Logical Statements

Jiseok Chae

Department of Mathematical Sciences
KAIST

Week 4

Contents

In MATLAB there are expressions where if the expression is true or not can be determined.

Inside MATLAB, `TRUE` is expressed as a `logical 1`, and `FALSE` is expressed as a `logical 0`.

```
>> 4 < 3

ans =

  logical

   0
```

Comparisons are logical expressions. There are:

- `>` : greater than
- `<` : less than
- `>=` : greater than or equal to
- `<=` : less than or equal to
- `==` : equal to
- `~=` : not equal to

You can compare values contained in variables.

```
>> x = 7 + 2; y = 3 * 3;
>> x ~= 9

ans =

  logical

   0

>> x == y

ans =

  logical

   1
```

There are operations you can do with logical expressions. Let `P` and `Q` be logical expressions. Then,

`P && Q` : `P` and `Q`.

`P || Q` : `P` or `Q`.

`~P` : not `P`.

Standard "common sense" logical operation rules apply. That is,
`P && Q` is `TRUE` if and only if both `P` and `Q` are `TRUE`,
`P || Q` is `TRUE` if and only if either `P` or `Q` is `TRUE`, and
`~P` is `TRUE` if and only if `P` is `FALSE`.

Order of priorities :

$$\sim \rightarrow \texttt{arithmetic operators} \rightarrow \texttt{comparisons} \rightarrow \&\& \rightarrow ||$$

If you are not sure, use parentheses.

```
>> y = 3 ^ 4;
>> (y > 10) && ~(y > 100)

ans =

  logical

   1
```

There are logical vectors; vectors containing logical values.

The most common source of logical vectors is from the *elementwise* application of logical operations.

```
>> v = [2, 7, 1, 8, 2, 8];
>> v > 5

ans =

  1×6 logical array

   0   1   0   1   0   1
```

Logical vectors provide a new way of extracting subvectors and submatrices.

```
>> x = [-3, 4, 1, 0, -2, 5];
>> x(x.^2 < 10)

ans =

    -3     1     0    -2
```

Contents

The well-known *Collatz conjecture*:

> Start with any positive integer n. If n is even, halve it. If n is odd, triple it and add one. If we keep repeating this process, do we always eventually reach the number 1?

A sequence generated by the rule above is called a *Collatz sequence*.

Suppose we want to design a function `collatz_next` which, given a positive integer `n`, computes the next term in a Collatz sequence.

In that case, we need a *conditional statement*, as the value depends on the parity of `n`:

$$\texttt{collatz\_next}(\texttt{n}) = \begin{cases} \frac{\texttt{n}}{2} & \text{if } \texttt{n} \text{ is even,} \\ 3\texttt{n}+1 & \text{if } \texttt{n} \text{ is odd.} \end{cases}$$

The basic template of such conditional statement is:

`if` ⟨*logical expression*⟩
    ⟨*commands to execute if* ⟨*logical expression*⟩ *is* `TRUE`⟩
`else`
    ⟨*commands to execute if* ⟨*logical expression*⟩ *is* `FALSE`⟩
`end`

If there is nothing to do when ⟨*logical expression*⟩ is `FALSE`, the `else` block can be omitted, as:

`if` ⟨*logical expression*⟩
    ⟨*commands to execute if* ⟨*logical expression*⟩ *is* `TRUE`⟩
`end`

The function `rem(a, b)` computes the remainder of `a` when divided by `b`.

So, one way to write the function `collatz_next` would be the following.

collatz_next.m

```
function next = collatz_next(n)
if rem(n, 2) == 0
    next = n / 2;
else
    next = 3 * n + 1;
end
```

What if there are multiple cases to consider?

You can add `elseif` blocks, as :

`if` ⟨*P*⟩
    ⟨*commands to execute if* ⟨*P*⟩ *is* `TRUE`⟩
`elseif` ⟨*Q*⟩
    ⟨*commands to execute if* ⟨*P*⟩ *is* `FALSE` *but* ⟨*Q*⟩ *is* `TRUE`⟩
`else`
    ⟨*commands to execute if both* ⟨*P*⟩ *and* ⟨*Q*⟩ *are* `FALSE`⟩
`end`

You can add `elseif` blocks as many as you want :

```
if ⟨P⟩
   ⟨commands to execute if ⟨P⟩ is TRUE⟩
elseif ⟨Q⟩
   ⟨commands to execute if ⟨P⟩ is FALSE but ⟨Q⟩ is TRUE⟩
elseif ⟨R⟩
   ⟨commands to execute if ⟨P⟩ and ⟨Q⟩ are FALSE but ⟨R⟩ is TRUE⟩
elseif ⟨S⟩
   ...
...
else
   ⟨commands to execute if ⟨P⟩, ⟨Q⟩, ⟨R⟩, ⟨S⟩, ... are all FALSE⟩
end
```

Let us see an example.

A bromothymol blue (BTB) solution is a pH indicator that changes color according to the pH value of the substance it is added to.

Its color turns

- yellow if pH < 6.0,
- green if $6.0 \leq$ pH ≤ 7.6, and
- blue if pH > 7.6.

Let us write a function `BTB(pH)` which takes the input `pH`, and returns the color a BTB solution would turn into.

BTB.m

```
function color = BTB(pH)
if pH < 6.0
    color = 'yellow';
elseif pH > 7.6
    color = 'blue';
else
    color = 'green';
end
```

The quotation marks are used to generate *strings*. A string is a sequence of letters. We will discuss about strings after the midterm period.

Examples of results:

```
>> BTB(pi)

ans =

    'yellow'

>> BTB(7.4)

ans =

    'green'

>> BTB(10)

ans =

    'blue'
```

Thank you!