Conditional Logic

Leigh J. Little SUNY Brockport September 16, 2022

Contents

1 Introduction **1**

2 Conditional Logic **1**

3 Relational Operators and Logical Calculations **2**
3.1 Back to the Social Security Script 3

4 Other `if` Testing Constructs **3**

5 Some Examples of `if-then` Testing **4**
5.1 Example 1: Illustration of Program Flow 4
5.2 Example 2: Test if a value is negative, positive or zero 5
5.3 Example 3: Piecewise Function Evaluation 5

6 Nesting of `if-then` Structures and Conjunction Operators **6**
6.1 Conjunction Operators 7
6.2 Example 4: Piecewise Function with Conjunction Operators 7

7 Comparing Floating Point Numbers **8**
7.1 Entering Values in Scientific Notation 9

1 Introduction

We have seen that it is possible to write simple scripts in MATLAB that will perform calculator type operations. However, in order to write more elaborate programs we need to incorporate some additional programming capabilities. Over the next lectures we will examine two of the most important of these; conditional logic and looping operations.

2 Conditional Logic

One of the most useful things you can do in a program is to ask it a question. Depending on the nature of a calculation, you might want to take several different courses of action depending on the current value of one or more variables. Conditional logic (also called conditional branching or if-then logic) is what permits you to ask your program a question.

Type the following into a script called `social.m` and run it:

```
clear
n = input('Input a value for n ');

if (n >= 65)
   disp('You are eligible for social security')
else
   disp('You are not eligible for social security')
end
```

Run your script for several value of `n`. What do you observe when you run your script?

This is an example of an `if-then-else` block. The sequence of statements will produce one of two different outputs depending on the value of `n` that the user inputs. We have asked our program a question:

```
if (n >= 65)
```

then told it what to do if this statement is true

```
disp('You are eligible for social security')
```

and what to do if the statement is false

```
disp('You are not eligible for social security')
```

This is the essence of conditional logic. We ask our program questions, then take different actions based on the answers to the questions. The questions we ask must always have a yes or no (true or false) answer.

The general syntax of an `if-then-else` block is

```
if (conditional statement)
   |
   |  Block of statements to execute if statement is true
   |
else
   |
   |  Block of statements to execute if statement is false
   |
end
```

Some general notes about `if` statements:

- The () around the conditional statement are optional, but it's easier to read your code if you put them in.
- You can have any number of statements in each block.
- The bodies of the `if` blocks are indented. This aids in making your code more readable. This is optional in MATLAB (but not Python), so I will require that you do this.
- The program will execute the statements in the appropriate block (depending on the result of the test), then jump out of the structure to the first executable statement past the `end` statement.
- Every `if` must have a corresponding `end`.

3 Relational Operators and Logical Calculations

Relational operators are used to compare the values of variables. Table 1 shows the available relational (or conditional) operators in MATLAB. Any logical comparison between variables is a calculation, but it is a

`>`	greater than
`>=`	greater than or equal to
`<`	less than
`<=`	less than or equal to
`==`	logical equality
`~=`	logical inequality
`&`	logical AND
`|`	logical OR
`~`	logical NOT (negation)

Table 1: Relational Operators in MATLAB.

special type of calculation called a *boolean operation*. The output from any boolean operation or comparison is either True (=1) or False (=0). For example, type the lines below into the MATLAB command window:

```
>> x = -1
x =
   -1
>> y = 3
y =
   3
>> x < y
ans =
     1
```

In this example, we have asked if $x < y$. Because the answer to this is True, the result of the comparison is True (=1). Similarly,

```
>> x > y
ans =
     0
>> x == y
ans =
     0
>> (x < 0) & (y > 0)
ans =
     1
```

The first 2 examples above return a value of 0 (False) because the comparisons `x > y` and `x == y` are false. Finally, the last example is called a compound relation. It is True because both `x < 0` AND `y > 0` are True.

3.1 Back to the Social Security Script

Because there are many ways to ask a question, the social security script can be written in more than one way. For example, the code below is equivalent to the first one

```
clear
n = input('Input a value for n ');

if (n < 65)
   disp('You are not eligible for social security')
else
   disp('You are eligible for social security')
end
```

4 Other `if` Testing Constructs

There are two other forms of the `if-then` construct. The first is the simple `if-then` statement

```
if (conditional statement)
   |
   |  Block of statements to execute if statement is true
   |
end
```

The second one is the `if-then-elseif` statement

```
if (conditional statement 1)
   |
   |  Block of statements to execute if statement 1 is true
   |
elseif (conditional statement 2)
   |
   |  Block of statements to execute if statement 2 is true
```

```
   |
elseif (conditional statement 3)
   |
   |  Block of statements to execute if statement 3 is true
   |
elseif(.....
   |
   |
   |
else
   |
   |  Block of statements to execute if none of the above statements are true
   |
end
```

Some comments on the `if-then-elseif` structure

- The program will perform each conditional test in the order they are listed.
- Suppose the first two conditions are false and the third condition is true. Then the third block of statements is executed. The remainder of the tests in the main `if` block are ignored and the program will jump to the first executable statement past the `end` statement.
- Make sure that `elseif` is one word.
- You can have as many `elseif` clauses as you need.
- The `else` is optional.
- There is only one `end` for the structure as a whole.

5 Some Examples of `if-then` Testing

Some other examples of conditional logic are shown in the examples below.

5.1 Example 1: Illustration of Program Flow

Consider the simple example below

```
n = 2;
if (n == 1)
   disp('n is equal to 1')
elseif(n == 2)
   disp('n is equal to 2')
elseif(n == 3)
   disp('n is equal to 3')
elseif(n == 4)
   disp('n is equal to 4')
else
   disp('n is not 1, 2, 3 or 4')
end
disp('End of test')
```

Save this as `if_ex1.m`, then run the code:

```
>> if_ex1
n is equal to 2
End of test
```

Here is the same code with the sequence commented:

```
n = 2;
if (n == 1)        %% This is false, so the block below
                   %% is skipped.
   disp('n is equal to 1')
elseif(n == 2)     %% This is true, so the block below
                   %% is executed.
   disp('n is equal to 2')
                   %% All the remaining clauses are ignored.
                   %% Flow proceeds to the display command
                   %% after the end statement
elseif(n == 3)
   disp('n is equal to 3')
elseif(n == 4)
   disp('n is equal to 4')
else
   disp('n is not 1, 2, 3 or 4')
end
disp('End of test')
```

5.2 Example 2: Test if a value is negative, positive or zero

If we wanted to perform a test to see if some value was negative, positive or zero, we could do the following

```
t = input('Input test value: ');
if (t > 0)
    disp('t is positive')
elseif (t < 0)
    disp('t is negative')
elseif (t == 0)
    disp('t is zero')
end
```

However, we can do this a different way. Because there are only 3 possibilities, if neither of the first 2 are true, we know the answer has to be be the third option, hence we could also do

```
t = input('Input test value: ');
if (t > 0)
    disp('t is positive')
elseif (t < 0)
    disp('t is negative')
else
    disp('t is zero')
end
```

There are 10 other ways we could perform the same test (by switching the order in which the testing is done).

5.3 Example 3: Piecewise Function Evaluation

Piecewise functions are often encountered in calculations. Suppose we wanted to evaluate the function

$$y = \begin{cases} x^2 & x \geq 0 \\ x^3 - x + 5 & x < 0. \end{cases}$$

The following code would accomplish this

```
x = input('Input x: ');
if (x >= 0)
   y = x^2;
else
```

```
   y = x^3-x+5;
end
```

The code below would also work

```
x = input('Input x: ');
if (x < 0)
   y = x^3-x+5;
elseif(x >= 0)
   y = x^2;
end
```

6 Nesting of `if-then` Structures and Conjunction Operators

We can only ask questions that have true or false answers. Some questions don't have simple yes or no answers and in such cases, the question must be broken down into a series of simple questions or must be expressed in terms of the conjunction operators (*i.e.,* `AND`, `OR` and `NOT` operators).

To allow for more complex questions, any type of `if-then` structure can be nested inside any other `if-then` structure. This means that the body of one `if` statement can itself contain an `if` statement. For example, consider evaluating a 2-dimensional piecewise function

$$z = \begin{cases} x^2 + y^2 & x \geq 0, y \geq 0 \\ x^2 - y^2 & x \geq 0, y < 0 \\ y^2 - x^2 & x < 0, y \geq 0 \\ -x^2 - y^2 & x < 0, y < 0 \end{cases}$$

This can be done using a sequence of nested `if-then-else` statements

```
x = input('Input x: ');
y = input('Input y: ');

if (x >= 0)          % If this is true, one of the first 2 equations is needed
   if (y >= 0)       % If this is true, we need the first equation
      z = x^2 + y^2
   elseif(y < 0)     % If this is true, we need the second equation
      z = x^2 - y^2
   end               % This end closes the first inner if
elseif (x < 0)       % If this is true, one of the last 2 equations is needed
   if (y >= 0)
      z = y^2 - x^2  % If this is true, we need the third equation
   elseif(y < 0)
      z = -x^2 - y^2  % If this is true, we need the fourth equation
   end               % This end closes the second inner if
end                  % This end closes the outer if
```

This could also be done using

```
x = input('Input x: ');
y = input('Input y: ');

if (x >= 0)
   if (y >= 0)
      z = x^2 + y^2
   else
      z = x^2 - y^2
   end               % This end closes the first inner if
else
   if (y >= 0)
```

```
      z = y^2 - x^2
   else
      z = -x^2 - y^2
   end               % This end closes the second inner if
end                  % This end closes the outer if
```

Some comments about nested `if-then` statements:

- Each `if` structure needs its own `end` statement.
- `if` statements that appear in a block must logically terminate in that block.
- There is a limit of 7 levels of nesting.

6.1 Conjunction Operators

The conjunction operators (`AND`, `OR` and `NOT`) can be used to build a more complex `if` statement. The truth tables below summarize how these are used

p	q	p `AND` q	p `OR` q
T	T	T	T
T	F	F	T
F	T	F	T
F	F	F	F

Table 2: Truth tables for `AND` and `OR` operators.

p	`NOT` p
T	F
F	T

Table 3: Truth table for `NOT` relation.

6.2 Example 4: Piecewise Function with Conjunction Operators

The previous example of a 2-dimensional piecewise function can be done using an `if-then-elseif` structure combined with the conjunction operators.

```
x = input('Input x: ');
y = input('Input y: ');

if (x >= 0 & y >= 0)
   z = x^2 + y^2;
elseif (x >= 0 & y < 0)
   z = x^2 - y^2;
elseif (x < 0 & y >= 0)
   z = y^2 - x^2;
elseif (x < 0 & y < 0)
   z = -x^2 - y^2;
end
```

Believe it or not, this is all we need to know about the mechanics and syntax of `if` statements. However, programming is like chess. There are not many rules, but it takes much practice to learn how to master them. The way to accomplish this is to write lots of programs. It is easy to write a set of `if` statements that look great, but don't do what you intended them to do. This is another example of a logic error. It is important that you test a more complicated sequence of `if` statements over all the various outcomes to make sure it is producing the proper behavior.

7 Comparing Floating Point Numbers

If you recall from the homework problem where you are testing to see if $c = a+b$, you observed that $c \neq a+b$. Why did this happen?

Recall that all numbers are represented internally in binary form. The binary forms of the values from the homework problem are:

$$\begin{aligned} 1.1_{(10)} &= 1.000\overline{1001}_{(2)} \\ 2.2_{(10)} &= 10.001\overline{1001}_{(2)} \\ 3.3_{(10)} &= 11.0\overline{1001}_{(2)}. \end{aligned}$$

Notice that while each of these numbers has a finite number of digits in decimal (base 10), their binary forms (base 2) have an infinite number of repeating digits. This means that each of these numbers will have a storage error when they are stored in memory. In addition, there may or may not be a rounding error when $a + b$ is computed.

MATLAB allows you to see the exact sequence of binary digits used to represent a given number in memory. This allows us to see the result of the calculation.

```
>> format hex
>> a = 1.1
a =
   3ff199999999999a
>> b = 2.2
b =
   400199999999999a
>> c = 3.3
c =
   400a666666666666   (last 4 = 0110)
>> a+b
ans =
   400a666666666667   (last 4 = 0111)
```

Examining the results above, we see that the value of c and the value of $a + b$ differ by 1 binary digit in the last place. This is enough to cause the test `if (c == a+b)` to have a result of False.

Returning to long formatting, we can make another important observation

```
>> format long e
>> a = 1.1
a =
     1.100000000000000e+00
>> b = 2.2
b =
     2.200000000000000e+00
>> c = 3.3
c =
     3.300000000000000e+00
>> a+b
ans =
     3.300000000000000e+00
```

Note that c and $a+b$ display as the same number in decimal even though we know from the above discussion that c and $a+b$ are different numbers. Just because two values have the same representation when converted to decimal does not mean that the numbers are the same. The reason for this is that there are many binary numbers between any two decimal representations.

The main concept you should take away from this example is that it is never safe to test two double precision (or floating point) numbers for equality. Instead, you should test whether two values are 'close enough' to be considered equal.

The *relative difference* (R) between two values a and b is defined as

$$R(a,b) = \frac{|a-b|}{|a|+|b|}.$$

When testing if the values of a and b are 'close enough,' you should use a test like

```
R = abs(a-b)/(abs(a)+abs(b))
if (R < tol) then
   .....    a and b are close enough
else
   .....    a and b are different
end
```

Here, `tol` is a tolerance that determines how close is 'close enough.'

The difficulty is that you need to determine a value for this tolerance. This frequently depends on the particular calculation being performed. In some cases, a tolerance like 10^{-6} is acceptable while in other cases, a tolerance of 10^{-12} might be necessary. For now you should use a tolerance of 10^{-14} unless indicated otherwise.

7.1 Entering Values in Scientific Notation

Sometimes it is necessary to use scientific notation when you are entering numerical values. For example, if you needed Avagadro's Number ($6.02 \cdot 10^{23}$), you can do

```
>> AG = 6.02e23
```

Similarly, you could enter the universal gravitational constant G ($6.67 \cdot 10^{-11}$) using

```
>> G = 6.67e-11
```

If you need to use 10^{-14} use the syntax

```
>> tol = 1.0e-14
```

Make sure not to use the syntax

```
>> tol = 10.0e-14
```

because this is actually 10^{-13}.