

CHAPTER 2

Invoking a Computation

*He was so learned that he could name a horse in nine languages;
so ignorant that he bought a cow to ride on.*

— Benjamin Franklin

When we first study a natural language such as English or Chinese, we usually learn single words and then ways to combine these words to form simple expressions: *Hello. How are you? Where is the pen? The book is on the table.* Only later do we study how to combine these simple expressions into more complex forms involving more elaborate grammar.

In the next several chapters, we will study a language intended for communicating with computers. This language, MATLAB, is designed particularly for describing algorithms in sufficient detail and precision that the computer can perform the computation we desire. Although computer languages are much less complex than natural languages, we will start our study with the simplest of expressions: ones that convey an instruction to perform a specific computation, the algorithm, which already is known to the computer. This is called “invoking a computation” but also is known less formally as “running a command” or “executing a command.”

Before you can invoke a computation in MATLAB, you need to start up a program called the MATLAB “interpreter.” You start this program in the familiar way, by clicking on an icon or whatever procedure you are used to on your computer. The MATLAB interpreter then displays a prompt, usually something like `>>`. You type text at the prompt and, after pressing the ENTER key, MATLAB interprets your text as the invocation of a computation: the interpreter is the genie that performs your commands.

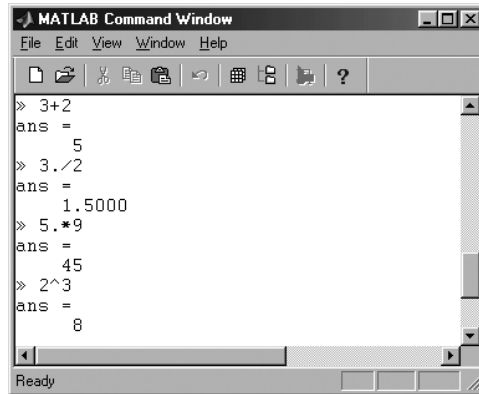


Figure 2.1. The MATLAB command window displays a prompt (`>>`) at which expressions can be typed to invoke a computation. On different computers, the window’s appearance may differ.

2.1 Expressions and Commands

Here is an example:

```

>> 3 + 2
ans: 5
    
```

We asked the computer to add 3 and 2. The computer did so and told us the answer. (Since the typographical requirements of a book are different from those of the computer screen, the examples shown in this book are printed in a somewhat different form than they would appear on the computer screen. The book uses an ordinary typewriter font to display a command and a *typewriter italic* font to display the response. Figure 2.1 shows how things look on the computer screen.)

Key Term

An *expression* is a statement that provides all of the information needed to invoke a computation. Here are some other examples of expressions and the output of the resulting computation:

```

>> 3 ./ 2
ans: 1.5000
>> 5 .* 9
ans: 45
>> 2^3
ans: 8
>> cos( 3.14159 )
ans: -1.0000
    
```

```
>> log10( 100 )
ans: 2
```

Key Term
Key Term

To execute a command, you type an expression followed by `ENTER`. MATLAB responds by *evaluating* the expression and returning the output, which is often called the *value* of the expression.

Whenever we invoke a computation, we need to provide three pieces of information:

1. What is the operator?
2. What are the inputs?
3. What should be done with the output?

Key Term

You can see from the previous examples that there are two different styles of specifying the operator and inputs. One of the styles is similar to the standard arithmetic notation learned by youngsters. The notation is this: The two numbers to be operated on are written on either side of a special symbol. The symbol itself tells which operation to perform. This is called the *infix* style of notation and is used mainly for the common arithmetic operations. Some of the symbols might be unfamiliar at first: $3 \cdot *2$ rather than 3×2 ; $3 \cdot ./2$ rather than $3 \div 2$; $3 \cdot ^2$ rather than 3^2 .

Key Term

The second style for specifying the operator is called *functional-style notation*. In this style, the name of the operator (e.g., `cos` or `log10`) is followed by one or more arguments enclosed in parentheses. When there is more than one argument, the arguments are separated by commas as in

```
>> plus(3,2)
ans: 5
>> twoCircles(0,1,2,1,0,3)
ans: -1.9490 0.5510
```

There is no fundamental difference between the functional-style notation and the infix notation: $(3+2)$ is precisely equivalent to `plus(3,2)`.

Students with tradition-minded teachers learn that in an expression such as $3 + 6$, the 3 is called the addend and the 6 the augend. Perhaps it’s silly to identify the two inputs with different names. After all, $3 + 6$ is the same as $6 + 3$. But this commutativity isn’t true in subtraction. In $7 - 4$, the 7 is called the minuend and the 4 the subtrahend; the distinction between the two inputs is important.

Terms such as *minuend* and *augend* are esoteric and obsolete. A more general and useful term is *argument*, which refers to each of the inputs of a computation. The computation to be performed is called the *operation*. In some operations, such as subtraction, the two arguments play different roles. We therefore keep track of which argument is which by ordering them: In $7 - 4$ the 7 is the first argument and the 4 is the second argument.

Key Term

All of the preceding examples are *simple expressions* that involve a single operation. The operations used in the preceding examples are addition, division, multiplication, exponentiation, the cosine, and the base-10 logarithm. MATLAB displayed the output of each computation in the command window as shown in Figure 2.1. Often, though, the output is needed as the input to another computation. This is done using *compound expressions* in which one or more of the arguments to an operation is itself an expression. For instance,

```
>> (3+2) ./ 5
ans: 1
```

The two arguments to the division operator are the expression (3+2) and the number 5. In order to perform the division, MATLAB first evaluates the expression (3+2), using that output as the first argument to the division operator. Other examples are as follows:

```
>> 3 + 15./ 5
ans: 6
>> sqrt(3.^2 + 4.^2)
ans: 5
>> 2.^2 + 1
ans: 5
>> 2.^(2 + 1)
ans: 8
>> 10.^log10(592.3)
ans: 592.3
```

These few examples may provide all the information you need to deduce how to invoke any arithmetic computation that might occur to you.

All of the preceding examples are *well-formed expressions*: They completely and properly describe a computation according to the rules of the MATLAB language. *Ill-formed expressions* are those that are meaningless to the interpreter. Using such expressions causes an error message to be generated by the interpreter. For example,

```
>> (3+2./5
???: (3+2./5
      |
      A closing right parenthesis is missing.
      Check for a missing ")" or a missing operator.
>> logg(10)
      logg(10)
      Undefined function or variable 'logg'.
```

Even well-formed expressions can generate warnings or errors when the computation cannot be meaningfully carried out:

```
>> 1./0
Warning: Divide by zero.
ans = Inf
```

A Matter of Style: _____

Some expressions may appear ambiguous to a human reader (for instance, does $3+15./5$ mean $3 + \frac{15}{5}$ or $\frac{3+15}{5}$?). MATLAB, like other computer languages, applies simple rules to eliminate any potential for ambiguity. For example, there is a rule that multiplication and division have a higher precedence than addition or subtraction, so that in $3+2./5$, the division occurs first and the addition second, resulting in an output 3.4. In interpreting the expression $1./3./2$, MATLAB applies a rule that says that when dealing with operations of equal precedence, the leftmost operation is applied first. Thus, $1./3./2$ corresponds with $\frac{1/3}{2}$ and not $\frac{1}{3/2}$.

Due to the rules, all well-formed expressions are completely unambiguous to the MATLAB interpreter. The problem is that the unambiguous intentions of the human reader or writer of a command may not match the unambiguous interpretation of the command by the computer. Even if you master the rules that the computer uses to interpret commands, keep in mind that some future reader of your programs, who perhaps has experience with another language with different rules, may misunderstand even a correctly formed statement.

Here’s a simple rule for avoiding problems:

When any potential for ambiguity exists, resolve it with parentheses!

2.2 Changing State: Assignment

Key Term

In the previous chapter, we saw how algorithms set up a state and modify it at each step. The state was defined as the information in the computer’s memory. The interpreter provides facilities for the manipulation of the information in memory. The memory is organized into *variables* each of which has two components: a name and a value. To create a variable, you use a special command that always has the same form

$$\text{name} = \text{expression}$$

Key Term

This is called an *assignment* statement; it associates a value to the variable. The value is not the expression itself, but the result of evaluating the expression. For example, you can create a variable named *a* that holds the value of $\sqrt{2}$: