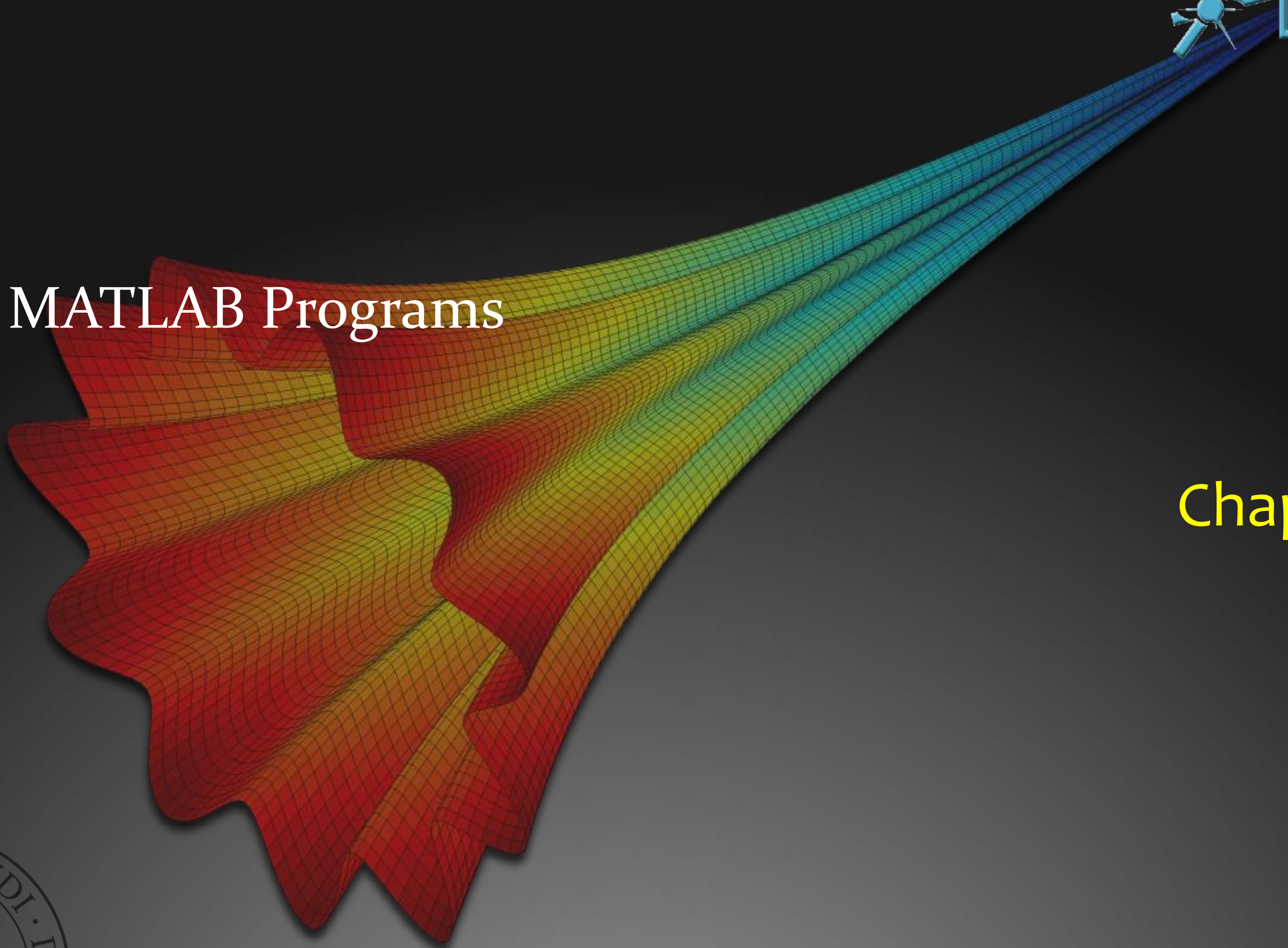
MATLAB Programs



Chapter 6

Linguaggio Programmazione Matlab-Simulink (2017/2018)

Types of Functions

- Categories of functions:
  - functions that calculate and return one value
  - functions that calculate and return more than one value
  - functions that just accomplish a task, such as printing, without returning any values
- They are different in:
  - the way they are called
  - what the function header looks like
- All are stored in code files with the extension .m

Generic Function Definition

- All function definitions consist of:
  - The function header
    - The reserved word **function**
    - Output arguments and the assignment operator (only if the function returns value(s)
    - Function name and input arguments
  - A block comment describing the function
  - The body of the function which includes all statements, including putting values in all output arguments, if there are any
  - **end**

Functions that Return >1 Value

- General form of a function that returns more than one value; it has multiple output arguments in the header
- The output arguments are separated by commas

functionname.m

```
function [output arguments] = functionname(input arguments)
% Comment describing the function
Statements here; these must include putting values in all
of the output arguments listed in the header
end
```

Calling the function

- Since the function is returning multiple values through the output arguments, the function call should be in an assignment statement with multiple variables in a vector on the left-hand side (the same as the number of output arguments in the function header) in order to capture all of them
- Otherwise, some will be lost

Example Function Call

- For example, if the function header is:

```
function [x,y,z] = fnname(a,b)
```

- This indicates that the function is returning 3 things, so a call to the function might be (assuming a and b are numbers):

```
[g,h,t] = fnname(11, 4.3);
```

- Or using the same names as the output arguments (it doesn’t matter since the workspace is not shared):

```
[x,y,z] = fnname(11, 4.3);
```

- This function call would only get the first value returned:

```
result = fnname(11, 4.3);
```

A function *tworan* that returns two random integers, each in the range from 10 to 20

tworan.m

```
function [ranx, rany] = tworan
ranx = randi([10,20]);
rany = randi([10,20]);
end
```

Example Function call:

```
[x, y] = tworan
```

A function *tworanb* that receives two integer arguments a and b and returns two random integers, each in the range from a to b

tworanb.m

```
function [ranx, rany] = tworanb(a,b)
ranx = randi([a,b]);
rany = randi([a,b]);
end
```

Example Function call:

```
[x, y] = tworanb(5, 50)
```

Functions that do not return anything

- A function that does not return anything has no output arguments in the function header, nor does it have the assignment operator
- The statements in the body would typically display or plot information from the input arguments

functionname.m

```
function functionname(input arguments)
% Comment describing the function
  statements here
end
```

Calling a function with no output

- Since no value is returned, the call to such a function is a statement
- For example, if this is the function header:

```
function fnname(x,y)
```

- A call to the function might look like this:

```
fnname(x,y)
```

- This would NOT be a valid call; since the function is not returning anything, there is no value to assign:

```
result = fnname(x,y);  % Invalid!
```

A function *prttworan* that prints two random integers, each in the range from 10 to 20

prttworan.m

```
function prttworan
fprintf( ‘One is %d\n’ , randi([10,20]))
fprintf( ‘The other is %d\n’ , randi([10,20]))
end
```

Example Function call:

```
prttworan
```

A function *prttworanb* that receives two integer arguments a and b and prints two random integers, each in the range from a to b

prttworanb.m

```
function prttworanb(a,b)
fprintf( ‘One is %d\n’ , randi([a,b]))
fprintf( ‘The other is %d\n’ , randi([a,b]))
end
```

Function call:

```
prttworanb(5,50)
```

Notes on Functions

- You do not always have to have input arguments to a function. If you do not, you can have (both in the function header and in the function call) empty (), or you can just leave them out
- The function header and function call have to match up:
  - the name has to be the same
  - the number of input arguments must be the same
  - the number of variables in the left-hand side of the assignment should be the same as the number of output arguments
  - if there are no output arguments, the function call is a statement
- Functions that return values do not normally print them, also – that is left to the calling function/script

Subfunctions

- When one function calls another, the two functions can be stored in the same code file with the same name as the primary function

primary.m

```
primary function header
    primary function body includes call to subfunction
end
subfunction header
     subfunction body
end
```

- The subfunction can only be called by the primary function

Example: Modular outline

- In a modular program, a script calls functions
- Given the following script (where x,y,z are 3 things)

```
 [x,y,z] = getinputs;
result = calcstuff(x,y,z);
displayit(x,y,z, result)
```

- With just that information, we can write the corresponding function headers (not the definitions, just the headers)

Example function headers

- function [x,y,z] = getinputs
- function result = calcstuff(x,y,z)
- function displayit(x,y,z, result)

Types of Errors

- *Syntax errors*: mistakes in language e.g. missing quote at the end of a string
- *Run-time* (or execution-time) errors: errors that are found during execution of a script or function, e.g. referring to an element in a vector that does not exist
- *Logical errors*: mistakes in reasoning e.g. using an expression like (0 < x < 10)

Debugging Methods

- There are several methods that can be used to find errors:
  - *Tracing*: using the **echo** statement which will show all statements as executed
  - Using MATLAB' s Editor/Debugger
  - Set breakpoints so values of variables/expressions can be examined at various points
    - **dbstop** sets a breakpoint
    - **dbcont** continues execution
    - **dbquit** quits debug mode

Code Cells and Publishing

- Code in scripts can be broken into sections called ***code cells***
- You can run one code cell at a time
- Code cells are created with comments that start with two %%
- Code in code cells can also be published in HTML format with plots embedded and with formatted equations
- Do this from the Publish tab in the Editor

Programming Style Guidelines

- If arguments are passed to a function in the function call, do not replace these values by using **input** in the function itself.
- Functions that calculate and return value(s) will not normally also print them.
- Functions should not normally be longer than one page in length
- Do not declare variables in the Command Window and then use them in a script, or vice versa.
- Pass all values to be used in functions to input arguments in the functions.

Exercises

- Write a function *perimarea* that calculates and returns the perimeter and area of a rectangle. Pass the length and width of the rectangle as input arguments.
- Write a function that receives a vector as an input argument and prints the individual elements from the vector in a sentence format.
- Write a function that will prompt the user for a string of at least one character, loop to error-check to make sure that the string has at least one character, and return the string.

Exercises

- For a right triangle with sides *a*, *b*, and *c*, where *c* is the hypotenuse and θ is the angle between sides a and c, the lengths of sides *a* and *b* are given by:

$$a = c * \cos(\theta)$$

$$b = c * \sin(\theta)$$

Write a script *righttri* that calls a function to prompt the user and read in values for the hypotenuse and the angle (in radians), and then calls a function to calculate and return the lengths of sides *a* and *b*, and a function to print out all values in a sentence format.

Exercises

- Modify the *readradius* function to error-check the user's input to make sure that the radius is valid. The function should ensure that the radius is a positive number by looping to print an error message until the user enters a valid radius.

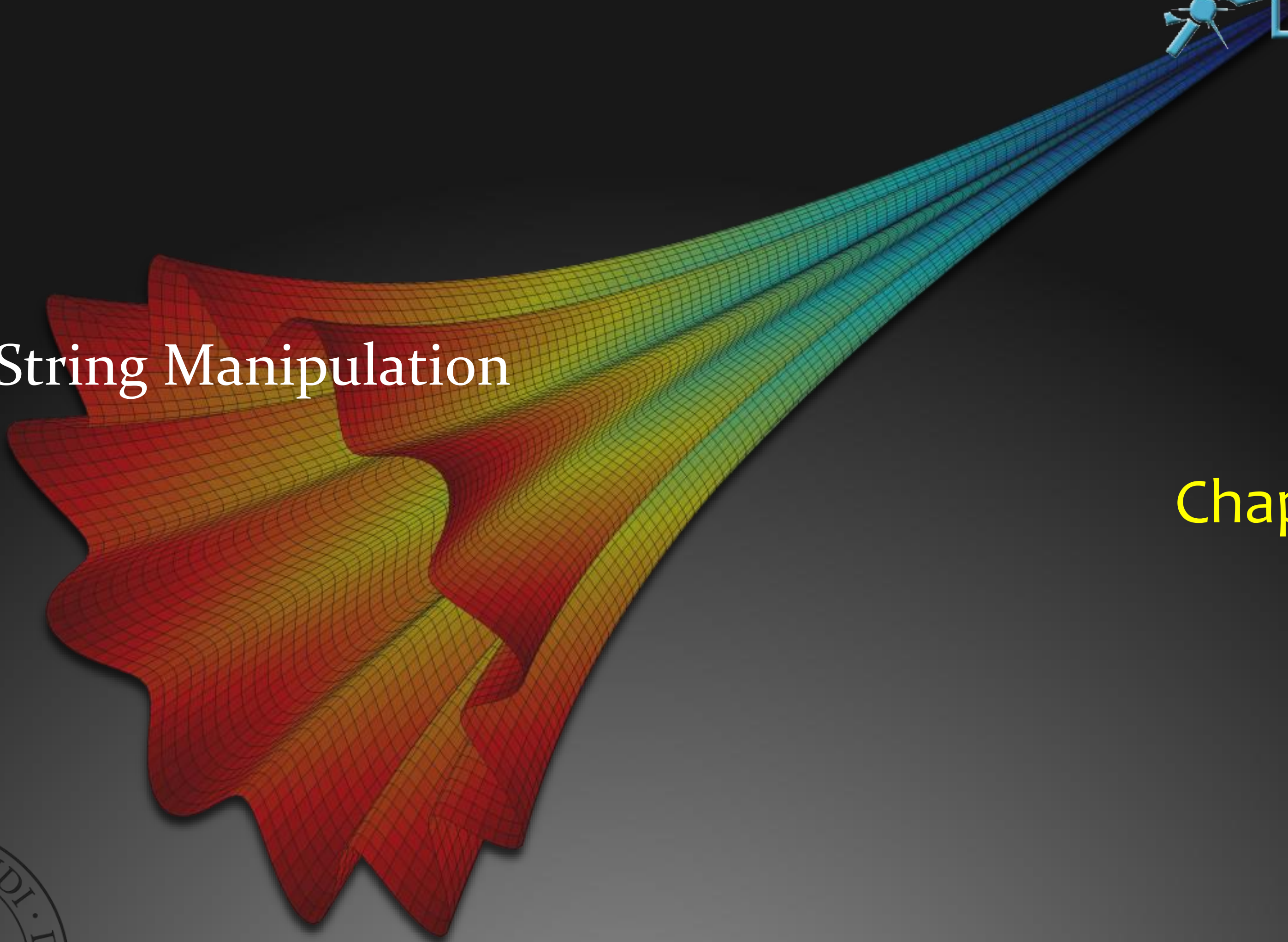
Exercises

- The following script is bad code in several ways. Use **checkcode** first to check it for potential problems, and then use the techniques described in this section to set breakpoints and check values of variables.

debugthis.m

```
for i = 1:5
   i = 3;
   disp(i)
end

for j = 2:4
   vec(j) = j
end
```



String Manipulation

Chapter 7

Linguaggio Programmazione Matlab-Simulink (2017/2018)

Strings: Terminology

- A ***string*** in MATLAB consists of any number of characters and is contained in single quotes
- strings are vectors in which every element is a single character
- A ***substring*** is a subset or part of a string
- ***Characters*** include letters of the alphabet, digits, punctuation marks, white space, and control characters
  - ***Control characters*** are characters that cannot be printed, but accomplish a task (such as a backspace or tab)
  - ***White space characters*** include the space, tab, newline, and carriage return
  - ***Leading blanks*** are blank spaces at the beginning of a string,
  - ***Trailing blanks*** are blank spaces at the end of a string
- Empty string is a string with length `0`, e.g. `''`

String Variables

- String variables can be created using
  - assignment statements
  - input function (with ‘s’ as the second argument)
- Since strings are vectors of characters, many built-in functions and operators that we’ve seen already work with strings as well as numbers – e.g., **length** to get the length of a string, or the transpose operator
- You can also index into a string variable to get individual characters or to get subsets of strings, or in other words, substrings

String Concatenation

- There are several ways to ***concatenate***, or join, strings
- To horizontally concatenate (creates one long string):
  - Using [ ], e.g.

    ```
    >> ['hello'   'there']
    ans =
    hellothere
    ```

  - Using **strcat**, e.g. strcat(‘hello’ , ‘there’)

    ```
    >> strcat('hello', 'there')
    ans =
    hellothere
    ```

  - There is a difference: if there are leading blanks, using [] will retain them whereas **strcat** will not

Vertical Concatenation

- Vertically concatenating strings creates a column vector of strings, which is basically a character matrix (a matrix in which every element is a single character)
- There are 2 ways to do this:
  - Using [ ] and separating with semicolons
  - Using **char**
- Since all rows in a matrix must have the same number of characters, shorter strings must be padded with blank spaces so that all strings are the same length ; the built-in function **char** will do that automatically

Character Matrices

- Both [ ] and char can be used to create a matrix in which every row has a string:

```
>> cmat = ['Hello';'Hi   '; 'Ciao '];
>> cmat = char('Hello', 'Hi', 'Ciao’);
```

- Both of these will create a matrix *cmat*:

H	e	l	l	o
H	i			
C	i	a	o	

- Shorter strings are padded with blanks, e.g.

```
cmat(2,:) is 'Hi   '
```

The **sprintf** function

- **sprintf** works just like **fprintf**, but instead of printing, it creates a string – so it can be used to customize the format of a string
- So, **sprintf** can be used to create customized strings to pass to other functions (e.g., **title**, **input**)

```
>> maxran = randi([1, 50]);
>> prompt = sprintf('Enter an integer from 1 to %d: ', maxran);
>> mynum = input(prompt);
Enter an integer from 1 to 46: 33
```

- Any time a string is required as an input, **sprintf** can create a customized string

String Comparisons

- **strcmp** compares two strings and returns logical 1 if they are identical or 0 if not (or not the same length)
- For strings, use this instead of the equality operator ==
- variations:
  - **strncmp** compares only the first n characters
  - **strcmpi** ignores case (upper or lower)
  - **strncmpi** compares n characters, ignoring case

Find and replace functions

- **strfind(string, substring)**: finds all occurrences of the substring within the string; returns a vector of the indices of the beginning of the strings, or an empty vector if the substring is not found
- **strrep(string, oldsubstring, newsubstring)**: finds all occurrences of the old substring within the string, and replaces with the new substring
  - the old and new substrings can be different lengths

The **strtok** function

- The **strtok** function takes a string and breaks it into two pieces and returns both strings
  - It looks for a *delimiter* (by default a blank space) and returns a *token* which is the beginning of the string up to the delimiter, and also the rest of the string, including the delimiter
  - A second argument can be passed for the delimiter
  - So – no characters are lost; all characters from the original string are returned in the two output strings
  - Since the function returns two strings, the call to **strtok** should be in an assignment statement with two variables on the left to store the two strings

Examples of **strtok**

```
>> mystring = 'Isle of Skye';
>> [first, rest] = strtok(mystring)
first =
Isle
rest =
 of Skye
>> length(rest)
ans =
     8
>> [f, r] = strtok(rest, 'y')
f =
 of Sk
r =
ye
```

The **eval** function

- The **eval** function evaluates a string as a function call or a statement
- Usually used when the contents of the string are not known ahead of time; e.g., the user enters part of it and then a customized string is created
- For example:

```
>> x = 1:5;
>> fn = input('Enter a function name: ', 's');
Enter a function name: cos
>> eval(strcat(fn, '(x)'))
ans =
    0.5403   -0.4161   -0.9900   -0.6536    0.2837
```

eval example

This is a very common application: a series of experiments has been run, resulting in files with the same name except for consecutive integers at the end of the name. We will write a **for** loop that will load files named ‘file1.dat’, ‘file2.dat’, … ‘file5.dat’ (assuming that they exist)

```
for i = 1:5
    eval(sprintf('load file%d.dat',i))
end
```

"is" & String/Number Functions

- "is" functions for strings:
  - **isletter** true if the input argument is a letter of the alphabet
  - **isspace** true if the input argument is a white space character
  - **ischar** true if the input argument is a string
  - **isstrprop** determines whether the characters in a string are in a category specified by second argument, e.g. 'alphanumeric'
- Converting from strings to numbers and vice versa:
  - **int2str** converts from an integer to a string storing the integer
  - **num2str** converts a real number to a string containing the number
  - **str2num** (and **str2double**) converts from a string containing number(s) to a number array
  - (Note: different from converting to/from ASCII equivalents)

Common Pitfalls

- Trying to use == to compare strings for equality, instead of the **strcmp** function (and its variations)
- Confusing **sprintf** and **fprintf**. The syntax is the same, but **sprintf** creates a string whereas **fprintf** prints
- Trying to create a vector of strings with varying lengths (the easiest way is to use **char** which will pad with extra blanks automatically)
- Forgetting that when using **strtok**, the second argument returned (the "rest" of the string) contains the delimiter.

Programming Style Guidelines

- Trim trailing blanks from strings that have been stored in matrices before using

- Make sure the correct string comparison function is used; for example, **strcmpi** if ignoring case is desired

Exercises

- Prompt the user for a string. Print the length of the string and also the first and last characters in the string. Make sure that this works regardless of what the user enters.
- In a loop, create and print strings with file names “file1.dat”, “file2.dat”, and so on for file numbers 1 through 5.
- Create an *x* vector. Prompt the user for ‘sin’, ‘cos’, or ‘tan’ and create a string with that function of *x* (e.g., ‘sin(x)’ or ‘cos(x)’). Use **eval** to create a *y* vector using the specified function.