CHAPTER 1

Introduction to MATLAB

KEY TERMS

prompt
programs
script files
toolstrip
variable
assignment statement
assignment operator
user
initializing
incrementing
decrementing
identifier names
reserved words
keywords
mnemonic
types
classes
double precision
floating point
unsigned
characters
character vectors
strings
default
continuation operator
ellipsis
unary
operand
binary
scientific notation
exponential notation
precedence
associativity
nested parentheses
inner parentheses
help topics
call a function
arguments
returning values
tab completion
constants
random numbers
seed
pseudorandom
open interval
global stream
character encoding
character set
relational expression
Boolean expression
logical expression
relational operators
logical operators
scalars
short-circuit operators
truth table
commutative
roundoff errors
range
casting
type casting
saturation arithmetic
locale setting
logarithm
common logarithm
natural logarithm

CONTENTS

1.1 Getting Into MATLAB 4
1.2 The MATLAB Desktop Environment ..5
1.3 Variables and Assignment Statements 6
1.4 Numerical Expressions .11
1.5 Characters and Strings ...18
1.6 Relational Expressions .19
1.7 Type Ranges and Type Casting 23
1.8 Built-in Numerical Functions 26
1.9 Alternate MATLAB Platforms 28
Summary 30
Common Pitfalls 30
Programming Style Guidelines ..30

MATLAB® is a very powerful software package that has many built-in tools for solving problems and developing graphical illustrations. The simplest method for using the MATLAB product is interactively; an expression is entered by the user and MATLAB responds immediately with a result. It is also possible to write

MATLAB®. https://doi.org/10.1016/B978-0-12-815479-3.00001-5

scripts and programs in MATLAB, which are essentially groups of commands that are executed sequentially.

This chapter will focus on the basics, including many operators and built-in functions that can be used in interactive expressions.

1.1 GETTING INTO MATLAB

MATLAB is a mathematical and graphical software package with numerical, graphical, and programming capabilities. It includes an integrated development environment, as well as both procedural and object-oriented programming constructs. It has built-in functions to perform many operations, and there are Toolboxes that can be added to augment these functions (e.g., for signal processing). There are versions available for different hardware platforms, in both professional and student editions. The MathWorks releases two versions of MATLAB annually, named by the year and 'a' or 'b'. This book covers the releases through Version R2018a. In cases in which there have been changes in recent years, these are noted.

When the MATLAB software is started, a window opens in which the main part is the Command Window (see Figure 1.1). In the Command Window, you should see:

```
>>
```

The `>>` is called the ***prompt***. In the Student Edition, the prompt instead is: `EDU>>`

In the Command Window, MATLAB can be used interactively. At the prompt, any MATLAB command or expression can be entered, and MATLAB will respond immediately with the result.

It is also possible to write ***programs*** in MATLAB that are contained in ***script files*** or MATLAB code files. Programs will be introduced in Chapter 3.

The following commands can serve as an introduction to MATLAB and allow you to get help:

- **demo** will bring up MATLAB Examples in the Help Browser, which has examples of some of the features of MATLAB
- **help** will explain any function; **help help** will explain how help works
- **lookfor** searches through the help for a specific word or phrase (Note: this can take a long time)
- **doc** will bring up a documentation page in the Help Browser

To exit from MATLAB, either type **quit** or **exit** at the prompt, or click on the (red) "x".

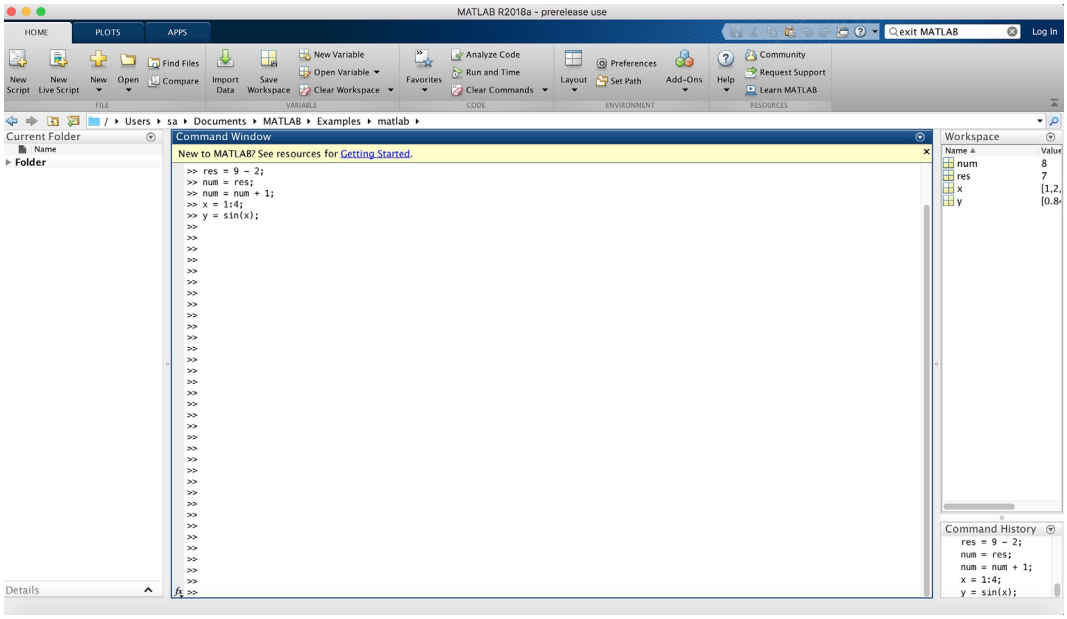

FIGURE 1.1
MATLAB command window.

1.2 THE MATLAB DESKTOP ENVIRONMENT

In addition to the Command Window, there are several other windows that can be opened and may be opened by default. What is described here is the default layout for these windows in Version R2018a, although there are other possible configurations. Different versions of MATLAB may show other configurations by default, and the layout can always be customized. Therefore, the main features will be described briefly here.

To the left of the Command Window is the Current Folder Window. The folder that is set as the Current Folder is where files will be saved. This window shows the files that are stored in the Current Folder. These can be grouped in many ways, for example by type, and sorted, for example by name. If a file is selected, information about that file is shown on the bottom where it says "Details."

To the right of the Command Window are the Workspace Window on top and the Command History Window on the bottom. The Command History Window shows commands that have been entered, not just in the current session (in the current Command Window), but previously as well. The Workspace Window will be described in the next section.

This default configuration can be altered by clicking the down arrow at the top right corner of each window. This will show a menu of options (different for each window), including, for example, closing that particular window and undocking that window. Once undocked, bringing up the menu and then clicking on the curled arrow pointing to the lower right will dock the window again. To make any of these windows the active window, click the mouse in it. By default, the active window is the Command Window.

The Desktop has a ***toolstrip***. By default, three tabs are shown ("HOME", "PLOTS", and "APPS"), although another, "SHORTCUTS", can be added.

Under the "HOME" tab, there are many useful features, which are divided into functional sections "FILE", "VARIABLE", "CODE", "ENVIRONMENT", and "RESOURCES" (these labels can be seen on the very bottom of the grey toolstrip area). For example, under "ENVIRONMENT", hitting the down arrow under Layout allows for customization of the windows within the Desktop Environment. Other toolstrip features will be introduced in later chapters when the relevant material is explained.

1.3 VARIABLES AND ASSIGNMENT STATEMENTS

To store a value in a MATLAB session, or in a program, a ***variable*** is used. The Workspace Window shows variables that have been created and their values. One easy way to create a variable is to use an ***assignment statement***. The format of an assignment statement is

```
variablename = expression
```

The variable is always on the left, followed by the = symbol, which is the ***assignment operator*** (unlike in mathematics, the single equal sign does *not* mean equality), followed by an expression. The expression is evaluated and then that value is stored in the variable. Here is an example and how it would appear in the Command Window:

```
>> mynum = 6
mynum =
     6
>>
```

Here, the ***user*** (the person working in MATLAB) typed "mynum = 6" at the prompt, and MATLAB stored the integer 6 in the variable called *mynum*, and then displayed the result followed by the prompt again. As the equal sign is the assignment operator, and does not mean equality, the statement should be read as "mynum gets the value of 6" (*not* "mynum equals 6").

Note that the variable name must always be on the left, and the expression on the right. An error will occur if these are reversed.

```
>> 6 = mynum
   6 = mynum
     ↑
Error: Incorrect use of '=' operator. To assign a value to a variable,
use '='. To compare values for equality, use '=='.
>>
```

Putting a semicolon at the end of a statement suppresses the output. For example,

```
>> res = 9 - 2;
>>
```

This would assign the result of the expression on the right side, the value 7, to the variable *res*; it just does not show that result. Instead, another prompt appears immediately. However, at this point in the Workspace Window, both the variables *mynum* and *res* and their values can be seen.

Note that in the remainder of the book, the prompt that appears after the result will not be shown.

The spaces in a statement or expression do not affect the result, but make it easier to read. The following statement, which has no spaces, would accomplish exactly the same result as the previous statement:

```
>> res = 9-2;
```

MATLAB uses a default variable named *ans* if an expression is typed at the prompt and it is not assigned to a variable. For example, the result of the expression 6 + 3 is stored in the variable *ans*:

```
>> 6 + 3
ans =
      9
```

This default variable, *ans*, is reused any time that only an expression, not an assignment statement, is typed at the prompt. Note that it is not a good idea to use *ans* as a name yourself or in expressions.

A shortcut for retyping commands is to hit the up arrow ↑, which will go back to the previously typed command(s). For example, if you decide to assign the result of the expression 6 + 3 to a variable named *result* instead of using the default variable *ans*, you could hit the up arrow and then the left arrow to modify the command rather than retyping the entire statement:

```
>> result = 6 + 3
result =
      9
```

This is very useful, especially if a long expression is entered and it contains an error, and it is desired to go back to correct it.

It is also possible to choose command(s) in the Command History window and rerun them by right-clicking. Consecutive commands can be chosen by clicking on the first or last and then holding down the Shift and up or down arrows.

To change a variable, another assignment statement can be used, which assigns the value of a different expression to it. Consider, for example, the following sequence of statements:

```
>> mynum = 3
mynum =
          3
>> mynum = 4 + 2
mynum =
          6
>> mynum = mynum + 1
mynum =
          7
```

In the first assignment statement, the value 3 is assigned to the variable *mynum*. In the next assignment statement, *mynum* is changed to have the value of the expression 4+2, or 6. In the third assignment statement, *mynum* is changed again, to the result of the expression *mynum + 1*. Since at that time *mynum* had the value 6, the value of the expression was 6+1, or 7.

At that point, if the expression *mynum+3* is entered, the default variable *ans* is used since the result of this expression is not assigned to a variable. Thus, the value of *ans* becomes 10, but *mynum* is unchanged (it is still 7). Note that just typing the name of a variable will display its value (the value can also be seen in the Workspace Window).

```
>> mynum + 3
ans =
       10
>> mynum
mynum =
        7
```

1.3.1 Initializing, Incrementing, and Decrementing

Frequently, values of variables change, as shown previously. Putting the first or initial value in a variable is called ***initializing*** the variable.

Adding to a variable is called ***incrementing***. For example, the statement

```
mynum = mynum + 1
```

increments the variable *mynum* by 1.

QUICK QUESTION!

How can 1 be subtracted from the value of a variable called *num*?

This is called ***decrementing*** the variable.

Answer:

```
num = num - 1;
```

1.3.2 Variable Names

Variable names are an example of ***identifier names***. We will see other examples of identifier names, such as function names, in future chapters. The rules for identifier names are as follows.

- The name must begin with a letter of the alphabet. After that, the name can contain letters, digits, and the underscore character (e.g., *value_1*), but it cannot have a space.
- There is a limit to the length of the name; the built-in function **namelengthmax** tells what this maximum length is (any extra characters are truncated).
- MATLAB is case-sensitive, which means that there is a difference between upper- and lowercase letters. So, variables called *mynum*, *MYNUM*, and *Mynum* are all different (although this would be confusing and should not be done).
- Although underscore characters are valid in a name, their use can cause problems with some programs that interact with MATLAB, so some programmers use mixed case instead (e.g., *partWeights* instead of *part_weights*)
- There are certain words called ***reserved words***, or ***keywords***, that cannot be used as variable names.
- Names of built-in functions (described in the next section) can, but should not, be used as variable names.

Additionally, variable names should always be ***mnemonic***, which means that they should make some sense. For example, if the variable is storing the radius of a circle, a name such as *radius* would make sense; *x* probably wouldn't.

The following commands relate to variables:

- **who** shows variables that have been defined in this Command Window (this just shows the names of the variables)
- **whos** shows variables that have been defined in this Command Window (this shows more information on the variables, similar to what is in the Workspace Window)
- **clearvars** clears out all variables so they no longer exist

- **clearvars** *variablename* clears out a particular variable
- **clearvars** *variablename1 variablename2* … clears out a list of variables (note: separate the names with spaces, not commas)
- **clear** is similar to **clearvars,** but also clears out functions

If nothing appears when **who** or **whos** is entered, that means there aren't any variables! For example, in the beginning of a MATLAB session, variables could be created and then selectively cleared (remember that the semicolon suppresses output).

```
>> who
>> mynum = 3;
>> mynum + 5;
>> who
Your variables are:
ans     mynum
>> clear mynum
>> who
Your variables are:
ans
```

These changes can also be seen in the Workspace Window.

1.3.3 Types

Every variable has a ***type*** associated with it. MATLAB supports many types, which are called ***classes***. (Essentially, a class is a combination of a type and the operations that can be performed on values of that type, but, for simplicity, we will use these terms interchangeably for now. More on classes will be covered in Chapter 11.)

For example, there are types to store different kinds of numbers. For float or real numbers, or in other words numbers with a decimal place (e.g., 5.3), there are two basic types: **single** and **double**. The name of the type **double** is short for ***double precision***; it stores larger numbers than the **single** type. MATLAB uses a ***floating point*** representation for these numbers.

There are many integer types, such as **int8**, **int16**, **int32**, and **int64**. The numbers in the names represent the number of bits used to store values of that type. For example, the type **int8** uses eight bits altogether to store the integer and its sign. As one bit is used for the sign, this means that seven bits are used to store actual numbers (0s or 1s). There are also ***unsigned*** integer types **uint8**, **uint16**, **uint32**, and **uint64**. For these types, the sign is not stored, meaning that the integer can only be positive (or 0).

The larger the number in the type name, the larger the number that can be stored in it. We will for the most part use the type **int32** when an integer type is required.

The type **char** is used to store either single ***characters*** (e.g., 'x') or ***character vectors***, which are sequences of characters (e.g., 'cat'). Both characters and character vectors are enclosed in single quotes.

The type **string** is used to store strings (e.g., "hello"). Strings are enclosed in double quotes. This type is new as of R2016b; the use of double quotes was introduced in R2017a.

The type **logical** is used to store **true**/**false** values.

Variables that have been created in the Command Window can be seen in the Workspace Window. In that window, for every variable, the variable name, value, and class (which is essentially its type) can be seen. Other attributes of variables can also be seen in the Workspace Window. Which attributes are visible by default depends on the version of MATLAB. However, when the Workspace Window is chosen, clicking on the down arrow allows the user to choose which attributes will be displayed by modifying Choose Columns.

By default, numbers are stored as the type **double** in MATLAB. The function **class** can be used to see the type of any variable:

```
>> num = 6 + 3;
>> class(num)
ans =
    'double'
```

1.4 NUMERICAL EXPRESSIONS

Expressions can be created using values, variables that have already been created, operators, built-in functions, and parentheses. For numbers, these can include operators such as multiplication and functions such as trigonometric functions. An example of such an expression is:

```
>> 2 * sin(1.4)
ans =
    1.9709
```

1.4.1 The Format Command and Ellipsis

The ***default*** in MATLAB is to display numbers that have decimal points with four decimal places, as shown in the previous example. (The default means if you do not specify otherwise, this is what you get.) The **format** command can be used to specify the output format of expressions.

There are many options, including making the format **short** (the default) or **long**. For example, changing the format to **long** will result in 15 decimal places.

This will remain in effect until the format is changed back to **short**, as demonstrated in the following.

```
>> format long
>> 2 * sin(1.4)
ans =
    1.970899459976920
>> format short
>> 2 * sin(1.4)
ans =
    1.9709
```

The **format** command can also be used to control the spacing between the MATLAB command or expression and the result; it can be either **loose** (the default) or **compact**.

```
>> format loose
>> 5*33

ans =

   165

>> format compact
>> 5*33
ans =
   165
>>
```

Especially long expressions can be continued on the next line by typing three (or more) periods, which is the ***continuation operator***, or the ***ellipsis***. To do this, type part of the expression followed by an ellipsis, then hit the Enter key and continue typing the expression on the next line.

```
>> 3 + 55 – 62 + 4 – 5 ...
+ 22 – 1

ans =
    16
```

1.4.2 Operators

There are in general two kinds of operators: ***unary*** operators, which operate on a single value, or ***operand***, and ***binary*** operators, which operate on two values or operands. The symbol "-", for example, is both the unary operator for negation and the binary operator for subtraction.

Here are some of the common operators that can be used with numerical expressions:

```
+ addition
– negation, subtraction
```

```
* multiplication
/ division (divided by e.g., 10/5 is 2)
\ division (divided into e.g., 5\10 is 2)
^ exponentiation (e.g., 5^2 is 25)
```

In addition to displaying numbers with decimal points, numbers can also be shown using ***scientific or exponential notation***. This uses ***e*** for the exponent of 10 raised to a power. For example, 2 * 10 ^ 4 could be written two ways:

```
>> 2 * 10 ^4
ans =
       20000
>> 2e4
ans =
       20000
```

1.4.2.1 Operator Precedence Rules

Some operators have ***precedence*** over others. For example, in the expression 4 + 5 * 3, the multiplication takes precedence over the addition, so first 5 is multiplied by 3, then 4 is added to the result. Using parentheses can change the precedence in an expression:

```
>> 4 + 5 * 3
ans =
    19
>> (4 + 5) * 3
ans =
    27
```

Within a given precedence level, the expressions are evaluated from left to right (this is called ***associativity***).

Nested parentheses are parentheses inside of others; the expression in the ***inner parentheses*** is evaluated first. For example, in the expression `5 - (6 * (4+2))`, first the addition is performed, then the multiplication, and finally the subtraction, to result in `-31`. Parentheses can also be used simply to make an expression clearer. For example, in the expression `((4 + (3*5)) - 1)`, the parentheses are not necessary, but are used to show the order in which the parts of the expression will be evaluated.

For the operators that have been covered thus far, the following is the precedence (from the highest to the lowest):

```
( )        parentheses
^          exponentiation
-          negation
*, /, \    all multiplication and division
+, -       addition and subtraction
```

PRACTICE 1.1

Think about what the results would be for the following expressions, and then type them in to verify your answers:

```
1\2
- 5 ^ 2
(-5) ^ 2
10-6/2
5 * 4/2 * 3
```

1.4.3 Built-in Functions and Help

There are many built-in functions in MATLAB. The **help** command can be used to identify MATLAB functions, and also how to use them. For example, typing **help** at the prompt in the Command Window will show a list of ***help topics*** that are groups of related functions. This is a very long list; the most elementary help topics appear at the beginning. Also, if you have any Toolboxes installed, these will be listed.

For example, one of the elementary help topics is listed as **matlab/elfun**; it includes the elementary math functions. Another of the first help topics is **matlab/ops,** which shows the operators that can be used in expressions.

To see a list of the functions contained within a particular help topic, type **help** followed by the name of the topic. For example,

```
>> help elfun
```

will show a list of the elementary math functions. It is a very long list, and it is broken into trigonometric (for which the default is radians, but there are equivalent functions that instead use degrees), exponential, complex, and rounding and remainder functions.

To find out what a particular function does and how to call it, type **help** and then the name of the function. For example, the following will give a description of the **sin** function.

```
>> help sin
```

Note that clicking on the *fx* to the left of the prompt in the Command Window also allows one to browse through the functions in the help topics. Choosing the Help button under Resources to bring up the Documentation page for MATLAB is another method for finding functions by category.

To ***call a function***, the name of the function is given followed by the ***argument(s)*** that are passed to the function in parentheses. Most functions then ***return value(s)***. For example, to find the absolute value of −4, the following expression would be entered:

```
>> abs(-4)
```

which is a ***call*** to the function **abs**. The number in the parentheses, the `-4`, is the ***argument***. The value 4 would then be ***returned*** as a result.

QUICK QUESTION!

What would happen if you use the name of a function, for example, **abs**, as a variable name?

Answer: This is allowed in MATLAB, but then **abs** could not be used as the built-in function until the variable is cleared (using either **clear** or **clearvars**). For example, examine the following sequence:

```
>> clearvars
>> abs(-6)
ans =
     6
>> abs = 11
abs =
    11
>> abs(-6)
Array indices must be positive integers or
logical values.
'abs' appears to be both a function and a
variable. If this is unintentional, use 'clear
abs' to remove the variable 'abs' from the
workspace.
  >> who
  Your variables are:
  abs  ans
  >> clearvars abs
  >> who
  Your variables are:
  ans
  >> abs(-6)
  ans =
      6
```

All of the operators have a functional form. For example, 2+5 can be written using the **plus** function as follows.

```
>> plus(2,5)
ans =
     7
```

MATLAB has a useful shortcut that is called the ***tab completion*** feature. If you type the beginning characters in the name of a function, and hit the tab key, a list of functions that begin with the typed characters pops up. Capitalization errors are automatically fixed.

Also, if a function name is typed incorrectly, MATLAB will suggest a correct name.

```
>> abso(-4)
Unrecognized function or variable 'abso'.
Did you mean:
>> abs(-4)
```

1.4.4 Constants

Variables are used to store values that might change, or for which the values are not known ahead of time. Most languages also have the capacity to store ***constants***, which are values that are known ahead of time and cannot possibly

change. An example of a constant value would be **pi**, or π, which is 3.14159…. In MATLAB, there are functions that return some of these constant values, some of which include:

pi 3.14159....

i $\sqrt{-1}$

j $\sqrt{-1}$

inf infinity ∞

NaN stands for "not a number," such as the result of 0/0

1.4.5 Random Numbers

When a program is being written to work with data, and the data are not yet available, it is often useful to test the program first by initializing the data variables to ***random numbers***. Random numbers are also useful in simulations. There are several built-in functions in MATLAB that generate random numbers, some of which will be illustrated in this section.

Random number generators or functions are not truly random. Basically, the way it works is that the process starts with one number, which is called the ***seed***. Frequently, the initial seed is either a predetermined value or it is obtained from the built-in clock in the computer. Then, based on this seed, a process determines the next "random number". Using that number as the seed the next time, another random number is generated, and so forth. These are actually called ***pseudorandom***—they are not truly random because there is a process that determines the next value each time.

The function **rand** can be used to generate uniformly distributed random real numbers; calling it generates one random real number in the ***open interval*** (0,1), which means that the endpoints of the range are not included. There are no arguments passed to the **rand** function in its simplest form. Here are two examples of calling the **rand** function:

```
>> rand
ans =
     0.8147
>> rand
ans =
     0.9058
```

The seed for the **rand** function will always be the same each time MATLAB is started, unless the initial seed is changed. The **rng** function sets the initial seed. There are several ways in which it can be called:

```
>> rng('shuffle')
>> rng(intseed)
>> rng('default')
```

With 'shuffle', the **rng** function uses the current date and time that are returned from the built-in **clock** function to set the seed, so the seed will always be different. An integer can also be passed to be the seed. The 'default' option will set the seed to the default value used when MATLAB starts up. The **rng** function can also be called with no arguments, which will return the current state of the random number generator:

```
>> state_rng = rng;  % gets state
>> randone = rand
randone =
    0.1270
>> rng(state_rng); % restores the state
>> randtwo = rand   % same as randone
randtwo =
    0.1270
```

Note
The words after the % are comments and are ignored by MATLAB.

The random number generator is initialized when MATLAB starts, which generates what is called the ***global stream*** of random numbers. All of the random functions get their values from this stream.

As **rand** returns a real number in the open interval (0, 1), multiplying the result by an integer *N* would return a random real number in the open interval (0, *N*). For example, multiplying by 10 returns a real in the open interval (0, 10), so the expression

```
rand*10
```

would return a result in the open interval (0, 10).

To generate a random real number in the range from *low* to *high*, first create the variables *low* and *high*. Then, use the expression `rand * (high − low) + low`. For example, the sequence

```
>> low = 3;
>> high = 5;
>> rand * (high − low) + low
```

would generate a random real number in the open interval (3, 5).

The function **randn** is used to generate normally distributed random real numbers.

1.4.5.1 Generating Random Integers

As the **rand** function returns a real number, this can be rounded to produce a random integer. For example,

```
>> round(rand*10)
```

would generate one random integer in the range from 0 to 10 inclusive (`rand*10` would generate a random real in the open interval (0, 10); rounding that will

return an integer). However, these integers would not be evenly distributed in the range. A better method is to use the function **randi**, which in its simplest form **randi(imax)** returns a random integer in the range from 1 to imax, inclusive. For example, **randi(4)** returns a random integer in the range from 1 to 4. A range can also be passed, for example, **randi([imin, imax])** returns a random integer in the inclusive range from imin to imax:

```
>> randi([3, 6])
ans =
     4
```

PRACTICE 1.2

Generate a random

- real number in the range (0,1)
- real number in the range (0, 100)
- real number in the range (20, 35)
- integer in the inclusive range from 1 to 100
- integer in the inclusive range from 20 to 35

1.5 CHARACTERS AND STRINGS

A character in MATLAB is represented using single quotes (e.g., 'a' or 'x'). The quotes are necessary to denote a character; without them, a letter would be interpreted as a variable name. Characters are put in an order using what is called a ***character encoding***. In the character encoding, all characters in the computer's ***character set*** are placed in a sequence and given equivalent integer values. The character set includes all letters of the alphabet, digits, and punctuation marks; basically, all of the keys on a keyboard are characters. Special characters, such as the Enter key, are also included. So, 'x', '!', and '3' are all characters. With quotes, '3' is a character, not a number.

Notice the difference in the formatting when a number is displayed versus a character:

```
>> var = 3
var =
     3
>> var = '3'
var =
    '3'
```

MATLAB also handles character arrays, which are sequences of characters in single quotes, and strings, which are sequences of characters in double quotes.

```
>> myword = 'hello'
myword =
    'hello'
>> yourword = "ciao"
yourword =
    "ciao"
```

The most common character encoding is the American Standard Code for Information Interchange, or ASCII. Standard ASCII has 128 characters, which have equivalent integer values from 0 to 127. The first 32 (integer values 0 through 31) are nonprinting characters. The letters of the alphabet are in order, which means 'a' comes before 'b', then 'c', and so forth. MATLAB actually can use a much larger encoding sequence, which has the same first 128 characters as ASCII. More on the character encoding, and converting characters to their numerical values, will be covered in Section 1.7.

1.6 RELATIONAL EXPRESSIONS

Expressions that are conceptually either true or false are called ***relational expressions***; they are also sometimes called ***Boolean expressions*** or ***logical expressions.*** These expressions can use both ***relational operators***, which relate two expressions of compatible types, and ***logical operators***, which operate on **logical** operands.

The relational operators in MATLAB are:

Operator	Meaning
>	greater than
<	less than
>=	greater than or equals
<=	less than or equals
==	equality
~=	inequality

All of these concepts should be familiar, although the actual operators used may be different from those used in other programming languages, or in mathematics classes. In particular, it is important to note that the operator for equality is two consecutive equal signs, not a single equal sign (as the single equal sign is already used as the assignment operator).

For numerical operands, the use of these operators is straightforward. For example, `3 < 5` means "3 less than 5", which is, conceptually, a true expression. In MATLAB, as in many programming languages, "true" is represented by the **logical** value 1, and "false" is represented by the **logical** value 0.

So, the expression `3 < 5` actually displays in the Command Window the value 1 (**logical**) in MATLAB. Displaying the result of expressions like this in the Command Window demonstrates the values of the expressions.

```
>> 3 < 5
ans =
  logical
   1
>> 2 > 9
ans =
  logical
   0
>> class(ans)
ans =
    'logical'
```

The type of the result is **logical**, not **double**. MATLAB also has built-in **true** and **false**.

```
>> true
ans =
  logical
     1
```

In other words, **true** is equivalent to **logical(1)** and **false** is equivalent to **logical(0)**.

Starting in R2016b, the output in the Command Window shows a header for most classes except for the default number type **double**, **char**, and **string**. If the type of a number result is not the default of **double**, the type (or class) is shown above the resulting value, as in the underlined "logical" in the previous expressions. However, in order to save room, these types will frequently not be shown for the rest of the book. Note: the class names **char** and **string** are not shown because the class is obvious from the single quotes (for **char**) or double quotes (for **string**).

Although these are **logical** values, mathematical operations could be performed on the resulting 1 or 0.

```
>> logresult = 5 < 7
logresult =
     1
>> logresult + 3
ans =
     4
```

Comparing characters (e.g., 'a' < 'c') is also possible. Characters are compared using their ASCII equivalent values in the character encoding. So, 'a' < 'c' is a **true** expression because the character 'a' comes before the character 'c'.

```
>> 'a' < 'c'
ans =
     1
```

The logical operators are:

Operator	Meaning
||	or
&&	and
~	not

All logical operators operate on **logical** or Boolean operands. The **not** operator is a unary operator; the others are binary. The **not** operator will take a **logical** expression, which is **true** or **false**, and give the opposite value. For example, `~(3 < 5)` is **false** as `(3 < 5)` is **true**. The **or** operator has two **logical** expressions as operands. The result is **true** if either or both of the operands are **true**, and **false** only if both operands are **false**. The **and** operator also operates on two **logical** operands. The result of an **and** expression is **true** only if both operands are **true**; it is **false** if either or both are **false**. The or/and operators shown here are used for ***scalars***, or single values. Other or/and operators will be explained in Chapter 2.

The || and && operators in MATLAB are examples of operators that are known as ***short-circuit*** operators. What this means is that if the result of the expression can be determined based on the first part, then the second part will not even be evaluated. For example, in the expression:

```
2 < 4 || 'a' == 'c'
```

the first part, 2 < 4, is **true** so the entire expression is **true**; the second part 'a' == 'c' would not be evaluated.

In addition to these logical operators, MATLAB also has a function **xor,** which is the exclusive or function. It returns **logical true** if one (and only one) of the arguments is **true**. For example, in the following only the first argument is **true**, so the result is **true**:

```
>> xor(3 < 5, 'a' > 'c')
ans =
     1
```

In this example, both arguments are **true** so the result is **false**:

```
>> xor(3 < 5, 'a' < 'c')
ans =
     0
```

Given the **logical** values of **true** and **false** in variables x and y, the ***truth table*** (see Table 1.1) shows how the logical operators work for all combinations. Note that the logical operators are ***commutative*** (e.g., $x \,||\, y$ is the same as $y \,||\, x$).

As with the numerical operators, it is important to know the operator precedence rules. Table 1.2 shows the rules for the operators that have been covered thus far in the order of precedence.

Table 1.1 Truth Table for Logical Operators

x	*y*	~*x*	*x* || *y*	*x* && *y*	xor(*x*,*y*)
True	True	False	True	True	False
True	False	False	True	False	True
False	False	True	False	False	False

Table 1.2 Operator Precedence Rules

Operators	Precedence
Parentheses: `( )`	Highest
Power `^`	
Unary: negation `(-)`, not `(~)`	
Multiplication, division `*, /, \`	
Addition, subtraction `+, -`	
Relational `<, <=, >, >=, ==, ~=`	
And `&&`	
Or `||`	Lowest

QUICK QUESTION!

Assume that there is a variable x that has been initialized. What would be the value of the expression

```
3 < x < 5
```

if the value of x is 4? What if the value of x is 7?

Answer: The value of this expression will always be **logical true**, or 1, regardless of the value of the variable x. Expressions are evaluated from left to right. So, first the expression $3 < x$ will be evaluated. There are only two possibilities: either this will be **true** or **false**, which means that either the expression will have the **logical** value 1 or 0. Then, the rest of the expression will be evaluated, which will be either $1 < 5$ or $0 < 5$. Both of these expressions are **true**. So, the value of x does not matter: the expression $3 < x < 5$ would be **true** regardless of the value of the variable x. This is a logical error; it would not enforce the desired range. If we wanted an expression that was **logical true** only if x was in the range from 3 to 5, we could write `3 < x && x < 5` (note that parentheses are not necessary).

PRACTICE 1.3

Think about what would be produced by the following expressions, and then type them in to verify your answers.

```
3 == 5 + 2
'b' < 'a' + 1
10 > 5 + 2
(10 > 5) + 2
'c' == 'd' - 1 && 2 < 4
'c' == 'd' - 1 || 2 > 4
xor('c' == 'd' - 1, 2 > 4)
xor('c' == 'd' - 1, 2 < 4)
10 > 5 > 2
```

Note: be careful about using the equality and inequality operators with numbers. Occasionally, ***roundoff errors*** appear, which means that numbers are close to their correct value but not exactly. For example, cos(pi/2) should be 0. However, because of a roundoff error, it is a very small number but not exactly 0.

```
>> cos(pi/2)
ans =
   6.1232e-17
>> cos(pi/2) == 0
ans =
     0
```

1.7 TYPE RANGES AND TYPE CASTING

The ***range*** of a type, which indicates the smallest and largest numbers that can be stored in the type, can be calculated. For example, the type **uint8** stores 2^8 or 256 integers, ranging from 0 to 255. The range of values that can be stored in **int8,** however, is from −128 to +127. The range can be found for any type by passing the name of the type as a string or character vector (which means in single quotes) to the functions **intmin** and **intmax**. For example,

```
>> intmin('int8')
ans =
-128
>> intmax('int8')
ans =
  127
```

There are many functions that convert values from one type to another. The names of these functions are the same as the names of the types. These names

can be used as functions to convert a value to that type. This is called ***casting*** the value to a different type, or ***type casting***. For example, to convert a value from the type **double**, which is the default, to the type **int32**, the function **int32** would be used. Entering the assignment statement

```
>> val = 6 + 3;
```

would result in the number 9 being stored in the variable *val*, with the default type of **double**, which can be seen in the Workspace Window. Subsequently, the assignment statement

```
>> val = int32(val);
```

would change the type of the variable to **int32**, but would not change its value. Here is another example using two different variables.

```
>> num = 6 + 3;
>> numi = int32(num);
>> whos
  Name    Size      Bytes    Class    Attributes
  num     1x1           8    double
  numi    1x1           4    int32
```

Note that **whos** shows the type (class) of the variables as well as the number of bytes used to store the value of a variable. One byte is equivalent to eight bits, so the type **int32** uses 4 bytes.

One reason for using an integer type for a variable is to save space in memory.

QUICK QUESTION!

What would happen if you go beyond the range for a particular type? For example, the largest integer that can be stored in **int8** is 127, so what would happen if we type cast a larger integer to the type **int8**?

```
>> int8(200)
```

Answer: The value would be the largest in the range, in this case 127. If, instead, we use a negative number that is smaller than the lowest value in the range, its value would be -128. This is an example of what is called ***saturation arithmetic***.

```
>> int8(200)
ans =
   127
>> int8(-130)
ans =
  -128
```

PRACTICE 1.4

- Calculate the range of integers that can be stored in the types **int16** and **uint16**. Use **intmin** and **intmax** to verify your results.
- Enter an assignment statement and view the type of the variable in the Workspace Window. Then, change its type and view it again. View it also using **whos**.

There is also a function **cast** that can cast a variable to a particular type. This has an option to cast a variable to the same type as another, using 'like'.

```
>> a = uint16(43);
>> b = 11;
>> whos
  Name   Size    Bytes   Class    Attributes
  a      1x1         2   uint16
  b      1x1         8   double
>> b = cast(b,'like',a);
>> whos
  Name   Size    Bytes   Class    Attributes
  a      1x1         2   uint16
  b      1x1         2   uint16
```

The numeric functions can also be used to convert a character to its equivalent numerical value (e.g., **double** will convert to a **double** value, and **int32** will convert to an integer value using 32 bits). For example, to convert the character 'a' to its numerical equivalent, the following statement could be used:

```
>> numequiv = double('a')
numequiv =
     97
```

This stores the **double** value 97 in the variable *numequiv*, which shows that the character 'a' is the 98th character in the character encoding (as the equivalent numbers begin at 0). It doesn't matter which number type is used to convert 'a'; for example,

```
>> numequiv = int32('a')
```

would also store the integer value 97 in the variable *numequiv*. The only difference between these will be the type of the resulting variable (**double** in the first case, **int32** in the second).

The function **char** does the reverse; it converts from any number to the equivalent character:

```
>> char(97)
ans =
    'a'
```

Note that, in earlier versions of MATLAB, quotes were not shown when the character was displayed.

As the letters of the alphabet are in order, the character 'b' has the equivalent value of 98, 'c' is 99, and so on. Math can be done on characters. For example,

to get the next character in the character encoding, 1 can be added either to the integer or the character:

```
>> numequiv = double('a');
>> char(numequiv + 1)
ans =
    'b'
>> 'a' + 2
ans =
    99
```

The first 128 characters are equivalent to the 128 characters in standard ASCII. MATLAB uses an encoding, however, that has 65,535 characters. The characters from 128 to 65,535 depend on your computer's ***locale setting***, which sets the language for your interface; for example, 'en_US' is the locale for English in the United States.

To shift the characters of a string "up" in the character encoding, an integer value can be added to a string. For example, the following expression will shift by one:

```
>> char("abcd" + 1)
ans =
    "bcde"
```

PRACTICE 1.5

- Find the numerical equivalent of the character 'x'.
- Find the character equivalent of 107.

1.8 BUILT-IN NUMERICAL FUNCTIONS

We have seen in Section 1.4.3 that the **help** command can be used to see help topics such as **elfun**, as well as the functions stored in each help topic.

MATLAB has many built-in trigonometric functions for sine, cosine, tangent, and so forth. For example, **sin** is the sine function in radians. The inverse, or arcsine function in radians is **asin**, the hyperbolic sine function in radians is **sinh**, and the inverse hyperbolic sine function is **asinh**. There are also functions that use degrees rather than radians: **sind** and **asind**. Similar variations exist for the other trigonometric functions.

In addition to the trigonometric functions, the **elfun** help topic also has some rounding and remainder functions that are very useful. Some of these include **fix**, **floor**, **ceil**, **round**, **mod**, **rem**, and **sign**.

Both the **rem** and **mod** functions return the remainder from a division; for example 5 goes into 13 twice with a remainder of 3, so the result of this expression is 3:

```
>> rem(13,5)
ans =
     3
```

QUICK QUESTION!

What would happen if you reversed the order of the arguments by mistake, and typed the following:

```
rem(5,13)
```

Answer: The **rem** function is an example of a function that has two arguments passed to it. In some cases, the order in which the arguments are passed does not matter, but for the **rem** function the order does matter. The **rem** function divides the second argument into the first. In this case, the second argument, 13, goes into 5 zero times with a remainder of 5, so 5 would be returned as a result.

Another function in the **elfun** help topic is the **sign** function, which returns 1 if the argument is positive, 0 if it is 0, and -1 if it is negative. For example,

```
>> sign(-5)
ans =
    -1
>> sign(3)
ans =
     1
```

PRACTICE 1.6

Use the **help** function to find out what the rounding functions **fix**, **floor**, **ceil**, and **round** do. Experiment with them by passing different values to the functions, including some negative, some positive, and some with fractions less than 0.5 and some greater. *It is very important when testing functions that you test thoroughly by trying different kinds of arguments!*

The **round** function has an option to round to a specified number of digits.

```
>> round(pi,3)
ans =
    3.1420
```

MATLAB has the exponentiation operator ^, and also the function **sqrt** to compute square roots and **nthroot** to find the nth root of a number. For example, the following expression finds the third root of 64.

```
>> nthroot(64,3)
ans =
     4
```

For the case in which $x = b^y$, y is the ***logarithm*** of x to base b, or in other words, $y = \log_b(x)$. Frequently used bases include $b = 10$ (called the ***common logarithm***), $b = 2$ (used in many computing applications), and $b = e$ (the constant e, which equals 2.7183); this is called the ***natural logarithm***. For example,

$100 = 10^2$ so $2 = \log_{10}(100)$
$32 = 2^5$ so $5 = \log_2(32)$

MATLAB has built-in functions to return logarithms:

log(x) returns the natural logarithm
log2(x) returns the base 2 logarithm
log10(x) returns the base 10 logarithm

MATLAB also has a built-in function **exp(n),** which returns the constant e^n.

QUICK QUESTION!

There is no built-in constant for e (2.718), so how can that value be obtained in MATLAB?

Answer: Use the exponential function **exp**; e or e^1 is equivalent to **exp(1).**

```
>> exp(1)
ans =
    2.7183
```

Note: Don't confuse the value e with the e used in MATLAB to specify an exponent for scientific notation.

In R2015b, the functions **deg2rad** and **rad2deg** were introduced to convert between degrees and radians, e.g.,

```
>> deg2rad(180)
ans =
    3.1416
```

1.9 ALTERNATE MATLAB PLATFORMS

The MathWorks has many products related to MATLAB, and several websites and associated apps. To view the products, go to the MathWorks' website:

https://www.mathworks.com

Clicking on Products will bring up a list of the products and services that are available. These include many toolboxes to use with MATLAB. Additional services include MATLAB Mobile, MATLAB Online, and MATLAB Drive.

MATLAB Mobile is a free app that is available for Android and Apple devices. The interface provides a command prompt that allows you to enter MATLAB commands and have them evaluated, just like in the Command Window. It also provides a history of commands. With MATLAB Mobile, you can connect

to MATLAB sessions running either in the MathWorks Cloud, or on your own computer. Also available is a set of sensor data. Hitting the Sensors icon brings up the ability to get data from the sensors that are built into your device. These include Position (including Latitude and Longitude), Acceleration, and Orientation (including Azimuth, Pitch, and Roll). By connecting your mobile device, you can acquire and log the sensor data in MATLAB. For example, you could take your device on a trip and then plot where it went! To do this, it is necessary to download both the MATLAB Mobile app and the sensor package for your computer. The **connector** command connects the devices, and the **mobiledev** function creates an object (more on objects in Chapter 11) that is used to acquire the sensor data. Read the MATLAB Mobile documentation online for more details.

MATLAB Online™ was introduced in R2017a. Most MATLAB licenses include access to MATLAB Online. No download is necessary, but you do need to log in to a MathWorks account. MATLAB Drive provides cloud storage for MATLAB files, and is available with any MathWorks account.

Also available through the website are options for practicing MATLAB code, reading blogs, posting questions, and sharing code files. The Cody website has a collection of problems that you can solve in order to practice your skills with MATLAB. An offshoot of Cody is Cody Coursework, which is a website that allows instructors to post problems for students to solve, and includes the ability for instructors to create auto-graders for the students' codes.

Explore Other Interesting Features

This section lists some features and functions in MATLAB, related to those explained in this chapter, that you wish to explore on your own.

- Workspace Window: there are many other aspects of the Workspace window to explore. To try this, create some variables. Make the Workspace window the active window by clicking the mouse in it. From there, you can choose which attributes of variables to make visible by choosing Choose Columns from the menu. Also, if you double click on a variable in the Workspace window, this brings up a Variable Editor window that allows you to modify the variable.
- Click on the *fx* next to the prompt in the Command Window, and under MATLAB choose Mathematics, then Elementary Math, then Exponents, and Logarithms to see more functions in this category.
- Use **help** to learn about the **path** function and related directory functions such as **addpath** and **which**.
- The **pow2** function
- Functions related to type casting including **typecast**
- Find the accuracy of the floating point representation for single and double precision using the **eps** function. ■

SUMMARY

COMMON PITFALLS

It is common when learning to program to make simple spelling mistakes and to confuse the necessary punctuation. Examples are given here of very common errors. Some of these include:

- Putting a space in a variable name
- Confusing the format of an assignment statement as

  ```
  expression = variablename
  ```

  rather than

  ```
  variablename = expression
  ```

  The variable name must always be on the left.
- Using a built-in function name as a variable name, and then trying to use the function
- Confusing the two division operators / and \
- Forgetting the operator precedence rules
- Confusing the order of arguments passed to functions—for example, to find the remainder of dividing 3 into 10 using **rem(3,10)** instead of **rem(10,3)**
- Not using different types of arguments when testing functions
- Forgetting to use parentheses to pass an argument to a function (e.g., "fix 2.3" instead of "**fix(2.3)**"). MATLAB returns the ASCII equivalent for each character when this mistake is made (what happens is that it is interpreted as the function of a string, "**fix('2.3')**").
- Confusing && and ||
- Confusing || and **xor**
- Putting a space in 2-character operators (e.g., typing "< =" instead of "<=")
- Using = instead of == for equality

PROGRAMMING STYLE GUIDELINES

Following these guidelines will make your code much easier to read and understand, and therefore, easier to work with and modify.

- Use mnemonic variable names (names that make sense; for example, *radius* instead of *xyz*)
- Although variables named *result* and *RESULT* are different, avoid this as it would be confusing
- Do not use names of built-in functions as variable names

- Store results in named variables (rather than using *ans*) if they are to be used later
- Do not use *ans* in expressions
- Make sure variable names have fewer characters than **namelengthmax**
- If different sets of random numbers are desired, set the seed for the random functions using **rng**

MATLAB Functions and Commands

demo	uint16	rand	mod
help	uint32	rng	rem
lookfor	uint64	clock	sign
doc	char	randn	sqrt
quit	string	randi	nthroot
exit	logical	xor	log
namelengthmax	true	intmin	log2
who	false	intmax	log10
whos	class	cast	exp
clearvars	format	asin	deg2rad
clear	sin	sinh	rad2deg
single	abs	asinh	connector
double	plus	sind	mobiledev
int8	pi	asind	
int16	i	fix	
int32	j	floor	
int64	inf	ceil	
uint8	NaN	round	

MATLAB Operators

assignment =	multiplication *	less than <	inequality ~=
ellipsis, or	divided by /	greater than or	or for scalars ||
continuation …	divided into \	equals >=	and for scalars &&
addition +	exponentiation ^	less than or equals	not ~
negation -	parentheses ( )	<=	
subtraction –	greater than >	equality ==	

Exercises

1. Create a variable *myage* and store your age in it. Add 2 to the value of the variable. Subtract 3 from the value of the variable. Observe the Workspace Window and Command History Window as you do this.
2. Create a variable to store the atomic weight of iron (55.85).
3. Explain the difference between these two statements:

```
result = 9*2
result = 9*2;
```

4. In what variable would the result of the following expression be stored:

```
>> 3 + 5
```

5. Use the built-in function **namelengthmax** to find out the maximum number of characters that you can have in an identifier name under your version of MATLAB.
6. Create two variables to store a weight in pounds and ounces. Use **who** and **whos** to see the variables. Use **class** to see the types of the variables. Clear one of them using **clearvars** and then use **who** and **whos** again.
7. Explore the **format** command in more detail. Use **help format** to find options. Experiment with **format bank** to display dollar values.
8. Find a **format** option that would result in the following output format:

```
>> 5/16 + 2/7
ans =
   67/112
```

9. Think about what the results would be for the following expressions, and then type them in to verify your answers.

```
25 / 5 * 3
4 + 2 ^ 3
(4 + 1) ^ 2
2 \ 12 + 5
4 - 1 * 5
```

10. There are 1.6093 kilometers in a mile. Create a variable to store a number of miles. Convert this to kilometers, and store in another variable.
11. Create a variable *ftemp* to store a temperature in degrees Fahrenheit (F). Convert this to degrees Celsius (C) and store the result in a variable *ctemp*. The conversion factor is C = (F – 32) * 5/9.
12. The following assignment statements either contain at least one error, or could be improved in some way. Assume that *radius* is a variable that has been initialized. First, identify the problem, and then fix and/or improve them:

```
33 = number
my variable = 11.11;
area = 3.14 * radius ^2;
x = 2 * 3.14 * radius;
```

13. Experiment with the functional form of some operators such as **plus**, **minus**, and **times**.
14. Explain the difference between constants and variables.
15. Generate a random

- real number in the range (0, 30)
- real number in the range (10, 100)
- integer in the inclusive range from 1 to 20

- integer in the inclusive range from 0 to 20
- integer in the inclusive range from 30 to 80

16. Get into a new Command Window, and type **rand** to get a random real number. Make a note of the number. Then, exit MATLAB and repeat this, again making a note of the random number; it should be the same as before. Finally, exit MATLAB and again get into a new Command Window. This time, change the seed before generating a random number; it should be different.
17. What is the difference between x and 'x'?
18. What is the difference between 5 and '5'?
19. The combined resistance R_T of three resistors R_1, R_2, and R_3 in parallel is given by

$$R_T = \frac{1}{\frac{1}{R_1} + \frac{1}{R_2} + \frac{1}{R_3}}$$

Create variables for the three resistors and store values in each, and then calculate the combined resistance.

20. What would be the result of the following expressions?

```
'b' >= 'c' - 1
3 == 2 + 1
(3 == 2) + 1
xor(5 < 6, 8 > 4)
```

21. Explain why the following expression results in 0 for false:

```
5 > 4 > 1
```

22. Explain why the following expression results in 1 for true:

```
result = -20;
0 <= result <= 10
```

23. Create two variables *x* and *y* and store numbers in them. Write an expression that would be **true** if the value of *x* is greater than five or if the value of *y* is less than ten, but not if both of those are **true**.
24. Use the equality operator to verify that 4*10 ^3 is equal to 4e3.
25. In the ASCII character encoding, the letters of the alphabet are in order: 'a' comes before 'b' and also 'A' comes before 'B'. However, which comes first - lower or uppercase letters?
26. Are there equivalents to **intmin** and **intmax** for real number types? Use **help** to find out.
27. Use **intmin** and **intmax** to determine the range of values that can be stored in the types **uint32** and **uint64**.
28. Use the **cast** function to cast a variable to be the same type as another variable.

29. Use **help elfun** or experiment to answer the following questions:
    - Is **fix(3.5)** the same as **floor(3.5)**?
    - Is **fix(3.4)** the same as **fix(-3.4)**?
    - Is **fix(3.2)** the same as **floor(3.2)**?
    - Is **fix(-3.2)** the same as **floor(-3.2)**?
    - Is **fix(-3.2)** the same as **ceil(-3.2)**?
30. For what range of values is the function **round** equivalent to the function **floor**? For what range of values is the function **round** equivalent to the function **ceil**?
31. Use **help** to determine the difference between the **rem** and **mod** functions.
32. Use the equality operator to verify that log10(1000) is 3.
33. Using only the integers 2 and 3, write as many expressions as you can that result in 9. Try to come up with at least 10 different expressions (e.g., don't just change the order). Be creative! Make sure that you write them as MATLAB expressions. Use operators and/or built-in functions.
34. A vector can be represented by its rectangular coordinates x and y or by its polar coordinates r and θ. Theta is measured in radians. The relationship between them is given by the equations:

    ```
    x = r * cos(θ)
    y = r * sin(θ)
    ```

    Assign values for the polar coordinates to variables *r* and *theta*. Then, using these values, assign the corresponding rectangular coordinates to variables *x* and *y*.
35. In special relativity, the Lorentz factor is a number that describes the effect of speed on various physical properties when the speed is significant relative to the speed of light. Mathematically, the Lorentz factor is given as:

$$\gamma = \frac{1}{\sqrt{1 - \frac{v^2}{c^2}}}$$

    Use 3×10^8 m/s for the speed of light, *c*. Create variables for *c* and the speed *v* and from them a variable *lorentz* for the Lorentz factor.
36. A company manufactures a part for which there is a desired weight. There is a tolerance of *N* percent, meaning that the range between minus and plus *N*% of the desired weight is acceptable. Create a variable that stores a weight, and another variable for *N* (e.g., set it to 2). Create variables that store the minimum and maximum values in the acceptable range of weights for this part.
37. A chemical plant releases an amount *A* of pollutant into a stream. The maximum concentration *C* of the pollutant at a point which is a distance *x* from the plant is:

$$C = \frac{A}{x}\sqrt{\frac{2}{\Pi e}}$$

Create variables for the values of A and x, and then for C. Assume that the distance x is in meters. Experiment with different values for x.

38. The geometric mean g of n numbers x_i is defined as the nth root of the product of x_i:

$$g = \sqrt[n]{x_1 x_2 x_3 \cdots x_n}$$

(This is useful, for example, in finding the average rate of return for an investment which is something you'd do in engineering economics). If an investment returns 15% the first year, 50% the second, and 30% the third year, the average rate of return would be `(1.15*1.50*1.30)^(1/3)`) Compute this.

39. Use the **deg2rad** function to convert 180 degrees to radians.
40. If you have an Apple or Android device, install MATLAB mobile and investigate the sensor information. This can be done without connecting the device to MATLAB.