line of the function is called the *header*, and it specifies how the user interacts with the function. The syntax of the function header (with `end`) is given below:

```
1 function [out1, out2] = functionName( in1, in2, in3 )
2     statements
3 end
```

Here, the function `functionName` is a three-input, two-output function.

3. The file ends with the keyword `end`.
4. The *body* consists of the statements that specify the function implementation. The body is written between the header and the closing `end` statement. There may be only very few statements in the body, or several hundreds or even thousands of lines of code in a function body.
5. The function header defines the inputs required by the function, as well as the outputs provided by the function.
 - (a) Functions may be designed with no input or output, or few inputs and outputs, or even variable inputs and outputs.
 - (b) Functional inputs and outputs are called **arguments** or **parameters**. The parameters defined in the header are to be used within the function body.
6. Commented lines of code immediately following the header provide function help documentation. When you type `help functionName` in the command line, the function help you defined appears in the command line.