MATLAB Functions and Conrol Structures

Maria Ptashkina Damian Romero

Barcelona Graduate School of Economics

September 2020

Agenda

Outline

Introduction

So far, we have used the built-in functions. Yet, any programming language cannot possibly contain all the functions a user might need

What if we want to issue more than a single command at a time? How can we create our own functions?

A very important characteristic of programming languages to support the creation of user-defined functions

Outline

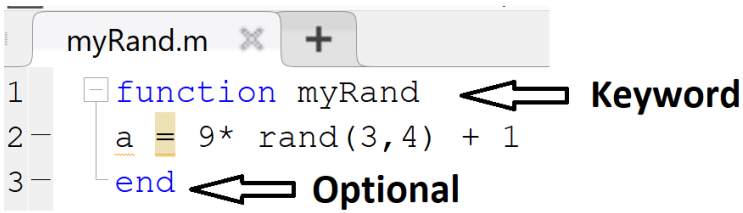
Basics

A built-in function `rand` generates random numbers from a uniform 0-1 distribution. What if we need a 3×4 matrix of numbers from 1 to 10?

We can create our own function in the Command Window `a = 9 * rand(3,4) + 1`, but calling it every time is annoying

Type `edit myRand` in the Command Window (or open a new script)

This opens a text editor with which we can enter our code



Important! Name your script (`m-file`) with the same name as the function! Always save your function before running!

Note: In MATLAB functions are separate files which have to locate in the current working directory (different from some other languages which keep functions in the memory)

Calling a Function

We can run our function

```
>> myRand
a =
    8.8783    6.7394    7.0851    7.2563
    5.6625    9.6192    3.6016    1.6119
    9.4926    3.1664    7.0463    3.2931
>> a
Unrecognized function or variable 'a'.
```

Wait, what? Why is a not defined?

The reason is that functions are like Las Vegas. What happens in a function stays in the function. It does not affect your workspace (this is important in programming logic)

Variables inside functions are called "local variables": they are accessible only by statements inside the function, and they exist only during the function call

Function Output and Input

So, function output is a local variable, and it disappears when the call is finished. But you can assign a value, eg. `a = myRand`

We can supply *parameters* to the function when we call it. These parameters are called "input arguments" (they are also local variables)

```
function a = myRand(low, high)
a = (high-low) * rand(3,4) + low;
```

Command Window

```
>> myRand(10,100)
ans =
   56.6735   82.0298   84.2782   25.6050
   97.5677   50.8418   17.5123   45.1844
   68.4092   48.9152   21.9854   84.8242
>> x = 0.5;
>> y = 0.6;
>> z =myRand(x,y)
z =
    0.5803    0.5527    0.5628    0.5015
    0.5060    0.5417    0.5292    0.5984
    0.5399    0.5657    0.5432    0.5167
```

Formal Definition

Multiple Outputs and Subfunctions

Modify `myRand` to also the sum of all elements and get multiple output

myRand.m

```
1  function [a, s] = myRand(low, high)
2    a = (high-low) * rand(3,4) + low;
3    s = sum(a(:));
```

Command Window

```
>> [x y] = myRand(0,1)
x =
    0.8555    0.1909    0.1206    0.3846
    0.6448    0.4283    0.5895    0.5830
    0.3763    0.4820    0.2262    0.2518
y =
    5.1335
```

We can also do this using subfunctions

myRand.m

```
1  function [a, s] = myRand(low, high)
2    a = (high-low) * rand(3,4) + low;
3    s = sumAllElements(a);
4  function summa = sumAllElements(M)
5    summa = sum(sum(M));
```

Anonymous Functions

If you have a very long code, you might not want to create a zillion `.m` files with functions

You can store multiple *function handles* in an array, and save and load them, as you would any other variable

An *anonymous function* is a one-line expression-based MATLAB function that does not require a program file, and you can create a handle to it

To create an anonymous function use the following syntax

```
sqr = @(n) n.^2;
x = sqr(3);
```

Exercise 1

Our current `myRand` function creates a 3-by-4 matrix of random numbers between the limits specified by the input arguments `low` and `high`

myRand.m

```
1  function [a, s] = myRand(low, high)
2  a = (high-low) * rand(3,4) + low;
3  s = sum(a(:));
```

1. Write a more general version of the function that *gets the dimensions as input arguments*, and then sums all the elements inside a matrix
2. Run this function for
   - a 2×4 matrix
   - for a row vector of length 3
   - for a column vector of length 10

Exercise 1: Solution

myRand.m

```
1  function [a, s] = myRand(low, high, rows, cols)
2  a = (high-low) * rand(rows, cols) + low;
3  s = sum(a(:));
4  end
```

```
[R ss] = myRand(0,5,2,4)
[R ss] = myRand(10,20,1,3)
[R ss] = myRand(0.01,0.02,10,1)
```

Outline

Control Structures: Selection

Executing the statements in the order that they were written by the programmer is called *sequential control*

Sequential control is the most natural and the most common sequence that occurs in any program written in any programming language, and it is the primary example of a *control construct*

A control construct is simply a method by which MATLAB (or other language) selects the next statement to be executed after the execution of the current statement has concluded

We will learn how to tell MATLAB to make decisions about the order of executing statements based on the values, which is called selection (branching in computer science)

if-statements

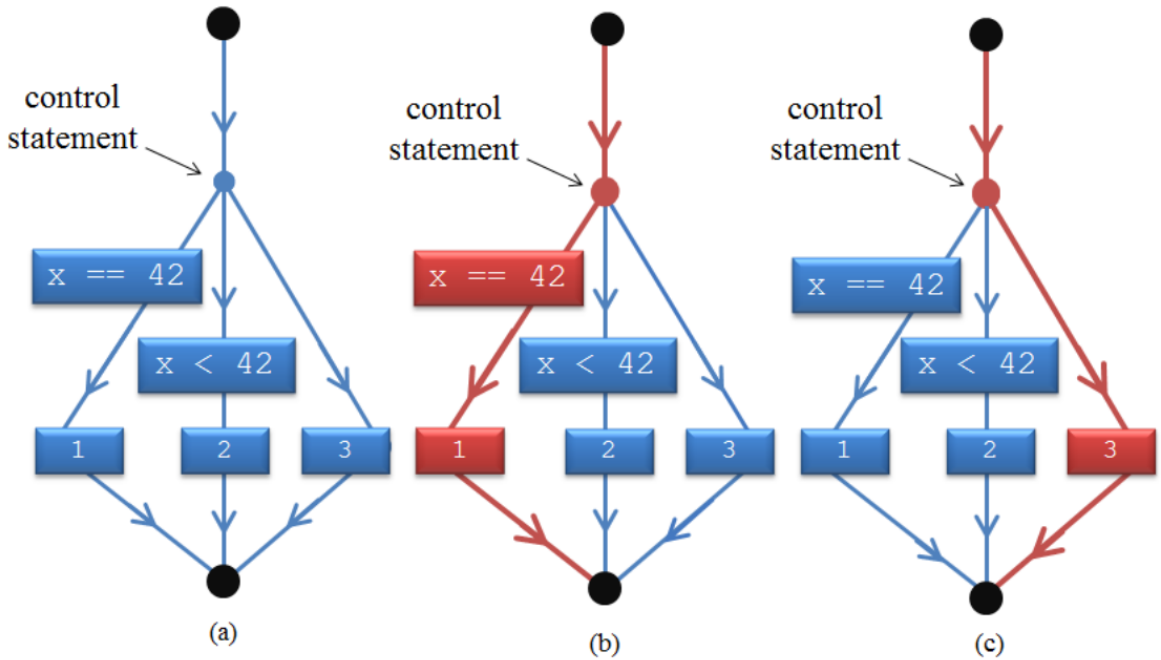
Simple example of a function using a selection control structure (you don't necessarily need to write a function if you don't want to reuse the statement)

guess_my_number.m

```
1  function guess_my_number(x)
2    if x == 42
3    fprintf('Wow! You guessed by number!\n');
4    elseif x < 2
5    fprintf('Too small. Try again.\n');
6    else
7    fprintf('Too big. Try again.\n');
8    end
```

We have introduced new keywords here — `if`, `elseif` and `else` — and a new operator — the double equals == (in a few slides we'll explain)

The Logic of Selection Control Structures



There are three possible paths as shown in (a): one for `x == 42`, one for `x < 42`, and one for every other possibility. If `x` equals 42, then the left branch is followed and block 1 is executed, as shown by the red path in (b). If `x` neither equals 42, nor is less than 42, then block 3 is executed, as shown in (c). The red path for `x < 42` is not shown but is analogous.

If-statement Summary

There are four common selection constructs: if-statements, if-else statements, if-elseif statements and if-elseif-else statements

```
if conditional
   block
end
```

```
if conditional
   block
else
   block
end
```

```
if conditional
   block
elseif conditional
   block
end
```

```
if conditional
   block
elseif conditional
   block
else
   block
end
```

Exercise 2

1. Create any two matrices `A` and `B`
2. Write an if statement which prints "Matrix A and B contain the same elements" in case matrices `A` and `B` are equal, and "Matrix A and B are different" in case they are not

Exercise 2: Solution

```
A = [0.4*ones(1,3);0.35*ones(2,2) 0.8*ones(2,1)];
B = 0.3*ones(3,3);

if A==B;
    disp('Matrix A and B contain the same elements')
else
    disp('Matrix A and B are different')
end
```

Note that you can also use a function `disp` instead of `fprintf`

Relational and Logical Operators

The operators, ==, and <, which we have seen in if-statements, are examples of *relational operators*

A relational operator produces a value that depends on the relation between the values of its two operands

- == is equal to
- = is not equal to
- >, < larger/smaller than
- >=, <= larger or equal/smaller or equal than

A *logical operator* produces a value that depends on the truth of its two operands

- & logical 'and'
- | logical 'or'
- $\sim$ logical negation

Logical Operators and Short Circuiting

Relational and Logical Operators Examples

Relational and logical operators can be used outside control structures. For example,

- to verify combination of statements

```
>> ([1,2,3,4]>1)&([1,2,3,4]<=3)
ans =
  1×4 logical array
   0   1   1   0
```

- to address elements using logical expressions

```
>> Q = [-1, 3, 4, -5; 6, -2, 8, 9];
>> between=Q(Q>1 & Q<6)
between =
     3
     4
```

- to change elements of a matrix

```
>> Q(1,Q(1,:)<0)=0
Q =
     0     3     4     0
     6    -2     8     9
```

Relational and Logical Operators and Functions

There are also some useful function for logical statements (you can use `help` to check them out): `isequal(A,B)`, `isnan(A)`, `all(A, dim)`, `any(A, dim)`. Some can be also nicely combined with logical operators

```
>> Q = [-1, 3, 4, -5; 6, -2, 8, 9];
>> Q(:,any(Q<0,1))=[]
Q =
     4
     8
```

Function `find` allows to find the indices (positions) of elements of a matrix or a vector that satisfy certain conditions

```
>> Q = [0, 3, 4, 0; 6, 0, 8, 9];
>> [I,J]=find(Q>=2 & Q<5)
I =
     1
     1
J =
     2
     3
```

Exercise 3

1. Create two 3-by-4 matrices as follows `g = [1 2 3 4; 5 6 7 8; 9 10 11 12]` and `h = [3 3 4 4; 5 5 6 6; 7 7 8 8]`
2. Create a logical array called `eq` which is going to take value 1 where elements of matrices `h` and `g` are equal, and 0 otherwise
3. Create a logical array called `bigger` which is going to take value 1 where elements of matrix `g` larger or equal to elements of `h`, and 0 otherwise
4. Create a logical array called `com` which is going to take value 1 where elements of matrix `g` are equal or strictly larger than elements of `h`, and 0 otherwise

Exercise 3: Solution

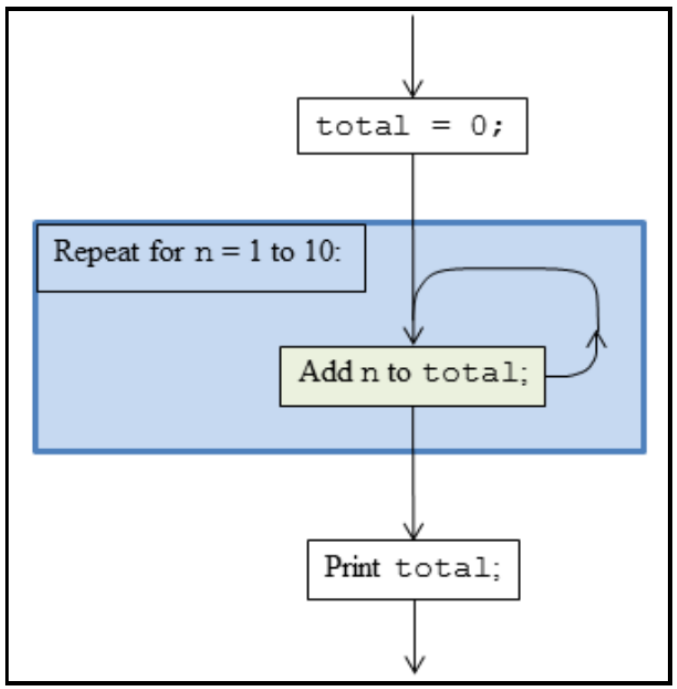
```
>> g = [1 2 3 4; 5 6 7 8; 9 10 11 12];
>> h = [3 3 4 4; 5 5 6 6; 7 7 8 8];
>> eq =(g==h);
>> bigger = (g >= h);
>> com = (g == h) | (g > h);
```

The Loop Concept

The loop is a control construct that can stipulate any number of executions of a block of statements

Sounds complicated... But in fact you've already been using loops without knowing it: in MATLAB almost every operator or built-in function that carries out a numerical calculation uses loops!

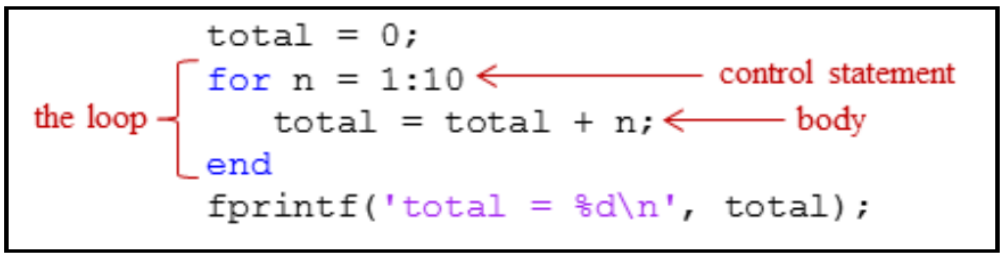
For example, this is what is behind the scene when you type `sum(1:10)`



A bit of Terminology and a for-loop

One execution of the body of a loop is called an *iteration* of the loop

Picture presents the example of a `for-loop`



A variable that is changed by the control statement of a loop is called a loop *index*

A good practice is *pre-allocation* of space for an entire array before calculating the values to put into the array

for-loop Example

Let's use a loop to create a vector the elements of a which are growing at a constant rate (20%)

```
c=nan(20,1); % pre-allocate the vector
c(1,1)=3; % initialize the first element
for i=2:length(c) % specify indexes
    c(i)=c(i-1)*(1.2); % issue command
end
```

`for-loop` is particularly useful to build vectors which elements are functions of the previous element

Exercise 4

Write a function called `vector_square` that takes one vector as an input argument and returns one row vector as output. If it is called like this, `v2 = vector_square(v1)` then `v2 = v1.^2`. However, this function must not use an array operation. It must instead use a for-loop.

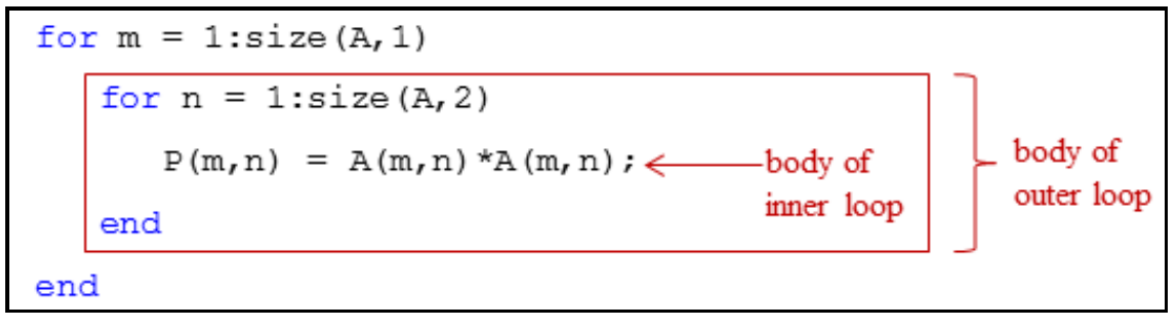
Exercise 4: Solution

```
function a = vector_square(x)
for ii = 1:length(x)
    a(ii) = x(ii)^2;
end
```

This exercise demonstrates that every array operation and every matrix operation can be translated into an equivalent for-loop version

Nested for-loops

Nested for-loops often occur when we are doing things with two-dimensional arrays



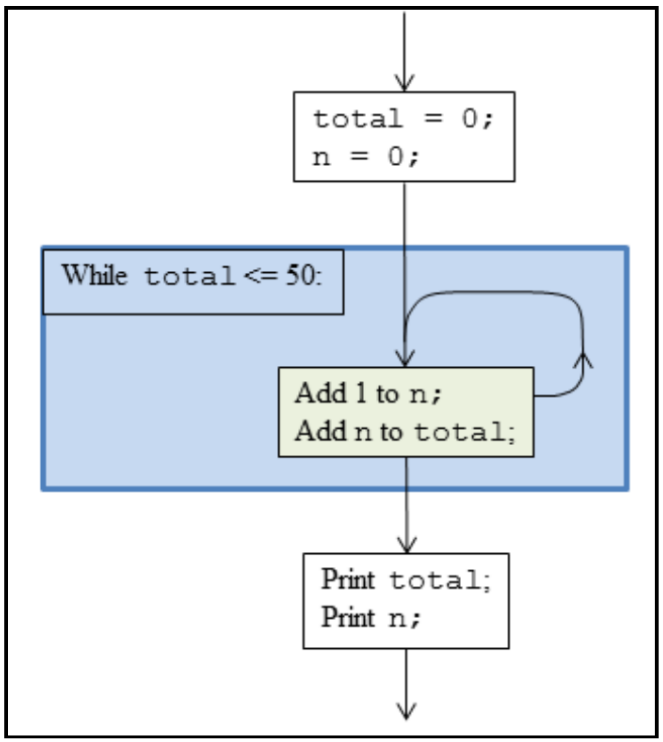
Example: creates an array of products of integers

```
N = 3000;
A = zeros(N);
for jj = 1:N
    for ii = 1:N
        A(ii,jj) = ii*jj;
    end
end
```

Saving Memory Allocation Time

A while-loop

Remember how `sum(1:10)` could be written as a `for-loop`? What if instead we wanted to sum the positive integers until we reached the first sum that is greater than 50?



Example

```
epsilon=1;
while (1+epsilon)>1.1
    epsilon=epsilon/2;
end
```

A while-loop

```
y = x;
while abs(y^2 - x) > 0.001*x
y = (x/y + y)/2;
end
```

To see what this loop has accomplished compute the square of the last `y=43.0092`

Since the square of `y` is approximately equal to `x`, we see that the loop sets `y` equal to an approximation of the square root of `x`

It is (almost) always a good idea to put a limit on the number of iterations to be performed by a while loop

Breaking a Loop

A while-loop can be broken if certain conditions are met using the command `break`

This command allows to "jump out" of the operational cycle implied by the loop The typical directive for this takes the form

```
while condition1
statements;
if condition2
break;
end;
end;
```

Note! MATLAB has no mechanism to terminate multiple loops. **Break** will only terminate the inner loop! The only way to cause the outer loop to terminate is to include a statement in the inner loop that writes a 'note' and another statement in the outer loop that reads it. That 'note', which in common programming terminology is called a **flag** which is a value indicating a special condition.

Infinite Loops And Control-c

A loop that continues iterating without any possibility of stopping is called an *infinite loop*

```
k = 0 ;
while true
    % useful code here
    k = k + 1 ;
    disp(k)
end
```

The most common mistake is that the programmer forgets to include a statement to increment a counter in the body of the loop

In these cases use `Control+C` to stop the loop

Exercise 5

Write a script that will use the random-number generator `rand` to determine the number of random numbers it takes to add up to 20

Hint: the logic is to draw random numbers (counting how many you draw), and add them up until the total sum reaches 20

Exercise 5: Solution

```
total = 0;              % initialize current sum (the test variable)
count = 0;              % initialize the counter (output of the program)
while total < 20        % loop until 20 is exceeded
    count = count + 1; % another loop repeat => another number added
    x = rand(1,1);
    total = total + x; % modify the test variable!
end
disp(['It took ',int2str(count),' numbers this time.'])
```

Outline

Formal Definition

A function declaration is `function [ out_arg1, out_arg2, ...] = function_name (in_arg1, in_arg2, ...)`
where `function` is a keyword and `function_name` must start with a letter and can contain only letters, digits and underscores

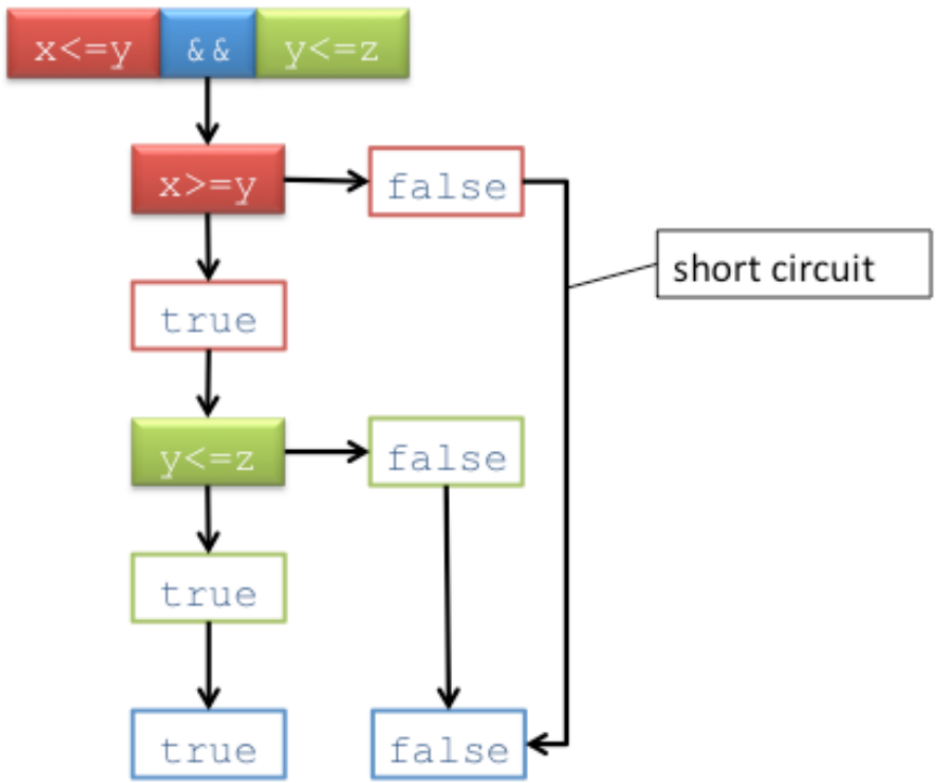
Each of the following is a valid function declaration

```
function func
function func(in1)
function func(in1, in2)
function out1 = func
function out1 = func(in1)
function [out1, out2] = func
function [out1, out2] = func(in1, in2)
```

Back

Logical Operators and Short Circuiting

`&` `&` and `|` `|` are logical operators which help MATLAB's functions and control statements work faster (in case you put single `&` or `|` MATLAB will make a suggestion)



Back

Saving Memory Allocation Time

To make MATLAB run faster (especially when you are solving large problems) it's better to write your code efficiently

- Pre-allocation is always a good idea (sometimes not possible if you don't know the size of the resulting array)
- Re-organizing to reduce re-allocation time: remember that MATLAB uses a column-major order? You can speed up your code by following its logic and, for example, place the colomns in the outer loop and rows in the inner loop

Back