

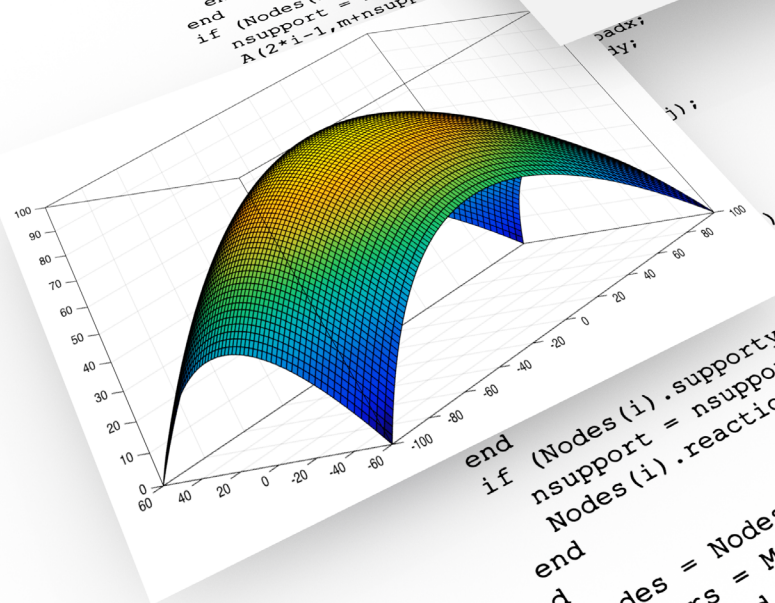
New
Now contains more
detailed descriptions

PROGRAMMING AND ENGINEERING COMPUTING WITH MATLAB® 2017

```
function [outNodes, outMembers] = solveTruss (Nodes, Members)  
n = size (Nodes, 2); m = size (Members, 2);  
if (m/3) < 2*n  
disp ('Unstable!');  
outNodes = 0; outMembers = 0; return  
elseif (m/3) > 2*n, indeterminate!');  
disp ('Statically indeterminate!');  
outNodes = 0; outMembers = 0; return  
end  
outNodes = zeros (2*n, 2*n); nsupport = 0;  
for i = 1:n  
A = zeros (2*n, 2*n);  
for j = 1:m  
if Members (j).node1 ==  
n1 = i; n2 = Membe  
if Members (j).no  
n1 = i; n2 = Membe  
elseif Members (j).no  
n1 = i; n2 = Membe  
end  
x1 = Nodes (n1).x; y1  
x2 = Nodes (n2).x; y2  
L = sqrt ((x2-x1)^2 + (y2-y1)^2);  
A (2*i-1, j) = (x2-x1)/L;  
A (2*i, j) = (y2-y1)/L;  
end  
end (Nodes (i).support == 1)  
if (Nodes (i).support == 1)  
nsupport = nsupport+1;  
A (2*i-1, m+nsupport) = 1;  
A (2*i, m+nsupport) = 0;  
end
```

Node No.	LoadX	LoadY	ReactionX	ReactionY
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0
4	0	0	0	0
5	0	0	0	0
6	0	0	0	0

Member No.	Node1	Node2	Force
1	1	2	0
2	1	3	0
3	2	4	0
4	3	4	0
5	4	5	0
6	4	6	0



```
end  
if (Nodes (i).support == 1)  
nsupport = nsupport+1;  
Nodes (i).reaction = forces (m+nsupport);  
end  
end  
outNodes = Nodes;  
outMembers = Members;  
disp ('Solved successfully.')
```

Huei-Huang Lee

Visit the following websites to learn more about this book:



[amazon.com](https://www.amazon.com)

[Google books](https://books.google.com)

[BARNES & NOBLE](https://www.barnesandnoble.com)

Chapter 2

Data Types, Operators, and Expressions

An expression is a syntactic combination of **numbers**, **variables**, **operators**, and **functions**. An expression always results in a **value**. The right-hand side of an assignment statement is always an expression. You may notice that most of the statements we demonstrated in Chapter 1 are assignment statements. It is fair to say that expressions are the most important building block of a program.

2.1	Unsigned Integers	69
2.2	Signed Integers	72
2.3	Floating-Point Numbers	74
2.4	Character and Strings	78
2.5	Logical Data	81
2.6	Arrays	84
2.7	Sums, Products, Minima, and Maxima	89
2.8	Arithmetic Operators	92
2.9	Relational and Logical Operators	99
2.10	String Manipulations	102
2.11	Expressions	105
2.12	Example: Function Approximations	108
2.13	Example: Series Solution of a Laplace Equation	113
2.14	Example: Deflection of Beams	115
2.15	Example: Vibrations of Supported Machines	117
2.16	Additional Exercise Problems	121

2.1 Unsigned Integers

[1] The figure below shows the classification of the 12 **basic data types** (highlighted with shaded boxes) in MATLAB. We called them "basic" since they are implemented in a computer's hardware level; all other data types (e.g., **array**, **cell array**, **structure**, etc.) in MATLAB are implemented in some software levels on the top of these basic data types.

By default, MATLAB assumes `double` for all numbers. For example,

```
a = 75
```

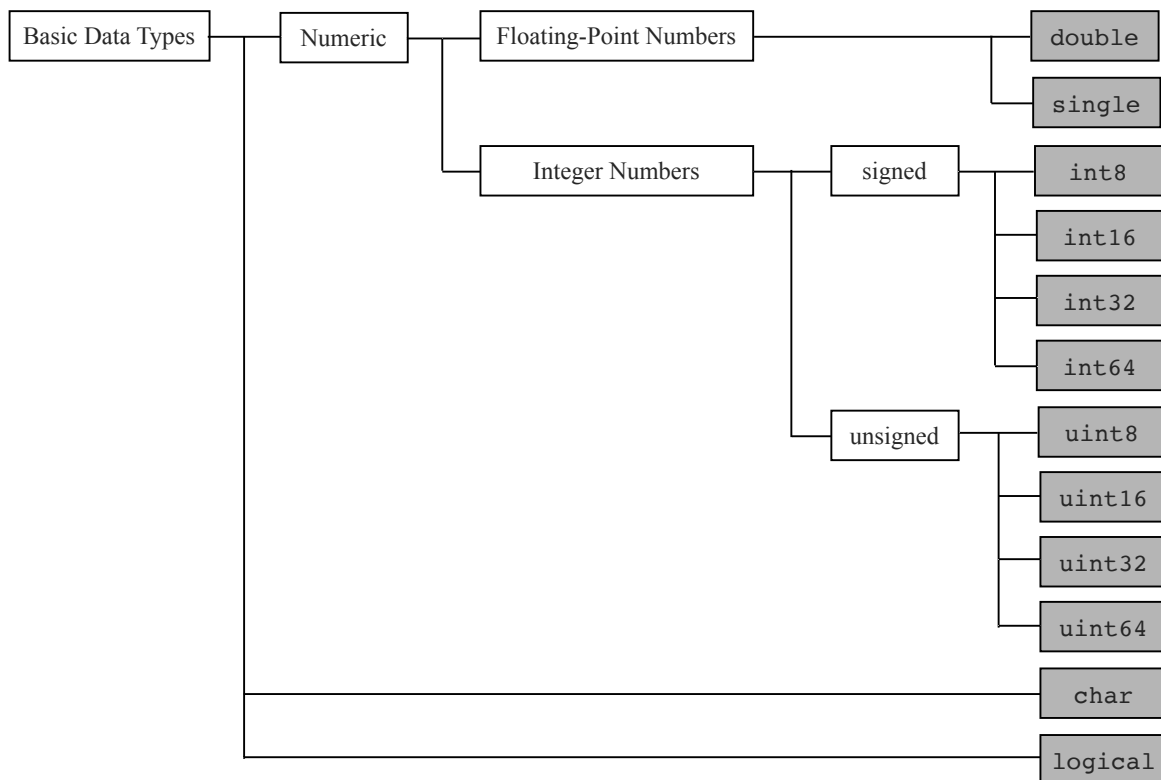
where the number 75 is treated as a `double`; therefore a variable `a` of the same type (`double`) is created to store the number. To create a number of a type other than `double`, you must explicitly use a data type conversion function; e.g.,

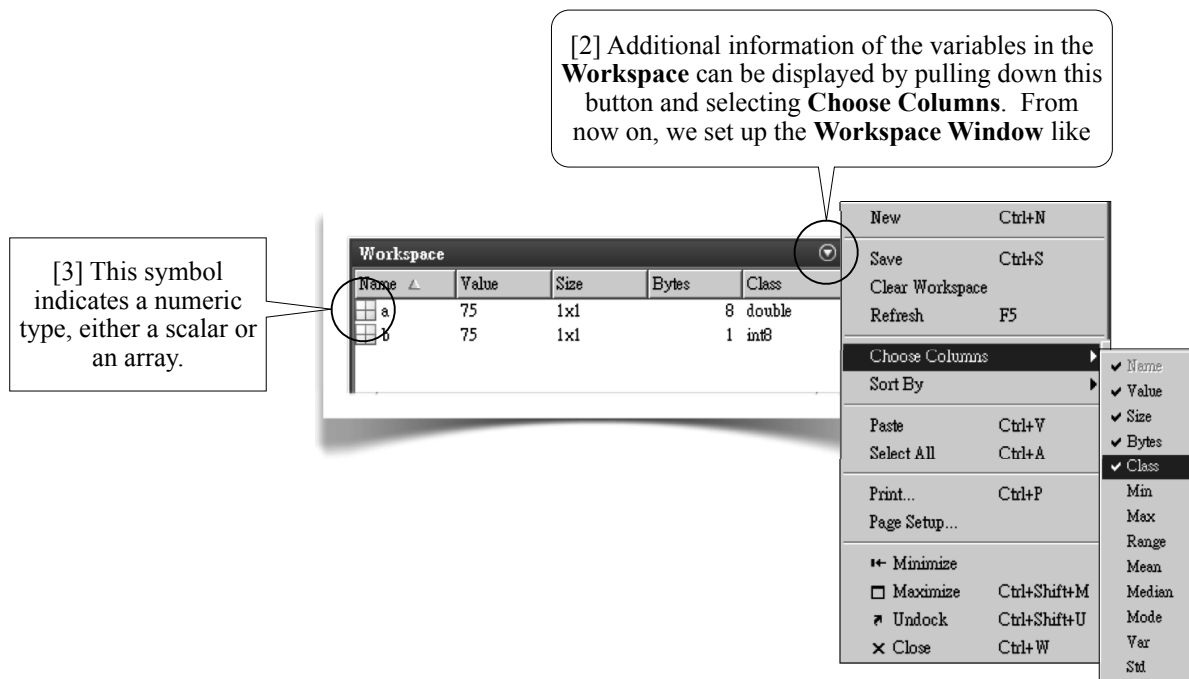
```
b = int8(75)
```

where the right-hand-side is an `int8`; therefore, a variable `b` of the type `int8` is created to stored the number. You may verify the types (or called **classes**) of the variables `a` and `b` by typing the command `whos`,

```
>> whos
      Name      Size      Bytes      Class
      a         1x1         8         double
      b         1x1         1         int8
```

This information can be displayed on the **Workspace Window** (see [2-3], next page). Note that 8 bytes of the memory is needed to store a `double`, while only 1 byte (8 bits) is needed to store an `int8`. But how? Starting from this section, we'll provide some exercises through which you will not only learn how these basic data are stored in the memory (i.e., how they are represented with 0s and 1s) but also learn some useful programming techniques. →





Unsigned Integers

[4] Computer representation of unsigned integers is straightforward. For example, the decimal number 75 is represented by a binary number 01001011, since computers use binary system. The binary number 01001011 can be converted into the decimal number 75 by the following procedure:

$$(0 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0)_{10} = 75_{10}$$

On the other hand, the decimal number 75 can be converted into the binary number by successive divisions of 2:

75	divided by 2 is	37,	remainder 1
37	divided by 2 is	18,	remainder 1
18	divided by 2 is	9,	remainder 0
9	divided by 2 is	4,	remainder 1
4	divided by 2 is	2,	remainder 0
2	divided by 2 is	1,	remainder 0
1	divided by 2 is	0,	remainder 1

After collecting the remainders bottom-up, you have the binary number 1001011 (which is equal to 01001011).

In general, a binary number $b_{n-1}b_{n-2}\dots b_1b_0$ (where each of b_{n-1} , b_{n-2} , ..., b_1 , and b_0 is a binary digit (bit), either 0 or 1) is converted into a decimal number by

$$(b_{n-1} \times 2^{n-1} + b_{n-2} \times 2^{n-2} + \dots + b_1 \times 2^1 + b_0 \times 2^0)_{10} \equiv \left(\sum_{k=0}^{n-1} b_k \times 2^k \right)_{10}$$

Obviously, the minimum value an unsigned integer can represent is zero; i.e., *an unsigned integer cannot represent a negative number*. The maximum value an unsigned integer can represent depends on how many binary digits (bits) it uses. For example, the maximum value for an 8-bit unsigned integer (`uint8`) is

$$(11111111)_2 = (2^8 - 1)_{10} = 255_{10}$$

In general, when n bits are used, the maximum value an unsigned integer can represent is $(2^n - 1)_{10}$.

Table 2.1 (next page) lists information about the four unsigned integer types in MATLAB, including their conversion functions, their minimum/maximum values, and the functions to find these minimum/maximum values. →

Example02_01.m: Unsigned Integers

[5] These statements demonstrate the concepts given in the last page. A **Command Window** session is shown in [6].

```

1 clear
2 d = 75
3 u = uint8(d)
4 bits = bitget(u, 1:8)
5 bits = fliplr(bits)

```

[6] This is a **Command Window** session of Example02_01.m.

```

6 >> clear
7 >> d = 75
8 d =
9     75
10 >> u = uint8(d)
11 u =
12     uint8
13     75
14 >> bits = bitget(u, 1:8)
15 bits =
16     1×8 uint8 row vector
17     1 1 0 1 0 0 1 0
18 >> bits = fliplr(bits)
19 bits =
20     1×8 uint8 row vector
21     0 1 0 0 1 0 1 1
22 >>

```

About Example02_01.m

[7] In line 7, the number 75 is treated as a `double` by default; therefore, a variable `d` of the type `double` is created to store the number. An 8-bit **floating-point** representation, to be discussed in Section 2.3, is used for `double` data.

In line 10, the function `uint8` converts the number `d` to an 8-bit unsigned integer format (without altering the value 75), and a variable `u` of the same type (`uint8`) is created to store the number.

In line 14, the function `bitget` retrieves the bits in `u` (from the first bit to the 8th bit, specified using a row vector `1:8`) and outputs a row vector of type `int8`. A variable `bits` of the same dimension and same type is then created to store the output. Note that, the first bit (right-most bit) is stored as the first element; the variable `bits` now has values `[1 1 0 1 0 0 1 0]` (line 17).

In line 18, the function `fliplr` flips the row vector `bits` left to right, making the bit pattern consistent with the conventional way `01001011` (i.e., the leftmost bit is stored as the first element in `bits`; see line 21). #

Table 2.1 Unsigned Integer Numbers

Conversion Function	Function to find the minimum value	Minimum value	Function to find the maximum value	Maximum value
<code>uint8</code>	<code>intmin('uint8')</code>	0	<code>intmax('uint8')</code>	255
<code>uint16</code>	<code>intmin('uint16')</code>	0	<code>intmax('uint16')</code>	65535
<code>uint32</code>	<code>intmin('uint32')</code>	0	<code>intmax('uint32')</code>	4294967295
<code>uint64</code>	<code>intmin('uint64')</code>	0	<code>intmax('uint64')</code>	18446744073709551615

Details and More: Help>MATLAB>Language Fundamentals>Data Types>Numeric Types

2.2 Signed Integers

Two's Complement Representation

[1] A simple idea to represent a negative integer is to use a bit as the "sign bit." For example, we may use the leftmost bit as the sign bit: 0 for positive and 1 for negative. Thus, if the bit pattern 01001011 represents the decimal number +75, then 11001011 could represent -75. One problem of this representation is that both the bit patterns 00000000 and 10000000 represent the same number, zero.

Another idea is to use the **one's complement representation**. It also uses a sign bit as before: 0 for positive and 1 for negative. If the sign bit indicates that it is a negative number, then its complement pattern (i.e., converting 0s to 1s and 1s to 0s) is interpreted as its absolute value. For example, since the leftmost bit of 10110100 indicates that it is a negative number, its complement pattern 01001011 (75_{10}) is interpreted as its absolute value. Thus, the bit pattern 10110100 is interpreted as -75. Many early computers used this method (see *Wikipedia* > *One's Complement*). The previous problem still exists for this approach: both the bits pattern 00000000 and 11111111 represent the same number, zero.

Modern computers use **two's complement representation**. It also uses a sign bit as before. If the sign bit indicates that it is a negative number, then its two's complement pattern (i.e., adding one to its complement pattern) is used to represent its absolute value. For example, since the leftmost bit of 10110101 indicates that it is a negative number, we take its complement (01001010), add one (01001011, which has a decimal value of 75), and interpret the bit pattern 10110101 as -75. Thus, 00000000 represents zero, and 11111111 represents -1. Table 2.2a gives some examples of unsigned/signed representation.

Table 2.2a Unsigned/Signed Representation

Bit pattern	Unsigned value	Signed value
000	0	0
001	1	1
010	2	2
011	3	3
100	4	-4
101	5	-3
110	6	-2
111	7	-1
00000000	0	0
11111111	255	-1
01111111	127	127
10000000	128	-128

Details and More:
Wikipedia > *Two's complement*

Minimum/Maximum Values

[2] From Table 2.2a, with 8 bits, the minimum signed value is 10000000 (-128_{10}) and the maximum value is 01111111 (127_{10}). In general, when n -bits are used, the minimum value is -2^{n-1} and the maximum value is $2^{n-1} - 1$. Table 2.2b lists information about the four signed integer types in MATLAB, including their conversion functions, their minimum/maximum values, and the functions to find these minimum/maximum values. →

Table 2.2b Signed Integer Numbers

Conversion Function	Function to find the minimum value	Minimum value	Function to find the maximum value	Maximum value
int8	intmin('int8')	-128	intmax('int8')	127
int16	intmin('int16')	-32768	intmax('int16')	32767
int32	intmin('int32')	-2147483648	intmax('int32')	2147483647
int64	intmin('int64')	-9223372036854775808	intmax('int64')	9223372036854775807

Details and More: *Help* > *MATLAB* > *Language Fundamentals* > *Data Types* > *Numeric Types*

Example02_02.m: Signed Integers

[3] These statements demonstrate some concepts about signed integers. A **Command Window** session is shown in [4].

```

1 clear
2 d = 200
3 u = uint8(d)
4 bits = fliplr(bitget(u, 1:8))
5 t = int8(u)
6 a = int16(u)
7 s = typecast(u, 'int8')
8 bits = fliplr(bitget(s, 1:8))

```

[4] This is a **Command Window** session of Example02_02.m.

```

9 >> clear
10 >> d = 200
11 d =
12     200
13 >> u = uint8(d)
14 u =
15     uint8
16     200
17 >> bits = fliplr(bitget(u, 1:8))
18 bits =
19     1×8 uint8 row vector
20     1  1  0  0  1  0  0  0
21 >> t = int8(u)
22 t =
23     int8
24     127
25 >> a = int16(u)
26 a =
27     int16
28     200
29 >> s = typecast(u, 'int8')
30 s =
31     int8
32     -56
33 >> bits = fliplr(bitget(s, 1:8))
34 bits =
35     1×8 int8 row vector
36     1  1  0  0  1  0  0  0
37 >>

```

About Example02_02.m

[5] Lines 1-3 are similar to those in Example02_01.m, page 71. Now, the variable `u`, an 8-bit unsigned integer, has a value of 200, which has a bit pattern 11001000, which can be confirmed in line 4 (also see line 20). Line 4 is equivalent to the combination of the line 4 and line 5 of Example02_01.m, page 71. In line 4, the output of the function `bitget` becomes the input argument of `fliplr`. We often write a statement this way to make it concise.

In line 5, the value 200 is converted to an `int8`; however, since the maximum value of an `int8` is 127 (see Table 2.2b, last page), the value is "overflowed" and only the maximum value (127) is stored in an `int8` (the extra value is truncated). Therefore, the variable `t` has a value of 127 (see line 24). To store the value 200 in a signed integer, we need at least an `int16`. Line 6 successfully converts the value 200 to an `int16` and stores it in the variable `a` (see line 28).

In line 7, the function `typecast` preserves the bit pattern of the unsigned value 200 (11001000) while changing its type to `int8`. Now, the bit pattern is interpreted as a value of -56 (see line 32), using two's complement representation: Since the leftmost bit of 11001000 indicates that it is a negative number, we take its complement (00110111), add one (00111000, which has a decimal value of 56), and interpret the bit pattern 10110101 as -56.

Line 8 confirms that the bit pattern in the variable `s` is indeed preserved, the same as that of `u`, 11001000 (see line 36). #

2.3 Floating-Point Numbers

[1] Your computer uses floating-point representation to store real numbers. MATLAB has two types of floating-point numbers: double precision (`double`) and single precision (`single`); a `double` uses 8 bytes (64 bits) of memory while a `single` uses 4 bytes (32 bits). As mentioned, `double` is the default data type and, therefore, is the most extensively used data type. Table 2.3a lists information about the two floating-point types, including their conversion functions, their minimum/maximum values, and the functions to find these minimum/maximum values.

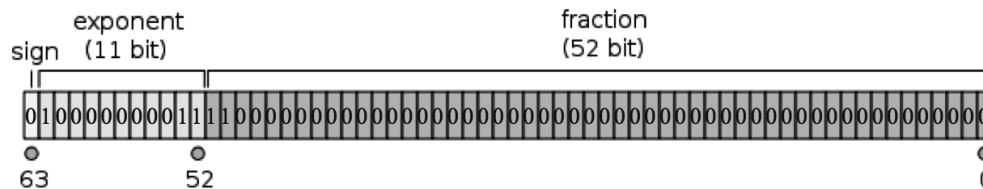
Floating-Point Representation

The figure below (source: https://en.wikipedia.org/wiki/File:IEEE_754_Double_Floating_Point_Format.svg, by Codekaizen) shows an example bit pattern of a double-precision floating-point number. It uses 64 bits in computer memory: 1 bit (the 63rd bit) for the sign, 11 bits (the 52nd-62nd bits) for the exponent, and 52 bits (the 0th-51st bits) for the fraction. The 64-bit pattern is interpreted as a value of

$$(-1)^{\text{sign}} (1.b_{51}b_{50} \dots b_0)_2 \times 2^{\text{exponent}-1023}$$

Thus, the bit pattern below is interpreted as

$$+(1.11)_2 \times 2^{1027-1023} = (2^0 + 2^{-1} + 2^{-2}) \times 2^4 = 1.75 \times 16 = (28)_{10}$$



Fractional Binary Numbers

[2] For those who are not familiar with the binary numbers, here is another example. A decimal number 258.369 is interpreted as

$$(258.369)_{10} = 2 \times 10^2 + 5 \times 10^1 + 8 \times 10^0 + 3 \times 10^{-1} + 6 \times 10^{-2} + 9 \times 10^{-3}$$

Similarly, a binary number 1101.01101 can be interpreted as

$$\begin{aligned} (1101.01101)_2 &= 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} + 0 \times 2^{-4} + 1 \times 2^{-5} \\ &= 8 + 4 + 0 + 1 + 0 + 0.25 + 0.125 + 0 + 0.03125 && \rightarrow \\ &= (13.40625)_{10} \end{aligned}$$

Table 2.3a Floating-Point Numbers

Conversion Function	Function to find the minimum value	Minimum value	Function to find the maximum value	Maximum value
<code>double</code>	<code>realmin('double')</code>	2.2251e-308	<code>realmax('double')</code>	1.7977e+308
<code>single</code>	<code>realmin('single')</code>	1.1755e-38	<code>realmax('single')</code>	3.4028e+38

Details and More: *Help>MATLAB>Language Fundamentals>Data Types>Numeric Types*


```

25 >> clear
26 >> format short
27 >> format compact
28 >> a = 1234.56789012345678901234
29 a =
30 1.2346e+03
31 >> fprintf('%%.20f\n', a)
32 1234.56789012345689116046
33 >> format long
34 >> a
35 a =
36 1.234567890123457e+03
37 >> single(a)
38 ans =
39 single
40 1.2345679e+03
41 >>

```

[6] This is a **Command Window** session of Example02_03b.m.

Table 2.3b Numeric Output Format

Function	Description or Example
<code>format compact</code>	Suppress blank lines
<code>format loose</code>	Add blank lines
<code>format short</code>	3.1416
<code>format long</code>	3.141592653589793
<code>format shortE</code>	3.1416e+00
<code>format longE</code>	3.141592653589793e+00
<code>format shortG</code>	short or shortE
<code>format longG</code>	long or longE
<code>format shortEng</code>	Exponent is a multiple of 3
<code>format longEng</code>	Exponent is a multiple of 3
<code>format +</code>	Display the sign (+/-)
<code>format bank</code>	Currency format; 3.14
<code>format hex</code>	400921fb54442d18
<code>format rat</code>	Rational; 355/133

Details and More: >> doc format

Screen Output Format

[7] Lines 18, 19, and 22 set **Command Window** output display format. The syntax is

```
format style
```

The `short` (line 18) sets the display of fixed-decimal format 4 digits after the decimal point, the `long` (line 22) 15 digits after the decimal point. The `compact` (line 19) suppresses blank lines to make the output lines compact. The opposite of `compact` is `loose` (default), which adds blank lines to make the output lines more readable. In this book, we always use `compact` style to save space.

Table 2.3b lists the available format styles. Remember, you may always consult the on-line documentation whenever a new command is encountered. For example:

```
>> doc format
```

Double Precision Floating-Point Numbers

In line 20, we assign a number of 24 significant figures to the variable `a` of `double`. We'll see (line 32) that, due to the limiting storage space (64 bits), not all the figures can be stored in the variable `a`. The number is displayed (line 30) in `short` format, i.e., 4 digits after the decimal point. Note that, in displaying the number, it is rounded to the last digit.

In line 21, we attempt to print the number with 20 digits after the decimal point. The result (line 32) shows that only the first 16 figures are the same as what was assigned to the variable `a`. The extra digits are lost due to the limiting storage space. We conclude that *a double-precision floating point number has 16 significant figures*.

With `long` format (line 22), the number is displayed with 15 digits after the decimal point (line 36). Note that, again, in displaying the number, it has been rounded to the last digit. →

Single-Precision Floating-Point Numbers

[8] Line 24 converts the value stored in the variable `a` (which is of type `double`, 64-bit long) to a single-precision floating-point number (32-bit long). The output (line 40) shows that it reduces to 8 significant figures, due to the shorter storage space. The extra digits are discarded during the conversion. We conclude that *a single-precision floating point number has 8 significant figures*.

The Variable `ans`

In line 24, there is no variable to store the result of conversion. Whenever a value is produced and there is no variable to store that value, MATLAB always uses the variable `ans` (short for **answer**) to store that value (see line 38). #

2.4 Characters and Strings

ASCII Codes

[1] In MATLAB, a **character** is represented using single quotes; e.g., 'A', 'b', etc. Internally, MATLAB uses 2 bytes (16 bits) to store a character according to **ASCII Code** (see *Wikipedia>ASCII*, also see [7-8], pages 79-80). An ASCII code is a number representing a character, either printable or non-printable. The ASCII codes of the character 'A', 'B', and 'C' are 65, 66, and 67, respectively. A character can be converted to a numeric value according to **ASCII Codes**. For example, since 'A' is internally represented by an ASCII code 65, `double('A')` results in a number 65.

The most frequently used non-printable character is the **newline** character (1.9[9], page 35). MATLAB uses '\n' to represent the newline character.

The notation such as 'ABC' is used to represent a row vector of characters; i.e., it is equivalent to ['A', 'B', 'C']. A row vector of characters is also called a **string**.

Example02_04a.m: Characters

[2] These statements demonstrate some concepts about **characters** and **strings**. A **Command Window** session is shown in [3] and the **Workspace** is shown in [4-5].

```

1 clear
2 a = 'A'
3 b = a + 1
4 char(65)
5 char('A' + 2)
6 c = ['A', 'B', 'C']
7 d = ['AB', 'C']
8 e = ['A', 66, 67]
9 f = 'ABC'
10 f(1)
11 f(2)
12 f(3)

```

[4] This symbol indicates a character type.

[3] This is a **Command Window** session of Example02_04a.m.

Name	Value	Size	Bytes	Class
a	'A'	1x1	2	char
ans	'C'	1x1	2	char
b	66	1x1	8	double
c	'ABC'	1x3	6	char
d	'ABC'	1x3	6	char
e	'ABC'	1x3	6	char
f	'ABC'	1x3	6	char

[5] Remember that this symbol indicates a numeric type (2.1[3], page 70). →

```

13 >> clear
14 >> a = 'A'
15 a =
16     'A'
17 >> b = a + 1
18 b =
19     66
20 >> char(65)
21 ans =
22     'A'
23 >> char('A' + 2)
24 ans =
25     'C'
26 >> c = ['A', 'B', 'C']
27 c =
28     'ABC'
29 >> d = ['AB', 'C']
30 d =
31     'ABC'
32 >> e = ['A', 66, 67]
33 e =
34     'ABC'
35 >> f = 'ABC'
36 f =
37     'ABC'
38 >> f(1)
39 ans =
40     'A'
41 >> f(2)
42 ans =
43     'B'
44 >> f(3)
45 ans =
46     'C'

```

About Example02_4a.m

[6] In line 2, a character `A` is assigned to a variable `a`, which is of type `char` (line 16).

In line 3, since `+` (plus) is a numeric operator, MATLAB converts the variable `a` to a numeric value, 65, and then adds 1. The result is 66 and the variable `b` is of numeric type, a `double` (line 19).

In line 4, the numeric value 65 is converted to a `char`, using the function `char`. The result is the character `'A'` (line 22).

In line 5, again, since `+` (plus) is a numeric operator, MATLAB converts the character `'A'` to a numeric value, 65, and then adds 2. The result is 67, which, after converting to `char` type, is the character `'C'` (line 25).

Numeric Operations Involving Characters

A numeric operator (`+`, `-`, etc.) always operates on numeric values, and the result is a numeric value. If a numeric operation involves characters, the characters are converted to numeric values according to ASCII codes.

We'll introduce numerical operations in Section 2.7 and Section 2.8 and string manipulations in Section 2.10.

String: Row Vector of Characters

Line 6 creates a row vector of three characters `'A'`, `'B'`, and `'C'`. It is displayed as `'ABC'` (line 28). A row vector of character is also called a **string**. The variable `c` is a **string**.

Line 7 seemingly creates a row vector of two elements. However, since `'AB'` itself is a row vector of two characters, the result is a row vector of three characters `'ABC'` (line 31). There is no difference between variables `c` and `d`; they are all strings of three characters `'ABC'`.

In line 8, since an array must have elements of the same data type, MATLAB converts the number 66 and 67 to characters according to ASCII Codes. The result is a row vector of three characters `'ABC'`. There is no difference between variables `c`, `d`, and `e`.

Line 9 demonstrates an easy way to create a vector of characters, a string. There is no difference between the variables `c`, `d`, `e`, and `f`. They are all vectors of three characters `ABC`, which can be confirmed in lines 35-46.

Example02_04b.m: ASCII Codes

[7] MATLAB stores characters according to ASCII Code. ASCII codes 32-126 represent all printable characters on a standard keyboard. This example prints a table of characters corresponding to the ASCII Codes 32-126 (see the output in [8], next page). →

```

47 clear
48 fprintf('    0 1 2 3 4 5 6 7 8 9\n')
49 for row = 3:12
50     fprintf('%2d ', row)
51     for column = 0:9
52         code = row*10+column;
53         if (code < 32) || (code > 126)
54             fprintf(' ')
55         else
56             fprintf('%c ', code)
57         end
58     end
59     fprintf('\n')
60 end

```

[8] This is the output of Example02_04b.m. Note that ASCII Code 32 corresponds to the space character.

```

      0 1 2 3 4 5 6 7 8 9
3      ! " # $ % & '
4      ( ) * + , - . / 0 1
5      2 3 4 5 6 7 8 9 : ;
6      < = > ? @ A B C D E
7      F G H I J K L M N O
8      P Q R S T U V W X Y
9      Z [ \ ] ^ _ ` a b c
10     d e f g h i j k l m
11     n o p q r s t u v w
12     x y z { | } ~

```

About Example02_4b.m

[9] Line 48 prints a heading of column numbers and a newline character, moving the cursor to the next line.

Each pass of the outer `for`-loop (lines 49-60) prints a row on the screen; the row numbers are designated as 3, 4, ... 12 for each pass. In the beginning of the loop (line 50), the row number is printed. Then 10 characters are printed using an inner `for`-loop (lines 51-58). At the end of the outer `for`-loop (line 59), a newline character is printed, moving the cursor to the next line.

Each pass of the inner `for`-loop (lines 51-58) prints a character aligning with the column number. Line 52 generates an ASCII code using the row-number and column-number; for example, the ASCII code corresponds to row-number 4 and column-number 5 is 45. If an ASCII code is less than 32 or larger than 126 (line 53) then two spaces are printed (line 54), otherwise the ASCII code is printed as a character followed by a space (line 56).

The expression `(code < 32) || (code > 126)` in the `if`-statement (line 53) is a logical expression. We'll introduce logical data in Section 2.5 and logical expressions in Section 2.9.

In line 56, the format specifier `%c` requires a character data, therefore `code` (a `double`) is converted to a character according to the ASCII Code. `#`

2.5 Logical Data

Logical Values: true and false

[1] The only logical values are `true` and `false`. MATLAB uses 1 byte (8 bits) to store a logical value. When a logical value is converted to a number, `true` becomes 1 and `false` becomes 0. When a numeric value is converted to a logical value, any non-zero number becomes `true` and the number 0 becomes `false`.

Example02_05.m: Logical Data Type

[2] These statements demonstrate some concepts about **logical** data. A **Command Window** session and the **Workspace** is shown in [3, 4], respectively.

```

1  clear
2  a = true
3  b = false
4  c = 6 > 5
5  d = 6 < 5
6  e = (6 > 5)*10
7  f = false*10+true*2
8  g = (6 > 5) & (6 < 5)
9  h = (6 > 5) | (6 < 5)
10 k = logical(5)
11 m = 5 | 0
12 n = (-2) & 'A'
```

[4] This symbol indicates a logical type. →

Name	Value	Size	Bytes	Class
<input checked="" type="checkbox"/> a	1	1x1	1	logical
<input checked="" type="checkbox"/> b	0	1x1	1	logical
<input checked="" type="checkbox"/> c	1	1x1	1	logical
<input checked="" type="checkbox"/> d	0	1x1	1	logical
<input checked="" type="checkbox"/> e	10	1x1	8	double
<input checked="" type="checkbox"/> f	1	1x1	8	double
<input checked="" type="checkbox"/> g	0	1x1	1	logical
<input checked="" type="checkbox"/> h	1	1x1	1	logical
<input checked="" type="checkbox"/> k	1	1x1	1	logical
<input checked="" type="checkbox"/> m	1	1x1	1	logical
<input checked="" type="checkbox"/> n	1	1x1	1	logical

```

13 >> clear
14 >> a = true
15 a =
16  logical
17  1
18 >> b = false
19 b =
20  logical
21  0
22 >> c = 6 > 5
23 c =
24  logical
25  1
26 >> d = 6 < 5
27 d =
28  logical
29  0
30 >> e = (6 > 5)*10
31 e =
32  10
33 >> f = false*10+true*2
34 f =
35  2
36 >> g = (6 > 5) & (6 < 5)
37 g =
38  logical
39  0
40 >> h = (6 > 5) | (6 < 5)
41 h =
42  logical
43  1
44 >> k = logical(5)
45 k =
46  logical
47  1
48 >> m = 5 | 0
49 m =
50  logical
51  1
52 >> n = (-2) & 'A'
53 n =
54  logical
55  1
```

[3] This is a **Command Window** session of Example02_05.m.

About Example02_05.m

[5] Line 2 assigns `true` to a variable `a`, which is then of type `logical` (line 16). When displayed on the **Command Window**, `true` is always displayed as 1 (line 17).

Line 3 assigns `false` to variable `b`. When displayed on the **Command Window**, `false` is always displayed as 0 (line 21).

Relational Operators

A relational operator (`>`, `<`, etc., to be introduced in Section 2.9) always operates on two numeric values, and the result is a logical value.

In line 4, the number 6 and the number 5 are operated using the logical operator `>`. The result of `6 > 5` is `true` and is assigned to `c`, which is of type `logical` and displayed as 1 (lines 24-25).

In line 5, the number 6 and the number 5 are operated using the logical operator `<`. The result of `6 < 5` is `false` and is assigned to `d`, which is of type `logical` and displayed as 0 (lines 28-29).

Numeric Operations Involving Logical Values

A numeric operator (`+`, `-`, etc., to be introduced in Section 2.8) always operates on numeric values, and the result is a numeric value. If a numeric operator involves logical values, the logical values are converted to numeric values: `true` becomes 1 and `false` becomes 0.

In line 6, the result of `6 > 5` is a logical value `true`, which is to be multiplied by the number 10. Since the multiplication (`*`) is a numeric operator, MATLAB converts `true` to the number 1, and the result is 10 (line 32), which is a `double` number. When a `double` number is output to the **Command Window**, its type is not shown; remember that `double` is the default data type.

In line 7, again, since the multiplication (`*`) and the addition (`+`) are numeric operators, MATLAB converts `false` to 0 and `true` to 1, and the result is 2 (line 35), which is a `double` number.

Logical Operators

A logical operator (`AND`, `OR`, etc.) always operates on logical values, and the result is a logical value. If a logical operation involves numeric values, the numeric values are converted to logical values (non-zero values become `true` and zero value becomes `false`).

MATLAB uses the symbol `&` for logical `AND` and the symbol `|` for logical `OR`. Table 2.5a (next page) lists the rules for logical `AND` (`&`). Table 2.5b (next page) lists the rules for logical `OR` (`|`).

In line 8, the result of `6 > 5` is `true` and the result of `6 < 5` is `false`. The result of logical `AND` (`&`) operation for a `true` and a `false` is `false` (lines 38-39).

In line 9, the result of local `OR` (`|`) operation for a `true` and a `false` is `true` (lines 42-43).

We'll introduce relational and logical operators in Section 2.9.

Conversion to Logical Data Type

Line 10 converts a numeric value 5 to logical data type. The result is `true` (lines 46-47). When converted to a logical value, any non-zero number becomes `true` and the number 0 becomes `false`. A `true` is displayed as 1 and a `false` is displayed as zero.

In line 11, since `|` (`OR`) is a logical operator, MATLAB converts the numbers 5 and 0 to logical values `true` and `false`, respectively. The result is `true` (lines 50-51).

In line 12, again, since `&` (`AND`) is a logical operator, MATLAB converts both the number `-2` and the character `'A'` to logical values `true`. The result is `true` (lines 54-55). →

Avoid Using `i`, `j`, and `l` as Variable Names

[6] In MATLAB, both letters `i` and `j` are used to represent the constant $\sqrt{-1}$. If you use them as variable names, they are overridden by the values assigned to them and no longer represent $\sqrt{-1}$. In this book, we'll avoid using them as variable names.

The letter `l` is often confused with the number `1`. In this book, we'll also avoid using it as a variable name. #

Table 2.5a Rules of Logical and (&)

AND (&)	true	false
true	true	false
false	false	false

Table 2.5b Rules of Logical or (|)

OR ()	true	false
true	true	true
false	true	false

2.6 Arrays

All Data Are Arrays

[1] MATLAB treats all data as arrays. A zero-dimensional array (1×1) is called a **scalar**. A one-dimensional array is called a **vector**, either a row vector ($1 \times c$) or a column vector ($r \times 1$). A two-dimensional array ($r \times c$) is called a **matrix**. A three-dimensional array ($r \times c \times p$) may be called a **three-dimensional array** or **three-dimensional matrix**. It is possible to create a four or more dimensional array; in practice, however, it is seldom used. The first dimension is called the **row dimension**, the second dimension is called the **column dimension**, and the third dimension is called the **page dimension**.

Example02_06a.m

[2] Type the following commands (also see [3]).

```
1 clear
2 a = 5
3 b = [5]
4 c = 5*ones(1,1)
5 D = ones(2, 3)
6 e = [1, 2, 3, 4, 5]
7 f = [1 2 3 4 5]
8 g = [1:5]
9 h = 1:5
10 k = 1:1:5
11 m = linspace(1, 5, 5)
```

Scalar

[4] Lines 13-21 show many ways to create the same scalar. Line 13 creates a single value 5. Line 16 creates a vector of one element, i.e., a scalar. Line 19 creates a 1×1 matrix (see an explanation below for the function `ones`). The variables `a`, `b`, and `c` are all **scalars**; they are all equal; there is no difference among these three variables.

Function ones

The function `ones` (line 19) creates an array of all ones with specified dimension sizes. Another example is shown in line 22. The syntax is

```
ones(sz1, sz2, ..., szN)
```

where `sz1` is the size of the first dimension (row dimension), `sz2` is the size of the second dimension (column dimension), and so forth. The function `ones` is one of the array creation functions. Table 2.6a (page 88) lists some array creation functions. →

```
12 >> clear
13 >> a = 5
14 a =
15     5
16 >> b = [5]
17 b =
18     5
19 >> c = 5*ones(1,1)
20 c =
21     5
22 >> D = ones(2, 3)
23 D =
24     1     1     1
25     1     1     1
26 >> e = [1, 2, 3, 4, 5]
27 e =
28     1     2     3     4     5
29 >> f = [1 2 3 4 5]
30 f =
31     1     2     3     4     5
32 >> g = [1:5]
33 g =
34     1     2     3     4     5
35 >> h = 1:5
36 h =
37     1     2     3     4     5
38 >> k = 1:1:5
39 k =
40     1     2     3     4     5
41 >> m = linspace(1, 5, 5)
42 m =
43     1     2     3     4     5
```

[3] This is a **Command Window** session of Example02_06a.m.

Row Vectors

[5] Lines 26-43 show many ways to create the same row vector. Line 26 creates a row vector using the square brackets ([]). Commas are used to separate elements in a row. The commas can be omitted (line 29).

Line 32 creates a row vector using the colon (:). The square brackets can be omitted (line 35). In a more general form, an increment number can be inserted between the starting number and the ending number (line 38; also see 1.3[6], page 18). The function `linspace` (line 41) creates a row vector of linearly spaced numbers. The syntax is

$$\text{linspace}(\textit{start}, \textit{end}, \textit{n})$$

where n is the number of total elements. If n is omitted, it defaults to 100.

There is no difference among the variables e , f , g , h , k , and m .

Example02_06b.m

[6] Type the following commands (also see [7]).

```
44 clear
45 a = zeros(1,5)
46 a(1,5) = 8
47 a(5) = 9
48 a([1, 2, 4]) = [8, 7, 6]
49 a(1:4) = [2, 3, 4, 5]
50 [rows, cols] = size(a)
51 len = length(a)
52 b = a
53 c = a(1:5)
54 d = a(3:5)
55 e = a(3:length(a))
56 f = a(3:end)
57 f(5) = 10
```

```
58 >> clear
59 >> a = zeros(1,5)
60 a =
61     0     0     0     0     0
62 >> a(1,5) = 8
63 a =
64     0     0     0     0     8
65 >> a(5) = 9
66 a =
67     0     0     0     0     9
68 >> a([1, 2, 4]) = [8, 7, 6]
69 a =
70     8     7     0     6     9
71 >> a(1:4) = [2, 3, 4, 5]
72 a =
73     2     3     4     5     9
74 >> [rows, cols] = size(a)
75 rows =
76     1
77 cols =
78     5
79 >> len = length(a)
80 len =
81     5
82 >> b = a
83 b =
84     2     3     4     5     9
85 >> c = a(1:5)
86 c =
87     2     3     4     5     9
88 >> d = a(3:5)
89 d =
90     4     5     9
91 >> e = a(3:length(a))
92 e =
93     4     5     9
94 >> f = a(3:end)
95 f =
96     4     5     9
97 >> f(5) = 10
98 f =
99     4     5     9     0    10
```

[7] This is a **Command Window** session of Example02_06b.m.

Indexing

[8] Line 59 creates a 1-by-5 array (i.e., a row vector) of all zeros. Remember that there are two ways to access an element of an array: **subscript indexing** and **linear indexing** (1.9[23], page 37). Line 62 uses subscript indexing, while line 65 uses linear indexing. For a vector (row vector or column vector), we usually use linear indexing.

Line 68 assigns three values to the 1st, 2nd, and 4th elements of the array a ; i.e., line 68 is equivalent to

$$a(1) = 8, a(2) = 7, a(4) = 6$$

Line 71, since $1:4$ means $[1, 2, 3, 4]$, assigns four values to the 1st-4th elements of the array a . →

Size and Length of an Array

[9] The function `size` (line 74) outputs dimension sizes of an array. In line 74, `size(a)` outputs two values: number of rows and number of columns; and a two-element vector is needed to store the output values.

The length of an array is the maximum dimension size of the array; i.e.,

$$\text{length}(a) \equiv \max(\text{size}(a))$$

In this case, the length of the array `a` is 5 (lines 79-81), which is the number of columns.

[10] Line 82 assigns the entire array `a` to a variable `b`, which becomes the same sizes and contents as the array `a`. Line 85 uses another way to assign all the values of the array `a` to a variable. The variables `a`, `b`, and `c` are the same in sizes and contents.

Lines 88, 91, and 94 demonstrate three ways to assign the 3rd, 4th, and 5th elements of the array `a` to a variable. The variables `d`, `e`, and `f` are all the same in sizes and contents. Note that, in line 94, the keyword `end` means the **last index**.

Line 97 attempts to assign a value to the 5th element of the array `f`, which is a row vector of length 3. MATLAB expands the array `f` to a row vector of length 5 to accommodate the value and pads zeros for the unused elements. `f` now is a row vector of length 5.

Example02_06c.m

[11] Type the following commands (also see [12]).

```
100 clear
101 a = [1, 2; 3, 4; 5, 6]
102 b = 1:6
103 c = reshape(b, 3, 2)
104 d = reshape(b, 2, 3)
105 e = d'
106 c(:,3) = [7, 8, 9]
107 c(4,:) = [10, 11, 12]
108 c(4,:) = []
109 c(:,2:3) = []
```

```
110 >> clear
111 >> a = [1, 2; 3, 4; 5, 6]
112 a =
113     1     2
114     3     4
115     5     6
116 >> b = 1:6
117 b =
118     1     2     3     4     5     6
119 >> c = reshape(b, 3, 2)
120 c =
121     1     4
122     2     5
123     3     6
124 >> d = reshape(b, 2, 3)
125 d =
126     1     3     5
127     2     4     6
128 >> e = d'
129 e =
130     1     2
131     3     4
132     5     6
133 >> c(:,3) = [7, 8, 9]
134 c =
135     1     4     7
136     2     5     8
137     3     6     9
138 >> c(4,:) = [10, 11, 12]
139 c =
140     1     4     7
141     2     5     8
142     3     6     9
143    10    11    12
144 >> c(4,:) = []
145 c =
146     1     4     7
147     2     5     8
148     3     6     9
149 >> c(:,2:3) = []
150 c =
151     1
152     2
153     3
```

[12] This is a **Command Window** session of Example02_06c.m. →

Function reshape

[13] Line 111 creates a 3-by-2 matrix. Line 116 creates a row vector of 6 elements. Line 119 reshapes the vector `b` into a 3-by-2 matrix. The reshaping doesn't alter the order of the elements (see 1.9[11-16], page 36); it alters dimensionality and dimension sizes. Note that `c` is different from `a`. To obtain a matrix the same as `a` from the vector `b`, we reshape `b` into a 2-by-3 matrix first (line 124) and then transpose it (line 128). Now `e` is the same as `a`.

Colon: The Entire Column/Row

Line 133 assigns 3 elements to the third column of `c`. Note that `[7, 8, 9]` is automatically transposed, becoming a column. Line 138 assigns 3 elements to the fourth row of `c`.

The colon (`:`) represents the entire column when placed at the row (first) index and represents the entire row when placed at the column (second) index.

Empty Data

Line 144 sets the fourth row of `c` to be empty, i.e., deleting the entire row. Line 149 sets the 2nd-3rd columns to empty, i.e., deleting the 2nd-3rd columns.

The `[]` represents an empty data.

Example02_06d.m

[14] Type the following commands (also see [15]).

```
154 clear
155 a = reshape(1:6, 3, 2)
156 b = [7; 8; 9]
157 c = horzcat(a, b)
158 d = [a, b]
159 e = b'
160 f = vertcat(d, e)
161 g = [d; e]
162 h = fliplr(c)
163 k = flipud(c)
```

```
164 >> clear
165 >> a = reshape(1:6, 3, 2)
166 a =
167     1     4
168     2     5
169     3     6
170 >> b = [7; 8; 9]
171 b =
172     7
173     8
174     9
175 >> c = horzcat(a, b)
176 c =
177     1     4     7
178     2     5     8
179     3     6     9
180 >> d = [a, b]
181 d =
182     1     4     7
183     2     5     8
184     3     6     9
185 >> e = b'
186 e =
187     7     8     9
188 >> f = vertcat(d, e)
189 f =
190     1     4     7
191     2     5     8
192     3     6     9
193     7     8     9
194 >> g = [d; e]
195 g =
196     1     4     7
197     2     5     8
198     3     6     9
199     7     8     9
200 >> h = fliplr(c)
201 h =
202     7     4     1
203     8     5     2
204     9     6     3
205 >> k = flipud(c)
206 k =
207     3     6     9
208     2     5     8
209     1     4     7
```

[15] This is a **Command Window** session of Example02_06d.m. →

Concatenation of Arrays

[16] Line 165 creates a 3-by-2 matrix **a** by reshaping the vector `[1 : 6]`: the first 3 elements become the first column, and the second 3 elements become the second column. Line 170 creates a column vector **b** of 3 elements.

Using function `horzcat`, line 175 concatenates **a** and **b** horizontally to create a 3-by-3 matrix **c**. Line 180 demonstrates a more convenient way to do the same job, using the comma `(,)` to concatenate arrays horizontally.

Line 185 transposes (see 1.6[12], page 27) the column vector **b** to create a row vector **e** of 3 elements.

Using the function `vertcat`, line 188 concatenates **d** and **e** vertically to create a 4-by-3 matrix **f**. Line 194 demonstrates a more convenient way to do the same job, using the semicolon `;` to concatenate arrays vertically.

Flipping Matrices

Using the function `fliplr` (flip left-side right), line 200 flips the matrix **c** horizontally. Using the function `flipud` (flip upside down), line 205 flips the matrix **c** vertically.

Functions for array replication, concatenation, flipping, and reshaping are summarized in Table 2.6b.

More Array Operations

We'll introduce arithmetic operations for numeric data, including arrays and scalars, in the next two sections. #

Table 2.6a Array Creation Functions

Function	Description
<code>zeros(n,m)</code>	Create an n-by-m matrix of all zeros
<code>ones(n,m)</code>	Create an n-by-m matrix of all ones
<code>eye(n)</code>	Create an n-by-n identity matrix
<code>diag(v)</code>	Create a square diagonal matrix with v on the diagonal
<code>rand(n,m)</code>	Create an n-by-m matrix of uniformly distributed random numbers in the interval (0,1)
<code>randn(n,m)</code>	Create an n-by-m matrix of random numbers from the standard normal distribution
<code>linspace(a,b,n)</code>	Create a row vector of n linearly spaced numbers from a to b
<code>[X,Y] = meshgrid(x,y)</code>	Create a 2-D grid coordinates based on the coordinates in vectors x and y.

Details and More: Help>MATLAB>Language Fundamentals>Matrices and Arrays

Table 2.6b Array Replication, Concatenation, Flipping, and Reshaping

Function	Description
<code>repmat(a,n,m)</code>	Replicate array a n times in row-dimension and m times in column-dimension
<code>horzcat(a,b,...)</code>	Concatenate arrays horizontally
<code>vertcat(a,b,...)</code>	Concatenate arrays vertically
<code>flipud(A)</code>	Flip an array upside down
<code>fliplr(A)</code>	Flip an array left-side right
<code>reshape(A,n,m)</code>	Reshape an array to an n-by-m matrix

Details and More: Help>MATLAB>Language Fundamentals>Matrices and Arrays

2.7 Sums, Products, Minima, and Maxima

[1] This section introduces some frequently used functions that calculate the sum, product, minima, and maxima of an array. These functions are summarized in Table 2.7.

Example02_07.m

[2] Type the following commands (also see [3]).

```

1  clear
2  a = 1:5
3  b = sum(a)
4  c = cumsum(a)
5  d = prod(a)
6  e = cumprod(a)
7  f = diff(a)
8  A = reshape(1:9, 3, 3)
9  g = sum(A)
10 B = cumsum(A)
11 h = prod(A)
12 C = cumprod(A)
13 D = diff(A)
14 p = min(a)
15 q = max(a)
16 r = min(A)
17 s = max(A)

```

Table 2.7
Sums, Products, Minima, and Maxima

Function	Description
<code>sum(A)</code>	Sum of array elements
<code>cumsum(A)</code>	Cumulative sum
<code>diff(A)</code>	Differences between adjacent elements
<code>prod(A)</code>	Product of array elements
<code>cumprod(A)</code>	Cumulative product
<code>min(A)</code>	Minimum
<code>max(A)</code>	Maximum

```

18  >> clear
19  >> a = 1:5
20  a =
21      1      2      3      4      5
22  >> b = sum(a)
23  b =
24      15
25  >> c = cumsum(a)
26  c =
27      1      3      6      10     15
28  >> d = prod(a)
29  d =
30      120
31  >> e = cumprod(a)
32  e =
33      1      2      6      24     120
34  >> f = diff(a)
35  f =
36      1      1      1      1
37  >> A = reshape(1:9, 3, 3)
38  A =
39      1      4      7
40      2      5      8
41      3      6      9
42  >> g = sum(A)
43  g =
44      6     15     24
45  >> B = cumsum(A)
46  B =
47      1      4      7
48      3      9     15
49      6     15     24
50  >> h = prod(A)
51  h =
52      6     120    504
53  >> C = cumprod(A)
54  C =
55      1      4      7
56      2     20     56
57      6     120    504
58  >> D = diff(A)
59  D =
60      1      1      1
61      1      1      1

```

[3] This is a **Command Window** session of Example02_07.m (continued at [4], next page). →

```

62 >> p = min(a)
63 p =
64     1
65 >> q = max(a)
66 q =
67     5
68 >> r = min(A)
69 r =
70     1     4     7
71 >> s = max(A)
72 s =
73     3     6     9

```

[4] This is a **Command Window** session of Example02_07.m (Continued).

Sums and Products of Vectors

[5] Let $a_i, i = 1, 2, \dots, n$, be the elements of a vector \mathbf{a} (either row vector or column vector). When applied to a vector, the output of the function `sum` (line 22) is a scalar b , where

$$b = a_1 + a_2 + \dots + a_n$$

In this example (line 24),

$$b = 1 + 2 + 3 + 4 + 5 = 15$$

When applied to a vector, the output of the function `cumsum` (cumulative sum; line 25) is a vector \mathbf{c} of n elements, where

$$c_1 = a_1 \text{ and } c_i = c_{i-1} + a_i; i = 2, 3, \dots, n$$

In this example (line 27),

$$c_1 = a_1 = 1$$

$$c_2 = c_1 + 2 = 3$$

$$c_3 = c_2 + 3 = 6$$

$$c_4 = c_3 + 4 = 10$$

$$c_5 = c_4 + 5 = 15$$

When applied to a vector, the output of the function `prod` (line 28) is a scalar d ,

$$d = a_1 \times a_2 \times \dots \times a_n$$

In this example (line 30),

$$d = 1 \times 2 \times 3 \times 4 \times 5 = 120$$

When applied to a vector, the output of the function `cumprod` (cumulative product; line 31) is a vector \mathbf{e} of n elements, where

$$e_1 = a_1 \text{ and } e_i = e_{i-1} \times a_i; i = 2, 3, \dots, n$$

In this example (line 33),

$$e_1 = a_1 = 1$$

$$e_2 = e_1 \times 2 = 2$$

$$e_3 = e_2 \times 3 = 6$$

$$e_4 = e_3 \times 4 = 24$$

$$e_5 = e_4 \times 5 = 120$$

When applied to a vector, the output of the function `diff` (line 34) is a vector \mathbf{f} of $n-1$ (not n) elements, where

$$f_i = a_{i+1} - a_i; i = 1, 2, \dots, n-1$$

In this example (line 36),

$$f_1 = 2 - 1 = 1$$

$$f_2 = 3 - 2 = 1$$

$$f_3 = 4 - 3 = 1$$

$$f_4 = 5 - 4 = 1$$

→

Sums and Products of Matrices

[6] Let A_{ij} , $i = 1, 2, \dots, n$, $j = 1, 2, \dots, m$ be the elements of an $n \times m$ matrix \mathbf{A} . When applied to a matrix, the output of the function `sum` (line 42) is a row vector \mathbf{g} , where g_j is the sum of the j^{th} column of the matrix \mathbf{A} ; i.e.,

$$g_j = A_{1j} + A_{2j} + \dots + A_{nj}; j = 1, 2, \dots, m$$

In this example (line 44),

$$g_1 = 1 + 2 + 3 = 6$$

$$g_2 = 4 + 5 + 6 = 15$$

$$g_3 = 7 + 8 + 9 = 24$$

Note that the summing is along the **first dimension** (i.e., the row dimension); this rule also applies to the functions `cumsum`, `prod`, and `cumprod` and also applies to three-dimensional arrays.

When applied to a matrix, the output of the function `cumsum` (line 45) is an $n \times m$ matrix \mathbf{B} , where

$$B_{1j} = A_{1j} \text{ and } B_{ij} = B_{(i-1)j} + A_{ij}; i = 2, 3, \dots, n; j = 1, 2, \dots, m$$

In this example (lines 47-49),

$$B_{11} = A_{11} = 1 \quad B_{12} = A_{12} = 4 \quad B_{13} = A_{13} = 7$$

$$B_{21} = B_{11} + 2 = 3 \quad B_{22} = B_{12} + 5 = 9 \quad B_{23} = B_{13} + 8 = 15$$

$$B_{31} = B_{21} + 3 = 6 \quad B_{32} = B_{22} + 6 = 15 \quad B_{33} = B_{23} + 9 = 24$$

When applied to a matrix, the output of the function `prod` (line 50) is a row vector \mathbf{h} ,

$$h_j = A_{1j} \times A_{2j} \times \dots \times A_{nj}; j = 1, 2, \dots, m$$

In this example (line 52),

$$h_1 = 1 \times 2 \times 3 = 6$$

$$h_2 = 4 \times 5 \times 6 = 120$$

$$h_3 = 7 \times 8 \times 9 = 504$$

When applied to a matrix, the output of the function `cumprod` (line 53) is an $n \times m$ matrix \mathbf{C} , where

$$C_{1j} = A_{1j} \text{ and } C_{ij} = C_{(i-1)j} \times A_{ij}; i = 2, 3, \dots, n; j = 1, 2, \dots, m$$

In this example (lines 55-57),

$$C_{11} = A_{11} = 1 \quad C_{12} = A_{12} = 4 \quad C_{13} = A_{13} = 7$$

$$C_{21} = C_{11} \times 2 = 2 \quad C_{22} = C_{12} \times 5 = 20 \quad C_{23} = C_{13} \times 8 = 56$$

$$C_{31} = C_{21} \times 3 = 6 \quad C_{32} = C_{22} \times 6 = 120 \quad C_{33} = C_{23} \times 9 = 504$$

When applied to a matrix, the output of the function `diff` (line 58) is an $(n-1) \times m$ (not $n \times m$) matrix \mathbf{D} , where

$$D_{ij} = A_{(i+1)j} - A_{ij}; i = 1, 2, \dots, n-1; j = 1, 2, \dots, m$$

In this example (lines 60-61),

$$D_{11} = 2 - 1 = 1 \quad D_{12} = 5 - 4 = 1 \quad D_{13} = 8 - 7 = 1$$

$$D_{21} = 3 - 2 = 1 \quad D_{22} = 6 - 5 = 1 \quad D_{23} = 9 - 8 = 1$$

Minima and Maxima

The output of the functions `min` or `max` for a vector are scalars (lines 62-67).

The output of the functions `min` or `max` for an $n \times m$ matrix is a row vector of m elements (lines 68-73), in which each element is the minimum/maximum of the corresponding column. #

2.8 Arithmetic Operators

[1] An arithmetic operator operates on one (unary operator) or two (binary operator) numeric data and the result is also a numeric data. If any of the operands is not a numeric data, it is converted to a numeric data. Table 2.8 lists some of the frequently used arithmetic operators.

Precedence Level of Operators

The precedence level of operators determines the order in which MATLAB evaluates an operation. We attach to each operator a precedence number as shown in Tables 2.8 (and Tables 2.9a, 2.9b); *the lower number has higher precedence level*. For operators with the same precedence level, the evaluation is from left to right. The parentheses () has highest precedence level (1), while the assignment = has lowest precedence level (13).

Names of Operators

An operator is actually a short hand of a function name. For example, $5+6$ is internally evaluated using the function call

```
>> plus(3,5)
ans =
     8
```

This feature is useful when creating classes and their associate operators. In Section 4.9, we'll demonstrate the creation of a class of polynomial, for which we'll implement the addition and subtraction of polynomials using the operators + and -.

Example02_08a.m

[2] These statements demonstrate some arithmetic operations on **matrices** (see the **Command Window** session in [3-4], next page). →

```
1 clear
2 A = reshape(1:6, 2, 3)
3 B = reshape(7:12, 2, 3)
4 C = A+B
5 D = A-B
6 E = B'
7 F = A*E
8 a = [3, 6]
9 b = a/F
10 c = b*F
11 G = F^2
12 H = A.*B
13 K = A./B
14 M = A.^2
15 P = A+10
16 Q = A-10
17 R = A*1.5
18 S = A/2
```

Table 2.8 Arithmetic Operators

Operator	Name	Description	Precedence level
+	plus	Addition	6
-	minus	Subtraction	6
*	mtimes	Multiplication	5
/	mrdivide	Division	5
^	mpower	Exponentiation	2
.*	times	Element-wise multiplication	5
./	rdivide	Element-wise division	5
.^	power	Element-wise exponentiation	2
-	uminus	Unary minus	4
+	uplus	Unary plus	4

Details and More:

Help>MATLAB>Language Fundamentals>Operators and Elementary Operations>Operator Precedence
Help>MATLAB>Language Fundamentals>Operators and Elementary Operations>Arithmetic

```

19 >> clear
20 >> A = reshape(1:6, 2, 3)
21 A =
22     1     3     5
23     2     4     6
24 >> B = reshape(7:12, 2, 3)
25 B =
26     7     9    11
27     8    10    12
28 >> C = A+B
29 C =
30     8    12    16
31    10    14    18
32 >> D = A-B
33 D =
34    -6    -6    -6
35    -6    -6    -6
36 >> E = B'
37 E =
38     7     8
39     9    10
40    11    12
41 >> F = A*E
42 F =
43    89    98
44   116   128
45 >> a = [3, 6]
46 a =
47     3     6
48 >> b = a/F
49 b =
50  -13.0000  10.0000
51 >> c = b*F
52 c =
53     3     6
54 >> G = F^2
55 G =
56    19289    21266
57    25172    27752
58 >> H = A.*B
59 H =
60     7    27    55
61    16    40    72
62 >> K = A./B
63 K =
64    0.1429    0.3333    0.4545
65    0.2500    0.4000    0.5000
66 >> M = A.^2
67 M =
68     1     9    25
69     4    16    36

```

[3] A Command Window session of Example02_08a.m (continued at [4]).

```

70 >> P = A+10
71 P =
72    11    13    15
73    12    14    16
74 >> Q = A-10
75 Q =
76    -9    -7    -5
77    -8    -6    -4
78 >> R = A*1.5
79 R =
80    1.5000    4.5000    7.5000
81    3.0000    6.0000    9.0000
82 >> S = A/2
83 S =
84    0.5000    1.5000    2.5000
85    1.0000    2.0000    3.0000

```

[4] A Command Window session of Example02_08a.m (Continued). →

Addition of Matrices

[5] Let A_{ij} , $i = 1, 2, \dots, n$; $j = 1, 2, \dots, m$ be the elements of an $n \times m$ matrix \mathbf{A} , and B_{ij} , $i = 1, 2, \dots, n$; $j = 1, 2, \dots, m$ be the elements of another $n \times m$ matrix \mathbf{B} . The addition (line 28) of the two matrices is an $n \times m$ matrix \mathbf{C} ,

$$C_{ij} = A_{ij} + B_{ij}$$

$$i = 1, 2, \dots, n; j = 1, 2, \dots, m$$

In this example (lines 30-31),

$$C_{11} = 1 + 7 = 8 \quad C_{12} = 3 + 9 = 12 \quad C_{13} = 5 + 11 = 16$$

$$C_{21} = 2 + 8 = 10 \quad C_{22} = 4 + 10 = 14 \quad C_{23} = 6 + 12 = 18$$

Subtraction of Matrices

The subtraction (line 32) of \mathbf{B} from \mathbf{A} is an $n \times m$ matrix \mathbf{D} ,

$$D_{ij} = A_{ij} - B_{ij}$$

$$i = 1, 2, \dots, n; j = 1, 2, \dots, m$$

In this example (lines 34-35),

$$D_{11} = 1 - 7 = -6 \quad D_{12} = 3 - 9 = -6 \quad D_{13} = 5 - 11 = -6$$

$$D_{21} = 2 - 8 = -6 \quad D_{22} = 4 - 10 = -6 \quad D_{23} = 6 - 12 = -6$$

Transpose of Matrices

The transpose (line 36; also see 1.6[12], page 27) of \mathbf{B} is an $m \times n$ matrix \mathbf{E} ,

$$E_{ij} = B_{ji}$$

$$i = 1, 2, \dots, m; j = 1, 2, \dots, n$$

In this example (lines 38-40)

$$E_{11} = B_{11} = 7 \quad E_{12} = B_{21} = 8$$

$$E_{21} = B_{12} = 9 \quad E_{22} = B_{22} = 10$$

$$E_{31} = B_{13} = 11 \quad E_{32} = B_{23} = 12$$

Multiplication of Matrices

The multiplication (line 41) of an $n \times m$ matrix \mathbf{A} by an $m \times p$ matrix \mathbf{E} is an $n \times p$ matrix \mathbf{F}

$$F_{ij} = \sum_{k=1}^m A_{ik} \times E_{kj}$$

$$i = 1, 2, \dots, n; j = 1, 2, \dots, p$$

In this example (lines 43-44), $n = 2$, $m = 3$, and $p = 2$, and the result is a 2×2 matrix:

$$F_{11} = 1 \times 7 + 3 \times 9 + 5 \times 11 = 89$$

$$F_{21} = 2 \times 7 + 4 \times 9 + 6 \times 11 = 116$$

$$F_{12} = 1 \times 8 + 3 \times 10 + 5 \times 12 = 98$$

$$F_{22} = 2 \times 8 + 4 \times 10 + 6 \times 12 = 128$$

Note that two matrices can be multiplied only if the two matrices have the same **inner dimension size**. →

Division of Matrices

[6] The division (line 48) of an $r \times m$ matrix \mathbf{a} by an $m \times m$ matrix \mathbf{F} (i.e., \mathbf{a}/\mathbf{F}) is an $r \times m$ matrix \mathbf{b} ; they are related by

$$\mathbf{b}_{r \times m} \times \mathbf{F}_{m \times m} = \mathbf{a}_{r \times m}$$

In this example (line 50), $r = 1$ and $m = 2$, and the result is a 1×2 row vector \mathbf{b} ,

$$\mathbf{b} = [-13 \quad 10]$$

since it satisfies (see lines 51-53)

$$\begin{bmatrix} -13 & 10 \end{bmatrix} \times \begin{bmatrix} 89 & 98 \\ 116 & 128 \end{bmatrix} = \begin{bmatrix} 3 & 6 \end{bmatrix}$$

Note that \mathbf{a} and \mathbf{F} must have the same **column size** and, in the above example, \mathbf{F} is a square matrix and the resulting matrix \mathbf{b} has the same dimension sizes as \mathbf{a} . In general, \mathbf{F} is not necessarily a square matrix. If \mathbf{F} is not a square matrix, then \mathbf{a}/\mathbf{F} will output a least-squares solution \mathbf{b} of the system of equations $\mathbf{b} \times \mathbf{F} = \mathbf{a}$ (see line 119, page 97, for example).

Exponentiation of Square Matrices

The exponentiation of a square matrix is the repeated multiplication of the matrix itself. For example (line 54)

$$\mathbf{F}^2 \equiv \mathbf{F} \times \mathbf{F}$$

In this example (lines 56-57)

$$\begin{aligned} \mathbf{F}^2 &= \begin{bmatrix} 89 & 98 \\ 116 & 128 \end{bmatrix} \times \begin{bmatrix} 89 & 98 \\ 116 & 128 \end{bmatrix} \\ &= \begin{bmatrix} 19289 & 21266 \\ 25172 & 27752 \end{bmatrix} \end{aligned}$$

Element-Wise Multiplication of Matrices

The element-wise multiplication ($\cdot *$ in line 58) operates on two $n \times m$ matrices of the same sizes \mathbf{A} and \mathbf{B} , and the result is a matrix \mathbf{H} of the same size,

$$\begin{aligned} H_{ij} &= A_{ij} \times B_{ij} \\ i &= 1, 2, \dots, n; j = 1, 2, \dots, m \end{aligned}$$

In this example (lines 60-61)

$$\begin{aligned} H_{11} &= 1 \times 7 = 7 & H_{12} &= 3 \times 9 = 27 & H_{13} &= 5 \times 11 = 55 \\ H_{21} &= 2 \times 8 = 16 & H_{22} &= 4 \times 10 = 40 & H_{23} &= 6 \times 12 = 72 \end{aligned}$$

Element-Wise Division of Matrices

The element-wise division ($\cdot /$ in line 62) also operates on two $n \times m$ matrices of the same sizes \mathbf{A} and \mathbf{B} , and the result is a matrix \mathbf{K} of the same size,

$$\begin{aligned} K_{ij} &= A_{ij} / B_{ij} \\ i &= 1, 2, \dots, n; j = 1, 2, \dots, m \end{aligned}$$

In this example (lines 64-65)

$$\begin{aligned} K_{11} &= 1/7 & K_{12} &= 3/9 = 1/3 & K_{13} &= 5/11 \\ K_{21} &= 2/8 = 0.25 & K_{22} &= 4/10 = 0.4 & K_{23} &= 6/12 = 0.5 \end{aligned} \quad \rightarrow$$

Element-Wise Exponentiation of Matrices

[7] The element-wise exponentiation (line 66) operates on an $n \times m$ matrix \mathbf{A} and a scalar q , and the result is an $n \times m$ matrix \mathbf{M} ,

$$M_{ij} = (A_{ij})^q$$

$$i = 1, 2, \dots, n; j = 1, 2, \dots, m$$

In this example (lines 68-69)

$$M_{11} = 1^2 = 1 \quad M_{12} = 3^2 = 9 \quad M_{13} = 5^2 = 25$$

$$M_{21} = 2^2 = 4 \quad M_{22} = 4^2 = 16 \quad M_{23} = 6^2 = 36$$

Operations Between a Matrix and a Scalar

Let $@$ be one of the operators $+$, $-$, $*$, $/$, $\cdot*$, $\cdot/$, or \cdot^{\wedge} , and s be a scalar, then $\mathbf{A}@s$ is an $n \times m$ matrix \mathbf{V} , where

$$V_{ij} = A_{ij}@s$$

$$i = 1, 2, \dots, n; j = 1, 2, \dots, m$$

and $s@\mathbf{A}$ is also an $n \times m$ matrix \mathbf{W} , where

$$W_{ij} = s@A_{ij}$$

$$i = 1, 2, \dots, n; j = 1, 2, \dots, m$$

For example (lines 70-85)

$$\mathbf{A} + 10 = \begin{bmatrix} 1+10 & 3+10 & 5+10 \\ 2+10 & 4+10 & 6+10 \end{bmatrix}$$

$$\mathbf{A} - 10 = \begin{bmatrix} 1-10 & 3-10 & 5-10 \\ 2-10 & 4-10 & 6-10 \end{bmatrix}$$

$$\mathbf{A} \times 1.5 = \begin{bmatrix} 1 \times 1.5 & 3 \times 1.5 & 5 \times 1.5 \\ 2 \times 1.5 & 4 \times 1.5 & 6 \times 1.5 \end{bmatrix}$$

$$\mathbf{A} / 2 = \begin{bmatrix} 1/2 & 3/2 & 5/2 \\ 2/2 & 4/2 & 6/2 \end{bmatrix}$$

In other words, an operation $\mathbf{A}@s$ or $s@\mathbf{A}$ can be thought of an element-wise operation in which the scalar s is expanded such that it has the same sizes as the matrix \mathbf{A} and each element has the same value s . For example

$$\mathbf{A} + 10 = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix} + \begin{bmatrix} 10 & 10 & 10 \\ 10 & 10 & 10 \end{bmatrix}$$

$$\mathbf{A} \times 1.5 = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix} \cdot * \begin{bmatrix} 1.5 & 1.5 & 1.5 \\ 1.5 & 1.5 & 1.5 \end{bmatrix}$$

→

Example02_08b.m

[8] These statements demonstrate some arithmetic operations on **vectors** (also see [9]). Remember that a vector is a special case of matrices. Thus operations on vectors are special cases of those on matrices.

```

86 clear
87 a = 1:4
88 b = 5:8
89 c = a+b
90 d = a-b
91 e = a*(b')
92 f = (a')*b
93 g = a/b
94 h = a.*b
95 k = a./b
96 m = a.^2

```

[9] This is a **Command Window** session of Example02_08b.m.

```

97 >> clear
98 >> a = 1:4
99 a =
100     1     2     3     4
101 >> b = 5:8
102 b =
103     5     6     7     8
104 >> c = a+b
105 c =
106     6     8    10    12
107 >> d = a-b
108 d =
109    -4    -4    -4    -4
110 >> e = a*(b')
111 e =
112     70
113 >> f = (a')*b
114 f =
115     5     6     7     8
116    10    12    14    16
117    15    18    21    24
118    20    24    28    32
119 >> g = a/b
120 g =
121     0.4023
122 >> h = a.*b
123 h =
124     5    12    21    32
125 >> k = a./b
126 k =
127     0.2000     0.3333     0.4286     0.5000
128 >> m = a.^2
129 m =
130     1     4     9    16

```

Arithmetic Operators for Vectors

[10] A vector is a special matrix, in which either the row-size or the column-size is equal to one. Thus, all the rules of the arithmetic operations for matrices apply to those for vectors (lines 97-130).

Division (/) by a Non-Square Matrix

In lines 119-121, since **b** is not a square matrix, **g** is not an exact solution of $\mathbf{g} * \mathbf{b} = \mathbf{a}$. Instead, **g** is the least-squares solution of the equation $\mathbf{g} * \mathbf{b} = \mathbf{a}$ (for details and more: `>> doc /`) In general, if **a** is an $r \times m$ matrix and **b** is a $t \times m$ matrix, then the result **g** of slash operator (/) is an $r \times t$ matrix.

In this case **a** is a 1×4 matrix and **b** is also a 1×4 matrix; therefore, the result **g** must be a 1×1 matrix, i.e., a scalar. MATLAB seeks the least-squares solution for the system of 4 equations:

$$g \times [5 \ 6 \ 7 \ 8] = [1 \ 2 \ 3 \ 4]$$

and the least-squares solution is $g = 0.4023$ (line 121). \rightarrow

Example02_08c.m

[11] These statements demonstrate some arithmetic operations on **scalars** (also see [12]). Remember that a scalar is a special case of matrices. Thus operations on scalar are special cases of those on matrices.

```
131 clear
132 a = 6
133 b = 4
134 c = a+b
135 d = a-b
136 e = a*b
137 f = a/b
138 g = a^2
139 h = a.*b
140 k = a./b
141 m = a.^2
```

[12] This is a **Command Window** session of Example02_08c.m.

```
142 >> clear
143 >> a = 6
144 a =
145     6
146 >> b = 4
147 b =
148     4
149 >> c = a+b
150 c =
151    10
152 >> d = a-b
153 d =
154     2
155 >> e = a*b
156 e =
157    24
158 >> f = a/b
159 f =
160    1.5000
161 >> g = a^2
162 g =
163    36
164 >> h = a.*b
165 h =
166    24
167 >> k = a./b
168 k =
169    1.5000
170 >> m = a.^2
171 m =
172    36
```

Arithmetic Operators for Scalars

[13] A scalar is a 1x1 matrix. Thus, all the rules of the arithmetic operations for matrices can apply to those for scalars (lines 142-172).

Note that, in cases of scalar operations, there is no difference between operators with or without a dot (.); i.e., for scalar operations, * is the same as .*, / is the same as ./, and ^ is the same as .^.

2.9 Relational and Logical Operators

Relational Operators

[1] A relational operator (Table 2.9a; also see 2.5[5], page 82) always operates on two numeric data (scalars, vectors, or matrices); the result is a logical data. If any of the operands is not a numeric data, it is converted to a numeric data.

As a rule, the two operands **A** and **B** must have the same sizes, and the result is a logical data of the same sizes (except `isequal`, which results a single logical value). However, when one of the operands is a scalar, e.g., $\mathbf{A} > s$, where s is a scalar and **A** is a matrix, the scalar is expanded such that it has the same sizes as the matrix **A** and each element has the same value s (also see 2.8[7], page 96).

Logical Operators

A logical operator (Table 2.9b; also see 2.5[5], page 82) operates on one or two logical data (scalars, vectors, or matrices), and the result is a logical data of the same sizes. If any of the operands is not a logical data, it is converted to a logical data: a non-zero value is converted to a `true` and a zero value is converted to a `false`.

Table 2.9a Relational Operators

Operator	Description	Precedence level
<code>==</code>	Equal to	8
<code>~=</code>	No equal to	8
<code>></code>	Greater than	8
<code><</code>	Less than	8
<code>>=</code>	Greater than or equal to	8
<code><=</code>	Less than or equal to	8
<code>isequal</code>	Determine array equality	

Details and More: Help>MATLAB>Language Fundamentals>Operators and Elementary Operations>Relational Operations

Table 2.9b Logical Operators

Operator	Description	Precedence level
<code>&</code>	Logical AND	9
<code> </code>	Logical OR	10
<code>~</code>	Logical NOT	4
<code>&&</code>	Logical AND (short-circuit)	11
<code> </code>	Logical OR (short-circuit)	12

Details and More: Help>MATLAB>Language Fundamentals>Operators and Elementary Operations>Logical Operations

Example02_09.m

[2] These statements demonstrate some relational and logical operations (also see [3, 4], next page). →

```

1 clear
2 A = [5,0,-1; 3,10,2; 0,-4,8]
3 Map = (A > 6)
4 location = find(Map)
5 list = A(location)
6 list2 = A(find(A>6))
7 list3 = A(find(A>0 & A<=8 & A~=3))
8 find(A)'
9 ~A
10 ~~A
11 isequal(A, ~~A)

```

```

12 clear
13 A = [5,0,-1; 3,10,2; 0,-4,8]
14 A =
15     5     0    -1
16     3    10     2
17     0    -4     8
18 Map = (A > 6)
19 Map =
20     3×3 logical array
21     0     0     0
22     0     1     0
23     0     0     1
24 location = find(Map)
25 location =
26     5
27     9
28 list = A(location)
29 list =
30     10
31     8
32 list2 = A(find(A>6))
33 list2 =
34     10
35     8
36 list3 = A(find(A>0 & A<=8 & A~=3))
37 list3 =
38     5
39     2
40     8
41 find(A)'
42 ans =
43     1     2     5     6     7     8     9
44 ~A
45 ans =
46     3×3 logical array
47     0     1     0
48     0     0     0
49     1     0     0
50 ~~A
51 ans =
52     3×3 logical array
53     1     0     1
54     1     1     1
55     0     1     1
56 isequal(A, ~~A)
57 ans =
58     logical
59     0

```

[3] This is a **Command Window** session of Example02_09.m.

Name	Value	Size	Bytes	Class
A	[5,0,-1;3,10,2;0,-4,8]	3x3	72	double
ans	0	1x1	1	logical
list	[10;8]	2x1	16	double
list2	[10;8]	2x1	16	double
list3	[5;2;8]	3x1	24	double
location	[5;9]	2x1	16	double
Map	3x3 logical	3x3	9	logical

[4] the **Workspace**.

About Example02_09.m

[5] In line 18, the operation `A > 6` results in a logical-value matrix the same size as `A` (lines 20-23). The logical matrix can be thought as a "map" indicating the locations of the elements that are greater than 6. Note that the parentheses in line 18 are not needed, since the assignment (`=`) has lowest precedence level (2.8[1], page 92). We sometimes add redundant parentheses to enhance the readability, avoiding confusion.

Function `find`

The function `find` (line 24) takes a **logical array** as input argument and outputs a **column vector** of numbers that are the **linear indices** (1.9[23], page 37) of the elements with the value `true` in the array.

Here (line 24), the 5th and 9th elements (in linear indexing) of `Map` have the value `true`, so it outputs a column vector consisting of 5 and 9 (lines 26-27).

In line 28, vector `location` is used to access the array `A`, the result is a column vector containing the numbers in `A(5)` and `A(9)` (lines 30-31).

If we are concerned only with the numbers themselves (not the locations), then the commands in lines 18, 24, and 28 can be combined, as shown in line 32, resulting in the same two values (lines 34-35).

Using function `find` with a logical expression as the input argument allows us to find the elements in an array that meet specific conditions. Suppose we want to find the numbers in the array `A` that are positive, less than or equal to 8, but not equal to 3; we may write the statement as shown in line 36, and the result is in lines 38-40. →

Just for Fun

[6] Just for fun: how about we type `find(A)`? Since the input argument must be a logical data, `A` is converted to logical type: all the non-zero values are treated as `true` and zero values are treated as `false`. The result is that the 1st, 2nd, 5th, 6th, 7th, 8th, and 9th elements have the `true` value (line 43). Note that, in line 41, we transpose the resulting column vector to a row vector to save the output space.

Logical NOT (~)

The logical NOT (~) reverses logical values: `true` becomes `false`, and `false` becomes `true`. If we apply it on a numerical array (line 44), a nonzero value becomes `false` and a zero value becomes `true` (lines 47-49). If we apply the logical NOT again on the previous result (line 50), the outcome is of course a logical array. Here, we want to show that for a numerical array, in general,

$$(\sim\sim A) \neq A$$

This is demonstrated in lines 56-59. The function `isequal` (line 56) compares two arrays and outputs `true` if they are equal in size and contents, otherwise outputs `false`.

Short-Circuit Logical AND (&&) and OR (| |)

Let `expr1` and `expr2` be two logical expressions. The result of `expr1&&expr2` is the same as `expr1&expr2`, but the former is more efficient (i.e., less computing time). Similarly, the result of `expr1| |expr2` is the same as `expr1|expr2`, but the former is more efficient. The operators `&&` and `| |` are called short-circuit logical operators (see Table 2.9b, page 99).

In `expr1&&expr2`, `expr2` is evaluated only if the result is not fully determined by `expr1`. For example, if `expr1` equals `false`, then the entire expression evaluates to `false`, regardless of the value of `expr2`. Under these circumstances, there is no need to evaluate `expr2` because the result is already known.

Similarly, in `expr1| |expr2`, `expr2` is evaluated only if the result is not fully determined by `expr1`. For example, if `expr1` equals `true`, then the entire expression evaluates to `true`, regardless of the value of `expr2`. Under these circumstances, there is no need to evaluate `expr2` because the result is already known.

Logical Indexing

In line 6, we are finding the elements in `A` such that they are greater than 6. This task can be accomplished by means of **logical indexing**:

```
>> A(A>6)
```

The result is the same as line 6. In other words, the logical matrix `Map` can be viewed as an **indexing matrix**, and

```
>> A(Map)
```

outputs all the elements `A(i, j)` for which their corresponding `Map(i, j)` is true.

Using the logical indexing, line 7 can be simplified as follows:

```
>> list3 = A(A>0 & A<=8 & A~=3)
```

```
#
```


2.10 String Manipulations

[1] A row vector of characters is also called a **string**. Single quotes are used to represent strings; e.g., 'ABC' represents a row vector of three characters (2.4[1], page 78). Table 2.10 (page 104) summarizes some useful functions for string manipulations.

Example02_10a.m: String Manipulations

[2] Type and run the following statements, which demonstrate some string manipulations. Input your name and age as shown in [3].

```

1  clear
2  a = 'Hello,';
3  b = 'world!';
4  c = [a, ' ', b];
5  disp(c)
6  name = input('What is your name? ', 's');
7  years = input('What is your age? ');
8  disp(['Hello, ', name, '! You are ', num2str(years), ' years old.'])
9  str = sprintf('Pi = %.8f', pi);
10 disp(str)
11 Names1 = [
12     'David '
13     'John '
14     'Stephen'];
15 Names2 = char('David', 'John', 'Stephen');
16 if isequal(Names1, Names2)
17     disp('The two lists are equal.')
18 end
19 name = deblank(Names1(2,:));
20 disp(['The name ', name, ' has ', num2str(length(name)), ' characters.'])

```

```

21 >> Example02_10a
22 Hello, world!
23 What is your name? Lee
24 What is your age? 60
25 Hello, Lee! You are 60 years old.
26 Pi = 3.14159265
27 The two lists are equal.
28 The name John has 4 characters.
29 >>

```

[3] This is a test run of Example02_10a.m.

[4] Remember that a string is a row vector of characters. The most convenient way to concatenate strings is using square brackets (line 4). `c` is a row vector of 13 characters.

The function `disp` displays a data of any type (lines 5 and 22). A newline character (`\n`) is automatically output at the end of the display. On the other hand, you need to append newline characters when using the function `fprintf` if you want the cursor to move to the next line.

Line 6 requests a data input from the screen. With the 's' as the second argument, the input data will be read as a string, and you don't have to include the single quotes for the string (line 23). Without the 's' as the second argument, you would have to use the single quotes (') to input a string, otherwise the data will be read as a `double` (lines 7 and 24). →

[5] The function `sprintf` (line 9) is the same as `fprintf` except that it writes formatted data to a string (rather than to the screen) and outputs the string. Here, we write the value of π with 8 digits after the decimal point in the string `str` (line 9) and display the string (lines 10 and 26).

In lines 11-14, a single statement continues for 4 lines, without using ellipses (`...`). A newline character between a pair of square brackets is treated as a semicolon plus an ellipsis. Thus, this statement creates 3 rows of strings; it is a 3-by-7 matrix of `char`. Note that rows of a matrix must have the same length. That's why we pad the first two strings with trailing blanks (lines 12-13) to make the lengths of the three strings equal.

Another way to create a column of strings (i.e., a matrix of `char`) is using the function `char`. Line 15 creates a column of strings exactly the same as that in lines 11-14. The function `char` automatically pads the strings with trailing blanks so that the three strings have the same length.

Lines 16-18 and 27 confirm the equality of the two matrices.

The function `deblank` (line 19) removes trailing blanks from a string. `Names1(2, :)` refers to the entire 2nd row (i.e., the string 'John '). After removing the trailing blanks, four characters remain in the string (lines 20 and 28).

Example02_10b.m: A Simple Calculator

[6] This program uses function `eval` to create a simple calculator. Type and run the program (see a test run in [7]).

```
30 clear
31 disp('A Simple Calculator')
32 while true
33     expr = input('Enter an expression (or quit): ', 's');
34     if strcmp(expr, 'quit')
35         break
36     end
37     disp([expr, ' = ', num2str(eval(expr))])
38 end
```

```
39 >> Example02_10b
40 A Simple Calculator
41 Enter an expression (or quit): 3+5
42 3+5 = 8
43 Enter an expression (or quit): sin(pi/4) + (2 + 2.1^2)*3
44 sin(pi/4) + (2 + 2.1^2)*3 = 19.9371
45 Enter an expression (or quit): quit
46 >>
```

[7] This is a test run of the program Example02_8b.m. #

Table 2.10 String Manipulations

Function	Description
<code>A = char(a,b,...)</code>	Convert the strings to a matrix of rows of the strings, padding blanks
<code>disp(X)</code>	Display value of variable
<code>x = input(prompt, 's')</code>	Request user input
<code>s = sprintf(format,a,b,...)</code>	Write formatted data to a string
<code>s = num2str(x)</code>	Convert number to string
<code>x = str2num(s)</code>	Convert string to number
<code>x = str2double(s)</code>	Convert string to double precision value
<code>x = eval(exp)</code>	Evaluate a MATLAB expression
<code>s = deblank(str)</code>	Remove trailing blanks from a string
<code>s = strtrim(str)</code>	Remove leading and trailing blanks from a string
<code>tf = strcmp(s1,s2)</code>	Compare two strings (case sensitive)
<code>tf = strcmpi(s1,s2)</code>	Compare two strings (case insensitive)
<i>Details and More:</i> <i>Help>MATLAB>Language Fundamentals>Data Types>Characters and Strings; Data Type Conversion</i>	

2.11 Expressions

[1] An expression is a syntactic combination of **data** (constants or variables; scalars, vectors, matrices, or higher-dimensional arrays), **functions** (built-in or user-defined), **operators** (arithmetic, relational, or logical), and **special characters** (see Table 2.11a, page 107). An expression always results in a **value** of certain type, depending on the operators.

Table 2.11b (page 107) lists some elementary math functions that are frequently used.

Example: Law of Sines

[2] The law of sines for an arbitrary triangle states that (see *Wikipedia*>*Trigonometry*):

$$\frac{a}{\sin \alpha} = \frac{b}{\sin \beta} = \frac{c}{\sin \gamma} = \frac{abc}{2A} = 2R$$

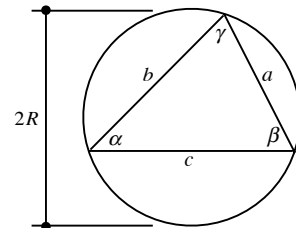
where α , β , and γ are the three angles of a triangle; a , b , and c are the lengths of the sides opposite to the respective angles; A is the area of the triangle; R is the radius of the circumscribed circle of the triangle:

$$R = \frac{abc}{\sqrt{(a+b+c)(a-b+c)(b-c+a)(c-a+b)}}$$

Knowing a , b , and c , then α , β , γ , and A can be calculated as follows:

$$\alpha = \sin^{-1} \frac{a}{2R}, \quad \beta = \sin^{-1} \frac{b}{2R}, \quad \gamma = \sin^{-1} \frac{c}{2R}$$

$$A = \frac{abc}{4R} \quad \text{or} \quad A = \frac{1}{2}bc \sin \alpha$$



Example02_11a.m: Law of Sines

[3] This script calculates the three angles α , β , γ of a triangle and its area A , given three sides $a = 5$, $b = 6$, and $c = 7$.

→

```

1 clear
2 a = 5;
3 b = 6;
4 c = 7;
5 R = a*b*c/sqrt((a+b+c)*(a-b+c)*(b-c+a)*(c-a+b))
6 alpha = asind(a/(2*R))
7 beta = asind(b/(2*R))
8 gamma = asind(c/(2*R))
9 sumAngle = alpha + beta + gamma
10 A1 = a*b*c/(4*R)
11 A2 = b*c*sind(alpha)/2

```

```

12  >> Example02_11a
13  R =
14      3.5722
15  alpha =
16      44.4153
17  beta =
18      57.1217
19  gamma =
20      78.4630
21  sumAngle =
22      180
23  A1 =
24      14.6969
25  A2 =
26      14.6969
27  >>

```

[4] This is the screen output of Example02_11a.m.

About Example02_11a.m

[5] Function `sind` (line 11) is the same as the function `sin` in 1.2[7] (page 13) except that `sind` assumes degrees (instead of radians) as the unit of the input argument. Similarly, the function `asind` (lines 6-8), inverse sine function, outputs an angle in degrees (the function `asin` outputs an angle in radians).

Line 9 confirms that the sum of the three angles is indeed 180 degrees (line 22). Lines 10-11 calculate the area of the triangle using two different formulas in [2], and they indeed have the same values (lines 23-26).

Example02_11b.m: Law of Cosines

[6] The law of cosines states that (see *Wikipedia>Trigonometry*; use the same notations in [2], last page):

$$a^2 = b^2 + c^2 - 2bc \cos \alpha \quad \text{or} \quad \alpha = \cos^{-1} \frac{b^2 + c^2 - a^2}{2bc}$$

With $a = 5$, $b = 6$, $c = 7$, the angle α , β , and γ can be calculated as follows:

```

28  clear
29  a = 5; b = 6; c = 7;
30  alpha = acosd((b^2+c^2-a^2)/(2*b*c))
31  beta = acosd((c^2+a^2-b^2)/(2*c*a))
32  gamma = acosd((a^2+b^2-c^2)/(2*a*b))

```

[7] This is the screen output of Example02_11b.m. The output is consistent with that in [4]. #

```

33  >> Example02_11b
34  alpha =
35      44.4153
36  beta =
37      57.1217
38  gamma =
39      78.4630
40  >>

```

Table 2.11a
Special Characters

Special characters	Description
[]	Brackets
{ }	Braces
()	Parentheses
'	Matrix transpose
.	Field access
...	Continuation
,	Comma
;	Semicolon
:	Colon
@	Function handle
<p><i>Details and More:</i> <i>Help>MATLAB>Language Fundamentals>Operators and Elementary Operations>MATLAB Operators and Special Characters</i></p>	

Table 2.11b
Elementary Math Functions

Function	Description
<code>sin(x)</code>	Sine (in radians)
<code>sind(x)</code>	Sine (in degrees)
<code>asin(x)</code>	Inverse sine (in radians)
<code>asind(x)</code>	Inverse sine (in degrees)
<code>cos(x)</code>	Cosine (in radians)
<code>cosd(x)</code>	Cosine (in degrees)
<code>acos(x)</code>	Inverse cosine (in radians)
<code>acosd(x)</code>	Inverse cosine (in degrees)
<code>tan(x)</code>	Tangent (in radians)
<code>tand(x)</code>	Tangent (in degrees)
<code>atan(x)</code>	Inverse tangent (in radians)
<code>atand(x)</code>	Inverse tangent (in degrees)
<code>atan2(y,x)</code>	Four-quadrant inverse tangent (radians)
<code>atan2d(y,x)</code>	Four-quadrant inverse tangent (degrees)
<code>abs(x)</code>	Absolute value
<code>sqrt(x)</code>	Square root
<code>exp(x)</code>	Exponential (base e)
<code>log(x)</code>	Logarithm (base e)
<code>log10(x)</code>	Logarithm (base 10)
<code>factorial(n)</code>	Factorial
<code>sign(x)</code>	Sign of a number
<code>rem(a,b)</code>	Remainder after division
<code>mod(a,m)</code>	Modulo operation
<p><i>Details and More:</i> <i>Help>MATLAB>Mathematics>Elementary Math</i></p>	

2.12 Example: Function Approximation

Taylor Series

[1] In the hardware level, your computer can only perform simple arithmetic calculations such as addition, subtraction, multiplication, division, etc. Evaluation of a function value such as $\sin(\pi/4)$ is usually carried out with software or firmware. But how? In this section, we use $\sin(x)$ as an example to demonstrate the idea. This section also guides you to familiarize yourself with the way of thinking when using matrix expressions.

The sine function can be approximated using a Taylor series (Section 9.6 gives more and details of the Taylor series):

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots \quad (\text{a})$$

The more terms, the more accurate the approximation.

Example02_12a.m: Scalar Expressions

[2] This script calculates the value of $\sin(\pi/4)$ using the Taylor series in Eq. (a). The screen output is shown in [3].

```

1 clear
2 x = pi/4;
3 term1 = x;
4 term2 = -x^3/(3*2);
5 term3 = x^5/(5*4*3*2);
6 term4 = -x^7/(7*6*5*4*3*2);
7 format long
8 sinx =term1+term2+term3+term4
9 exact = sin(x)
10 error = (sinx-exact)/exact

```

```

11 sinx =
12     0.707106469575178
13 exact =
14     0.707106781186547
15 error =
16     -4.406850247592559e-07

```

[3] This is the screen output.

About Example02_12a.m

[4] We used 4 terms to calculate the function value $\sin(x)$ (lines 2-8, 11-12). Line 9 calculates the function values using the built-in function `sin` (line 14), which is used as a baseline for comparison. Line 10 calculates the error of the approximation (line 16). We conclude that, with merely four terms, the program calculates a function value accurately to an order of 10^{-7} . →

Using For-Loop

[5] In theory, an infinite number of terms of polynomials is required to achieve the exact value of $\sin(x)$. We need a general representation of these terms. We may rewrite the Taylor series in Eq. (a) as follows:

$$\sin x = \sum_{k=1}^{\infty} (-1)^{k+1} \frac{x^{2k-1}}{(2k-1)!} \quad (\text{b})$$

We now use a `for`-loop (Sections 1.14, 3.4) to calculate $\sin(\pi/4)$.

Example02_12b.m: Use of For-Loop

[6] Type and run this program. The screen output is the same as that of Example02_12a.m ([3], last page).

```

17 clear
18 x = pi/4; n = 4; sinx = 0;
19 for k = 1:n
20     sinx = sinx + ((-1)^(k-1))*(x^(2*k-1))/factorial(2*k-1);
21 end
22 format long
23 sinx
24 exact = sin(x)
25 error = (sinx-exact)/exact

```

About Example02_12b.m

[7] In line 18, the variable `sinx` is initialized to zero. The statement within the `for`-loop (line 20) runs four passes. In each pass, the variable `k` is assigned 1, 2, 3, and 4, respectively; a term is calculated according to the formula in Eq. (b) and added to the variable `sinx`. At the completion of the `for`-loop, `sinx` has the function value (lines 23 and 11-12). To increase the accuracy of the value, you may simply increase the number of items (see [8], for example).

[8] This is the screen output when 6 items are used (i.e., change to `n = 6` in line 18). Note that the error reduces to an order of 10^{-12} .

```

sinx =
    0.707106781179619
exact =
    0.707106781186547
error =
   -9.797690960678494e-12

```

Example02_12c.m: Vector Expressions

[9] This script produces the same output as that of Example02_12b.m ([3], last page) using a vector expression (line 29) in place of the `for`-loop used in Example02_12b.m (lines 19-21). →

```

26 clear
27 x = pi/4; n = 4; k = 1:n;
28 format long
29 sinx = sum((-1).^(k-1)).*(x.^(2*k-1))./factorial(2*k-1)
30 exact = sin(x)
31 error = (sinx-exact)/exact

```


About Example02_12c.m

[10] In line 27, the variable `k` is created as a row vector of four elements; `k = [1,2,3,4]`. The `for`-loop (lines 19-21) is now replaced by a vector expression (line 29), which uses the function `sum` and element-wise operators (`.^`, `.*`, and `./`). To help you understand the statement in line 29, we break it into several steps:

```
step1 = k-1
step2 = (-1).^step1
step3 = 2*k-1
step4 = x.^step3
step5 = step2.*step4
step6 = factorial(step3)
step7 = step5./step6
step8 = sum(step7)
sinx = step8
```

Using `k = [1,2,3,4]`, following the descriptions of element-wise operations (2.8[6-7], pages 95-96) and the function `sum` for vectors (2.7[5], page 90), we may further decode these steps as follows:

```
step1 = [0,1,2,3]
step2 = (-1).^[0,1,2,3] ≡ [1,-1,1,-1]
step3 = [1,3,5,7]
step4 = x.^[1,3,5,7] ≡ [x,x^3,x^5,x^7]
step5 = [1,-1,1,-1].*[x,x^3,x^5,x^7] ≡ [x,-x^3,x^5,-x^7]
step6 = factorial([1,3,5,7]) ≡ [1,6,120,5040]
step7 = [x,-x^3,x^5,-x^7]./[1,6,120,5040] ≡ [x,-x^3/6,x^5/120,-x^7/5040]
step8 = x-x^3/6+x^5/120-x^7/5040
sinx = x-x^3/6+x^5/120-x^7/5040
```

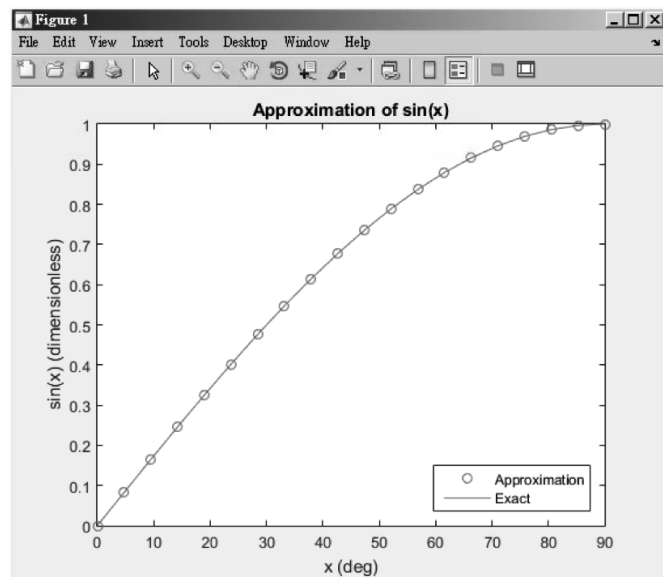
Substituting `x` with `pi/4`, we have `sinx = 0.707106469575178`.

Example02_12d.m: Matrix Expressions

[11] This script calculates `sin(x)` for various `x` values and produces a graph as shown in [12], next page. A matrix expression (line 37) is used in this script. →

```
32 clear
33 x = linspace(0,pi/2,20);
34 n = 4;
35 k = 1:n;
36 [X, K] = meshgrid(x, k);
37 sinx = sum((-1).^(K-1)).*(X .^ (2*K-1))./factorial(2*K-1));
38 plot(x*180/pi, sinx, 'o', x*180/pi, sin(x))
39 title('Approximation of sin(x)')
40 xlabel('x (deg)')
41 ylabel('sin(x) (dimensionless)')
42 legend('Approximation', 'Exact', 'Location', 'southeast')
```

[12] Example02_12d.m plots an approximation of $\sin(x)$ using 4-term Taylor series for various x values (circular marks); it also plots a solid curve representing the exact function values.



Function meshgrid

[13] Line 33 creates a row vector of 20 x -values using the function `linspace` (2.6[5], page 85). Line 35 creates a row vector of four integers.

Line 36 generates 2-D grid coordinates X and K based on the coordinates in the vectors x and k ; the sizes of X and K is also based on the sizes of x and k . If the length of x is n_x and the length of k is n_k , then both X and K are n_k -by- n_x matrices; each row of X is a copy of x , and each column of K is a copy of k' . In other words, line 36 is equivalent to the following statements:

```
nx = length(x); nk = length(k);
X = repmat(x, nk, 1);
K = repmat(k', 1, nx);
```

where the function `repmat` was used in lines 5-6 of Example01_06.m (page 25) and explained in 1.6[11-12] (pages 26-27). Actually, lines 5-6 of Example01_06.m can be replaced by the following statement (try it):

```
[Time, Theta] = meshgrid(time, theta)
```

Here, in line 36, both X and K are matrices. X contains angle values varying along column-direction while keeping constant in row-direction. K contains item-numbers (1, 2, 3, and 4) varying along row-direction while keeping constant in column-direction. Please verify this yourself.

Matrix Expression (Line 37)

Analysis of line 37 is similar to that of line 29 (see [10], last page). However, now we're dealing with matrices. The argument of the function `sum` is now a 4-by-20 matrix, each column corresponding to an angle value, each row corresponding to a k value. The function `sum` sums up the four values in each column (2.7[6], page 91), resulting a row vector of 20 values, which will be plotted as circular marks.

Plotting Marks and Curve

Line 38 plots the 20 values with circular marks, along with the exact function values, by default, using a solid line. Line 42 adds legends (Section 5.7) on the lower-right corner of the graphic area. →

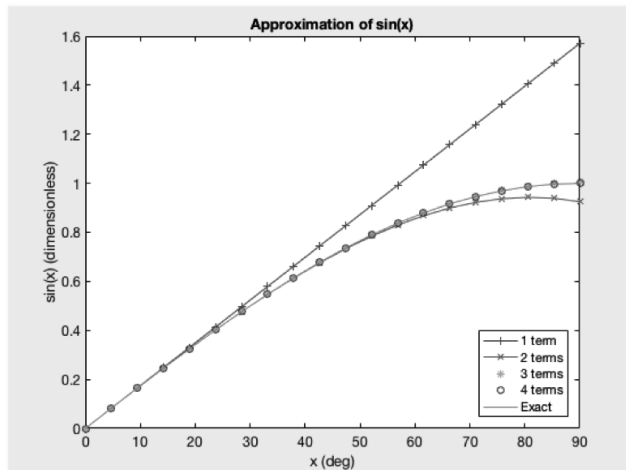
Example02_12e.m: Multiple Curves

[14] This script plots four approximated curves and an exact curve of $\sin(x)$ as shown in [15], the four approximated curves corresponding to the Taylor series of 1, 2, 3, and 4 items, respectively.

```

43 clear
44 x = linspace(0,pi/2,20);
45 n = 4;
46 k = (1:n);
47 [X, K] = meshgrid(x, k);
48 sinx = cumsum((-1).^(K-1).*(X.^(2*K-1))./factorial(2*K-1));
49 plot(x*180/pi, sinx(1,:), '+-', ...
50      x*180/pi, sinx(2,:), 'x-', ...
51      x*180/pi, sinx(3,:), '*', ...
52      x*180/pi, sinx(4,:), 'o', ...
53      x*180/pi, sin(x))
54 title('Approximation of sin(x)')
55 xlabel('x (deg)')
56 ylabel('sin(x) (dimensionless)')
57 legend('1 term', '2 terms', '3 terms', '4 terms', 'Exact', ...
58        'Location', 'southeast')

```



[15] Example02_12e.m plots four approximated curves and an exact curve of $\sin(x)$, for comparison.

About Example02_12e.m

[16] Line 48 looks like line 37 except that the function `cumsum` (2.7[6], page 91) is used instead of `sum`. The function `cumsum` calculates the cumulative sums of the four values in each column, resulting in a 4-by-20 matrix, the k^{th} row representing the approximated function values when k terms are added up. Lines 49-53 plot the four approximated curves, each a row of the function values, and the exact curve.

Line Styles and Marker Types

[17] In line 49, the notation `'+-'` specifies a plus **marker** and a solid **line style**. Similarly, in line 50, `'x-'` specifies a cross marker and a solid line style. Table 5.5a (page 226) lists various line styles and marker types.

Legend

Lines 57-58 add a **Legend** (Section 5.7) on the "southeast" (i.e., the lower-right) of the **Axes**. #

2.13 Example: Series Solution of a Laplace Equation

Laplace Equations

[1] Laplace equations have many applications (see *Wikipedia>Laplace's equation* or any Engineering Mathematics textbooks). Consider a Laplace equation in a two-dimensional, cartesian coordinate system:

$$\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = 0$$

subject to the boundary conditions $\phi(x,0) = \phi(x,1) = \phi(1,y) = 0$ and $\phi(0,y) = y(1-y)$, where $0 \leq x \leq 1$ and $0 \leq y \leq 1$.

A series solution of the equation is

$$\phi(x,y) = 4 \sum_{k=1}^{\infty} \frac{1 - \cos(k\pi)}{(k\pi)^3} e^{-k\pi x} \sin(k\pi y) \quad (a)$$

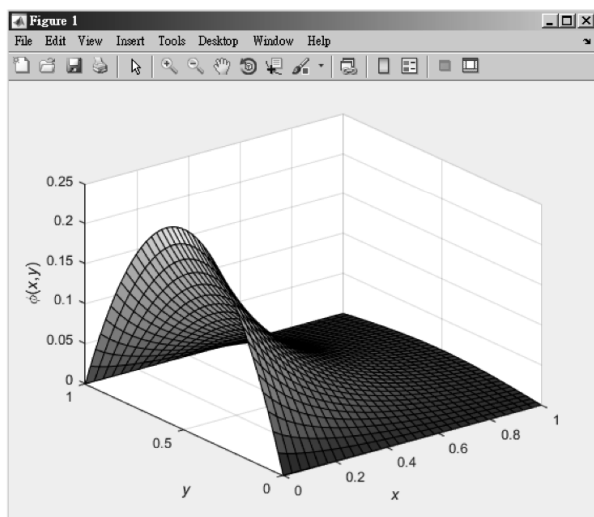
This can be verified by substituting the solution into the equation and the boundary conditions.

Example02_13.m: Series Solution of a Laplace Equation

[2] This script calculates the solution $\phi(x,y)$ according to Eq. (a) and plots a three-dimensional surface $\phi = \phi(x,y)$ [3].

```

1 clear
2 k = 1:20;
3 x = linspace(0,1,30);
4 y = linspace(0,1,40);
5 [X,Y,K] = meshgrid(x, y, k);
6 Phi = sum(4*(1-cos(K*pi))./(K*pi).^3.*exp(-K.*X*pi).*sin(K.*Y*pi), 3);
7 surf(x, y, Phi)
8 xlabel('\itx')
9 ylabel('\ity')
10 zlabel('\phi(\itx\rm,\ity\rm)')
```



[3] This program plots a surface representing the solution Eq. (a) of a Laplace equation.

[4] **Workspace.** →

Workspace	
Name	Value
k	1x20 double
K	40x30x20 double
Phi	40x30 double
x	1x30 double
X	40x30x20 double
y	1x40 double
Y	40x30x20 double

3-D Grid Coordinates Created with `meshgrid`

[5] Line 5 generates 3-D grid coordinates `X`, `Y`, and `K` defined by the vectors `x`, `y`, and `k`. If the length of `x`, `y`, and `k` are `nx`, `ny`, and `nk`, respectively, then `X`, `Y`, and `K` have sizes `ny-by-nx-by-nk` (see [4], last page). This statement is equivalent to the following statements:

```
nx = length(x); ny = length(y); nk = length(k);
X = repmat(x, ny, 1, nk);
Y = repmat(y', 1, nx, nk);
K = repmat(reshape(k, 1, 1, nk), ny, nx, 1);
```

The function `reshape` in the last statement reshapes the vector `k` to a $1 \times 1 \times nk$ vector, which is called a **page-vector** (remember that a $1 \times m$ matrix is called a **row-vector**, and an $m \times 1$ matrix is called a **column-vector**). Note that `X` varies in column-dimension while keeping constant in both row-dimension and page-dimension; `Y` varies in row-dimension while keeping constant in both column-dimension and page-dimension; `K` varies in page-dimension while keeping constant in both row-dimension and column-dimension.

3-D Array Expressions (Line 6)

While the expression in line 37 of `Example02_12d.m` (page 110) is a matrix expression, the expression in line 6 (last page) is a 3D extension of matrix expressions.

Remember, by default, function `sum` sums up the values along the first dimension (i.e., the row-dimension, see 2.7[6], page 91); however, you may specify the dimension along which the summing is performed. Here, line 6, the second argument of the function `sum` specifies that the summing is along the third-dimension (i.e., the page-dimension), resulting in a 40-by-30 matrix.

Function `surf`

In line 9 of `Example01_06.m` (page 25), the function `surf` takes three matrices as input arguments. Here, in line 7 (last page) the first two input arguments are row vectors. Line 7 is equivalent to the following statements

```
nx = length(x); ny = length(y);
X = repmat(x, ny, 1);
Y = repmat(y', 1, nx);
surf(X, Y, Phi);
```

Greek Letters and Math Symbols

Lines 8-10 add label texts to the plot. Here, to display the Greek letter ϕ , the character sequence `\phi` is used (line 10).

The character sequence `\it` (italicize) is used to specify the beginning of a series of italicized characters, and `\rm` is used to remove the italicization.

The character sequence for frequently used Greek letters and math symbols is listed in Table 5.6a (page 229). #

2.14 Example: Deflection of Beams

Simply Supported Beams

[1] Consider a simply supported beam subject to a force F (see [2], *source: https://en.wikipedia.org/wiki/File:Simple_beam_with_offset_load.svg, by Hermoere*). The beam has a cross section of width w and height h , and a Young's modulus E . The deflection y of the beam as a function of x is (Reference: *W. C. Young, Roark's Formula for Stress & Strain, 6th ed, p. 100*)

$$y = -\theta x + \frac{Rx^3}{6EI} - \frac{F}{6EI} \langle x-a \rangle^3$$

where

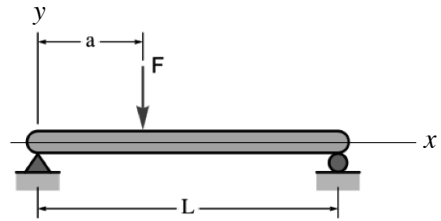
$$\theta = \frac{Fa}{6EIL}(2L-a)(L-a), R = \frac{F}{L}(L-a), I = \frac{wh^3}{12}$$

$$\langle x-a \rangle^3 = \begin{cases} 0, & \text{if } x \leq a \\ (x-a)^3, & \text{if } x > a \end{cases}$$

Physical meaning of these quantities is as follows: θ is the clockwise rotational angle of the beam at the left end; R is the reaction force on the beam at the left end; I is the area moment of inertia of the cross section.

In this section, we'll plot a deflection curve and find the maximum deflection and its corresponding location, using the following parameters:

$$w = 0.1 \text{ m}, h = 0.1 \text{ m}, L = 8 \text{ m}, E = 210 \text{ GPa}, F = 3000 \text{ N}, a = L/4$$



[2] This is the simply supported beam considered in this section.

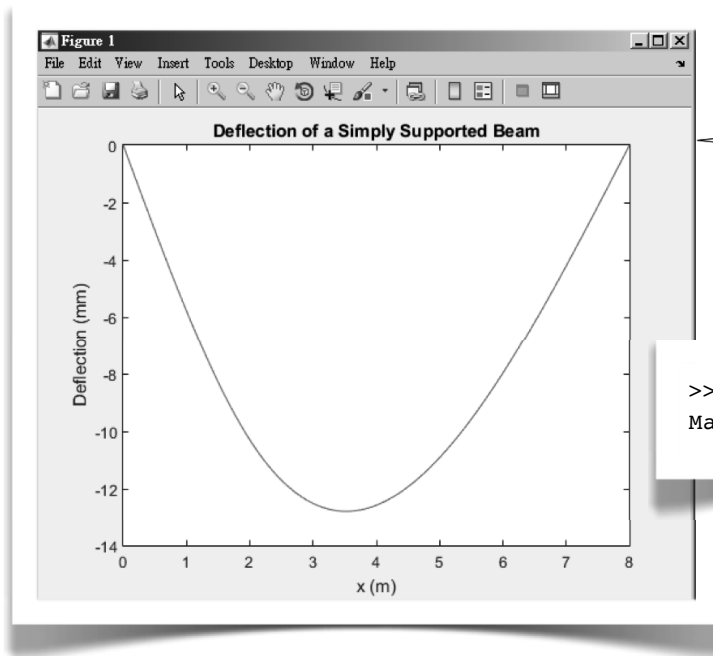
Example02_14.m: Deflection of Beams

[3] This script produces a graphic output as shown in [4] (next page) and a text output as shown in [5] (next page). →

```

1 clear
2 w = 0.1;
3 h = 0.1;
4 L = 8;
5 E = 210e9;
6 F = 3000;
7 a = L/4;
8 I = w*h^3/12;
9 R = F/L*(L-a);
10 theta = F*a/(6*E*I*L)*(2*L-a)*(L-a);
11 x = linspace(0,L,100);
12 y = -theta*x+R*x.^3/(6*E*I)-F/(6*E*I)*((x>a).*((x-a).^3));
13 plot(x,y*1000)
14 title('Deflection of a Simply Supported Beam')
15 xlabel('x (m)'); ylabel('Deflection (mm)')
16 y = -y;
17 [ymax, index] = max(y);
18 fprintf('Maximum deflection %.2f mm at x = %.2f m\n', ymax*1000, x(index))

```



[4] This program plots a curve representing the deflection of the simply supported beam in [1-2], last page.

```
>> Example02_14
Maximum deflection 12.78 mm at x = 3.56 m
```

[5] Text output of the program.

Logical Operators in Numeric Expressions

[6] The function

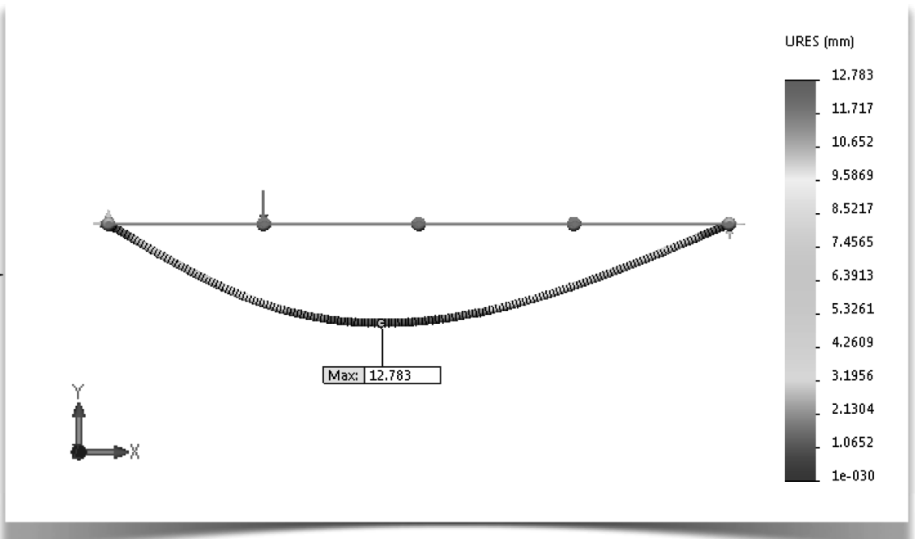
$$\langle x-a \rangle^3 = \begin{cases} 0, & \text{if } x \leq a \\ (x-a)^3, & \text{if } x > a \end{cases}$$

is implemented (see line 12) using a logical operator

$$(x > a) .* ((x - a) .^ 3)$$

Note that, in a numeric expression, `true` is converted to 1 and `false` is converted to 0.

[7] This is the solution output by a program using finite element methods; the maximum deflection (12.783 mm) is consistent with the value in [5]. #



2.15 Example: Vibrations of Supported Machines

Free Vibrations of a Supported Machine

[1] The figure below (source: https://commons.wikimedia.org/wiki/File:Mass_spring_damper.svg, by Pbroks13) represents a machine supported by a layer of elastic, energy-absorbing material. In the figure, m is the mass of the machine; k is the spring constant of the support; i.e., $F_s = -kx$, where F_s is the elastic force acting on the machine and x is the displacement of the machine; c is the damping constant; i.e., $F_d = -c\dot{x}$, where F_d is the damping force acting on the machine and \dot{x} is the velocity of the machine. (See *Wikipedia>Damping*.)

Imagine that you lift the machine upward a distance δ from its static equilibrium position and then release. The machine would vibrate up-and-down. It is called a **free vibration**, since no external forces are involved.

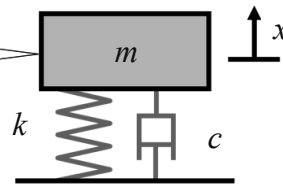
To derive the governing equation, consider that the machine moves with a displacement x and a velocity \dot{x} . The machine is subject to an elastic force $F_s = -kx$ and a damping force $F_d = -c\dot{x}$. Newton's second law states that the resultant force acting on the machine is equal to the multiplication of the mass m and its acceleration \ddot{x} . We have $-kx - c\dot{x} = m\ddot{x}$, or

$$m\ddot{x} + c\dot{x} + kx = 0$$

with the initial conditions (ICs)

$$x(0) = \delta, \quad \dot{x}(0) = 0$$

[2] In this section, we use this mass-spring-damper model to represent a machine supported by a layer of elastic, energy-absorbing material.



Undamped Free Vibrations

[3] First, we neglect the damping effects of the supporting material, i.e., $c = 0$. The equation reduces to

$$m\ddot{x} + kx = 0, \quad \text{ICs: } x(0) = \delta, \quad \dot{x}(0) = 0 \quad (\text{a})$$

The solution for the equation is

$$x(t) = \delta \cos \omega t \quad (\text{b})$$

where ω (with SI unit rad/s) is the natural frequency of the undamped system,

$$\omega = \sqrt{\frac{k}{m}} \quad (\text{c})$$

The solution (b) can be verified by substituting it to the differential equation and the initial conditions (a).

The natural period (with SI unit s) is then

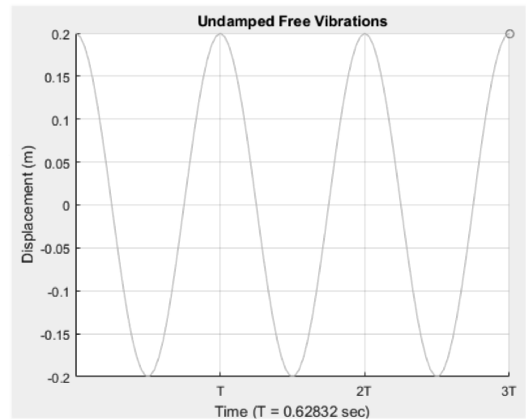
$$T = \frac{2\pi}{\omega} \quad (\text{d})$$

Example02_13a.m ([4], next page) calculates and plots the solution, using the following parameters

$$m = 1 \text{ kg}, \quad k = 100 \text{ N/m}, \quad \delta = 0.2 \text{ m}$$

Note that these values are arbitrarily chosen for instructional purposes; they may not be practical in the real-world. →

[5] This is the output of Example02_15a.m. Without damping effects, the machine vibrates forever; i.e., the amplitudes never fade away.



Example02_15a.m: Undamped Free Vibrations

[4] This program calculates and plots the solution $x(t)$ in Eqs. (b, c), last page. The graphic output is shown in [5].

```

1 clear
2 m = 1; k = 100; delta = 0.2;
3 omega = sqrt(k/m);
4 T = 2*pi/omega;
5 t = linspace(0, 3*T, 100);
6 x = delta*cos(omega*t);
7 axes('XTick', T:T:3*T, 'XTickLabel', {'T', '2T', '3T'});
8 axis([0, 3*T, -0.2, 0.2])
9 grid on
10 hold on
11 comet(t, x)
12 title('Undamped Free Vibrations')
13 xlabel(['Time (T = ', num2str(T), ' sec)'])
14 ylabel('Displacement (m)')
```

Damped Free Vibrations

[6] Now we include the damping effects of the supporting material and assume $c = 1$ N/(m/s). The equation becomes

$$m\ddot{x} + c\dot{x} + kx = 0, \quad \text{ICs: } x(0) = \delta, \quad \dot{x}(0) = 0 \quad (\text{e})$$

There exists a critical damping $c_c = 2m\omega$ such that when $c > c_c$, the machine doesn't oscillate and it is called an **over-damped** case. When $c < c_c$, the machine oscillates and it is called an **under-damped** case. When $c = c_c$, the machine also doesn't oscillate and it is called a **critically-damped** case. In our case,

$$\omega = \sqrt{\frac{k}{m}} = \sqrt{\frac{100 \text{ N/m}}{1 \text{ kg}}} = 10 \text{ rad/s}$$

$$c_c = 2m\omega = 2(1 \text{ kg})(10 \text{ rad/s}) = 20 \text{ N/(m/s)}$$

Since $c < c_c$, the system is an under-damped case. The solution for the under-damped case is

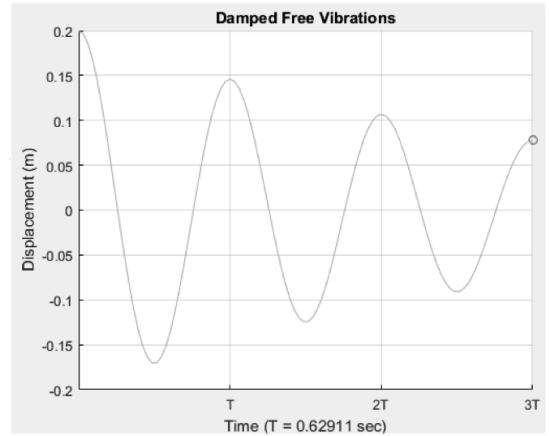
$$x(t) = \delta e^{-\frac{ct}{2m}} \left(\cos \omega_d t + \frac{c}{2m\omega_d} \sin \omega_d t \right) \quad (\text{f})$$

where ω_d (with SI unit rad/s) is the natural frequency of the damped system,

$$\omega_d = \omega \sqrt{1 - \left(\frac{c}{c_c} \right)^2} \quad (\text{g})$$

where c/c_c is called the **damping ratio**. In our case $c/c_c = 0.05$. Note that, for small damping ratios, $\omega_d \approx \omega$. \rightarrow

[8] The output of Example02_15b.m. Due to the inclusion of the damping effects, the vibrations gradually fade away.



Example02_15b.m: Damped Free Vibrations

[7] This program calculates and plots the solution $x(t)$ in Eqs. (f, g), last page. The graphic output is shown in [8].

```

15 clear
16 m = 1; k = 100; c = 1; delta = 0.2;
17 omega = sqrt(k/m);
18 cC = 2*m*omega;
19 omegaD = omega*sqrt(1-(c/cC)^2);
20 T = 2*pi/omegaD;
21 t = linspace(0, 3*T, 100);
22 x = delta*exp(-c*t/(2*m)).*(cos(omegaD*t)+c/(2*m*omegaD)*sin(omegaD*t));
23 axes('XTick', T:T:3*T, 'XTickLabel', {'T', '2T', '3T'});
24 axis([0, 3*T, -0.2, 0.2])
25 grid on
26 hold on
27 comet(t, x)
28 title('Damped Free Vibrations')
29 xlabel(['Time (T = ', num2str(T), ' sec)'])
30 ylabel('Displacement (m)')

```

Harmonically Forced Vibrations

[9] Now, assume there is a rotating part in the machine and, due to eccentric rotations, the rotating part generates an up-and-down harmonic force $F \sin \omega_f t$ on the support, where ω_f is the angular frequency of the rotating part and F is the centrifugal forces due to the eccentric rotations. We assume $F = 2$ N and $\omega_f = 2\pi$ rad/s (i.e., 1 Hz). Adding this force to Newton's second law, we have the equation

$$m\ddot{x} + c\dot{x} + kx = F \sin \omega_f t \quad (\text{h})$$

The solution consists of two parts: (a) The free vibrations caused by the initial conditions. This part will eventually vanish due to the damping effects, as shown in [8], and is called the **transient response**. (b) The vibrations caused by the harmonic forces. This part remains even after the transient vibrations vanish and is called the **steady-state response**, described by

$$x(t) = x_m \sin(\omega_f t - \varphi) \quad (\text{i})$$

where x_m is the amplitude and φ is the **phase angle** (see *Wikipedia>Phase (wave)*) of the vibrations,

$$x_m = \frac{F/k}{\sqrt{\left[1 - \left(\frac{\omega_f}{\omega}\right)^2\right]^2 + \left[2\left(\frac{c}{c_c}\right)\left(\frac{\omega_f}{\omega}\right)\right]^2}} \quad (\text{j})$$

$$\varphi = \tan^{-1} \frac{2\left(\frac{c}{c_c}\right)\left(\frac{\omega_f}{\omega}\right)}{1 - \left(\frac{\omega_f}{\omega}\right)^2} \quad (\text{k}) \rightarrow$$

Example02_15c.m: Forced Vibrations

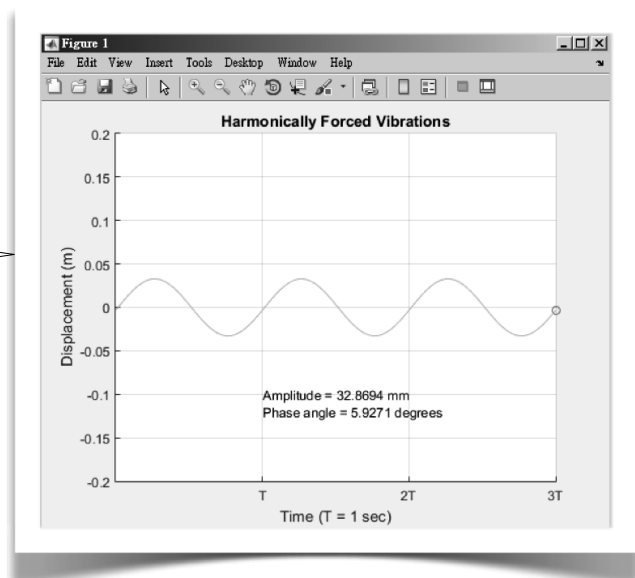
[10] This program plots the steady-state response $x(t)$ in Eqs. (i, j, k), last page. The graphic output is shown in [11].

```

31 clear
32 % System parameters
33 m = 1; k = 100; c = 1;
34 f = 2; omegaF = 2*pi;
35
36 % System response
37 omega = sqrt(k/m);
38 cC = 2*m*omega;
39 rC = c/cC;
40 rW = omegaF/omega;
41 xm = (f/k)/sqrt((1-rW^2)^2+(2*rC*rW)^2);
42 phi = atan((2*rC*rW)/(1-rW^2));
43 T = 2*pi/omegaF;
44 t = linspace(0, 3*T, 100);
45 x = xm*sin(omegaF*t-phi);
46
47 % Graphic output
48 axes('XTick', T:T:3*T, 'XTickLabel', {'T','2T','3T'});
49 axis([0, 3*T, -0.2, 0.2])
50 grid on
51 hold on
52 comet(t, x)
53 title('Harmonically Forced Vibrations')
54 xlabel(['Time (T = ', num2str(T), ' sec)'])
55 ylabel('Displacement (m)')
56 text(T,-0.1,['Amplitude = ', num2str(xm*1000), ' mm'])
57 text(T,-0.12,['Phase angle = ', num2str(phi*180/pi), ' degrees'])

```

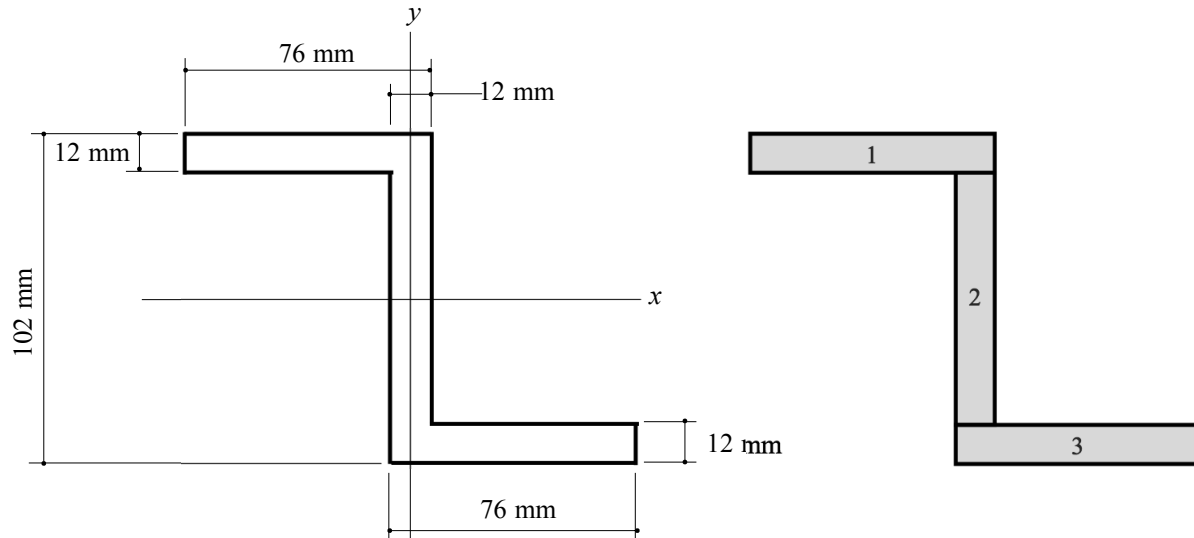
[11] The output of
Example02_15c.m. #



2.16 Additional Exercise Problems

Problem02_01: Moment of Inertia of an Area

Write a script to calculate the moments of inertia (*Wikipedia>Second moment of area*) I_x and I_y of a Z-shape area shown below. Check your results with the following data: $I_x = 4,190,040 \text{ mm}^4$ and $I_y = 2,756,960 \text{ mm}^4$. A hand-calculation procedure is listed in the table below, in which the Z-shape area is divided into three rectangles; their area properties are calculated separately and then totaled.



Rectangle	b mm	h mm	\bar{x} mm	\bar{y} mm	\bar{I}_x mm ⁴	\bar{I}_y mm ⁴	A mm ²	$A\bar{y}^2$ mm ⁴	$A\bar{x}^2$ mm ⁴	I_x mm ⁴	I_y mm ⁴
1	76	12	-32	45	10944	438976	912	1846800	933888	1857744	1372864
2	12	78	0	0	474552	11232	936	0	0	474552	11232
3	76	12	32	-45	10944	438976	912	1846800	933888	1857744	1372864
Total							2760			4190040	2756960

Notes: $\bar{I}_x = bh^3/12$, $\bar{I}_y = hb^3/12$, $A = bh$, $I_x = \bar{I}_x + A\bar{y}^2$, $I_y = \bar{I}_y + A\bar{x}^2$

Problem02_02: Binomial Coefficient

The binomial coefficient (*Wikipedia>Binomial coefficient*) is given by

$$C_x^n = \frac{n!}{x!(n-x)!}$$

where both n and x are integer number and $x \leq n$. Write a script that allows the user to input the values of n and x , calculates C_x^n , and reports the result.

Use the following data to verify your program: $C_3^{10} = 120$, $C_{10}^{15} = 3003$, and $C_4^{100} = 3,921,225$.

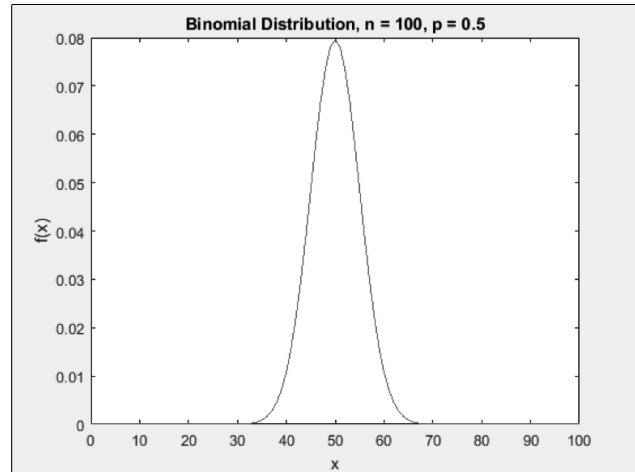
Problem02_03: Binomial Distribution

The binomial distribution (*Wikipedia>Binomial distribution*) is given by

$$f(x) = C_x^n p^x (1-p)^{n-x}, \quad x = 0, 1, 2, \dots, n$$

where p is a real number ($0 < p < 1$) and C_x^n is given in Problem02_02, last page. Write a script that allows the user to input the values of n and p , and produces a binomial distribution curve.

Use $n = 100$ and $p = 0.5$ to verify your program, which produces a binomial distribution curve as shown to the right.



Problem02_04: Thermal Stresses in a Pipe

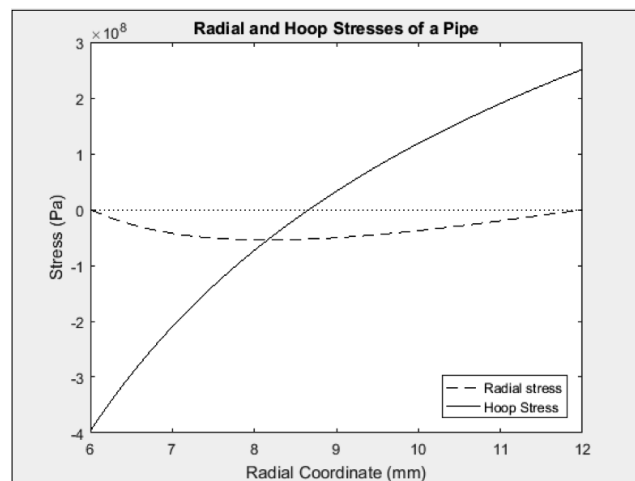
The radial stress σ_r and hoop stress σ_h in a long pipe due to a temperature T_a at its inner surface of radius a and a temperature T_b at its outer surface of radius b are, respectively, (*A. H. Burr and J. B. Cheatham, Mechanical Analysis and Design, 2nd ed., Prentice Hall, p. 496.*)

$$\sigma_r = \frac{\alpha E (T_a - T_b)}{2(1-\nu) \ln(b/a)} \left[\frac{a^2}{b^2 - a^2} \left(\frac{b^2}{r^2} - 1 \right) \ln(b/a) - \ln(b/r) \right]$$

$$\sigma_h = \frac{\alpha E (T_a - T_b)}{2(1-\nu) \ln(b/a)} \left[1 - \frac{a^2}{b^2 - a^2} \left(\frac{b^2}{r^2} + 1 \right) \ln(b/a) - \ln(b/r) \right]$$

where r is the radial coordinate of the pipe (originated at the center), E is the Young's modulus, ν is the Poisson's ratio, and α is the coefficient of thermal expansion.

Write a script that allows the user to input the values of a , b , T_a and T_b , and generates a σ_r -versus- r curve and a σ_h -versus- r curve as shown right, which uses the following data: $a = 6$ mm, $b = 12$ mm, $T_a = 260^\circ\text{C}$, $T_b = 150^\circ\text{C}$, $E = 206$ GPa, $\nu = 0.3$, $\alpha = 2 \times 10^{-5} / ^\circ\text{C}$.

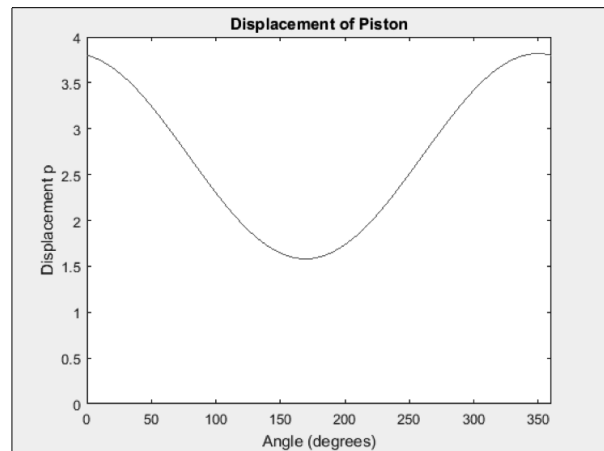


Problem02_05: Displacement of a Piston

The displacement p of the piston shown in 6.3[3-7] (page 261) is given by

$$p = a \cos \theta + \sqrt{b^2 - a^2 \sin^2 \theta}$$

Write a script to plot the displacement p as a function of angle θ (in degrees; $0 \leq \theta \leq 360^\circ$) when $a = 1.1$ and $b = 2.7$.

**Problem02_06: Calculation of π**

The ratio of a circle's circumference to its diameter, π , can be approximated by the following formula:

$$\pi = \sum_{k=0}^n \left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6} \right) \left(\frac{1}{16} \right)^k$$

Write a script that allows the user to input the value of n , and outputs the calculated value of π .

Problem02_07

Write a script to generate a mesh, using `mesh(x, y, z)`, defined by

$$z(x, y) = \frac{32}{3\pi} \sum_{k=0}^{50} \frac{\sin(k\pi/4)}{k^2} \sin(k\pi x) \cos(k\pi y)$$

