MATH 3341: Introduction to Scientific Computing Lab

Libao Jin

University of Wyoming

February 10, 2021

Lab 03: Functions and Control Flows

Control Flows

Relational Operators

Symbol	Meaning
==	equal to
~=	not equal
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
`logical 1`	true
`logical 0`	false

Relation Operators: Conditional Statement for Scalars

- `num1 == num2` or `eq(num1, num2)`: `logical 1` if `num1` is equal to `num2`, otherwise `logical 0`.
- `num1 ~= num2` or `ne(num1, num2)`: `logical 1` if `num1` is not equal to `num2`, otherwise `logical 0`.
- `num1 > num2` or `gt(num1, num2)`: `logical 1` if `num1` is greater than `num2`, otherwise `logical 0`.
- `num1 >= num2` or `ge(num1, num2)`: `logical 1` if `num1` is greater than or equal to `num2`, otherwise `logical 0`.
- `num1 < num2` or `lt(num1, num2)`: `logical 1` if `num1` is less than `num2`, otherwise `logical 0`.
- `num1 <= num2` or `le(num1, num2)`: `logical 1` if `num1` is less than or equal to `num2`, otherwise `logical 0`.

Relation Operators: Conditional Statement for Scalars

Example: Compare two scalars:

```
2 <= 2                  % logical 1 (true)
2 >= 2                  % logical 1 (true)
2 > 1                   % logical 1 (true)
2 ~= 1                  % logical 1 (true)
2 == 1                  % logical 0 (false)
'a' == "a"              % logical 1 (true)
'a' == 'b'              % logical 0 (false)
'a' ~= 'b'              % logical 1 (true)
'abc' == "abc"          % logical 1 (true)
```

Relation Operators: Conditional Statement for Arrays

- `comparison = vec == num`: `comparison` is an array of which `comparison(i) = vec(i) == num`.
- `comparison = vec ~= num`: `comparison` is an array of which `comparison(i) = vec(i) ~= num`.
- `comparison = vec > num`: `comparison` is an array of which `comparison(i) = vec(i) > num`.
- `comparison = vec >= num`: `comparison` is an array of which `comparison(i) = vec(i) >= num`.
- `comparison = vec < num`: `comparison` is an array of which `comparison(i) = vec(i) < num`.
- `comparison = vec <= num`: `comparison` is an array of which `comparison(i) = vec(i) <= num`.

Relation Operators: Conditional Statement for Arrays

Example: Compare a vector to a scalar:

```
x =
   [0 1 2 3 4 5 6]
x < 3
% [1 1 1 0 0 0 0]
x <= 3
% [1 1 1 1 0 0 0]
x > 3
% [0 0 0 0 1 1 1]
x >= 3
% [0 0 0 1 1 1 1]
x == 3
% [0 0 0 1 0 0 0]
x ~= 3
% [1 1 1 0 1 1 1]
```

Relation Operators: Conditional Statement for Arrays

Example: Define a piecewise function:

$$h(x) = \begin{cases} f(x) = x + 2 & \text{if } x < 3, \\ g(x) = 6 - x & \text{if } x \geq 3. \end{cases}$$

```
x =                          [0 1 2 3 4 5 6]
x <  3                    % [1 1 1 0 0 0 0]
f =                          [2 3 4 5 6 7 8]  % x + 2
x >= 3                    % [0 0 0 1 1 1 1]
g =                          [6 5 4 3 2 1 0]  % 6 - x
fx = f .* (x <  3)        % [2 3 4 0 0 0 0]
gx = g .* (x >= 3)        % [0 0 0 3 2 1 0]
h = fx + gx               % [2 3 4 3 2 1 0]
```

Relation Operators: Conditional Statement for Arrays

- `comparison = vec1 == vec2`: `comparison` is an array of which `comparison(i) = vec1(i) == vec2(i)`.
- `comparison = vec1 ~= vec2`: `comparison` is an array of which `comparison(i) = vec1(i) ~= vec2(i)`.
- `comparison = vec1 > vec2`: `comparison` is an array of which `comparison(i) = vec1(i) > vec2(i)`.
- `comparison = vec1 >= vec2`: `comparison` is an array of which `comparison(i) = vec1(i) >= vec2(i)`.
- `comparison = vec1 < vec2`: `comparison` is an array of which `comparison(i) = vec1(i) < vec2(i)`.
- `comparison = vec1 <= vec2`: `comparison` is an array of which `comparison(i) = vec1(i) <= vec2(i)`.

Relation Operators: Conditional Statement for Arrays

Example: Compare two vectors:

```
x =                                    [1 2 3 4 5 6]
y =                                    [3 2 1 6 5 4]
x == y                                 % [0 1 0 0 1 0]
x ~= y                                 % [1 0 1 1 0 1]
x >  y                                 % [0 0 1 0 0 1]
x >= y                                 % [0 1 1 0 1 1]
x <  y                                 % [1 0 0 1 0 0]
x <= y                                 % [1 1 0 1 1 0]
'abc' == ['a', 'b', 'c']               % [1 1 1]
'abc' == ['a', 'b', 'd']               % [1 1 0]
```

Logical Operators

Symbol	Meaning
&	element-wise logical AND
|	element-wise logical OR
&&	short-circuit logical AND
||	short-circuit logical OR
~	logical NOT

Logical Operators

- `condition1 & condition2`: `logical 1` if both `condition1` and `condition2` are `logical 1`, otherwise, `logical 0`.
- `condition1 | condition2`: `logical 1` if either `condition1` or `condition2` is `logical 1`, otherwise, `logical 0`.
- `~condition`: `logical 1` if `condition` is `logical 0`, otherwise, `logical 1`.
- `condition1 && condition2`: same as `condition1 & condition2` but `condition2` will be skipped if `condition1` is `logical 0`.
- `condition1 || condition2`: same as `condition1 & condition2` but `condition2` will be skipped if `condition1` is `logical 1`.

Conditional Branch

This kind of control flow executes a set of statements only if some condition is met. There are `if` statements and `switch` statements in MATLAB.

Conditional Branch: `if` Statements

`if` statements conditionally execute statements. The general forms of the if statement are

- Single branch

```
if   conditionIsMet
     blockStatements
end
```

- Two branches

```
if   conditionIsMet
     blockStatements1
else
     blockStatements2
end
```

Conditional Branch: `if` Statements

`if` statements conditionally execute statements. The general forms of the if statement are

- N branches

```
if  conditionIsMet1
    blockStatements1
elseif conditionIsMet2
    blockStatements2
elseif conditionIsMet3
    blockStatements3
  .
  .
  .
else
    blockStatementsN
end
```

Conditional Branch: `if` Statements

Example: Check whether `n` is even.

```
n = 5;
if mod(n, 2) == 0
    disp('n = 5 is an even number');
else
    disp('n = 5 is an odd number');
end
```

Conditional Branch: `if` Statements

Example: Check whether `year` is a leap year.

```
year = 2020;
if mod(year, 400) == 0
    is_leap_year = true;
elseif mod(year, 4) == 0 && mod(year, 100) ~= 0
    is_leap_year = true;
else
    is_leap_year = false;
end
```

Conditional Branch: `if` Statements

Example: Check whether `year` is a leap year.

Combining the first and second branches:

```
year = 2020;
if mod(year, 400) == 0 || ...
   (mod(year, 4) == 0 && mod(year, 100) ~= 0)
    is_leap_year = true;
else
    is_leap_year = false;
end
```

Conditional Branch: `switch` Statements

`switch` statements switch among several cases based on expression. The general form of the switch statement is

```
switch switch_expr
    case case_expr1,
        blockStatements1
    case {case_expr2, case_expr3, ..., case_exprN}
        blockStatements2
      .
      .
      .
    otherwise,
        blockStatementsN
end
```

Conditional Branch: `switch` Statements

Example: Check whether `day` is weekday.

```
day = 'Monday';
switch day
    case {'Monday', 'Tuesday', 'Wednesday', ...
            'Thursday', 'Friday'}
        fprintf('%s is weekday.\n', day);
    otherwise
        fprintf('%s is weekend.\n', day);
end
```

Bug: What about `day = 'Sunnyday'`?

Conditional Branch: `switch` Statements

Example: Check whether `day` is weekday.

Fix our first bug by adding a new case:

```
day = 'Monday';
switch day
    case {'Monday', 'Tuesday', 'Wednesday', ...
          'Thursday', 'Friday'}
        fprintf('%s is weekday.\n', day)
    case {'Saturday', 'Sunday'}
        fprintf('%s is weekend.\n', day)
    otherwise
        fprintf('Error!\n')
end
```

Loop

A loop executes a set of statements zero or more times, until some condition is met. There are `for` and `while` loops in MATLAB.

Loop

Question: What should we do if we want to `disp('Repeating is BORING!')` for 100 times?

```
disp('Repeating is BORING!')
disp('Repeating is BORING!')
disp('Repeating is BORING!')
   .
   .
   .
disp('Repeating is BORING!')
```

Loop: for-loop

A for-loop repeats statements a specific number of times. The general form of a for statement is:

```
for loopCounter = expr
    blockStatements
end
```

Loop: for-loop

Example: Loop over an array and display the value of each entry:

```
array = [1,3,5,7]
for i = 1:length(array)
    disp(array(i))
end
```

or

```
array = [1,3,5,7]
for i = array
    disp(i)
end
```

Loop: for-loop

Example: Loop over an array and display the value of each entry:

```
for i = [1,3,5,7]
    disp(i)
end
```

or

```
for i = 1:2:7
    disp(i)
end
```

Loop: for-loop

Question: What should we do if we want to `disp('Repeating is BORING!')` for 100 times?

Use a `for`-loop:

```
for i = 1:100
    disp('Repeating is BORING!')
end
```

Loop: `while`-loop

A `while`-loop repeats statements an indefinite number of times. The general form of a while statement is:

```
while conditionIsMet
    blockStatements
end
```

Loop: while-loop

Question: What should we do if we want to `disp('Repeating is BORING!')` for 100 times?

Use a `while`-loop:

```
i = 1;
while i <= 100
    disp('Repeating is BORING!')
    i = i + 1;
end
```

Functions

Anonymous Functions

- An *anonymous function* is a function associated with a variable whose data type is `functionHandle`. Anonymous functions can accept inputs and return outputs.
- To define an anonymous function:

  ```
  functionHandle = @(variableList) expression
  ```

- Except for the cases when the function is meant to perform matrix operations, the operators in the `expression` would usually be element-wise operators , e.g., `.*`, `./`, `.^`. We usually assume that the inputs are arrays rather than just scalars.

Anonymous Functions

- Anonymous function of one variable:
  - $f(y) = \sin(y)$

    ```
    f = @(y) sin(y)
    ```
- Anonymous function of two variables:
  - $g(x, y) = x^2 + y^2 - 1$

    ```
    g = @(x, y) x .^ 2 + y .^ 2 - 1
    ```
- Composition of anonymous functions:
  - $h(z) = e^{\sin z} = e^{f(z)}$

    ```
    h = @(z) exp(f(z))
    same as h = @(z) exp(sin(z))
    ```

Anonymous Functions

Example: Define a piecewise anonymous function:

$$h(x) = \begin{cases} f(x) = x + 2 & \text{if } x < 3, \\ g(x) = 6 - x & \text{if } x \geq 3. \end{cases}$$

```
x =                          [0 1 2 3 4 5 6]
x <  3                     % [1 1 1 0 0 0 0]
f =                          [2 3 4 5 6 7 8]   % x + 2
x >= 3                     % [0 0 0 1 1 1 1]
g =                          [6 5 4 3 2 1 0]   % 6 - x
fx = f .* (x <  3)         % [2 3 4 0 0 0 0]
gx = g .* (x >= 3)         % [0 0 0 3 2 1 0]
hx1 = fx + gx              % [2 3 4 3 2 1 0]

h = @(y) (y + 2) .* (y < 3) + (6 - y) .* (y >= 3)
hx2 = h(x)                 % same as hx1
```

Function Files

Defining functions can save you from writing the same code over and over again. Here is the syntax to define a function:

```
function [outputList] = functionName(inputList)
%FUNCTIONNAME Summary of the function
% Details of the function goes here such as
% syntax, author, date, copyright info, and etc.

% function body goes here
% define every variable in the outputList
% using variables in the inputList

end
```

Function Files: Naming Convention

The naming convention of function files is similar to that of script files. However, it is strongly recommended that the function name of function definition should be same as the filename.

For example, if we define a function with header

```
function thisIsAFunction(a, b, c)
```

then it should be store to a file named `thisIsAFunction.m`. If the function name and the file name are not consistent, MATLAB would take the file name as the function name.

Function Files: sumProd

```
function [summation, product] = sumProd(x)
%SUMPROD Calculate the summation and product of
% all elements in x
% Syntax:
%   [summation, product] = sumProd(x)
%   summation = sumProd(x)

% Initialize variables summation and product
summation = 0;
product = 1;
for i = 1:length(x)
    summation = summation + x(i);
    product = product * x(i);
end

end
```

Function Files: `isEven`

Recall the script for the example: Check whether `n` is even.

```
n = 5;
if mod(n, 2) == 0
    disp('n = 5 is an even number');
else
    disp('n = 5 is an odd number');
end
```

Function Files: isEven

We can convert the script to a function as below:

```
function isNEven = isEven(n)
%ISEVEN Check whether n is even

isNEven = mod(n, 2) == 0;
if isNEven
    fprintf('n = %d is an even number', n);
else
    fprintf('n = %d is an odd number', n);
end

end
```

Then we can call the function: `is4Even = isEven(4)`.

Function Files: `isLeapYear`

Recall the script for the example: Check whether `year` is a leap year.

```
year = 2020;
if mod(year, 400) == 0
    is_leap_year = true;
elseif mod(year, 4) == 0 && mod(year, 100) ~= 0
    is_leap_year = true;
else
    is_leap_year = false;
end
```

Function Files: isLeapYear

We can convert it to a function as below:

```
function is_leap_year = isLeapYear(year)
%ISLEAPYEAR: Check whether year is a leap year.

if mod(year, 400) == 0
    is_leap_year = true;
elseif mod(year, 4) == 0 && mod(year, 100) ~= 0
    is_leap_year = true;
else
    is_leap_year = false;
end

end
```

Then we can call `is_2020_leap_year = isLeapYear(2020)`.

Function Files: isWeekday

Recall the script for the example: Check whether day is weekday.

```
day = 'Monday';
switch day
    case {'Monday', 'Tuesday', 'Wednesday', ...
          'Thursday', 'Friday'}
        fprintf('%s is weekday.\n', day)
    case {'Saturday', 'Sunday'}
        fprintf('%s is weekend.\n', day)
    otherwise
        fprintf('Error!\n')
end
```

Function Files: isWeekday

We can convert it to a function as below:

```
function isWeekday(day)
%ISWEEKDAY Check whehter day is a weekday.

switch day
    case {'Monday', 'Tuesday', 'Wednesday', ...
          'Thursday', 'Friday'}
        fprintf('%s is weekday.\n', day)
    case {'Saturday', 'Sunday'}
        fprintf('%s is weekend.\n', day)
    otherwise
        fprintf('Error!\n')
end

end
```

Anonymous Function vs. Function File

- Anonymous functions are helpful when you are using functions with a simple definition.
- Otherwise, writing a function file is recommended.