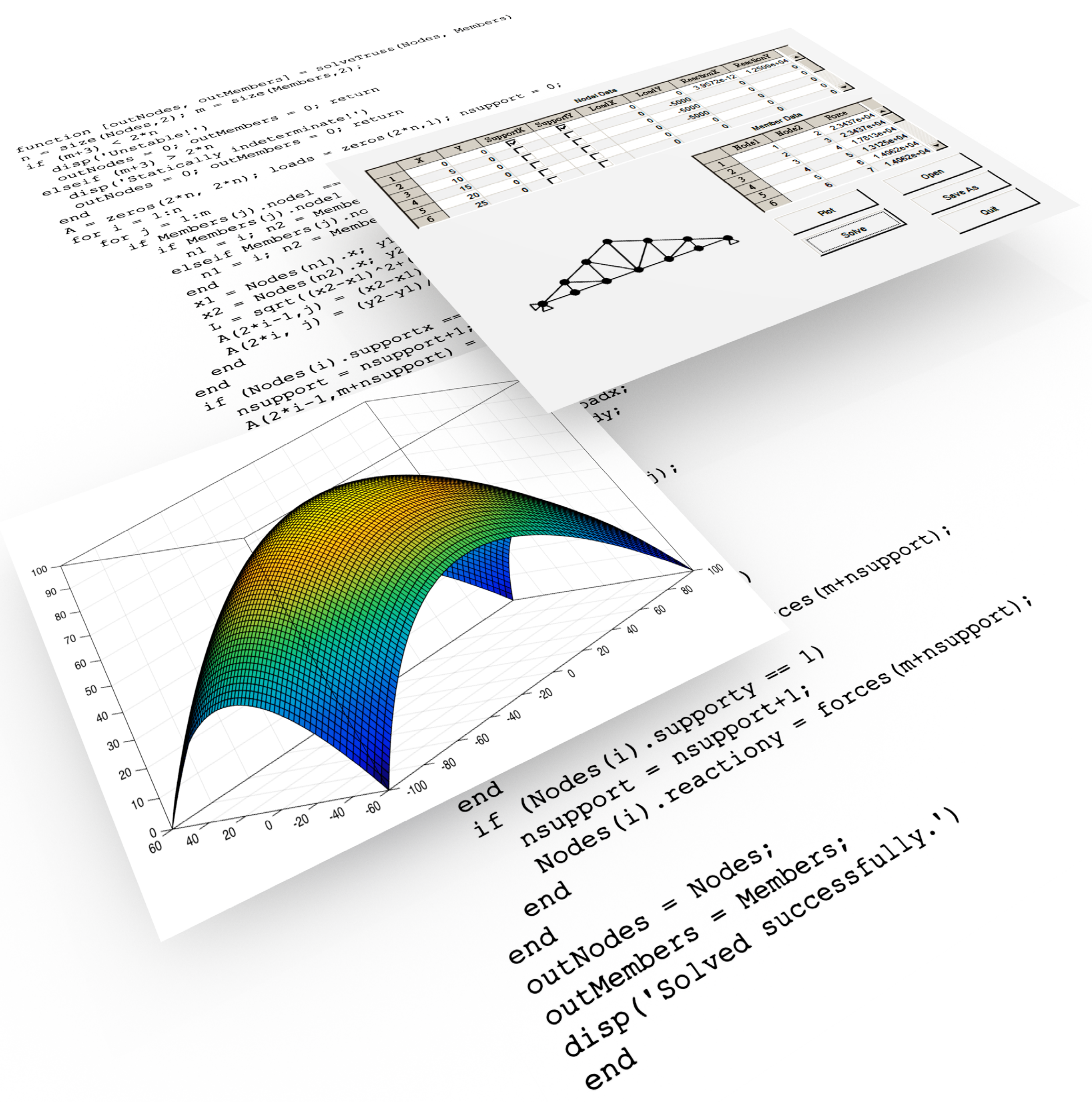
Huei-Huang Lee

Programming with MATLAB 2016



Chapter 2
Data Types, Operators, and Expressions

An expression is a syntactic combination of **data**, **operators**, and **functions**. An expression always results in a **data**. 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 Numeric Data Types

Example02_01.m: Numeric Data Types

[1] On the **Command Window**, type the following commands (all command files in this book are available for free download; please see 1.11[14], page 29):

```
 1   >> clear, clc
 2   >> format short
 3   >> format compact
 4   >> a = 1234.56789012345678901234
 5   a =
 6      1.2346e+03
 7   >> fprintf('%.20f\n', a)
 8   1234.56789012345690000000
 9   >> format long
10   >> a
11   a =
12      1.234567890123457e+03
13   >> b = single(a)
14   b =
15      1.2345679e+03
16   >> int16(a)
17   ans =
18      1235
19   >> c = 234
20   c =
21      234
22   >> d = int16(c)
23   d =
24      234
25   >> int8(d)
26   ans =
27      127
28   >> uint8(d)
29   ans =
30      234
31   >> intmax('int8')
32   ans =
33      127
34   >> intmin('int8')
35   ans =
36     -128
37   >> intmax('uint8')
38   ans =
39      255
40   >> intmin('uint8')
41   ans =
42      0
```

Screen Output Format

[2] Lines 2, 3, and 9 set **Command Window** output display format. The syntax is

```
format style
```

The `short` (line 2) sets the display of fixed-decimal format with 4 digits after the decimal point, the `long` (line 9) with 15 digits. The `compact` (line 3) suppress blank lines to make the output lines compact. The opposite of `compact` is `loose` (default), which adds blank lines to make the output lines more readable. In this book, we always use `compact` style to save space.

Table 2.1a lists all the available format styles. Remember, you may always consult the on-line documentation whenever a new command is encountered. For example:

```
>> doc format
```

→

Table 2.1a 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`
or Help>MATLAB>Language Fundamentals>Entering Commands>format

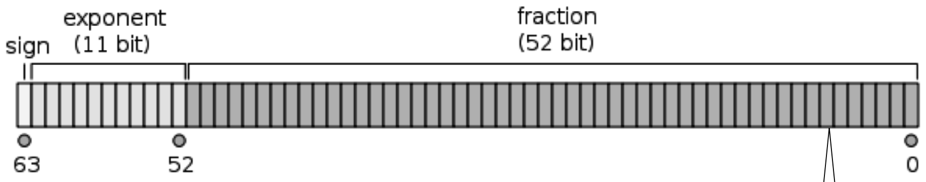
Double-Precision Floating-Point Numbers

[3] Table 2.1b lists all numeric data types supported by MATLAB. By default, MATLAB stores numbers in `double`, double-precision floating-point format (see [4]). Lines 4-12 show some important concepts of this data type.

In line 4, we assign a number of 24 significant figures to variable `a`. We'll see (line 8) that, due to the limiting storage space [4], not all the figures can be stored in the variable `a`. Lines 5-6 display the number in `short` format, i.e., with 4 digits after the decimal point. Note that, in displaying the number, it has been rounded to the last digit.

In line 7, we attempt to print the number with 20 digits after the decimal point. The result (line 8) shows that only 16 significant figures are stored in the variable `a`. The extra digits are lost during the assignment (line 4).

With `long` format (line 9), the number is displayed with 15 digits after the decimal point (lines 10-12). Again, in displaying the number, it has been rounded to the last digit.



[4] A double-precision floating-point number uses 64 bits in computer memory: 1 bit for the sign, 11 bits for the exponent, and 52 bits for the fraction. (*Details and More: Wikipedia>Double-precision floating-point format.*) (*Figure source: https://en.wikipedia.org/wiki/File:IEEE_754_Double_Floating_Point_Format.svg, by Codekaizen*)

Table 2.1b Numeric Data Types

Function	Description
`double`	Convert to double-precision number
`single`	Convert to single-precision number
`int8`	Convert to 8-bit signed integer
`int16`	Convert to 16-bit signed integer
`int32`	Convert to 32-bit signed integer
`int64`	Convert to 64-bit signed integer
`uint8`	Convert to 8-bit unsigned integer
`uint16`	Convert to 16-bit unsigned integer
`uint32`	Convert to 32-bit unsigned integer
`uint64`	Convert to 64-bit unsigned integer

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

Integer Numbers

[6] Line 16 converts the number `a` to an integer 16-bits long. The result is 1235 (lines 17-18). Note that, during the conversion, the number is rounded to the nearest integer.

The Variable `ans`

[7] In line 16, there is no variable to store the result of conversion. Whenever a value is produced and there is no variable to store that value, MATLAB always uses the variable `ans` (short for **answer**) to store that value.

Single-Precision Floating-point Numbers

[5] Line 13 converts the value stored in variable `a` (which is of type `double`, 64-bits long) to a single-precision floating-point number (32-bits long) and stored in the variable `b`. The output (lines 14-15), shows that it reduces to 8 significant figures, due to the shorter storage space. The extra digits are discarded during the conversion in line 13.

[8] Line 19 creates a variable `c`, storing the number 234. By default, the number is stored as `double`, so the variable `c` has a 64-bit floating-point format.

Line 22 converts the number `c` to an `int16`, a 16-bit signed integer. MATLAB uses **two's complement representation** (see [9]) for all signed integers. →

Two's Complement Representation

[9] Table 2.1c lists some examples of unsigned representation and two's complement representation. The unsigned representation should be self-explanatory.

In two's complement representation, the first bit is reserved for the sign: 0 for positive and 1 for negative. Thus, binary numbers with zero leading bit have the same values when interpreted in either unsigned representation or two's complement representation. To interpret a binary number with one as leading bit, *you take its complement, add 1 to the complement, translate into decimal, and finally add a negative sign*. For example, the binary number 101 is equal to decimal -3. (The complement of 101 is 010. Adding 1, it becomes 011, which is a decimal 3. Finally, add a negative sign to become -3.)

Knowing this, we may calculate the minimum and maximum values of integer data types. Table 2.1d lists the minimum and maximum values of various integer data types supported by MATLAB, along with the functions to calculate theses values (also see lines 31-42).

[10] Line 25 attempts to convert the number 234 to an `int8`. However, because the maximum number an `int8` can store is 127 (see Table 2.1d), the extra values are truncated and the result is 127 (lines 26-27).

[11] Line 28 converts the number 234 to an `uint8` without any problem (lines 29-30), since the maximum number an `uint8` can store is 255 (see Table 2.1d). →

Table 2.1c Examples of Two's Complement Representation

Bits	Unsigned value	Two's complement 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 2.1d Minimum/Maximum Integer Numbers

Function	Minimum value	Function	Maximum value
`intmin('int8')`	-128	`intmax('int8')`	127
`intmin('int16')`	-32768	`intmax('int16')`	32767
`intmin('int32')`	-2147483648	`intmax('int32')`	2147483647
`intmin('int64')`	-9223372036854775808	`intmax('int64')`	9223372036854775807
`intmin('uint8')`	0	`intmax('uint8')`	255
`intmin('uint16')`	0	`intmax('uint16')`	65535
`intmin('uint32')`	0	`intmax('uint32')`	4294967295
`intmin('uint64')`	0	`intmax('uint64')`	18446744073709551615

Details and More: Types `>> doc intmin` *or* `>> doc intmax`

Fractional Binary Numbers

[12] For a decimal number 258.369, we implicitly interpret it 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

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

Double-Precision Floating-Point Format

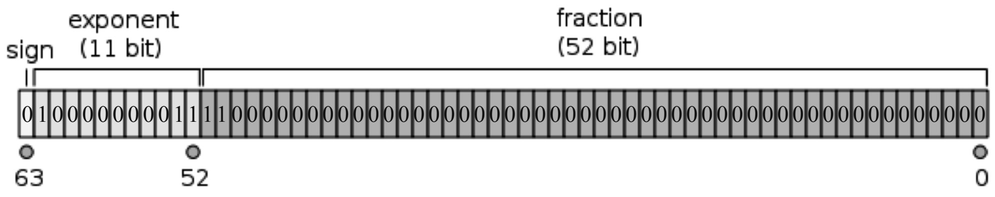
[13] A double-precision real number is stored with 64 bits (see [4], page 54). The 64-bit pattern is interpreted as a value of

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

where *exponent* is stored in the bits 52-62 as an 11-bit unsigned integer (see Table 2.1c, page 55). For example, the bit pattern below is interpreted as

$$(1.\,11)_2 \times 2^{1027-1023} = (2^0 + 2^{-1} + 2^{-2})\times 2^4 = 1.75\times16 = (28)_{10}$$

Thus, the double-precision number 28 is stored as follows:



2.2 Character Data Type

Example02_02a.m: Character Data Type

[1] On the **Command Window**, type the following commands:

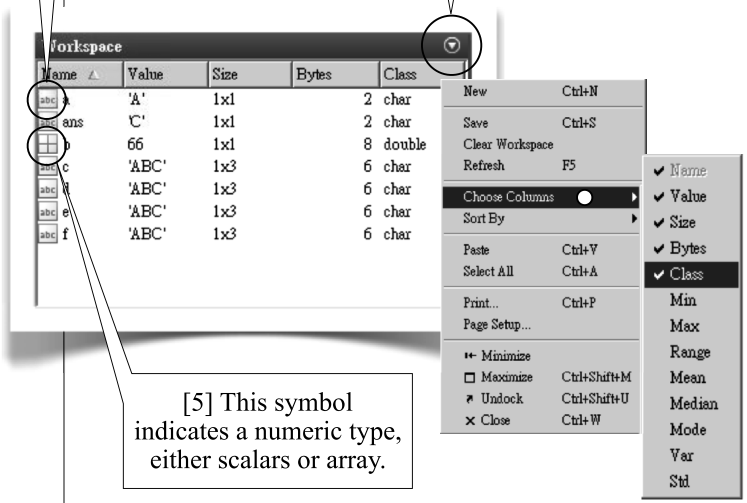
```
 1   >> clear, clc
 2   >> a = 'A'
 3   a =
 4   A
 5   >> b = a + 1
 6   b =
 7       66
 8   >> char(65)
 9   ans =
10   A
11   >> char('A'+2)
12   ans =
13   C
14   >> c = ['A', 'B', 'C']
15   c =
16   ABC
17   >> d = ['AB', 'C']
18   d =
19   ABC
20   >> e = ['A', 66, 67]
21   e =
22   ABC
23   >> f = 'ABC'
24   f =
25   ABC
26   >> f(1)
27   ans =
28   A
29   >> f(2)
30   ans =
31   B
32   >> f(3)
33   ans =
34   C
35   >> whos
35   Name     Size    Bytes   Class
36   a        1x1         2   char
37   ans      1x1         2   char
38   b        1x1         8   double
39   c        1x3         6   char
40   d        1x3         6   char
41   e        1x3         6   char
42   f        1x3         6   char
```

Function whos

[2] The function `whos` (line 35) lists the variables in the **Workspace**, with their sizes and classes (i.e., data types). These information also can be displayed on the **Workspace Window** (see [3]).

[3] Various properties of the variables in the **Workspace** can be shown by pulling down this button and select **Choose Columns**.

[4] This symbol indicates character type.



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

[5] This symbol indicates a numeric type, either scalars or array.

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

In line 5, 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`.

MATLAB uses 16 bits to store a character. When converted to a number, a `char` data is treated as a `uint16`. For example, the character `A`, when converted to a number, is 65 (see [7]).

In line 8, the numeric value 65 is converted to a `char`. The result is the character `A`.

In line 11, 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` (see [7]). →

ASCII Codes

[7] MATLAB stores characters according to ASCII Code (see *Wikipedia>ASCII*, also see [10]). An ASCII code is a number representing a character, either printable or non-printable. The ASCII codes of the character A, B, and C are 65, 66, and 67, respectively (see [10]).

Numeric Operations Involving Characters

[8] 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 Sections 2.5 and 2.6 and string manipulations in Section 2.8.

String: Row Vector of Characters

[9] Line 14 creates a row vector of three characters `A`, `B`, and `C`. It is displayed as `ABC` (line 16). A row vector of character is called a **string**. The variable `c` is a **string**.

Line 17 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`. There is no difference between variables `c` and `d`; they are all strings of three characters `ABC`.

In line 20, since an array must have elements of the same data type, MATLAB converts the number 66 and 67 to characters. The result is a row vector of three characters `ABC`. There is no difference between variables `c`, `d`, and `e`.

Line 23 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 characters `ABC`, which can be confirmed in lines 26-34.

Example02_02b.m: ASCII Codes

[10] MATLAB stores characters according to ASCII Code. ASCII codes 32-127 represent all printable characters on a standard keyboard. This example prints a table of characters corresponding to the ASCII codes 32-127.

```
43  clear
44  fprintf('    0 1 2 3 4 5 6 7 8 9\n')
45  for row = 3:12
46      fprintf('%2d  ', row)
47      for column = 0:9
48          code = row*10+column;
49          c = char(code);
50          fprintf('%c ', c)
51      end
52      fprintf('\n')
53  end
```

The screen output is

```
    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 { | } ~
```

Note that ASCII code 32 corresponds to the space character. #

2.3 Logical Data Type

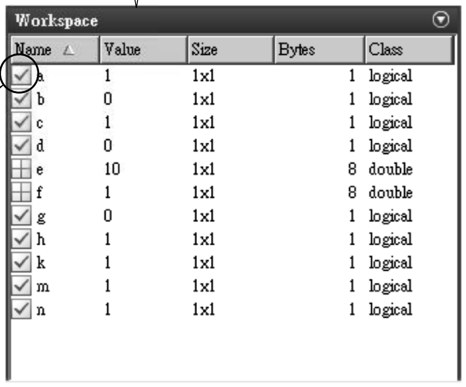
Example02_03.m: Logical Data Type

[1] On the **Command Window**, type following commands:

```
 1  >> clear, clc
 2  >> a = true
 3  a =
 4       1
 5  >> b = false
 6  b =
 7       0
 8  >> c = 6 > 5
 9  c =
10       1
11  >> d = 6 < 5
12  d =
13       0
14  >> e = (6 > 5)*10
15  e =
16      10
17  >> f = false*10+true
18  f =
19       1
20  >> g = (6 > 5) & (6 < 5)
21  g =
22       0
23  >> h = (6 > 5) | (6 < 5)
24  h =
25       1
26  >> k = logical(5)
27  k =
28       1
29  >> m = 5 | 0
30  m =
31       1
32  >> n = (-2) & 'A'
33  n =
34       1
```

[2] The **Workspace Window** looks like this.

[3] 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	1	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

Logical Values: `true` and `false`

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

[5] Line 2 assigns `true` to variable `a`, which is then of type `logical`. When displayed on the **Command Window**, `true` is always displayed as 1 (line 4).

Line 5 assigns `false` to variable `b`. When displayed on the **Command Window**, `false` is always displayed as 0 (line 7).

Relational Operators

[6] A relational operator (>, <, etc.) always operates on two numeric values, and the result is a logical value.

In line 8, the number 6 and the number 5 are operated using the logical operator `>`. The result is `true` and is assigned to `c`, which is of type `logical` and displayed as 1 (line 10).

In line 11, the number 6 and the number 5 are operated using the logical operator `<`. The result is `false` and is assigned to `d`, which is of type `logical` and displayed as 0 (line 13). →

Numeric Operations Involving Logical Values

[7] A numeric operator (+, -, etc.) 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 14, the result of (6 > 5) is a logical value `true`, which is to multiply by the number 10. Since the multiplication (`*`) is a numeric operation, MATLAB converts `true` to the number 1, and the result is 10 (line 16), which is a `double` number.

In line 17, again, since the multiplication (*) and the addition (+) are numeric operations, MATLAB converts `false` to 0 and `true` to 1, and the result is 1 (line 19), which is a `double` number.

Logical Operators

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

MATLAB uses the symbol `&` for logical AND and the symbol `|` for logical OR. Table 2.3a lists the rules for logical `AND` (`&`). Table 2.3b lists the rules for logical `OR` (`|`).

In line 20, 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` (line 22).

In line 23, the result of local OR (`|`) operation for a `true` and a `false` is `true` (line 25).

We'll introduce relational and logical operators in Section 2.7.

Conversion to Logical Data Type

[9] Line 26 converts a numeric value 5 to logical data type. The result is `true`. When converted to a logical value, any non-zero number becomes `true` and the number 0 becomes `false`.

In line 29, since `|` is a logical operation, MATLAB converts the numbers 5 and 0 to logical values `true` and `false`, respectively. The result is `true` (line 31).

In line 32, again, since `&` is a logical operation, MATLAB converts both the number -2 and the character 'A' to logical values `true`. The result is `true` (line 34).

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

[10] 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 2.3a Rules of Logical `and` (`&`)

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

Table 2.3b Rules of Logical `or` (`|`)

`OR (&)`	`true`	`false`
`true`	`true`	`true`
`false`	`true`	`false`

2.4 Arrays

Example02_04a.m

[1] On the **Command Window**, type the following commands:

```
 1  >> clear, clc
 2  >> a = 5
 3  a =
 4       5
 5  >> b = [5]
 6  b =
 7       5
 8  >> c = 5*ones(1,1)
 9  c =
10       5
11  >> D = ones(2, 3)
12  D =
13       1     1     1
14       1     1     1
15  >> e = [1, 2, 3, 4, 5]
16  e =
17       1     2     3     4     5
18  >> f = [1 2 3 4 5]
19  f =
20       1     2     3     4     5
21  >> g = [1:5]
22  g =
23       1     2     3     4     5
24  >> h = 1:5
25  h =
26       1     2     3     4     5
27  >> k = 1:1:5
28  k =
29       1     2     3     4     5
30  >> m = linspace(1,5,5)
31  m =
32       1     2     3     4     5
```

All data are arrays

[2] 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 ($r \times c \times p$) is simply called a **three-dimensional array**.

Scalar

[3] Lines 2-10 show many ways to create the same scalar. Line 2 creates a single value 5. Line 5 creates a vector of one element. Lines 8 creates a 1x1 matrix (see [4] for an explanation of the function `ones`). The variables `a`, `b`, and `c` are all **scalars**; they are all equal; there is no difference among these three variables.

Dimensionality and Dimension Sizes

[4] The function `ones` (line 8) creates an array of all ones with specified dimension sizes. Another example is shown in line 11. The syntax is

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

where `sz1` is the size of the first dimension, and so forth. The first dimension is called the **row** dimension; the second dimension is called the **column** dimension; the third dimension is called the **page** dimension.

The function `ones` is one of the array creation functions. Table 2.4a lists some other array creation functions.

In 1.13[5] (page 31), the variable `Photo` is a 384-by-512-by-3 array. The ith **row**, jth **column**, kth **page** of the array is referred to as `Photo(i,j,k)`, which stores a color intensity (a number between 0-127) of the pixel at the ith row (starting from top row) and jth column (starting from the left column) of the image in 1.13[3], page 31.

Row Vectors

[5] Lines 15-32 show many ways to create the same row vector. Line 15 creates a row vector using the square brackets (`[ ]`). Commas are used to separate elements in a row. The commas can be omitted (line 18).

Line 21 creates a row vector using the colon (`:`). The square brackets can be omitted (line 24). In a more general form, an increment number can be inserted between the starting number and the ending number (line 27; also see 1.3[4], page 14). The function `linspace` (line 30) is useful when creating a row vector of linearly spaced numbers. The syntax is

```
linspace(start,end,number-of-elements)
```

if *`number-of-element`* is omitted, it defaults to 100.

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

Example02_04b.m

[6] Continue to type the following commands:

```
33  >> clear, clc
34  >> a = zeros(1,5)
35  a =
36       0     0     0     0     0
37  >> a(1,5) = 8
38  a =
39       0     0     0     0     8
40  >> a(5) = 9
41  a =
42       0     0     0     0     9
43  >> a([1,2,4]) = [8,7,6]
44  a =
45       8     7     0     6     9
46  >> a(1:4) = [2,3,4,5]
47  a =
48       2     3     4     5     9
49  >> [rows, cols] = size(a)
50  rows =
51       1
52  cols =
53       5
54  >> len = length(a)
55  len =
56       5
57  >> b = a
58  b =
59       2     3     4     5     9
60  >> c = a(1:5)
61  c =
62       2     3     4     5     9
63  >> d = a(3:5)
64  d =
65       4     5     9
66  >> e = a(3:length(a))
67  e =
68       4     5     9
69  >> f = a(3:end)
70  f =
71       4     5     9
72  >> f(5) = 10
73  f =
74       4     5     9     0    10
```

[7] Line 34 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 index** (1.8[14], page 25). Line 37 uses subscript indexing, while line 40 uses linear indexing. For a vector, row vector or column vector, we usually use linear indexing.

[8] Line 43 assigns three values to the 1st, 2nd, and 4th elements of the array `a`. An array can be indexed this way.

Line 46, since `1:4` means `[1,2,3,4]`, assigns four values to the 1st-4th elements of the array `a`.

Size and Length of an Array

[9] Function `size` outputs dimension sizes of an array. In line 49, `size(a)` outputs two values: number of rows and number of columns, respectively, and a two-element vector is needed to stored 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 (lines 54-56), which is the number of columns.

[10] Line 57 assigns the entire array `a` to variable `b`, which becomes the same sizes and contents as the array `a`. Line 60 uses another way to assign all the values of the array `a` to a variable. The variables `a`, `b`, and `c` are equal in size and content.

[11] Lines 63, 66, and 69 demonstrate three ways to assign the 3rd, 4th, and 5th elements of the array `a` to a variable. The variables `d`, `e`, and `f` are all equal in sizes and contents.

Note that, in line 69, the keyword `end` means the **last index**.

[12] Line 72 attempts to assign a value to the 5th element of the array `f`, which is a row vector of length 3. MATLAB expands the array `f` to a row vector of length 5 to accommodate the value and pads zeros for the unused elements. `f` now is a row vector of length 5. →

Example02_04c.m

[13] Continue to type following commands:

```
 75   >> clear, clc
 76   >> a = [1, 2; 3, 4; 5, 6]
 77   a =
 78        1     2
 79        3     4
 80        5     6
 81   >> b = 1:6
 82   b =
 83        1     2     3     4     5     6
 84   >> c = reshape(b, 3, 2)
 85   c =
 86        1     4
 87        2     5
 88        3     6
 89   >> d = reshape(b, 2, 3)
 90   d =
 91        1     3     5
 92        2     4     6
 93   >> e = d'
 94   e =
 95        1     2
 96        3     4
 97        5     6
 98   >> c(:,3) = [7, 8, 9]
 99   c =
100        1     4     7
101        2     5     8
102        3     6     9
103   >> c(4,:) = [10, 11, 12]
104   c =
105        1     4     7
106        2     5     8
107        3     6     9
108       10    11    12
109   >> c(4,:) = []
110   c =
111        1     4     7
112        2     5     8
113        3     6     9
114   >> c(:,2:3) = []
115   c =
116        1
117        2
118        3
```

[14] Line 76 creates a 3-by-2 matrix. Line 81 creates a row vector of 6 elements. Line 84 reshapes the vector into a 3-by-2 matrix. The reshaping doesn't alter the order of the elements; it alters dimensionality and dimension sizes. Note that `c` is different from `a`. To obtain a matrix the same as `a` from the vector `b`, we reshape `b` into a 2-by-3 matrix first (line 89) and then transpose it (line 93). Now `e` is the same as `a`.

Colon: The Entire Column/Row

[15] Line 98 assigns 3 elements to the third column of `c`. Line 103 assigns 3 elements to the fourth row of `c`.

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

`[ ]`: Empty Data

[16] Line 109 sets the fourth row of `c` to empty, i.e., deleting the entire row. Line 114 sets the 2nd-3rd columns to empty, i.e., deleting the two columns.

The `[ ]` is used to represent the empty data. →

Table 2.4a Array Creation Functions

Function	Description
`zeros`	Create an array of all zeros
`ones`	Create an array of all ones
`eye`	Create an identity matrix
`diag`	Create a diagonal matrix
`rand`	Create an array of uniformly distributed random numbers
`randn`	Create an array of normally distributed random numbers
`linspace`	Create a row vector of linearly spaced numbers
`logspace`	Create a row vector of logarithmically spaced numbers
`meshgrid`	Create a rectangular grid in 2-D or 3-D space

Details and More: Help>MATLAB>Language Fundamentals>Matrices and Arrays>Array Creation and Concatenation

Example02_04d.m

[17] Continue to type following commands:

```
119  >> clear, clc
120  >> a = reshape(1:6, 3, 2)
121  a =
122       1     4
123       2     5
124       3     6
125  >> b = [7; 8; 9]
126  b =
127       7
128       8
129       9
130  >> c = horzcat(a, b)
131  c =
132       1     4     7
133       2     5     8
134       3     6     9
135  >> d = [a, b]
136  d =
137       1     4     7
138       2     5     8
139       3     6     9
140  >> e = b'
141  e =
142       7     8     9
143  >> f = vertcat(d, e)
144  f =
145       1     4     7
146       2     5     8
147       3     6     9
148       7     8     9
149  >> g = [d; e]
150  g =
151       1     4     7
152       2     5     8
153       3     6     9
154       7     8     9
155  >> h = fliplr(c)
156  h =
157       7     4     1
158       8     5     2
159       9     6     3
160  >> k = flipud(c)
161  k =
162       3     6     9
163       2     5     8
164       1     4     7
```

Concatenation of Arrays

[18] Line 120 creates a 3-by-2 matrix `a` by reshaping the vector `[1:6]`. Line 125 creates a column vector `b` of 3 elements.

Using function `horzcat`, line 130 concatenates `a` and `b` horizontally to create a 3-by-3 matrix `c`. Line 135 demonstrates a more convenient way to do the same thing, using the comma (`,`) to concatenate arrays horizontally.

Line 140 transposes (see 1.6[10], page 19) the column vector `b` to create a row vector `e` of 3 elements.

Using function `vertcat`, line 143 concatenates `d` and `e` vertically to create a 4-by-3 matrix `f`. Line 149 demonstrates a more convenient way to do the same thing, using the semicolon (`;`) to concatenate arrays vertically.

Flipping Matrices

[19] Using function `fliplr` (flip left-side right), line 155 flips the matrix `c` horizontally. Using function `flipud` (flip upside down), line 160 flips the matrix `c` vertically.

Functions for array replication, concatenation, flipping, and reshaping are summarized in Table 2.4b.

More Array Operations

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

Table 2.4b Array Replication, Concatenation, Flipping, and Reshaping

Function	Description
`repmat`	Replicate an array
`horzcat`	Concatenate an array horizontally
`vertcat`	Concatenate an array vertically
`flipud`	Flip an array upside down
`fliplr`	Flip an array left-side right
`reshape`	Reshape an array

2.5 Sums, Products, Minima, and Maxima

Example02_05.m

[1] On the **Command Window**, type the following commands:

```
 1  >> clear, clc
 2  >> a = 1:5
 3  a =
 4       1     2     3     4     5
 5  >> b = sum(a)
 6  b =
 7      15
 8  >> c = cumsum(a)
 9  c =
10       1     3     6    10    15
11  >> d = prod(a)
12  d =
13     120
14  >> e = cumprod(a)
15  e =
16       1     2     6    24   120
17  >> f = diff(a)
18  f =
19       1     1     1     1
20  >> A = reshape(1:9, 3, 3)
21  A =
22       1     4     7
23       2     5     8
24       3     6     9
25  >> g = sum(A)
26  g =
27       6    15    24
28  >> B = cumsum(A)
29  B =
30       1     4     7
31       3     9    15
32       6    15    24
33  >> h = prod(A)
34  h =
35       6   120   504
36  >> C = cumprod(A)
37  C =
38       1     4     7
39       2    20    56
40       6   120   504
41  >> D = diff(A)
42  D =
43       1     1     1
44       1     1     1
```

Sums and Products of Vectors

[2] Let a_i, $i = 1, 2, \ldots, n$, be the elements of a vector **a** (either row vector or column vector). The output of function `sum` (line 5) is a scalar b, where

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

In this example (line 7),

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

The output of function `cumsum` (cumulative sum; line 8) is a vector **c** of n elements, where $c_1 = a_1$ and

$$c_i = c_{i-1} + a_i,\ i = 2, 3, \ldots, n$$

In this example (line 10),

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

The output of function `prod` (line 11) is a scalar d,

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

In this example (line 13),

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

The output of function `cumprod` (cumulative product; line 14) is a vector **e** of n elements, where $e_1 = a_1$ and

$$e_i = e_{i-1} \times a_i,\ i = 2, 3, \ldots, n$$

In this example (line 16),

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

The output of function `diff` (line 17) is a vector **f** of n-1 elements, where

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

In this example (line 19),

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

→

Sums and Products of Matrices

[3] Let A_{ij}, $i = 1, 2, \dots, n$, $j = 1, 2, \dots, m$ be the elements of an $n \times m$ matrix **A**. The output of function `sum` (line 25) is a row vector **g**, where

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

In this example (line 27),

$$g_1 = 1 + 2 + 3 = 6$$
$$g_2 = 4 + 5 + 6 = 15$$
$$g_3 = 7 + 8 + 9 = 24$$

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

The output of function `cumsum` (line 28) is an $n \times m$ matrix **B**, where $B_{1j} = A_{1j}$, $j = 1, 2, \dots m$, and

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

In this example (lines 30-32),

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

The output of function `prod` (line 33) is 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 35),

$$h_1 = 1 \times 2 \times 3 = 6$$
$$h_2 = 4 \times 5 \times 6 = 120$$
$$h_3 = 7 \times 8 \times 9 = 504$$

The output of function `cumprod` (line 36) is an $n \times m$ matrix **C**, where $C_{1j} = A_{1j}$, $j = 1, 2, \dots, m$, and

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

In this example (lines 38-40),

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

The output of function `diff` (line 41) is an $(n-1) \times m$ 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 43-44),

$$D_{11} = 2 - 1 = 1 \quad D_{12} = 5 - 4 = 1 \quad D_{13} = 8 - 7 = 1$$
$$D_{21} = 3 - 2 = 1 \quad D_{22} = 6 - 5 = 1 \quad D_{23} = 9 - 8 = 1$$

Example02_05.m (Continued)

[4] Continue to type the following commands:

```
45  >> p = min(a)
46  p =
47         1
48  >> q = max(a)
49  q =
50         5
51  >> r = min(A)
52  r =
53         1     4     7
54  >> s = max(A)
55  s =
56         3     6     9
```

Minima and Maxima

[5] The output of functions `min` or `max` for a vector are scalars (lines 45-50).

The output of functions `min` or `max` for an $n \times m$ matrix is a row vector of m elements (lines 51-56), in which each element is the minimum/maximum of the corresponding column. #

Table 2.5
Sums, Products, Minima, and Maxima

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

2.6 Arithmetic Operators

Example02_06a.m

[1] On the **Command Window**, type following commands:

```
 1   >> clear, clc
 2   >> A = reshape(1:6, 2, 3)
 3   A =
 4        1     3     5
 5        2     4     6
 6   >> B = reshape(7:12, 2, 3)
 7   B =
 8        7     9    11
 9        8    10    12
10   >> C = A+B
11   C =
12        8    12    16
13       10    14    18
14   >> D = A-B
15   D =
16       -6    -6    -6
17       -6    -6    -6
18   >> E = B'
19   E =
20        7     8
21        9    10
22       11    12
23   >> F = A*E
24   F =
25       89    98
26      116   128
27   >> a = [3, 6]
28   a =
29        3     6
30   >> b = a/F
31   b =
32     -13.0000   10.0000
33   >> c = b*F
34   c =
35        3     6
36   >> G = F^2
37   G =
38          19289       21266
39          25172       27752
```

Arithmetic Operators for Matrices

[2] Let $A_{ij},\ i=1,\ 2,\ \dots,\ n,\ j=1,\ 2,\ \dots,\ m$ be the elements of an $n\times m$ matrix **A**, and $B_{ij},\ i=1,\ 2,\ \dots,\ n,\ j=1,\ 2,\ \dots,\ m$ be the elements of another $n\times m$ matrix **B**. The addition (line 10) of the two matrices is an $n\times m$ matrix **C**,

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

In this example (lines 12-13),

$$C_{11}=1+7=8 \quad C_{12}=3+9=12 \quad C_{13}=5+11=16$$
$$C_{21}=2+8=10 \quad C_{22}=4+10=14 \quad C_{23}=6+12=18$$

The subtraction (line 14) of **B** from **A** is an $n\times m$ matrix **D**,

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

In this example (lines 16-17),

$$D_{11}=1-7=-6 \quad D_{12}=3-9=-6 \quad D_{13}=5-11=-6$$
$$D_{21}=2-8=-6 \quad D_{22}=4-10=-6 \quad D_{23}=6-12=-6$$

The transpose (line 18; also see 1.6[10], page 19) of **B** is an $m\times n$ matrix **E**,

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

In this example (lines 20-22)

$$E_{11}=B_{11}=7 \quad E_{12}=B_{21}=8$$
$$E_{21}=B_{12}=9 \quad E_{22}=B_{22}=10$$
$$E_{31}=B_{13}=11 \quad E_{32}=B_{23}=12$$

→

Arithmetic Operators for Matrices (Continued)

[3] The multiplication (line 23) of an $n \times m$ matrix **A** by an $m \times p$ matrix **E** is an $n \times p$ matrix **F**

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

In this example (lines 25-26), n = 2, m = 3, and p = 2, and the result is a 2×2 matrix:

$$\begin{aligned} F_{11} &= 1\times7+3\times9+5\times11 = 89 \\ F_{21} &= 2\times7+4\times9+6\times11 = 116 \\ F_{12} &= 1\times8+3\times10+5\times12 = 98 \\ F_{22} &= 2\times8+4\times10+6\times12 = 128 \end{aligned}$$

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

The division (line 30) 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

$$\mathbf{b} \times \mathbf{F} = \mathbf{a}$$

In this example (line 32), r = 1 and m = 2, and the result is a 1×2 row vector **b**,

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

since it satisfies

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

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 [9], for example).

The exponentiation of a square matrix is the repeated multiplication of the matrix itself. For example (line 36)

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

In this example (lines 38, 39)

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

Example02_06a.m (Continued)

[4] Continue to type following commands:

```
40   >> H = A.*B
41   H =
42          7    27    55
43         16    40    72
44   >> K = A./B
45   K =
46        0.1429    0.3333    0.4545
47        0.2500    0.4000    0.5000
48   >> M = A.^2
49   M =
50          1     9    25
51          4    16    36
```

Element-Wise Operators (Dot Operators)

[5] The element-wise multiplication (`.*` in line 40) 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,

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

In this example (lines 42, 43)

$$\begin{aligned} &H_{11} = 1\times7 = 7 \quad &H_{12} = 3\times9 = 27 \quad &H_{13} = 5\times11 = 55 \\ &H_{21} = 2\times8 = 16 \quad &H_{22} = 4\times10 = 40 \quad &H_{23} = 6\times12 = 72 \end{aligned}$$

The element-wise division (`./` in line 44) 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,

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

In this example (lines 46, 47)

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

The element-wise exponentiation (line 48) operates on an $n \times m$ matrix **A** and a scalar q, and the result is an $n \times m$ matrix **M**,

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

In this example (lines 50, 51)

$$\begin{aligned} &M_{11} = 1^2 = 1 \quad &M_{12} = 3^2 = 9 \quad &M_{13} = 5^2 = 25 \\ &M_{21} = 2^2 = 4 \quad &M_{22} = 4^2 = 16 \quad &M_{23} = 6^2 = 36 \end{aligned}$$

→

Example02_06a.m (Continued)

[6] Continue to type the following commands:

```
52   >> P = A+10
53   P =
54        11     13     15
55        12     14     16
56   >> Q = A-10
57   Q =
58        -9     -7     -5
59        -8     -6     -4
60   >> R = A*1.5
61   R =
62        1.5000    4.5000    7.5000
63        3.0000    6.0000    9.0000
64   >> S = A/2
65   S =
66        0.5000    1.5000    2.5000
67        1.0000    2.0000    3.0000
```

Operations Between a Matrix and a Scalar

[7] Let @ be one of the operators `+`, `-`, `*`, `/`, `.*`, `./`, or `.^`, and *s* be a scalar, then **A**@s is an $n \times m$ matrix **V**, where

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

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

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

For examples (lines 52-67)

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

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

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

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

Example02_06b.m

[8] Type the following commands:

```
68   >> clear, clc
69   >> a = 1:4
70   a =
71         1     2     3     4
72   >> b = 5:8
73   b =
74         5     6     7     8
75   >> c = a+b
76   c =
77         6     8    10    12
78   >> d = a-b
79   d =
80        -4    -4    -4    -4
81   >> e = a*(b')
82   e =
83        70
84   >> f = (a')*b
85   f =
86         5     6     7     8
87        10    12    14    16
88        15    18    21    24
89        20    24    28    32
90   >> g = a/b
91   g =
92        0.4023
93   >> h = a.*b
94   h =
95         5    12    21    32
96   >> k = a./b
97   k =
98       0.2000   0.3333   0.4286   0.5000
99   m = a.^2
100  m =
101        1     4     9    16
```

Arithmetic Operators for Vectors

[9] A vector is a special matrix, in which either the row size or the column size is equal to one. Thus, all the rules of the arithmetic operations for matrices apply to those for vectors (lines 68-101).

Note that, in lines 90-92, 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. →

Example02_06c.m

[10] Type the following commands:

```
102  >> clear, clc
103  a = 6
104  a =
105       6
106  >> b = 4
107  b =
108       4
109  >> c = a+b
110  c =
111      10
112  >> d = a-b
113  d =
114       2
115  >> e = a*b
116  e =
117      24
118  >> f = a/b
119  f =
120      1.5000
121  >> g = a^2
122  g =
123      36
124  >> h = a.*b
125  h =
126      24
127  >> k = a./b
128  k =
129      1.5000
130  >> m = a.^2
131  m =
132      36
```

Arithmetic Operators for Scalars

[11] A scalar is a 1x1 matrix. Thus, all the rules of the arithmetic operations for matrices can apply to those for scalars (lines 102-132).

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

Precedence Level of Operators

[12] 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 2.6, 2.7a, and 2.7b; the lower number has higher precedence level. For operators with the same precedence level, the evaluation is from left to right. The parentheses `()` has highest precedence level, while the assignment = has lowest precedence level. #

Names of Operators

[13] 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
```

and `5+6-4` is evaluated using two function calls

```
>> minus(plus(5,6),4)
ans =
     7
```

This feature is useful when creating data types (classes) and their associate operators. In Section 4.6, we'll demonstrate the creation of a data type of polynomial, for which we'll implement the addition and subtraction of polynomials using the operators `+` and `-`.

Table 2.6 lists each operator's corresponding function name, along with its precedence level (see [12]). #

Table 2.6 Arithmetic Operators

Operator	Name	Description	Precedence level
`+`	`plus`	Addition	5
`-`	`minus`	Subtraction	5
`*`	`mtimes`	Multiplication	4
`/`	`mrdivide`	Division	4
`^`	`mpower`	Exponentiation	2
`.*`	`times`	Element-wise multiplication	4
`./`	`rdivide`	Element-wise division	4
`.^`	`power`	Element-wise exponentiation	2
`-`	`uminus`	Unary minus	3
`+`	`uplus`	Unary plus	3

Details and More: Help>MATLAB>Language Fundamentals>Operators and Elementary Operations>Arithmetic Help>MATLAB>Language Fundamentals>Operators and Elementary Operations>Arithmetic>Concepts>Operator Precedence

2.7 Relational and Logical Operators

Example02_07.m

[1] On the **Command Window**, type following commands:

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

Relational Operators

[2] A relational operator (Table 2.7a; also see 2.3[6], page 59) always operates on two numeric values, and the result is a logical value. If either of the operands is not a numeric value, it is converted to a numeric value.

Logical Operators

[3] A logical operator (Table 2.7b; also see 2.3[8], page 60) operates on one or two logical values, and the result is a logical value. If any of the operands is not a logical value, it is converted to a logical value: a non-zero value is converted to a `true` and a zero value is converted to a `false`. →

Table 2.7a Relational Operators

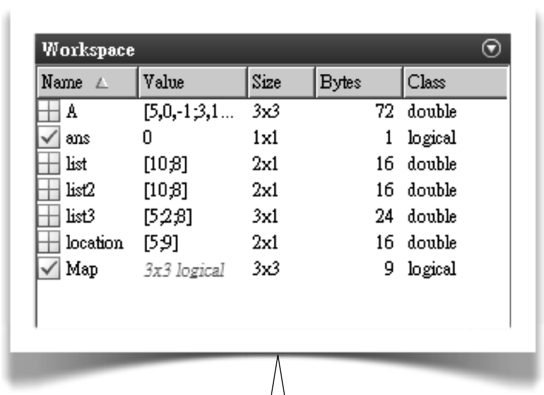
Operator	Description	Precedence level
`==`	Equal to	7
`~=`	No equal to	7
`>`	Greater than	7
`<`	Less than	7
`>=`	Greater than or equal to	7
`<=`	Less than or equal to	7
`isequal`	Determine array equality	1

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

Table 2.7b Logical Operators

Operator	Description	Precedence level
`&`	Logical AND	8
`|`	Logical OR	9
`~`	Logical NOT	3
`&&`	Logical AND (short-circuit)	10
`||`	Logical OR (short-circuit)	11

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



Workspace

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

[4] the **Workspace**.

[5] In line 7, the operation `A>6` creates a logical-value matrix the same size as `A` (lines 9-11). The logical matrix can be thought as a "map" indicating the locations of the elements that are greater than 6. Note that the parentheses in line 7 are not needed, since the assignment (=) has lowest precedence level (2.6[12], page 70). We sometimes add redundant parentheses to enhance the readability, avoiding confusion.

Function `find`

[6] Function `find` (line 12) takes a **logical array** as input argument and outputs a **column vector** of numbers that are the **linear indices** (1.8[14], page 25) of the elements with value `true` in the array.

Here (line 12), 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 14-15).

Just for fun: how about we type `find(A)`? Since `A` is assumed a logical array, it is converted to logical type; all the non-zero values are treated as `true`. The result is that the elements 1, 2, 5, 6, 7, 8, and 9 have the `true` value (lines 29-31). Note that, in line 29, we transpose the resulting column vector to a row vector to save output space.

[7] In line 16, the location vector is used to access the array `A`, the result is a column vector containing the numbers in `A` that are greater than 6 (lines 18-19).

If we are concerned only with the numbers themselves (not the locations), then the commands in lines 7, 12, and 16 can be combined, as shown in line 20.

More Involved Logical Expression

[8] 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 24.

Logical NOT

[9] The logical NOT reverses logical values, i.e., `true` becomes `false`, and `false` becomes `true`. If we apply it on a numerical array (line 32), a nonzero value becomes `false` and a zero value becomes `true` (lines 34-36).

If we apply the logical NOT again on the previous results (line 37), the outcome is, of course, a logical array. Here, we want to emphasize that, for a numerical array, in general,

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

This is demonstrated in lines 42-44. The function `isequal` (line 42) compares two arrays and outputs `true` if they are equal in size and contents, otherwise outputs `false`.

Short-Circuit Logical AND and OR

[10] 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 2.7b).

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. #

2.8 String Manipulations

Example02_08a.m: String Manipulations

[1] Create a program like this, save as Example02_08a.m, and run the program. Input your name and age as shown in [2].

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

[2] This is the screen input and output of Example02_08a.m

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

[3] The `echo` command (line 1) controls the display (echoing) of statements on the **Command Window**. The default is `off`. We insert this command to make sure the statements are not repeated in the screen.

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

Function `disp` displays a data of any type (lines 5 and 23). A newline character (`\n`) is automatically added at the end of the display. On the other hand, you need to append newline characters when using function `fprintf`.

[5] Line 6 requests a data input from the screen. Without the `'s'` as the second argument, you would have to use the single quotes (`''`) to input a string. With the `'s'` as the second argument, you can omit the single quotes (line 24).

By default, the input data without single quotes is assumed a `double` (lines 7 and 25).

[6] Function `sprintf` (line 10) writes formatted data to a string and outputs the string. Here, we display the the value of π with 8 digits after the decimal point (lines 10, 11, and 27).

[7] In lines 12-15, the statement continues for 4 lines, without using ellipses (`...`). A newline character between a pair of square brackets is treated as a semicolon. Thus, this statement creates 3 rows of strings; it is a 3-by-7 matrix of `char`.

Rows of a matrix must have the same length. That's why we pad the first two strings with trailing blanks to make the lengths of the three strings equal. →

[8] Another way to create a column of strings (i.e., a matrix of `char`) is using function `char`. Line 16 creates a column of strings exactly the same as that in lines 12-15. Function `char` automatically pads the strings with trailing blanks to make the the lengths of the strings equal.

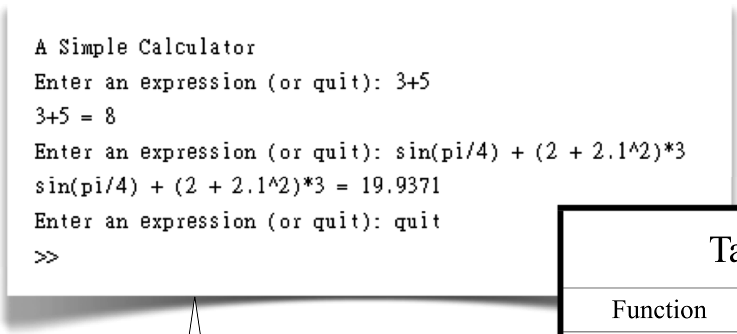
Lines 17-19 and 28 confirm the equality of the two matrices.

[9] Function `deblank` (line 20) removes trailing blanks from a string. `Names1(2,:)` means the entire 2nd row (i.e., the string `'John    '`). After removing the trailing blanks, four characters remain in the string (lines 21-22, 29).

Example02_08b.m: A Simple Calculator

[10] This program uses function `eval` to create a simple calculator. Type and run the program (see [11]). Study each statement.

```
clear, clc, echo off
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
```



```
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
>>
```

[11] This is a test run of the program Example02_08b.m. #

Table 2.8 String Manipulations

Function	Description
`char`	Create character arrays
`disp`	Write a data on the screen
`input`	Read data from the screen
`sprintf`	Write formatted data to a string
`num2str`	Convert number to string
`str2num`	Convert string to number
`eval`	Evaluate a MATLAB expression
`deblank`	Remove trailing blanks from a string
`strtrim`	Remove leading and trailing blanks from a string
`strcmp`	Compare two strings (case sensitive)
`Strcmpi`	Compare two strings (case insensitive)

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

2.9 Expressions

Law of Sines

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

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

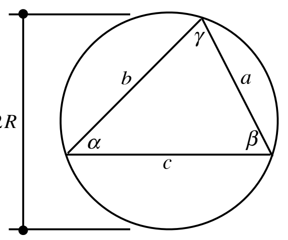
where α, β, and γ are the angles of a triangle; *a*, *b*, and *c* are the lengths of sides opposite to the respective angles; *A* is the area of the triangle; *R* is the radius of the circumscribed circle of the triangle:

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

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

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

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



Expressions

[4] An expression is a syntactic combination of **data** (constants or variables; scalars, vectors, matrices, or higher-dimensional arrays), **functions** (built-in or user-defined), **operators** (arithmetic, relational, or logical), and **special characters** (see Table 2.9a, next page). An expression always results in a **data**.

Table 2.9b (next page) lists some elementary math functions that are frequently used.

About Example02_09a.m

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

Line 9 verifies that the sum of the three angles is indeed 180 degrees (lines 20-21). Lines 10-11 calculate the area using two different formulas in [1], and they indeed have the same output values (lines 22-25). →

Example02_09a.m: Law of Sines

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

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

[3] This is the screen output of Example02_09a.m.

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

Example02_09b.m: Law of Cosines

[6] The law of cosines states (see *Wikipedia>Trigonometry;* use the same notations in [1]):

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

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

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

```
alpha =
   44.4153
beta =
   57.1217
gamma =
   78.4630
```

[7] This is the screen output of Example02_09b.m. The output is consistent with that in [3]. #

Table 2.9a
Special Characters for Expressions

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

Details and More:
Help>MATLAB>Language Fundamentals>Special Characters>Special Characters>Special Characters

Table 2.9b
Some Elementary Math Functions

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

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

2.10 Example: Function Approximation

Taylor Series

[1] In the hardware level, your computer can only perform simple arithmetic calculations such as addition, subtraction, multiplication, division, etc. Evaluation of a function value such as sin(x) 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 this way of thinking when using MATLAB expressions.

The sine function can be approximated using a Taylor series (also see Section 9.6):

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$

The more terms, the more accurate the approximation.

Example02_10a.m: Scalar Expressions

[2] This script calculates the value $\sin(\pi/4)$ using the Taylor series in [1].

```
 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
```

[3] This is the output of Example02_10a.m.

[7, 10] This is also the output of Example02_10b.m or Example02_10c.m. →

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

About Example02_10a.m

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

Using For-Loop

[5] In theory, representing a function such as sin(x) requires infinite number of terms of polynomials. We need a general representation of these terms. We may rewrite the Taylor series in [1] as follows:

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

We now use a for-loop (1.14[4], page 33; also see Section 3.4) to calculate $\sin(\pi/4)$.

Example02_10b.m: Using For-Loop

[6] Type and run this program. The screen output [7] is the same as that of Example02_10a.m. →

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

About Example02_10b.m

[8] In line 18, variable `sinx` is initialized to zero. The statement within the for-loop (line 20) runs four passes. In each pass, variable `k` is assigned 1, 2, 3, and 4, respectively, and a term is calculated and added to variable `sinx`. At the completion of the for-loop, variable `sinx` has the function value (lines 23, 11-12).

Example02_10c.m: Vector Expressions

[9] Using vector expressions, this script produces the same output [10] as that of Example02_10b.m. ←

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

About Example02_10c.m

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

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

Use `k = [1,2,3,4]`, follow the descriptions of element-wise operations (2.6[5], page 68) and function `sum` for vectors (2.5[2], page 65), and we have

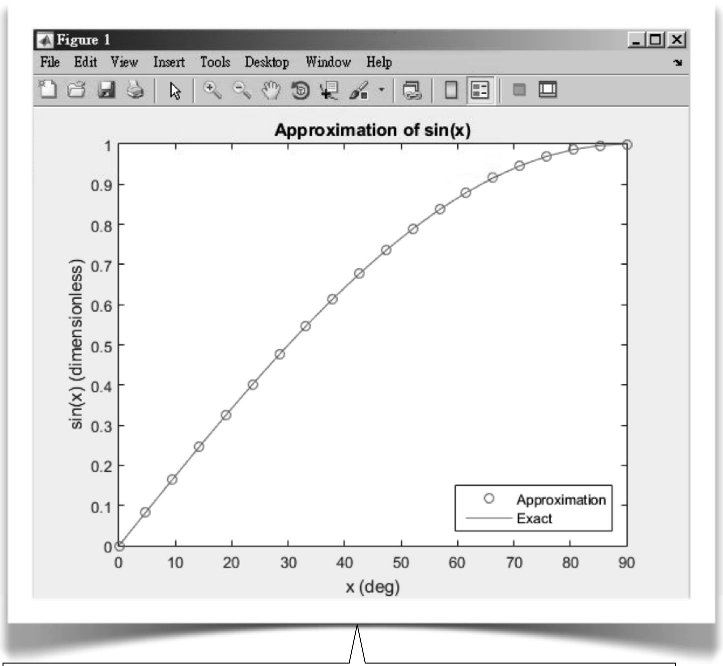
```
step1 = [0,1,2,3]
Step2 = (-1).^[0,1,2,3] = [1,-1,1,-1]
step3 = [1,3,5,7]
step4 = x.^[1,3,5,7] = [x,x^3,x^5,x^7]
step5 = [1,-1,1,-1].*[x,x^3,x^5,x^7] = [x,-x^3,x^5,-x^7]
step6 = factorial([1,3,5,7]) = [1,6,120,5040]
step7 = [x,-x^3,x^5,-x^7]./[1,6,120,5040] = [x,-x^3/6,x^5/120,-x^7/5040]
step8 = x-x^3/6+x^5/120-x^7/5040
sinx = x-x^3/6+x^5/120-x^7/5040
```

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

Example02_10d.m: Matrix Expressions

[12] Using matrix expressions, this script calculates sin(*x*) for various *x* values and produces a graphic output like [13].

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



[13] Example02_10d.m plots an approximation of `sin(x)` for various `x` values (circular marks) and a curve (solid line) representing the exact function values.

The Matrix Expression in Line 37

[15] Analysis of line 37 is similar to that of line 29 (see [11], last page). However, now we're dealing with matrices. The argument of the function `sum` is 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 (2.5[3], page 66), resulting a row vector of 20 values.

Function `meshgrid`

[14] Line 33 creates a row vector of 20 angle values using function `linspace` (2.4[5], page 61). Line 35 creates a row vector of four integers.

Line 36 generates two matrices `X` and `K`, using the information in vectors `x` and `k`. This statement 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` is used in lines 5-6 of 1.6[1] (page 18) and explained in 1.6[9, 10] (page 19). Actually, lines 5-6 of 1.6[1] can be replaced by the following statement:

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

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

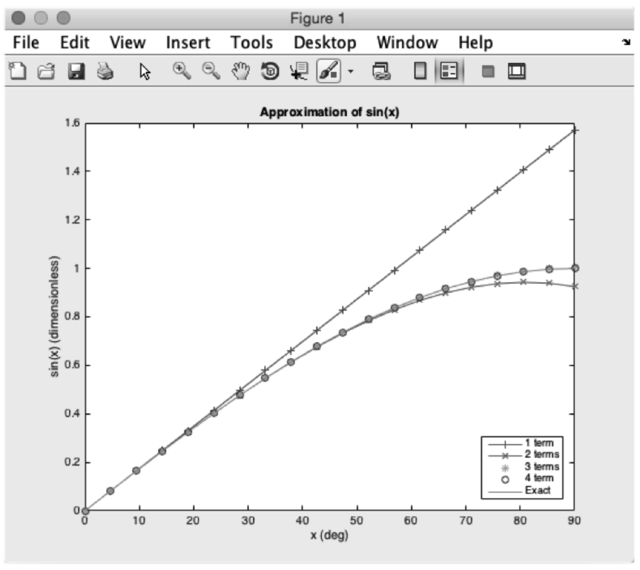
[16] Line 38 plots the 20 values with circular marks, along with the exact function values, by default, using a solid line.

Line 42 adds legends (Section 5.7) on the lower-right corner of the graphic area. →

Example02_10e.m: Multiple Curves

[17] This script plots four approximated curves and an exact curve of sin(*x*) as shown in [18], the four approximated curves corresponding to Taylor series of 1, 2, 3, and 4 items, respectively.

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



[18] Example02_10e.m plots four approximated curves and an exact curve of sin(*x*), for comparison.

[19] Line 48 looks like line 37 except function `cumsum` (2.5[3], page 66) is used instead of `sum`. The function `cumsum` calculates the cumulative sums of the four values in each column, resulting in a 4-by-20 matrix, the kth row representing the approximated function values when `k` terms are used. Lines 49-53 plot the four approximated curves, each a row of the function values, and the exact curve.

Line Styles and Marker Types

[20] In line 49, the notation `'+-'` specifies a plus marker and a solid line style. Similarly, in line 50, `'x-'` specifies a cross marker and a solid line style. Table 5.5a (page 169) lists various line styles and marker types. #

2.11 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 two-dimensional, cartesian coordinate system:

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

subject to 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

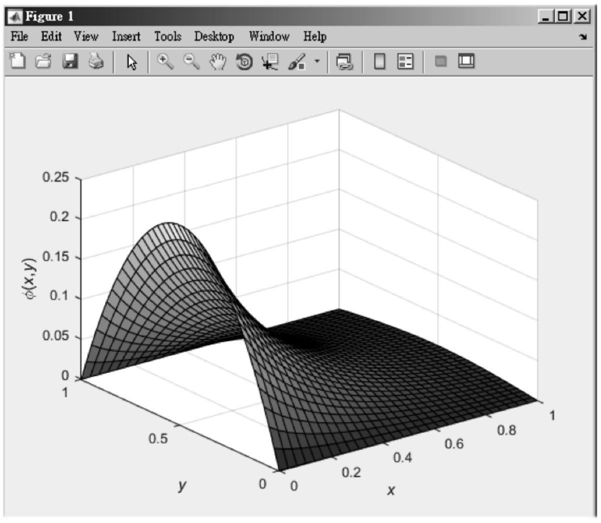
$$\phi(x,y)=4\sum_{k=1}^{\infty}\frac{1-\cos(k\pi)}{(k\pi)^3}e^{-k\pi x}\sin(k\pi y)$$

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

Example02_11.m: Series Solution of a Laplace Equation

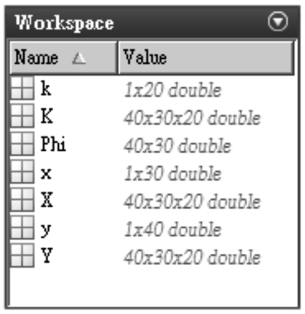
[2] This script calculates the solution $\phi(x,y)$ according to [1] and plots a three-dimensional surface $\phi=\phi(x,y)$ [3].

```
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 of the Laplace equation in [1]

[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 Arrays Created with `meshgrid`

[5] Line 5 generates three 40x30x20 arrays `X`, `Y`, and `K` (see [4], last page), using the information in vectors `x`, `y`, and `k`. This statement is equivalent to the following three 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 4th statement reshapes the vector `k` to a $1 \times 1 \times nk$ vector, which is called a **page-vector** similar to that a $1 \times m$ matrix is called a row-vector and an $m \times 1$ matrix is called a column-vector. Note that `X` varies in column-dimension while it keeps constant in both row-dimension and page-dimension; `Y` varies in row-dimension while it keeps constant in both column-dimension and page-dimension; `K` varies in page-dimension while it keeps constant in both row-dimension and column-dimension.

3-D Array Expressions (Line 6)

[6] While the expression in line 37 of Example02_10d.m (page 79) is a matrix expression, the expression in line 6 is a 3D extension of matrix expressions.

By default, function `sum` sums up the values along the first dimension (i.e., the row-dimension, see 2.5[3], page 66); however, you may specify the dimension along which the summing is performed. Here, the second argument of the function `sum` specifies that the summing is along the third-dimension (i.e., the page-dimension).

Function `surf`

[7] In 1.6[13] (page 19), function `surf` takes three matrices as input arguments. Here, in line 7, 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

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

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

The character sequence for some other Greek letters and math symbols are listed in Table 5.6a (page 173). #

2.12 Example: Deflection of Beams

Simply Supported Beams

[1] Consider a simply supported beam subject to a force *F* [2]. 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, page 100*)

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

where

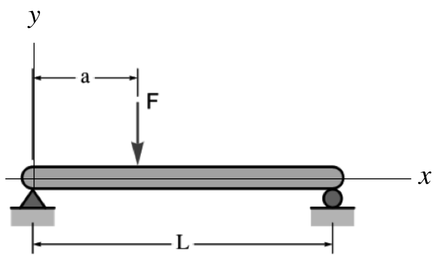
$$\theta = \frac{Fa}{6EIL}(2L-a)(L-a),\ R = \frac{F}{L}(L-a),\ I = \frac{wh^3}{12}$$

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

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

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

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

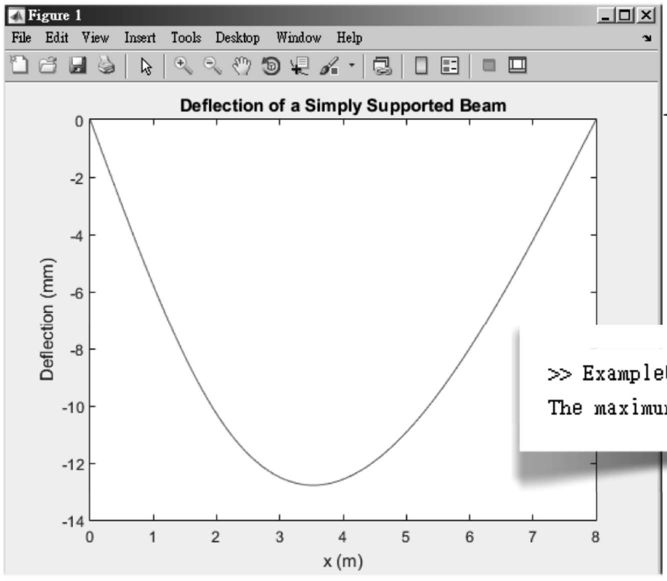


[2] This is the simply supported beam considered in this section. (*Figure source: https://en.wikipedia.org/wiki/File:Simple_beam_with_offset_load.svg, by Hermanoere*)

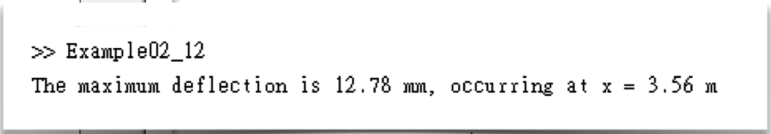
Example02_12.m: Deflection of Beams

[3] This script generates a graphic output as shown in [4] and a text output as shown in [5]. →

```
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('The maximum deflection is %.2f mm, occurring at x = %.2f m\n', ...
   ymax*1000, x(index))
```



[4] This program plots a curve representing the deflection of the simply supported beam in [2].



```
>> Example02_12
The maximum deflection is 12.78 mm, occurring at x = 3.56 m
```

[5] Screen output of the program.

Logical Operators in Numeric Expressions

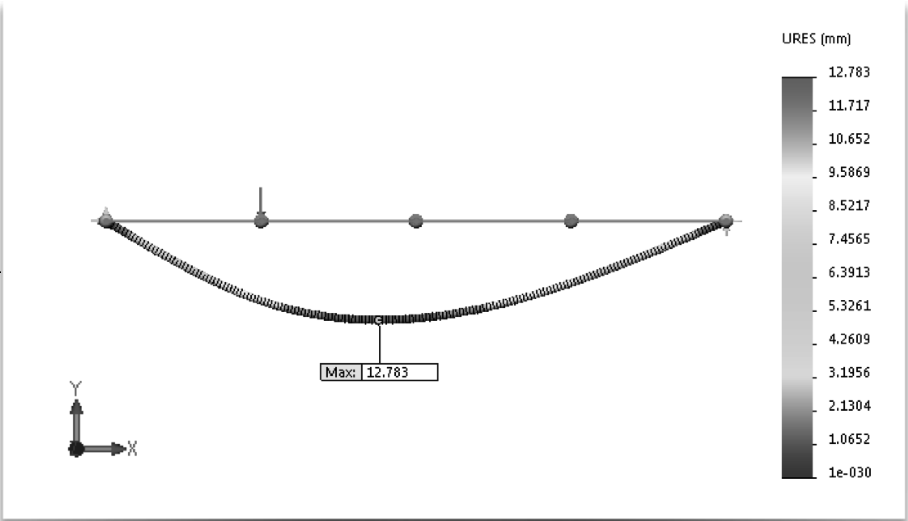
[6] The function

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

is implemented (see line 12) using a logical operator

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

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



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

2.13 Example: Vibrations of Supported Machines

Free Vibrations of a Supported Machine

[1] The mass-spring-damper model in [2] represents a machine supported by a layer of elastic, energy-absorbing material. In [2], m is the mass of the machine; k is the spring constant of the support, i.e., $F_s = -kx$, where F_s is the elastic force acting on the machine and x is the displacement of the machine; c is the damping constant, i.e., $F_d = -c\dot{x}$, where F_d is the damping force acting on the machine and $\dot{x}$ is the velocity of the machine. (See *Wikipedia>Damping*.)

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

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

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

with the initial conditions

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

Undamped Free Vibrations

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

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

with the initial conditions described in [1]. The solution for the equation is

$$x(t) = \delta \cos \omega t$$

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

$$\omega = \sqrt{\frac{k}{m}}$$

This solution can be verified by substituting it to the differential equation and the boundary conditions.

The natural period (with SI unit s) is then

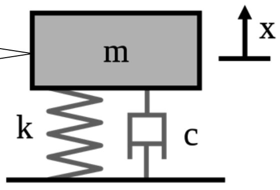
$$T = \frac{2\pi}{\omega}$$

Example02_13a.m calculates and plots the solution, using the following data

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

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

[2] In this section, we use this mass-spring-damper model to represent a machine supported by a layer of elastic, energy-absorbing material. (*Figure source: https://commons.wikimedia.org/wiki/File:Mass_spring_damper.svg, by Pbroks13*)



Example02_13a.m: Undamped Free Vibrations

[4] Type and run this program. See [5] for the graphic output. →

```
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)')
```

Damped Free Vibrations

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

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

with the initial conditions described in [1]. 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

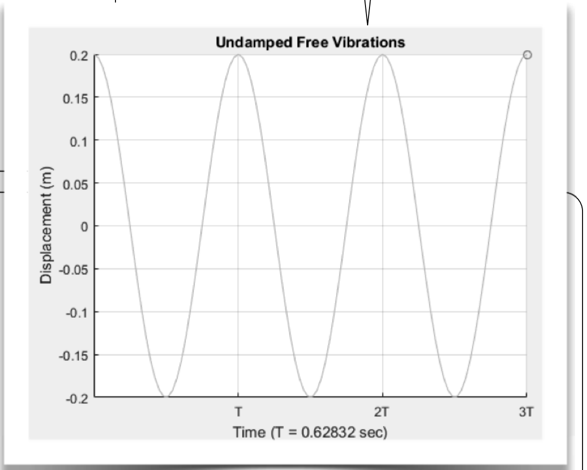
$$x(t)\ = \delta e^{-\frac{ct}{2m}}(\cos\omega_d t + \frac{c}{2m\omega_d}\sin\omega_d t)$$

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

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

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

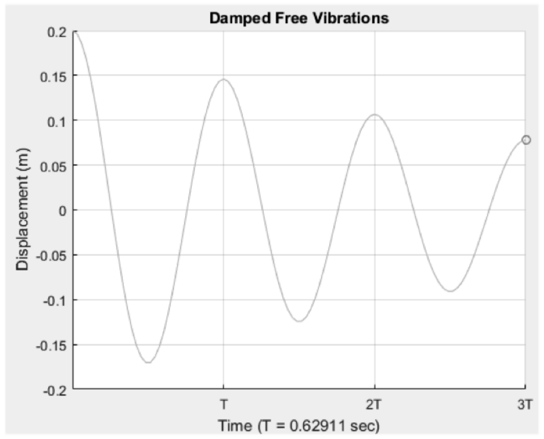
[5] This is the output of Example02_13a.m. Without damping effects, the machine vibrates forever; i.e., the amplitudes never fade away.



Example02_13b.m: Damped Free Vibrations

[7] This program plots the solution described in [6]. →

```
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)')
```



[8] The output of Example02_13b.m. Due to the inclusion of the damping effects, the vibrations gradually fade away.

Harmonically Forced Vibrations

[9] Now, assume there is a rotating part in the machine and, due to eccentric rotations, the rotating part generates an up-and-down harmonic force $F \sin \omega_f t$ on the support, where ω_f is the frequency of the harmonic force and F is the amplitude of the harmonic forces. 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 differential equation

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

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

$$x(t) = x_m \sin(\omega_f t - \varphi)$$

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

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

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

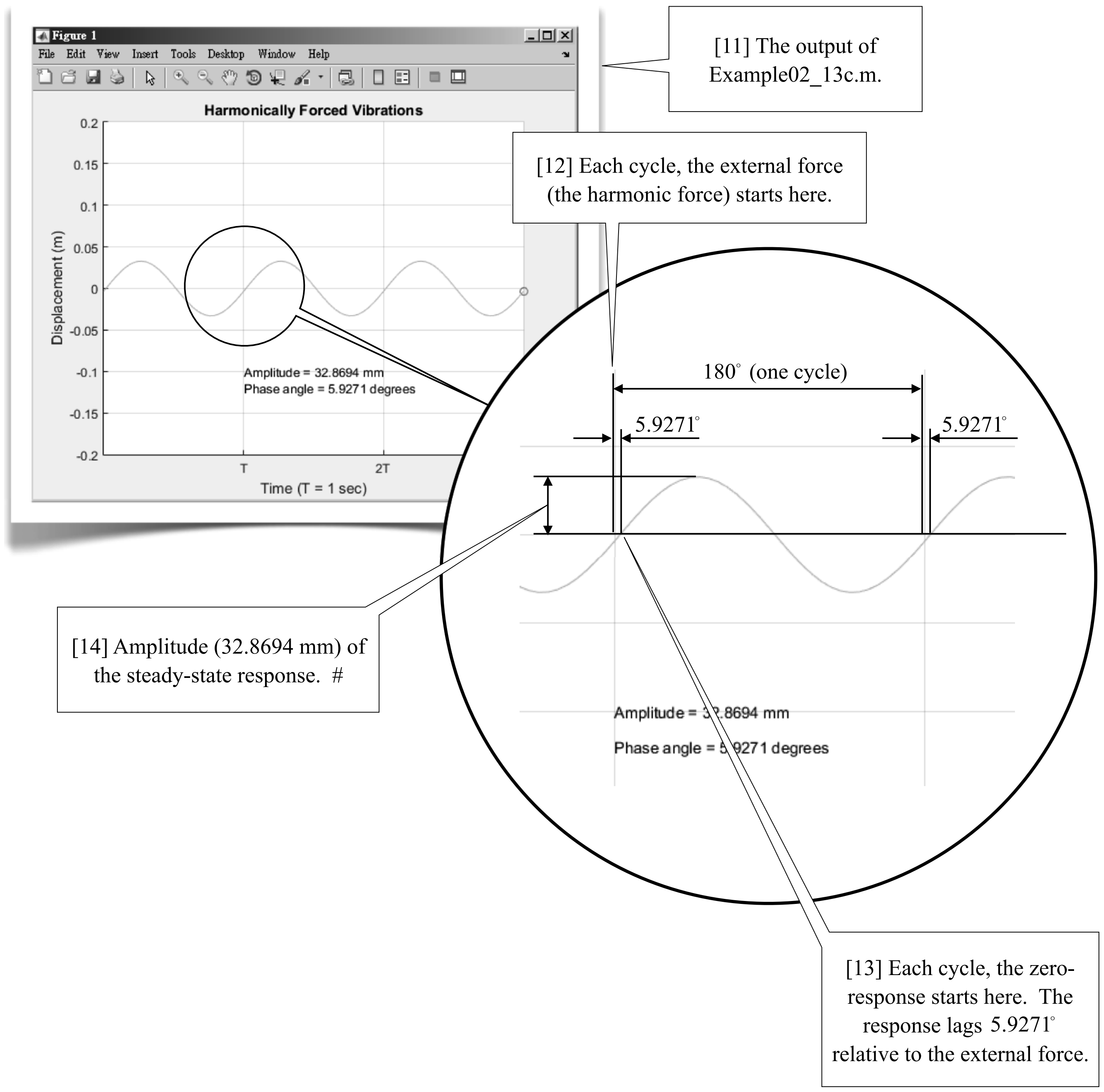
Example02_13c.m: Forced Vibrations

[10] This program plots the solution described in [9]. See the output in [11-14], next page. →

```
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'])
```



[11] The output of Example02_13c.m.

[12] Each cycle, the external force (the harmonic force) starts here.

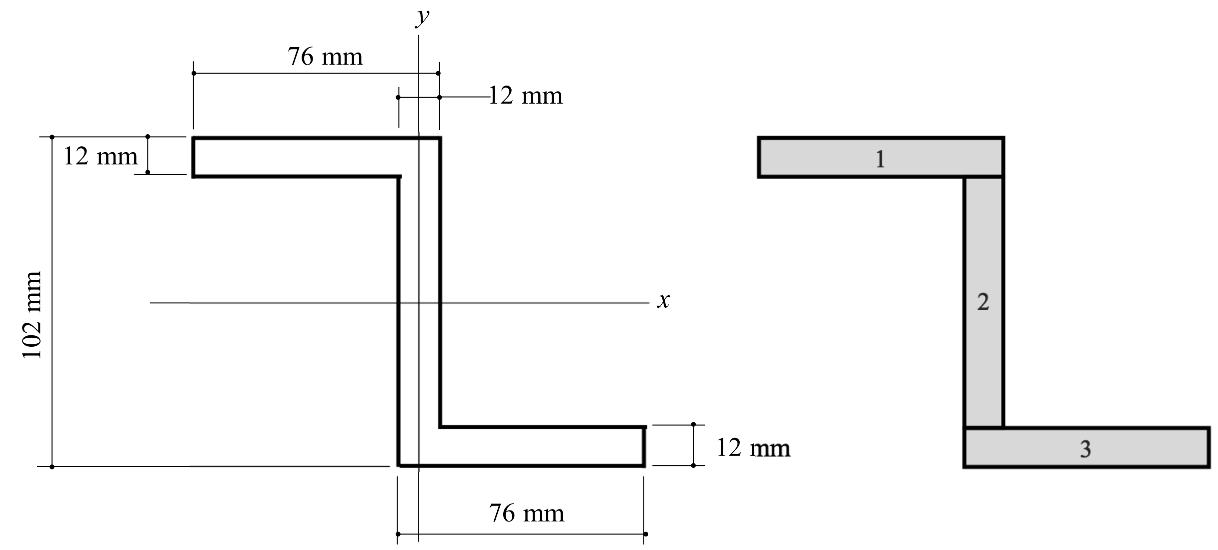
[13] Each cycle, the zero-response starts here. The response lags 5.9271° relative to the external force.

[14] Amplitude (32.8694 mm) of the steady-state response. #

2.14 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 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 $C_4^{100} = 3{,}921{,}225$.

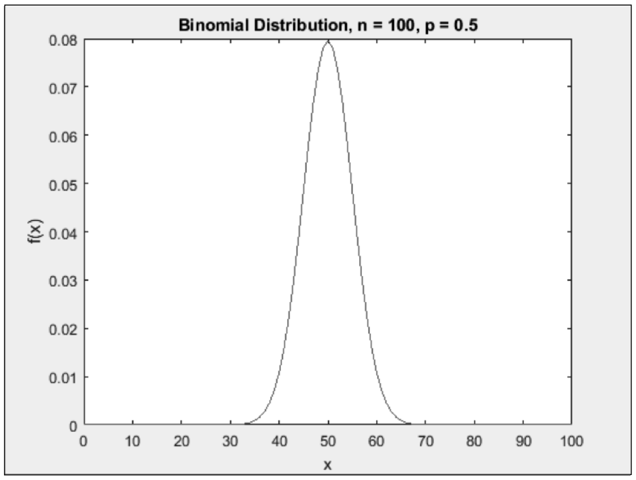
Problem02_03: Binomial Distribution

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 the last problem. 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

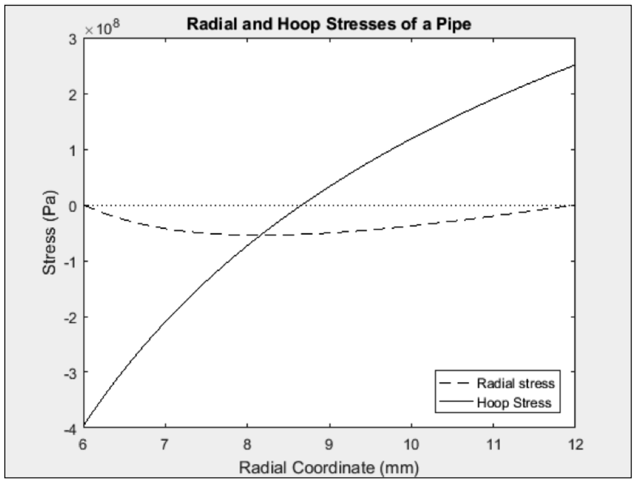
The radial stress σ_r and hoop stress σ_h in a long pipe due to a temperature T_a at its inner surface of radius *a* and a temperature T_b at its outer surface of radius *b* are, respectively, (*A. H. Burr and J. B. Cheatham, Mechanical Analysis and Design, 2nd ed., Prentice Hall, p. 496.*)

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

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

where *r* is the radial coordinate of the pipe, *E* is the Young's modulus, ν is the Poisson's ratio, and α is the coefficient of thermal expansion.

Write a script that allows the user to input the values of *a*, *b*, T_a and T_b, and generates a σ_r-versus-*r* curve and a σ_h-versus-*r* curve as shown right, which uses the following data: $a = 6$ mm, $b = 12$ mm, $T_a = 260^\circ C$, $T_b = 150^\circ C$, $E = 206$ GPa, $\nu = 0.3$, $\alpha = 2\times10^{-5}\ /^\circ$C.

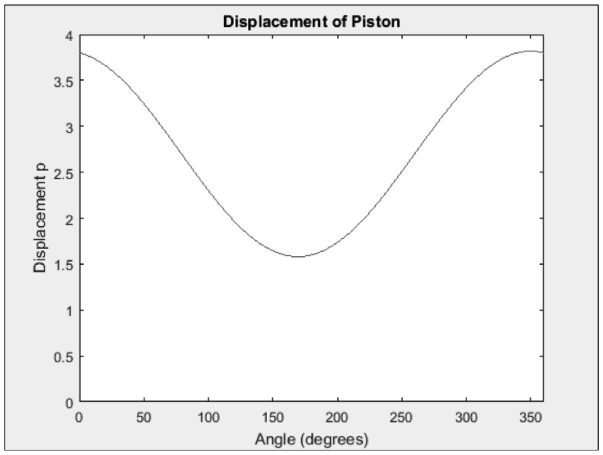


Problem02_05: Displacement of a Piston

The displacement p of the piston shown in 6.3[3-7] (page 197) 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:

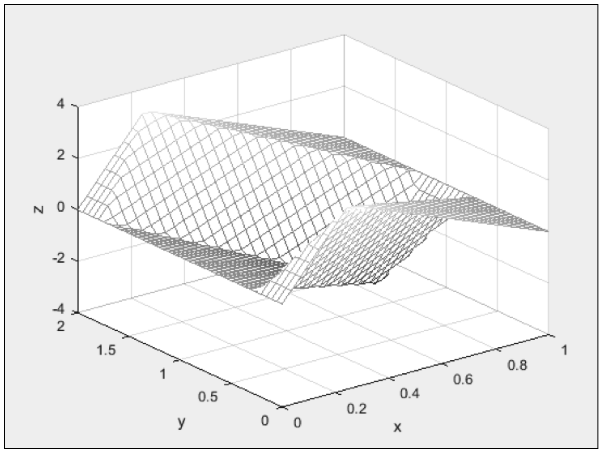
$$\pi = \sum_{k=0}^{n} \left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6} \right) \left(\frac{1}{16} \right)^k$$

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

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



Problem02_08: Double-Precision Floating-Point Format

A double-precision real number is stored with 64 bits (see 2.1[4], page 54); the details are exemplified in 2.1[13], page 56. Write a script that allows the user to input a real number and prints the 64-bit pattern on the screen. For example, if the user input the real number 28, the screen output should be:

```
0100000000111100000000000000000000000000000000000000000000000000
```

One way to accomplish this is to use function `typecast` to "cast" the 64-bit real number into an `uint64` integer, and then use function `dec2bin` to convert the `uint64` to a binary number in string.