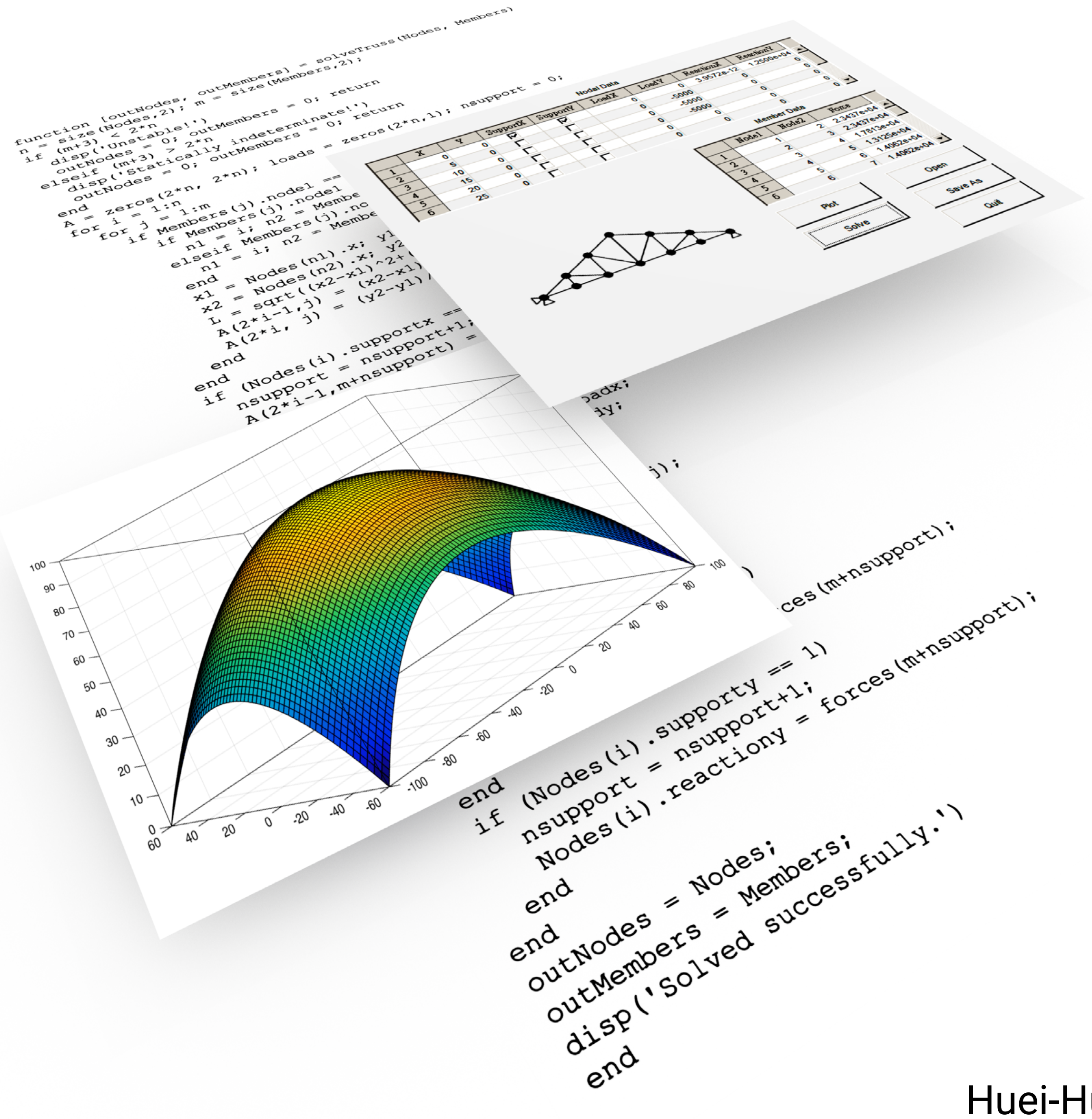
PROGRAMMING AND ENGINEERING COMPUTING WITH MATLAB® 2019



Huei-Huang Lee

SDC PUBLICATIONS

Better Textbooks. Lower Prices.
www.SDCpublications.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

[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
```

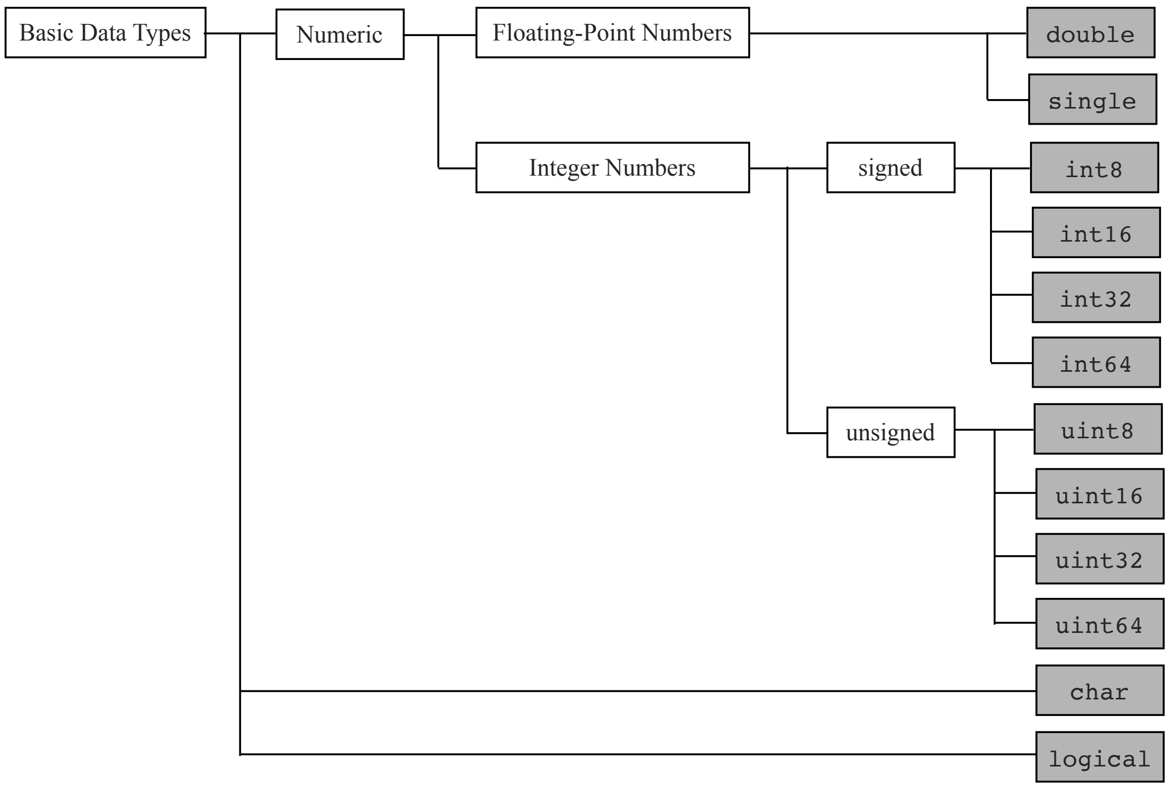
where the number 75 is treated as a `double`; therefore a variable `a` of the type `double` is created to store the number. To create a number of a type other than `double`, you must explicitly use a data type conversion function; e.g.,

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

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

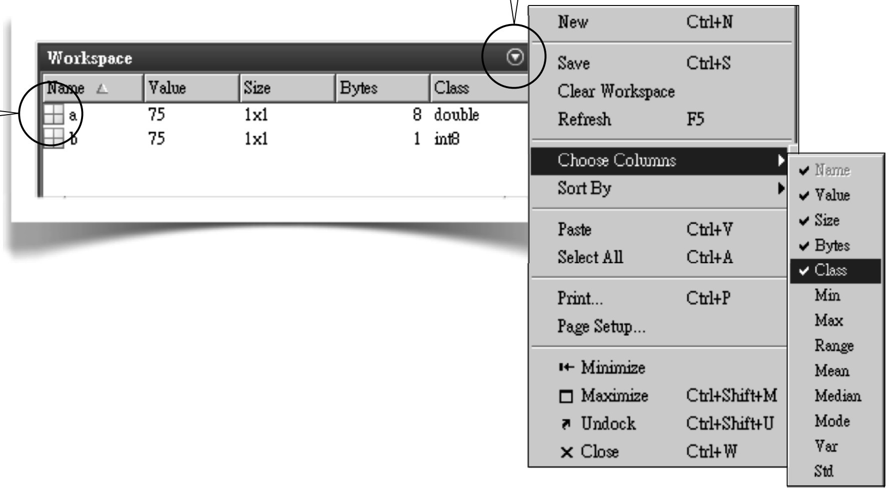
```
>> whos
  Name      Size      Bytes     Class
  a         1x1           8     double
  b         1x1           1     int8
```

This information can be displayed in the **Workspace Window** (see [2-3], next page). Note that 8 bytes of the memory are 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. ↵



[2] Additional information of the variables in the **Workspace** can be displayed by pulling down this button and selecting **Choose Columns**. For the rest of the book, we set up the **Workspace Window** like this. ←

[3] This symbol indicates a numeric type, either a scalar or an array. ↓



Name	Value	Size	Bytes	Class
a	75	1x1	8	double
b	75	1x1	1	int8

Unsigned Integers

[4] Computer representation of unsigned integers is straightforward. For example, the decimal number 75 is represented by a binary number 01001011. A binary number can be converted to a decimal number by treating 1 at the n^{th} digit as 2^{n-1}. Thus, the binary number 01001011 is converted to a decimal number 75 as following:

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

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 $b_{n-1}b_{n-2}\ldots b_1 b_0$ (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

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

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 = dec2bin(u)
5   number = bin2dec(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 = dec2bin(u)
15   bits =
16       '1001011'
17   >> number = bin2dec(bits)
18   number =
19       75
```

About Example02_01.m

[7] In line 2, 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 64-bit **floating-point** representation, to be discussed in Section 2.3, is used for `double` data.

In line 3, 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 type `uint8` is created to store the number.

In line 4, the function `dec2bin` converts the decimal number `u` to a character string `bits` representing binary number. The bit pattern in lines 15-16 is consistent with that in [4], last page. Note that, in line 16, leading zeros are not present. In line 5, the function `bin2dec` converts the character string `bits` back to the decimal number 75 (see lines 18-19). Both the functions `dec2bin` and `bin2dec` convert numbers using the procedure described in [4], last page. #

Table 2.1 Unsigned Integer Numbers

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

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

2.2 Signed Integers

[1] With 8 bits, the positive integer 75 is represented by 01001011. How to represent the negative integer -75? A simple idea is to use a bit as the "sign bit." For example, we might use the leftmost bit as the sign bit: 0 for positive and 1 for negative. Thus, if 01001011 represents +75, then 11001011 represents -75. A problem of this representation is that both 00000000 and 10000000 represent the same number, zero. Another problem is that two numbers with the opposite signs can not be added in a simple way; e.g., 01001011 + 11001011 = 00010100, which is not correct; adding -75 to 75 should result in zero.

One's Complement Representation

Another idea is to use 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 (*Wikipedia> One's Complement*). One of the previous problems still exists with this approach: both 00000000 and 11111111 represent the same number, zero.

Two's Complement Representation

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

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

Minimum/Maximum Values

[2] From Table 2.2a (last page), 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 (last page) 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. ↓

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 = dec2bin(u)
5   t = int8(u)
6   s = typecast(u, 'int8')
7   a = int16(u)
8   bits = dec2bin(a)
```

```
 9   >> clear
10   >> d = 200
11   d =
12      200
13   >> u = uint8(d)
14   u =
15     uint8
16      200
17   >> bits = dec2bin(u)
18   bits =
19       '11001000'
20   >> t = int8(u)
21   t =
22     int8
23      127
24   >> s = typecast(u, 'int8')
25   s =
26     int8
27      -56
28   >> a = int16(u)
29   a =
30     int16
31      200
32   >> bits = dec2bin(a)
33   bits =
34       '11001000'
```

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

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, confirmed in line 4 (also see line 19).

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 "overflown" and only the maximum value (127) is stored in an `int8`. Therefore, the variable `t` has a value of 127 (see lines 21-23).

In line 6, 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 lines 25-27), 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.

To store the value 200 in a signed integer, we need at least an `int16`. Line 7 successfully converts the value 200 to an `int16` and stores it in the variable `a` (also see lines 29-31); line 8 confirms that the bit pattern of `u` is preserved in `a` (also see lines 33-34). #

2.3 Floating-Point Numbers

2.3a 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 (next page) 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

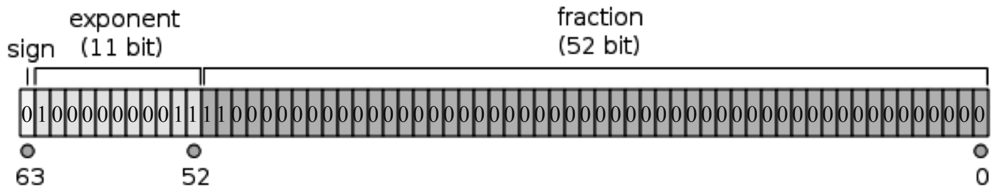
The figure below (*source: https://en.wikipedia.org/wiki/File:IEEE_754_Double_Floating_Point_Format.svg, by Codekaizen*) shows an example bit pattern of a double-precision floating-point number. It uses 64 bits in computer memory: 1 bit (the 63rd bit) for the sign, 11 bits (the 52nd-62nd bits) for the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

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

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

$$(258.369)_{10} = 2\times10^2 + 5\times10^1 + 8\times10^0 + 3\times10^{-1} + 6\times10^{-2} + 9\times10^{-3}$$

Similarly, a binary number 1101.01101 can be interpreted as

$$\begin{aligned}(1101.01101)_2 &= 1\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0 + 0\times2^{-1} + 1\times2^{-2} + 1\times2^{-3} + 0\times2^{-4} + 1\times2^{-5}\\ &= 8+4+0+1+0+0.25+0.125+0+0.03125\\ &= (13.40625)_{10}\end{aligned}$$

↵

Example02_03a.m: Floating-Point Numbers

[3] These statements confirm that the decimal number 28 is indeed represented by the bit pattern in [1], last page. A **Command Window** session is shown in [4].

```
1   clear
2   d = 28
3   a = typecast(d, 'uint64')
4   b = dec2bin(a, 64)
```

Line 2 creates a double-precision floating-point number 28 and stores it in the variable `d`. The bit pattern should be like the one in [1], last page.

In line 3, the function `typecast` preserves the 64-bit pattern while changing its type to `uint64`. Now, the bit pattern is interpreted as a value of 4628574517030027264 (see line 12), which can be calculated by

$$2^{62} + 2^{53} + 2^{52} + 2^{51} + 2^{50} = 4628574517030027264$$

Line 4 demonstrates another way (than using `bitget` and `fliplr`) to display the bit pattern. The function `dec2bin(a,64)` retrieves the bit pattern from an integer number `a` and outputs the bit pattern in a text form (i.e., a string, to be introduced in the next section). The result is shown in line 15, the same as the one in [1], last page. ↓

```
 5   >> clear
 6   >> d = 28
 7   d =
 8       28
 9   >> a = typecast(d, 'uint64')
10   a =
11     uint64
12      4628574517030027264
13   >> b = dec2bin(a, 64)
14   b =
15       '0100000000111100000000000000000000000000000000000000000000000000'
16   >>
```

[4] This is a **Command Window** session of Example02_03a.m. #

Table 2.3a Floating-Point Numbers				
Conversion Function	Function to find the minimum value	Minimum value	Function to find the maximum value	Maximum value
`double`	**`realmin('double')`**	2.2251e-308	**`realmax('double')`**	1.7977e+308
`single`	**`realmin('single')`**	1.1755e-38	**`realmax('single')`**	3.4028e+38
Details and More: Help>MATLAB>Language Fundamentals>Data Types>Numeric Types				

2.3b Significant Digits of Floating-Point Numbers

Example02_03b.m: Siginificant Digits of Floating-Point Numbers

[1] These statements explore the number of significant digits of floating-point numbers. A **Command Window** session is shown in [2], next page. →

```
1   clear
2   format short
3   format compact
4   a = 1234.5678901234567890 1234
5   fprintf('%.20f\n', a)
6   format long
7   a
8   b = single(a)
9   fprintf('%.20f\n', b)
```

```
10  >> clear
11  >> format short
12  >> format compact
13  >> a = 1234.567890123456789 01234
14  a =
15     1.2346e+03
16  >> fprintf('%.20f\n', a)
17  1234.56789012345689116046
18  >> format long
19  >> a
20  a =
21       1.234567890123457e+03
22  >> b = single(a)
23  b =
24    single
25     1.2345679e+03
26  >> fprintf('%.20f\n', b)
27  1234.56787109375000000000
```

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

Screen Output Format

[3] Lines 2, 3, and 6 set **Command Window** output display format. The syntax is

`format` *`style`*

The `short` (line 2) sets the display of fixed-decimal format 4 digits after the decimal point, the `long` (line 6) 15 digits after the decimal point. The `compact` (line 3) suppresses blank lines to make the output lines compact (also see 1.2[9-10], page 14). 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 (next page) lists 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 4, we assign a number of 24 significant figures to the variable `a` of `double`. We will see that, due to the limited storage space (64 bits), not all the figures can be stored in the variable `a`. The number is displayed (line 15) 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 5, we attempt to print the number with 20 digits after the decimal point. The result (line 17) shows that only the first 16 digits are the same as what was assigned to the variable `a`. The extra digits are lost due to the limited storage space. We conclude that *a double-precision floating point number has 16 significant digits*.

With long format (line 6), the number is displayed with 15 digits after the decimal point (lines 7, 21). Note that, again, in displaying the number, it has been rounded to the last digit. ↵

Single-Precision Floating-Point Numbers

[4] Line 8 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 25) shows that it reduces to 8 significant digits, due to the shorter storage space. The extra digits are discarded during the conversion. This is also confirmed with line 9; its output in line 27 is accurate up to the 8th digits. We conclude that *a single-precision floating point number has 8 significant digits*. #

Table 2.3b Numeric Output Format

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

Details and More: >> doc format

2.4 Characters and Strings

2.4a 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 2.4b, page 80). An ASCII code is a number representing a character, either printable or non-printable. The ASCII codes of the characters `'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], next page. →

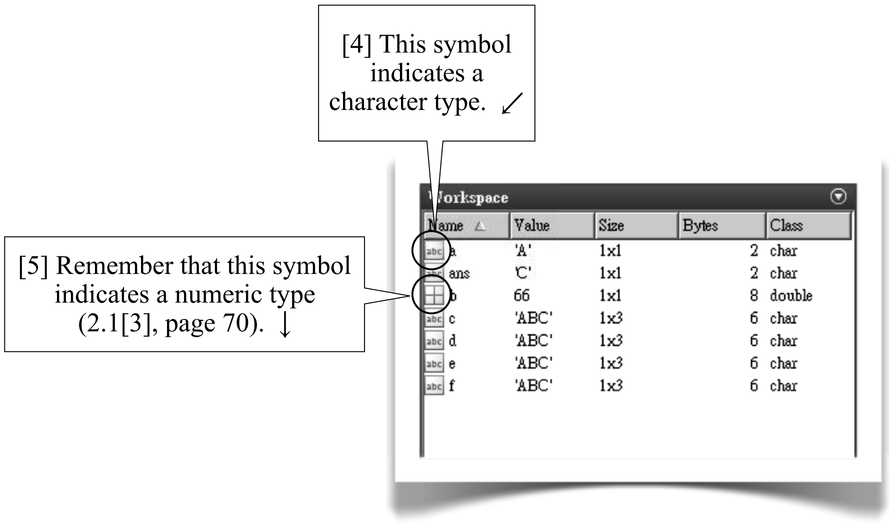
```
clear
a = 'A'
b = a + 1
char(65)
char('A' + 2)
c = ['A', 'B', 'C']
d = ['AB', 'C']
e = ['A', 66, 67]
f = 'ABC'
f(1)
f(2)
f(3)
```

```
>> clear
>> a = 'A'
a =
    'A'
>> b = a + 1
b =
    66
>> char(65)
ans =
    'A'
>> char('A' + 2)
ans =
    'C'
>> c = ['A', 'B', 'C']
c =
    'ABC'
>> d = ['AB', 'C']
d =
    'ABC'
>> e = ['A', 66, 67]
e =
    'ABC'
>> f = 'ABC'
f =
    'ABC'
>> f(1)
ans =
    'A'
>> f(2)
ans =
    'B'
>> f(3)
ans =
    'C'
```

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

[4] This symbol indicates a character type. ↙

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



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

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`; both are strings of three characters `'ABC'`.

In line 8, since an array must have elements of the same data type and since there is a character in the array, 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`, confirmed in lines 10-12.

The Variable `ans`

In lines 4-5 and 10-12, there are no variables to store the result of the expressions. Whenever there is no variable to store the resulting value of an expression, MATLAB always uses the variable `ans` (short for **answer**) to store that value (also see lines 21, 24, 39, 42, 45). #

2.4b ASCII Codes

Example02_04b.m: ASCII Codes

[1] MATLAB stores characters according to ASCII Code. ASCII codes 32-126 represent printable characters on a standard keyboard. This example prints a table of characters corresponding to the ASCII Codes 32-126 (see the output in [2]). →

```
 1  clear
 2  fprintf('    0 1 2 3 4 5 6 7 8 9\n')
 3  for row = 3:12
 4      fprintf('%2d  ', row)
 5      for column = 0:9
 6          code = row*10+column;
 7          if (code < 32) || (code > 126)
 8             fprintf('  ')
 9          else
10             fprintf('%c ', code)
11          end
12      end
13      fprintf('\n')
14  end
```

```
    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 { | } ~
```

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

About Example02_4b.m

[3] Line 2 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 3-14) 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 4), the row number is printed. Then 10 characters are printed using an inner `for`-loop (lines 5-12). At the end of the outer for-loop, a newline character is printed (line 13), moving the cursor to the next line.

Each pass of the inner `for`-loop (lines 5-12) prints a character aligning with the column number. Line 6 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 7) then two spaces are printed (line 8), otherwise the ASCII code is printed as a character followed by a space (line 10).

The expression `(code < 32) || (code > 126)` in the `if`-statement (line 7) is a logical expression. We'll introduce logical data in Section 2.5 and logical expressions in Section 2.9.

In line 10, 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 one 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 zero becomes `false`. ↓

Example02_05.m: Logical Data Type

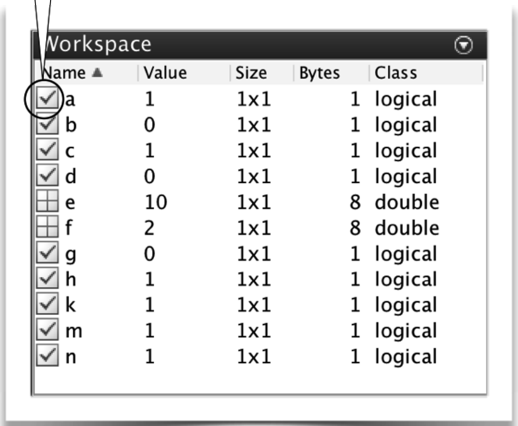
[2] These statements demonstrate some concepts about **logical** data. A **Command Window** session and the **Workspace** is shown in [3-4], respectively. ↘

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

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

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

[4] This symbol indicates a logical type. ↵



Name ▲	Value	Size	Bytes	Class
✓ a	1	1x1	1	logical
✓ b	0	1x1	1	logical
✓ c	1	1x1	1	logical
✓ d	0	1x1	1	logical
⊞ e	10	1x1	8	double
⊞ f	2	1x1	8	double
✓ g	0	1x1	1	logical
✓ h	1	1x1	1	logical
✓ k	1	1x1	1	logical
✓ m	1	1x1	1	logical
✓ n	1	1x1	1	logical

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 a 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 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 zero 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 the letter `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

2.6a Arrays Creations

All Data Are Treated as 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 four or more dimensional arrays; in practice, however, they are 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)
```

```
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. ←

Scalar

[4] Lines 2-4 show three ways to create the same scalar. Line 2 creates a single value 5. Line 3 creates a vector of one element, i.e., a scalar. Line 4 creates a 1x1 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` (lines 4-5) creates an array of all ones with specified dimension sizes. 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. ↵

Row Vectors

[5] Lines 6-11 show many ways to create the same row vector. Line 6 creates a row vector using the square brackets (`[ ]`). Commas are used to separate elements in a row. The commas can be omitted (line 7).

Line 8 creates a row vector using the colon notation (`:`). The square brackets can be omitted (line 9). In a more general form, an increment number can be inserted between the starting number and the ending number (line 10; also see 1.3[6], page 18). The function `linspace` (line 11) creates a row vector of linearly spaced numbers. The syntax is

`linspace(`*`start`*`, `*`end`*`, `*`n`*`)`

where *n* is the total number of elements. If *n* is omitted, its default is 100.

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

2.6b Arrays Indexing

Example02_06b.m

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

```
 1   clear
 2   a = zeros(1,5)
 3   a(1,5) = 8
 4   a(5) = 9
 5   a([1, 2, 4]) = [8, 7, 6]
 6   a(1:4) = [2, 3, 4, 5]
 7   [rows, cols] = size(a)
 8   len = length(a)
 9   b = a
10   c = a(1:5)
11   d = a(3:5)
12   e = a(3:length(a))
13   f = a(3:end)
14   f(5) = 10
```

[3] Line 2 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 3 uses subscript indexing, while line 4 uses linear indexing. For a vector (row vector or column vector), we usually use linear indexing.

Line 5 assigns three values to the 1st, 2nd, and 4th elements of the array `a`; i.e., line 5 is equivalent to

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

Line 6, since `1:4` means `[1,2,3,4]`, assigns four values to the 1st-4th elements of the array `a`. ↵

```
15   >> clear
16   >> a = zeros(1,5)
17   a =
18        0     0     0     0     0
19   >> a(1,5) = 8
20   a =
21        0     0     0     0     8
22   >> a(5) = 9
23   a =
14        0     0     0     0     9
25   >> a([1, 2, 4]) = [8, 7, 6]
26   a =
27        8     7     0     6     9
28   >> a(1:4) = [2, 3, 4, 5]
29   a =
30        2     3     4     5     9
31   >> [rows, cols] = size(a)
32   rows =
33        1
34   cols =
35        5
36   >> len = length(a)
37   len =
38        5
39   >> b = a
40   b =
41        2     3     4     5     9
42   >> c = a(1:5)
43   c =
44        2     3     4     5     9
45   >> d = a(3:5)
46   d =
47        4     5     9
48   >> e = a(3:length(a))
49   e =
50        4     5     9
51   >> f = a(3:end)
52   f =
53        4     5     9
54   >> f(5) = 10
55   f =
56      4     5     9     0    10
```

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

Size and Length of an Array

[4] The function `size` (line 7) outputs dimensional sizes of an array. In line 7, `size(a)` outputs two values: number of rows and number of columns; and a two-element vector is used to receive the output values.

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

`length(a) ≡ max(size(a))`

In this case, the length of the array `a` is 5 (line 8; also see lines 36-38, last page). ↓

[5] Line 9 assigns the entire array `a` to a variable `b`, which becomes the same sizes and contents as the array `a`. Line 10 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 11-13 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 the same in sizes and contents. Note that, in line 13, the keyword `end` means the **last index** of the underlying array.

Line 14 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. #

2.6c Colon: Entire Column/Row

Example02_06c.m

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

```
 1  clear
 2  a = [1, 2; 3, 4; 5, 6]
 3  b = 1:6
 4  c = reshape(b, 3, 2)
 5  d = reshape(b, 2, 3)
 6  e = d'
 7  c(:,3) = [7, 8, 9]
 8  c(4,:) = [10, 11, 12]
 9  c(4,:) = []
10  c(:,2:3) = []
```

```
11  >> clear
12  >> a = [1, 2; 3, 4; 5, 6]
13  a =
14       1     2
15       3     4
16       5     6
17  >> b = 1:6
18  b =
19       1     2     3     4     5     6
20  >> c = reshape(b, 3, 2)
21  c =
22       1     4
23       2     5
24       3     6
25  >> d = reshape(b, 2, 3)
26  d =
27       1     3     5
28       2     4     6
29  >> e = d'
30  e =
31       1     2
32       3     4
33       5     6
34  >> c(:,3) = [7, 8, 9]
35  c =
36       1     4     7
37       2     5     8
38       3     6     9
39  >> c(4,:) = [10, 11, 12]
40  c =
41       1     4     7
42       2     5     8
43       3     6     9
44      10    11    12
45  >> c(4,:) = []
46  c =
47       1     4     7
48       2     5     8
49       3     6     9
50  >> c(:,2:3) = []
51  c =
52       1
53       2
54       3
```

[2] This is a **Command Window** session of Example02_06c.m. ↲

Function `reshape`

[3] Line 2 creates a 3-by-2 matrix. Line 3 creates a row vector of 6 elements. Line 4 reshapes the vector `b` into a 3-by-2 matrix. The reshaping doesn't alter the order of the elements stored in the array (see 1.9[11-16], page 36); it alters dimensionality and dimension sizes. Note that `c` is different from `a` (see lines 14-16 and 22-24). To obtain a matrix the same as `a` from the vector `b`, we reshape `b` into a 2-by-3 matrix (line 5) and then transpose it (line 6). Now `e` is the same as `a` (lines 31-33).

Colon: The Entire Column/Row

Line 7 assigns 3 elements to the third column of `c`. Note that `[7, 8, 9]` is automatically transposed, becoming a column. Line 8 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 9 sets the fourth row of `c` to be empty, i.e., deleting the entire row. Line 10 sets the 2nd-3rd columns to be empty, i.e., deleting the 2nd-3rd columns.

The `[]` represents an empty data. #

2.6d Concatenation and Flipping

Example02_06d.m

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

```
 1  clear
 2  a = reshape(1:6, 3, 2)
 3  b = [7; 8; 9]
 4  c = horzcat(a, b)
 5  d = [a, b]
 6  e = b'
 7  f = vertcat(d, e)
 8  g = [d; e]
 9  h = fliplr(c)
10  k = flipud(c)
```

```
11  >> clear
12  >> a = reshape(1:6, 3, 2)
13  a =
14       1     4
15       2     5
16       3     6
17  >> b = [7; 8; 9]
18  b =
19       7
20       8
21       9
22  >> c = horzcat(a, b)
23  c =
24       1     4     7
25       2     5     8
26       3     6     9
27  >> d = [a, b]
28  d =
29       1     4     7
30       2     5     8
31       3     6     9
32  >> e = b'
33  e =
34       7     8     9
35  >> f = vertcat(d, e)
36  f =
37       1     4     7
38       2     5     8
39       3     6     9
40       7     8     9
41  >> g = [d; e]
42  g =
43       1     4     7
44       2     5     8
45       3     6     9
46       7     8     9
47  >> h = fliplr(c)
48  h =
49       7     4     1
50       8     5     2
51       9     6     3
52  >> k = flipud(c)
53  k =
54       3     6     9
55       2     5     8
56       1     4     7
```

[2] This is a **Command Window** session of Example02_06d.m. ↵

Concatenation of Arrays

[3] Line 2 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 3 creates a column vector `b` of 3 elements.

Using function `horzcat`, line 4 concatenates `a` and `b` horizontally to create a 3-by-3 matrix `c`. Line 5 demonstrates a more convenient way to do the same job, using a comma (`,`) to concatenate arrays horizontally.

Line 6 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 7 concatenates `d` and `e` vertically to create a 4-by-3 matrix `f`. Line 8 demonstrates a more convenient way to do the same job, using a semicolon (`;`) to concatenate arrays vertically.

Flipping Matrices

Using the function `fliplr` (flip left-side right), line 9 flips the matrix `c` horizontally. Using the function `flipud` (flip upside down), line 10 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
`zeros(n,m)`	Create an n-by-m matrix of all zeros
`ones(n,m)`	Create an n-by-m matrix of all ones
`eye(n)`	Create an n-by-n identity matrix
`diag(v)`	Create a square diagonal matrix with v on the diagonal
`rand(n,m)`	Create an n-by-m matrix of uniformly distributed random numbers in the interval (0,1)
`randn(n,m)`	Create an n-by-m matrix of random numbers from the standard normal distribution
`linspace(a,b,n)`	Create a row vector of n linearly spaced numbers from a to b
`[X,Y] = meshgrid(x,y)`	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
`repmat(a,n,m)`	Replicate array a n times in row-dimension and m times in column-dimension
`horzcat(a,b,...)`	Concatenate arrays horizontally
`vertcat(a,b,...)`	Concatenate arrays vertically
`flipud(A)`	Flip an array upside down
`fliplr(A)`	Flip an array left-side right
`reshape(A,n,m)`	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]). ↘

```
clear
a = 1:5
b = sum(a)
c = cumsum(a)
d = prod(a)
e = cumprod(a)
f = diff(a)
A = reshape(1:9, 3, 3)
g = sum(A)
B = cumsum(A)
h = prod(A)
C = cumprod(A)
D = diff(A)
p = min(a)
q = max(a)
r = min(A)
s = max(A)
```

```
>> clear
>> a = 1:5
a =
     1     2     3     4     5
>> b = sum(a)
b =
    15
>> c = cumsum(a)
c =
     1     3     6    10    15
>> d = prod(a)
d =
   120
>> e = cumprod(a)
e =
     1     2     6    24   120
>> f = diff(a)
f =
     1     1     1     1
>> A = reshape(1:9, 3, 3)
A =
     1     4     7
     2     5     8
     3     6     9
>> g = sum(A)
g =
     6    15    24
>> B = cumsum(A)
B =
     1     4     7
     3     9    15
     6    15    24
>> h = prod(A)
h =
     6   120   504
>> C = cumprod(A)
C =
     1     4     7
     2    20    56
     6   120   504
>> D = diff(A)
D =
     1     1     1
     1     1     1
```

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

Table 2.7
Sums, Products, Minima, and Maxima

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

```
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 **a** (either row vector or column vector). When applied to a vector, the function `sum` (line 3) outputs 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 function `cumsum` (cumulative sum; line 4) outputs a vector **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),

$$\begin{aligned} c_1 &= a_1 = 1 \\ c_2 &= c_1 + 2 = 3 \\ c_3 &= c_2 + 3 = 6 \\ c_4 &= c_3 + 4 = 10 \\ c_5 &= c_4 + 5 = 15 \end{aligned}$$

When applied to a vector, function `prod` (line 5) outputs 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 function `cumprod` (cumulative product; line 6) outputs a vector **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),

$$\begin{aligned} e_1 &= a_1 = 1 \\ e_2 &= e_1 \times 2 = 2 \\ e_3 &= e_2 \times 3 = 6 \\ e_4 &= e_3 \times 4 = 24 \\ e_5 &= e_4 \times 5 = 120 \end{aligned}$$

When applied to a vector, the function `diff` (line 7) outputs a vector **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),

$$\begin{aligned} f_1 &= 2 - 1 = 1 \\ f_2 &= 3 - 2 = 1 \\ f_3 &= 4 - 3 = 1 \\ f_4 &= 5 - 4 = 1 \end{aligned}$$

↵

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 **A**. When applied to a matrix, the function `sum` (line 9) outputs a row vector **g**, where g_j is the sum of the jth column of the matrix **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 function `cumsum` (line 10) outputs an $n \times m$ matrix **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),

$$\begin{array}{lll} B_{11} = A_{11} = 1 & B_{12} = A_{12} = 4 & B_{13} = A_{13} = 7 \\ B_{21} = B_{11} + 2 = 3 & B_{22} = B_{12} + 5 = 9 & B_{23} = B_{13} + 8 = 15 \\ B_{31} = B_{21} + 3 = 6 & B_{32} = B_{22} + 6 = 15 & B_{33} = B_{23} + 9 = 24 \end{array}$$

When applied to a matrix, the function `prod` (line 11) outputs a row vector **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 function `cumprod` (line 12) outputs an $n \times m$ matrix **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),

$$\begin{array}{lll} C_{11} = A_{11} = 1 & C_{12} = A_{12} = 4 & C_{13} = A_{13} = 7 \\ C_{21} = C_{11} \times 2 = 2 & C_{22} = C_{12} \times 5 = 20 & C_{23} = C_{13} \times 8 = 56 \\ C_{31} = C_{21} \times 3 = 6 & C_{32} = C_{22} \times 6 = 120 & C_{33} = C_{23} \times 9 = 504 \end{array}$$

When applied to a matrix, the function `diff` (line 13) outputs an $(n-1) \times m$ (not $n \times m$) matrix **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),

$$\begin{array}{lll} D_{11} = 2 - 1 = 1 & D_{12} = 5 - 4 = 1 & D_{13} = 8 - 7 = 1 \\ D_{21} = 3 - 2 = 1 & D_{22} = 6 - 5 = 1 & D_{23} = 9 - 8 = 1 \end{array}$$

Minima and Maxima

The output of the functions `min` or `max` for a vector are scalars (lines 14-15, 64, 67).

The output of the functions `min` or `max` for an $n \times m$ matrix is a row vector of m elements (lines 16-17, 70, 73), in which each element is the minimum/maximum of the corresponding column. #

2.8 Arithmetic Operators

2.8a Matrix Operations

[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 in page 99); *the lower number has higher precedence*. 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). ↵

```
clear
A = reshape(1:6, 2, 3)
B = reshape(7:12, 2, 3)
C = A+B
D = A-B
E = B'
F = A*E
a = [3, 6]
b = a/F
c = b*F
G = F^2
H = A.*B
K = A./B
M = A.^2
P = A+10
Q = A-10
R = A*1.5
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

```
>> clear
>> A = reshape(1:6, 2, 3)
A =
     1     3     5
     2     4     6
>> B = reshape(7:12, 2, 3)
B =
     7     9    11
     8    10    12
>> C = A+B
C =
     8    12    16
    10    14    18
>> D = A-B
D =
    -6    -6    -6
    -6    -6    -6
>> E = B'
E =
     7     8
     9    10
    11    12
>> F = A*E
F =
    89    98
   116   128
>> a = [3, 6]
a =
     3     6
>> b = a/F
b =
  -13.0000   10.0000
>> c = b*F
c =
     3     6
>> G = F^2
G =
       19289       21266
       25172       27752
>> H = A.*B
H =
     7    27    55
    16    40    72
>> K = A./B
K =
    0.1429    0.3333    0.4545
    0.2500    0.4000    0.5000
>> M = A.^2
M =
     1     9    25
     4    16    36
```

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

```
>> P = A+10
P =
    11    13    15
    12    14    16
>> Q = A-10
Q =
    -9    -7    -5
    -8    -6    -4
>> R = A*1.5
R =
    1.5000    4.5000    7.5000
    3.0000    6.0000    9.0000
>> S = A/2
S =
    0.5000    1.5000    2.5000
    1.0000    2.0000    3.0000
```

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

Addition of Matrices

[5] Let A_{ij}, $i = 1, 2, \ldots, n; j = 1, 2, \ldots, m$ be the elements of an $n \times m$ matrix **A**, and B_{ij}, $i = 1, 2, \ldots, n; j = 1, 2, \ldots, m$ be the elements of another $n \times m$ matrix **B**. The addition (line 4) of the two matrices is an $n \times m$ matrix **C**,

$$C_{ij} = A_{ij} + B_{ij}$$
$$i = 1, 2, \ldots, n; j = 1, 2, \ldots, m$$

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 5) of **B** from **A** is an $n \times m$ matrix **D**,

$$D_{ij} = A_{ij} - B_{ij}$$
$$i = 1, 2, \ldots, n; j = 1, 2, \ldots, m$$

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 6; also see 1.6[12], page 27) of **B** is an $m \times n$ matrix **E**,

$$E_{ij} = B_{ji}$$
$$i = 1, 2, \ldots, m; j = 1, 2, \ldots, n$$

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 7) of an $n \times m$ matrix **A** by an $m \times p$ matrix **E** is an $n \times p$ matrix **F**

$$F_{ij} = \sum_{k=1}^{m} A_{ik} \times E_{kj}$$
$$i = 1, 2, \ldots, n; j = 1, 2, \ldots, p$$

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 9) of an $r \times m$ matrix **a** by an $m \times m$ matrix **F** (i.e., **a**/**F**) is an $r \times m$ matrix **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 **b**,

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

since it satisfies (see lines 10 and 53)

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

Note that **a** and **F** must have the same **column size** and, in the above example, **F** is a square matrix and the resulting matrix **b** has the same dimension sizes as **a**. In general, **F** is not necessarily a square matrix. If **F** is not a square matrix, then **a**/**F** will output a least-squares solution **b** of the system of equations $\mathbf{b} \times \mathbf{F} = \mathbf{a}$ (see line 8 in 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 11)

$$\mathbf{F}\,\hat{}\,2 \equiv \mathbf{F} \times \mathbf{F}$$

In this example (lines 56-57)

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

Element-Wise Multiplication of Matrices

The element-wise multiplication (`.*` in line 12) operates on two $n \times m$ matrices of the same sizes **A** and **B**, and the result is a matrix **H** of the same size,

$$H_{ij} = A_{ij} \times B_{ij}$$
$$i = 1,\ 2,\ \ldots,\ n;\ j = 1,\ 2,\ \ldots,\ m$$

In this example (lines 60-61)

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

Element-Wise Division of Matrices

The element-wise division (`./` in line 13) also operates on two $n \times m$ matrices of the same sizes **A** and **B**, and the result is a matrix **K** of the same size,

$$K_{ij} = A_{ij} / B_{ij}$$
$$i = 1,\ 2,\ \ldots,\ n;\ j = 1,\ 2,\ \ldots,\ m$$

In this example (lines 64-65)

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

↵

Element-Wise Exponentiation of Matrices

[7] The element-wise exponentiation (line 14) operates on an $n \times m$ matrix **A** and a scalar q, and the result is an $n \times m$ matrix **M**,

$$M_{ij} = \left(A_{ij}\right)^q$$
$$i = 1,\ 2,\ \ldots,\ n;\ j = 1,\ 2,\ \ldots,\ m$$

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 `+`, `-`, `*`, `/`, `.*`, `./`, or `.^`, and s be a scalar, then **A**@s is an $n \times m$ matrix **V**, where

$$V_{ij} = A_{ij} @ s$$
$$i = 1,\ 2,\ \ldots,\ n;\ j = 1,\ 2,\ \ldots,\ m$$

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

$$W_{ij} = s @ A_{ij}$$
$$i = 1,\ 2,\ \ldots,\ n;\ j = 1,\ 2,\ \ldots,\ m$$

For example (lines 15-18)

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

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

$$\mathbf{A} \times 1.5 = \begin{bmatrix} 1\times1.5 & 3\times1.5 & 5\times1.5 \\ 2\times1.5 & 4\times1.5 & 6\times1.5 \end{bmatrix}$$

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

In other words, an operation **A**@s or s@**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 **A** and each element has the same value s. For example

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

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

#

2.8b Vector Operations

Example02_08b.m

[1] These statements demonstrate some arithmetic operations on **vectors** (also see [2]). Remember that a vector is a special case of matrices. Thus operations on vectors are special cases of those on matrices. ↓

```
 1   clear
 2   a = 1:4
 3   b = 5:8
 4   c = a+b
 5   d = a-b
 6   e = a*(b')
 7   f = (a')*b
 8   g = a/b
 9   h = a.*b
10   k = a./b
11   m = a.^2
```

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

```
12   >> clear
13   >> a = 1:4
14   a =
15        1     2     3     4
16   >> b = 5:8
17   b =
18        5     6     7     8
19   >> c = a+b
20   c =
21        6     8    10    12
22   >> d = a-b
23   d =
24       -4    -4    -4    -4
25   >> e = a*(b')
26   e =
27       70
28   >> f = (a')*b
29   f =
30        5     6     7     8
31       10    12    14    16
32       15    18    21    24
33       20    24    28    32
34   >> g = a/b
35   g =
36       0.4023
37   >> h = a.*b
38   h =
39        5    12    21    32
40   >> k = a./b
41   k =
42       0.2000    0.3333    0.4286    0.5000
43   >> m = a.^2
44   m =
45        1     4     9    16
```

Arithmetic Operators for Vectors

[3] 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.

Division (/) by a Non-Square Matrix

In line 8, since `b` is not a square matrix, `g` is not an exact solution of `g*b = a`. Instead, `g` is the least-squares solution of the equation `g*b = 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:

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

and the least-squares solution is $g = 0.4023$ (line 36). #

2.8c Scalar Operations

Example02_08c.m

[1] These statements demonstrate some arithmetic operations on **scalars** (also see [2]). Remember that a scalar is a special case of matrices. Thus operations on scalars are special cases of those on matrices. ↓

```
clear
a = 6
b = 4
c = a+b
d = a-b
e = a*b
f = a/b
g = a^2
h = a.*b
k = a./b
m = a.^2
```

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

```
>> clear
>> a = 6
a =
     6
>> b = 4
b =
     4
>> c = a+b
c =
    10
>> d = a-b
d =
     2
>> e = a*b
e =
    24
>> f = a/b
f =
    1.5000
>> g = a^2
g =
    36
>> h = a.*b
h =
    24
>> k = a./b
k =
    1.5000
>> m = a.^2
m =
    36
```

Arithmetic Operators for Scalars

[3] A scalar is a 1x1 matrix. Thus, all the rules of the arithmetic operations for matrices can apply to those for scalars.

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 in a single logical value). However, when one of the operands is a scalar, e.g., **A** > *s*, where *s* is a scalar and **A** is a matrix, the scalar is expanded such that it has the same size as the matrix **A** and each element has the same value *s* (also see 2.8a[7], page 96).

Table 2.9a Relational Operators

Operator	Description	Precedence level
`==`	Equal to	8
`~=`	Not equal to	8
`>`	Greater than	8
`<`	Less than	8
`>=`	Greater than or equal to	8
`<=`	Less than or equal to	8
`isequal`	Determine array equality	

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

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 is converted to a `false`. ↓

Table 2.9b Logical Operators

Operator	Description	Precedence level
`&`	Logical AND	9
`|`	Logical OR	10
`~`	Logical NOT	4
`&&`	Logical AND (short-circuit)	11
`||`	Logical OR (short-circuit)	12

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

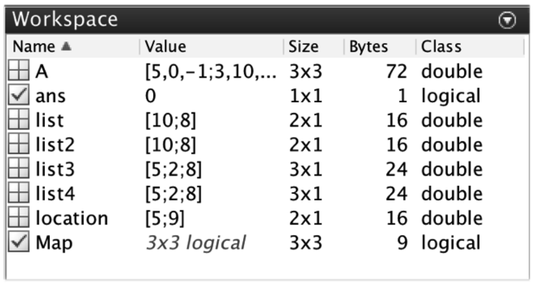
Example02_09.m

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

```
clear
A = [5,0,-1; 3,10,2; 0,-4,8]
Map = (A > 6)
location = find(Map)
list = A(location)
list2 = A(find(A>6))
list3 = A(find(A>0 & A<=8 & A~=3))
list4 = A(A>0 & A<=8 & A~=3)
~A
~~A
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   >> list4 = A(A>0 & A<=8 & A~=3)
42   list4 =
43        5
44        2
45        8
46   >> ~A
47   ans =
48    3×3 logical array
49      0   1   0
50      0   0   0
51      1   0   0
52   >> ~~A
53   ans =
54     3×3 logical array
55      1   0   1
56      1   1   1
57      0   1   1
58   >> isequal(A, ~~A)
59   ans =
60    logical
61     0
```

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



Workspace

Name ▲	Value	Size	Bytes	Class
A	[5,0,-1;3,10,...	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
list4	[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 3, the expression `A > 6` results in a logical-value matrix the same size as `A` (lines 20-23). The logical matrix can be thought of as a "map" indicating the locations of the elements that are greater than 6. Note that the parentheses in line 3 are not needed, since the assignment (=) has lowest precedence (2.8a[1], page 92). We sometimes add redundant parentheses to enhance the readability, avoiding confusion.

Function `find`

The function `find` (line 4) 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 4), 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 5, the 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 3-5 can be combined, as shown in line 6, 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 7, and the result is in lines 38-40. ↵

Logical Indexing

[6] 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.

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

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

In other words, the function `find` in lines 6 and 7 can be removed. This is demonstrated in line 8 and lines 43-45.

Logical NOT (~)

The logical NOT (~) reverses logical values: `true` becomes `false`, and `false` becomes `true`. If we apply it on a numerical array (line 9), a nonzero value becomes `false` and a zero value becomes `true` (lines 49-51). If we apply the logical NOT again on the previous result (line 10), 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 11 and 59-61. The function `isequal` (line 11) 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. #

2.10 String Manipulations

2.10a 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 (2.4a[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]. ↓

```
clear
a = 'Hello,';
b = 'world!';
c = [a, ' ', b];
disp(c)
name = input('What is your name? ', 's');
years = input('What is your age? ');
disp(['Hello, ', name, '! You are ', num2str(years), ' years old.'])
str = sprintf('Pi = %.8f', pi);
disp(str)
Names1 = [
    'David  '
    'John   '
    'Stephen'];
Names2 = char('David', 'John', 'Stephen');
if isequal(Names1, Names2)
    disp('The two lists are equal.')
end
name = deblank(Names1(2,:));
disp(['The name ', name, ' has ', num2str(length(name)), ' characters.'])
```

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

```
>> Example02_10a
Hello, world!
What is your name? Lee
What is your age? 60
Hello, Lee! You are 60 years old.
Pi = 3.14159265
The two lists are equal.
The name John has 4 characters.
>>
```

About 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 as shown in line 4; `c` is now 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).

The function `sprintf` (line 9) is the same as `fprintf` except that it writes to a string (rather than to the screen). Here, we write the value of π with 8 digits after the decimal point in a string and then assign to 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 (lines 11-14) 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). #

2.10b Example: A Simple Calculator

Example02_10b.m: A Simple Calculator

[1] This program uses function `eval` to create a simple calculator. Type and run the program, and see a test run in [2], next page. ↵

```
clear
disp('A Simple Calculator')
while true
    expr = input('Enter an expression (or quit): ', 's');
    if strcmp(expr,'quit')
        break
    end
    disp([expr, ' = ', num2str(eval(expr))])
end
```

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

```
>> Example02_10b
A Simple Calculator
Enter an expression (or quit): 3+5
3+5 = 8
Enter an expression (or quit): sin(pi/4) + (2 + 2.1^2)*3
sin(pi/4) + (2 + 2.1^2)*3 = 19.9371
Enter an expression (or quit): quit
>>
```

Table 2.10 String Manipulations

Function	Description
`A = char(a,b,...)`	Convert the strings to a matrix of rows of the strings, padding blanks
`disp(X)`	Display value of variable
`x = input(prompt,'s')`	Request user input
`s = sprintf(format,a,b,...)`	Write formatted data to a string
`s = num2str(x)`	Convert number to string
`x = str2num(s)`	Convert string to number
`x = str2double(s)`	Convert string to double precision value
`x = eval(exp)`	Evaluate a MATLAB expression
`s = deblank(str)`	Remove trailing blanks from a string
`s = strtrim(str)`	Remove leading and trailing blanks from a string
`tf = strcmp(s1,s2)`	Compare two strings (case sensitive)
`tf = strcmpi(s1,s2)`	Compare two strings (case insensitive)

Details and More:
Help>MATLAB>Language Fundamentals>Data Types>Characters and Strings; Data Type Conversion

2.11 Expressions

2.11a Example: Law of Sines

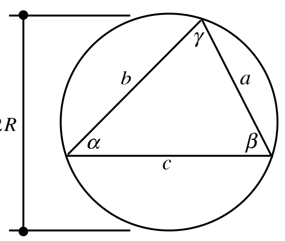
[1] An expression is a syntactic combination of **data** (constants or variables; scalars, vectors, matrices, etc.), **functions** (built-in or user-defined), **operators** (arithmetic, relational, or logical), and **special characters** (see Table 2.11a, page 107). An expression always results in a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

Table 2.11b (page 107) lists some frequently used math functions. ↓

Example: Law of Sines

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

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



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:

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

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

$$\alpha = \sin^{-1}\frac{a}{2R},\ \beta = \sin^{-1}\frac{b}{2R},\ \gamma = \sin^{-1}\frac{c}{2R} \qquad \text{(c)}$$

$$A = \frac{abc}{4R} \text{ or } A = \frac{1}{2}bc\sin\alpha \qquad \text{(d)}$$ ↓

Example02_11a.m: Law of Sines

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

```
clear
a = 5;
b = 6;
c = 7;
R = a*b*c/sqrt((a+b+c)*(a-b+c)*(b-c+a)*(c-a+b))
alpha = asind(a/(2*R))
beta = asind(b/(2*R))
gamma = asind(c/(2*R))
sumAngle = alpha + beta + gamma
A1 = a*b*c/(4*R)
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 5 calculates the radius of the circumscribed circle according to Eq. (b), its output in lines 13-14.

Lines 6-8 calculate the three angles according to equations in (c), their outputs are in lines 15-20.

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). #

2.11b Example: Law of Cosines

Example02_11b.m: Law of Cosines

[1] The law of cosines states that (see *Wikipedia>Trigonometry*, with the same notations in 2.11a[2], last page):

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

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

```
1   clear
2   a = 5; b = 6; c = 7;
3   alpha = acosd((b^2+c^2-a^2)/(2*b*c))
4   beta = acosd((c^2+a^2-b^2)/(2*c*a))
5   gamma = acosd((a^2+b^2-c^2)/(2*a*b))
```

↓

```
 6   >> Example02_11b
 7   alpha =
 8      44.4153
 9   beta =
10      57.1217
11   gamma =
12      78.4630
13   >>
```

[2] This is the screen output of Example02_11b.m. The outputs are consistent with those in 2.11a[4]. #

Table 2.11a
Special Characters

Special characters	Description
`[]`	Brackets
`{}`	Braces
`()`	Parentheses
`'`	Matrix transpose
`.`	Field access
`...`	Continuation
`,`	Comma
`;`	Semicolon
`:`	Colon
`@`	Function handle

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

Table 2.11b
Elementary Math Functions

Function	Description
`sin(x)`	Sine (in radians)
`sind(x)`	Sine (in degrees)
`asin(x)`	Inverse sine (in radians)
`asind(x)`	Inverse sine (in degrees)
`cos(x)`	Cosine (in radians)
`cosd(x)`	Cosine (in degrees)
`acos(x)`	Inverse cosine (in radians)
`acosd(x)`	Inverse cosine (in degrees)
`tan(x)`	Tangent (in radians)
`tand(x)`	Tangent (in degrees)
`atan(x)`	Inverse tangent (in radians)
`atand(x)`	Inverse tangent (in degrees)
`atan2(y,x)`	Four-quadrant inverse tangent (radians)
`atan2d(y,x)`	Four-quadrant inverse tangent (degrees)
`abs(x)`	Absolute value
`sqrt(x)`	Square root
`exp(x)`	Exponential (base *e*)
`log(x)`	Logarithm (base *e*)
`log10(x)`	Logarithm (base 10)
`factorial(n)`	Factorial
`sign(x)`	Sign of a number
`rem(a,b)`	Remainder after division
`mod(a,m)`	Modulo operation

Details and More: Help>MATLAB>Mathematics>Elementary Math

2.12 Example: Function Approximation

2.12a Scalar Expressions

Taylor Series

[1] At 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 detail on the Taylor series):

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

The more terms, the more accurate the approximation. ↙

Example02_12a.m: Scalar Expressions

[2] This script evaluates $\sin(\pi/4)$ 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 to the accuracy of an order of 10^{-7}. #

2.12b Use of For-Loop

[1] 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. 2.12a(a) as follows:

$$\sin x = \sum_{k=1}^{\infty} (-1)^{k-1} \frac{x^{2k-1}}{(2k-1)!} \qquad \text{(a)}$$

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

Example02_12b.m: Use of For-Loop

[2] Type and run this program. The screen output is the same as 2.12a[3], last page. ↓

```
1  clear
2  x = pi/4; n = 4; sinx = 0;
3  for k = 1:n
4      sinx = sinx + ((-1)^(k-1))*(x^(2*k-1))/factorial(2*k-1);
5  end
6  format long
7  sinx
8  exact = sin(x)
9  error = (sinx-exact)/exact
```

About Example02_12b.m

[3] In line 2, the variable `sinx` is initialized to zero. The statement within the `for`-loop (line 4) 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. (a) and added to the variable `sinx`. At the completion of the `for`-loop, `sinx` has the function value; the output is the same as 2.12a[3], last page. To increase the accuracy of the value, you may simply increase the number of items (see [4], for example). ↓

[4] This is the screen output when 6 items are used (i.e., change to `n = 6` in line 2). Note that the error reduces to an order of 10^{-12}. #

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

2.12c Vector Expressions

Example02_12c.m: Vector Expressions

[1] This script produces the same outputs as 2.12a[3], last page, using a vector expression (line 4) in place of the `for`-loop used in Example02_12b.m. ↵

```
1  clear
2  x = pi/4; n = 4; k = 1:n;
3  format long
4  sinx = sum(((-1).^(k-1)).*(x.^(2*k-1))./factorial(2*k-1))
5  exact = sin(x)
6  error = (sinx-exact)/exact
```

About Example02_12c.m

[2] In line 2, the variable `k` is created as a row vector of four elements; `k = [1,2,3,4]`. The `for`-loop in Example02_12b.m is now replaced by a vector expression (line 4), which uses the function `sum` and element-wise operators (`.^`, `.*`, and `./`). To help you understand the statement in line 4, 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 (2.8a[6-7], pages 95-96) and the function `sum` for vectors (2.7[5], page 90), we may further elaborate 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
```

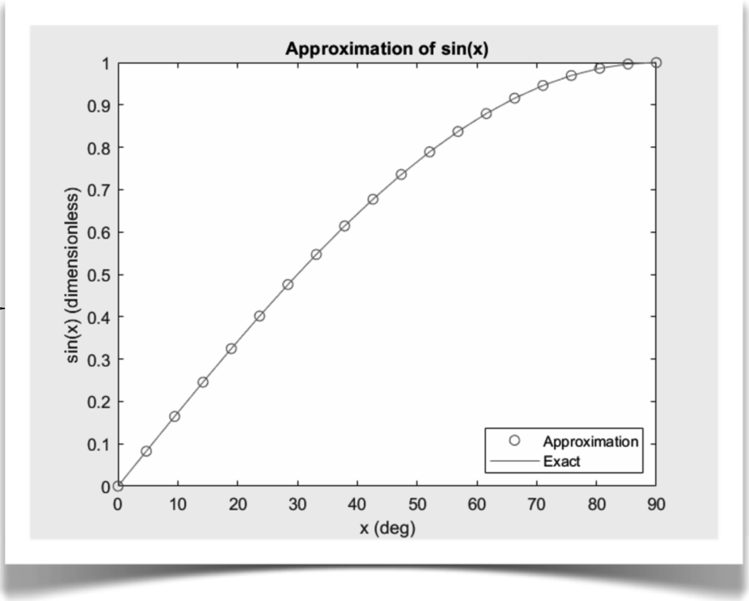
Substituting `x` with `pi/4`, we have `sinx = 0.707106469575178`. #

2.12d Matrix Expressions

Example02_12d.m: Matrix Expressions

[1] This script calculates sin(*x*) for various *x* values and produces a graph as shown in [2], next page. A matrix expression (line 6) is used in this script. ↵

```
 1  clear
 2  x = linspace(0,pi/2,20);
 3  n = 4;
 4  k = 1:n;
 5  [X, K] = meshgrid(x, k);
 6  sinx = sum(((-1).^(K-1)).*(X .^ (2*K-1))./factorial(2*K-1));
 7  plot(x*180/pi, sinx, 'o', x*180/pi, sin(x))
 8  title('Approximation of sin(x)')
 9  xlabel('x (deg)')
10  ylabel('sin(x) (dimensionless)')
11  legend('Approximation', 'Exact', 'Location', 'southeast')
```



[2] 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`

[3] Line 2 creates a row vector of 20 *x*-values using the function `linspace` (2.6a[5], page 85). Line 4 creates a row vector of four integers.

Line 5 generates 2-D grid coordinates `X` and `K` based on the coordinates in the row vectors `x` and `k`; the sizes of `X` and `K` are also based on the sizes of `x` and `k`. If the length of `x` is `nx` and the length of `k` is `nk`, then both `X` and `K` are `nk`-by-`nx` matrices; each row of `X` is a copy of `x`, and each column of `K` is a copy of `k'`. In other words, line 5 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 yourself):

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

In line 5 of Example02_12d.m, last page, 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 6)

Analysis of line 6 is similar to line 4 of Example02_12c (see 2.12c[2], 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 in a row vector of 20 values, which are plotted as circular marks in [2].

Plotting Marks and Curve

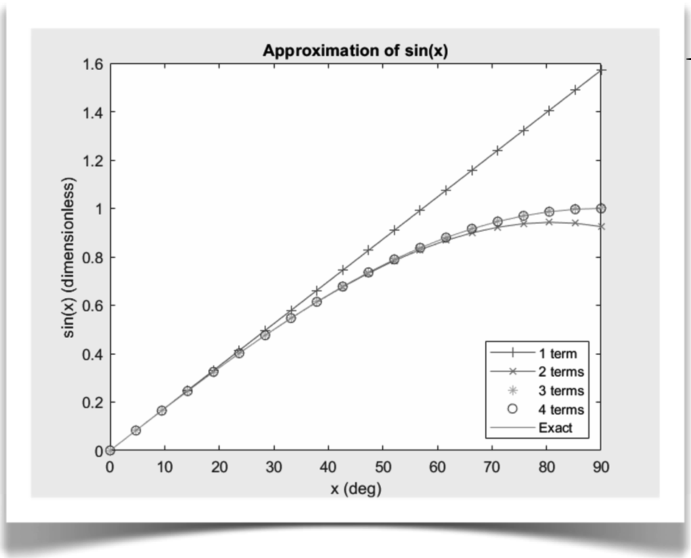
Line 7 plots the 20 values with circular marks, along with the exact function values, by default, using a solid line. Line 11 adds legends (Section 5.7) on the lower-right corner of the graphic area. #

2.12e Multiple Curves

Example02_12e.m: Multiple Curves

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

```
 1  clear
 2  x = linspace(0,pi/2,20);
 3  n = 4;
 4  k = 1:n;
 5  [X, K] = meshgrid(x, k);
 6  sinx = cumsum(((-1).^(K-1)).*(X .^ (2*K-1))./factorial(2*K-1));
 7  plot(x*180/pi, sinx(1,:), '+-', ...
 8       x*180/pi, sinx(2,:), 'x-', ...
 9       x*180/pi, sinx(3,:), '*', ...
10       x*180/pi, sinx(4,:), 'o', ...
11       x*180/pi, sin(x))
12  title('Approximation of sin(x)')
13  xlabel('x (deg)')
14  ylabel('sin(x) (dimensionless)')
15  legend('1 term', '2 terms', '3 terms', '4 terms', 'Exact', ...
16         'Location', 'southeast')
```



[2] Example02_12e.m plots four approximated curves and an exact curve of sin(*x*), for comparison. ↓

About Example02_12e.m

[3] Line 6 looks like line 6 in Example02_12d.m except that the function `cumsum` (2.7[6], page 91) is used in place of `sum`. The function `cumsum` calculates the cumulative sums of the four values in each column, resulting in a 4-by-20 matrix, the kth row representing the approximated function values when `k` terms are added up. Lines 7-11 plot the four approximated curves, each a row of the function values, and the exact curve. ↓

Line Styles and Marker Types

[4] In line 7, the notation `'+-'` means a plus **marker** and a solid **line style**. Similarly, in line 8, `'x-'` means a cross marker and a solid line style. Table 5.5a (page 226) lists various line styles and marker types.

Legend

Lines 15-16 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 $0 \le x \le 1$ and $0 \le y \le 1$.

A series solution of the equation is

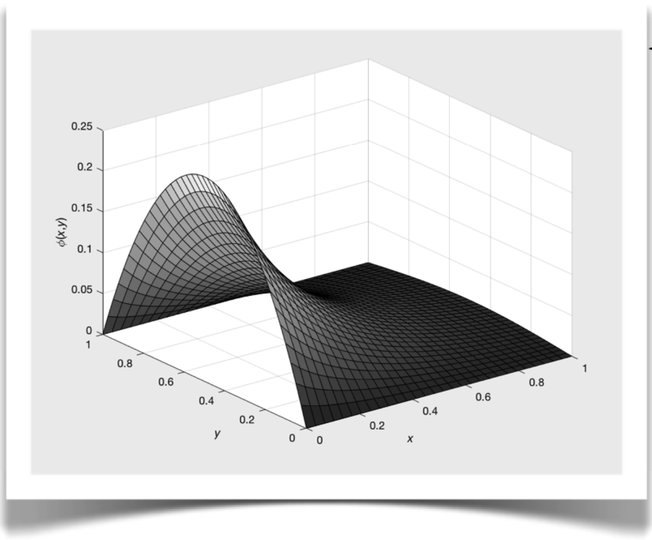
$$\phi(x,y) = 4\sum_{k=1}^{\infty} \frac{1-\cos(k\pi)}{(k\pi)^3} e^{-k\pi x} \sin(k\pi y) \tag{a}$$

You may verify the solution by substituting it into the equation and the boundary conditions. ↓

Example02_13.m: Series Solution of a Laplace Equation

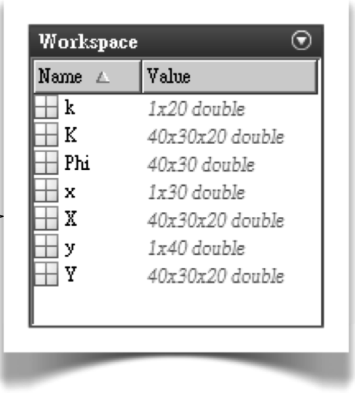
[2] This script calculates the solution $\phi(x,y)$ according to Eq. (a) and plots a three-dimensional surface $\phi = \phi(x,y)$ [3]. ↓

```
clear
k = 1:20;
x = linspace(0,1,30);
y = linspace(0,1,40);
[X,Y,K]  = meshgrid(x, y, k);
Phi = sum(4*(1-cos(K*pi))./(K*pi).^3.*exp(-K.*X*pi).*sin(K.*Y*pi), 3);
surf(x, y, Phi)
xlabel('\itx')
ylabel('\ity')
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 row vectors `x`, `y`, and `k`. If the lengths 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 a **row-vector**, and an $m \times 1$ matrix is 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 6 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, in 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 signal the beginning of a series of italicized characters, and `\rm` is used to signal the removing of 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

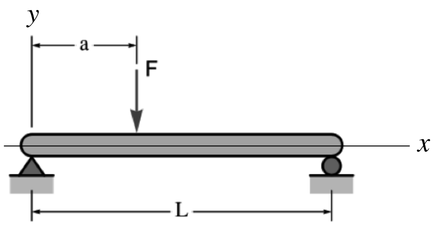
[1] Consider a simply supported beam subject to a force F (see [2], *source: https://en.wikipedia.org/wiki/File:Simple_beam_with_offset_load.svg, by Hermanoere*). The beam has a cross section of width w and height h, and a Young's modulus E. The deflection y of the beam as a function of x is (*Reference: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

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

where

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

$$< x-a >^3 \ = \begin{cases} 0, \text{ if } x \le a \\ (x-a)^3, \text{ if } x > a \end{cases}$$



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

Physical meanings of these quantities are 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.

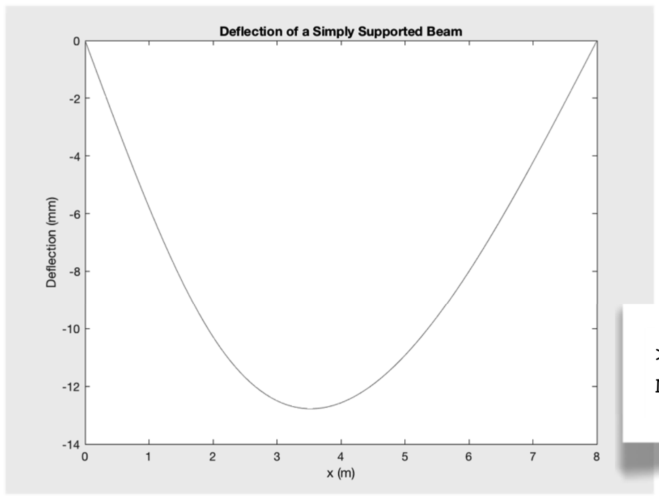
In this section, we'll plot a deflection curve and find the maximum deflection and its corresponding location, using the following parameters:

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

Example02_14.m: Deflection of Beams

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

```
clear
w = 0.1;
h = 0.1;
L = 8;
E = 210e9;
F = 3000;
a = L/4;
I = w*h^3/12;
R = F/L*(L-a);
theta = F*a/(6*E*I*L)*(2*L-a)*(L-a);
x = linspace(0,L,100);
y = -theta*x+R*x.^3/(6*E*I)-F/(6*E*I)*((x>a).*((x-a).^3));
plot(x,y*1000)
title('Deflection of a Simply Supported Beam')
xlabel('x (m)'); ylabel('Deflection (mm)')
y = -y;
[ymax, index] = max(y);
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

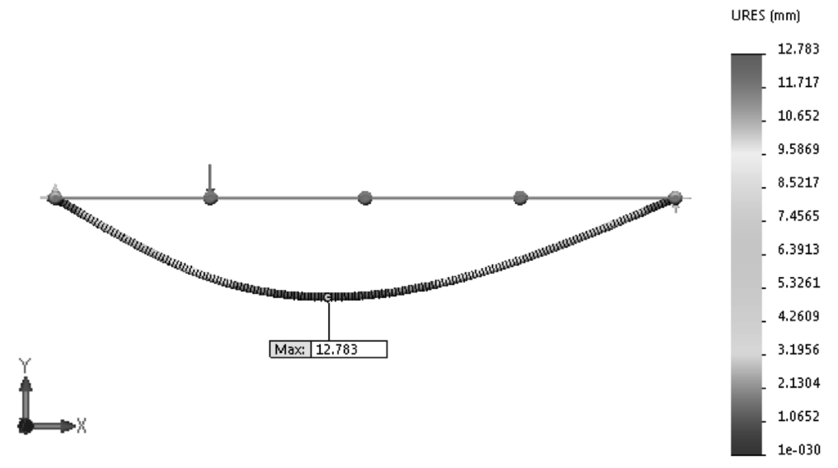
[6] The function

$$<x-a>^3 = \begin{cases} 0, \text{ if } x \le a \\ (x-a)^3, \text{ if } x > a \end{cases}$$

is implemented (see line 12) using a logical operator

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

Remember 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

2.15a Undamped Free Vibrations

Free Vibrations of a Supported Machine

[1] The figure shown in [2] (*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. (*Reference: 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.

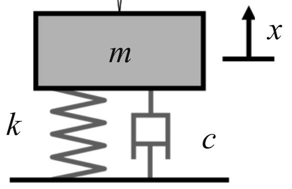
To derive the governing equation, consider that the machine moves with a displacement *x* and a velocity $\dot{x}$. The machine is subject to an elastic force $F_s = -kx$ and a damping force $F_d = -c\dot{x}$. Newton's second law states that the resultant force acting on the machine is equal to the multiplication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

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

[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

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

The solution for Eq. (a) is

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

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

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

You may verify the solution (b) by substituting it to Eq. (a).

The natural period (with SI unit s) is then

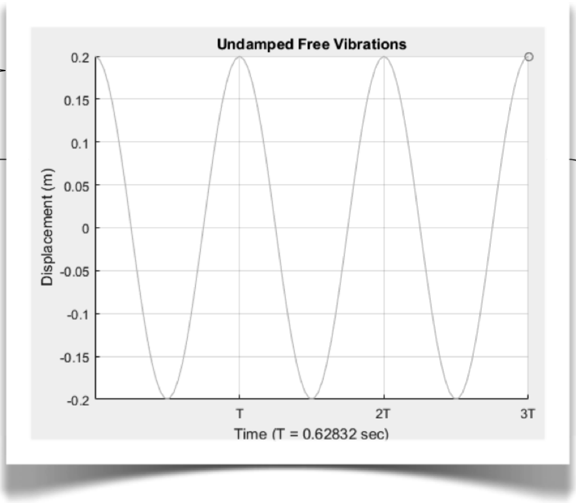
$$T = \frac{2\pi}{\omega} \qquad \text{(d)}$$

Example02_15a.m ([4], next page) calculates and plots the solution, using the following parameters

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

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]. ↑

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

2.15b Damped Free Vibrations

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

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

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

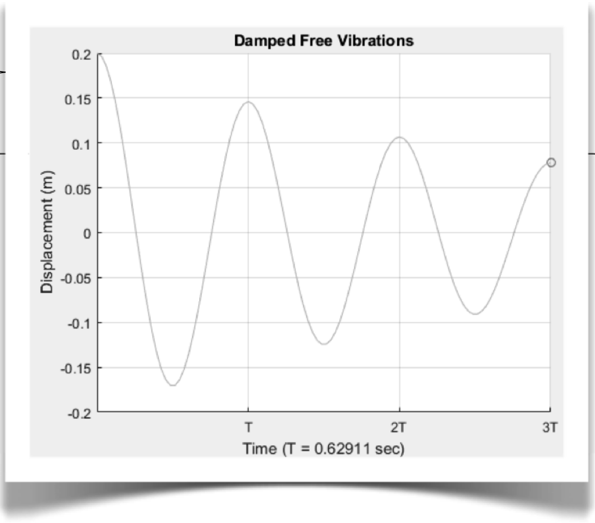
$$x(t) = \delta e^{-\frac{ct}{2m}}(\cos\omega_d t + \frac{c}{2m\omega_d}\sin\omega_d t) \tag{b}$$

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

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

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

[3] 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

[2] This program calculates and plots the solution $x(t)$ in Eqs. (b, c), last page. The graphic output is shown in [3]. ↑

```
clear
m = 1; k = 100; c = 1; delta = 0.2;
omega = sqrt(k/m);
cC = 2*m*omega;
omegaD = omega*sqrt(1-(c/cC)^2);
T = 2*pi/omegaD;
t = linspace(0, 3*T, 100);
x = delta*exp(-c*t/(2*m)).*(cos(omegaD*t)+c/(2*m*omegaD)*sin(omegaD*t));
axes('XTick', T:T:3*T, 'XTickLabel', {'T','2T','3T'});
axis([0, 3*T, -0.2, 0.2])
grid on
hold on
comet(t, x)
title('Damped Free Vibrations')
xlabel(['Time (T = ', num2str(T), ' sec)'])
ylabel('Displacement (m)')
```

2.15c Harmonically Forced Vibrations

[1] Now, imagine that 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

$$m\ddot{x} + c\dot{x} + kx = F\sin\omega_f t \tag{a}$$

The solution of Eq. (a) is a superposition of two parts: First, the free vibrations caused by the initial conditions. This part will eventually vanish due to the damping effects, as shown in 2.15b[3], and is called the **transient response**. Second, 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

$$x(t) = x_m \sin(\omega_f t - \varphi) \tag{b}$$

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

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

$$\varphi = \tan^{-1}\frac{2\left(c/c_c\right)\left(\omega_f/\omega\right)}{1-\left(\omega_f/\omega\right)^2} \tag{d}$$

↵

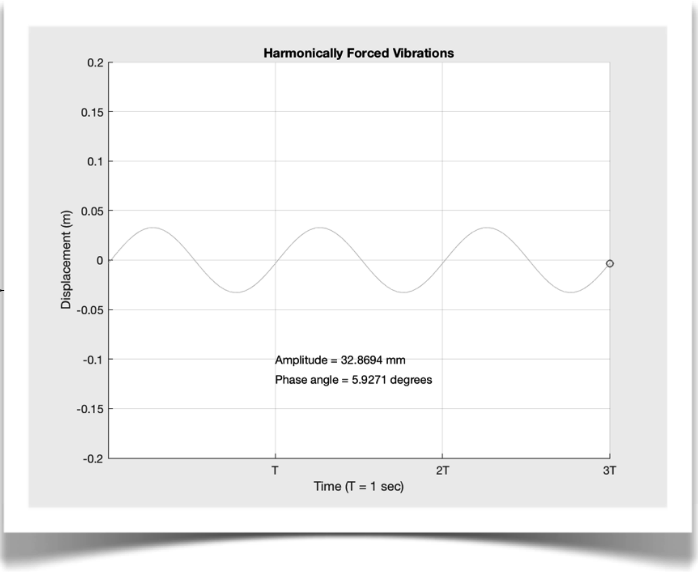
Example02_15c.m: Forced Vibrations

[2] This program plots the steady-state response *x*(*t*) in Eqs. (b, c, d), last page. The graphic output is shown in [3]. ↓

```
clear
% System parameters
m = 1; k = 100; c = 1;
f = 2; omegaF = 2*pi;

% System response
omega = sqrt(k/m);
cC = 2*m*omega;
rC = c/cC;
rW = omegaF/omega;
xm = (f/k)/sqrt((1-rW^2)^2+(2*rC*rW)^2);
phi = atan((2*rC*rW)/(1-rW^2));
T = 2*pi/omegaF;
t = linspace(0, 3*T, 100);
x = xm*sin(omegaF*t-phi);

% Graphic output
axes('XTick', T:T:3*T, 'XTickLabel', {'T','2T','3T'});
axis([0, 3*T, -0.2, 0.2])
grid on
hold on
comet(t, x)
title('Harmonically Forced Vibrations')
xlabel(['Time (T = ', num2str(T), ' sec)'])
ylabel('Displacement (m)')
text(T,-0.1,['Amplitude = ', num2str(xm*1000), ' mm'])
text(T,-0.12,['Phase angle = ', num2str(phi*180/pi), ' degrees'])
```

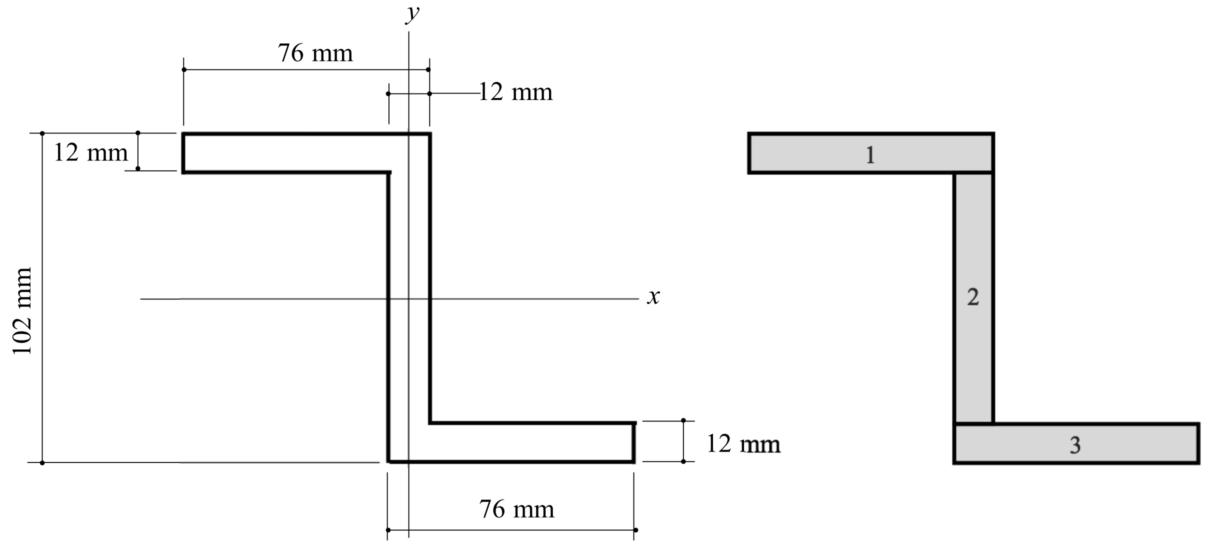


[3] The output of Example02_15c.m. #

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: $I_x = 4{,}190{,}040$ mm^4 and $I_y = 2{,}756{,}960$ 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 $x \le n$. 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 $C_4^{100} = 3{,}921{,}225$.

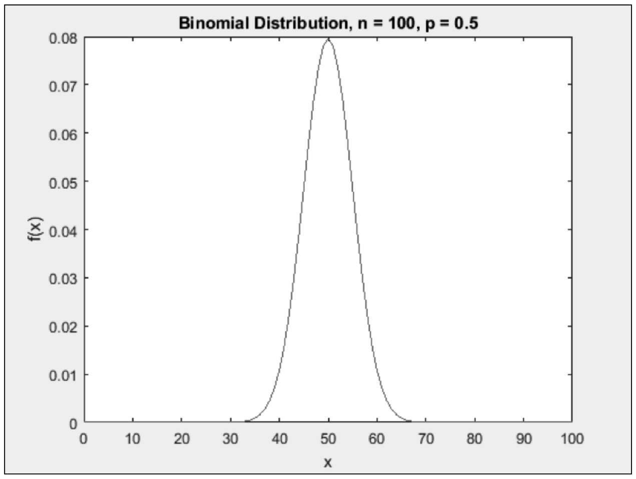
Problem02_03: Binomial Distribution

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

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

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

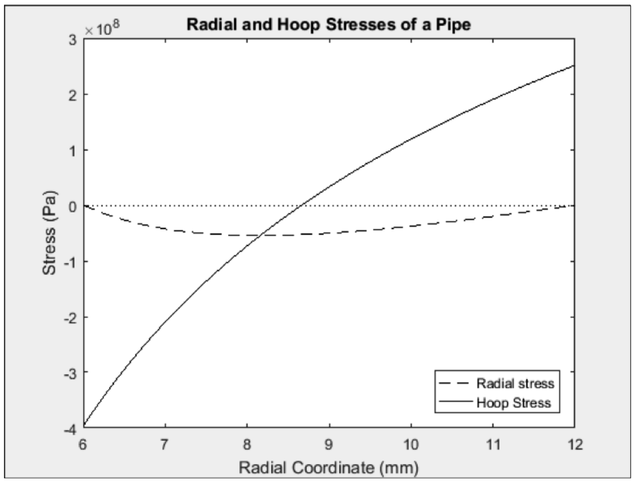
The radial stress σ_r and hoop stress σ_h in a long pipe due to a temperature T_a at its inner surface of radius a and a temperature T_b at its outer surface of radius b are, respectively, (*A. H. Burr and J. B. Cheatham, Mechanical Analysis and Design, 2nd ed., Prentice Hall, p. 496.*)

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

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

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 to the right, which uses the following data: $a = 6$ mm, $b = 12$ mm, $T_a = 260^\circ C$, $T_b = 150^\circ C$, $E = 206$ GPa, $\nu = 0.3$, $\alpha = 2\times10^{-5}\ /^\circ$C.

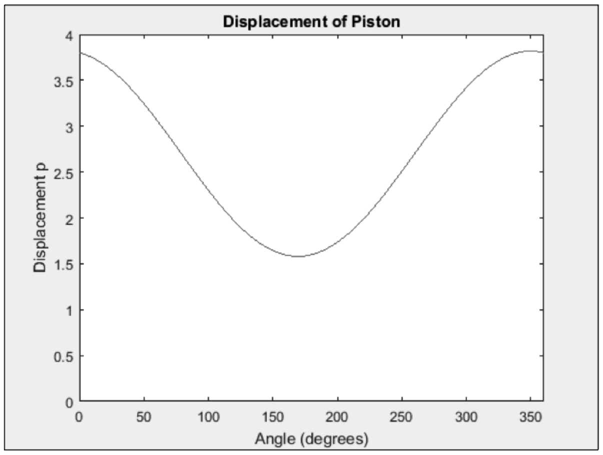


Problem02_05: Displacement of a Piston

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

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

Write a script to plot the displacement p as a function of angle θ (in degrees; $0 \le \theta \le 360^\circ$) 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:

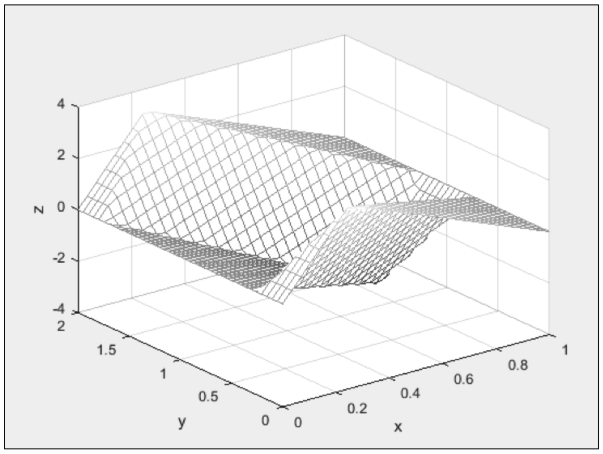
$$\pi = \sum_{k=0}^{n}\left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6}\right)\left(\frac{1}{16}\right)^k$$

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

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


z
y
x
4
2
0
-2
-4
2
1.5
1
0.5
0
0
0.2
0.4
0.6
0.8
1